Informatics

Verifying Automotive Software Components Using C Model Checkers

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Logic and Computation

eingereicht von

Timothée Durand
Matrikelnummer 11831520

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Ing. Georg Weissenbacher, D.Phil.
Mitwirkung: Privatdoz. Dipl.-Ing. Dr.techn. Wilfried Steiner

Timothée Durand [illegible]

Wien, 15. August 2020

Timothée Durand

[illegible]

Georg Weissenbacher

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Informatics

Verifying Automotive Software Components Using C Model Checkers

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Logic and Computation

by

Timothée Durand

Registration Number 11831520

to the Faculty of Informatics

at the TU Wien

Advisor: Associate Prof. Dipl.-Ing. Georg Weissenbacher, D.Phil.
Assistance: Privatdoz. Dipl.-Ing. Dr.techn. Wilfried Steiner

Vienna, 15th August, 2020

Timothée Durand

Timothée Durand

[illegible]

Georg Weissenbacher

Technische Universität Wien
A-1040 Wien • Karlsplatz 13 • Tel. +43-1-58801-0 • www.tuwien.at

Erklärung zur Verfassung der Arbeit

Timothée Durand

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 25. August 2020

Timothée Durand

Timothée Durand

Acknowledgements

I would like to express my deepest appreciation to Georg Weissenbacher, professor at the Technical University of Vienna, for his supervision of this master thesis, his great availability and his unwavering support.

I am extremely grateful to Wilfried Steiner, who offered me the formidable opportunity to perform this thesis at TTTech and work on an exciting project with real-world implications, and I also would like to thank him for his helpful review of this thesis.

The successful completion of this work would have been impossible without the help of Jakob Zwirchmayr, who always showed a great enthousiasm toward this project and gave precious advices. I also really appreciated working with Edin Arnautovic, Paraskevas Karachatzis, Diego-Alejandro Parra-Guzman and Fiona Sigmund, with whom it has been a pleasure to interact with on a daily basis, and who always shared useful tips and knowledge.

Special thanks to Prof. Jean-François Boulicaut, who provided valuable feedback on my work and kept me in touch with the National Institute of Applied Sciences (INSA) of Lyon.

I would like to salute all my friends from the Technical University of Vienna, who have made this Double Degree an amazing experience. I will never forget the endless hours spend together working on various projects. Kudos to Matthias König who helped with correcting the Kurzfassung.

Finally, I would like to address my most sincere recognition to my girlfriend and family, for their encouragement and patience for the duration of this project.

Kurzfassung

Softwaresysteme, die für die Automobilindustrie entwickelt werden, erleben derzeit einen enormen Anstieg der Komplexität, der mit einer Erhöhung der Sicherheitsanforderungen einhergeht. Die Sicherstellung ihrer Korrektheit ist eine herausfordernde und kostspielige Aufgabe, welcher die Forschung nach neuen technologischen Lösungen motiviert. Software-Verifikationswerkzeuge, die auf C-Code abzielen, sind zwar in der Industrie noch nicht weit verbreitet, zeigen aber jedes Jahr beeindruckende Leistungsverbesserungen, was sie zu guten Kandidaten für die Gewährleistung der Sicherheit eingebetteter Systeme macht.

Das Ziel dieser Studie ist es, die Fähigkeit modernster Verifikationswerkzeuge zu bewerten, welche die Abwesenheit von Laufzeitfehlern bei vier Softwarekomponenten unterschiedlicher Komplexität nachzuweisen. Diese Komponenten aus der realen Welt werden von TTTech entwickelt, einer Firma, die auf sicherheitsbezogene Lösungen für die Automobilindustrie spezialisiert ist.

Zunächst erstellen wir ein generisches Umgebungsmodell, das auf dem AUTOSAR-Standard basiert, der in der Automobilindustrie weit verbreitet ist. Dieses Modell zielt darauf ab, eine Komponente von der übrigen Software-Plattform für die Verifikation zu isolieren, und verwendet bereits existierende, durch den Standard definierte Spezifikationen. Anschließend prüfen wir den Code mit Ultimate Automizer, CPAChecker und CBMC, ergänzt durch Ideen wie z.B. *k*-Induktion oder Analyse des variablen Bereichs.

Unsere Ergebnisse zeigen, dass Verifikationswerkzeuge in der Lage sind, die Fehlerfreiheit von drei von vier Komponenten erfolgreich nachzuweisen, jedoch für die komplexeste Komponente keine definitive Antwort geben können. Die leistungsfähigste Verifikationsmethode ergibt sich aus der Kombination der Ergebnisse verschiedener Code-Analysen, wobei CBMC das endgültige Urteil mittels *k*-Induktion feststellt. Für den problematischen Fall geben wir Einblicke in die Ursachen der Schwierigkeiten und die nächsten Schritte, die zu deren Überwindung erforderlich sind. Wir kommen zu dem Schluss, dass die Einführung von Verifikationswerkzeugen in den Entwicklungsprozess positive Veränderungen der allgemeinen Codequalität mit sich bringen kann.

Abstract

Software systems developed for the automobile industry are currently witnessing a tremendous increase in complexity, happening in concert with an escalation of safety requirements. Ensuring their correctness is a challenging and costly task, which motivates the research for new technological solutions. While not yet broadly adopted in industry, software verification tools targeting C code demonstrate impressive performance improvements every year, which makes them good candidates for ensuring the safety of embedded systems.

This study aims at evaluating the capacity of state-of-the-art verification tools for proving the absence of run-time errors on four software components of various complexity. These real-world components are developed by TTTech, a firm specialized in safety-related automotive solutions.

Firstly, we establish a generic environment model based on the AUTOSAR standard, which is broadly adopted in the automobile industry. This model aims at isolating a component from the rest of the software platform for verification, and uses already-existing specifications defined by the standard. We then check the code using Ultimate Automizer, CPAChecker and CBMC extended with several ideas, such as k-induction or variable range analysis.

Our results show that verification tools are able to successfully prove the absence of errors in three out of four components, and cannot give a definite answer for the most complex one. The most capable verification method is obtained by combining the results of different code analyzes, with CBMC establishing the final verdict using k-induction. For the problematic case, we give insights into the causes of difficulties, and next steps required to overcome them. We conclude that introducing verification tools in the development process can bring positive changes to the general code quality.

Contents

CHAPTER 1

Introduction

The automobile software industry is currently witnessing an unprecedented growth. While electronic equipment has been a common sight in vehicles for multiple decades, there has recently been a plethora of new use cases for information technology equipements in cars, firstly for infotainment functionalities but most importantly, for performing driver assistance tasks of ever-increasing complexity. Indeed, highly automated driving solutions are expected to arrive on the market in the 2020s, and car manufacturers already propose Advanced Driver Assistance Systems (ADAS), performing tasks like automatic parking, collision avoidance, lane centering or pedestrian protection.

The general increase in complexity is happening simultaneously with an increase of safety requirements for the underlying cyber-physical systems. Engineers have to come up with new ways to certify the reliability of hardware and software components, operating systems and intra-vehicle communication protocols. In this light, the ISO 26262 functional safety standard for road vehicles was established to classify the risks linked to system malfunctions, and provide development guidelines to match the required safety guarantees. The integrity levels range from ASIL-A to ASIL-D. For the strictest integrity levels (ASIL-C and D), the norm *highly recommends* the use of formal verification tools in the development process to detect potential defects, or ideally to prove the system's correctness. However, the technical details concerning the integration of such tools in development process remains up to the interpretation of industry actors.

1.1 Problem statement

TTTech Auto is currently developing a software platform tailored toward these safety needs. Its aims is to facilitate the integration of heterogeneous automotive software components to run together on one or multiple hardware hosts, and to coordinate their execution in a safe manner. The platform guarantees isolation of components, provides deterministic execution schedules and a deterministic communication system, and can host services with safety integrity levels up to ASIL-D. As often in the automobile industry, the implementation comprise of hand-written as well as auto-generated C code, and there is a strong motivation to increase the software verification efforts targeting some critical components of the platform, and give stronger safety guarantees.

Verification of embedded C programs is a well know and studied research topic, and many (commercial and free) tools exist to handle this task. However, formal verification is far from being broadly adopted in the world of automotive software, as hard-to-tackle technical challenges emerge rapidly: especially, the state-space-explosion problem is often dissuasive, as verification tools run-time and memory usage increase exponentially with the complexity of the verification task. Thus, model checking is often limited to rather small programs and impractical in the case of a whole automobile software stack.

One of the most severe class of bugs that can be found in C code is *Run-Time errors.* They occur upon execution of C statements that are invalid with regards to the C standard, such as array accesses out of bounds or invalid pointer dereferences. Eleminating run-time errors is an absolute priority for ensuring software safety, that is why we will focus primarly on this task within this study.

The overall goal of this thesis will be to provide an answer to the following question:

> *Can modern verification tools targeting C code be used to prove the absence of bugs in safety critical components of the Automobile Industry? What are their limits, and how can we use them to their greatest potential?*

1.2 Approach

In the first part, components to verify and safety targets are identified through discussion with relevant interlocutors at TTTech. Besides, the architecture and the code is thoroughly explored to gain knowledge about the software.

In parallel, a *literature review* allows us to get familiar with different verification tools, their capacities and limitation. The objective is to identify which tools have been applied successfully on automobile software, or more generally on embedded systems with strong safety requirements. Furthermore, reviewing the techniques implemented in modern tools of the SV-COMP should allow to identify ways to tackle verification challenges encountered along the project.

Then, an iterative approach is adopted for getting started with verification and modelization tasks. Initially, we focus on simple verification tasks using only one tool (CBMC). The first verification trials are conducted on dummy software components, to acquire relevant know-how about interfacing the tool with industrial code. Then, a real-world component is inspected, which introduces proper modeling and interface abstraction challenges. More complex verification and modeling tasks are progressively introduced.

An automotive software component typically communicates with the rest of the platform using so called *interfaces.* These interfaces have to be abstracted. To do so, we must become familiar with the AUTOSAR norm which describe the specification format of these interfaces. Using AUTOSAR specification allows us to improve the precision of abstracted interfaces and conduct a better analysis.

For quantifying the success of verification approaches and comparing tools, some evaluation criteria need to be defined. Commonly used metrics include: the number of program properties successfully proved, disproved, or unproved (inconclusive analysis), the verification running time, the memory usage or the general code coverage.

After successful verification results have been demonstrated, work on the industrialization of the tool will begin. The objective is to demonstrate that the verification can be performed continuously on the platform, as opposed to only being one-time proof of concept. Eventually, the aim is to follow a continuous development process of the verification framework, by iteratively increasing the code coverage, giving stronger verification results, or offering new functionalities.

1.3 Outline

This thesis is structured as follows: in Chapter 2, we introduce some elements of context which are of high relevance to our work: we give an overview of norms that define the architecture of modern automobile software, and describe the architecture of the platform under investigation.

In Chapter 3, we present the different software verification tools that we have considered in our study. We also describe the common format defined by the Competition on Software Verification (SV-COMP) organizers that we used to interface tools with the targeted code.

In Chapter 4, we describe the environment model that was deployed: we list the hypotheses that we made about the verified software, as well as the methodology that we followed for isolating a component to verify from the rest of the platform. Finally, we describe the different pre-processing and code augmentation steps that are performed to transform the company code-base into a *verification-ready* form.

Chapter 5 contains the description of three software components on which we run our experiments, that are presented in Chapter 6 and characterize the success of our approach.

Finally, we discuss the results in Chapter 7, where we try to identify causes of successes and blocking points, summarize our finding and describe further works.

1.4 Related work

Software verification is a very important and active research topic, and many approaches have been developed and successfully applied to industrial problems. We will not aim for a detailed survey, but rather mention some of existing tools with the techniques that they implement, with their respective strength or weaknesses. Then we will quickly present how similar verification challenges have been dealt within the world of automotive software.

The first method to be mentioned is *static analysis with abstraction*. Static analysis aims at approximating the program behavior without executing it, to prove safety properties on that program. The analysis is generally a *sound over-approximation*, which means that no bugs will be missed. However, it might produce a high number of spurious bug reports (bugs that are reported but do not appear in the program). Besides, as safety violations are detected in the abstract domain, it cannot produce proper counter example of violated safety properties, and can only point to the program location where a bug may occur.

One well-known tool for abstract static analysis is the Astrée static analyzer [CCF$^+$05], which was successfully applied in aeronautics software checking [DS07]. Another successful static analyzer worth mentioning is Facebook's INFER [CD11]. It targets mainly memory safety, and is able to synthesise pre and post-conditions guaranteeing safety on isolated pieces of code. It is therefore well adapted to an incremental development process (see [CDD$^+$15]), as new additions into the code can efficiently be checked against previously known results regarding the rest of the codebase.

Another main subcategory of tools is model checkers, which aim at converting a program into a state-transition graph, and later verifying whether this model satisfies certain safety properties. Often, this satisfiability solving is implemented with an external *SAT* or *SMT solver*, and therefore benefits from their frequent performance improvement. The main advantage of these tools is that they provide detailed counter-examples when a safety violation is found. The developer can thus examine the execution trace leading to the bug reported.

The main drawback of model checkers is their sensibility to state space explosion: the number of reachable states to inspect can increase exponentially with the number of program variables, or the presence of (unbounded) loops, which greatly impacts the running time of these tools, or render the analysis unfeasible. Several techniques exists to mitigate this effect, notably bounded model checking. Here, the size of the inspected graph is bounded by some constant. The analysis is thus incomplete, but in some context (for example, real time embedded systems), this partial model is enough to prove safety in a program.

One model checker of particular interest for this project is CBMC [KT14], a bounded model checker for C programs. The main motivations for using CBMC is that the tool is freely available, mature and allows for fast prototyping.

The competition on software verification (SV-COMP) [Bey16] is an international competition of software verification tools. Every year, many competitors are evaluated on an significant amount of verification tasks on C and Java programs. This master thesis will benefit from two byproducts of the SV-COMP: First, all competitive tools are freely available and open source, which allow us to experiment with them. Secondly, SV-COMP defines a standard verification API that all competing tools must support in order to take place in the competition. Therefore, it defines a good basis to experiment with multiple state-of-the-art verification tools.

The research for automotive software verification contains several examples of application of such tools. However, the tools are often used to verify some very specific piece of software, such as small critical software component, OS cores, etc, and is not widely used. Therefore, automatic verification in this field seems to still be in its infancy.

Fang et al. [FKDO12] successfully used the SPIN model checker to test a multi-core real time OS based on AUTOSAR specifications. It should be noted that they do not check the OS directly but rather a model written in PROMELA, the modeling language used by SPIN. Moreover, they list the model creation as “one of the most difficult parts of the whole development process”, and cannot be automatized, therefore a similar approach cannot be considered on our side.

