Automotive embedded software design using formal methods

Vassil Todorov

université
PARIS-SACLAY

Automotive embedded software design using formal methods

Thèse de doctorat de l'Université Paris-Saclay

École doctorale n° 580, Sciences et Technologies de l'Information et de la Communication
Spécialité de doctorat: Informatique
Unité de recherche: Université Paris-Saclay, CNRS, Laboratoire de recherche en informatique, 91405, Orsay, France
Référent: CentraleSupélec

Thèse présentée et soutenue à Gif-sur-Yvette, le 9 décembre 2020, par

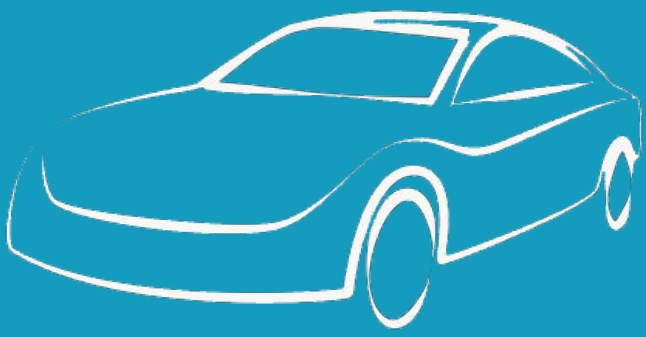
Vassil TODOROV

Composition du jury:

Pascale Le Gall Professeur, CentraleSupélec	Présidente et examinatrice
Gérard Berry Professeur émérite, Collège de France	Rapporteur et examinateur
Cesare Tinelli Professeur, University of Iowa	Rapporteur et examinateur
Sylvie Putot Professeur, Ecole Polytechnique	Examinatrice
Fabrice Kordon Professeur, Sorbonne Université	Examinateur
Sylvain Conchon Professeur, Université Paris-Saclay	Examinateur
Frédéric Boulanger Professeur, CentraleSupélec	Directeur de thèse
Safouan Taha Maître de conférences, CentraleSupélec	Co-encadrant de thèse

Thèse de doctorat
NNT: 2020UPASG026

Automotive Embedded Software Design Using Formal Methods



```
(assert (not (=> (and (= FlipFlopReset~0.L3$0 (>= Timer~0.L5$0 2400))
      $PropDecelerationHigh$0 (>= AtLeastNTicks~0.L5$0 0)
      (>= Timer~0.L5$0 AtLeastNTicks~0.L5$0)
      (=> FlipFlopReset~0.L3$0 true)
      (=> false FlipFlopReset~0.L3$0))
      (and (= FlipFlopReset~0.L3$1 (>= Timer~0.L5$1 2400))
      $PropDecelerationHigh$1 (>= AtLeastNTicks~0.L5$1 0)
      (>= Timer~0.L5$1 AtLeastNTicks~0.L5$1)
      (=> FlipFlopReset~0.L3$1 true)
      (=> false FlipFlopReset~0.L3$1)))))
(check-sat)
(echo "@DONE")
; Yices2: sat
; Yices2: @DONE
(get-model)
(echo "@DONE")
```

Vassil Todorov

AUTOMOTIVE EMBEDDED SOFTWARE DESIGN USING FORMAL METHODS

DOCTORAL THESIS

Vassil Todorov

A thesis submitted in fulfillment of the requirements
for the degree of

Doctor of Philosophy in Computer Science

Committee in charge

Pr. Gérard Berry (Collège de France)	Reviewer
Pr. Cesare Tinelli (University of Iowa, USA)	Reviewer
Pr. Sylvie Putot (Ecole Polytechnique, France)	Examiner
Pr. Pascale Le Gall (CentraleSupélec, France)	Examiner
Pr. Fabrice Kordon (Sorbonne Université, France)	Examiner
Pr. Sylvain Conchon (Université Paris-Saclay, France)	Examiner
Pr. Frédéric Boulanger (CentraleSupélec, France)	Thesis advisor
Ass. Pr. Safouan Taha (CentraleSupélec, France)	Thesis co-advisor
Armando Hernandez (Groupe PSA, France)	Industrial tutor

December 2020

This document was typeset using LaTeX and inspired from the typographical style `classicthesis` by André Miede and Ivo Pletikosić, and the nice book style adaptation by Hai Nguyen Van.

Dedicated to the loving memory of my dear aunt
Prof. Magdalina Todorova
1954 – 2019

ABSTRACT

The growing share of driver assistance functions, their criticality, as well as the prospect of certification of these functions, make their verification and validation necessary with a level of requirement that testing alone cannot ensure.

For several years now, other industries such as aeronautics and railways have been subject to equivalent contexts. To respond to certain constraints, they have locally implemented formal methods. We are interested in the motivations and criteria that led to the use of formal methods in these industries in order to transpose them to automotive scenarios and identify the potential scope of application.

In this thesis, we present our case studies and propose methodologies for the use of formal methods by non-expert engineers. Inductive model checking for a model-driven development process, abstract interpretation to demonstrate the absence of run-time errors in the code and deductive proof for critical library functions.

Finally, we propose new algorithms to solve the problems identified during our experiments. These are, firstly, an invariant generator and a method using the semantics of data to process properties involving long-running timers in an efficient way, and secondly, an efficient algorithm to measure the coverage of the model by the properties using mutation techniques.

RÉSUMÉ

La part croissante des fonctions d'assistance à la conduite, leur criticité, ainsi que la perspective d'une certification de ces fonctions, rendent nécessaire leur vérification et leur validation avec un niveau d'exigence que le test seul ne peut assurer.

Depuis quelques années déjà d'autres domaines comme l'aéronautique ou le ferroviaire sont soumis à des contextes équivalents. Pour répondre à certaines contraintes ils ont localement mis en place des méthodes formelles. Nous nous intéressons aux motivations et aux critères qui ont conduit à l'utilisation des méthodes formelles dans ces domaines afin de les transposer sur des scénarios automobiles et identifier le périmètre potentiel d'application.

Dans cette thèse, nous présentons nos études de cas et proposons des méthodologies pour l'usage de méthodes formelles par des ingénieurs non-experts. Le model checking inductif pour un processus de développement utilisant des modèles, l'interprétation abstraite pour démontrer l'absence d'erreurs d'exécution du code et la preuve déductive pour des cas de fonctions critiques de librairie.

Enfin, nous proposons de nouveaux algorithmes pour résoudre les problèmes identifiés lors de nos expérimentations. Il s'agit d'une part d'un générateur d'invariants et d'une méthode utilisant la sémantique des données pour traiter efficacement des propriétés comportant du temps long, et d'autre part d'un algorithme efficace pour mesurer la couverture du modèle par les propriétés en utilisant des techniques de mutation.

Резюме

Нарастващият дял на функциите за помощ на водача, тяхната критичност, както и перспективите за сертифицирането им, правят тяхната проверка и валидиране на ниво, на което само тестването не е достатъчно.

От няколко години насам други области, като аеронавтиката и железопътният транспорт, се намират в подобен контекст. Ние се интересуваме от мотивациите и критериите, довели до използването на формални методи в тези области и как те биха могли да се приложат в автомобилна среда.

В тази теза представяме нашите изследвания и предлагаме методологии за тяхното използване от инженери, които не са специалисти по формални методи. Индуктивна проверка на модели за процес на разработка използващ модели, статичен анализ базиран на абстрактна интерпретация, за да се демонстрира липсата на грешки при изпълнение на код и дедуктивен анализ за критични библиотечни функции.

А накрая, предлагаме нови алгоритми за решаване на проблемите, идентифицирани по време на нашите експерименти. Става въпрос от една страна за генератор на инварианти и метод, използващ семантиката на данните за ефективна обработка на свойства, включващи времеви таймери, и от друга страна за ефективен алгоритъм за измерване на покритието на модел по свойства, използвайки мутационни техники.

CONTENTS

1

INTRODUCTION

> *Computing is fundamentally invisible. When your tires are flat, you look at your tires, they are flat. When your software is broken, you look at your software, you see nothing.*
>
> — Gérard Berry

A few years ago, the idea of cars driving themselves on our streets seemed unbelievable. However, the rapid advances in machine learning in recent years make us think that one day it could become reality. Today, trained machine learning algorithms are capable of driving cars in most of the common situations. Most of the time, the decisions taken by those algorithms are good but sometimes they can be wrong, which could cause fatalities. The problem is that when they are wrong, nobody can explain why and fix the problem. We can just train them more and hope/pray that next time there will be fewer errors. Furthermore, machine learning algorithms could not be certified in the sense of a critical system because there is no specification against which an implementation could be verified. As autonomous vehicles can be considered safety-critical systems similar to trains and airplanes, it is expected that authorities require their certification in the future.

In order to ensure the safety of the system and provide safety arguments for the certification authority, we cannot use only machine learning algorithms. The decisions taken by the machine learning algorithms should be supervised by more classical algorithms based on domain expert knowledge. These non-machine learning supervision algorithms will be the safety-critical part of the software and could then be certified.

1.1 THE CAR – A SOFTWARE-DRIVEN ELECTRONIC DEVICE

In the 20^{th} century automobile, the engine was the core technology. Two main periods can be distinguished for the engine control: mechanical control before the 70-80s and electronic control after. The electronic fuel control injection system offered a better fuel dosage and reduced the fuel consumption compared to the previous carburetor-based mechanical injection system.

In the 21^{st} century automobile, we observe a transition from a hardware-driven machine to a software-driven electronic device. Today, software, large

computing power, and advanced sensors enable most modern innovations, from efficiency to connectivity to electrification to autonomous driving and new mobility solutions.

However, as the importance of electronics and software has grown, so has complexity. As portrayed in Figure 1, the complexity of software-based automotive functions is increasing rapidly.

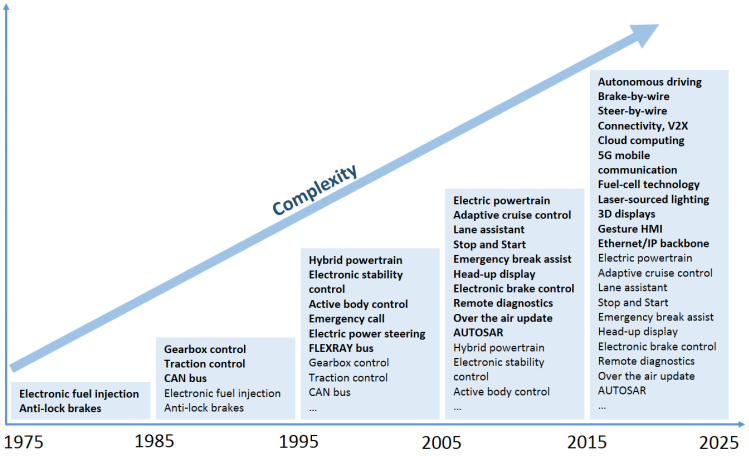


Figure 1: Automotive innovations and complexity

To some extent, we can observe a link between the complexity and the number of software Lines Of Code (LOC) contained in a modern car. In 2010, some vehicles had about 10 million LOC; by 2020, this expanded by a factor of 10, to roughly 100 million lines.

1.2 PROBLEM

The effort to verify these systems increases with the amount of software to verify. Nowadays, test scenarios and peer reviews are used during the verification process but it is practically impossible to test all the combinations of possible states in which the system can be put. And in practice, it is not necessary to make such an effort for all the embedded software because only a little part can be considered as critical. For example, a radio that reboots during driving cannot harm the passengers of the car but if the front lights go off during night driving it can cause fatalities. Therefore, it is important to have the highest robustness and reliability for those functions that are safety-related and that have a potential impact on human lives or can cause economic losses.

Model-based design has been introduced to cope with the complexity issue and with the cost of fixing bugs that are found during the late phases of the design. By allowing the simulation of a model of the software in the early design phases, Model Based Design allows fixing the model as soon as possible

and limiting the propagation of errors down to the implementation. However, similar to testing, simulation suffers from the same limitations: the number of scenarios to be checked in a model to get full confidence goes beyond any reasonable subset of scenarios that can be examined in practice and cannot guarantee the absence of errors for highly critical systems.

A way to bring higher guarantees for safety-critical software systems is to use formal verification techniques. Formal methods are techniques based on mathematical logic, which goal is to bring in the software and hardware verification the same rigorous mathematical background already used by other engineering disciplines. Formal methods can be seen as exhaustive testing but need some methodologies and tools in order to be used in practice by non-expert software engineers.

1.3 RESEARCH OBJECTIVES

The work presented in this thesis focuses on the introduction of formal methods in an automotive industrial context in order to bring more guarantees and robustness for the safety-critical part of the embedded software. Indeed, there is a gap to bridge between the formal methods capabilities and the systems and specifications present in the industry. In practice, verification of arbitrary functional properties on realistic systems often requires expert knowledge about the analyzed system and verification technique(s) used. The spread of formal verification in the industrial community is hindered by this need for costly and time-consuming expert intervention. Our work contributes to the understanding of how formal methods can be used by the engineers and for what type of scenario.

The goal of this thesis is also to improve on the state of the art of formal verification in our application domain by:

- studying the use of formal methods in other industries (railway, aviation) and identify which techniques are used and at which design stage;
- experimenting with different formal methods on representative automotive models and code in order to get an insight on their limitations and eventually find workarounds;
- proposing methodologies that could be used by non-expert engineers and use cases on which they could be applied.

1.4 CONTRIBUTIONS

To achieve these goals, we studied in the literature the industrial use of existing formal methods – *abstract interpretation*, *model checking*, *deductive proof* – and then applied them to some automotive use cases.

We proposed a new algorithm for invariant generation for symbolic model checking of properties involving time. It provides useful relational invariants based on a template that strengthen the inductive proof.

We also proposed a mutation framework to measure the coverage of the proved properties, which is important when using formal methods in a certification context. In this context, a demonstration of the coverage of the proof when using formal methods is generally required.

Finally, our experiments served to propose some methodologies that can be used by serious (motivated) non-experts software engineers. These methodologies go from providing contracts for the inputs and outputs of the designed functions to how to describe in a formal way what the function is expected to do, where to find good requirements that can be formalized, what are the strategies to begin with and how to structure the proved function and its formal requirements.

1.5 PLAN

We divided this thesis as follows:

- Chapter 1: introduces the PhD work and states the challenges we tackle.
- Chapter 2: presents the industrial context in which our work evolves.
- **Part 1: Formal methods and Certification standards** presents the automotive context
  - Chapter 3: presents an overview of the safety standards.
  - Chapter 4: presents the formal methods from the theoretical foundations to the applications that can be found in the industry today.
- **Part 2: Automotive embedded software design using formal methods** presents our contributions
  - Chapter 5: presents some methodologies that can be applied by engineers and some hints for writing formal properties.
  - Chapter 6: presents our new improved invariant generation algorithm for long duration properties and its integration in JKind.
  - Chapter 7: proposes an efficient algorithm and a new coverage metrics for evaluating the quality of properties that are proved valid using model checking.
  - Chapter 8: presents our experiments with automatic deductive proof of a discrete-valued function calculating a square root.
- Chapter 9: concludes this thesis and outlines future research directions.

2

AUTOMOTIVE SOFTWARE DESIGN AND DEVELOPMENT

> *There is not a vehicle currently available to US consumers that is self-driving. Every vehicle sold to US consumers still requires the driver to be actively engaged in the driving task, even when advanced driver assistance systems are activated. If you are selling a car with an advanced driver assistance system, you're not selling a self-driving car. If you are driving a car with an advanced driver assistance system, you don't own a self-driving car.*
>
> — Robert Sumwalt (2020, Feb. 25), Chairman of National Transportation Safety Board

The modern car has evolved from a mechanical device to a distributed cyber-physical system, which relies on software to function correctly. Starting from the 1970s, the amount of software and electronics used in a car has gradually increased from as little as one Electronic Control Unit (ECU) to as much as 150 in 2020.

The main reason to introduce electronics in the 1970s was the 1973 oil crisis followed by another oil crisis in 1979. We needed to reduce the fuel consumption, and electronic engine control helped in doing so. Another reason was the introduction of safety-related functions such as Anti-lock Braking System (ABS) in the early 1970s preventing the wheels from locking up during braking. But the part of embedded software significantly increased since 2000 with the introduction of driver assistance functions. At the beginning, it was simply for Cruise Control and then it was coupled with a radar to have an Adaptive Cruise Control adjusting automatically the car speed to the front vehicle speed. Today the radar can be coupled with a camera and control the braking in case of emergency, keep the lane in traffic jams or on highway, thus taking control over the steering wheel. Since 2010, we observe the evolution of new advanced connected services and infotainment systems based on the Linux kernel, which have significantly contributed to the increase of the number of lines of code in the car, achieving 100 million lines of code.

The software in a modern car provides plenty of new opportunities, but it also requires to be more careful in the design, implementation, verification and validation phases. Although the practices in software engineering include

methods and tools to respect safety and security requirements of the software, they are often applied in an automotive-specific manner.

Today we can clearly see that the automotive industry is moving to a field less dominated by mechanical engineering but with a growing part of electronic and software engineering. The trend of using software will continue to increase and there is a need for professional software engineering. To maintain a higher quality level of the software, we need to have *rigorous processes* of software engineering, providing guarantees that the software will not be harmful to its users.

One of the good practices in software engineering is the high-level design of software systems, referred to as *software architecture*. The software architecture offers the possibility to define how the software functions are distributed on software components, what interface mechanisms they use and how components interact with each other.

The other important phase in the software design is its verification. This phase is of a greater importance, in particular when developing highly critical safety systems. Testing is the general practice but we can also use formal methods for some parts of the software and obtain higher confidence that bugs are absent.

In the following, we describe each phase of the automotive software development process and discuss it from a verification point of view.

2.1 THE V-MODEL

The *V-Model* shown in Figure 2 is the mostly used methodology today for developing embedded software in the automotive industry. However, new methodologies brought from the software engineering (agile/scrum) start to break through.

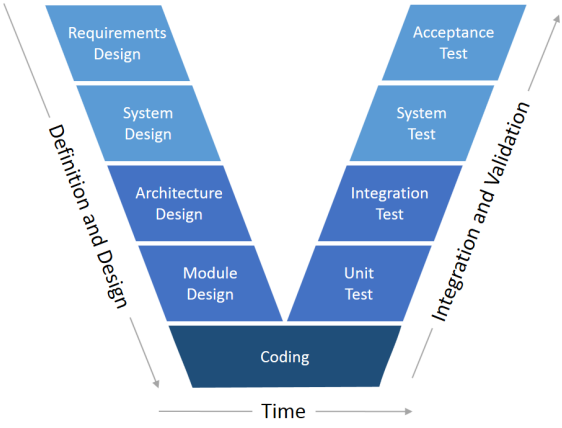


Figure 2: The V-Model illustrated

The V-Model is a full life cycle methodology that emphasizes the importance of testing. Actually, tests are designed in parallel with each phase of the life cycle. This early design of test cases with each development phase can be an effective way for early defect detection and removal. This method can be considered an extension of the *waterfall model*.

ISO 26262, the standard for functional safety of road vehicles, proposes also to use the V-Model as a reference phase model for the product development at the software level [ISO18].

The V-Model is suitable for developments in which requirements are well defined and stable, and the technology to be used is well understood. This is especially necessary in a certification context for safety-critical software. The safety assessor needs to have a stable documentation of the project to be able to analyze it within a given time.

2.2 REQUIREMENTS ENGINEERING

As we saw in the previous section, the first and very important phase required by the V-Model and the ISO 26262 standard is to completely elicit and document all the *requirements* necessary for the project prior to the actual development.

Definition 1 (*Requirement*)

(1) *A condition or capability needed by a user to solve a problem or achieve an objective.*

(2) *A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed documents.*

(3) *A documented representation of a condition or capability as in (1) or (2).*

[IEEE 610.12-1990]

This work is done by the system engineer who captures the needs from different stakeholders (marketing, regulations, etc.) and writes down textual requirements of what the system is expected to do. Each requirement has a unique number and a version and is stored in a database. If a requirement should be changed, its version is incremented.

2.2.1 *Requirement Types*

In the literature [Poh10] and many requirement engineering standards, three main types of requirements are generally presented :

- Functional requirements
- Quality requirements
- Constraints

Functional requirements specify the functionalities/services that the system should or should not provide. Different levels of functional requirements can exist depending on the level of details. Requirements defining the data, functions and behavior of the system are in most cases solution-oriented since they are defined in a way that mainly supports the realization of the system. They describe *how* the system should be developed. An example for such a requirement can be the following statement: *"When LightSwitch = ManualON and Ignition = ON Then LightsOutput = ON Else LightsOutput = OFF."*. In contrast, *user requirements* are more abstract and give the goals of the system in a solution-neutral way. They describe the problem or *what* the system is expected to do. An example for such a requirement can be the following statement: *"If the brake ECU detects a fault, the system shall inform the user by showing an alert message."*.

Quality requirements specify quality properties such as performance, reliability, stability for entire system or for a component, service or function. Very often, they influence the architecture of the system. An example for such a requirement can be the following statement: *"When LightsOutput changes its value the application module should emit the new value on the network in less than 50 ms."*.

In practice, the term *"non-functional requirements"* is sometimes used but very often it refers to either quality requirements or underspecified functional requirements. In the reference book about Requirements engineering [Poh10], the author recommends avoiding the category of "non-functional requirements" when writing specifications. He proposes to classify them as quality requirements or refine them to functional ones.

Besides defining functional and quality requirements for a system, *constraints* are also documented during the requirements design phase. They are organizational or technological requirements that restrict the development process or the properties of the system to be developed. They have different origins: cultural, legal, physical, project and so on. An example for such a requirement can be the following statement: *"The source code should be Motor Industry Software Reliability Association (MISRA) compliant."*.

2.3 SOFTWARE ARCHITECTURE

Software architecture is fundamental for the automotive software design. It is a high-level design view of the system and combines multiple views used to communicate with the project teams. It is also used to make technical decisions about the organization of the functionalities of the system. The software architecture can help understanding and predicting the performance of the system before it is designed.

2.3.1 *AUTOSAR*

AUTomotive Open System ARchitecture (AUTOSAR) is a global partnership between automotive actors founded in 2003. Its objective was to create and establish an open and standardized software architecture for automotive ECUs. It defines the reference *architecture and methodology* for the development of automotive software systems and provides the language (meta-model) for their architectural models.

Today, AUTOSAR proposes two main platforms: the *Classic Platform*[1] and the *Adaptive Platform*[2]. To give an example, in Figure 3, we show the *Classic Platform*, which contains three software layers:

- an application layer called Application Software (ASW);
- a middleware software represented by the Runtime Environment (RTE);
- a Basic Software (BSW) including the real-time operating system and services.

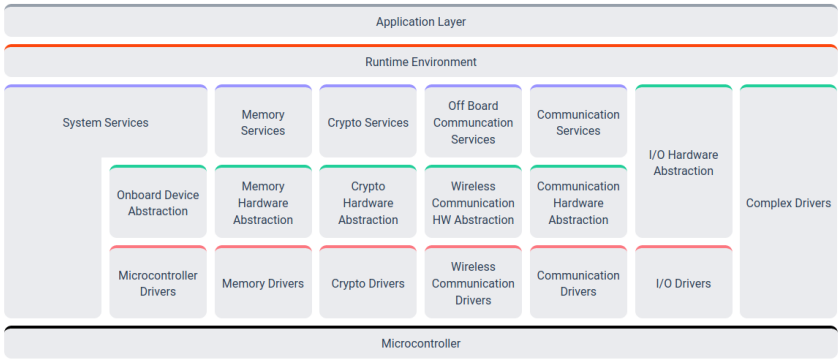


Figure 3: AUTOSAR Classic Platform Release R19-11

The application layer is composed of hardware-independent software and contains application modules (components) that are visible and used by the final user. It is in this layer that formal methods are most valuable to be deployed because these modules are updated more often than the operating system and the low-level services (BSW). Another reason is that most of the modules are model-based and thus can support model checking.

The RTE is automatically generated during compilation phase. It is used for managing communications between application software components and for communicating with the BSW services. The BSW provides different services that are specified by the standard and the operating system. As the services are standardized by AUTOSAR and the operating system shared between multiple car manufacturers, we can consider their validation done by the practice.

1 AUTOSAR Classic Platform: https://www.autosar.org/standards/classic-platform
2 AUTOSAR Adaptive Platform: https://www.autosar.org/standards/adaptive-platform

2.3.2 *AUTOSAR and Software Verification*

From the perspective of this thesis, the AUTOSAR standard is important in the sense that it formalizes some important elements about the system that can be used for software verification, even if not using mathematical logic. We can use them for example to extract semantic information about the software such as specific types declaration for physical dimensions (speed, acceleration, time). They can be used to facilitate the proof of some properties combining objects of same type together to generate more useful invariants. We discuss this topic in Chapter 6.

As in the AUTOSAR definitions we can find the ranges for the inputs, outputs, internal variables, other parameters, we can automatically extract this information to help the deductive proof of a function or the static analysis based on abstract interpretation. For the deductive proof, we could provide contracts in the form of pre-/post- conditions that normally take a lot of time to be written manually. For the sound static analysis, we could provide assertions to obtain more precision reducing thus the amount of false alarms. This could be the subject of a future work.

2.4 MODEL-BASED DESIGN VS MANUAL CODING

Traditionally, the entire software of the car was written manually based on the low-level detailed software requirements. Assembly language was used at the beginning but it was progressively replaced by higher-level languages such as C or C++.

Today, the trend is to replace C programming by higher-level domain specific languages for some part of the control software. The principle is to make a model of the needed functional behavior by simply dragging and dropping library blocks and linking them. The obtained *specification model* is then simulated by the designer to check if its behavior corresponds to the specification. Once the specification model is validated there are two ways to embed it in the ECU. The oldest one was to use the model as software specification and write manually the code corresponding to the model. This method could be error-prone and time consuming. That is why the preferred method today is to transform the specification model into a *design model* that can be used to generate the code for the target ECU automatically. The design model is optimized to fit in the ECU's memory and resources and integrates the AUTOSAR routines and other system libraries not present in the specification model.

2.4.1 *Traditional Manual Coding*

After the system requirements have been written and allocated to an ECU, a software architecture is defined for it and the design of the software modules

can begin. This process consists of refining the system specification to have a new detailed specification ready for implementation by the software developer. This specification looks like pseudocode and uses the names of the variables that will be present in the final code. However, it can abstracts some complex services or routines that the developer is aware of. For example, it can require calculating a square root with some precision without providing technical details how it should be calculated. The software developer uses C or C++ languages to code this detailed specification. The module is then compiled and unit tests are run to verify its behavior. Coverage tests are also run to check the coverage of the code, and static analysis is used to check the MISRA compliance and the presence of run-time errors. The code is then reviewed by another person that was not involved in the coding. After its approval, it is integrated, compiled with the target compiler and tested again as a black-box. The integration is a complex process in which a set of application modules that are compatible with each other are assembled with the basic software (real-time operating system, network services, memory services, etc.), parameter values chosen for the concrete project, and all the AUTOSAR glue is generated and assembled with the rest. Because of the generic nature of the software (it can goes on multiple vehicle projects), there can be a few thousand parameters to be configured.

The choice of the programming language depends on the problem to be treated: *C* is commonly used because it provides an optimized code necessary for the embedded micro-controllers that have limited resources. *Assembly* language is still used for some low-level functions of the operating system, even in modern multi-core real-time operating systems. *C++* is more suitable for data-oriented functions where a large amount of data need to be processed, such as Advanced Driver-Assistance Systems (ADAS)[3] functions.

2.4.2 *Model-Based Design*

Model-Based Design (MBD), as the name supposes, uses models for the design of software. The model should reflect the behavior of the function of a car and is created in a formalism that reflects the physical world rather than the software world. Different kinds of models may exist: Physical process model, Environment model, Prototyping model, System model, Implementation model, and so on.

Designing using models has impacts on the design process and the competence of the designers, who should be trained before using a modeling tool. A typical process is shown in Figure 4.

The process starts with the description of the function of a car as mathematical equations defined by the inputs, the outputs, and a focus on the internal data flow between them. At this stage, the designer often operates by using

3 ADAS are electronic systems that assist drivers in driving and parking functions.

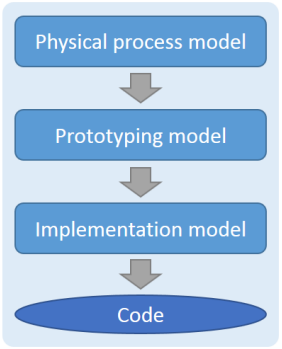


Figure 4: Model-based design process

mathematical models and ignores some practical issues such as the modular arithmetic used by computers, the precision of the floating-point numbers, the limited size of the memory, etc. The difference between reasoning with mathematical numbers and the implemented code can result in divergence of the final behavior from the one that was calculated with the mathematical model. After the physical process is modeled, simulated and validated, two other models are developed to prepare and produce the code: a prototyping model and an implementation model. The prototyping model is used to verify the behavior of the entire function within its environment. Generally, this model is not optimized to run on a micro-controller. The implementation model is an optimization (CPU usage, memory consumption) and adaptation of the prototyping model that integrates AUTOSAR routines. This adaptation is error-prone: overflows and different behaviors can appear and should be verified, using for example the same simulation scenarios that were used during prototyping.

Once the implementation model is completed and validated, it is used to generate the code in the target programming language – usually C.

There are two types of code generators: *unqualified* (example: Embedded Coder for Simulink) and *qualified* (example: SCADE Suite Compilator (KCG) for SCADE Suite). The difference is that for an unqualified code generator the code should be reviewed and additional testing called *back-to-back testing* should be done to demonstrate that the model and the code have the same behavior. Back-to-back tests are tests that have a good coverage ratio and such that their results are the same for the model and for the generated code. With a qualified code generator, the behaviors of the model and the code are guaranteed by the tool certification to be the same. In this case, a generated code review and a back-to-back testing are not necessary.

2.5 TOWARDS THE AUTONOMOUS CAR

Research projects in innovation stimulated by public authorities around the world aim to clear the way for a step-by-step introduction of automated vehi-

cles. The projects comprise a number of aspects like standardization, testing, safety, or in-vehicle technology. As mentioned in [noa15] it is expected that automated driving will:

- Improve safety by reducing human driving errors
- Significantly contribute to the optimization of traffic flow
- Help to reduce fuel consumption and CO_2 emissions
- Enhance the mobility of elderly people and unconfident drivers

Several forecasts [WH16] predict a limited availability of automated driving functions at level 2 and 3 (partial and conditional automation) in 2020 and a wide availability by 2040 until level 4. Today's ADAS such as Adaptive Cruise Control (ACC), Lane Keeping Assist (LKA), or Pedestrian Detection (PD) will form the backbone of tomorrow's mobility. Vehicles will communicate with each other and with the infrastructure. Vehicle-to-Vehicle (V2V) communication will allow vehicles to exchange traffic data information (e.g. nearby accidents) and data about their driving intention. Vehicle-to-Infrastructure (V2I) communication will be used to optimize the road traffic and thereby will help to reduce pollution. The role allocation between a human driver and an automated driving system in this scenario is specified by the Society of Automotive Engineers (SAE) as *six levels of driving automation* going from no automation to full automation (see Table 1).

<table>
<tr><th>SAE Level</th><th>Name</th><th colspan="2">Narrative definition</th></tr>
<tr><td colspan="4">Human driver monitors the driving environment</td></tr>
<tr><td>0</td><td>No Automation</td><td colspan="2">The full-time performance by the human driver of all aspects of the dynamic driving task, even when "enhanced by warning or intervention systems"</td></tr>
<tr><td>1</td><td>Driver Assistance</td><td>The driving mode-specific execution by a driver assistance system of "either steering or acceleration/deceleration"</td><td rowspan="2">using information about the driving environment and with the expectation that the human driver performs all remaining aspects of the dynamic driving task</td></tr>
<tr><td>2</td><td>Partial Automation</td><td>The driving mode-specific execution by one or more driver assistance systems of both steering and acceleration/deceleration</td></tr>
<tr><td colspan="4">Automated driving system monitors the driving environment</td></tr>
<tr><td>3</td><td>Conditional Automation</td><td rowspan="3">The driving mode-specific performance by an automated driving system of all aspects of the dynamic driving task</td><td>with the expectation that the human driver will respond appropriately to a request to intervene</td></tr>
<tr><td>4</td><td>High Automation</td><td>even if a human driver does not respond appropriately to a request to intervene</td></tr>
<tr><td>5</td><td>Full Automation</td><td>under all roadway and environmental conditions that can be managed by a human driver</td></tr>
</table>

Table 1: Levels of driving automation for on-road vehicles according to SAE J3016

There is a key distinction between level 2 ("Partial Automation") and level 3 ("Conditional Automation") as in the latter case the system performs the entire dynamic driving task (execution of steering, acceleration, braking and monitoring of the environment). In contrast to level 4 ("High Automation"), in level 3 the driver is expected to be ready for taking over the control upon demand (within a predefined time period e.g. 10 seconds) at all times. At level 4, the driver is no more asked to take over the control.

We present in Table 2 a brief overview of driver assistance systems that have already been introduced or systems that are on the way to be introduced in the market. The majority of the systems today are level 1 or 2. Levels 3 and 4 are still under development.

Level of automation	**Current and future vehicle automation systems and functions**	**Market introduction**
0	Lane change assist (LCA)	Available
0	Lane departure warning (LDW)	Available
0	Front collision warning (FCW)	Available
0	Park distance control (PDC)	Available
1	Adaptive cruise control (ACC)	Available
1	Park assist (PA)	Available
1	Lane keeping assist (LKA)	Available
2	Park assistance	Available
2	Traffic jam assist	Available
3	Traffic jam chauffeur/pilot	2020+
3	Motorway chauffeur (MWC)	2020+
3	Highway pilot	2020+
4	Piloted parking	2020+
5	Robot taxi (fully automated private vehicle)	2030+

Table 2: Summary of current and future vehicle automation systems and functions

Figure 5 illustrates the Groupe PSA's "Autonomous Vehicle for All" program roll-out. The first level 3 functions to be proposed by the group will be Traffic Jam Chauffeur (TJC) and Highway Chauffeur (HC) respectively for automated driving in traffic jams and on highways.

There is an important liability gap between levels $\leqslant 2$ and levels $\geqslant 3$. Actually, for levels up to level 2 the driver is responsible for the entire driving, the assistance functions only assist him. Functions at levels $\geqslant 3$ can take the control over the driving for some period of time. If there is an accident during this period, the car manufacturer's liability can be engaged for production defects: manufacturing defects, design defects, incorrect or missing instructions.

2.6 PROVING THE SAFETY OF THE AUTONOMOUS VEHICLE

To help proving that the critical software of an autonomous vehicle is safe, we propose in this thesis to introduce formal methods in some parts of the

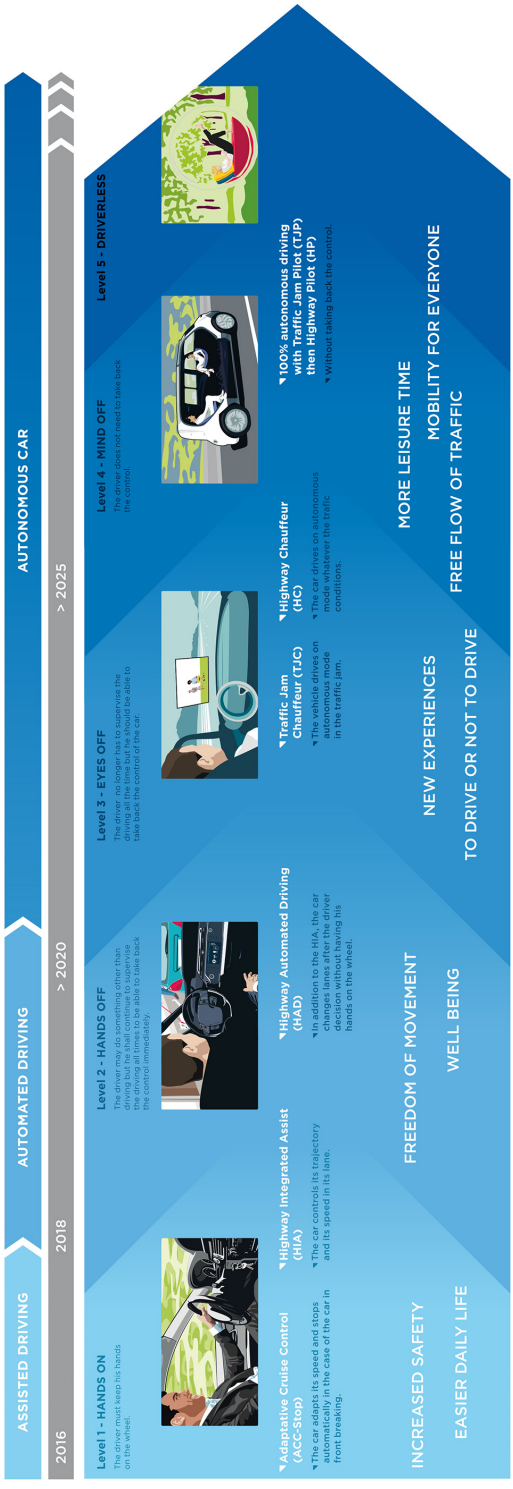


Figure 5: Groupe PSA's "Autonomous Vehicle for All" program roll-out

development process. We expect that if autonomous cars are to be massively launched in the future, the authorities could require their certification, as is actually the case for the railway and aviation industries. Probably, the machine learning algorithms could not be certified because they have no specification to be verified that they respect. Internal experiments at Groupe PSA have shown that machine learning algorithms are not 100% reliable. To guarantee the safety of the passengers, they should be supervised by simpler algorithms developed in a classical manner (based on requirements). Supervising consists in checking that the results (the outputs) of the executed complex function correspond to what is defined by the specification and expected by the supervising function. Otherwise, the supervisor takes over and put the system into a safe mode. For an autonomous vehicle, this safe mode could be to stop in the emergency lane on the highway. The critical part of the software is then the supervisor, which could be certified.

The supervisor is not a new concept. It already exists in the trains and can for example decide to activate the emergency braking when the autopilot is not reacting well. In this case, the supervisor has the highest critical level (SIL 4) and is certified at that level.

The general principle of a supervisor and a supervised modules is shown in Figure 6. It consists of a supervised function, for which a basic Quality Management (QM) is sufficient. It means that this function has no Automotive Safety Integrity Level (ASIL)[4] level and is not considered critical. This function may be too complex to be verified, for instance containing complex algorithms or even neural networks. In order to control that its critical outputs obey the specification, we use a supervisor module, which is critical (ASIL A to D) and is developed using traditional verification and certification methods.

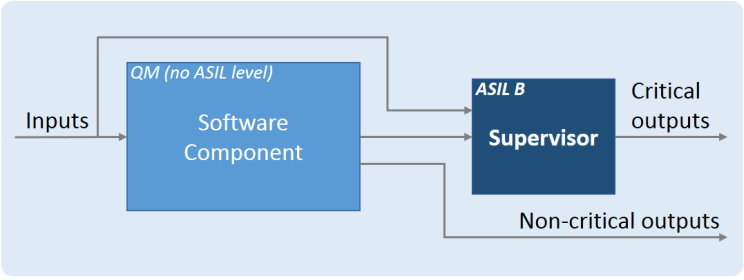


Figure 6: Example of a supervisor monitoring a software component

Today, there are already multiple supervisors in the automotive software. For example, if there is a problem with the lighting control software, a supervisor can take over and turn on the lights. The different strategies and mechanisms used at Groupe PSA, such as supervision and partitioning for coexistence of critical and non-critical modules, are discussed in [MBF18].

4 ASIL is a risk classification scheme defined by the ISO 26262.

One of our targets for the formal methods application are the supervision modules. They are intended to be simple and they are based on clearly defined safety requirements. We want to obtain a proof that no matter what happens, the safety requirements expressed as safety properties will be respected by the software without exceptions. A safety property is generally invariant and should always be true.

The other potential targets for the application of formal methods in the automotive software are the source code analysis based on abstract interpretation to prove the absence of run-time errors or estimate the Worst-Case Execution Time (WCET) and the unitary function proof based on deductive methods.

2.7 CONCLUSIONS

This chapter presented the context in which the automotive embedded software is developed in respect to the safety standard ISO 26262. We introduced the V-Model, based on the central notions of specification and requirement. New methodologies such as Agile are being currently introduced. They need to be adapted if a certification of the software were required in the future.

We briefly presented AUTOSAR, which defines the reference architecture and methodology for the development of automotive software. AUTOSAR is a great source of inspiration for research because it offers a quantity of data that can be used to verify the software in different ways.

We presented the MBD paradigm, which is more and more used for designing software components in replacement of the traditional manual coding.

We presented the way towards the autonomous car. Today most of the driver's assistance systems are level 2 but in the next years level 3 will probably appear and become the standard.

We presented how a complex software module can be supervised. These supervisors are of a particular interest for the application of formal methods because they are simpler than the supervised module and are based on a safety specification, which can be formalized and proved.

In the next chapter, we present an overview of the safety standards to understand what could be necessary to prepare for the autonomous vehicles tomorrow.

Part I

FORMAL METHODS AND CERTIFICATION STANDARDS

3

SAFETY STANDARDS AND CERTIFICATION

> *All too often, writers of standards focus on questions of what constitutes good practice, and lose sight of what the followers of those standards truly need to demonstrate in order to show safety. Safety is demonstrated not by compliance with prescribed processes, but by assessing hazards, mitigating those hazards, and showing that the residual risk is acceptable.*
>
> — A. Rae (2007). Acceptable Residual Risk: Principles, Philosophy and Practicalities. *2nd IET International Conference on System Safety*.

The correctness and the quality of the embedded software play a key role in the safety as a whole. To ensure safety, engineers make a significant effort to show the correctness of individual subsystems and their integration using testing or simulation. They eventually need to certify that the developed vehicle as a whole is indeed safe using artifacts and evidences produced during the development process. Such a certification process can help to increase the safety confidence in the product and to reduce the automaker's liability.

However, software certification in the automotive domain is not yet a common practice compared to other safety-critical domains, such as aviation, railway, nuclear and medical [YLK16]. Even if the safety standard ISO 26262 has been well adopted, the authorities do not require for the time being to certify the car's software. It is probably because the driver is still responsible for the driving task but with the advent of higher levels of automation, things could change.

3.1 SAFETY STANDARDS

3.1.1 *Why do we Need Standards?*

In some areas, we can easily imagine the usefulness of the standards. For example, the electronic devices that we buy have a standard electric plug (for a country or a continent) and we can plug it to a standard electrical socket. The size of the car's tires and some other pieces are standardized in order to replace them easily.

For safety-critical software, it is less obvious why we need a standard. Actually, it comes from the disasters people have experienced. A disaster occurs and people ask that it never occurs again. It results in a standard written by some experts to which the industry must comply. It generally results in increased product quality and provides a better protection for the users.

Standards can also be useful to new companies for their product development. They can find some good practices, techniques and tools listed in the standard. In the event of a court case, if a company complies with a standard it can defend itself by claiming they are following the industry best practices.

Another point of view is that of a company that wants to buy a safety-critical system for which the standard helps formalizing the contract. It is easier to require that the product must comply with ISO 26262 at ASIL B level than to write the explicit conditions of acceptable failure rate, tools to be used for the development process, and so on.

3.1.2 *Goal- and Prescription-Based Standards*

The standards can be classified into two categories: *prescriptive* and *goal-based* [Hob15].

Prescriptive standards prescribe and proscribe means: processes, procedures, techniques and tools. If a certificate were issued for a prescriptive standard, a company could for example claim that their product meets the requirements of ISO 26262 standard.

Goal-based standards, in contrast, require some goals to be achieved and leave the selection of appropriate processes, techniques and tools to the development organization. This type of standard needs more effort to be done by the development team because they need to not only define their processes but also justify their adequacy. The auditing for certification is also much more difficult because there is no checklist of prescriptions to be checked. However, if a certificate were issued to a company, it could in principle claim that their product is safe. The avionics standard, DO-178C, is probably the most goal-based standard that exists today.

3.1.3 *Functional Safety and IEC 61508 Derivated Standards*

Functional safety can be described as the part of the safety of a system that depends on automatic protection operating correctly in response to its inputs or failure in a predictable manner. For example, if for some reasons (software bug, component failure, etc.) the car's lights switched off when they should be on (driver has requested them, or in automatic mode when it is dark) a safety function that permanently monitors the conditions can decide to turn them on to protect the driver.

IEC 61508 is a basic and maybe the most important standard for functional safety from which are derived some other standards for different industries. The standard is entitled *Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems* and covers general project management and the development of hardware and software components.

The standard considers that a piece of equipment can give rise to safety hazards and it should be accompanied by a safety function that monitors it and moves it into a safe state when a hazardous situation is detected.

From a software development point of view, the most interesting part is the prescription of tools and techniques that are not recommended, recommended or highly recommended for each stage of the software development life cycle.

IEC 61508 has been specialized for a number of industries as shown in Figure 7. The linkage level between IEC 61508 and the industry-specific standards varies between industries. The railway standards (EN 5012x), are probably the closest to IEC 61508. The medical standard IEC 62304 is not considered a derivation from IEC 61508 because it is not directly concerned by functional safety. The aviation standard DO-178C is another example of standard that is not derived from IEC 61508.

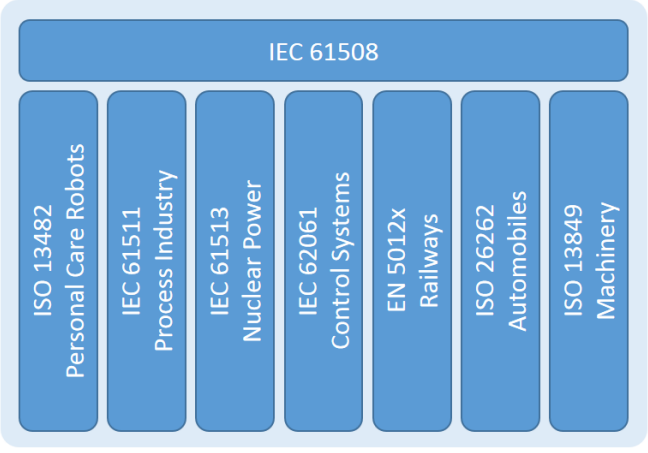


Figure 7: Derivation of safety standards from IEC 61508

3.1.4 *Railway – IEC 62279 / EN 5012x*

IEC 62279 provides a specific interpretation of IEC 61508 for railway applications. It reuses from IEC 61508 the Safety Integrity Level (SIL) levels (SIL 0 to 4, SIL 4 is the most critical) and is intended to cover the development of software for railway systems. On the other hand, CENELEC emitted a set of EN standards for railway applications such as EN 50126 (system), 50128 (software) and 50129 (hardware). In EN 50128, the requirements for SIL 3 are the same as for

SIL 4, and the requirements for SIL 1 are the same as for SIL 2. So in practice, only three levels are used: SIL 0, SIL 1/2 and SIL 3/4. We can note that in SIL 3 and SIL 4 formal methods are highly recommended.

3.1.5 *Medical – IEC 62304*

IEC 62304 is a standard, which specifies life cycle requirements for the development of software within medical devices. It defines three classes of risk: Class A when no injury or damage to the health of a patient or device operator would result if the software failed; Class B if non-serious injury is possible; Class C if death or serious injury is possible (pacemakers, defibrillators, etc.). IEC 62304 focuses exclusively on processes (software development, management and configuration) and its only reference to formal methods is somewhat negative. Paragraph 5.2.6 states, "This standard does not require the use of a formal specification language."

3.1.6 *Aviation – DO-178C*

DO-178C/ED-12C, called "Software Considerations in Airborne Systems and Equipment Certification" provides guidance to the development of safety-critical avionics software and is used for its certification. It was jointly developed by Radio Technical Commission for Aeronautics (RTCA) and European Organisation for Civil Aviation Equipment (EUROCAE) and defines five different critical levels for the software named Design Assurance Level (DAL): Catastrophic (A), Hazardous (B), Major (C), Minor (D); No effect (E). They are determined from the safety assessment process and hazard analysis by examining the effects of a failure condition in the system.

DO-178C is particular in that it does not provide prescriptions on tools and techniques to be used. Instead, it provides objectives that should be achieved, which is more difficult but the system is considered safer compared to prescription-based standards.

DO-178C also provides the following supplements for tool qualification and technologies:

- DO-330 "Software Tool Qualification Considerations";
- DO-331 "Model-Based Development and Verification Supplement to DO-178C and DO-278A";
- DO-332 "Object-Oriented Technology and Related Techniques Supplement to DO-178C and DO-278A";
- DO-333 "Formal Methods Supplement to DO-178C and DO-278A".

We are particularly interested in DO-333, which provides guidelines how formal methods can be applied in a certification context and which activities they can replace.

3.1.7 *Automotive – ISO 26262*

ISO 26262 is the specialization of IEC 61508 for the automotive domain. Part 6 deals with software and includes tables of recommended (+) and highly-recommended techniques (++) for different phases of the development process and for each Automotive Safety Integrity Level – ASIL (from level A to D, D is the most critical). ASIL replaces the concept of IEC 61508's SIL which is based on probability of failure per hour of use. Instead, in ISO 26262 the events that could cause injury are listed and an ASIL is calculated (using a table proposed in the standard) for each of them by considering three aspects:

- *Severity* (S0 to S3): classifies the type of injuries that could result from this event;
- *Probability of Exposure* (E0 to E4): probability that the event occurs in normal operation;
- *Controllability* (C0 to C3): classifies the difficulty of the driver to control the situation and avoid injury.

For example, a hazardous event combining S3 (fatal injury), E4 (high probability) and C3 (uncontrollable) is assigned to ASIL D level. For many positions, there is no ASIL level assigned – "QM" is listed instead. It means that there is no requirement to comply with ISO 26262 but the product must be developed in accordance with a quality management process approved in an international standard such as ISO 16949 or ISO 9001.

In [YLK16], Toyota prepares the advent of the certification by providing a survey of the software certification approaches in different domains (aviation, medical, railway). They propose to use assurance cases as a promising technique to automotive software certification. The authors also point out that even if ISO 26262 provides some recommendations to improve the software quality, it does not provide a quantified and rigorous process for software certification.

3.2 CERTIFICATION AND QUALIFICATION

The quality of the safety-critical embedded software has been, for many years now, ensured by applying precise and constraining development processes on each activity of the system development: from requirements elicitation to integration of the embedded code through software design, code production and code verification. For each of these activities artifacts are produced aiming at

providing evidence of the means used for the activity achievement. These artifacts are mandatory to provide parts of the information required by the certification bodies.