More recently, Berger et al. [BKÁ+18] applied the model checking tool *BTC embedded tester* to verify two R&D prototypes from Ford. The BTC tool relies heavily on CBMC that is used along with some extensions; CBMC and BTC developers collaborate closely [SKB+17]. They verify auto-generated C code from *Simulink* against functional requirements translated into temporal logic (LTL). They note that formalizing requirements from text into LTL is one of the most tedious part of their development. This work has since been extended in a master thesis [Wes19] where the same verification task is tested using SV-COMP competitors’ tools. The result is that CBMC and BTC are still superior to other tools for this particular use case. Another master thesis, completed in 2018 by Roland Mittag, focusses on the application of Static Analysis on AUTOSAR components, and give insights about commonly used testing methods for automobile software [Mit18].

CBMC has also been used for verifying correct API call sequence in automotive software by Kim and Choi [KC16]. Interestingly, they develop a method to use the tool to check general program properties (similar to those expressed in temporal logic). The tool does not allow for directly checking such properties, thus they proposed to check them by automatically insert assertions in relevant parts of the program. Another way of using bounded model checker to verify LTL properties has been described by Morse et al. in [MCNF15], using ESBMC extended with several ideas, like monitor threads, providing more formal guarantees.

CHAPTER 2

Project context

The aim of this part is to introduce all major concepts to help motivate and contextualize the work done for Software Verification at TTTech. In particular, we want to illustrate the challenges of this project and why verification is needed.

2.1 AUTOSAR Standard

In order to contextualize this work, we first need to introduce AUTOSAR and its most important concepts. The AUTomotive Open System ARchitecture is standardization organization created in 2002, and results from a collaboration between multiple major actors of the automobile industry (BMW, Ford, Toyota, or Volkswagen, to name a few). The main objective of AUTOSAR is to produce a series of standardized specifications for automobile software running on Electronic Control Units (ECUs), which control the car's logic and electronic subsystems. A modern car typically uses a high number of ECUs: AUTOSAR defines a common software architecture which prevent constructors from having to re-develop their software from scratch whenever the underlying hardware is changing. In this thesis, we will use the term "AUTOSAR " to designate both the AUTOSAR standard and the AUTOSAR organization.

These specifications describe the modular configuration of the software, and define interfaces between these modules. By complying to this standard, automobile constructors and equipment manufacturers experience a simplified development process, collaborate more easily with other AUTOSAR partners and benefit from an improved re-usability of developed software.

The motto of AUTOSAR is "Cooperate on standards, compete on implementation." [Bun11]: Indeed, the AUTOSAR consortium itself does not deliver any code. Software vendors and Original Equipement Manufacturers (OEMs) each propose their own implementation of the standard, with various degree of performance and different targets.

An overview of the AUTOSAR classic architecture can be seen Figure 2.1. The most distinctive aspect of this architecture is its layered structure: each layer introduces a new level of abstraction and interfaces. The principal layers are:

- The **Basic Software (BSW)** is a set of standardized software modules, managing the base functionalities of the AUTOSAR platform. Its role is to abstract the hardware implementation and provide standard services for the higher layers, for example regarding memory management or diagnostic functionalities.
- The **Application Layer** is composed of several **Application Software Components (SW-Cs)**, which carry out the vehicle applications (e.g. door locking systems, cruise control systems, etc). SW-Cs interact only with the RTE, and as such, they are meant to be completely platform-independent and reusable on any AUTOSAR -compliant ECU architecture.
- The **Runtime Environment (RTE)** [R4.15c] is a standardized API which makes the junction between the platform-specific Basic Software and the portable Application Layer. The RTE defines interfaces that allows different Software Components to communicate with one another, and give them access to Basic Software services.

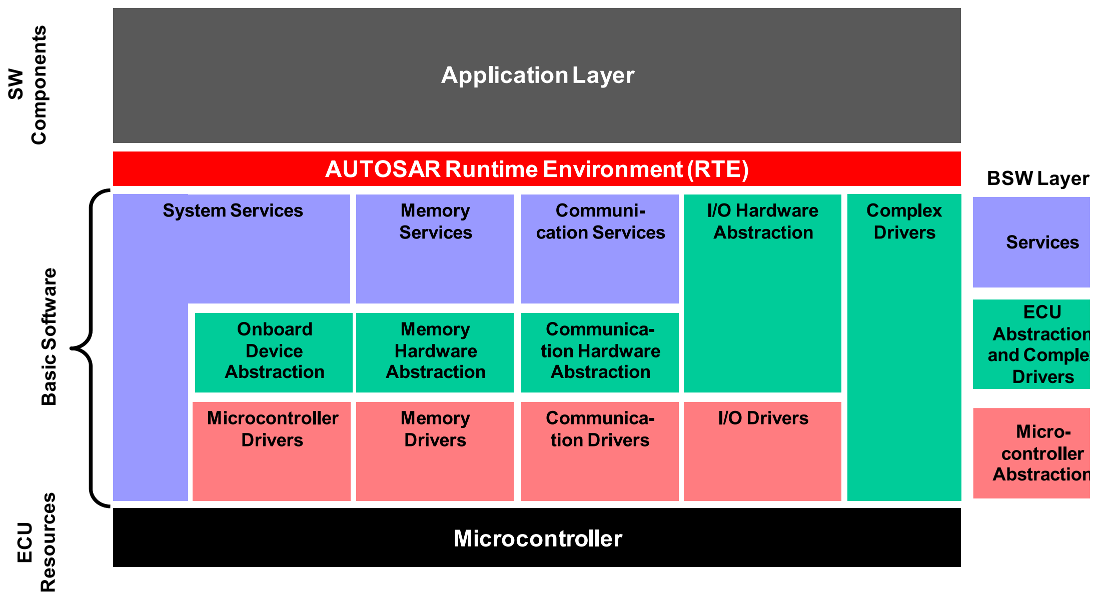


Figure 2.1: AUTOSAR classic platform architecture [Bun11] [R4.15a]

AUTOSAR supports the development of safety related systems [R4.15b], by specifying several mechanism regarding safe memory access, execution, timing or information exchange. However, AUTOSAR is not a complete safe solution by itself, as it is still possible to develop unsafe systems that use AUTOSAR safety mechanisms. The norm of reference regarding functional safety of automobile software is the ISO 26262.

2.2 ISO 26262 and future norms

ISO 26262 [ISO18b] is a norm introduced in 2011 that is concerned with the safety of electronic and electrical systems within road vehicles. It gives guidelines that are to be adopted all along the development process of a system in order to ensure its functional safety. These include requirements to be followed during the requirements specification, design, implementation, integration, verification, validation and configuration phases of system development.

The norm defines four different levels of safety, designated Automotive Safety Integrity Levels (ASILs), ranging from ASIL-A (the least stringent) to ASIL-D (the most stringent). The ASIL levels is used to classify risks and depends on three criterions: *Severity*, *Exposure* (i.e. Probability of failure) and *Controllability*.

In the frame of AUTOSAR , it is typical to give an ASIL rating to a given SW-C (see [R4.15b]). A component rated ASIL-D would indicate that its failure would cause a direct risk of life-threatening injuries.

In terms of software safety, the norm gives clear guidelines toward which verification and/or testing techniques to put in place during development [ISO18a]. Recommendations for each ASIL are contained in Table 2.2. The practices are either:

- "++" highly recommended,
- "+" recommended, or
- "o" the norm does not give recommendations for or against its usage.

Method		ASIL			
		A	B	C	D
1d	Semi-formal verification	+	+	++	++
1e	Formal verification	o	o	+	+
1f	Control flow analysis	+	+	++	++
1g	Data flow analysis	+	+	++	++
1h	Static code analysis	++	++	++	++
1i	Static analyses based on abstract interpretation	+	+	+	+

Figure 2.2: ISO 26262 - methods for software unit verification - excerpt from [ISO18a]

Rows 1d to 1i clearly indicates that the usage of different software verification tools based on formal methods are recommended or highly recommended during the validation phase. For this reason, TTTech and other actors of the industry have strong incentives to introduce such tools in their processes.

In addition to the ISO 26262, a new norm, the ISO/SAE 21434 - *road vehicle: Cybersecurity Engineering* - is currently in the works and is expected to be published in late 2020 [BMGS20]. As automobiles are becoming increasingly connected with their environment, a serious increase in potential cyber-attacks vectors is observed. Like the previously mentioned norm, the ISO/SAE 21434 proposes a method based on risk evaluation and management, and gives guidelines to be observed during the development of systems (an overview of the content can be found in [SGM18]).

While the definitive content of the document is not yet published, one can imagine that the norm would hardly be complete without considering cyber-security risks associated with software logic bugs. One classic example of such exploits are stack and buffer overflows, that can allow an attacker to take full control of the targeted machine. Cai et. al. have demonstrated last year the possible execution of arbitrary code on a BMW ECU using a stack overflow vulnerability in the Navigation Update Service [CWZ+19]. Tools based on formal methods could surely have detected this attack point, and have already proved their effectiveness for eliminating similar attack vectors [FLR17] [CGD+16].

2.3 System under investigation

Our target is an automotive software platform focused on safety that is being developed by TTTech. The key technology of this product its safe scheduling solution: it allows different software tasks to be coordinated in a time-triggered, deterministic fashion, guaranteeing that critical tasks are *always executed in time*. This technology is based on several innovations in the domain of real time software with Time-Triggered architecture that were developed by TU Wien and TTTech in the past thirty years [KB03]. It also implements various safety features such as error management, task supervision and has fail-operational capabilities. These ensure that various software components are encapsulated and guarantee *freedom from interference*, so that a failing component would not cause the failure of the whole system.

In terms of architecture, the platform is designed to be generic, and can be deployed on various operating systems or hardware. In particular, its architecture allows it to be deployed on multiple ECUs, namely (one or more) **Performance Hosts** and a **Safety Host**. The Performance Host(s) typically operate on a high performance CPU with hardware accelerators (GPUs, TPUs), which do not comply with highest ASIL ratings. The Safety Host is subject to higher ASIL requirements (typically ASIL-D), is in charge of safety critical and system supervion tasks running on ASIL-D compliant hardware.

The platform provides an implementation of the most central element of the AUTOSAR standard: it comes with its own inter-host communication system, allowing components running on different systems to interact seamlessly through the AUTOSAR RTE. Other services of the AUTOSAR BSW, are also implemented by TTTech, while some lower levels of the BSW are implemented by the underlying OS.

2.3.1 From System Definition to generated C code

The developement of a new AUTOSAR compliant system is a well defined process referred-to as *AUTOSAR Methodology*. It is a thorough design and architecture task, during which detailed description of 1. software component (data types, interfaces), 2. ECU resources (harware, peripheral requirements) and 3. various system constraints (e.g. network topologies) are defined. The result of this work is the *System Definition*, which in practice is a set of AUTOSAR XML (ARXML) files containing information needed to deploy the system on the platform and defining the software architecture.[1]

One of the roles of a configurable automotive software solution is to process ARXML files of the *System Definition*, and generate executable files (or source code ready to be compiled) satisfying the specifications, that will serve as a basis for the application-oriented software developed by OEMs. As we have denoted earlier, one of the crucial aspect ensuring the safety of the system is the determinism of the execution: this hints that most elements relative to the system configuration will end up being *encoded statically in the code*. For example, the execution schedule, or the memory areas that each component uses are predetermined and realized with C constants. In this view, an important part of the knowledge and complexity is located in the code generators. As proving their correctness is challenging, the verification and validation effort is rather conducted on the generated code, and must therefore be automated.

Porting of the software to a new platform involves many different verification and validation steps, at different modeling and integration stages[2]. The system also provides a self-test procedure [McC85], under which it will test its own functionalities against expected values.

However, such test methods are only as good as the number and the extent of the tests cases that are operated. Verification techniques based on formal methods have the potential to provide stronger correctness proof and safety guarantees, by exploring a wider state space.

2.3.2 Continuous Verification

Finally, one of the emphases that was put on this study is to adopt a verification approach suited to continuous integration and fast development cycles.

While the development of an ECU specific software (i.e. targeting one particular car model) is done with long term *Software Development Lifecycle Models*, such as the V-Model [Rup10], automotive companies still highly benefit from modern software development practices, such as Continuous Integration. Continuous Integration allows developers to detect bugs or design flaws early in development, and thus reduce development costs and testing efforts down the line.

[1]This is a simplified view, as the design of the platform and of subsequent software specification has several intermediate steps. Am interesting overview of the AUTOSAR design process and the bridge between system architecture and software architecture can be found in [MAK14].

[2]This is in fact one major advantage of this platform: being able to perform precise tests early in the developement process (with so-called Software-In-the-Loop methods), thus avoiding the discovery of flaws late in the development process, which generally induce high cost to correct them.

One of the key ingredients of a successful Continuous Integration is the automatic testing system, which is in charge of performing daily tests on the newly introduced code. The code is typically tested against known results (regression tests), and different code quality standards, such as MISRA-C, a set of C development guidelines broadly adopted in the Automobile Industry.

Static Analyzers also commonly appear in Continuous Integration pipelines [NŠST18] [ZSO+17]. Together with code linters, these tools also contribute to improving the code quality by signaling bugs or potential bugs, as well as lines of code with ambiguous semantics. They can also be used to forbid certain code constructs that, while correct, might be dangerous or have compiler-dependent behaviors.

While these various code analyzers have their merits, they generally do not provide formal guarantees about the tested code, in contrast with tools that we are targeting in this thesis.

As we want to investigate formal verification tools usability within the context of continuous integration, we have to consider the following constraints:

- **Low developer interaction:** as the software verification is thought to happen daily or weekly, it must not become a burden for developers and lead to an increase of development costs. Therefore, it must be as automated as possible, that is why we will focus on approaches that *do not require developers to write new specifications in the code.*
- **Reliable results:** Developers will accept the introduction of a new tool in their development environment only if they are convinced that it is adding value to their work. For this reason, the tool must provide reliable results. Every reported error will require investigation from a developer: their time should not be wasted, so we should ensure that all reported errors are genuine.
- **Reasonable running time:** Finally, as verification should be conducted daily or weekly, it must provide its results within a reasonable number of hours.

CHAPTER 3

Software Verification tools

3.1 Basic concepts

We want to introduce here some elementary concepts related to software verification that will be referred to in the following. The three following definitions are derived from Jhala and Majumdar "Software Model Checking" [JM09].

- **Definition: Software verification problem**: Given a program P and an error location ε, a program is said to be "safe" if ε is unreachable and "unsafe" if ε is reachable.
- **Definition: Sound Analysis:** An algorithm for the software verification problem is *sound* if, for every program P and every error location ε, if the program returns "safe", then ε is unreachable.
  A sound verifier will always find and report all errors in a given program, but might report additional, *spurious errors*. Therefore, manual inspection is required for filtering the results.
- **Definition: Complete Analysis:** An algorithm for the software verification problem is *complete* if, for every program P and every error location ε, if the algorithm return "unsafe", then the error location ε is reachable.
  A complete analyzer will only report genuine errors, but it might not detect all errors present in the program under analysis[1].