Certification stands for the process of ensuring that a product (system or equipment) is "approved" for use. When a company wants/needs to certify a product against a particular standard, it can self-declare that the product meets all the requirements of the appropriate standard, possibly asking an external auditing company to confirm the declaration. Alternatively, the company could employ an independent certification body to assess and certify the product.

In order to be able to emit certificates, the certification body must be accredited to issue certificates for a particular standard. Certification bodies are accredited by national accreditation authorities to carry out particular certifications. There is no accreditor for the accreditation authorities – they verify each other's compliance in continuous evaluations.

Qualification is used in the context of tools when these ones are expected to be used for the "elimination, reduction or automation" of certification activities "without its [the tool] output being verified" [DO-178C]. Each standard defines different levels of qualification e.g. from TQL-1 (the strongest) to TQL-5 (TQL stands for Tool Qualification Level) in DO-178C or TCL-3 (the strongest) to TCL-1 (TCL stands for Tool Confidence Level) in ISO 26262. The approach of qualification is also different between avionics and automotive standards. For avionics, according to the chosen TQL, a specific set of objectives have to be fulfilled to ensure the tool qualification. For the automotive, it is based on the confidence of the tools and their error detection capabilities (ISO 26262, part 8). For example, a code generator (can introduce bugs when generating a code from a model) is permitted to be used without qualification (TCL-1) if it is accompanied by qualified error detection tools.

3.3 CONCLUSIONS

In this chapter, we introduced some safety standards, their usefulness and notions about certification and qualification. Some of the standards are more rigorous because they require achieving objectives instead of giving prescriptions to follow. Some of them are also proposing to use formal methods, which is a good motivation for their application in the industry.

In the next chapter, we present the formal methods from their theoretical foundations to their concrete applications in the industry.

4

FORMAL METHODS – FROM THEORY TO PRACTICE

Formal methods should be part of the education of every computer scientist and software engineer, just as the appropriate branch of applied mathematics is a necessary part of the education of all other engineers.

— John Rushby, SRI International

Engineering, as a whole, relies on mathematical models to take decisions about the design. However, software development has traditionally been less formal by using testing to confirm its goodness. The goal of formal methods is to bring the same rigorous mathematical background already used by other engineering disciplines to both software and hardware. Formal methods apply mathematical logic and discrete mathematics to model the behavior of a system and formally verify that the model of the system satisfies the properties that it is expected to have. Generally, the expected properties are written as a set of requirements called a specification. The properties can be *functional* or *safety*. It is practically impossible to verify all the functional properties because a simple embedded computer can have more than twenty thousand requirements allocated. Thus during the design of the system specification, the important properties should be identified. They are not necessarily only safety-related but can also be security-related or even effects seen by the user that could reduce its confidence in the system e.g. a radio-navigation that is rebooting frequently is not safety-related problem but the user could doubt about the quality of the rest of the system. Finally, formal verification can be seen as exhaustive testing and yielding systems of a very high dependability in a cost-effective manner.

Basically, a formal method is a formal model and a formal analysis. Even if formal models can benefit the software development process e.g. for automatic code generation, the most beneficial aspect from our point of view is the formal analysis of the formal models. Formal analysis is used to prove or guarantee that the software complies with the requirements specification. During the analysis phase, bugs can be found not only in the software but also in the specification. Because the specification is often written in natural language, it can contain ambiguities or contradictions. Formalizing the specification is a way to review it and fix its discrepancies.

In this chapter, we propose a brief introduction into different types of formal methods that seemed to us promising from industrial point of view for software verification. Hardware verification is also possible but it is not in the focus of this work. Finally, we illustrate the industrial application of some formal methods and the reasons why they were adopted.

4.1 FORMAL METHODS AND TOOLS – A BRIEF INTRODUCTION

The first work on formal methods dates back to 1949 [Tur49]. The central idea is to *guarantee the behavior* of a computing system using mathematical methods.

For many years, formal methods have essentially been used in academia. Today, formal methods get more and more connected with applied engineering and many industries are using them in their development process. They are better applied in the electronic hardware design because the hardware cannot be updated piecewise. Software tools and technologies (such as requirements and design methodology) are still changing rapidly.

We propose here a short presentation of the three main categories of formal methods to introduce the principles and terminology. We also present industrial and academic tools for each category. A complete survey of tools and techniques for formal verification can be found in [DKW08]. Another report on the trends in formal verification of autonomous robots software presents the research done for tools such as FIACRE and RT BIP [Ing19]. These reports are general and present multiple tools and methods but we will stay more focused on tools that could be used by the automotive industry for embedded software verification.

4.1.1 *Abstract Interpretation*

Abstract Interpretation is a formal method introduced by Cousot et al. [CC77] in 1977. It automatically computes information about the program behavior without executing it. Most questions about this behavior are either undecidable or it is practically impossible to compute an answer. The main goal of the abstract interpretation is to efficiently compute an approximate/abstract representation of the program and bring *sound* guarantees about dynamic properties. *Sound* means that the approximation calculated is reliable and bugs are not missed (no *false negatives*). A spurious alarm also known as a *false positive* is a warning about a bug that does not exist in the program. Sometimes false positives can be generated due to the approximation precision. A detailed introduction of the formal verification by abstract interpretation is given in [CC10].

4.1.1.1 *Purpose of Static Analysis based on Abstract Interpretation*

Static analysis based on abstract interpretation can be used to prove that no timing or space constraints are violated, no numeric precision problems are

present when using floating-point numbers or that run-time errors are absent: the absence of errors is guaranteed for the entire program as the method is based on mathematical proof. These are non-functional properties and they are required by most of the safety standards.

A demonstration that the software respects timing (WCET) and space constraints is required for example by the aviation safety standard (DO-178C). Even if a safety standard do not require this demonstration, it is useful for the software architect to know if the timing is correct (all real-time tasks meet their deadlines) or if the stack size is large enough to keep the data in all possible execution paths. Without using abstract interpretation, WCET tends to be over-approximated because modern processors possess caches and pipelines, which significantly reduces the execution time. When using abstract interpretation for timing and stack size worst-case estimation, the results are more precise and a less powerful and cheaper micro-controller could be chosen. Moreover, for real-time applications, WCET is more important to know than the average-case performance for which the processors are optimized.

The numeric precision problem is presented in [DGP+09]. It consists of computing value intervals that the variables of the program can take at each control point, for all possible executions, with two different semantics:

- semantics corresponding to executions of the program on an ideal machine that handles real mathematical numbers;
- semantics corresponding to the implementation of floating-point numbers and modular integers with finite precision, by means of IEEE-754.

An upper bound for the difference between values taken by variables for these two semantics is then computed, and breaks it down according to the contributions of the different control points. This enables the user to determine the main sources of numerical imprecision in his or her program.

A run-time error is:

- any bad memory manipulation including out-of-bounds array access, NULL or invalid pointer dereference
- any arithmetic error and undefined behavior in the calculus including divisions by zero, modular arithmetic overflow or floating-point arithmetic Not a Numer (NaN)
- any usage of the programming language that is being defined in the standard as undefined behavior (e.g. section J2 in C99 standard);
- any violation of an assertion provided by the user (e.g. check of the output values ranges or other program invariants).

4.1.1.2 *The core of the abstract interpretation: abstract domains*

The core of the abstract interpretation are the different methods of abstraction, which are called *domains*. The difference between the domains is the precision of the results that can be obtained for different use cases. To illustrate one of these abstractions, let us take the following example:

$$-8.5871 \times -45.6587 \times 2.5678$$

Even though the calculation of the solution to the problem takes time if it is done by hand, it is possible to apply the rules of the multiplication sign to determine that the result is always positive (*invariant*). The determination of the sign of this calculation is an example of *abstraction*. With this technique, we can verify a set of properties of a final result, such as the sign, without having to do the whole calculation. This property of the program can be useful if, for example, the result of this calculation must be used as a parameter of the function calculating the square root of a number.

Over the years, other domains were developed and extended to tools using abstract interpretation. They are *non-relational* such as the *intervals domain* (recording the minimum and maximum value for each variable) or *relational* such as the *octagons domain* (can find invariant relations between two variables, e.g. a < b). More details about this method and its abstractions can be found in [CC10].

4.1.1.3 *Using Partitioning to Increase Precision*

When the behavior of a program strongly depends on the precise values of certain expressions, such as test conditions or array indexes, the use of abstract domains can produce results that are too imprecise to prove the absence of run-time errors. In this case, it is recommended to proceed by case analyses, by conducting several local analyses of the same part of the program to join results later in the program, where the problematic expressions have no longer an important role. This case analysis is called partitioning [BCC+03, MR05]. We take an example from [Bou11] to show that without partitioning Astrée (a static analysis tool based on abstract interpretation) cannot prove the absence of division by zero in the code shown in Figure 8.

```
1  typedef enum {FALSE = 0, TRUE = 1} Boolean;
2  volatile Boolean B;
3  void main () {
4  unsigned int X, Y;
5  if (B) {X = 0; Y = 1;}
6  else {X = 1; Y = 0;};
7  X = 1 / (X + Y);
8  }
```

Figure 8: Example of code where Astrée cannot prove the absence of division by 0

Actually, the other abstract domains return the invariants $X \in \{0,1\}$ and $Y \in \{0,1\}$ before the division, but none automatically captures the (implicit) relation $X + Y = 1$. To increase the precision locally, the user can add two Astrée directives before the `if` and after the final calculation of X to perform two separate analyses of the division:

- one for the case where $B = \mathrm{TRUE}$ and therefore $X = 1/(0+1)$;
- one for the case where $B = \mathrm{FALSE}$ and therefore $X = 1/(1+0)$.

With this partitioning, after the division we obtain the invariant X = 1 that can also be useful for other parts of the program.

4.1.1.4 *Difference between Sound and Unsound Static Analyzers*

The tools that are using abstract interpretation to analyze programs are called *sound static analyzers*. They can be opposed to another category of static analyzers called *unsound*. They differ by the analysis depth and exhaustiveness. Sound static analyzers guarantee to find all bugs even if they can bring some false alarms. If there are no alarms, it means that the code is free from the class of bugs the tool can find. At the opposite, *unsound static analyzers* use heuristics to find some common bugs but they are not exhaustive and cannot guarantee the absence of bugs. However, using at first step an unsound static analyzer to fix the bugs it founds and then as a second step, using a sound static analyzer can reduce the effort of analyzing the alarms produced by the sound static analyzer. In general, the code should be as clean as possible before using sound static analysis. For example, if we have a constantly increasing value such as *a*++ in an infinite loop's body it should be protected from overflowing. Otherwise, the sound static analyzer will show an overflow alarm.

4.1.1.5 *ISO 26262 Recommends Abstract Interpretation*

The latest version of the ISO 26262 standard published in 2018 added a special item for the static analysis based on abstract interpretation (see Table 3 and Table 4). *Sound static analysis* is now *recommended* (noted by +) for all critical ASIL levels for software unit verification and for verification of software integration.

4.1.2 *Model Checking*

Model checking is a formal method introduced in the early 1980s by two teams: Edmund M. Clarke, E. Allen Emerson [CE82], and Jean-Pierre Queille, Joseph Sifakis [QS82]. E. M. Clarke, E. Allen Emerson and J. Sifakis won the 2007 Turing Award, frequently referred to as the 'Nobel Prize' of computing for their work on model checking. This method was proposed for verifying finite state concurrent systems. The advantage of this restriction (finite state systems) is that the verification could be done automatically by performing an

Methods	ASIL			
	A	B	C	D
Walk-through	++	+	o	o
Pair-programming	+	+	+	+
Inspection	+	++	++	++
Semi-formal verification	+	+	++	++
Formal verification	o	o	+	+
Control flow analysis	+	+	++	++
Data flow analysis	+	+	++	++
Static code analysis	++	++	++	++
Static analyses based on abstract interpretation	+	+	+	+
Requirements-based test	++	++	++	++
Interface test	++	++	++	++
Fault injection test	+	+	+	++
Resource usage evaluation	+	+	+	++
Back-to-back comparison test between model and code	+	+	++	++

Table 3: Methods for software unit verification (ISO 26262 – Table 7)

Methods	ASIL			
	A	B	C	D
Requirements-based test	++	++	++	++
Interface test	++	++	++	++
Fault injection test	+	+	++	++
Resource usage evaluation	++	++	++	++
Back-to-back comparison test between model and code	+	+	++	++
Verification of the control flow and data flow	+	+	++	++
Static code analysis	++	++	++	++
Static analyses based on abstract interpretation	+	+	+	+

Table 4: Methods for verification of software integration (ISO 26262 – Table 10)

exhaustive search of the state space of the system (*explicit state model checking*) to determine if some specification is true or not. Given sufficient resources, the procedure would always terminate with *yes* or *no* answer. When the answer is *no* a *counterexample* is produced to show an execution trace leading to the error state. A counterexample is generally represented by a sequence of values for the input variables of the model and can be analyzed or simulated by the user to understand the violation. This is the most valuable feature of the model checking for the system design engineers.

As hardware controllers and communication protocols are finite state systems, model checking was first adopted to verify their correctness.

The difficulty in software verification compared to the hardware, is that instead of using bits, the software variables are at least represented as bytes (8, 16, 32 bits are common in practice), which for industrial models resulted in a state space explosion [DLS06]. Furthermore, for some systems it was necessary to reason with mathematical real numbers, which resulted in infinite state

systems. To cope with such complexities, model checking was combined with various *abstraction*, *induction* and *invariant generation* principles.

4.1.2.1 *The Process of Model Checking*

Applying model checking consists of several tasks:

- **Modeling**: consists of converting a design into a formal language accepted by the model checking tool (also called model checker). Sometimes, the modeling may need to abstract some irrelevant details or parts of the design that cannot be taken into account by the model checker e.g. nonlinear arithmetic calculations.
- **Specification**: consists of defining what properties the design must satisfy. The properties are usually defined using *temporal logic* [Pnu77] or *synchronous observers* [HLR93]. The difficulty in this point is to choose relevant properties. As reported in [CM14b], writing good formal properties shares many similarities with writing good requirements and is as much art as science. This report also mentions that properties that cut across an entire system often find the most errors and that the best sources of formal properties are found in the safety-related requirements for the system. We comment on the methodological aspect of choosing properties in Chapter 5.
- **Verification**: once we have a model and properties, ideally the verification can be done automatically. However, in practice it often needs human assistance for example for analyzing the results. In case of a negative result, the user can use the provided trace (counterexample) to analyze the origin of the problem: incorrect modeling or incorrect specification. Sometimes the verification task can fail to terminate due to the size of the model, which is too large to fit into the computer memory. In all of these cases after fixing a problem or changing the parameters of the model checker, the verification should be redone.

4.1.2.2 *Symbolic Model Checking*

In the original implementation of model checking algorithm, transition relations were represented explicitly by adjacency lists (explicit state model checking). It was limited only for small systems. To verify larger systems, McMillan proposed in 1987 to use a symbolic representation [BCM+89, McM93]. It was based on the Bryant's Ordered Binary Decision Diagrams (OBDD) [Bry86]. OBDD provided a canonical form for Boolean formulas that was more compact than conjunctive or disjunctive normal form. This representation captures some of the regularity in the state space determined by sequential circuits and protocols and it became possible to verify systems with extremely large number of states.

However, verifying software causes some problems even for implicit state model checking. Software is less structured than hardware. In addition, concurrent software is usually asynchronous, that is, most of the activities are executed independently without a global synchronizing clock. The most successful technique to cope with concurrent software was based on *partial order reduction* [GP93]. It aims to reduce the number of states to be explored by removing equivalent interleaving events. Another technique to reduce the state explosion by *symmetry* [CFJ93] was proposed for concurrent systems and protocols, which often contain replicated components.

While these symbolic techniques have greatly increased the size of the systems that can be verified, many realistic software systems are still too large to be handled. Thus it is important to associate symbolic methods with new techniques to extend the size of the systems that can be verified. Four such techniques are *compositional reasoning*, *abstraction*, *induction* and *invariant generation*.

4.1.2.3 *Compositional Reasoning*

Compositional reasoning is a technique that can be used on large models when they can be decomposed in independent components [BCC97, GNP18]. The specification is also decomposed into local properties that describe each component. If it is possible to show that the system satisfies each local property, and if the conjunction of the local properties implies the global specification, then the complete system must satisfy this specification as well. The difficulty is to determine the local properties that can infer the global specification. When this is not feasible because of mutual dependencies between the components, they should be analyzed one by one, making assumptions about the behavior of the other components. The assumptions must later be discharged when the correctness of the other components is established. This strategy is called *assume-guarantee reasoning* [Pnu89, GL94, PDH99, FQ03, GPC04, GBPG08].

This technique is crucial for the scalability of real industrial systems. A successful implementation of compositional reasoning for reactive systems verification can be found in the KIND 2 model checker [CMST16].

4.1.2.4 *Abstraction*

Abstraction is a technique that does not consider directly the model but an abstract version of it to reduce the complexity of model checking. The abstraction consists in approximating the behaviors of the model, so if this abstraction meets the required specification the original model will do. For example, we could abstract a square root calculation by providing the range of the values the result can take. The difficulty is to find the right level of abstraction.

The *Counterexample Guided Abstraction Refinement (CEGAR)* approach [CGJ+00] proposes an automatic iterative abstraction-refinement methodology.

It consists of generating an abstract model based on the analysis of the concrete model. If in the abstract model an erroneous counterexample is found, it is removed and the abstract model is refined until no more erroneous counterexamples are found.

An alternative form of abstraction-refinement comes from using the so-called *Craig interpolants* [McM05, McM06]. While the intention is similar to that of CEGAR, a different method of analysis, based on proofs of inconsistency, is performed to determine the refined formula, requiring a solver that supports interpolant generation.

It is also possible to use *abstract interpretation* combined with model checking to abstract complex models [CC99].

Slicing [Tip95] is another technique for reducing the size of the initial model based on a cutting criterion without losing information. If the criterion is property related then it is called *property-based slicing*.

The size of the model can also be reduced by removing variables that do not influence the proved property. This technique is called the *Cone Of Influence (COI) reduction* [BCC97] and is often used in modern model checkers.

Another form of abstraction is called *predicate abstraction* where a predicate is converted into a Boolean variable. For example, the predicate $a > b + c$ can be replaced by the Boolean a_b_c.

4.1.2.5 *Induction: SMT-model checking*

Before explaining how *induction* is used with model checking, it is necessary to introduce some other frameworks that induction will use. Today, induction is the most scalable technique for industrial software model checking combined with SAT or Satisfiability Modulo Theories (SMT)-based model checkers rather than OBDD-based ones. It has been observed [Bry86] that SAT/SMT solvers are often able to solve much larger formulas than classical techniques based on OBDD.

The *Boolean Satisfiability Problem* (commonly abbreviated as *SAT*) for a given propositional logic formula, consists in determining whether there exists a variable assignment such that the formula evaluates to true (for example, it will find $a = true$ and $b = true$ for the formula $a \wedge b$). Because verification of properties on software models and other practical problems can be formulated as SAT instances, it motivated the research since fifty years. The original algorithm used behind the SAT solvers is called Davis–Putnam–Logemann–Loveland (DPLL) algorithm [DLL62] and is still used today improved and combined with Conflict-Driven Clause Learning (CDCL) [MSS03].

One area of particular interest is in exploring these highly efficient satisfiability techniques to solve formulas from non-Boolean domains is known as *SMT* [Tin10]. It refers to the problem of determining whether a first-order formula is satisfiable (SAT) with respect to some logical theory. There are now several powerful and sophisticated SMT solvers providing decision procedures

for the most common theories: Alt-Ergo [CCIM18], CVC4 [BCD+11], MathSAT5 [CGSS13], SMTInterpol [CHN12], Yices2 [Dut14] and Z3 [DMB08]. An initiative called *SMT-LIB* [BST+10] was proposed to standardize the language and the theories used to request the SMT solvers. Today the SMT-LIB repository also proposes more than 100 000 benchmarks and there is an annual competition called *SMT-COMP* to compare the different SMT-solvers and stimulate their improvement.

A number of practical applications of SAT/SMT, as we will see later, involve iterative solving called *incremental SAT/SMT*. It consists of communicating minimal changes in the formula when calling the SAT/SMT solver and thus reuse the learned clauses from the previous calls without recalculating them. It could be *stack based* or *assumption based*. In a stack based incremental solving, each pushing to the stack creates a local context. Assertions added under a push are removed after a matching pop. Furthermore, any lemmas that are derived under a push are also removed. We use it to freeze a state and be able to resume to it. In an assumption based incremental solving, the learned clauses are not removed.

For some applications, SAT/SMT solvers are expected to provide *unsatisfiable cores* [ZM03]. It is a subset of the original formula used to prove unsatisfiability and can be used for example to provide a traceability between the model and the proved properties [GBW+18].

Bounded Model Checking (BMC) [BCCZ99] is a technique used to discover property violations that in practice can represent real bugs. In BMC, a model (represented by a transition system) and a property (represented by its negation) are jointly unwound for a given number of steps. The obtained formula (conjunction between the model and the property) is then passed to a SAT/SMT solver. If the formula is satisfiable, it means that the property is violated (a bug is found) and we obtain a counterexample (an execution trace) for the property up to the selected number of steps. If the formula is unsatisfiable, it means that the property is valid. For a valid property, there could be many unsatisfiable cores. Each of them presents the variables necessary for the proof.

SAT/SMT-based model checking is generally well suited to check whether a given model satisfies a given *inductive property*. The advantage of the induction is that a property can be proved valid for a system without exploring its entire state space. We use I to denote the initial state of the system and T – the transition relation.

Definition 2 (*Inductive property*)

Given a state space S and a state $s \in S$, a property P is *inductive* iff:

(1) P holds in the initial state, that is $\forall s.\, I(s) \Rightarrow P(s)$, and

(2) P holds in all states reachable from states that satisfy P, that is $\forall s, s'.\, P(s) \wedge T(s,s') \Rightarrow P(s')$

If the first condition (base case) does not hold, then it should be possible to extract a counterexample from the invalidity proof. The problem is that if P is not inductive (provable via induction), the second condition (step case) fails and nothing can be concluded. Unfortunately, this often arises when attempting to verify real systems. We need to strengthen P automatically by providing additional lemmas or invariants that are always true for the system. Thus, we can have more chances to succeed. Next, we present some techniques for strengthening the properties.

4.1.2.6 *k-Induction*

An automated technique to increase the strength of the property is to increase the depth of the unwinding and look at progressively larger windows of the system execution paths. It is called k-induction. It unrolls the property over k steps of the transition system and checks its validity. For example, 1-induction consists of the formulas proposed in Definition 2, whereas 2-induction consists of the following formulas:

Definition 3 (2-*induction*)

Base step 1: $\forall s.\, I(s) \Rightarrow P(s)$

Base step 2: $\forall s, s'.\, I(s) \wedge T(s,s') \Rightarrow P(s')$

Inductive step: $\forall s, s', s''.\, P(s) \wedge T(s,s') \wedge P(s') \wedge T(s',s'') \Rightarrow P(s'')$

There are two base step checks and one inductive step check. In general, for an arbitrary k, k-Induction consists of k base step checks and one inductive step check as in the following formulas:

Definition 4 (*k-Induction*)

Base step 1: $I(s_0) \Rightarrow P(s_0)$

$\vdots$

Base step k: $I(s_0) \wedge T(s_0,s_1) \wedge \cdots \wedge T(s_{k-2},s_{k-1}) \Rightarrow P(s_{k-1})$

Inductive step: $P(s_0) \wedge T(s_0,s_1) \wedge \cdots \wedge P(s_{k-1}) \wedge T(s_{k-1},s_k) \Rightarrow P(s_k)$

We say that a property is k-inductive if it satisfies the k-Induction constraints for the given value of k. We begin by checking for $k = 1$ and then k is increased until proving the formulas. The hope is that the additional formulas in the antecedent of the inductive step make it provable. We use a SAT/SMT solver to check the validity of the formulas.

4.1.2.7 *Property-Directed Reachability*

The previous approach for strengthening the property rely on an unwinding of the transition relation, that is the formula given to the solver consists of multiple copies of the transition relation. The resulting memory consumption can be prohibitive for large systems. *Property-Directed Reachability (PDR)* also known as *IC3* [Bra11, Bra12] is a technique that performs SAT-based reachability checking without making copies of the transition relation. PDR was also generalized for SMT solvers [CG12, HB12].

It consists in dividing the initial problem into a big number of small problems. A sequence of frames blocking the dangerous states is constructed incrementally, mixing backward analysis of the proof obligation and forward propagation taking into account the initial states. The process is repeated until the sequence reaches a fix-point or a bad state is shown to be reachable. Compared to *k*-Induction, there are much more requests sent to the solver but the formulas to be decided are much simpler.

4.1.2.8 *Invariant Generation*

Invariant generation techniques construct auxiliary inductive assertions for strengthening the property to become inductive and to be proved automatically. Invariant means that these assertions hold over all iterations of a program loop or over all reachable states of a transition system. Automatic invariant generation has been investigated since the 1970s. In [MP95, BL99, TNW+10], the authors present a number of methods for generating invariants to prove safety properties. These methods could be classified as either *top-down* or *bottom-up* [BM08].

Top-down invariant generation techniques focus on a given (non-inductive) property to be proved to calculate auxiliary assertions that can make the property inductive. One such technique is PDR.

Bottom-up invariant generation considers the system and uses it to deduce properties of it. It could be based on *abstract interpretation* [CC77] or based on a *template* also called *instantiation-based invariant discovery* [KGT11]. Abstract interpretation is used for example in KIND 2 [CMST16] to provide bounds on the state variables and to generalize counterexamples [WDD+12].

We are particularly interested in the *instantiation-based invariant discovery* because compared to the others it produces more complex invariants in a rather efficient way. Furthermore, during our experiments it showed to cope better with industrial models and properties. It is based on a somewhat *brute-force discovery scheme* by sifting through a large set of automatically generated formulas, which are instances of the same template. The approach relies on the efficiency of the SAT/SMT solvers to quickly generate counter-models. A specialized template can be used to solve different problems. As example, we successfully enriched the template of JKIND [GBW+18] to handle long-duration time properties by automatic generation of relational invariants [TTBH19a].

This is one of the contributions of this dissertation and we present it in details in Chapter 6. Another application of the templates is to identify "mode variables" in models containing state machines and generate invariants specifically for them [KGTW12]. This technique was implemented in KIND [HT08] and showed increased precision and speed. The discovery of invariants to strengthen and automate the proof is also a topic of interest for the aviation industry. In [Cha14], Adrien Champion proposed a heuristic for potential relational invariants generation directed by the proof objective. In practice, when combining invariants with k-induction, the k to obtain the proof is often very small even equal to one.

A complementary technique to reduce the number of invariants in order to show only those that were useful for the proof of a property is called *Inductive Validity Cores (IVC)* [GWGH19]. It can also be used to show the coverage that a proved property has on a model. The coverage of the proved properties is of great interest for the industry. We worked on a new method using *mutation and incremental SMT solving* providing a more detailed traceability metrics than IVC. This method is our second major contribution and we present it in details in Chapter 7.

4.1.3 *Deductive Methods*

Deductive methods use mathematical arguments to establish each property of a formal model. Proofs are normally constructed using a theorem proving tool, either automatically or in an interactive way.

4.1.3.1 *Introduction*

The foundations of the proof of logical properties on an imperative language program were put forward by C. A. R. Hoare in 1969 [Hoa69]. Based on the precise semantics of a computer program, Hoare proposed to prove certain properties by mathematical deductive reasoning, generally at the end of the program.

He introduced a notation called the *Hoare triple*, which associates a program Q, start hypotheses P, and expected output properties R:

$$P\ \{Q\}\ R$$

The logical meaning of this triple corresponds to: if P is true, then after executing program Q, R will be true if Q terminates. The calculus of Hoare's triples is, in general, undecidable.

The proving by application of Hoare's rules is an intellectual process and is not tool driven. It is up to the author of the proof to define the correct properties between each instruction of the program and to establish its demonstration by applying the different theorems. This activity is not adapted to process thousands of lines of code in an acceptable time.

An initial automation of the process of proving programs was brought by the calculation of the Weakest Precondition (WP) from Dijkstra in 1975 [Dij75]. The principle consists in automatically calculating the most general property *WP(S,P)* holding before a statement *S* such that the property *P* holds after the execution of *S*:

$$WP(S, P)\ \{S\}\ P$$

The calculus of WP is defined for each instruction. The proof process consists in calculating WP by going backward from the end of the program for which we want to prove *P*, up to the beginning. For full correctness, *S* must terminate. A detailed explanation of how WP is calculated for different instructions is given in [Gri87].

The returned predicate from the WP calculation can rapidly become rather complex. Efficient (quadratic instead of exponential) verification condition generation (including WP generation) were proposed in the following papers: [SAB08, BL05, FFS01]. In order to automate the process, all modern tools based on WP use automatic theorem provers or SMT solvers as back-end.

4.1.3.2 *Tools for Deductive Reasoning*

We can distinguish between two categories of tools for deductive reasoning: *based on contracts* (annotations representing the specification a program must implement) or *based on an abstract machine notation* (begins with an abstraction of the program and refines it until all the details are integrated).

Caveat [RSB+99, SFF04] and *Frama-C WP* [KKP+15] are tools for deductive reasoning on C programs based on contracts annotations. Although the objective of the two tools is similar (prove the correctness of C programs based on WP calculus) there exist some differences. Caveat was created for industrial use at Airbus and it was qualified together with Alt-Ergo (Why3 [FP13] was not used by Caveat to address the different automatic provers). Frama-C uses a different specification language named ANSI/ISO C Specification Langage (ACSL)[1] for writing annotations. ACSL is a language based on the paradigm of programming by contracts [JM97]. Frama-C lies on top of the Why3 framework to request different provers or SMT solvers.

SPARK with the latest SPARK 2014 version is a tool set for deductive reasoning based on the SPARK language (a formally defined subset of the Ada programming language) [Bar00, Bar03]. Similar to Frama-C WP, it uses the Why3 platform to request the supported provers. The differences between SPARK and Frama-C are discussed in [Moy09].

Frama-C and SPARK can also be used for dynamic software verification [KMMS16], which has the advantage to use the contracts already written for the proof and test them on a running software.

1 ACSL specification language: https://frama-c.com/acsl.html

The *B-method* [Abr96] uses a particular abstract machine notation to begin reasoning on a simpler abstract representation of a program (for example: "the airplane has one landing gear"). After multiple *refinements* (more details are added at each step, for example: "the airplane has three landing gears") until a deterministic version is achieved (the implementation). This deterministic version, also known as B0, can be directly translated into a programming language for compilation. Individual refinement steps must be proved correct, that is the effect of the concrete specification must not contradict the effect of the abstract/refined specification, in order for the final program to enjoy the same properties as the original specification. Each step thus generates a number of proof obligations that must be discharged by an automatic prover. The idea is that the correctness of each individual step is in principle much easier to establish than the overall correctness.

The B-method was originally developed in the 1980s by Jean-Raymond Abrial in France and the UK. *B* is related to the *Z* notation (also originated by Abrial) and the name was chosen to take the first free letter of the alphabet after *Z* (there was already the *A* programming language, so Abrial took *B* for its new language)[2]. The B-method was initially provided with two complete tool sets (Atelier-B from ClearSy (France) and B-Toolkit from B-Core (UK)) and a methodology that could be used in the industry. The first real industrial success was Meteor line 14 driverless metro in Paris (110 000 lines of B models) [LSP07]. Recently, another formal method using refinement called *Event-B*[3] has been developed to cover the modeling of a complete system. It came with the Rodin tool developed and maintained by Systerel. B and Event-B are compared in [Abr18].

4.1.4 *Combining Program Verification Methods*

Since no single formal method can ultimately solve the verification problem, the current trend is to combine formal methods.

For example, as proposed in [Sai00] one can rely on a user designed abstraction and derive a finite abstract model of the program semantics by abstract interpretation, prove the correctness of the abstraction by deductive methods and later verify the abstract model by model-checking.

In one of our experiments [TTBH19b], we also used a combination of formal methods to help the deductive proof. We used the results obtained by an abstract interpretation tool to substitute a verification condition that the deductive prover could not prove. This experiment is presented in details in Chapter 8.

2 Seminar lecture of Jean-Raymond Abrial at Collège de France: https://www.college-de-france.fr/site/gerard-berry/seminar-2015-04-01-17h30.htm

3 Event-B and the Rodin Platform: http://www.event-b.org

4.2 INDUSTRIAL APPLICATIONS OF FORMAL METHODS

Formal methods can contribute to the development of systems in two ways (and ideally both together):

- They can improve the development process, leading to a better product, and/or reduced time and cost;
- They can contribute to the assurance and certification of the system.

In order to apply formal methods in the industry, these two items should be estimated in terms of potential gain: how many bugs are removed when using formal methods in the development process; how time and cost are reduced; what certification activity is reduced when using formal languages and tools; etc.

Formal methods can be applied at different stages of the development process. Considering a standard V-model, they can be used at requirements design phase to check their consistency. At system design stage, they can be applied to formalize and check some system functional properties. At architecture design stage, they can be used to check the integration of the software [CGM+12, LBCG16, BW16]. However, the most of the applications that exist today are found at the module design phase (the lowest part of the V-model). Nevertheless, the most effective applications seems to be in the early design phases because of the errors coming from the natural language specification as proposed in Rusby's report [Rus95]. This report also gives an overview of the industrial application of formal methods and their role in the certification of critical systems. Another more recent report on the industrial application of formal methods is given by Woodcock in [WLBF09]. It proposes a survey of the formal techniques and where they are mostly used. It introduces the notions of *lightweight* and *heavyweight* formal methods. Lightweight formal methods emphasize partial specification and focused application on safety for example. They are easier to be introduced in the industry than heavyweight ones, which aim at full formalization of the system.

4.2.1 *Formal Methods Comparison*

There is no universal formal method and thus we need to know which formal method can be used for what kind of problem and at what level of abstraction. This comparison can also depend on the point of view (industrial or academic). Here we present a panorama of formal verification tools initially proposed by Xavier Leroy[4], which from our point of view reflects best the industrial practice.

It uses three axes: *automation level*, *security* and *scalability*. The security axis represent also the *expressiveness* level of the tool category.

4 Panorama of formal tools: https://xavierleroy.org/talks/ERTS2018.pdf

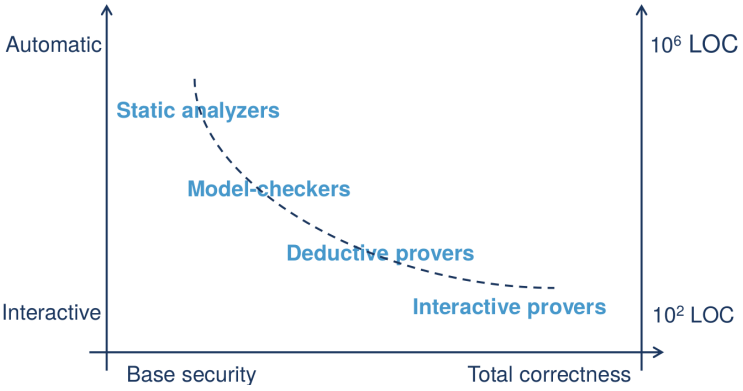


Figure 9: Panorama of formal verification tools (X. Leroy)

Static analyzers based on abstract interpretation are focused on efficiency rather than expressiveness. Thus these tools have the highest automation level and are the most scalable (few million lines of code can be analyzed). They work on the code level (in comparison model checkers work on a model level) and provide basic security guarantees.

Model checkers work generally on a model representing the system. Sometimes, models can have formal semantics (based on a formal language) and if they are accompanied by a certified code generator such as KCG, the produced code is proved to be equivalent to the model. Thus, the guarantees obtained on the model level are valid on the code level. The properties that can be expressed provide more security compared to the static analyzers but model checking is less scalable because of the state space explosion problem.

Deductive program provers work generally on the level of the code. They provide more security and correctness compared to the model checkers because of the use of more expressive languages (based on First-Order Logic (FOL)). Because of the amount of annotations needed to be written (sometimes as much as the code), these methods are generally less scalable.

Interactive provers (also known as *proof assistants*) use a Higher-Order Logic (HOL) and have the best expressiveness and security level. They aim at full functional correctness rather than efficiency. They are rarely used in the industry and we do not focus our work on them. However, their use in the industry can be indirect – using tools or applications that are already proved correct with a proof assistant by an expert.

4.2.2 *Abstract Interpretation Applications*

In 1993 Connected Components Corporation created in the U.S.A. by W.L. Harrison used abstract interpretation for compiler design [Har97].

In 1998 AbsInt Angewandte Informatik GmbH was created in Germany by R. Wilhelm and C. Ferdinand. The company commercialized the program an-

alyzer generator PAG and an application (*aiT*) to determine the worst-case execution time for modern computer architectures with memory caches and pipelines [AFMW96]. Lately, the company commercialized a sound static analyzer for C/C++ code called *Astrée* [CCF+05]. Astrée was initially developed by the computer science laboratory at the Ecole normale supérieure in Paris and then was transferred to AbsInt to be commercialized for Airbus. Airbus is using Astrée to prove the absence of run-time errors in their critical code. The objective to obtain zero false alarms was achieved by non-expert users by fine-tuning Astrée (e.g. use of partitioning, particular abstract domains) [DS07, SD07]. Astrée and some of its industrial applications in the aviation and space industries are presented in [Bou11]. Today, Astrée focuses on large-scale analysis with some few million lines of code [Kss+19]. The aim is to be able to analyze for example large automotive codes at the integration stage, which is recommended by ISO 26262 since 2018 (see Table 4).

In 1999 Polyspace Technologies was created in France by A. Deutsch and D. Pilaud to develop and commercialize ADA and C program analyzers. At some extent, the development of *Polyspace* was motivated from the failure of Ariane 501 maiden flight in 1996. This failure was due to an overflow (trying to put a 64 bits floating-point number into a 16 bits integer). Before creating Polyspace Technologies A. Deutsch proposed to use a Software Architect activity [LMR+98], which consisted of static analysis of the embedded control software of Ariane 502 and 503. As this space software was written in ADA, *Polyspace* supported since its first versions ADA. Nowadays, C and C++ are also supported. Polyspace tend to be more user-friendly than Astrée and requires less user expertise, which is a good point for beginners. However, when a finer control of the tool is necessary (e.g. select a particular abstract domain for better precision or increase precision for some part of the code) it is often impossible to modify the tool's automatic strategies. Polyspace and its industrial applications are presented in [Bou11]. Examples are given for the automotive, medical, military, nuclear domains. The book [Bou11] also presents a study about the use of static analysis with Polyspace to evaluate the robustness of the software by propagating the ranges of the input variables and observe if they generate run-time errors. Finally, in [BNSV14] the National Aeronautics and Space Administration (NASA) considered that Polyspace was very successful for Ada code but fell short for C/C++ with difficulties to scale and reporting a big number of warnings.

In 2013 was founded TrustInSoft as a spun off from the French Alternative Energies and Atomic Energy Commission (CEA). It developed a static analyzer called *TrustInSoft Analyzer (TIS Analyzer)* using abstract interpretation and based on the open-source *Frama-C value* plugin [CCM09]. The objective was to simplify the use of a static analyzer by proposing to take into account user's test scenarios or even to extend the ranges of some input variables and find run-time errors or security issues. Today, the CEA continues to improve Frama-C in parallel by proposing the plugin *EVA* [Bü17] as an open-source alternative.

It is generally hard to compare the different sound static analyzers. We can cite a study done by the French Electric Utility Company (EDF) for the analysis of the nuclear plants control software [Our15]. They compared Polyspace to Frama-C and found that the results of Frama-C were more precise. Actually, Polyspace raised too many orange alarms (potential bugs to be confirmed).

CodePeer is a sound static analyzer developed by SofCheck and AdaCore to analyze ADA programs. The idea was to propose a deeper code understanding by providing in an IDE annotations about the returned value or compute the contracts of a procedure even if the code is not completely written. A detailed presentation is given in [Bou11].

IKOS (Inference Kernel for Open Static Analyzers) is an open-source sound static analyzer for C/C++ programs developed at NASA Ames [BNSV14]. The novelty is that it can parse C/C++ dialects or idioms that are commonplace in the embedded world by using Low Level Virtual Machine (LLVM) [LA04] instead of Common Intermediate Language (CIL) [NMRW02].

Other static analyzers based on abstract interpretation are *Julia* [FCS18] for Java and *Infer* [noa20] developed at Facebook for Java and C/C++.

Fluctuat [GP15] is a static analyzer developed by the CEA. It is dedicated to the numerical precision analysis of C programs and uses abstract interpretation. Airbus is using it to check the numeric precision of the floating-point calculations [DGP+09]. A typical scenario is to see if there is a regression in the precision between the previous version of the software and the current one.

Today, sound static analyzers report bugs and even if they can provide the values calculated for each variable, for complex programs, it is sometimes difficult to understand the origin and the propagation of the problem. In [Moy10], the author proposes that tools should compute functional properties like values, relations, preconditions, postconditions, and dependencies but also non-functional properties like coverage, memory footprint, WCET, and profiling. It can help a programmer understand complex behaviors and detect subtle bugs.

4.2.3 *Model Checking Applications*

In order to be accepted by the users, a new technology needs to be as transparent as possible to the process that the engineer is used to have. The difficulty in the transfer of model checking to the industry is that it cannot be transparent and the complete process is changed. It requires developers to become part of the verification process (by specifying properties). It requires test team to learn a completely new tool with new concepts that are not transparent at all for someone who understands testing in terms of executing the design through scenarios.

Even if at the beginning the transfer to industry was operated mostly by start-ups, in 1990 Intel became the first exception and the first design company to seriously support model checking [Kur08]. Other companies such as IBM have also applied model checking for hardware verification.

The application of model checking on software was much harder than for hardware. There were cases of successful application to restricted forms of C language [BR02, KT14, PDBC+15, ARG+20] but the full support of the C syntax seems to be a major difficulty, which limits the industrial use. Furthermore, model-based design is massively used today in the automotive industry in replacement of direct C coding. Thus it is more convenient to use model checking at the model level.

A successful application of model checking for the industry was proposed by Gérard Berry in 2000, when Esterel Technologies was created (Esterel comes from the name of a beautiful mountain in Southern France and also from Real-Time, "temps réel" in French). The background technology of the first product proposed by the company – Esterel Studio – was the *Esterel* synchronous formal language [BG92]. It was meant to bring synchronous programming language benefits into the industry (initially telecommunications and then electronic design automation). It was also applied to avionics software verification at Dassault Aviation [BBF+00]. In 2003, Esterel Technologies bought a Lustre-based (one of the other synchronous programming language) tool set named Safety Critical Application Development Environment (SCADE) and the two academic communities behind these languages proposed a way to merge them. *Lustre* [CPHP87, HCRP91] (the name comes from "Lucid synchone temps-réel" in reference to the Lucid dataflow language [WA85]) was used for the dataflow constructions and Esterel (based on SyncCharts [And03]) was used for the synchronous automata. It resulted in the *SCADE Suite* modeling tool and its Scade 6 formal language [CPP17]. Because SCADE Suite is based on a formal language, it can be used for formal analysis of high-level models for critical applications in aerospace, railway and automotive, thereby circumventing the obstacles of the C code verification.

To make the use of model checking easier, a tool called *Prover Plug-In* from Prover Technology was integrated in SCADE Suite and named Design Verifier (DV). It was based on the use of *synchronous observers* [HLR93, BD05] to design the properties with the same principles and blocks as the model is designed. It consists of connecting the observer block (representing a modeled property) to the inputs and outputs of the model. Compared to the use of temporal logic in classical model checking, it has the advantage that the engineers are already familiar with the formalism and do not need a special expertise to use model checking.

Another modeling tool that lately adopted a similar concept of application of model checking was *Simulink/Stateflow* from MathWorks but some differences with SCADE Suite exist:

- Compared to SCADE Suite, Simulink and Stateflow does not have a consistent semantics, which means that some features of these tools should be avoided when modeling embedded software [BP13]. These features could only be used for prototyping but not for formal verification;

- The evolution of the Prover plug-in integrated with Simulink/Stateflow seems to be managed internally by MathWorks. SCADE Suite continues to integrate updates and last improvements from Prover Technologies. For example, today SCADE Design Verifier (SCADE DV) can reason with floating-point numbers but not Simulink Design Verifier (SLDV), which still uses rational numbers;

- SLDV can be used for automatic test generation, which is used for back-to-back testing to bring some guarantees that the code behaves as the model (the Simulink/Stateflow code generator is not certified). As SCADE Suite has a certified code generator, back-to-back testing is not necessary and there is no automatic test generation.

- With the last versions of SCADE Suite it is possible to export the code sent to the prover in the High Level Language (HLL) [OBC18]. Thus it can be proved by other model checkers such as S3 from Systerel [PDBC+15].

- For Simulink/Stateflow, NASA Ames has developed a framework[5] to parse and export a subset of the Simulink blocks into Lustre language. The aim is to use other model checkers such as KIND 2 [CMST16] and JKIND [GBW+18] for proving properties. As far as we know, there is no tool that can export SCADE Suite into Lustre. The reason seems to be that there are many Lustre variants and no unique standard, which motivated the creation of the HLL.

An experience showing the use of SLDV can be found in [EFJ10]. In this report, Safe River shows how they used model checking to prove the correctness of the safety properties of the train tracking function for an Automatic Train Protection (ATP) system based on Thales specifications.

SCADE DV is used in railway but also in aviation. The HLL language used to communicate the model and the properties to the model checker, is the result of a collaboration between Prover Technology and RATP. It is the successor of Prover's Tecla language. The main goal of HLL was to add the necessary features to enable formal verification of Communication-Based Train Control (CBTC) systems.

The Airbus experience with model checking is detailed by Thomas Bochot in his thesis [Boc09]. It examines all the experiments done using SCADE DV on the Flight Control System (FCS) models and the encountered problems, in particular the possibility to provide several counterexamples and their structural coverage [BVWW10].

The company that has done the greatest experiments of model checking for software verification in an industrial context is probably Collins Aerospace. They had developed a framework [CWM08] to convert Simulink or SCADE models into Lustre and use different model checkers as back-end. As far as

5 CoCoSim: https://ti.arc.nasa.gov/tech/rse/research/cocosim

we now, this framework is no more supported but they continued developing tools for software design verification based on Lustre language and using JKind [GBW+18] as a back-end. All the studies are published on their website[6]. A first study [WCM+08] permitted to discover 12 errors in analyzing 62 properties of a system model developed by Lockheed Martin. Another study [Mil09] used model checking to analyze a complex cockpit application and fix 98 design errors. An important topic for the industry is the use of model checking to obtain certification credits in a certification context. This topic is discussed in [Cof10, CHHL13, CM14a, CM14b].

4.2.4 *Deductive Proof Applications*

The motivations to introduce deductive proof in the industry can be different but it is always necessary to take into account the return of investment. A strategy to justify it can be to replace a costly development process activity such as unit testing. Airbus used this strategy – replacing unit testing by unit proof – to introduce *Caveat* tool [RSB+99] in 2002 for the development of Airbus A380. As reported in [Bou11], the first motivations for introducing unit proof was the mastering of the verification process and its costs in the face of a constant increase in the complexity of the software being developed. Even if there were an extra cost to qualify Caveat and Alt-Ergo, Airbus obtained a global cost reduction and quality improvement. Caveat was able to analyze C programs (with some restrictions in terms of language constructs) and had its own specification language based on a first-order logic. Today, Caveat is no longer supported and Airbus moved to a specifically developed version of *Frama-C WP*, which has a better ratio of automatically proved goals. The entire work on Caveat and Frama-C was done in partnership with CEA and the industrial applications of Frama-C are described in [DMLK+16].

To further automate the development process of embedded handwritten C code, Airbus proposes to use another language named Compilable Design Description Assistant (CoDDA) to describe the software architecture and a new language named Design Contract Specification Language (DCSL) based on a FOL to describe the the software low-level requirements [BDE+18, BCD+20]. This tool chain is used to address multiple methods of software verification: unit proof (DCSL is translated into ACSL for Frama-C WP), testing (for goals that cannot be proved automatically), static analysis (flow control check with Fan-C [Del12] and value range check with Astrée). A 12-day training was necessary for the engineers. From an engineer point of view it was more interesting (a more intellectual activity) to do unit proof than unit testing.

The first serious application of *SPARK* came with the project SHOLIS (27 kloc of SPARK code) in 1995 [Cha00, CS14]. 75.5% of the proof goals were proven automatically and the rest was proved in an interactive way. Another

6 Collins Aerospace formal methods group: http://loonwerks.com

successful application of SPARK was for the mission computer software of the Lockheed-Martin C130J. As far as we know, the applications of SPARK today are mostly in the military and defense domain, but some applications were done in other domains such as automotive [Sch19]. A recent report [DFJ+18] presents the use of SPARK at Thales and a methodology with different levels of adoption of formal methods – from *stone* (the lowest guarantees) to *platinum* (total correctness) level.

B and the *B-method* are used today mainly in the railway industry. When inventing it, Jean-Raymond Abrial thought that it could be successfully deployed in other industries such as energy, automotive, aeronautics, space, etc. However, he says that today people in charge of these industries consider that it is too difficult to modify engineering approaches, which have been established for many years [Abr18]. In [LDPM17], the authors summarize 25-year return of experience in the effective application of B and Event-B in diverse application domains (railways, smartcard, automotive) and report on the difficulties they have encountered.

4.2.5 *Interactive Proof Applications*

Interactive provers need expert knowledge to be used and are rarely directly used in the industry by the engineers. However, they can be used indirectly. For example, the interactive proof assistant *Coq* [BCHPM04] was used to formally specify, verify and generate the *CompCert* optimizing compiler. In 2011, Airbus evaluated its performance for the development of level A critical flight control software and the results were rather promising [BFFFL+11]. Another practical use was to formally verify the *seL4 OS kernel* [KEH+09] using *Isabelle/HOL* proof assistant [NPW02], which provided similar performance as other L4 kernels but with formal guarantees. Prove & Run also provides a proven OS kernel called ProvenCore, which is a minimal trusted operating system that can run alongside a conventional OS in an embedded systems.