Naturally, the ultimate goal of every software verification tool is to accomplish soundness and completeness, which would require the inspection of exactly all possible program

[1]The concept of a *complete* formal system in logic slightly differs from the common interpretation of "completeness". A proof system said to be is complete if all *true* properties can be proven to be *true* using this system, but a complete logical proof system is not necessarily able to identify *false* properties.

states. While this might be possible for some small category of programs, it is in general *undecidable*. In practice, verifiers have to cope with the *State Space Explosion Problem* [CKNZ11]: the number of program states increases exponentially with the number of program variables, inspecting all of them in reasonable time is largely unfeasible.

In order to circumvent the state explosion problem, analyses resort to approximating the set of reachable program states or execution traces that can appear in the program. Two cases emerge:

- **Over-approximations** consider a superset of program states and traces when running the analysis. This can be achieved by the mean of abstract representations, such as using superposition of states instead of real program states. Over-approximations generally produce *Sound analyses*.
- **Under-approximations** only analyze a subset of possible program states, which is commonly done by introducing bounds in the program. The bounds can artificially limit the number of iterations of loops, limit the size of manipulated memory objects (e.g. maximum array sizes, maximum lists length), or limit the total number of instructions. Under-approximations typically produce *Complete Analyses*.

Finally, software analysis methods often make use of **non-deterministic values**, which could also be designated as *unspecified values*. When a variable is modeled with non-determinitism, it means that *any value* that the variable could take is considered. The program may consequently follow *any computation path* resulting from the different possible values of the said variable. Non-determinism is useful to over-approximate the behavior of a function, as it helps modeling all possible results that this function could return.

3.1.1 Verification tools selection criteria

The offer for software verification tools is broad and diverse. Given the scope of our project, we consider tools matching the following criteria:

- *Targeting standard, plain C code:* We consider only software verification tool which take plain C code as an input. In particular, we refrain from using tools that require additional handwritten specifications to be added in the code.
- *Freely available:* The software verification tool has to be freely available, which favors academic tools.
- *Active:* We only consider tools that are still under active development.
- *Simple of usage:* Tools that are unable to analyse the code *as-is* will not be considered, as we cannot maintain different versions of the code for compliance against each and every tool.

- *Covering all constructs of the C language* We will aim for tools that can model the C language completely. Some tools cannot prove properties depending on floating points, bit vectors, or memory allocations and will therefore not be considered.

Because of the prior mentioned reasons, a natural fit for our requirements is the tools taking part in the SV-COMP [Bey20a]. The SV-COMP is a competition that is held every year at the *Tools and Algorithms for the Construction and Analysis of Systems* (TACAS) [BP20] conference, and that aims at evaluating the state of the art verification tools on an important database of C programs, for which various properties must be proven. Except from the *bug coverage* criterion, the tools participating in the competition match all the above-mentioned rules, it was thus highly interesting for us to evaluate the most promising tools of the SV-COMP on our case study. Apart from the results, the SV-COMP defines an Application Programming Interface (API) for program verification tools that standardizes the tools behavior; we describe it in Section 3.3.

3.2 Selected verification tools

3.2.1 CBMC

C Bounded Model Checker (CBMC) has been selected very early on in the project as it is a very mature tool for model checking of C program that has already been the object of numerous industrial applications[2]. Initially presented in 2004 [CKL04], CBMC has acquired some maturity and multiple performance improvements over time, even if the core principle of its analysis remains unchanged.

CBMC implements *bounded model checking* [BCCZ99], an under approximation technique under which all program loops are unwound up to k times. For instance, `while` loops in the program are replaced by k duplication of the loop body, nested in k `if` statements using the loop condition.

The program obtained is subsequently translated into a boolean formula, using an encoding of C statements into equivalent boolean logic expressions. The encoding is built in such a way that the program is correct if and only if the formula is *unsatisfiable*, that is, if no assignment of boolean variables making the formula true exist. If such an assignment can be found, it indicates the existence of a property violation in the program, and can later be used to build a real execution trace leading to the error location.

The formula is then given to a SAT solver such as MiniSAT [SE05] which is in charge of giving the final diagnostic. In case of violation, a counter-example is built from the satisfiability witness, and provides the erroneous execution trace to the user.

CBMC can therefore prove the correctness of a program for which all loops are bounded, and is efficient for finding shallow bugs. Moreover, CBMC comes with an entire framework

[2]See the list of applications of CBMC available at `http://www.cprover.org/cbmc/applications/` (accessed on 26th July, 2020)

(`goto-cc`, `goto-instrument`), which is able to compile a program with standard C compiler workflows, and perform useful code transformation on the code. This framework is very practical and quickly became indispensable for performing our experiments. We have been using mainly the latest release of CBMC (5.12), on which we performed several bug fixes and adapted some functionalities.

3.2.2 CBMC augmented with k-induction

While bounded model checking is efficient, it does not constitute a complete proof system for programs containing unbounded loops. To alleviate this problem, bounded model checker can be augmented with k-induction [DMRS03] which provide them with a way to achieve *soundness*. k-induction is an extension of mathematical induction, in which we are able to prove the validity of a property for all n, by proving that it holds for a *base case* (typically, for $n = 0$), and proving that, if it holds for an arbitrary n, it then necessarily holds for the next value $(n + 1)$, the *inductive case*.

The same principle can be applied to unbounded loops in programs, using the same upper limit k as bounded model checkers. The proof is also performed in two steps:

- **Base case**: Check that the program is correct for the first k iterations of the loop (this is analoguous to plain *bounded model checking*).
- **k-Inductive case**: Enter the loop at an arbitrary loop iteration i (this can be done by assigning *non-deterministic values* to all loop variables). Suppose that the program is correct for the next k iterations; check that it is then correct for the $k + i + 1$-th iteration. If this holds, the program is proven correct for all loop iterations.

We found that this principle was easily applicable in our case study by doing a few code transformations, that are described more in details in Section 4.3.3[3] Since its introduction, k-induction has been extended further than these basic principles, for instance by using synthesized loop invariants to strengthen the assumptions [RICB17] [BDW15]. However, implementing them ourselves was out of the scope of this thesis, hence we have only been applying tools based on these techniques, such as CPAChecker .

3.2.3 CPAChecker

CPAChecker [BK11], (Configurable Program Analysis - Checker) would be better defined as a verification framework than a verification tool. It was presented in 2009 as a platform containing one basic algorithm that can easily be extended to implement new verification techniques and ideas. The central data-structure manipulated by the algorithm is a

[3]Credits should also be given to Lucas Westhofen, that has demonstrated in his thesis the viability of the method in a comparable case study [Wes19]. Nevertheless, our application of k-induction differs in terms of implementation, and in terms of targeted properties.

control flow automaton, a graph where nodes represent program location and edges represent operations. The CPA algorithm [BHT07] acts on this data-structure with the use of three different operations: the *transfer*, *merge* and *stop* routines, and uses an abstract domain D as a representation for abstract program states.

Using these four elements, the CPA algorithm can compute a set of reachable abstract states from the initial program state, and the final set of reachable states is used to give conclusion about program properties. For example, if the final set of reachable states contains a state where a variable $i \in [1, 10]$ is used for dereferencing an array of size 5, an *array out of bound error* can be reported.

Without diving into further implementation details, this means that any verification technique can be implemented in this tool, simply by providing an implementation for the three routines *transfer*, *merge* and *stop* mentioned above. Furthermore, multiple analyses can be performed at the same time and combine their results dynamically, which can dramatically strenghten conclusions. Nowadays CPAChecker contains implementations of almost all state-of-the art verification techniques, such as *k-induction* [BDW15], *Counter-Example Guided Abstraction Refinement (CEGAR)* [Löw13], *Predicate Abstraction* [LW12], or *Symbolic Execution* [BL18].

Using all these techniques sequentially, CPAChecker systematically reached the top of the leaderboard in the lastest editions of the SV-COMP. However, it is thought primarily as a platform for experimenting and comparing new software verification ideas, and does not pretend to be a robust tool ready for industrial applications.

In our experiments, we have been using CPAChecker release 1.9.1 with the default SV-COMP 2020 configuration, except for memory-leak detection features that we have disabled.

3.2.4 Ultimate Automizer

Ultimate Automizer [HCD$^+$13] has been chosen as a third contender as it has achieved the second place in the last SV-COMP after CPAChecker, while also competing in all categories, and has generally achieved good results in competitions over the last years. It is developed after a new approach to model checking based on automata that was introduced together with the tool in 2013.

At its core, Ultimate Automizer performs a CEGAR algorithm, which functions in the following fashion: suspected erroneous execution paths are identified an examined. If the erroneous path can actually occur in the program, it is returned and constitutes a counter-example to the program validity. Otherwise, a sequence of predicates proving the path's infeasibility in the original program is computed, and used to improve the precision of the current model. The algorithm continues until either a counterexample is found, or no more suspected error traces can be found [HCD$^+$18].

The efficiency of CEGAR depends heavily on the selection of inspected execution traces, and how much can be learned from spurious counter-examples. Ultimate Automizer uses many diverse and different techniques to find these traces and benefits from frequent improvements. In our experiments, we have been using Ultimate Automizer release 0.1.25, that has been released for the SV-COMP 2020.

3.3 SV-COMP API

The SV-COMP defines a standard format for verification tasks [Bey20b], that will simplify the interfacing of our case study with the verification tools. Firstly, it defines a list of functions with verification-relevant semantics:

- **`__VERIFIER_error()`**: Indicates a forbidden location that a correct program should not reach. A standard C assertion – `assert( condition )` – can be subsequently defined as:
  `if(!condition) { __VERIFIER_error(); }`.
- **`__VERIFIER_assume( condition )`**: Indicates that a condition necessarily hold on this location. Its semantics are equivalent to the following implementation: `if(!condition) { while(1); }`, which essentially prevents the program to go any further if the condition is not matched.
- **`__VERIFIER_nondet_`*`type`*`()`**: Returns a non-deterministic value of a given *`type`*.
- **`__VERIFIER_atomic_begin()` (or `_end()`)**: Designates the beginning or the end of an atomic section in a multi-threaded environment.

In addition to these functions, each C program that is part of the SV-COMP benchmark is shipped with a *specification file*, containing one or more Linear Temporal Logic (LTL) formulas stipulating additional checks to be done in the program. These formulas utilise the following elements of the LTL syntax: **G** `condition` indicates that `condition` should *always* be true, and **F** `condition` indicates that the program *must* eventually reach a state where `condition` is true. The list of possible specifications are available in Table 3.1

The use of this API in our particular use case will be described in Chapter 4: Environment Model and Code Preparation.

3.4 Other code analyzers

3.4.1 Frama-C

Additionally to the previously mentioned tools, we have been using Frama-C [CKK$^+$12] to perform several analyses on the code that we would characterize as pre-processing steps. Frama-C is a software analysis platform with different plugins performing various

LTL formula	Semantics	Relevant
`G valid-free`	All memory deallocation are valid	No
`G valid-deref`	All dereferenced pointers refer to valid memory areas	**Yes**
`G valid-memtrack`	All allocated memory is tracked	No
`G valid-memcleanup`	Every allocated memory is eventually deallocated	No
`G !overflow`	The result of an integer operation is outside of the range of values that can be represented by the resulting type	**Yes**
`F end`	The end of the program is eventually reached.	No

Table 3.1: List of possible LTL formulas listed in the SV-COMP standard specification file. The last column indicate the relevance of the specification for our case study (more details are given in Section 4.1).

tasks, such as analyses based on abstract interpretation, deductive verification, concolic test generations or code slicing. It is designed for being applied on large scale industrial projects and constitutes a formidable toolbox for software analysis.

For this project, we have mostly been using the Value Analysis plugin which is able to compute an over-approximation of variable values for all program locations, and also report alarms when potential bugs in the code are detected. Frama-C , however, uses its own specification language, ACSL [BFM+08], and is not natively compatible with the SV-COMP API described in Section 3.3. We have created a restricted implementation of the SV-COMP API specific to Frama-C , to ensure that it would give us results coherent with our verification tasks. This implementation was however very limited, and we have not been using Frama-C for computing proofs. Our usage of Frama-C will be described in Section 4.3.2.

CHAPTER 4

Environment Model and Code Preparation

One of the most important tasks of this work in terms of workload was to construct a proper environment model and code transformation that make the verification feasible, correct, and suited to analyses by verifiers.

The environment model has two objectives:

- **Limit the scope of the verification**: with unlimited time and memory resources, we could directly feed the whole source code of the program to the software verification tool. Since this is infeasible in practice, our objective is to only verify a limited part of the code at once, while providing an acceptable simulation of the abstracted code's behavior.

- **Provide a sound but tight over-approximation**: while we abstract away part of the code, we still want to ensure that we detect *all possible bugs* that could appear in the code that we are actually checking. We can guarantee this only if we exhaustively explore all reachable states that could occur during the program execution (potentially exploring states that cannot actually occur in the program).

The following part is organized as follows: we first introduce our environment hypotheses, we then present our code abstraction methodology, and finally we describe the steps for generating a test harness - the final input for software verifiers.

4.1 Environment hypotheses

Verification tasks that we have considered during our case studies all follow the basic SW-C structure. Essentially, our test target is entirely represented by two *Runnable* functions:

- The `Runnable_Init()` function is called at the beginning at the execution. Its role is to initialize all global variables to bring the SW-C in a valid initial state. The system gives us a *guarantee* that the `Runnable_Init` function will be called before anything else happens.

- The `Runnable_Step()` function is called at regular intervals (typically, every 10 milliseconds), until the end of the execution. Its goal is to maintain the state of the system and to perform application-related tasks. Additionally, we have the guarantee that all loops embedded in the `Step()` function are bounded: this derives from the fact that all runnables of the system exhibit a **bounded Worst Case Execution Time (WCET)**.

The execution of SW-C can be simulated with the code template Figure 4.1.

```c
// Global variables declaration and initialization

int main()
{
  Runnable_Init(); // Further initialize globals

  while(1)
  {
    Runnable_Step();

    wait(delay);
  }
}

```

Figure 4.1: Software Components basic code structure

An additional property respected by the programs under investigation, is the *absence of dynamically allocated memory* in the code. This is indeed one of the restrictions imposed by the MISRA-C standard, that is to be respected in the whole most automotive codebases.

4.1.1 Model limitations

Our model of the SW-C execution carries several intrinsic hypotheses:

- **Single-threaded program / no-interference hypothesis**: we suppose that global variables cannot be arbitrarily modified between two executions of the `Step()` function.
- **Complete initialization hypothesis**: the only initial state of the system that we consider is the state obtained after the call to `Runnable_Init()`. However, some global variables, that could be labelled "configuration variables", may have other possible values. Consider a global variable `const int DEBUG=0`, that could be used to statically enable or disable some part of the code for debugging. In this study, we decided to not change initial values of global variables, as determining valid initial ranges for all variables was not feasible automatically, and changing them to arbitrary ranges would simply introduce too many errors.