4.3 FORMAL METHODS AND CERTIFICATION

There seems to be two reasons why an applicant might consider using formal methods in support of certification [Rus95]: (1) to achieve the same level of quality control and assurance as by other means, but to derive some other benefits such as reduced cost; or (2) to provide a greater level of quality control and assurance than can be achieved by other means. For example, for reason number (1) we can cite the replacement of unit tests by unit proof with Frama-C WP at Airbus Group [Bou11] and the use of SCADE for code generation. The reason number (2) was the motivation for the HACMS program [FLR17] for providing greater assurance in cyber security.

Several concrete applications of formal methods in the certification of safety-critical software systems have been conducted by: Airbus Group presented by Souyris in 2009 [SWDD09], Bedin França in 2011 [BFFFL+11] and Brahmi in 2018 [BDE+18]; Dassault Aviation presented by Moy in 2013 [MLD+13]; Collins Aerospace presented by Cofer in 2014 [CM14a] and Wagner in 2017 [LAC+17]; in the railway company RATP presented by Halchin in 2020 [HAAS+20].

An extensive discussion on advances regarding software certification is provided in a 2013 Dagstuhl report [CHHL13].

The qualification of some formal tools seems to be particularly difficult because of the complexity of the tools. An interesting approach through generation of proof certificates and verifying them with a qualified proof checker (simpler to qualify than a model checker) is proposed in [LAC+17].

4.4 CHALLENGES FOR THE APPLICATION OF FORMAL METHODS

We can found in the literature many publications about the application of formal methods but we should distinguish two cases: formal methods that are really used for production code and formal methods that have been experimented by a research department of a company or research laboratory but still not deployed in production. The most of the publications are about the second category. The reason is that there are few formal tools that can be used by the engineers without training them for months. Rustan Leino, a prominent researcher in formal program verification, reported that program verification is currently unusable, because current tools require too much expertise from the user [LM10]. To massively introduce formal methods, the tools should be accessible in a cost-effective way to serious (motivated) non-expert users. He put a strong emphasis on the user-interface considerations and a better explanation of the errors reported to the user.

Toyota has been investigating how they can use formal verification and what are the challenges for them. In [JDK+14] they discuss about the challenges of applying formal methods for model-based design and propose to extract design knowledge from the simulations of the design to construct a formal model. Other papers [HKAS14, HAK+16, HAKA16, HAKA17, Are19] seem to be more research papers than an industrial application of formal methods.

4.5 CONCLUSIONS

In this chapter, we proposed a brief overview and comparison of formal methods that can be used in the industry. Static analysis based on abstract interpretation provides a sound approximation of the values that the variables can take in the code and can be used to guarantee the absence of run-time errors. This method is recommended by ISO 26262 for all ASIL levels. Model checking can be used together with model-based design to guarantee that the safety

requirements are correctly implemented early in the design phase and there are no regression when the model is updated. In our opinion, even if there are some limitations, SAT/SMT induction-based model checking is one of the most promising techniques because it gives a simple way for non-expert engineers to describe the properties using the same graphical language they use for the model. Furthermore, SAT/SMT solvers are being constantly improved so some limitations could be overwhelmed in the future. Finally, deductive proof can be used when manually coding C and Ada programs or for proving critical library functions.

In the next chapter, we present some methodologies that can be applied by non-expert engineers to use formal methods in the automotive embedded software development.

Part II

AUTOMOTIVE SOFTWARE DESIGN USING FORMAL METHODS

5

METHODOLOGIES FOR USING FORMAL METHODS IN AN AUTOMOTIVE CONTEXT

Simplicity is the ultimate sophistication.

— Leonardo da Vinci

The automotive industry has nowadays largely adopted model-based design tools such as Simulink and SCADE to design models and generate embedded code. Recent advances in formal analysis tools have made it practical to formally verify important properties of these models to ensure that design defects are identified and corrected early in the development process. For cases where formal analysis cannot be used at the model level, other tools can statically analyze the code to find run-time errors that could cause the software to behave unexpectedly.

Formal analysis tools are classified in the literature as *lightweight* and *heavyweight* based on how easy they are to be used and on their automation level. Lightweight formal tools do not require deep expertise, by opposition to heavyweight ones, which are more complex, less automatic, but more expressive and powerful. In the automotive industry, we target lightweight tools that are mature enough to be used and are provided with tool support. Academic tools are not to be neglected as well because they can provide ways to understand a problem that are not supported by an industrial tool.

This chapter describes how formal analysis tools can be introduced into the development process to decrease the cost and increase the quality of critical automotive systems. We target their potential use by non-expert engineers that are already familiar with model-based design and C code. We provide methodologies and some examples of concrete applications.

5.1 RELATED WORK

Formal methods exist since 1960's but they began to be used in the industry since 1980. One of the reasons was that there were no methodologies presenting how they could be used in the industry.

For the first time, Balzer and his colleagues [BCG83] proposed in 1983 a way to integrate formal methods in an industrial development process. This integration comprises:

- *requirements* analysis;
- construction of a *formal specification* and development of a functional *prototype*;
- development of the *implementation* (code), having as a basis the model and the prototype.

The novelty of Balzer's life cycle was that it explicitly advocated the use of formal methods at different development stages, and in particular, where the relationship between the requirements, the formal specification and the implementation was concerned. An aggregate representation of Balzer's life cycle is shown in Figure 10.

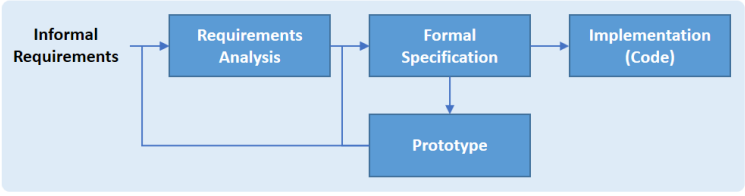


Figure 10: Balzer's software life cycle

Adapting Balzer's principles to the contemporary formal methods, their integration in the V cycle can be done in one of the following ways:

- *Vertical application*: the *Formal Specification* phase of Balzer's life cycle encompasses the inner design stages of the V life cycle (see Figure 11) from *System Design* to *Coding*. This is, for instance, the approach followed by B, VDM and SCADE. It is also referred as the *correct-by-construction* approach.
- *Horizontal application*: Balzer's life cycle can be applied on each design stage. For example, during the system architecture design, it is translated into a formal specification, which is then formally analyzed and verified. In this context, the application of the Balzer's *Implementation* stage is optional.

For our methodologies, we take inspiration from Balzer's life cycle and apply it in a vertical way for tools that have no significant impact on our current development process (for example, using model checking to verify an existing model) or in a horizontal way for particular design stages (for example, using abstract interpretation to verify the code).

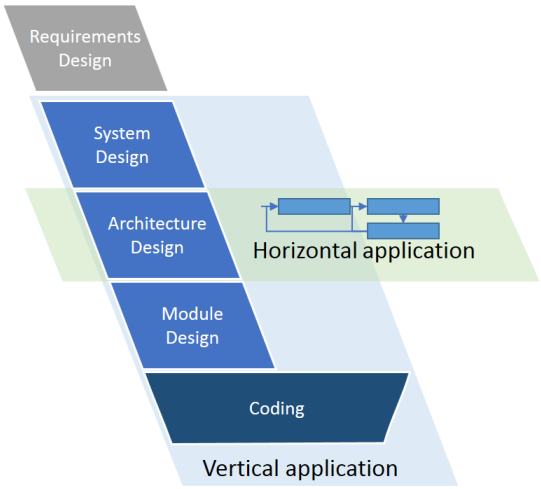


Figure 11: Horizontal and vertical applications of Balzer's life cycle

5.2 METHODOLOGY FOR MODEL-BASED DESIGN

When using a modeling tool, different errors can be introduced. They may be due to ambiguous or wrong specifications, wrong requirements interpretation, overflows in the arithmetic operations, divisions by zero, copy-paste blocks with parameters that should be adjusted differently and so on. Even if the code is generated automatically, all the errors introduced in the model will be present in the final code. Neither the automatic code generator, nor the compiler will prevent such errors. If they are not discovered early, these errors can be found when the software is already in the car resulting in higher costs to fix them.

5.2.1 *Motivation and Objectives*

We want to apply inductive SMT-based model checking to prove that safety requirements are implemented without bugs. We can even extend the scope to other requirements that could have an impact on the driving or the image of the brand. Current model checking tools are very powerful and provide much more automation than theorem or deductive provers. In general, they require less user expertise but the user must be able to rewrite textual requirements in a formal language. Some of these tools are mature enough to be used in the industry and, in our opinion, the benefits of using formal methods are greatest at the model's level. We propose to introduce model checking during the design of the implementation model to obtain the highest benefits (see Figure 12).

To demonstrate in a certification context, that all the requirements are implemented in the production code, we can use testing and/or formal proof. The principal motivation to use formal proof is its exhaustiveness and reduced

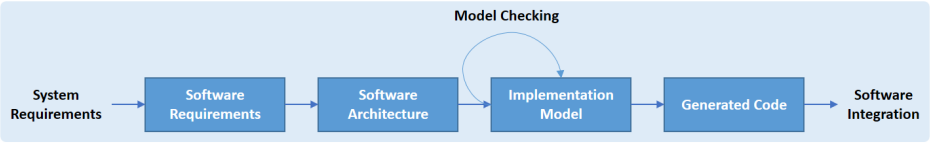


Figure 12: Methodology for using model checking in a model-based design

cost. Actually, writing good test scenarios with one hundred percent Modified Condition/Decision Coverage (MC/DC) coverage is rather complicated and can take much more time compared to defining formal properties and proving them. Because of this difficulty with testing, DO-178C and DO-333 accept the use of formal proof instead of coverage testing.

5.2.2 *High and Low-Level Requirements*

Multiple levels of requirements can exist in a model-based design process. In most of the industries, there are at least two levels of requirements: system level (High-Level Requirements (HLR)) and software level (Low-Level Requirements (LLR)). System requirements are often expressed on the bounds of the system (inputs and outputs). On the other hand, software requirements are detailed and in addition of the inputs and outputs of a system, they can also specify the behavior of the internal data of the system.

During design phase, the implementation model is tested to verify the software requirements (LLR) but it is often difficult to conclude at that stage that system requirements (HLR) are met. Actually, the system requirements validation is done later in the design process on the target platform, which can result in costly bug fixing.

We propose to introduce formal proof in the safety-critical models for system requirements (HLR) to have an early insight about the system's validation. The advantage of verifying system level requirements on the implementation model is that verifying a single high-level property can find many errors in the implementation. The errors found can be design errors or specification inconsistencies. In the following, we propose some guidelines for writing good formal properties.

5.2.3 *Guidelines for Writing Good Formal Properties*

Writing good formal properties has many similarities with writing good requirements and is as much an art as a science. Actually, once the properties are written it is easy to replay or update them, but writing the first set of properties for a function can be challenging. However, it is possible to put a set of properties into a library and reuse them for other projects.

One of the best sources for formal properties is the safety-related requirements for the system. In the automotive domain, these requirements can be

found in a document called *Technical Safety Concept (TSC)* or *Functional Safety Concept (FSC)* (document required by ISO 26262). These requirements are not only important from the safety point of view but very often, their implementation affects several parts of the system. This cross-system scope of the properties is beneficial to find most of the errors.

Another excellent source of formal properties is the system (high-level) requirements specification because its intent is to describe what the system should do instead of how. A good practice is to transform the system requirement into multiple textual properties and then rewrite them using a formal language.

When system developers are available, a good strategy is to ask them what are the things they are the most worried about in their system. Refining their concerns into properties can require an ongoing dialogue but their knowledge of the system can be invaluable.

User manuals can be another excellent source for writing formal properties. Actually, these manuals contain fewer details than the system requirements and are more user oriented.

Once all the sources of properties have been exhausted, we can carefully analyze the model, looking for anything that has not been checked by a property. For example, we can look for inputs and outputs that are not present in the current properties, and try to write properties about them. The discovery of such inputs can also be automated with techniques such as slicing and mutation coverage (see Chapter 7). Surprisingly, even writing properties directly from the model itself will often expose errors in understanding the semantics of the model and should not be excluded as a strategy.

When a property is found to be false, this only means that there is a discrepancy between the property and the model. Before looking for problems in the model, we should begin by looking for missing assumptions in the property. It is generally better to write and check the properties incrementally rather than writing all the properties and check them after that. In this way, we gain a better knowledge of the conditions that are necessary to validate the properties, and of the assumptions about the environment of the system. This knowledge helps in writing the rest of the properties.

5.2.4 *Synchronous Observers*

We propose to use the method called *synchronous observers* [HLR93], mentioned in Chapter 4, to model the properties in a model-based design environment. An observer "observes" the behavior of the model and decides if it respects the properties. Technically, an observer is a model connected to the model under verification in a synchronous way. Figure 13 shows a typical architecture for verification, which includes two categories of observers:

- *environment assumptions* (A): used to constrain the input values. For the assumptions, it is possible to use the previous values of the outputs. The assumption observer computes a Boolean output A that the model checker is not allowed to falsify. This avoids that the model checker generates counterexamples that violate the assumptions;
- *properties* (P): modeled to produce a Boolean output P (one per property), which is *TRUE* while the model satisfies the property under the environment assumptions;

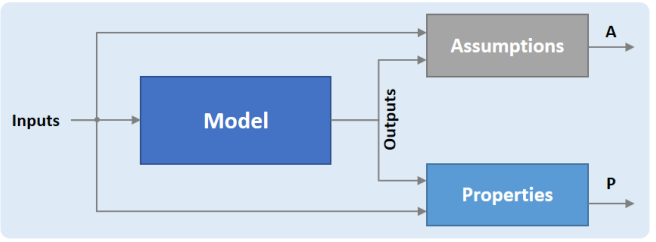


Figure 13: Model-based design model checking using synchronous observers

It is a good design practice before beginning the modeling of the properties to add environmental constraints (assumptions) to reduce the range of the inputs in order to get a faster proof. Once all the properties are proved valid, we can remove the assumptions and try to prove again the properties. It will show us if the model is robust for input values that we do not expect to receive. An example of properties expressed as synchronous observers is given in Chapter 6.

5.2.5 *Libraries and Imported Functions*

In a model-based design tool, not everything can be expressed only with the blocks provided by the tool. In practice, the implementation model often uses library blocks, for example to call some AUTOSAR routines. These functions are represented in SCADE as imported operators and their behavior is unknown in the modeling tool. In this case, the model checker considers that all the variables of the imported operator are free variables, that is they can take every value of their type. We can use assumptions (*assume* operator) to constrain the possible values of the inputs, if we know them. These constraints should be defined in the top-level (root) operator. An alternative approach of constraining the ranges for input variables is to use their type. For example, instead of using an integer type to return an error code, we can use an enumeration. If a property is falsified, we need to use a stubbed version of the imported operators in order to simulate the counterexample. Actually, without the stubs, SCADE cannot run the traces obtained from the counterexample.

To simplify the model checking process, a good practice whenever possible is to have a model to be proved without imported functions. The imported functions should be kept outside of this model as shown in Figure 14.

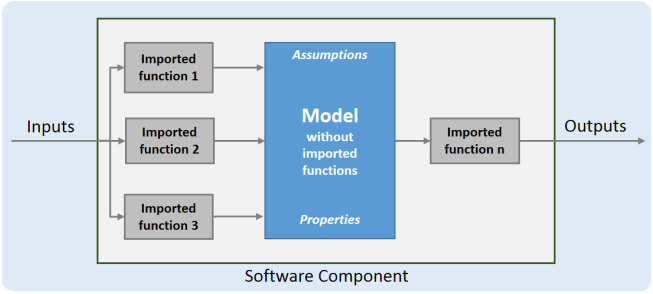


Figure 14: Model checking without imported functions

More generally, taking into account within the design choices at the beginning of the design process that a model will be verified by model checking is an important point that can largely facilitate the verification process.

5.2.6 *Workflow*

The workflow for using model checking in a MBD process is presented in Figure 15. It consists of successive tasks that can be iterated as long as we need, beginning from designing a model, expressing and proving properties and analyzing the results.

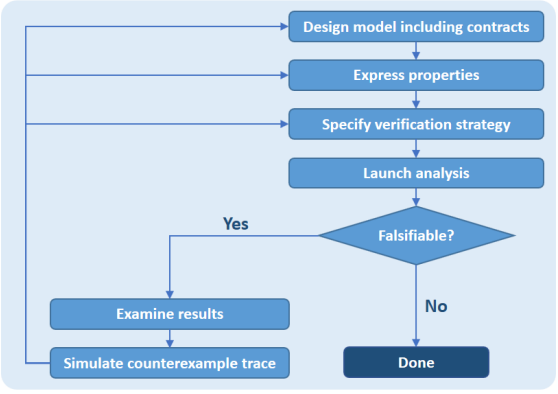


Figure 15: Model checking workflow in a MBD process

5.2.7 *Run-time Errors Check*

In addition to the proof of functional properties modeled as observers, the model checker can also be used to prove the absence of some run-time errors. For example, SCADE DV proposes to check the absence of divisions by zero and overflows.

5.2.8 *Proving Non-regression*

The model checker can also be used for proving non-regression by checking model equivalence. We can simply create a new operator/block with an instance of the old model and check whether their outputs are equivalent. Unlike regression testing, this method saves time and brings exhaustiveness (for all combinations of values of the inputs, the outputs of the two models provide the same values).

5.2.9 *Strategies*

Model checkers have plenty of useful options but they can be challenging to understand for non-expert engineers. That is why industrial tools propose some strategies combining the most common options of the model checker to address different stages of the development process. For example, SCADE DV proposes the following strategies:

- **Debug strategy**: based on BMC. It cannot prove the validity of a property. It is meant to be used during development to quickly find counterexamples when the model is suspected to be incorrect;
- **Induction strategy**: based on k-Induction (see Definition 4). It is meant to perform non-regression when knowing the induction depth, which is provided as a parameter to the strategy;
- **Prove strategy**: based on k-Induction, invariant generation and interpolation. It is meant to validate a design once the development is stable;
- **Custom strategy**: based on different options of the model checker. It can be adapted upon the customer's needs to fine-tune the parameters of the invariant generation, interpolation or induction engines.

5.2.10 *Limitations*

While models containing only Boolean logic (Boolean inputs and outputs) are well-suited for model checking, most model checkers can also handle models with enumerated types and small integers. Some model checkers can handle models with real numbers (KIND 2) and others can also handle models with

small floating-point numbers (SCADE DV) as they are bit-blasted and solved by a SAT solver.

Models containing nonlinear arithmetic cannot be handled by the commercial tools today. KIND 2 and JKIND also do a check for nonlinear arithmetic and stop if it is present in the model. As modern SMT solvers integrate theories for reasoning on nonlinear fragments, we experimented it on a nonlinear version of our Cruise controller function, enabling JKIND to use this theory. It took two days to prove a property on that model compared to some seconds for a linear model. As the SMT solvers are competing every year[1], we hope that these theories will be improved in the future.

5.2.11 *Experiments*

5.2.11.1 *Application 1 – Cruise Controller*

Our first experiment was to model with SCADE Suite an already existing cruise controller function (a linear one and a nonlinear one) to see if we are able to find more bugs thanks to the formal approach.

As a result, we were able to express and prove all the critical properties of the linear model in a reasonable time. For the nonlinear one, some of the properties were rejected by the model checker because it cannot work with nonlinear arithmetic.

We discovered that the presence of counters and time in the properties was rather challenging for model checking. We discuss this topic and present our contribution in Chapter 6.

Another interesting point is that the nonlinear cruise controller use a square root function, which computes a square root by applying a discrete-valued method (lookup table) instead of the standard C library function. As this square root function can be proposed in a library of imported functions for the final embedded code, we experimented how we can prove its correctness for a given contract using deductive methods. We present this experiment in details in Chapter 8.

5.2.11.2 *Application 2 – ADAS function*

In 2018, we experimented the use of SCADE Suite by a function design engineer who, after a short training, was able to model a function that was managing the automation level of a car. We wanted to prove that the safety requirements from the FSC were implemented in the model without errors. After rewriting these requirements to match the names and the values of the variables used by the model, we could prove them valid or invalid.

1 SMT-COMP 2020: https://smt-comp.github.io/2020

We found that some requirements were missing the exact conditions in which they should be applied and fixed them. For the other invalid properties, the reason was an error in the design.

We noticed an interesting point about the use of the prover. It can be used to prove properties one by one or to prove multiple properties at the same time. The difference is that checking multiple properties together can bring a bigger cone of influence and more lemmas (invariants) about the entire system, which can be beneficial if the properties are provable by the prover. However, in case of unprovable properties, checking them all together can render all the properties unprovable. If checking them all together seems to take too much time, it would be preferable to check them one by one. A good practice is to check them one by one using the *Debug* strategy to try to falsify them at a certain depth or for a given time. Properties that can not be falsified can be grouped and checked all together using other strategies such as *Induction* or *Prove*.

5.2.11.3 *Application 3 – Lighting Supervisor*

In 2020, we did another experiment with SCADE Suite to see how formal proof can be integrated in the software design process and produce formally proved production code. It was a step further compared to the 2018 experiment because in addition of the proof, it also targeted the integration with the AUTOSAR environment.

The first function to be developed was a module that supervises the lighting of a car, which is critical. It ensures that no matter what happens, the car lights will stay switched on unless the driver turns them off. After passing with success all the test scenarios, we used SCADE DV to prove eleven properties based on the FSC. Two bugs were found with the proof and the designer could fix them rapidly. These bugs uncovered an inconsistency between the specification and the model and were never found before with testing. If the designer had achieved 100% MC/DC coverage, maybe he could have found these bugs by testing, but achieving 100% MC/DC for some models may take a few weeks or months of work. With the proof process, it took only some minutes to describe the properties and analyze them.

Finally, we experimented the proof of absence of run-time errors for this module using model checking and abstract interpretation tools. We first used Astrée, which found one alert about a variable that was not initialized and two alerts about potential overflows. We fixed the first error by indicating in SCADE a pragma for a default state of an automaton. The problem came from the fact that SCADE, working at a higher level of abstraction compared to the code, knew that a state variable was always initialized because of the finite number of states present in an automaton. At the code level, this information is lost. Another way to solve this issue was to provide Astrée extra information about the number of possible states. As for the overflows, they could actually occur

and result in an endless loop. We also used SCADE DV to check for run-time errors and it found the same. After fixing them, the two tools proved there were no more run-time errors.

5.3 METHODOLOGY FOR SOUND STATIC ANALYSIS

Abstract interpretation is the most automated of the three formal techniques (see Chapter 4) and typically requires less user expertise. We use it to check non-functional requirements (e.g. run-time errors) that are specified in the tool (the user does not need to specify them).

Figure 16 presents the place in the development process where abstract interpretation can be introduced without too much impact. It can be used in the implementation model design phase to analyze the generated code or during coding (if MBD approach is not used). In the software integration phase, it could be beneficial for verifying in particular for data races and run-time errors that cannot be seen in a unitary component verification.

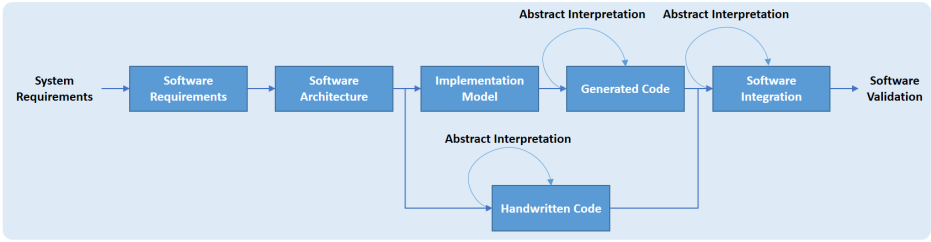


Figure 16: Methodology for using abstract interpretation

5.3.1 *Component-Level Analysis*

Since 2018, ISO 26262 recommends the use of abstract interpretation for software unit verification (see Table 3). We want to detect potential run-time errors and fix them early, preventing late-stage integration problems. In a process, where an unsound[2] static analysis tool is used, abstract interpretation can complement it by providing a precious information for understanding issues of the unsound tool when they are difficult to understand.

To apply component-level sound static analysis, very often the component alone will not be enough. Because the component may not contain a main function, may call library functions or even the definition of its data structures may be absent (they are generated at compilation time by the AUTOSAR tool chain), we need to have stubs for all the functions and data definitions that are absent. These stubs should be representative of what will be generated by the AUTOSAR tool chain. Unlike unsound static analyzers, which could analyze

2 Unsound means that the tool is unable to report all potential problems.

partially defined code and show zero alarms if the code is unreachable, sound static analyzers need to have access to all the types and functions definitions used by the analyzed code otherwise we can obtain false alarms or unreachable and not analyzed code. For a component-level analysis, a good metrics to check is the code reachability, which must be 100%.

5.3.2 *Complete System Analysis*

ISO 26262 also recommends the use of abstract interpretation for the verification of software integration (see Table 4). When analyzing programs integrated with the operating system, there can be a significant use of libraries. One possible solution is to include the source code of the libraries with the program but this is not always convenient (case of complex libraries, unavailable source code or not written in C). Furthermore, AUTOSAR increases significantly the number of library functions to be called. An alternative, proposed by some tools such as Astrée is to provide a stub for the standard C functions and also for the AUTOSAR standard functions.