Finally, our approach tests the code as it is given to the compiler. This means that we only test each function *as it is currently used* in the software; and not all their possible use cases. For example, one could imagine that a function misbehaves for some particular input that cannot appear in the current version of the software, but that may occur in a future version. Despite this, we can guarantee, in case of successful verification, that the software will not crash in its current configuration.

4.2 Interface Abstraction

There are two kind of interfaces to non-targeted code that we had to abstract: the RTE API and the Portable Operating System Interface (POSIX) API. Components which strictly adhere to the AUTOSAR SW-C definition can only access the RTE API. However, some other components (*Complex Device Driver*) have additional accesses to some BSW and Operating System (OS)-specific interfaces.

In our case, it was not necessary to stub BSW interfaces, but we encountered some POSIX API calls for which abstraction was necessary.

Concretely, abstracting an API is done by implementing non-deterministic stubs for all functions of the API that are used in our target code.

A *non-deterministic stub* is a function definition that simulates the behavior of a real function. The stub gives non-deterministic values to all *output value* of the function, and try to respect valid value ranges if they are known. An example of such a stub is presented Figure 4.2. In this example, `wheel_id` is not modified as it is an *input value*, and by the semantics of the C language, altering its value would not change anything from the perspective of the caller.

As a general rule for stubs, *output values* are defined as:

- return value of functions, and,
- all parameters given as non constant pointers (pointers without the keyword `const`)

```c
#define OK   0
#define NOK  1

/*** read_wheel_torque specifications *********************************/
/* \param[in]  wheel_id    ID of the wheel.                            */
/* \param[out] torque_val  torque sensor value. Range: [-50.0f,50.0f] */
/* \return                 Read status.         Range: {OK, NOK}      */
/**********************************************************************/
int read_wheel_torque(int wheel_id, float *torque_val);

// non-deterministic stub
int read_wheel_torque(int wheel_id, float *torque_val)
{
  *torque_val = __VERIFIER_nondet_float();
  __VERIFIER_assume(*torque_val > -50.0f && *torque_val < 50.0f);

  int ret_val = __VERIFIER_nondet_int();
  __VERIFIER_assume(ret_val == OK || ret_val == NOK);
  return ret_val;
}

```

Figure 4.2: Example of a function declaration and the corresponding *non-deterministic stub*

4.2.1 AUTOSAR RTE Abstraction

Exploiting AUTOSAR specifications

In the standard AUTOSAR workflow, the RTE implementation is automatically generated from specifications given in the form of ARXMLs files. In the particular case of software components, a SW-C-specific specification file enumerates all RTE interfaces that this component can access: in AUTOSAR jargon, this is called the list of *Port Prototypes*. These prototypes refer to *Port Interfaces*, which define the `DataTypes` of objects being exchanged, and are located in another ARXML file (the system-wide "system definition" file). The complete SW-C generation methodology can be seen Figure 4.3.

For creating our stubs, information relative to the communication paradigms of each port of the RTE was not crucial. Rather, we only needed to retrieve the signature of API function, alongs with *DataTypes specifications*, that are required to increase the precision of our stubs.

Each AUTOSAR `DataType` has specific characteristics that can be extracted from ARXML files. Rather than giving a complete list of possible `DataTypes`-related information found in ARXMLs, we want to quickly mention here the most relevant parts:

- **Composition**: Each `DataType` can be a composition of one multiple subtypes (the type definition in C will be resp. `typedef` or `struct`). The reference to the underlying type(s) is given in the `<IMPLEMENTATION-DATA-TYPE-REF>` node(s) of the ARXML. The type composition is recursive, and ends when a *base type* (e.g. `uint8`, `uint32`...) is reached.

- **Constraints**: For every `DataType`, maximal and/or minimal values can be defined. These values can appear either in the `<DATA-CONSTRAINT>` node, or as can be defined as part of a computation method (`COMPU-METHOD`) of type `SCALE-LINEAR` or `LINEAR`.
- **Possible Values**: Instead of a valid range, a discrete set of possible values of a type can be defined. These are given by the `TEXTTABLE` category of computation methods. The values appearing in text tables will be represented as `#define`'d constants in the C implementation.
- **Size**: Finally, the size of the `DataType`'s representation in memory is specified for base types and for array types.

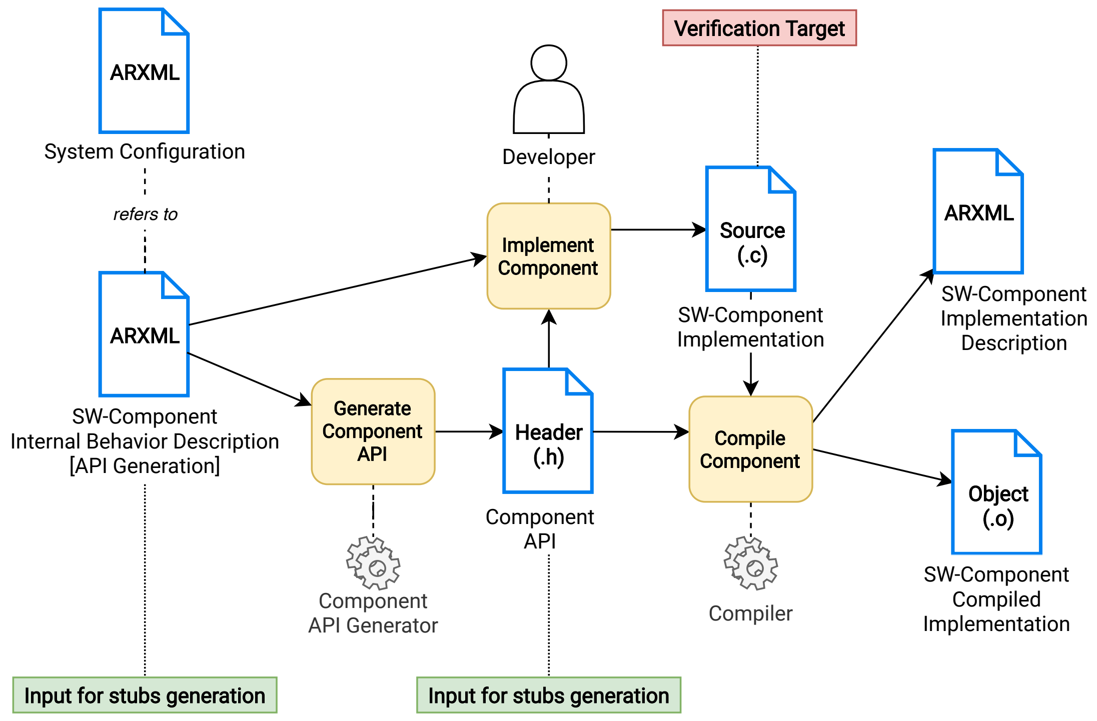


Figure 4.3: Software Component generation process (derived from [R4.15c] section 3.1)

Stubs generation

We used all aforementioned pieces of information to generate, for each AUTOSAR `DataType`, a *non-deterministic modifier*, and a *non-deterministic generator*. Using those, we can automatically generate non-deterministic stubs for all RTE API functions.

```c
// ...
// a non-deterministic modifier for the AUTOSAR type `Dt_GPS_Position`
void modif_nondet__Dt_GPS_Position(Dt_GPS_Position *gps_position)
{
  modif_nondet__Dt_lat(&gps_position->latitude);
  modif_nondet__Dt_lon(&gps_position->longitude);
  modif_nondet__Dt_alt(&gps_position->altitude);
}

// a non-deterministic modifier for the AUTOSAR type `Std_ReturnType`
Std_ReturnType modif_nondet__Std_ReturnType(Std_ReturnType *tmp)
{
  *tmp = __VERIFIER_nondet_uint8();
  __VERIFIER_assume( *tmp == E_OK || *tmp == E_NOT_OK );
  return ret_val;
}

// a non-deterministic generator for the AUTOSAR type `Std_ReturnType`
Std_ReturnType generate_nondet__Std_ReturnType(void)
{
  Std_ReturnType ret_val;
  modif_nondet__Std_ReturnType(ret_val);
  return ret_val;
}

// a non-deterministic rte api stub
Std_ReturnType Rte_Read__Dt_GPS_Position(Dt_GPS_Position *gps_position)
{
  modif_nondet__Dt_GPS_Position(gps_position);
  return generate_nondet__Std_ReturnType();
}
```

Figure 4.4: Example of a generated non-deterministic RTE API stub

An example of generated stub is given Figure 4.4. The stub provide a non-deterministic implementation of the RTE function `Rte_Read__Dt_GPS_Position`[1]: in this case, both the input parameter and the return value are non-deterministically modified.

The RTE API Abstraction was automatized using a number of Python scripts taking as input one C header file (.h) per software component and ARXMLs. The different steps involved in the process and used tools are shown in Table 4.1.

4.2.2 POSIX API Abstraction

The need to abstract some interfaces of the POSIX API revealed itself during the testing phase. During the first verification experiments on a *Complex Device Driver*, CBMC reported a high number of function with missing definitions. In this case, the tool replaces

[1] function and type names have been simplified and do not respect AUTOSAR naming conventions.

Processing step	Tooling
Retrieving API signatures from C headers	PyCParser [Ben12]
Parsing ARXML specification files	Custom parser based on LXML [BFB+05]
Creating the Stubs (generating C code)	Python + Jinja2 [Ron]

Table 4.1: Tools used for the RTE API stubs generation

missing definitions by non-deterministic stubs, but it became quickly evident that those were too imprecise, as CBMC reported a high number of spurious bug that could be traced back directly to these stubs.

Writing stubs for POSIX function is common practice for verifying POSIX-compliant software (see for example `pthread` API stubs written by Inverso et. al. in [ITF+14]). CBMC and most verification tools are generally shipped with their own abstraction of some POSIX and standard C library functions), but they rarely cover the whole standards as this requires many work-hours. The list of missing functions that we had to implement is available in Table 4.2.

Since the amount of functions to stub was reasonably small, we implemented them manually and had the opportunity to embed in the stubs some *safety-checks* that we were able to derive from POSIX specification.

These *safety-checks* take the form of assumptions and assertions that we insert inside the function definitions, to ensure that the calls to the API from the verified software are valid: for example that a socket used for communication is created before being used, or that data-buffers used for networking refer to valid memory areas.

Function name	Short description
`socket`	create an endpoint for communication
`bind`	bind a name to a socket
`setsockopt`	get and set options on sockets
`sendto`	send a message on a socket
`recvfrom`	receive a message from a socket
`shmopen`	open POSIX shared memory objects
`mmap`	map files or devices into memory
`ftruncate`	truncate a file to a specified length
`open`	open and possibly create a file
`ioctl`	control device (non-POSIX)
`clockgettime`	retrieve and set the time of the specified clock

Table 4.2: List of stubbed POSIX functions

To illustrate our statement, we give an example of a stub *safety-check* used for network communication with socket. For communication to take place, the program must first

```
int _no_socket_fd = nondet_int();
int socket(...)
{
  // check correctness of input parameters

  file_descriptor = __VERIFIER_nondet_int();
  if(file_descriptor > -1)
  {
    __VERIFIER_assume(_no_socket_fd != file_descriptor);
    return file_descriptor
  }
  else return -1;
}

size_t recvfrom(int sock_fd, void* buffer, size_t buff_len, ...)
{
  __VERIFIER_assert(sock_fd >= -1, "sock_fd is a valid file descriptor");
  __VERIFIER_assert(sock_fd != _no_socket_fd, "sock_fd is a valid socket");

  // ...
  // non-deterministically modify the receive buffer
}

```

Figure 4.5: Snippet of the POSIX API verification stubs

create an endpoint using `socket()`, which returns -1 in case of failure, or a valid *file descriptor* (positive integer) upon success. Later, to receive data from this socket, the program shall call `recvfrom()` using the file descriptor that was returned by `socket()`.

The stubs displayed Figure 4.5 implements this *safety-check* using the verification tools' abilities: we use a global variable `_no_socket_fd` to memorize which integer values are valid socket file descriptors. When a file descriptor get assigned to a socket, we inform the verifier that `_no_socket_fd` *shall not* be equal to this file descriptor. When we later try to use the socket, we check that `_no_socket_fd` is *not equal* to the given file descriptor. The verifier should be able to infer that this is indeed the case. The advantage of this *safety-check* is that it can check the validity of arbitrarly many sockets, without using any datastructure: we rely solely on the capabilities of verification tools.

For some functions like `ioctl`, it was more difficult to implement precise *safety-checks*: the function behavior is not well standardized, depends on the underlying device drivers, and cover numerous and diverse use-cases. We therefore did our best to implement the few `ioctl` use cases that appeared in the targeted code, but did not cover all possibilities of usage of this function.

4.3 Code Transformation and Harness generation

One important challenge that we encountered along the way was to transform the code from its industrial form, to a form suitable for Software Verifiers. Moreover, as we have mentioned in Section 2.3.2 we are aiming for an approach adapted for Continuous Integration. This implies that all code transformation steps must be performed without human intervention, to permit the automatic verification of new code introduced daily or weekly.

4.3.1 Compiling and re-generating C code

The compilation of an industrial-scale C software project is quite an elaborate process. Usually, the complete software is an assembly of numerous C libraries, that are compiled independently and later linked together to form the final program. Furthermore, each source file is usually compiled with specific definitions (`#define`) and a specific set of include directories.

This highly contrasts with the acceptable input of most verification tools: CPAChecker , for instance, can only deal with pre-processed C files. A file merging feature exists, but is experimental[2] and did not fit our needs. Ultimate Automizer has similar limitations, and Frama-C proposes a different building paradigm that we did not evaluate.

Thanksfully, CBMC features `goto-cc`, a “compiler” which is able to perform all steps described above, with the difference that it outputs goto-models (a representation of C programs as control flow graph) instead of object files (.o). When the final model is obtained (after linking together all goto-models), `goto-instrument` allows to convert back the goto-model to C as a single source file (option `-dump-c`). The model, at this point, carries enough information to reconstruct a single C file very close syntactically to the original program, while remaining semantically equivalent. Unfortunately, the source file we obtained through this process had multiple problems rendering it un-compilable (none of the releases tested - 5.6, 5.8, 5.11, 5.12 - were able to output fully correct code).

We have also tried other approaches for generating a single C file, such as using the C Intermediate Language (CIL) [NMRW02]-tools for merging C files, but we found that the development of the CIL project halted four years ago[3], and is not compatible with modern GCC versions. We also tried to use Frama-C to merge files, as it is built on CIL and still actively developed. We found that Frama-C rejected some parts of the codebase, and that the resulting program after merging files was too different from the original for exploitation, and was also uncompilable.

The only solution we found was to fork the most recent release of CBMC and correct its implementation to solve all of the issues found in the exported C code. Problems included:

[2]as mentioned on their github page `https://github.com/sosy-lab/cpachecker/blob/trunk/README.md` (accessed on 6th July, 2020).

[3]as seen on the homepage of the project `https://github.com/cil-project/cil` (accessed on 6th July, 2020).

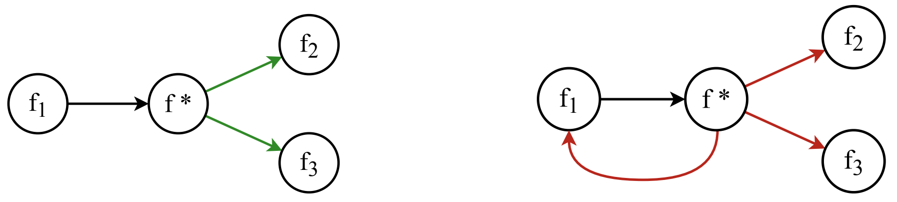
1. wrongly ordered, missing or superfluous type definitions, 2. invalid type casts (for example, casts between array types) 3. poor handling of variadic arguments, 4. occasional suppression of loops, 5. missing variable declarations after code instrumentation, and various minor syntactic problems. Solving these problems was quite a lengthy task but proved to be very useful in the long run, as CBMC was absolutely necessary for making multiple code transformations that makes the code acceptable for other verifiers.

4.3.2 Code augmentation

Having a single C file opened up the opportunity to use Frama-C for performing valuable pre-processing operations, namely:

- **Code Slicing**: Frama-C 's slicing plugin is able to determine which program instructions are relevant with regard to a set of chosen criterions. In our case, we have been using the most basic option, in which Frama-C removes all uncalled functions and unused global variables. While this, in theory, should not improve the verifiers performance, it greatly improved the readability of the final source code and also cleans up the verifiers input, as verifiers do not include irrelevant parts of the code in their output, giving us more realistic coverage report indicators.

- **Function Pointer Analysis**: We encountered some difficulties during the verification with CBMC that were caused by badly resolved function pointers. As CBMC starts the analysis by unrolling the transition relation, it needs a complete representation of the program *call graph* before running the model checking algorithm. Function pointers are therefore removed by a case-distinction over function pointer values, however, the pointer analysis performed by CBMC is very coarse. This tends to add infinite recursions in the call graph (see Figure 4.6), rendering the analysis infeasible with bounded model checking tools.



(a) True values taken by the function pointer f* (b) Erroneous values for f* induce a recursion

Figure 4.6: Control flow graphs demonstrating the consequences of imprecise function pointer analysis

Frama-C 's value analysis plugin is able to infer value-ranges for all global program variables, including function pointers, and is more accurate that the analysis performed by CBMC . Using a newly added feature of CBMC that we slightly improved[4], we were able to use the results of Frama-C pointer analysis to restrict the set of targets of each function pointers, solving the recursion problem.

During the verification, we check if the function pointer values computed with this method are actually correct (the apparition of an unexpected function pointer value triggers an assertion violation).

The flowchart describing the entire process to model the environment and prepare sources for verification can be seen Figure 4.7. Once the program has reached the right side of the diagram, the verification task is fully compatible with all tools participating in the SV-COMP, as long as they support all C constructs present in the program.

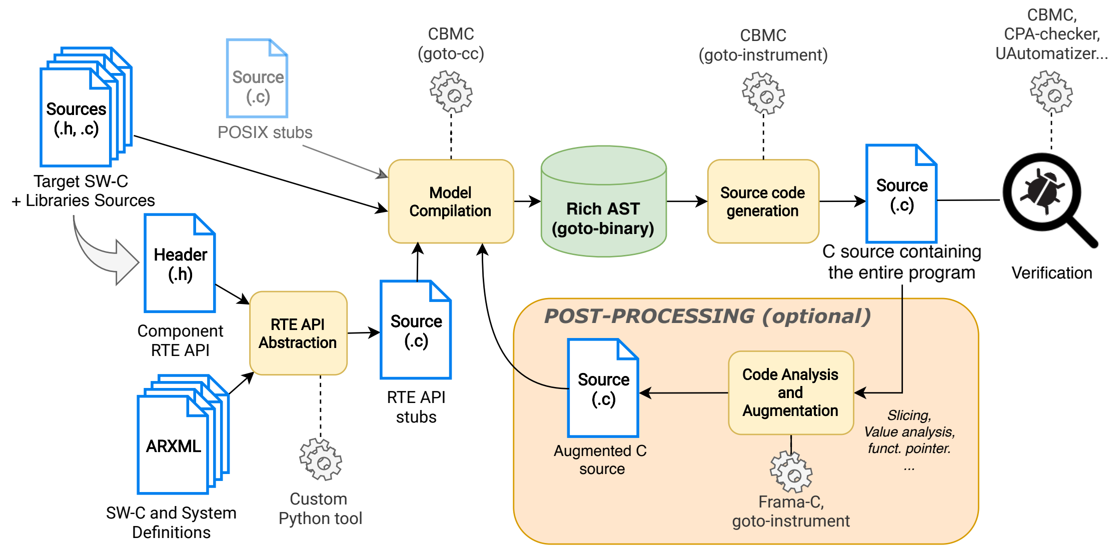


Figure 4.7: Environment Modelization and Code Setup Process

4.3.3 Setting up the code for k-induction

As mentioned in Section 3.2.2, we have been using a series of code transformations to perform k-induction proof on software components. Since the *base-case* of the proof is essentially identical to plain bounded model checking, we only need to implement the *inductive case*.

k-induction aims at proving that a particular property P hold for all loop iterations. In our case, the property P can be formulated as: “none of the assertions present in the code have been violated”. In order to track assertion violations, we use a global variable `assertion_was_violated` in conjunction with a custom implementation of

[4]the `-restrict-function-pointer` option of `goto-instrument` was added to CBMC in March 2020. We improved the function pointer type-checking method to make it more permissive (less strict with compatible function pointer types), and created function pointer variables for all function calls to ensure that Frama-C would compute their value-domain.

```c
int assertion_was_violated = 0;

void delayed_assert(int condition, int line_number)
{
  if(!condition)
  {
    // we must record that the property is violated
    if(assertion_was_violated == 0)
      assertion_was_violated = line_number;

    // we nondetermistically record this property's line number
    else if(nondet())
      assertion_was_violated = line_number;

  }
}

```

Figure 4.8: k-induction with CBMC : custom assertion function

the `assert` function called `delayed_assert`, which records the exact line number where an assertion violation occurred. Later on, we check with a standard assertion if the global variable evaluate to zero (no violations), or to a line number (where the assertion would be violated). The implementation of this transformation, along with an example, is given Figure 4.8 and Figure 4.9.

Storing the line number in the global variables allows the model checker to report precisely which assertion has failed, and to generate counter-examples on demand for a given assertion violation with the model checker (CBMC option `-property`). However, it should be noted that only the assertion `assertion_was_violated == 0` is strictly necessary for the correctness proof.

The simulation of an arbitrary loop iteration is based on the non-deterministic modification of **all** loop variables (as seen Figure 4.9, line 252). To do so, it is essential to detect all global variables[5] that can get modified in the call to `Runnable_Step()`. In practice, we use Frama-C 's value analysis plugin which returns the values of all global variables at the end of the function `Runnable_Step()`. Having them identified, we modify them non-determistically using the SV-COMP API, which constitutes an over-approximation of an arbitrary loop iteration.

It should be noted that CBMC propose its own implementation of k-induction, but we observed that it was very rudimentary or disfunctional. In our experience, it was not capable of finding all variables that could be modified in loop body, and was sometime causing verifier crashs. Consequently, we went with our own implementation.

[5]in theory, we would also need to consider static function variables. However, the code transformation presented in the previous section automatically promotes all static function variables to global variables, so we do not have to consider them.

Improving k-induction with value analysis

For improving the precision of the induction hypothesis, we have exploited the results of the value analysis performed by Frama-C . In particular, we extract the ranges of global variables in the end of the `Runnable_Step()` function, and integrated the property $R :=$ "*global variables are in range*" to the program correctness property P that must be proven. For doing so, we add corresponding assertions (base case) and assumptions (inductive case) in the generated code, as illustrated Figure 4.9 (lines 241, 262 and 271).

While this increases the workload on the verifier (more properties need to be proven), this also greatly reduces the possible state space and adds to the analysis a degree of precision that is sometimes necessary for obtaining successful correctness proofs.

```
void base_case_entry_point()
{
  Runnable_Init();
  int i=0;
  while(i < k)
  {
    Runnable_step();
    // check correctness of i-th iteration
    __VERIFIER_assert( assertion_was_violated == 0);

    // check validity of global variable(s) domain(s)
    __VERIFIER_assert( 0 <= global_var && global_var <= 10);
    i++;
  }

  // observe which assertion has failed
  __VERIFIER_assert( assertion_was_violated != 39  );
  __VERIFIER_assert( assertion_was_violated != 123 );
}

void inductive_case_entry_point()
{
  Runnable_Init()

  // modify loop variable non-deterministically
  some_global_loop_variable = __VERIFIER_nondet();

  int i=0;
  while(i < k)
  {
    // induction hypothesis
    __VERIFIER_assume( assertion_was_violated == 0);
    __VERIFIER_assume( 0 <= global_var && global_var <= 10);
    Runnable_step();
    i++;
  }

  // check program correctness
  __VERIFIER_assert(assertion_was_violated == 0);

  // check validity of global variable(s) domain(s)
  __VERIFIER_assert( 0 <= global_var && global_var <= 10);

  // observe which assertion has failed
  __VERIFIER_assert( assertion_was_violated != 39  );
  __VERIFIER_assert( assertion_was_violated != 123 );
}

```

Figure 4.9: k-induction with CBMC : base case and induction hypothesis

CHAPTER 5

Case Study

5.1 Targeted components

In order to study the feasibility of software verification applied to TTTech's code base, we have focused our study on three SW-C: The **Life Cycle Service (LCS)**, the **Middleware** and the **Vehicle Communication Service (ApCom)**. These three components have been chosen as targets because their implementation is representative of the code that is found on the platform: managing successful verification of these components would constitute a valuable proof of concept.

Addressing these components as SW-Cs is somewhat imprecise: indeed, AUTOSAR SW-Cs in theory have access to the RTE interface only, but the components we are examining actually have access to lower-level interfaces, as they implement platform-related services (e.g. vehicle communication management) rather than application-related services (e.g. managing the car's dashboard display). This access to lower-level code increases the verification complexity, however, we argue that it makes the challenge more interesting, and furthermore, it is in conformance with TTTech's need for verification of safety-critical code.

Nevertheless, all of these components still conform to the paradigm that was defined in Section 4.1, having *initialization* and *step* functions, and an RTE interface means that modeling them as SW-Cs makes sense from the verification point of view.

In the following, we give a brief description of each component:

5.1.1 LCS

The Life Cycle Service is responsible for managing the life-cycle states of the ECU and hosts, and can be queried by other components to get the current state or request state transitions. Examples of system states include the *Startup phase*, the *Running phase* or the *Shutdown phase*. The life cycle service is divided in two separate components, the *master* (LCS-M) and *slave* (LCS-S), running on the safety host or the performance hosts respectively.

5.1.2 Middleware

The Middleware provides an implementation of the AUTOSAR RTE that abstracts all communications for the higher level application SW-C. It coordinates inter and intra-host communication and performs for this purpose tasks such as packet handling or routing, and makes heavy use of lower-level communication APIs.

5.1.3 ApCom

The Vehicle Communication service (ApCom) is yet another component abstracting a communication interface. It performs the junction between SW-C hosted on the machine and automobile-specific communication buses (e.g. CAN, FlexRay or Ethernet), and performs the necessary conversions between the internal and physical representation of application DataTypes.

5.2 Code metrics

In order to give more insights about the difficulty levels of these verification tasks, we have compiled in this sections several code metrics. The final C source file that we obtained for each component contains more than the code that we are targeting, as several sizeable libraries are linked with the component-specific code, while not being used entirely. In order to exclude these chunks of code from the metric calculation, we perform slicing of unreachable functions, and then compute the metrics.

This is achieved with Frama-C using the following command,

```
frama-c <source-file> -main entry_point -eva -slevel 0
-no-val-alloc-returns-null -slice-calls entry_point
-slicing-level 0 -then-on 'Slicing export'
-print -ocode <sliced-file>
```

followed by:

```
frama-c -metrics <sliced-file> &&
frama-c -metrics -metrics-ast cabs <sliced-file>
```

The slicing is performed on an over-approximative value range analysis on all program variables, based on Frama-C 's `eva` plugin. The sliced code is then exported to a C file, on which we compute the metrics that are presented in Table 5.1.

Categories	**LCS-S**	**LCS-M**	**ApCom**	**Middleware**
Operations				
Pointer dereferences	50	115	2222	2170
Additions & Substractions	31	129	330	3662
Multiplication & Divisions	36	76	898	471
Bitwise operations	10	14	11	304
Control flow				
If	119	243	1276	948
Loops	4	17	77	76
Function calls	129	309	1347	1328
Function Returns	66	136	365	329
Complexity and Difficulty				
Lines of C code	1469	4923	15973	16536
Program Locations	529	1182	5935	7061
Global Variables	34	94	427	584
MacCabe Cyclomatic Complexity[1]	187	410	1681	1895
Halstead Length	4716	12656	51610	126484
Halstead Volume	43646.71	132721.38	589193.68	1635554.75
Halstead Difficulty	209.86	875.99	1526.54	6282.49

Table 5.1: Code metrics for verification targets

In addition to counting metrics, we give the values of McCabe's Cyclomatic complexity [McC76]. Roughly speaking, it gives a measure of the control-flow-graph's complexity and denotes the minimal number of different paths that exist in the code. Additionally, we provide Halstead Length, Volume and Difficulty [H+77], three measure derived from the number of *operators/operands* and the number of *distinct operators/operands* appearing in the program's expressions. They are generally used to express the difficulty of programming or understanding a given program from the perpective of a human operator. It is worth mentioning that these metrics are commonly used to measure the complexity of *hand-written code*, while important parts of the code-base inspected here are produced by *code generators*.

These metrics precisely illustrate the difference of complexity between the software components of our case study. The Life Cycle Service slave and master are the most simple programs, as the maintenance of a state machine does not require computa-

[1] It should be noted that MacCabe's cyclomatic complexity is generally used to measure the complexity of a function, and should have in this context values inferior to 30. In our case, we compute a program-wide cyclomatic complexity, which is not comparable in terms of scale, but should be useful to illustrate the differences in complexity between our targets.

tional complexity. As the master component performs supervision and some platform initialization-related tasks, it displays a slightly higher complexity than the slave component, which merely maintain the state machines according to the received instructions.

The last two components, ApCom and Middleware, are **primarily constituted of auto-generated code** displaying a **repetitive structure**, which is the main reason explaining the relatively high complexity metrics.

The Vehicle Communication component represents an increase in complexity, which can be explained by the vast amount of distinct *DataTypes* that must be handled, as they are translated between the internal representation and their representation on the vehicle buses. Data elements are passed around as pointers, and also provide function pointers to their peculiar conversion, reading and writing functions, which implies an important amount of pointer arithmetic and dereferences that is confirmed by the metrics.