The embedded software is composed of tasks and interrupts that can be run on a specific core of the microprocessor. To analyze a complete AUTOSAR system, Astrée proposes to import the operating system's configuration (generally an AUTOSAR XML file), which can be obtained from the AUTOSAR tool chain. Then the complete system analysis can be run. In practice, it needs a significant amount of RAM (more than 128 GB). A recent paper [Kss+19] presents the experience of Robert Bosch GmbH for a complete system analysis of industrial software. The experimental results confirm that sound static analysis can be successfully applied for integration verification of large-scale automotive systems.

5.3.3 *Hints for Reducing False Alarms*

Writing code has many similarities as writing poetry – it can be easy to read and understand or very difficult. Static analyzers based or not on abstract interpretation can have a partial "understanding" for a code, which can result in false alarms. During our experiments, we found such cases and propose some hints for decreasing the number of false alarms:

- When using a protection (a check if a variable is within an interval), the nearer the protection code is to the problematic construction (potential instruction that can cause a division by zero, overflow or out-of-bound array access) the best will be the precision of the static analyzer. For example, if we have an increasing counter in a loop, we should protect it from overflowing by checking before the increment that the result will be in the bounds of its type. Same is for all division operations that should be protected from dividing by zero;

- When using arrays, we should check that the index is inside the authorized interval. It is also preferable that the index is a variable instead of another array. For example, the instruction $a = tab1[tab2[x]]$ could be rewritten as: $b = tab2[x]; a = tab1[b];$ In this way, the static analyzer will propagate all the invariants found for b everywhere it is used in the code or even infer relational invariants.
- Partitioning (see Section 4.1.1.3) is a rather useful feature to be activated on parts of the code, where there seems to be a false alarm. It increases the precision of the analysis locally to remove false alarms.

5.4 CONCLUSIONS

In this chapter, we proposed two methodologies for using formal methods in an automotive context: one for models, based on model checking, and one for static code analysis, based on abstract interpretation. We targeted lightweight formal methods for non-expert engineers. We gave some examples of application of these methodologies and the obtained results.

We did not experiment deductive proof at the system level because it is not supported by tools usable by non-expert engineers. However, it can be used at the code level to verify code using Hoare logic and we made some experiments. We propose some preliminary ideas for methodology in Chapter 8.

To conclude, we think that the most important thing when using formal methods is to keep the software and its architecture as simple as we can for a better scaling of the formal analysis tools.

In the following chapters, we present the improvements we have done on the invariant generator of JKIND to take into account long running time properties. It is followed by a new algorithm for coverage metrics for evaluating the quality of properties (specification) that are proved valid. We finish by presenting our experience using deductive proof for proving the correctness of a discrete-valued function.

6

INVARIANT GENERATION FOR MODEL CHECKING OF TIME PROPERTIES

The noblest pleasure is the joy of understanding.

— Leonardo da Vinci

Modern automotive embedded software is mostly designed using model-based design tools such as Simulink or SCADE, and source code is generated automatically from the models. Formal proof using symbolic model checking has been integrated in these tools and can provide a higher assurance by proving safety-critical properties. Our experience shows that proving properties involving time is rather challenging when they involve long durations and timers. These properties are generally not inductive and even advanced techniques such as PDR/IC3 are unable to handle them on production models in reasonable time. As timers are something very common in industrial models, this difficulty motivated us to understand the problem and look for a solution.

In this chapter, we first present our industrial use case and comment on the results obtained with the existing model checkers. Then we present our new invariant generation algorithm and methodology for selecting invariants according to physical dimensions. They enable the proof of properties with long-running timers. Finally, we discuss their implementation and benchmarks.

6.1 USE CASE PRESENTATION

6.1.1 *Model and Environment*

We illustrate the use of symbolic model checking to prove the correctness of safety properties on a representative production model of a cruise control function. This function manages the speed of the car, switches to the right operating mode, manages the user interface, detects faults and decides whether the function should be turned on or off. It uses only linear arithmetic over integers.

We used ANSYS SCADE Suite to design the model from low-level software textual requirements. The properties to be checked were also modeled in SCADE from high-level system safety requirements. The proof was done

with SCADE DV [BD05], which is a symbolic model checker integrated in SCADE. We chose SCADE for our experiments because it has formal foundations [CPP17] compared to Simulink, which is more simulation oriented and without a single formal background [ZZWF]. Actually, SCADE has a formal language based on Lustre, thus we could compare its internal model checker with other open source model checkers for Lustre.

At Groupe PSA, the embedded software is developed according to a standard V-Model methodology. HLR (system requirements) are allocated to an ECU. Then they are decomposed into LLR (software requirements) used to develop the code (handwritten or model-based).

We present our use case environment in Figure 17. It is composed of a SCADE model, properties, and assumptions when needed. We used multiple model checkers to compare their performances (GATeL [MB05], SCADE DV, JKind, Kind 2, and addressed multiple SMT solvers in the back-end (CVC4, MathSAT, SMTInterpol, Yices2, Z3). GATeL has its own SMT called Colibri developed at CEA. SCADE DV has its own SMT provided by Prover Technologies. JKind can use CVC4, MathSAT, SMTInterpol, Yices2 and Z3 via SMT-LIB. Kind 2, the successor of PKind and Kind, can use CVC4, Yices2 and Z3 also via SMT-LIB. At the moment of our experiments, Kind 2 was unable to use IC3 with Yices2.

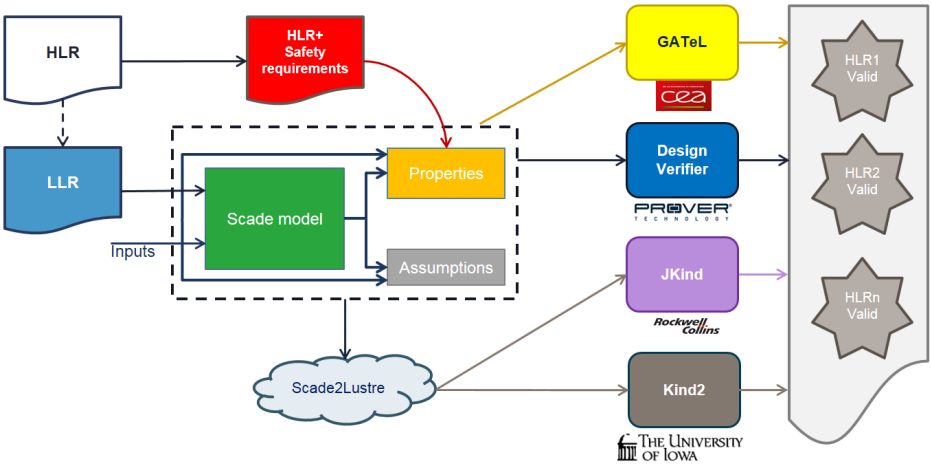


Figure 17: Verification using multiple model checkers and multiple SMT solvers

Figure 18 shows the principal blocks of our SCADE model. It contains an automaton for managing modes, a function for enabling/disabling the cruise control and a function for managing transitions. These components communicate with each other. We want to prove the correctness of the model by writing safety-properties as observers over the SCADE model.

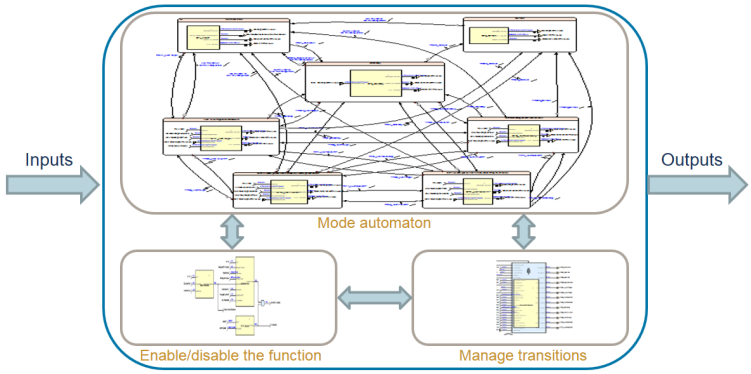


Figure 18: Cruise controller SCADE model's principal blocks

6.1.2 *Writing Formal Properties*

We formalized our properties from the safety-related requirements, and extended them to all HLR concerning the deactivation of the function. We have safety-related requirements separated from the HLRs because they are written by a safety engineer and the HLRs are written by the function designer. We want to prove the validity of all these properties i.e. no matter what happens, the cruise controller will deactivate upon the specified conditions. Some of these requirements are listed in Table 5.

REQ-01	A simple press on the *Cancel* button shall disable the cruise controller.
REQ-02	Switching off the ignition shall disable the cruise controller.
REQ-03	In order to respect the safety objectives when the brake pedal sensor is not working: a deceleration (*Decel*) under a defined threshold value ($T2$) and the brake pedal not seen pressed during 2 seconds shall turn off the function.

Table 5: System requirements used for model checking

REQ-01 and REQ-02 are typical safety properties without timers (stateless invariants) that are well handled and easy to prove with the actual model checkers.

REQ-03 uses time. When time is increased in the property and in the model, this makes the proof difficult because the number of states explodes. For our experiments, we modeled this property at three different levels:

- We name *PG* the *global property* that is checked at the bounds of the whole system (1300 lines of Lustre code and 78 nodes), see Figure 19.
- Then we keep the model of the whole system but we rewrite *PG* into *PL*, which is the same property *expressed locally* on the bounds of the node that implements the authorization function, see Figure 20.

- Finally, we *isolate* the node that implements the authorization function (320 lines of Lustre code and 3 nodes) to reduce the state-space, and call *PI* the property to be checked, which is the same as *PL*, only the context is different, see Figure 21.

The inductive model checkers that we considered implement slicing algorithms such as the COI. Roughly, the cone of influence of a property is the structural part of the design on which the property depends. Before starting an analysis, the COI is computed in order to remove the parts of the design that have no influence on the property under analysis. We decided to check the *PL* property in order to see the efficiency of the slicing algorithms, and also to use it for compositional analysis, see Section 6.1.3. It is equivalent to *PI* but has some sort of environment that can give preconditions and reduce its state space.

In the case of *PI* (Figure 21), we noticed that two things made the proof difficult. Firstly, trying to prove a property over a *long period of time*, such as 2 minutes instead of 2 seconds, takes too much memory or time for some model checkers. We call this property *PI-X* where *X* is the number of 50 ms *time steps* (for example, 2 seconds represent 40 steps). Secondly, we want to check the difficulties that a model checker would have when checking a valid property that does not match exactly what the code does. We consider two variants of *PI-X* depending on the deceleration threshold:

- *PI-X-T2*: $(\mathrm{Decel} < \mathrm{T2} \wedge \mathrm{X}) \Rightarrow$ PI is the original property;
- *PI-X-T1*: $(\mathrm{Decel} < \mathrm{T1} \wedge \mathrm{X}) \Rightarrow$ PI has a stronger precondition because $\mathrm{T1} < \mathrm{T2}$, thus it is a weaker property, which is valid when *PI-X-T2* is valid.

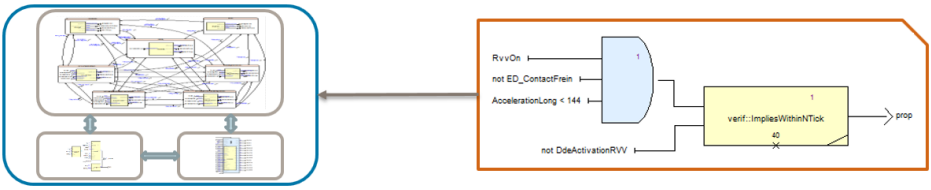


Figure 19: Property *PG-40* expressed on the bounds of the model

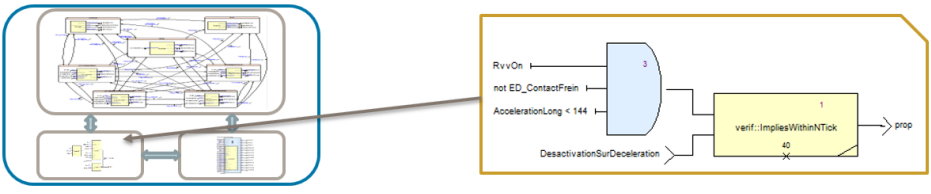


Figure 20: Property *PL-40* expressed on a sub-node

Figure 21: Property *PI-40* expressed on an isolated sub-node

Our final goal was to prove the global property *PG* (involving the entire model) directly, but we noticed that for long-running time properties it was impossible to scale. We decomposed it in two smaller properties and used a compositional approach to prove the property on the entire model. Property *PL* (expressed locally on the node implementing it) was used for compositional reasoning as discussed in the next section.

6.1.3 *Compositional Approach*

A compositional approach reduces the complexity of the verification of a big model by dividing it into 2 or more components. We divided our model into two components as shown in Figure 22:

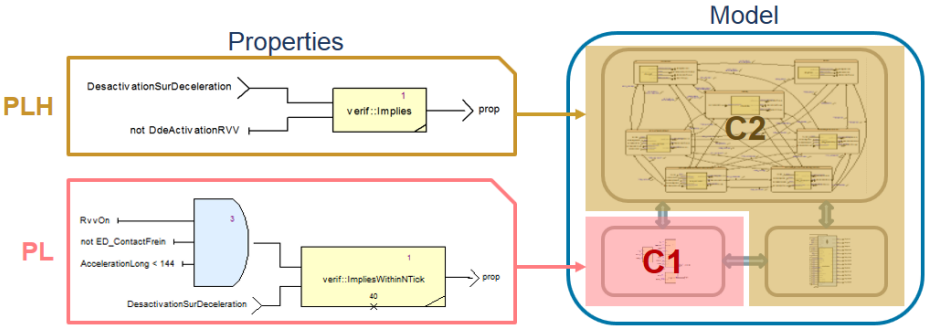


Figure 22: Compositional approach for properties PL and PLH

- *C1* (pink): The authorization function takes into account the inputs of the model and produces an intermediate result.
- *C2* (brown): The rest of the model that produces outputs which uses the intermediate result.

Then we used two properties applied on C1 and C2, that put together, are equivalent to the global property *PG*:

- *PL* (pink): Property *PL* is expressed locally on the node implementing the authorization function. It takes into account the inputs of the whole model and the intermediate output.

- *PLH* (brown): The local output of the authorization function is used to prove the global model output.

6.1.4 *Results Analysis*

In this section, we comment on the results obtained for *PG*, *PL* and *PI* with a small number of time steps (40, which is equivalent to 2 seconds) and with a large number of time steps (2400, equivalent to 2 minutes). We also used two different values for the properties deceleration threshold: *T1* and *T2*, where T1 < T2. Experiments were run on an Intel® Xeon® CPU E5-2609 v2 @ 2.50GHz and 64 GiB of memory with SCADE DV, KIND 2 1.1.0 and JKIND 4.0.1. We used all available SMT solvers with KIND 2 and JKIND but found that KIND 2 generally works best with Z3 (4.7.1) and JKIND with Yices2 (2.5.2). Our results listed below are obtained with these solvers. KIND 2 does not support PDR/IC3 with Yices2 and thus cannot be compared to JKIND with Yices2, which supports it. The timeout option of the model checkers was set to 2 hours (wall-clock time). The results obtained by GATeL were unsound and we do not comment further on this model checker.

6.1.4.1 *Invariant Generation is Mandatory*

Our first experiment was to disable PDR and invariant generation processes and we found that this type of time properties were not *k*-inductive even for 2-step models. We needed additional invariants to strengthen the property.

6.1.4.2 *PDR/IC3 only for Small Timers and Models*

Our results show that PDR is a good strengthening algorithm only for small numbers of time steps and small models. With the time span of properties and the size of the model, there is a combinatorial explosion.

6.1.4.3 *Threshold Impact*

The deceleration threshold *T1* affected essentially SCADE DV. We noticed that KIND 2 and JKIND had no problem with it, even if it slowed down the proof. The SMT solvers behind them have stronger theories on integers. The threshold *T2* affected essentially KIND 2, as long-running timers were impossible to prove with it.

6.1.4.4 *Subnode Property PI*

Increasing the time span from 40 to 2400 steps for the small model with property *PI* took more time with KIND 2, but resulted in a timeout for JKIND. JKIND and KIND 2 use a different implementation of the template-based invariant generation techniques described in [KGT11]. Even when the PDR process produces

the proof, it sometimes uses invariants provided by the invariant generator, we noted it *PDR+Invgen* in the results below.

6.1.4.5 *Compositional Approach with Property PL*

We used property *PL* combined with *PLH* to decompose the complex property *PG* into two simpler problems. Proving *PL* is almost equivalent to *PI* when using slicing because it eliminates the code that is not concerned by the property. JKIND was unable to prove the long time *PL* property, and KIND 2 showed that for the *T1* threshold it was possible to prove it using its invariant generator, but not for *T2*.

6.1.4.6 *Global Property PG*

Our final goal was to prove the global property *PG*, taking into account the entire model with a long-running timer (2400 steps). We encountered some difficulties with SCADE to prove it when using the *T1* threshold, and KIND 2 was unable to prove it with the *T2* threshold. JKIND was unable to prove long-running timers at all. This motivated us to try to understand these difficulties.

Table 6 and Table 7 present the results obtained with different number of time steps and thresholds. The measured time is in seconds of wall-clock time. As KIND 2 and JKIND run multiple engines such as PDR, *k*-Induction and invariant generation in parallel, we put the engine that provided the first result. The second one helped the first with a useful invariant.

	SCADE DV	**Kind2 / Z3**	**JKind / Yices2**
PI-40	2972	141.7 | PDR+Invgen	10.7 | Invgen
PI-2400	Timeout	139.6 | PDR+Invgen	Timeout
PL-40	Timeout	156.8 | Invgen	12.6 | PDR
PL-2400	Timeout	1353 | PDR+Invgen	Timeout
PG-40	Timeout	373.7 | Invgen	7064 | PDR
PG-2400	Timeout	155.2 | Invgen	Timeout

Table 6: Results using deceleration threshold *T1*

	SCADE DV	**Kind2 / Z3**	**JKind / Yices2**
PI-40	3	3.8 | Invgen	0.8 | Invgen
PI-2400	2	8.1 | Invgen	Timeout
PL-40	7	23.9 | Invgen	12.5 | PDR
PL-2400	11	Timeout	Timeout
PG-40	9	1370 | Invgen	51.2 | PDR
PG-2400	11	Timeout	Timeout

Table 7: Results using deceleration threshold *T2*

6.2 APPROACH AND CONTRIBUTION

Although our model used linear arithmetic over integers, we noticed that increasing the time span of our global property based on REQ-03, e.g. from 2 seconds to 2 minutes, made the proof with SCADE DV fail in a reasonable time (24 hours). To understand the problem, we translated the SCADE model with its properties into the Lustre language, and used open source SMT-based model checkers, putting them into debug mode to analyze the situation.

6.2.1 *SCADE to Lustre Transformation*

As SCADE has a textual language inherited from Lustre, we developed a tool based on an XSLT transformation called *SCADE2Lustre*. We used SCADE to convert our model into the SCADE textual language and then we transformed this textual representation of our model into Lustre code using our tool. As JKIND does not support SCADE automata, we rewrote our automaton in Lustre and checked using JKIND and KIND 2 that we had the same proof results as those obtained by SCADE DV.

6.2.2 *Understanding the Problem*

We analyzed our model with different numbers of time steps, different algorithms such as *k*-Induction, PDR/IC3 and invariant generation, at different levels of abstraction (properties *PG*, *PL*, *PI*). We also used different model checkers and different SMT solvers as back-ends. We found use cases with long-running timers in production models that all available model checkers were unable to prove. In order to understand the problem we decided to use and modify JKIND for its particular implementation of IVC [GGW16]. We used IVC to get the invariants that had enabled the proof. It was useful for understanding what candidates we needed to generate for the proof.

6.2.2.1 *k-Induction*

The basic idea behind *k*-Induction is to make use of invariants that are not 1-inductive. With the increase of *k*, there is a combinatorial explosion, so it can run for a very long time. It was the case for our property involving time because it was not *k*-inductive for a small *k*. This is why we needed a smarter invariant generator to help strengthen the property before *k* goes too high.

6.2.2.2 *PDR/IC3*

PDR can strengthen the property, but the number of invariants it constructs from the property explodes when the time span of the property and the size of the model increase. Because of the interval generation, most invariants are

useless for our proof and just slow down the proof process. Furthermore, these invariants appear to find relations only between variables and constants but not between multiple variables. For long-running time properties on production models, PDR suffered from the same combinatorial explosion problem as *k*-Induction.

6.2.2.3 *The* JKind *Invariant Generator*

JKind uses a template-based lemma generation, as described in [KGT11], for its invariant generation procedure. In order to obtain invariants to strengthen the proof, JKind creates a list of candidates representing literals. Four different types of candidates are generated automatically:

- **Boolean candidates**: all boolean system variables and their negations e.g. *a* and *not a* where *a* is a boolean.
- **Init candidates**: integer variables are compared with $\geqslant$ and $\leqslant$ operators to their initial values e.g. $(i \geqslant 0), (i \leqslant 0)$ where o is the initial value of *i*.
- **Subrange candidates**: variables of an integer subrange type are compared for equality to all the values in the subrange, e.g. $(s = 0), (s = 1), (s = 2)$ for $s \in [0..2]$.
- **Enum candidates**: variables of an enum type are compared for equality to all the values of the enum.

The invariant generator checks all propositional formulae (with boolean operators) involving these literals, whether they are invariants or not. The number of formulae grows exponentially with the number of literals. To avoid this combinatorial explosion, JKind reduces its candidates to those listed above and no relational candidates (explained in Section 6.2.3) are considered.

All these candidates were not strong enough for proving our long-running timer properties.

6.2.3 *Contribution*

JKind uses multiple cooperative engines in parallel, including *k*-Induction, PDR and template-based invariant generation. We worked on the improvement of the invariant generation. We noticed that our property used a constant value for the number of time steps and the code also used a constant for it. The same was true for other clauses in the property. We needed invariants that could provide information about the relation (essentially a comparison) between constants and variables of the property and constants and variables of the model. In order to find relations between the property and the model, we propose two new additional categories of relational candidates (atoms):

- **INT** $\times$ **INT**: for all integer variables in the model and the property, add a comparison relation with the $\geqslant$ operator, e.g. $Variable1 \geqslant Variable2$
- **INT** $\times$ **CONST**: for all integer variables and constants in the model and the property, add comparison relations with the $\geqslant$ and $\leqslant$ operators, e.g. $Variable1 \geqslant Constant1$; $Variable1 \leqslant Constant1$

We implemented this new invariant generation algorithm in JKind and applied it to a sub-node of our model with a large number of time steps (property PI-2400). We were able to prove the property within a few seconds although it was impossible to prove before. Once we could prove the property, it was possible to use IVC to find the invariant that had enabled the proof. We used it to understand what were the most useful candidates we needed to generate for the proof.

Next, we wanted to prove the entire model with a long time property (property PG-2400). With our new invariant generator, we had the needed candidates but for the entire model their number was too big. We noticed also that numerous candidates did not make sense, e.g. when comparing a variable about speed to a constant about deceleration, or comparing counters with non counter elements. To get interesting invariants, we propose to use the physical type (speed, deceleration, counter, etc.) of the variables and the constants, and to keep only candidates that compare elements of the same physical type. We explain this in details in the next section.

6.2.3.1 *Physical types methodology*

A physical quantity is a physical property that can be quantified by measurement. A physical quantity can be expressed as the combination of a number and a unit. For example, in the physical world, we measure the quantity of speed using the unit $\mathrm{m\,s^{-1}}$ and its derivations. The same is true for other physical quantities. In the automotive and other industries, most of the external interfaces of a function represent a physical quantity (speed, deceleration, battery voltage, etc.) and has a physical unit.

At the code level, information about units is lost and only numbers are present. Fortunately, at the software architecture design level, the physical units are present. As most of the software is designed using model-based design tools, this information can be used for model verification. We propose to use this semantic information to have a deeper understanding of the variables and to generate less invariant candidates while increasing their usefulness. As a methodology, we propose the introduction of physical types at the model level for tools such as Simulink or SCADE. Instead of using a base representation types such as *int*, we declare a type for each physical quantity, e.g. *tSpeed*, *tDeceleration*, *tVoltage*, *tCounter* etc. Then all the variables and constants are typed according to their appropriate physical type. Actually, these new types are just aliases of the base representation types, but they carry more semantics for our

algorithm. Thus we can recognize data of the same physical type and reason on them using the appropriate relations, see Figure 23.

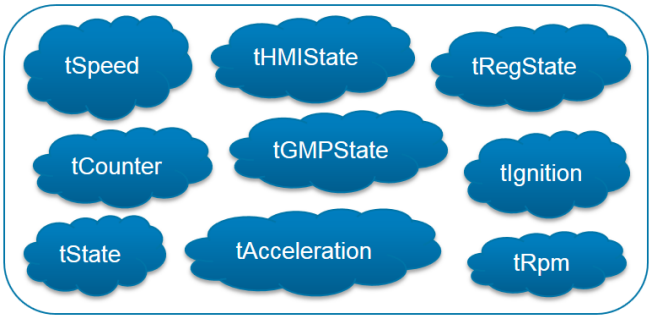


Figure 23: Constants and variables partitioned by their physical types

For our use case, we defined physical types in SCADE during the design phase. Then, we used them for all the constants and variables of the model. This takes little when done during the design stage. As types are immediately evaluated and shown on the SCADE model, it also gave a better readability during the review of the model. Once the model was validated, we converted it into Lustre code using our *SCADE2Lustre* converter, which preserves types.

6.2.3.2 *Timers patterns*

We wanted to optimize further our algorithm, and to push only the most relevant candidates. By analyzing the useful candidates from the minimal invariants used for the proof (based on IVC), we noticed that all the variables that were useful were assigned a previous value (they correspond to state variables). We propose to eliminate variables that are not assigned a previous value, which correspond to combinatorial variables for which SMT solvers are very efficient, so invariant generation is not necessary. An example of state and combinatorial variables is shown in Figure 24.

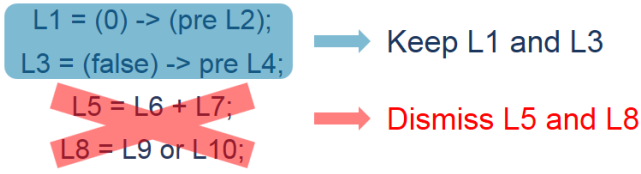


Figure 24: Variables encoding a state are kept. The others are dismissed.

6.2.3.3 *Implementation in* JKind

We introduced our improvement on a GitHub branch of JKind[1] called "invgen-timers". We modified JKind to be able to preserve the original Lustre types because they were lost after inlining. We introduced an option "-inv_gen_level" proposing more and more candidates when the level increases:

- **Level 0**: Default JKind level before our improvements
- **Level 1**: Use the physical types methodology with INT × INT and INT × CONST relational candidates, restricted to state variables (variables with an assignment of a value from a previous state). This level performs best if our physical types methodology is applied.
- **Level 2**: Uses INT × INT and INT × CONST relational candidates no matter their type, restricted to state-variables. This level works for models that do not use physical semantic types.
- **Level 3**: Uses INT × INT and INT × CONST relational candidates including state-variables and combinatorial variables. This level can be used if the other levels do not provide the necessary invariants.

The idea behind this new option is to provide different amount of invariants so that the user can begin with the lowest level. If the property cannot be proved with it, the next level could be used until the property is proved. Beginning with the highest level may degrade the performance for properties where a lower level would be sufficient.

6.3 RESULTS AND BENCHMARKS

In this section, we examine the results obtained by our invariant generation algorithm and methodology using physical types compared to the results obtained with the official versions of JKind and Kind 2. We also used JKind's and Kind's benchmarks to find use cases about timers and compare performances. Finally, we asked Collins Aerospace for use cases about timers and found that some properties on production models, which were not proved before with JKind, were now proved within a few seconds thanks to our improvements.

6.3.1 *Our Use Cases*

We summarize here the results obtained with our cruise controller model.

In Table 8 and Table 9 we present the results in seconds obtained using our methodology (JKind new) based on physical types compared to the previous results (Kind 2 and JKind official versions).

1 JKind on GitHub: https://github.com/agacek/jkind

	SCADE DV	Kind 2	JKind (official)	JKind (new)
PI-40	2972	141.7	10.7	0.2 | Invgen
PI-2400	Timeout	139.6	Timeout	0.2 | Invgen
PL-40	Timeout	156.8	12.6	4.7 | Invgen
PL-2400	Timeout	1353	Timeout	5.3 | Invgen
PG-40	Timeout	373.7	7064	4.3 | Invgen
PG-2400	Timeout	155.2	Timeout	4.8 | Invgen

Table 8: Results using our new invgen and types for threshold *T1*

	SCADE DV	Kind2	JKind (official)	JKind (new)
PI-40	3	3.8	0.8	0.1 | Invgen
PI-2400	2	8.1	Timeout	0.1 | Invgen
PL-40	7	23.9	12.5	4.2 | Invgen
PL-2400	11	Timeout	Timeout	4.2 | Invgen
PG-40	9	1370	51.2	4.2 | Invgen
PG-2400	11	Timeout	Timeout	4.4 | Invgen

Table 9: Results using our new invgen and types for threshold *T2*

We notice that JKind new (Level 1) outperforms the official versions of the other model checkers for both deceleration threshold values T1 and T2.

6.3.2 JKind *Benchmark*

JKind provides with its source files, 56 Lustre programs with properties to be proved. We used it to compare the performance of our different algorithms with the official one. We did not find long-running timers in this benchmark and the new levels of invariant generation we introduced in JKind did not bring better results. We suppose that this benchmark was tuned for the current JKind version, as there are no unsolvable problems in it (everything can be proved or invalidated).

6.3.3 *Kind Benchmark*

We also used a suite of 1047 Lustre programs developed as a benchmark for Kind [HT08]. Most of them were very small and not containing timers. Their properties were proven in less than a second. However, we found some programs that were using timers. We present their results in Table 10 using our implementation of the three different levels of invariant generation (L1, L2 and L3), compared to the JKind and Kind 2 official versions. The timeout was set to 10 minutes, Z3 was used with Kind 2 and Yices2 with JKind.

The full names of these programs are:

- P1: DRAGON_11.lus
- P2: DRAGON_11_e1_2450.lus

Program	Kind2	JKind	JKind-L1	JKind-L2	JKind-L3
P1	Timeout	13.2	4.3	9.2	6.6
P2	Timeout	9.9	7.9	10.3	4
P3	Timeout	13.2	9.4	7.6	13.6
P4	Timeout	6.5	8.7	10.1	3.8
P5	Timeout	9.1	58.7	6.9	5.6
P6	Timeout	Timeout	Timeout	1.7	1.3
P7	Timeout	Timeout	1.8	1.4	Timeout
P8	19.3	50.4	6.3	4.4	7.7
P9	0.4	1.3	1.3	0.2	0.2

Table 10: Results using our new invgen on Kind benchmark

- P3: DRAGON_11_e1_2450_e1_5887.lus
- P4: DRAGON_11_e1_2450_e2_1483.lus
- P5: DRAGON_11_e2_5396_e3_282.lus
- P6: durationThm_3_e3_442_e6_113.lus
- P7: durationThm_3_e7_334_e8_369.lus
- P8: microwave05.lus
- P9: twisted_counters.lus

This benchmark does not use physical types. All the variables and constants are of type integer, real or boolean. Our level 1 invariant generator is more suitable when physical types are used. However, we can see that levels 2 and 3 performed well on these programs containing counters.

6.3.4 *Collins Aerospace Use Cases*

At Collins Aerospace, Lustre is used as an intermediate language to make formal proofs of high-level properties. Some models have properties with long-running timers. First, they provided us with a representative version of their production model with a property using 6000 time steps that was impossible to prove before. We proved it in a few seconds. Then we shared our new version of JKind with them so that they try it on their internal production models that they could not share with us. They told us that it proved in a few seconds properties that were not proved before.

6.4 CONCLUSIONS

In this chapter, we proposed an algorithm that brings an improvement to invariant generation, enabling the automatic proof of properties involving long-running timers, which are present in most embedded software. This algorithm

consists in injecting new relational invariant candidates to enrich the invariant generation. However, if too few candidates do not allow concluding, too many candidates can slow down the proof and lead to timeouts. That is why we propose a new methodology using physical types (speed, deceleration, etc.), which restricts the number of candidates to only those that make sense (e.g. deceleration variables compared to deceleration constants or speed variables compared to other speed variables) and may therefore be useful for the proof. Our algorithm is applicable to all forms of inductive model checkers. We have implemented it as part of the open source model checker JKIND. We have shown that it outperforms the official versions of JKIND, KIND 2 and SCADE DV on benchmarks and on several industrial use cases.

We also want to show a way to improve the state of the art in formal methods. When using model checking in the industry, we do not have access to advanced academic model checkers and solvers because industrial companies essentially use black-box tools that cannot be put in a debug mode or modified. If the proof is not possible, it is very difficult to understand why. We used the Lustre language as an intermediate language between the black-box tools and the open source model checkers. This allowed us to understand what was missing to automatically strengthen the proofs, and to implement our new algorithm in JKIND as a proof of concept.

The presented method for generating invariants is working fine when the counter in the property and the counter in the model evolve at the same rate and sequentially. This is the most common case for industrial models. There can exist models that increment counters at different rates. For the moment, these models need additional invariants provided by the user as assertions.

7

COVERAGE MEASURE BASED ON MUTATION AND MODEL CHECKING

What saves a man is to take a step. Then another step.

— Antoine de Saint-Exupéry

When using formal verification on Simulink or SCADE models, an important question about their certification is how well the specified properties cover the entire model. A method using unsat cores and inductive model checking called IVC has been recently proposed within modern SMT-based model checkers such as JKind. The IVC algorithm determines a minimal set of model elements necessary to establish a proof and gives back the traceability to the design elements (lines of code) necessary for the proof. These metrics are interesting but are rather coarse grain for certification purposes.

The problem is that even if the model checker has proved all the properties to be valid, we cannot answer the question about whether our model contains features that are not covered by the properties. Unlike testing, where we can follow the execution trace, the proof process uses the whole model, but many parts of it may not be necessary to prove the properties. This problem has been studied using the following approaches: mutation proof [CKKV01, Cla07, GKD07, SAP+05] and IVC [BGWC18, BCB18, Ber19, GWGH19].

The mutation approach shown in Figure 25 consists in mutating a model for which safety properties were proved valid, and trying to prove the same properties on the mutated models (*mutants*) again. If they are proved valid (the mutant has *survived*), the mutant reveals a part of the model that is not covered by the properties. There can also be dead code that will never be accessed. The algorithms used to compute coverage in the aforementioned papers can under-approximate which parts of the model are necessary to prove the properties and tend to be computationally very expensive because there are many mutated models to be verified.

The algorithms using IVC proposed in the articles cited above are based on the *Unsatisfiable Core* support built into current SMT solvers. They can efficiently generate over-approximated inductive validity cores or exhaustively compute minimal ones. The inductive validity cores represent the minimal sets of model

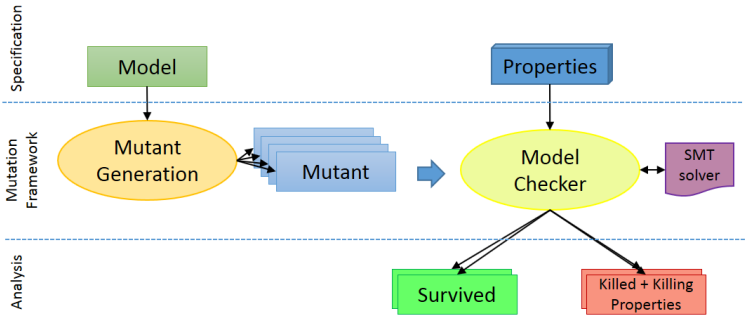


Figure 25: Mutation proof framework

elements necessary to construct inductive proofs. The authors show that calculating IVC is more efficient than classical state-of-the-art mutation. Calculating IVC gives the coverage of properties in terms of lines of code of the model, which is more precise than a simple syntactic slicing, but does not look inside the lines of code and therefore does not consider the coverage of elementary operations inside an equation.

We propose to go further in the precision of the coverage and zoom into the lines of code. Actually, a property can be covered by a line of code but inside the line there may still be some code that has no impact on the property. We argue that it is inside the lines of code that some subtle bugs can still subsist, and it is useful to uncover them. We use mutation to mutate some operators of the model, and symbolic model checking combined with induction-based techniques (*k*-induction, PDR/IC3), and take advantage of the incremental query capabilities of modern SMT solvers (see Section 4.1.2.5). We observed that mutation-based coverage for model checking is no longer out of reach, and this technique scales with our industrial use cases. We implemented this algorithm in the JKind open-source model checker, which is based on the Lustre formal language. Lustre is used as base language for SCADE, so we could transform a SCADE model into Lustre. Simulink can also be transformed into Lustre using the CoCoSim framework developed at NASA Ames[1].

7.1 PRELIMINARIES

In this section, we introduce the architecture of the industrial inductive model checker JKind [GBW+18] which is representative of other model checkers such as Kind 2 and PKind.

7.1.1 *The* JKind *Model Checker*

JKind is an open-source industrial infinite-state model checker for safety properties. Models and properties are written in Lustre, a synchronous data-flow

1 CoCoSim: https://ti.arc.nasa.gov/tech/rse/research/cocosim

language, using theories of real and integer arithmetic. JKind uses SMT-solvers (SMTInterpol, Z3, Yices, CVC4, MathSAT) to prove or falsify the properties. It is structured as several parallel *engines* that cooperate to prove properties. Some engines are directly responsible for proving properties, some contribute to that effort by generating invariants, and others are for post-processing proofs or counterexample results. Each engine can be enabled or disabled separately. The architecture of JKind is shown in Figure 26. At the center of this architecture the **Director** allows any engine to broadcast information (invariants, valid and invalid properties) to the other engines.

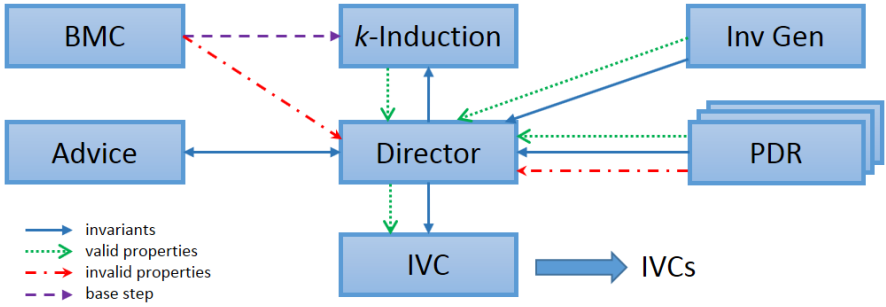


Figure 26: The JKind model checker architecture

The **BMC** engine performs a standard iterative unrolling of the transition relation to find counterexamples or to serve as the base case of *k*-induction. The BMC engine guarantees that any counterexample it finds is minimal in the number of steps from the initial state. The ***k*-Induction** engine performs the inductive step of *k*-induction, possibly using invariants generated by other engines. The **Invariant Generation** engine uses a template-based invariant generation technique using its own *k*-induction loop. The **PDR** engine performs property directed reachability using the implicit abstraction technique [CGMT13]. Unlike BMC and *k*-induction, each property is handled separately by a different PDR sub-engine. The **Advice** engine saves invariants from previous runs of JKind and reuses them for new proofs to decrease the verification time.

A great effort was done in JKind on the post-processing of the results. We can cite the *Smoothing* counterexamples feature based on MaxSat which minimizes the number of changes to input variables. The other important post-processing feature is IVC.

For a proven property, an IVC is a subset of Lustre equations from the input model for which the property still holds. An IVC is *minimal* when no equation can be removed without breaking the provability. Depending on the model and property, there may exist several IVC with different sizes. A *minimum* IVC has the smallest number of equations, and is not necessarily unique. Computing a minimum IVC is more difficult than computing any IVC, because it involves an exhaustive search. The IVC engine uses a heuristic algorithm to efficiently produce minimal IVC but not minimum ones. As a side-effect, the IVC algo-

rithm also minimizes the set of invariants used to prove a property, and shares this reduced set with other engines (notably the Advice engine).

7.1.2 *IVC Formalizations*

In this section we re-use and adapt the formalization of IVC given by Ghassabani et al. in [GGW+17] to compare IVC to our mutation proof using similar definitions of coverage.

7.1.2.1 *Models, Requirements and Provability.*

Given a state space U, a transition system (I, T) consists of an initial state predicate $I : U \to \mathit{bool}$ and a transition step predicate $T : U \times U \to \mathit{bool}$. A safety property $P : U \to \mathit{bool}$ is a state predicate that holds on a transition system (I, T) when it satisfies the following formulas:

$$\forall u.\ I(u) \Rightarrow P(u)$$
$$\forall u, u'.\ P(u) \wedge T(u, u') \Rightarrow P(u')$$

When this is the case, we write $(I, T) \vdash P$.

Coming from the Lustre model consisting of a set of equations $\{eq_1 \dots eq_n\}$, the transition relation T has the structure of a top-level conjunction $T = t_1 \wedge \cdots \wedge t_n$ where each t_i is an equality corresponding to eq_i. By further abuse of notation, T is identified with the set of its top-level equalities. When an equation is removed from the Lustre model, an equality t_i is removed from T and the transition relation becomes $T \setminus \{t_i\}$.

Definition 5 (*Inductive Validity Core (IVC)*)

> $S \subseteq T$ *for* $(I, T) \vdash P$ *is an Inductive Validity Core, iff* $(I, S) \vdash P \wedge \forall t_i \in S.\ (I, S \setminus \{t_i\}) \nvdash P$.

As defined here, we are only interested in minimal sets that satisfy a property P. Note that given $(I, T) \vdash P$, P always has at least one *IVC*, which is not necessarily unique. For example, consider 2 boolean variables a and b initialized to true, *i.e.* $I = a \wedge b$, and assigned true at each step $T = (t_1 : a = \mathit{true}) \wedge (t_2 : b = \mathit{true})$. If $P = a \vee b$ then both $\{t_1\}$ and $\{t_2\}$ are *IVC*s. We note $\mathrm{AIVC}(P)$ the set of all *IVC*s of P. Computing the AIVC for each property, one gets a clear picture of all the model elements constrained by the property. The set AIVC for all properties demonstrates a complete mapping from the requirements to the design elements, which is called *complete traceability* [MWGH16].

7.1.2.2 *Property and Model Coverage.*

The article by Ghassabani et al. [GGW+17] defines the two following metrics of coverage.

Definition 6 (*MayCov*)

$t_i \in T$ *is covered by* P iff $t_i \in \text{MAY-COV}(P)$, *where* $\text{MAY-COV}(P) = \{t_i \mid \exists S \in AIVC(P) \cdot t_i \in S\}$.

Definition 7 (*MustCov*)

$t_i \in T$ *is covered by* P iff $t_i \in \text{MUST-COV}(P)$, *where* $\text{MUST-COV}(P) = \{t_i \mid \forall S \in AIVC(P) \cdot t_i \in S\}$.

This categorization of coverage helps to identify the role and relevance of each design element in satisfying a property. MUST-COV specifies the parts of the model that are absolutely necessary for the property satisfaction. Any change to these parts will affect the provability of the property. On the other hand, MAY-COV parts are relevant to the proof but may be modified without affecting the satisfaction of P. The MAY-COV heuristic leads to higher coverage scores, because $\text{MUST-COV}(P) \subseteq \text{MAY-COV}(P)$.

In JKIND, the IVC engine computes one IVC and avoids exploring all possible ones. Therefore, it partially computes the $\text{MAY-COV}(P)$ and it does not handle $\text{MUST-COV}(P)$.

7.1.2.3 *Mutation.*

A mutator is a function that mutates any transition predicate T to a set of mutants $\{T^1_{mut}, \ldots, T^m_{mut}\}$, where each mutant T^i_{mut} is obtained by applying a small change to T.

A very simple mutator is the one that simply removes an equality t_i from T, which amounts to removing the corresponding line of code from the Lustre model. In our framework, we call this basic mutator `eq_remove` (see Section 7.3). The authors of [GWG17] only consider this simple mutator and define the corresponding coverage as follows:

Definition 8 (*Mutation Coverage*)

$t_i \in T$ *is covered by property* P iff $t_i \in \text{MUT-COV}(P)$, *where* $\text{MUT-COV}(P) = \{t_i \mid (I, T) \vdash P \wedge (I, T \setminus \{t_i\}) \nvdash P\}$.

An immediate corollary proved in [GWG17] states that if an equation is covered by such a mutation, it is also covered by all IVC and conversely:

Lemma 9 (*MutCov and MustCov*)

$\text{MUT-COV}(P) = \text{MUST-COV}(P)$.

The MUT-COV metrics can be generalized to more advanced mutators. In Section 7.3, we will show how the MUT-COV metrics can be improved to give a very precise coverage inside each t_i detected within MUST-COV or MAY-COV.

7.2 MODEL COVERAGE TECHNIQUES

An important question for the certification of safety-critical systems is whether the requirements and tests are covering the implementation. For example, in ISO 26262, which is the functional safety standard for road vehicles, tests are derived from requirements. An argumentation of why the performed tests give sufficient coverage shall be provided. As the critical level increases, a more rigorous method for test coverage (statement, branch, MC/DC) is required. If complete coverage is not achieved, an analysis is performed to decide whether additional tests or/and requirements are needed to increase coverage. DO-178C with its supplement DO-333 (Formal Methods) go further in offering the possibility to use formal methods in replacement of all structural coverage objectives (including heavyweight MC/DC), but arguments showing that coverage is achieved by the formal proof should then be provided, see Table 11.

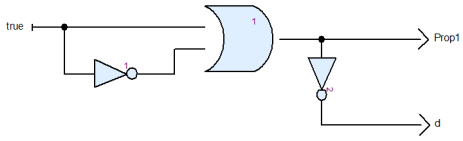
DO-178C Table A-7 Objective	DO-333 Table FM.A-7 Objective
1. Test procedures are correct.	FM1. Formal analysis cases and procedures are correct.
2. Test results are correct and discrepancies explained.	FM2. Formal analysis results are correct and discrepancies explained.
3. Test coverage of High Level Requirements (HLRs) is achieved.	FM3. Coverage of HLRs is achieved.
4. Test coverage of Low Level Requirements (LLRs) is achieved.	FM4. Coverage of LLRs is achieved.
5. Test coverage of software structure (modified condition/decision coverage) is achieved.	FM5 – FM8. Verification coverage of software structure is achieved.
6. Test coverage of software structure (decision coverage) is achieved.	(A single objective that replaces the four structural coverage objectives in DO-178C)
7. Test coverage of software structure (statement coverage) is achieved.	
8. Test coverage of software structure (data coupling and control coupling) is achieved.	
9. A verification of additional code, that cannot be traced to source code, is achieved.	FM9. Verification of property preservation between source and object code.
N/A	FM10. Formal method is correctly defined, justified, and appropriate.

Table 11: DO-333 accepts replacing MC/DC coverage by formal proof coverage

In this section, we present different techniques for model coverage, going progressively from coarse-grained coverage to fine-grained coverage. We consider the application of these techniques to the domain of inductive symbolic model checking. We propose to use mutation-based proof, taking advantage of the possibility to request SMT solvers in an incremental way, in order to look inside the operators in a way MC/DC does for testing. We show that the performance of this technique is equivalent to IVC and therefore quite faster than state of the art mutation-based methods. To the best of our knowledge, this technique has never been studied. The closest related work on mutation-based proof does not use inductive model checking for software verification nor incremental SMT

solving. The work of Chockler et al. [CKKV01] presents an algorithm to re-use previously computed inductive invariants and counterexamples to identify the parts of a hardware system that are covered by a property. In [Cla07], Claessen presents a coverage analysis based on Linear Temporal Logic (LTL) that gives the possibility to have underconstrained properties. In [GKD07], the authors present an approach to estimate coverage in BMC. They generate coverage properties for each important signal for hardware verification purposes. Finally, in [SAP+05], Sayantan et al. present a method for determining the coverage of a formal LTL specification against a high-level fault model for hardware verification.

7.2.1 *Simple Running Example*




```
1  node demo () returns (Prop1: bool; d: bool);
2  var
3      L1, L2, L3, L4: bool;
4  let
5      L1 = L2 or L3;
6      L2 = true;
7      L3 = not L2;
8      L4 = not L1;
9      Prop1 = L1;
10     d = L4;
11     --%PROPERTY Prop1;
12 tel
```

Figure 27: A simple running example in Lustre

We use a simple running example to illustrate the difference between slicing, IVC and mutation proof. Consider the SCADE model shown both graphically and textually in Figure 27. The property `Prop1` we want to prove concerns the output of an `OR` block which takes a constant input equal to true and its negation. Obviously this property is always true. The Lustre code is obtained by using the SCADE option "Convert to textual" and we just add the comment on line 11 to tell JKind which output represents our safety property to be proved (invariant that shall always be true).