Finally, the Middleware is certainly the most complex component, as it is charged with coordinating the communication between all SW-Cs on the platform. The intensive manipulation of communication buffers is characterized by the high number of pointer dereferences and additions. Additionnally, we see in this component a fair amount of bitwise operations, throwing light on the interaction with lower level POSIX communication interfaces making heavy uses of flags, as well as the manipulation of data frames.

5.3 Instrumentation of program properties

The main goal of the verification tasks that we run is to prove the absence of Run-Time errors in the program. The most severe sources of Run-Time errors in C code are instructions depicting **Undefined Behaviors**. Undefined behavior appears when an instruction's semantics are not specified by the C standard, and yields unpredictable results. This means that the compiler designer can himself chose the result of the operation, which might very well be an early program termination (crash), unwanted modifications in arbitrary memory areas, unexpected values appearing in program variables, or security vulnerabilities to attackers.

The C standard documents 199 undefined behaviors (see Annex J.2 of [ISO99]), many of which can be detected during the compilation phase and signaled with a warning. However, in many cases, undefined behaviors can only be detected during the execution of the program. Depending on the compiler implementation, the undefined behavior can result in a crash, or fail silently, which can make them difficult to detect through standard testing methods.

Our verification procedure allows us to detect a broad range of undefined behaviors, as well as behaviors that are defined but generally not wanted (labelled "Dangerous"). The list is available in Table 5.2. This list of detectable bugs is derived from the list of vulnerabilities that can be instrumented natively by CBMC .

Erroneous operation	Category	Implementation
(Static) Array access out of bound	Undefined	Assertion
(Pointer) Array access out of bound	Undefined	Tool-specific
Invalid pointer comparison	Undefined	Tool-specific
Invalid pointer dereference	Undefined	Tool-specific
Division by zero	Undefined	Assertion
Signed Integer overflow	Undefined	Assertion
Unsigned Integer overflow (“wrap”)	Dangerous	Assertion
Lossy type conversion	Dangerous	Assertion
Pointer Arithmetic overflow	Undefined	Tool-specific
Undefined shift	Undefined	Assertion
Float overflow	Undefined	Assertion
Apparition of NaNs	Dangerous	Assertion

Table 5.2: Targeted undefined and dangerous behaviors

Instrumenting vulnerabilities in the code

Using the code generation functionality (see Section 4.3.1), CBMC is capable of inserting assertions in all code locations that present a potential vulnerability with regard to operations listed in Table 5.2. Yet, while some safety-checks are easily implemented as assertions (e.g. division by zero: `assert(denominator != 0)`), most of the memory-related checks cannot be encoded as such: indeed, the C language does not provide ways to detect if a memory area is valid, or to retrieve the size of an object pointed-to by an arbitrary pointer. For these reasons, memory-related checks were not instrumented in the code as assertions, and their implementation were left to the verification tools.

Besides, the assertions generated by CBMC for signed, unsigned, pointer integer overflow, as well as undefined shift were not expressed with valid C code. The assertions for checking the absence of overflows were originally inserted in the following form, for a given type `my_custom_type`:

```c
typedef unsigned int uint32;
assert( !overflow('+', uint32, operand_A, operand_B) );
```

This code has several issues: 1. the function overflow is not defined, 2. it is not a valid C function as it takes a type as a parameter, 3. it is also not possible (to our knowledge) to implement it as a macro. We have therefore slightly modified CBMC 's implementation to replace this assertion by the following,

```c
assert( !overflow_signed_int('+', operand_A, operand_B) );
```

and implemented the corresponding function for all basic C integer types. This function returns 1 if adding operands A and B would result in an overflow, and 0 otherwise. The code for detecting signed integer overflows and unsigned integer wrap-arounds before they occur was inspired from the definitions of rules INT30-C and INT32-C of the SEI CERT C Coding Standard [Sea14].

For undefined shifts, we discovered that CBMC sometimes generates an array-like notation to extract the value of one or multiple bits (e.g. `bits = 9849[3,5]` to extract the third, fourth and fifth bits of `9849`), which is not a valid C operator. This has been replaced by a valid C expression implementing the desired semantics (i.e. using the same example, `bits=(9849»3)&∼(∼0U«(5-3))`).

Finally, we have found while running experiments that CBMC 's assertions for pointer arithmetic overflow were not properly covering undefined behavior related to pointer arithmetic. In particular, the C standard forbids the incrementation of a pointer to an array past the memory area that it is accessing (see §6.5.6 Additive operators, para. 8 of [ISO99]), as demonstrated in the following code snippet:

```c
int* pointer_to_array = &array[0] + sizeof(array) + 1; // undefined behavior
```

In contrast, the analysis performed by CBMC on pointer arithmetic only verifies if the underlying integer representation of the pointer overflows, which is an incomplete test and does not cover this undefined behavior. This issue was reported to the tool's authors, who acknowledged the uncompleteness of the current analysis, that would need to be extended in a future version[2]. Since this safety-check appeared to be unreliable at the time of writing[3], we decided to exclude it from our case-study.

Safety checks locations for each target

We list in Table 5.3 the number and type of assertion that were added at potential bug locations in the code for each verification target.

It should be noted array accesses out of bounds and invalid pointer dereferences are grouped together, since they have precisely the same semantics (an array access on a pointer is the combination of a pointer incrementation and a pointer dereference).

[2]See corresponding issue on GitHub: `https://github.com/diffblue/cbmc/issues/5426` (Accessed 20th July, 2020)

[3]Another bug can be seen at `https://github.com/diffblue/cbmc/issues/5284` (Accessed 20th July, 2020)

Vulnerability type	Number of assertions per component			
	LCS-S	LCS-M	ApCom	Middleware
(Static) Array access out of bound	3	32	7	997
(Pointer) Array access out of bound Invalid pointer dereference	328	1612	15080	6456
Invalid pointer comparison	0	0	0	3
Division by zero	0	1	8	2
Signed Integer overflow	3	5	29	569
Unsigned Integer overflow (“wrap”)	5	70	73	1140
Lossy type conversion	13	66	250	482
Undefined shift	8	5	1	16
Float overflow	0	0	50	5
Apparition of NaNs	0	0	42	5
(Correct function pointer analysis)	6	15	22	5
Total	366	1806	15562	9680

Table 5.3: Code metrics for verification targets

CHAPTER 6

Experiments and results

All experiments were conducted on a Intel Core i7-6700 CPU @3.40GHz (4 cores / 8 threads) with 24GB of memory, running x86_64-linux (Ubuntu 18.04.4). The benchmarks were performed with the help of *BenchExec*, a tool for facilitating benchmarks that allows to track different metrics such as memory or CPU usage, and was developed specifically for measuring verification tools' performances [BLW19].

6.1 Proving the absence of Run-Time errors:

The experiments have been performed with a method comparable with a real-world development, testing and validation process. In a first phase, we have been examinating all errors reported by verification tools to determine if 1. they corresponded to genuine bugs, 2. were due to approximations of the environment model, or 3. were spurious bug-reports from the verifiers. We then manually corrected all potential errors that were found in the program, and evaluated the capacity of verifier to provide a correctness proof on these supposedly *bug-free* programs.

For doing so, each verifier was given a maximum running time of one hour (3600 seconds of CPU Time) per SW-C, and a maximum memory usage of 21GB.

The results are regrouped in Tables 6.1, 6.2, 6.3 and 6.4. The result "Success" appears if the verifier implements a sound proof system, terminates, and reports that no bug was found in the code. The result is "Unknown" if the verifier does not give a definitive answer, which could stem from an exhaustion of memory resources (OOM), time resources (Timeout), or a bug withing the verification causing an early termination (Exception). Finally, we report the result "Failure" if the verifier's result is known to be invalid (typically, if the verifier reports a spurious counter-example).

Note that only verification methods marked with a "□" are able to provide a sound correctness proof of the program (guaranteeing the absence of run-time errors).

	Verification method	Mem. (MB)	Time (s)	# of proven properties	Result	Reason
	CBMC	354.7	550.4	370 (all)	Unknown	Bound
	CBMC + k-Ind. B	1967.0	391.7	370 (all)	Success	-
	CBMC + k-Ind. I	2423.3	423.1	370 (all)	Success	-
☐	CBMC + k-Ind. $\forall$	2423.3	814.8	370 (all)	Success	-
	CBMC + k-Ind. B + values	1967.2	387.2	370 (all)	Success	-
	CBMC + k-Ind. I + values	2423.6	424.0	370 (all)	Success	-
☐	CBMC + k-Ind. $\forall$ + values	2423.6	811.2	370 (all)	Success	-
	CPAChecker reach safety	5869.2	MAX	?	Unknown	Timeout
	CPAChecker deref. safety	3035.0	87.6	?	Unknown	Exception
☐	CPAChecker (total)	5869.2	MAX	?	Unknown	Timeout
	U-Automizer reach safety	4667.1	MAX	?	Unknown	Timeout
	U-Automizer deref. safety	4801.5	MAX	?	Unknown	Timeout
☐	U-Automizer (total)	4801.5	MAX	?	Unknown	Timeout
☐	*Overall*				Success	

Table 6.1: Correctness proof results on the LCS-S

	Verification method	Mem. (MB)	Time (s)	# of proven properties	Result	Reason
	CBMC	1358.1	1044.0	1824 (all)	Unknown	Bound
	CBMC + k-Ind. B	2486.0	1134.4	1824 (all)	Success	-
	CBMC + k-Ind. I	4808.0	2412.0	1823	Unknown	Ind. fail.
☐	CBMC + k-Ind. $\forall$	4808.0	3546.4	1823	Unknown	Ind. fail.
	CBMC + k-Ind. B + values	2488.5	1146.7	1824 (all)	Success	-
	CBMC + k-Ind. I + values	4617.8	2420.0	1824 (all)	Success	-
☐	CBMC + k-Ind. $\forall$ + values	4617.8	3566.7	1824 (all)	Success	-
	CPAChecker reach safety	MAX	1829.0	?	Unknown	OOM
	CPAChecker deref. safety	6304.8	138.1	?	Unknown	Exception
☐	CPAChecker (total)	MAX	1967.1	?	Unknown	OOM/Exc
	U-Automizer reach safety	865.6	149.7	?	Unknown	Exception
	U-Automizer deref. safety	866.9	144.2	?	Unknown	Exception
☐	U-Automizer (total)	866.9	293.9	?	Unknown	Exception
☐	*Overall*				Success	

Table 6.2: Correctness proof results on the LCS-M

	Verification method	Mem. (MB)	Time (s)	# of proven properties	Result	Reason
	CBMC	527.4	187.3	15597 (all)	Unknown	Bound
	CBMC + k-Ind. B	3967.8	1533.5	15597 (all)	Success	-
	CBMC + k-Ind. I	3754.6	1566.6	15597 (all)	Success	-
☐	CBMC + k-Ind. $\forall$	3967.8	3100.1	15597 (all)	Success	-
	CBMC + k-Ind. B + values	3968.6	1528.5	15597 (all)	Success	-
	CBMC + k-Ind. I + values	3756.2	1582.8	15597 (all)	Success	-
☐	CBMC + k-Ind. $\forall$ + values	3968.6	3111.3	15597 (all)	Success	-
	CPAChecker reach safety	11964.9	MAX	?	Unknown	Timeout
	CPAChecker deref. safety	12066.7	243.5	?	Failure	Spurious bug
☐	CPAChecker (total)	11964.9	MAX	?	Unknown	Timeout
	U-Automizer reach safety	1100.5	175.2	?	Unknown	Exception
	U-Automizer deref. safety	1094.4	169.7	?	Unknown	Exception
☐	U-Automizer (total)	1100.5	344.9	?	Unknown	Exception
☐	*Overall*				Success	

Table 6.3: Correctness proof results on the ApCom

	Verification method	Mem. (MB)	Time (s)	# of proven properties	Result	Reason
	CBMC	7039.6	MAX	?	Unknown	Timeout
	CBMC `-unwind 4`	MAX	547.87	?	Unknown	OOM
	CBMC + k-Ind. B	5177.4	MAX	?	Unknown	Timeout
	CBMC + k-Ind. I	5175.1	MAX	?	Unknown	Timeout
☐	CBMC + k-Ind. $\forall$	5177.4	MAX	?	Unknown	Timeout
	CBMC + k-Ind. B + values	5215.1	MAX	?	Unknown	Timeout
	CBMC + k-Ind. I + values	4908.0	MAX	?	Unknown	Timeout
☐	CBMC + k-Ind. $\forall$ + values	5215.1	MAX	?	Unknown	Timeout
	CPAChecker reach safety	3216.1	MAX	?	Unknown	Timeout
	CPAChecker deref. safety	7440.9	347.5	?	Failure	Spurious bug
☐	CPAChecker (total)	7440.9	MAX	?	Unknown	Timeout
	U-Automizer reach safety	736.9	66.4	?	Unknown	Exception
	U-Automizer deref. safety	766.5	66.1	?	Unknown	Exception
☐	U-Automizer (total)	766.5	132.5	?	Unknown	Exception
☐	*Overall*				Unknown	

Table 6.4: Correctness proof results on the Middleware

6.2 Result analysis and interpretation

Of all verification tasks, the investigated verification methods have been able to prove the correctness for three of them. For the last one, it appears that none of the verifiers had the capacity to finish the proof with the given resources. In the following, we interpret the results and give some insights about the causes of failed verifications.

6.2.1 CPAChecker

The full correctness proof conducted with CPAChecker is divided in two step: The *reach-safety* step verifies that none of the assertions in the program are being violated, and the *dereference stafety* proof ensures that all pointer dereferences and array accesses are correct. Our experiments with CPAChecker shows that the tool is not yet mature enough for handling our code base.

Reach Safety All reachability safety analysis by CPAChecker ran out of resources before a verdict could be given. According to the logging files, the algorithm seems to spend almost all of its time searching for loop invariants. In comparison, the time invested by CPAChecker for value analysis, CEGAR or k-induction is very limited, which might explain the poor results.

Dereference Safety We have been unable to perform pointer dereference safety checks using CPAChecker in the presence of string literals. The following snippet displays two lines of valid C code (triggering no errors or warning when compiled with usual compilers) that CPAChecker has been unable to process.

```c
const char*  first_configuration =  "Hello world";
const char* second_configuration = &"Hello world"[0];
```

In the first configuration, CPAChecker would immediatly stop the analysis and report the string assignment as an invalid pointer dereference, which constitutes a *spurious counter-example*.

In the second configuration, CPAChecker would perform the analysis for a couple of minutes, but eventually terminate and throw an exception, reporting an issue with the conversion of `(&"Hello world"[0])` to a `char` pointer in the internal program representation.

For this reason, we found ourselves unable to check the code for both the LCS-M and the LCS-S component. For the middleware, CPAChecker checker reported an invalid pointer dereference that we have identified to be a spurious bug report, after careful inspection of the reported trace. Indeed, the error was signaled on a dereference which was clearly guarded and occured just a few instructions after the variable allocation. A spurious report was also signaled on the ApCom component, where a dereference to a constant global pointer was reported. This pointer was initialized in the beginning of the program with the address of a global integer constant and was never later modified. Therefore, it could not possibly point to an invalid memory location.

6.2.2 Ultimate Automizer

While successfully running on some toy examples that we experimented with, we were unable to have Ultimate Automizer run on the complete code-base for most SW-C. Despite substantial efforts for removing C constructs that the tool was not about to process (such as gcc `__attribute__` and `_Noreturn` extensions, along with all functions using variadic arguments), we hit a severe blocking point as the tool systematically terminated raising an exception during the verification. Due to the absence of meaningful information about the root of the problem in the log files (only a backtrace was given), we were unable to identify which line of code in the input program was the source of the error.

During the middleware verification, Ultimate Automizer additionally reported a syntax error at a location displaying well-written C code. We have tried to reformulate the problematic statement with several variants, but none of them led to a successful parsing from the tool. The error occured in the module `CACSL2BoogieTranslator`, and we suspect that the line number mentioned in the error report does not correspond to the line number which really caused the issue.

Only for the LCS-S, the tool was about to complete its verification run until the end of the allowed time frame, although it was not able to prove the program's correctness.

6.2.3 CBMC

CBMC has been tested in three different configuration:

- CBMC : the plain bounded model checking configuration, tested with a bound of 5 on the main loop, and other loops left unbounded.
- CBMC + k-Ind.: CBMC augmented with k-induction, with one run for the base case (B) and one for the inductive case (I), the combination of which provides a sound proof for all loop iterations ($\forall$). A value of five was used for k, as it was the highest we could go while keeping manageable running times, and proved to be sufficient.
- CBMC + k-Ind. + values: CBMC using *value ranges* calculated by Frama-C for strengthening the k-induction hypothesis. We again have one run for the base case (B) and one for the inductive case (I), with k also set to a value of five.

CBMC CBMC alone does not constitute a sound proof system: therefore all of its results are marked as Unknown, even when the algorithm finished reporting no errors[1]. Only for the Middleware, CBMC was not able to finish in time. In this case, we observed that CBMC was not able to unroll the full transition relation and build the SAT formula

[1] Technically, the *unwinding assertion* on the main loop cannot ever hold. More details about the unwinind assertions can be found at `http://www.cprover.org/cprover-manual/cbmc/unwinding/` (Accessed 13th August, 2020)

within a one hour time range, which is due to the high number of nested loops that greatly increase the complexity of the unrolled program expression, despite the fact that all loops had finite bounds.

Upon observing this, we have been trying to force a fixed bound of four for all loops in the program (using the command line option `-unwind 4`), which greatly simplifies the program expression (sacrificing soundness). However, a second blocking point emerged, as CBMC ran out of memory during the post-processing of the boolean formula. We tracked down the cause of this issue to be large arrays of shared memory, that tend to be represented explicitely in the internal program representation of CBMC, which greatly complicates the verification task. As we have learned from other users' experience, it seems unlikely that providing more memory to CBMC will solve the issue[2]. Unfortunately, this limitation means that CBMC cannot even be used for bug-hunting on the Middleware, as the verification can not proceed without completion of the pre-processing step.

One way to fix this issue would be to use a smarter abstraction for arrays. For example, CBMC using the Z3 SMT solver [DMB08] as a backend can avoid representing arrays explicitely, and instead rely on an SMT encoding that makes use of a *theory of arrays*. Unfortunately, all of the CBMC releases that we have tried suffered from run-time errors when using Z3 as a backend. We have tested CBMC release 5.6, 5.8, 5.10, 5.11 and 5.12, and all of them raised different kind of exceptions during the encoding of C statements to their SMT representations. Investigating these problems and fixing them would be one of the task with highest priority in the continuation of this verification project.

CBMC with k induction (and value analysis) CBMC augmented with k induction ended up being the most capable method explored in this thesis. Even with low values for k, the verification tool is capable to gather enough information about the feasible state space to perform the proof. We interepretate it as a sign that loop counters in our program do not have an important variability, and that loop invariants required to compute the proof must be rather simple. On the other hand, we observed in the case of the LCS-M that k-induction alone was unable to prove the program, and that global variable value ranges computed by Frama-C are necessary for completing the proof.

We were able to identify precisely an assertion that caused problem to plain k-induction: One array access appeared in a code section which execution depends on a non-determinitic value return by the RTE API stubs. During the first k iteration of the inductive proof, the verifier can "chose to avoid" the problematic section, consequently not learning anything about valid ranges for the array access. On the last iteration ($k+1$), the verifier will finally discover this section and report an array out of bound error, as it does not know the values of the array index. A minimal code snippet illustrating this behavior is available Figure 6.1. In this example, k-induction alone cannot find valid ranges for `array_index` and `tmp`, which are necessary to prove the program's correctness.

[2]A JBMC user experienced the same problem as us, where would CBMC consumes up to 128GB of RAM in the presence of large arrays `https://github.com/diffblue/cbmc/issues/211`

```c
// global variable initialization
int array_index = 0;
int array[5];
int tmp;

int main()
{
  // inductive case: non-determinitic modification of loop variables
  array_index = __VERIFIER_nondet_int();
  tmp         = __VERIFIER_nondet_int();

  for( /* k steps */ )
  {
    // induction hypothesis
    __VERIFIER_assume( /* no assertion was violated */ );

    // loop body
    tmp = array_index;
    array_index++;

    if( array_index == 5 )
      array_index = tmp;

    if(__VERIFIER_nondet()) // this section can be skipped at will
    {
      assert(0 <= array_index && array_index <= 4);
      array[array_index];
    }
  }
  __VERIFIER_assert( /* no assertion was violated */ ); // k-induction cannot
     prove this
}

```

Figure 6.1: Example of k-induction failure solved with value analysis

Frama-C can easily find that both variables are in the range [0..4], which is sufficient for completing the proof.

k-induction with value analysis proved to be a powerful enough proof system for our use cases, proving the correctness of the LCS-S, LCS-M and ApCom. Unfortunately, the method did not scale in the case of the Middleware, due to excessive running times, but most importantly due to the poor management of large arrays by CBMC mentioned in the previous section.

6.2.4 Verifiers performance and task complexity

It can be observed that the verifiers running time and memory consumption is only weakly correlated to the complexity of verification tasks. Indeed, while the LCS-M shows complexity and difficulty metrics clearly lower than the ApCom, proving the correctness of the latter required fewer resources. Similarly, proving the correctness of the Middleware has been out of reach for all tested software verifier, while the component is not drastically more complex than the ApCom, according to code metrics.

One of the most decisive factors that we have identified to be of high impact on CBMC 's running time is the number of nested loops with high bounds. An example of this observation has been made with the attempt to verify a critical function containing two nested loop with fixed bounds equal to 256. While the function merely spanned a hundred lines of code, verifying it took more time that all verification tasks that we have documented here (in total, a dozen of hours was needed for the proof). In practice, such functions need to be abstracted.

One final observation that we would like to point out is the important difference in performance between CBMC in its standard *bounded model checking* configuration, and CBMC used for proving the *base case* of k-induction. As we have pointed out in Section 3.2.2, the verification performed in the two cases are equivalent, yet, we observe significant performance gaps between them. The suspected reason behind this is the different encoding we used for error detection. For k-induction, all assertions were expressed with the `__VERIFIER_delayed_assert` function in the C code, while plain-CBMC parsed C code without assertions, and added them later in its internal representation, which apparently turns out to be more efficient.

This suggests that the performance of our k-induction implementation could be significantly improved by encoding it directly within CBMC 's internal representation (or by resolving the issues with the already-existing implementation of k-induction within `goto-instrument`).

CHAPTER 7

Discussion and Conclusion

7.1 Summary of results

Our experiments have demonstrated that the verification of AUTOSAR SW-Cs for proving the absence of Run-Time errors is definitely achievable for verification tasks of reasonable size. Unfortunately, none of the approaches that we adopted so far has been able to cope with the most ambitious verification task that was part of this case study. However, the most important source of this limitation being a bug in CBMC, we have hopes that solving it would allow us to go further with experiments, and increase the amount of code covered by our verification procedure.

Benefits for software quality

We have observed along the project that verification tools can bring a real benefit to the development process and the general quality of the software. Firstly, the tools guarantees to find bug locations that could possibly be missed through standard testing methods, in particular bugs associated with array-out-of-bounds errors. These errors are hard to detect as an out-of-bound writes do not necessarily cause any directly observable errors, but can nevertheless have unexpected consequences. As we have seen in Section 5.3, these also generated the highest number of safety checks (assertions) inserted in the code.

Secondly, we have observed that using non-deterministic stubs for the RTE API, permitted to observe the behavior of the program against the whole feasible range of each AUTOSAR DataType. This allows not-only to detect Run-Time errors, but also to verify the compliance of the software with the ARXML specifications, performing a broad exploration that could not be achieved through standard testing, or even fuzzing.

7.2 Limitations of the current approach and envisioned improvements

Currently, several improvements over the current approach can be identified:

- **Improve code coverage**: While we achieved some success, one of the current indicator that we would like to increase is the code coverage achieved by the verification. Several factors are preventing us from verifying big parts of the code: Firstly, some of the code cannot be processed by the tool in its current form, as it contain some non-standard, platform-specific C extensions that CBMC and Frama-C cannot process. Secondly, verifying new parts of the code-base will likely come with obstacles that are unknown to us today. Experience has taught us that targeting new chuncks of code for verification always brings unexpected challenges.
- **Strenghten the experimentation around k-induction**: as the most successful approach, continuing the development around k-induction is a natural progression to improve the proof system and allow a broader variety of use cases. Currently, we benefit from the single-loop hypothesis, however we have encountered occasional unbounded loops in AUTOSAR SW-C, in particular when the loop condition depends on some non-deterministic value coming from the AUTOSAR stubs. Therefore our approach would benefit from being extended to program with multiple, nested unbounded loops. Moreover, stronger proof methods based on k-induction exist and are implemented in other experimental tools, such as *DepthK* [RRI$^+$17], which could be evaluated.
- **Harvest more information from the ARXML files**: Currently we have mainly been using *DataType* information extracted from AUTOSAR specifications, and we still rely on SW-C headers provided by the user for generating our stubs. The abstraction of AUTOSAR interfaces could be streamlined by relying only on specifications for generating stubs. The stubs themselves could also be refined to provide a tighter over-approximation, by modeling the RTE API semantics more precisely.

7.2.1 Difficult interfaces between the code and verification tools

Reflecting on Section 4.3.1, one of most time consuming aspect that was encountered during this project has been the difficulty of preparing the industrial codebase for verification with the software verification tools evaluated in this study. Some critical aspects were the:

- **Incompatibilities with build frameworks**: with the availability of `goto-cc`, a tool mimicking GCC, CBMC has proven to be the tool the most adapted to C build frameworks. Still, `goto-cc` has some drawbacks, like the inability to resolve conflicts when linking the same library twice, or various problems with linking system libraries.

- **Incompatibilities with non-Linux platforms** Of course, compiling code that is not meant to be build with GCC (e.g. windows code, or code deployed on more exotic platforms that are common in embedded system) causes even more problems that we did not solve entirely to this day. We encounter here the same issues that rises when maintaining code for different architectures, but they are exacerbated by the fact that verification tools are not as well-maintained as compilers.

- **Different C subsets accepted by tools** Obtaining code that would be suitable for both Frama-C , CPAChecker and CBMC has proven to be a real challenge, as all these tools accept a slightly different subset of the C language, either because of implementation choices, because of non supported constructs, or because of bugs. We would have appreciated to test a wider range of tools in this study, but the introduction of new tools systematically comes with high time costs.

While this is not a scientific challenge, we argue that these issues are of high importance for the software verification field, as they are a hurdle for the broad adoption of verification tools in industry. Consequently, the number of companies ready to invest enough man-hours for evaluating verification tools is lessened.

7.3 Further works

7.3.1 Industrialization of results

This project would demonstrate its full usefulness only when deployed as a standard tool within the industrial build, test and validation process. Accomplishing this deployment is still a long way to go, as the whole verification framework, and in particular the verification results would benefit from a more streamligned interface, and several steps requiring human intervention would need to be eliminated. However, there is to our knowledge no major technical challenge to be expected, hence the number of work-hours required for performing this task should be easy to quantify.

7.3.2 Proving more advanced program properties

This study focused on detecting undesired behavior in C code, to demonstrate the feasibility of software verification on TTTech's code. While the results obtained are certainly valuable, more advanced program properties could certainly be targeted, as proving the absence of undefined behaviors comes short of proving the correctness of a software platform.

One of the following working topic where verification based on formal method could gain traction is the verification of non-interference properties between AUTOSAR SW-Cs. Indeed, a correctness proof of a given SW-C could easily be negated by the presence on the same platform of another faulty component which might inadvertently write in memory areas reserved to others. For guaranteeing that this cannot happen and ensure component isolation, strong non-interference proofs are needed. It remains to be determined whether the current approach could be adapted for proposing a non-interference proof system (CBMC has been used for such purposes [MTD16]), or if a different tooling environment would be more appropriate.

7.4 Conclusion

In this thesis, we have demonstrated that modern verification tools targeting C code are capable of proving the absence of Run-Time errors for an important subset of automotive software components. For the problematic cases, we tried to point out precisely the causes of failure of model-checking.

We have shown that the widely-adopted automotive standard AUTOSAR provided a great advantage for verification, as standardized interfaces and specifications is crucial to devise a environment model precise-enough to achieve successful verification of said components.

The focus has been put on building a fully-automated solution, that is able to give its conclusions about the software without requiring human intervention at any step.

We experienced that the efforts of standardization of verifiers interfaces made possible by the SV-COMP are extremely useful for using various tools on the input code. On another hand, the results observed in our case study vastly differed from the results of the international competition, as best ranking tools struggled to verify our software. This suggests that the verification tasks evaluated in the SV-COMP do not compare well to our real-world problems.

We have observed that none of the verification tool was able to provide the desired results out-of-the-box. Conversely, a combination of different analyses proved to be the only way to achieve full correctness proofs. Therefore, we conclude that verification techniques should not be evaluated on an individual basis, but their results should be reused.

k-induction turned out to be the most capable technique for dealing with programs contain exactly one unbounded loop. Therefore we strongly advise researchers to focus on this method and its derivative for further work on verification of embedded software components displaying a similar structure.

List of Figures

List of Tables

Glossary

Complex Device Driver Software entity which is only sparsely standardized by AUTOSAR, in comparison to Software Components or Basic Software Modules. It typically has access to other BSW module as well as the RTE and may have access to harware specific interfaces.[AUT15]. 23, 26

Software Development Lifecycle Model Methodology used for developing, testing and maintaining a software. Common models include the *Waterfall Model*, the *V-Model*, the *Spiral-model* or the *AGILE methodology* [Rup10]. 11

System Definition Set of formalized specifications describing the architecture of an AUTOSAR compliant platform. It contains definitions of Software Component interfaces, a description of the underlying harware (ECU), communication networks, etc. In practice, the *System Definition* is given in the form of ARXML files.. 11

C Bounded Model Checker (CBMC) CBMC is a Bounded Model Checker for C and C++ programs. It supports C89, C99, most of C11 and most compiler extensions provided by gcc and Visual Studio. see: `http://www.cprover.org/cbmc/` [KT14]. 15, 16, 26, 27, 29–32, 34, 38–40, 44, 45, 47–55

Not a Number (NaN) result of an invalid floating point operation, such as the square root of -1. 39

Portable Operating System Interface (POSIX) "[POSIX] defines a standard operating system interface and environment, including a command interpreter (or "shell"), and common utility programs to support applications portability at the source code level."[IEE17]. Among other, the POSIX API proposes functionnalities for Process creation and control, file and directory operations, I/O operations, Real-Time software, Mutli-Threaded software, etc. [Wal95]. 23, 26–28, 55, 57

Continuous Integration *"Continuous integration (CI) is the practice of automating the integration of code changes from multiple contributors into a single software project. The CI process is comprised of automatic tools that assert the new code's correctness before integration. A source code version control system is the crux of the CI process. The version control system is also supplemented with other* *checks like automated code quality tests, syntax style review tools, and more."* Definition retrived on `https://www.atlassian.com/continuous-delivery/continuous-integration`: Accessed 72020. Atlassian is a major developer of CI-related technology.. 11, 12

Acronyms

ADAS	Advanced Driver Assistance Systems. 1
ApCom	Vehicle Communication Service. 35, 36
API	Application Programming Interface. 15, 18, 19, 23–28, 32, 36, 48, 51, 52, 55, 57, 59
ARXML	AUTOSAR XML. 11, 24–27, 51, 52, 59
ASIL	Automotive Safety Integrity Level. 9, 10
BSW	Basic Software. 8, 10, 23, 59
CIL	C Intermediate Language. 29
ECU	Electronic Control Unit. 7, 8, 10, 11, 35
LCS	Life Cycle Service. 35
LTL	Linear Temporal Logic. 18, 19, 57
OEM	Original Equipement Manufacturer. 7, 11
OS	Operating System. 23
RTE	Runtime Environment. 8, 10, 23–27, 35, 36, 48, 51, 52, 55, 57, 59
SV-COMP	Competition on Software Verification. 3, 15, 17–19, 32, 54, 57
SW-C	Application Software Component. 8, 9, 22–24, 35, 36, 38, 43, 47, 51–53
TACAS	*Tools and Algorithms for the Construction and Analysis of Systems*. 15
WCET	Worst Case Execution Time. 22

Bibliography

[AUT15] *Complex Driver design and integration guideline*, r4.2.2 edition, 2015. Accessed on 2nd July, 2020 at: `https://www.autosar.org/fileadmin/user_upload/standards/classic/4-2/AUTOSAR_EXP_CDDDesignAndIntegrationGuideline.pdf`.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *International conference on tools and algorithms for the construction and analysis of systems*, pages 193–207. Springer, 1999.

[BDW15] Dirk Beyer, Matthias Dangl, and Philipp Wendler. Boosting k-induction with continuously-refined invariants. In *International Conference on Computer Aided Verification*, pages 622–640. Springer, 2015.

[Ben12] E Bendersky. PyCParser–C parser and AST generator written in python, 2012.

[Bey16] Dirk Beyer. Reliable and reproducible competition results with benchexec and witnesses (report on sv-comp 2016). In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 887–904. Springer, 2016.

[Bey20a] Dirk Beyer. Advances in automatic software verification: SV-COMP 2020. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 347–367. Springer, 2020.

[Bey20b] Dirk Beyer. SV-COMP rules, 2020. Accessed on 20th June, 2020 at `https://sv-comp.sosy-lab.org/2020/rules.php`.

[BFB$^+$05] S Behnel, M Faassen, I Bicking, H Joukl, S Sapin, MA Parent, O Grisel, K Buchcik, F Wagner, E Kroymann, et al. lxml: XML and HTML with Python, 2005. Accessed on 1st July, 2020 at `https://lxml.de/`.

[BFM$^+$08] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. Acsl: Ansi c specification language, 2008.

[BHT07] Dirk Beyer, Thomas A Henzinger, and Grégory Théoduloz. Configurable software verification: Concretizing the convergence of model checking and program analysis. In *International Conference on Computer Aided Verification*, pages 504–518. Springer, 2007.

[BK11] Dirk Beyer and M Erkan Keremoglu. Cpachecker: A tool for configurable software verification. In *International Conference on Computer Aided Verification*, pages 184–190. Springer, 2011.

[BKÁ+18] Philipp Berger, Joost-Pieter Katoen, Erika Ábrahám, Md Tawhid Bin Waez, and Thomas Rambow. Verifying auto-generated c code from simulink. In *International Symposium on Formal Methods*, pages 312–328. Springer, 2018.

[BL18] Dirk Beyer and Thomas Lemberger. Cpa-symexec: efficient symbolic execution in cpachecker. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 900–903, 2018.

[BLW19] Dirk Beyer, Stefan Löwe, and Philipp Wendler. Reliable benchmarking: Requirements and solutions. *International Journal on Software Tools for Technology Transfer*, 21(1):1–29, 2019.

[BMGS20] Robert Bramberger, Helmut Martin, Barbara Gallina, and Christoph Schmittner. Co-engineering of safety and security life cycles for engineering of automotive systems. *ACM SIGAda Ada Letters*, 39(2):41–48, 2020.

[BP20] Armin Biere and David Parker. *Tools and Algorithms for the Construction and Analysis of Systems: 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25–30, 2020, Proceedings, Part I.* Springer Nature, 2020.

[Bun11] Stefan Bunzel. Autosar–the standardized software architecture. *Informatik-Spektrum*, 34(1):79–83, 2011. Accessed on 25th June, 2020 at: `https://link.springer.com/content/pdf/10.1007/s00287-010-0506-7.pdf`.

[CCF+05] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The astrée analyzer. In *European Symposium on Programming*, pages 21–30. Springer, 2005.

[CD11] Cristiano Calcagno and Dino Distefano. Infer: An automatic program verifier for memory safety of c programs. In *NASA Formal Methods Symposium*, pages 459–465. Springer, 2011.

[CDD+15] Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving fast with software verification. In *NASA Formal Methods Symposium*, pages 3–11. Springer, 2015.

[CGD+16] Stephen Chong, Joshua Guttman, Anupam Datta, Andrew Myers, Benjamin Pierce, Patrick Schaumont, Tim Sherwood, and Nickolai Zeldovich. Report on the nsf workshop on formal methods for security. *arXiv preprint arXiv:1608.00678*, 2016.

[CKK+12] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c. In *International conference on software engineering and formal methods*, pages 233–247. Springer, 2012.

[CKL04] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[CKNZ11] Edmund M Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. Model checking and the state explosion problem. In *LASER Summer School on Software Engineering*, pages 1–30. Springer, 2011.

[CWZ+19] Zhiqiang Cai, Aohui Wang, Wenkai Zhang, M Gruffke, and H Schweppe. 0-days & mitigations: Roadways to exploit and secure connected bmw cars. *Black Hat USA*, 2019:39, 2019.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[DMRS03] Leonardo De Moura, Harald Rueß, and Maria Sorea. Bounded model checking and induction: From refutation to verification. In *International Conference on Computer Aided Verification*, pages 14–26. Springer, 2003.

[DS07] David Delmas and Jean Souyris. Astrée: from research to industry. In *International Static Analysis Symposium*, pages 437–451. Springer, 2007.

[FKDO12] Ling Fang, Takashi Kitamura, Thi Bich Ngoc Do, and Hitoshi Ohsaki. Formal model-based test for autosar multicore rtos. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 251–259. IEEE, 2012.

[FLR17] Kathleen Fisher, John Launchbury, and Raymond Richards. The hacms program: using formal methods to eliminate exploitable bugs. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 375(2104):20150401, 2017.

[H+77] Maurice Howard Halstead et al. *Elements of software science*, volume 7. Elsevier New York, 1977.

[HCD$^+$13] Matthias Heizmann, Jürgen Christ, Daniel Dietsch, Evren Ermis, Jochen Hoenicke, Markus Lindenmann, Alexander Nutz, Christian Schilling, and Andreas Podelski. Ultimate automizer with smtinterpol. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 641–643. Springer, 2013.

[HCD$^+$18] Matthias Heizmann, Yu-Fang Chen, Daniel Dietsch, Marius Greitschus, Jochen Hoenicke, Yong Li, Alexander Nutz, Betim Musa, Christian Schilling, Tanja Schindler, et al. Ultimate automizer and the search for perfect interpolants. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 447–451. Springer, 2018.

[IEE17] IEEE, Institute of Electrical and Electronics Engineers, Inc. Staff, CORPORATE. IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(R)) Base Specifications, Issue 7, IEEE Std 1003.1-2017, 2017.

[ISO99] ISO/IEC 9899: 1999 Programming Languages-C. Standard, International Organization for Standardization, Geneva, CH, March 1999.

[ISO18a] ISO 26262-6: Road vehicles – Functional safety. Part 6 - Product development at the software level. Standard, International Organization for Standardization, Geneva, CH, March 2018.

[ISO18b] ISO 26262: Road vehicles – Functional safety. Standard, International Organization for Standardization, Geneva, CH, March 2018.

[ITF$^+$14] Omar Inverso, Ermenegildo Tomasco, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. Bounded model checking of multi-threaded C programs via lazy sequentialization. In *International Conference on Computer Aided Verification*, pages 585–602. Springer, 2014.

[JM09] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Computing Surveys (CSUR)*, 41(4):1–54, 2009.

[KB03] Hermann Kopetz and Günther Bauer. The Time-Triggered Architecture. *Proceedings of the IEEE*, 91(1):112–126, 2003.

[KC16] Dongwoo Kim and Yunja Choi. Light-weight api-call safety checking for automotive control software using constraint patterns. In *2016 6th International Conference on IT Convergence and Security (ICITCS)*, pages 1–5. IEEE, 2016.

[KT14] Daniel Kroening and Michael Tautschnig. CBMC–C bounded model checker. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 389–391. Springer, 2014.

[Löw13] Stefan Löwe. Cpachecker with explicit-value analysis based on cegar and interpolation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 610–612. Springer, 2013.

[LW12] Stefan Löwe and Philipp Wendler. Cpachecker with adjustable predicate analysis. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 528–530. Springer, 2012.

[MAK14] Georg Macher, Eric Armengaud, and Christian Kreiner. Automated generation of AUTOSAR description file for safety-critical software architectures. *Informatik 2014*, 2014.

[McC76] Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976.

[McC85] Edward J McCluskey. Built-in self-test techniques. *IEEE Design & Test of Computers*, 2(2):21–28, 1985.

[MCNF15] Jeremy Morse, Lucas Cordeiro, Denis Nicole, and Bernd Fischer. Model checking ltl properties over ansi-c programs with bounded traces. *Software & Systems Modeling*, 14(1):65–81, 2015.

[Mit18] Roland Mittag. Entwicklung Statischer Analysen für AUTOSAR Steuergerätesoftware. Master's thesis, Technische Universität Chemnitz, 2018.

[MTD16] Pasquale Malacaria, Michael Tautchning, and Dino DiStefano. Information leakage analysis of complex c code and its application to openssl. In *International Symposium on Leveraging Applications of Formal Methods*, pages 909–925. Springer, 2016.

[NMRW02] George C Necula, Scott McPeak, Shree P Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *International Conference on Compiler Construction*, pages 213–228. Springer, 2002.

[NŠST18] Stefan Niæetin, Robert Šandor, Goran Stupar, and Nikola Tesliæ. Maximizing the efficiency of automotive software development environment using open source technologies. In *2018 IEEE 8th International Conference on Consumer Electronics-Berlin (ICCE-Berlin)*, pages 1–3. IEEE, 2018.

[R4.15a] Layered software architecture. Standard, AUTOSAR, 2015. Accessed on 1st July, 2020 at: `https://www.autosar.org/fileadmin/user_upload/standards/classic/4-2/AUTOSAR_EXP_LayeredSoftwareArchitecture.pdf`.

[R4.15b] Overview of functional safety measures in AUTOSAR. Standard, AUTOSAR, 2015. Accessed on 1st July, 2020 at: `https:`

//www.autosar.org/fileadmin/user_upload/standards/classic/4-2/AUTOSAR_EXP_FunctionalSafetyMeasures.pdf.

[R4.15c] Specification of RTE. Standard, AUTOSAR, 2015. Accessed on 1st July, 2020 at: https://www.autosar.org/fileadmin/user_upload/standards/classic/4-2/AUTOSAR_SWS_RTE.pdf.

[RICB17] Herbert Rocha, Hussama Ismail, Lucas Cordeiro, and Raimundo Barreto. Model checking embedded c software using k-induction and invariants. In *Embedded Software Verification and Debugging*, pages 159–182. Springer, 2017.

[Ron] Armin Ronacher. Jinja2 documentation. *Welcome to Jinja2—Jinja2 Documentation (2.8-dev)*. Accessed on 1st July, 2020 at http://mitsuhiko.pocoo.org/jinja2docs/Jinja2.pdf.

[RRI+17] Williame Rocha, Herbert Rocha, Hussama Ismail, Lucas Cordeiro, and Bernd Fischer. Depthk: A k-induction verifier based on invariant inference for c programs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 360–364. Springer, 2017.

[Rup10] Nayan B Ruparelia. Software development lifecycle models. *ACM SIGSOFT Software Engineering Notes*, 35(3):8–13, 2010.

[SE05] Niklas Sorensson and Niklas Een. Minisat v1. 13-a sat solver with conflict-clause minimization. *SAT*, 2005(53):1–2, 2005.

[Sea14] Robert C Seacord. *The CERT C coding standard: 98 rules for developing safe, reliable, and secure systems.* Pearson Education, 2014.

[SGM18] Christoph Schmittner, Gerhard Griessnig, and Zhendong Ma. Status of the development of iso/sae 21434. In *European Conference on Software Process Improvement*, pages 504–513. Springer, 2018.

[SKB+17] Peter Schrammel, Daniel Kroening, Martin Brain, Ruben Martins, Tino Teige, and Tom Bienmüller. Incremental bounded model checking for embedded software. *Formal Aspects of Computing*, 29(5):911–931, 2017.

[Wal95] Stephen R Walli. The POSIX family of standards. *StandardView*, 3(1):11–17, 1995.

[Wes19] Lukas Westhofen. Verifying automotive c code using modern software model checkers. Master's thesis, Rheinische-Westfälische Technische Hochschule Aachen, 2019.

[ZSO+17] Fiorella Zampetti, Simone Scalabrino, Rocco Oliveto, Gerardo Canfora, and Massimiliano Di Penta. How open source projects use static code analysis tools in continuous integration pipelines. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 334–344. IEEE, 2017.