7.2.2 *Slicing*

The backward static slicing (or slicing for short) is a coarse-grained technique that allows to remove the parts of the code that do not affect the properties to be

proved. It works by simply calculating the dependency graph for the variables used in the properties. Modern inductive model checkers use slicing to reduce the size of the queries sent to the SAT/SMT solver. It is interesting to see how much of the code is removed and to check if we really need this code or if our properties are simply not complete enough. After slicing, `d` and `L4` are removed and we obtain the lines:

```
1  L1 = L2 or L3;
2  L2 = true;
3  L3 = not L2;
4  Prop1 = L1;
```

7.2.3 *Inductive Validity Cores*

IVC are much smaller and more precise than static slicing. For our short example, the IVC engine will either remove the equation of `L3` because `L1` does not depend on it since `L2` is true, or it will keep the equation of `L3` and remove the equation of `L2` since the equation of `L1` is a tautology when we consider the equation of `L3`. When running IVC on `Prop1`, it turns out that we obtain the first inductive validity core: `{L1, L2}`

```
1  L1 = L2 or L3;
2  L2 = true;
```

7.2.4 *A Simple Mutator for* Must-Cov*: Equation remover*

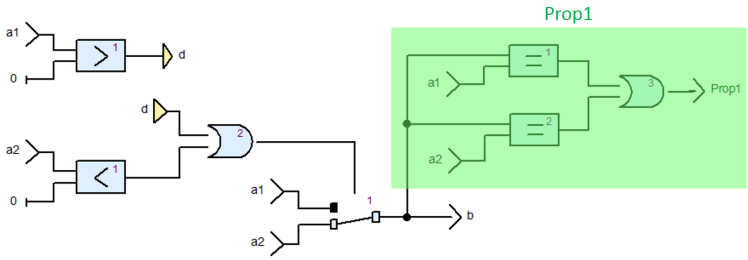
We want to go further than IVC, so we propose to use a simple mutator called "equation remover" which removes equations one by one and replays the proof process in an incremental way (using the SMT-LIB [BST+10] *pop* and *push* commands). Our equation remover does not affect the properties because we want to mutate only the model and not the specification. If after removing an equation the properties are still proved (surviving mutant), it means that the removed equation has no impact on the proof. If the properties do not hold anymore (killed mutant), this means that the removed equation is essential for the proof. This mutator computes the *minimum* core defined as Must-Cov in Section 7.1, whereas IVC is working in May-Cov mode. Using this technique, we obtain that only the equation of `L1` is essential for any proof of `Prop1` :

```
1  L1 = L2 or L3;
```

7.2.5 *Using Other Mutators for Deep Coverage*

We propose to add other mutation operators to zoom inside a line of code/equation and see what is covered by the properties. We explain these operators

in detail in Section 7.3. For the moment, we give an example to see the difference between mutation and IVC. Our example is shown in Figure 28.




```
1  node demo2 (a1, a2: int)
2    returns (Prop1: bool; b: int);
3  var
4    d: bool;
5  let
6    d = (a1 > 0);
7    b = if d or (a2 < 0) then a1 else a2;
8    Prop1 = (b = a1 or b = a2);
9    --%PROPERTY Prop1;
10 tel;
```

Figure 28: Example of inlined code and *if-then-else* operator mutations

This model takes two inputs `a1` and `a2`, and depending on whether their value is positive or negative, `a1` or `a2` is assigned to the output `b`. We have a property `Prop1` specifying that the output `b` should take the value of `a1` or `a2`. If slicing is applied to this model, it will remove nothing because `Prop1` depends statically on the entire model. However, applying IVC tells us that we should only keep `b` to cover our property `Prop1`. It is more precise than slicing because `d` is not necessary to prove that property (`b` is always equal to `a1` or `a2`).

Now, let us apply some mutations such as: replacing the condition of the *if* statement by *true* or *false*, replacing *or* by *xor*, replacing $>$ by $<$ etc. This leads to 22 possible mutations.

For `Prop1` we have 5 mutants killed out of 22. If we want to cover 100% of the code, we need to kill all mutants. To achieve this coverage, we need to strengthen our properties. We add a second property: `Prop2 = ((a1 > 0) => b = a1)`. At this stage IVC covers 100% of the model as `d` is now necessary to `Prop2`. However, only 14 mutants are killed out of 22 (Figure 29). For example, if the condition of the *if* statement at line 7 (Figure 28) is replaced by *true*, *Prop1* and *Prop2* are proved valid. This means that the condition has no impact on these properties. Let us add a third property: `Prop3 = ((a2 < 0) => b = a1)`. This time, we kill 16 mutants out of 22. Finally, we need a fourth property: `Prop4 = (((a1 <= 0) and (a2 >= 0)) => b = a2)` to kill all 22 mutants.

```
INDUCTIVE VALIDITY CORE: b, d
++++++++++++++++++++++++++++++++++++++++++++++++++
MUTATION:
KILLED    at 6:3     equal_false            by [Prop2]
KILLED    at 6:3     equation_remove        by [Prop2]
KILLED    at 6:3     init_false             by [Prop2]
KILLED    at 6:11    g2l                    by [Prop2]
KILLED    at 6:13    (const int 0 -> 1)     by [Prop2]
KILLED    at 7:3     init_-1                by [Prop1, Prop2]
KILLED    at 7:3     equal_5                by [Prop1, Prop2]
KILLED    at 7:3     equal_-2               by [Prop1, Prop2]
KILLED    at 7:3     equation_remove        by [Prop1, Prop2]
KILLED    at 7:3     init_5                 by [Prop1, Prop2]
KILLED    at 7:7     ifelsethen             by [Prop2]
KILLED    at 7:7     ifelse                 by [Prop2]
KILLED    at 7:12    or2right               by [Prop2]
KILLED    at 7:12    or2xor                 by [Prop2]
SURVIVED  at 6:3     init_true
SURVIVED  at 6:3     equal_true
SURVIVED  at 6:11    g2ge
SURVIVED  at 7:7     ifthen
SURVIVED  at 7:12    or2left
SURVIVED  at 7:19    l2g
SURVIVED  at 7:19    l2le
SURVIVED  at 7:21    (const int 0 -> 1)
```

Figure 29: IVC and Mutation proof results on *demo2* for properties *Prop1* and *Prop2*

7.3 FROM MUTATION TESTING TO MUTATION PROOF

Mutation testing is used to evaluate the quality of a test suite that is a set of test cases. It consists in modifying the program under test in small ways. Each mutated version of the program is called a *mutant* and test cases are replayed on it to detect whether its behavior is different from the behavior of the original version. This process is called 'killing the mutant'. The more mutants are killed, the better are the test cases. The quality of a test suite is measured as the percentage of killed mutants. Mutants that are left can be killed by specifying additional test cases or justified as equivalent to the original program. *Mutators* are mutation operators used to generate mutants, and they tend to mimic standard programming errors. A mutation builds a mutant by applying a mutator on some position in the code. Taking ideas from mutation testing, we developed a *mutation proof* framework for standard inductive model checking using incremental SMT solving. In this section, we present our mutators and describe our mutation proof algorithm.

7.3.1 *Mutators*

Our mutators directly modify the Lustre code. We implemented classical mutators, but more advanced ones may be easily added to our framework. We present our mutators in Table 12.

Our first category of mutators are the *boolean mutators*. For example, the `and2or` mutator transforms `a AND b` into `a OR b`. Then we have *relational mu-*

Mutator	Description
or2xor	OR is mutated to XOR
xor2implies	XOR is mutated to $\Longrightarrow$
implies2and	$\Longrightarrow$ is mutated to AND
and2or	AND is mutated to OR
or2left	X OR Y is mutated to X
or2right	X OR Y is mutated to Y
and2left	X AND Y is mutated to X
and2right	X AND Y is mutated to Y
rm_not	NOT is removed
eq2neq	$=$ is mutated to $\neq$
neq2eq	$\neq$ is mutated to $=$
g2ge	$>$ is mutated to $\geqslant$
ge2g	$\geqslant$ is mutated to $>$
l2le	$<$ is mutated to $\leqslant$
le2l	$\leqslant$ is mutated to $<$
g2l	$>$ is mutated to $<$
l2g	$<$ is mutated to $>$
ge2le	$\geqslant$ is mutated to $\leqslant$
le2ge	$\leqslant$ is mutated to $\geqslant$
plus2minus	$+$ is mutated to $-$
minus2plus	$-$ is mutated to $+$
rm_minus	$-$ is removed
ifthen	IF condition is replaced by TRUE
ifelse	IF condition is replaced by FALSE
ifelsethen	THEN and ELSE statements are reversed
ConstantMutator	Constant is replaced by 1
eq_remove	Removes an entire equation/line of code

Table 12: Mutators for deep coverage measurement

tators such as ge2le, which transforms a '$\geqslant$' operator into '$\leqslant$'. We also have some *arithmetic mutators* such as plus2minus, which replaces '$+$' by '$-$'. *Branching mutators* act on *if-then-else* statements replacing the condition by TRUE or FALSE or reversing the THEN and ELSE statements. Finally, we have the *constant mutator* that replaces all constants by 1, and the *equation remover mutator* that removes an entire line of code as seen before.

7.3.2 *Our Contribution: Mutation Proof Algorithm*

Our contribution is the mutation proof algorithm that can be applied to modern inductive model checkers. It takes as input the proved properties and the invariants found during the proof process. It uses BMC and k-induction to retry the proof on mutants. Then, it returns a verdict: *KILLED* (proof fails with a counterexample), *SURVIVED* (proof succeeds), or *UNKNOWN* (proof fails with no counterexample). Our quality metrics is the ratio of killed mutants over the total number of mutants. The more mutants are killed, the better is the

quality of the specification, because the better is the coverage of the model by the properties in the specification.

Algorithm 1: Mutation proof algorithm

```
input : M, P
output: report
1  Prove P : {P_0, P_1 ...} on M
2  Invs ← invariants from the proof of P on M
3  k_proof ← maximum k-depth for proving P on M
4
5  foreach mutation LCM do
6      M_mut ← MUTATE(M, LCM)
7      if BMC((M_mut, ∅, ∅), P, k_proof) = SAT then
8          M_SAT ← getModel()
9          report += KILLED(mut:LCM, KillingProps:{P_i ∈ P | M_SAT ⊨ ¬P_i})
10     else
11         SI ← FilterInvs(Invs, M_mut)
12         UP ← ∅
13         SP ← P
14         while KIND((M_mut, SI, ∅), SP, k_proof) = SAT do
15             M_SAT ← getModel()
16             UP = UP ∪ {P_i ∈ SP | M_SAT ⊨ ¬P_i}
17             SP = P \ UP
18         if SP = P then
19             report += SURVIVED(mut:LCM)
20         else
21             if BMC((M_mut, SI, SP), UP, k_kill) = SAT then
22                 M_SAT ← getModel()
23                 report += KILLED(mut:LCM, KillingProps:{P_i ∈ UP|M_SAT ⊨ ¬P_i})
24             else
25                 report += UNKNOWN(mut:LCM, SurvivingProps:SP)
```

Before describing our algorithm, let us define its variables and functions: P are the specification Properties, M is the original Model, M_{mut} is the current mutated Model (Mutant), LCM represents a mutation in the form Line:Column of code and Mutator, function $MUTATE(M, LCM)$ returns the mutant M_{mut} corresponding to LCM applied to M, KP are the Killing Properties, SI are the Surviving Invariants, SP are the Surviving Properties, UP are the Unknown Properties, k_{kill} is a parameter for maximum k-depth to kill a mutant, functions $BMC((Model, Invariants, ValidProperties), Prop, k)$ and $KIND(\dots)$ run respectively BMC and K-INDuction on a model together with its invariants and its valid properties to check new properties Prop at depth k and answer UNSAT (all Prop are valid) or SAT (some of Prop are not valid). When the answer is SAT, the function $getModel()$ gives the counterexample. Finally, function $FilterInvs(invariants, M_{mut})$ filters the $invariants$ of the original Model M using BMC and k-induction to find the ones that survive the mutation and are still invariants of the current mutant M_{mut}.

Starting from the proof of P on M which requires the generation of invariants $Invs$ and induction at depth k_{proof}, our algorithm applies a mutation LCM at each iteration to obtain a mutant M_{mut} and retries the proof of P on M_{mut}.

It runs first BMC at depth k_{proof} to verify whether all properties in P hold on M_{mut} for the first k_{proof} steps. If it is not the case, the mutant M_{mut} is already killed by some properties in P reported within the verdict KILLED. When all properties in P hold, which means that the base step is valid, the algorithm will try the k-induction step after filtering the invariants Invs of M to keep only those that are still valid for M_{mut}. When the k-induction step succeeds (UNSAT), all properties in SP are k-inductive and survive, otherwise we use the counterexample model to find the properties that are not k-inductive, add them to the unknown properties UP, and we try again the k-induction on the remaining properties SP $\setminus$ UP. We add the non k-inductive properties to UP because they can be valid but may require a k-induction of a higher depth. The verdict is SURVIVED when the k-induction succeeds at the first iteration and in this case all properties in P hold for M_{mut} (i.e. P = SP and UP is empty). If UP is not empty, we run again BMC at maximum depth k_{kill} to try to kill the current mutant by any property from UP. If this last attempt to kill M_{mut} fails, we return the verdict UNKNOWN.

7.4 IMPLEMENTATION AND INITIAL RESULTS

7.4.1 *Implementation*

We implemented our algorithm on a GitHub fork of JKIND[2]. Our algorithm, shown in Figure 30, runs as a separate engine (module) of JKIND and starts at the end of the proof process. It retrieves the invariants and k_{proof} used for proving the properties and returns mutations verdicts.

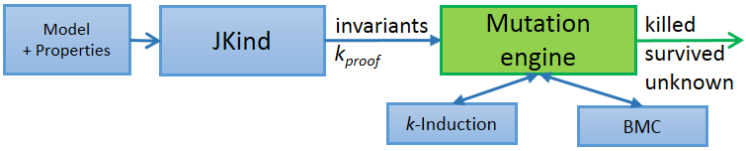


Figure 30: Mutation engine implementation in JKIND

7.4.2 *Optimizations*

Our implementation is very efficient because instead of submitting the entire mutated model to the SMT-solver it works in an incremental way, using *pop* and *push* only on the mutated lines. Furthermore, to take maximum advantage of this incremental feature, we group the mutations of the same line of code and run them all on the same SMT-solver instance.

We introduced two major optimizations as parameters in JKIND: *parallelMutants* and *ivcMutation*. Firstly, unlike IVC, which cannot be parallelized, our

2 JKIND with Mutation on GitHub: https://github.com/v-todorov/jkind

mutation algorithm can run each mutation proof on a different thread. We group mutations that affect a given line of code. Different groups can be executed in parallel. The second optimization is intended for large models and runs the mutation only over the resulting minimal core produced by IVC. Thus IVC eliminates the unused part of the model, and mutation runs faster based on the results of IVC. The designer should be informed of the unused part in order to be able to write some additional properties about it.

7.4.3 *Initial Results*

We used the benchmark of JKind (from GitHub), which provides Lustre files and properties to be proved. We selected 22 example Lustre files with only valid properties, because it is not useful to analyze the coverage of invalid properties. We used a laptop equipped with an Intel Xeon E-2176M CPU and 32GB RAM to run the benchmarks. We applied IVC alone, Mutation with equation removing only, and Mutation with all mutators activated. We activated the *parallelMutants* option to use the 6 cores of our CPU and we did not activate IVC when running Mutation. The results are shown in Figure 31. On the left, we see the results that compare Mutation with only the equation removing mutator (Mut-Eq) to IVC. We notice that in 82% of the use cases we obtained equal times for calculating IVC and Mut-Eq, in 9% of the cases mutation (Mut-Eq) was faster than IVC and in another 9% it was slower. For Mutation using all mutators (Mut-All), we had same execution times in 59% of the cases, mutation (Mut-All) was faster than IVC in 5% of the cases, and it was slower in 36% of the cases.

The unsat cores given by most SMT solvers being not necessarily minimal, IVC needs some backtracking to reduce them to minimal ones. The IVC implementation in JKind is sequential and requires calculation power. On the other hand, our algorithm runs in parallel and uses incremental SMT solving. Thus, we obtain a greater coverage precision thanks to the mutation, with an equivalent performance most of the time.

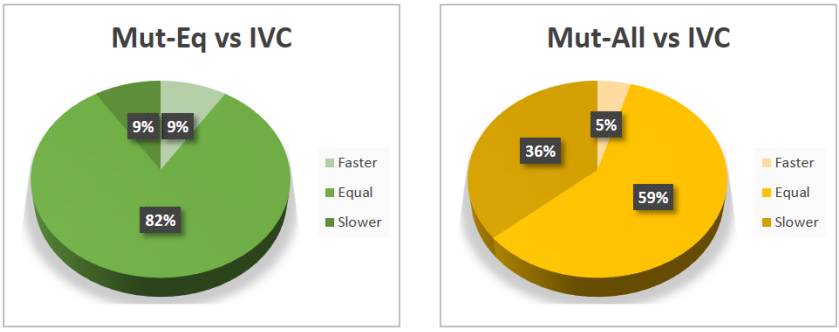


Figure 31: Comparison between equation remover mutation/full mutation and IVC

7.4.4 *Industrial Use Case Results*

We also used a representative industrial use case that is a cruise control function developed in SCADE (1250 lines of Lustre code), with some valid safety properties coming from high level requirements [TTBH19a]. Using IVC, as well as using mutation with equation removing only, shows that all lines of code were covered and therefore necessary to the specification proof, but when running our mutation proof framework with all mutators activated, we only obtained 39% of killed mutations. This means that we need to strengthen the properties e.g. by adding additional ones to kill the 61% surviving mutations. In particular, we found some interesting mutations of *if-then-else* statements revealing branches that were not covered by the original properties.

7.5 CONCLUSIONS

In this chapter, we proposed a new coverage metrics for evaluating the quality of properties (specification) that are proved valid using model checking on a given model (program). The algorithm we used is particularly efficient unlike classical mutation testing techniques. Its efficiency comes from the fact that instead of submitting each mutant to the SMT solver, we only submit the original model once and we iteratively remove (pop) an equation and push its mutated version to check all mutants. The mutation process can also be run in parallel and thus its performance is almost equivalent to IVC, another heuristic algorithm to find the coverage of the properties on a model. The main advantage of our mutation framework over IVC is that we can look inside the lines of code and see the effect of mutating a constant, a variable or an operator.

8

DEDUCTIVE PROOF APPLIED TO A DISCRETE-VALUED FUNCTION

> *Negative results are just what I want. They're just as valuable to me as positive results. I can never find the thing that does the job best until I find the ones that don't.*
>
> — Thomas A. Edison

In the automotive software, discrete-valued functions (also known as Lookup table (LUT)) are frequently used to avoid complex calculations because of the limited resources of the ECU.

For our study, we take as example a discrete-valued function calculating a square root using a linear interpolation table. During the design stage in an MBD process, the simulation can use a complex function (see Figure 32) but in the implementation it is replaced by an optimized version.

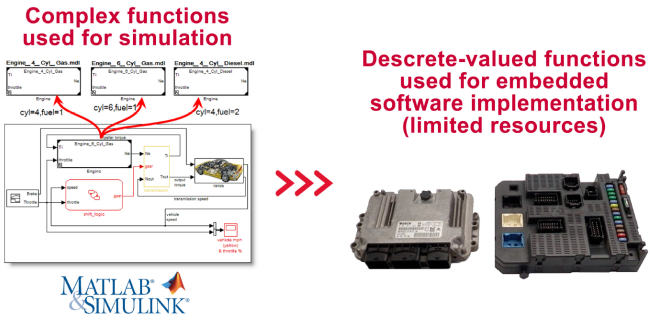


Figure 32: Complex functions vs. discrete-valued ones

In this chapter, we give details about the application of deductive proof to production code, the problems we encountered with off-the-shelf tools, and some approaches to solve this type of problems. Our function has been implemented in C and we used Frama-C WP for proving its correctness. As some of the goals were impossible to prove with Frama-C and its solvers, we implemented it in SPARK (based on Ada) to prove it with GNATprove. We discuss the results as well as how other methods such as Abstract interpretation can be combined with deductive proof.

8.1 ENVIRONMENT

We present our environment in Figure 33. We have C code, which is annotated with contracts using the ACSL language. We use two different features of Frama-C WP. First, we use it to parse and then transform the C code together with the contracts into Verification Conditions (VC) that are directly sent to the SMT solvers. Second, we also use Frama-C WP to transform the C code together with the contracts into WhyML language files. The Why3 framework then transforms the WhyML files into VC and addresses the SMT solvers. The main difference between these two approaches is that the direct SMT-LIB output was initially developed for the Colibri SMT solver, which does not support quantifiers. Thus, the direct SMT-LIB output provides a set of quantifier-free formulas. The other way, through WhyML, allows for richer theories and supports quantified formulas even within the specification contracts.

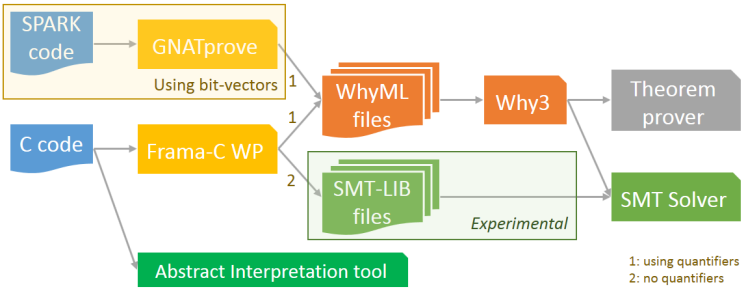


Figure 33: Environment for deductive proof on C and SPARK code

We used GNATprove to prove the equivalent code written in SPARK. This approach is similar to using Frama-C with WhyML and quantified formulas. The advantage of SPARK for our use case is that we can use bit vector types for modular arithmetic and thus facilitate the proof.

Because we experienced some difficulties with the analysis of our C code, we also analyzed it with an Abstract interpretation tool to get additional confidence.

8.2 EXPERIMENT

We took the C code implemented in an on-board computer to prove its correctness using deductive proof. The function calculates the square root Y of X by linear integer interpolation between two known points (X_a, Y_a) and (X_b, Y_b) using the following formula:

$$Y = Y_a + (X - X_a)\frac{(Y_b - Y_a)}{(X_b - X_a)}$$

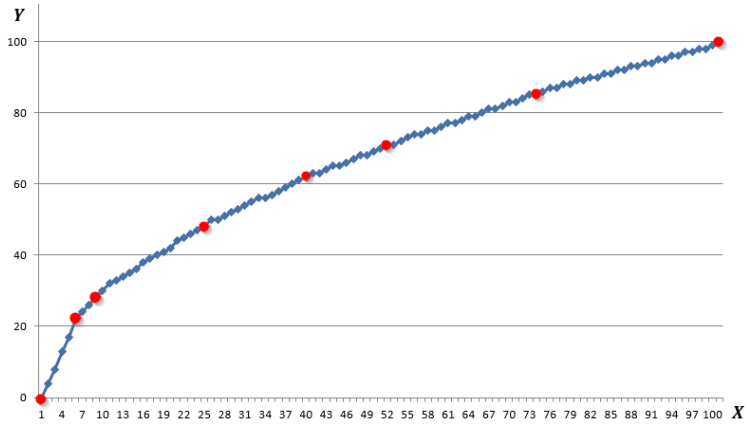


Figure 34: Square root calculation in [0, 1.00] by linear interpolation from eight known values

This code is used in an implementation on an on-board computer, which cannot use floating-point numbers. We calculate the square root for numbers between 0.00 and 100.00 using an integer representation. We consider it as a fixed-point number (multiplied by 100 to have a precision of 2 digits after the decimal separator), thus the input range is between 0 and 10000 (representing 0 and 100.00) and the returned result is a linearly interpolated value between 0 and 1000 (to be interpreted as a number between 0 and 10.00). We want to prove that the calculation is correct for a given precision.

We proceeded in two steps. First, we proved a simplified version of the code using only eight values in the interpolation table (Figure 34) and limited to the range $[0, 1.00]$. These values were a subset of the full table present in the code, which contains 41 values. Then, we added the other values in the table and updated the contracts to take into account the new bounds. To our surprise, this did not scale up with Frama-C. We worked with the developers of Frama-C to understand why (we explain it in Section 8.3).

The code of our main function is given in Figure 35. This function takes a number and returns its square root using a table for some known values or interpolates a value when the number is between two known points. Using the ACSL annotation language, we define two behaviors for this function: whenever the number is less than 10000 the function is defined, otherwise it returns the maximum value i.e. 1000. For more readability, we removed some intermediate values from the two tables.

As we have a loop, in formal verification we need to specify a loop invariant for it. A loop invariant is a predicate (condition) that holds for every iteration of the loop (before and after the iteration). This predicate should be strong enough and its automatic generation is generally a difficult problem.

With Frama-C we also need to define precisely which variables are modified (assigned) during the loop. In our example, i is incremented on each iteration.

```
/*@ assigns \nothing;
behavior in_range:
assumes number <= 10000;
ensures number-30 <= (\result)*(\result)/100 <= number+10;
behavior out_of_range:
assumes number > 10000;
ensures \result == 1000;
complete behaviors in_range, out_of_range;
disjoint behaviors in_range, out_of_range;
*/
uint16 IntSqrt(uint16 number) {
uint8 i = 0;
uint16 TabX[41] = {0,5,10,25,40,...,7500,8000,8600,9200,10000};
uint16 TabY[41] = {0,22,32,50,63,...,866,894,927,959,1000};
/*@ loop invariant 0 <= i <= 40 && number >= TabX[i];
loop assigns i;
loop variant 40-i; */
for (i = 0 ; i < 40 ; i++) {
if ((number >= TabX[i]) && (number <= TabX[i+1])) {
return(LinearInterpolation(TabX[i], TabY[i], TabX[i+1], TabY[i+1],
    number));
}
}
return TabY[40];
}
```

Figure 35: Annotated square root function for Frama-C WP automatic proof

For the loop to be proved, we also need to write a variant function which is a function whose value is monotonically decreased with respect to a (strict) well-founded relation by the iteration of the loop. It is used to ensure the termination of the loop.

Then we rewrote the function in SPARK[1] to see whether it would scale better. Figure 36 presents the SPARK code. The main difference between C and SPARK is that we can specify a `bit vector` data type in SPARK, which is then communicated to the SMT solver via Why3. For our use case, it helped the solver to reason using modular arithmetic. Most SMT solvers used as back-end of Why3 have a theory of bit vectors. If we do not use bit vectors, the SMT solver is reasoning by default using non-modular arithmetic.

The proof of the simplified code succeeded on both Frama-C and SPARK. However, when using the full table of 41 values, Frama-C failed where only SPARK succeeded.

We also analyzed our complete C code with Astrée [Mau04] from AbsInt, a static analysis tool using abstract interpretation, to prove some difficult goals. The abstract interpretation results can be used as assumptions for Frama-C WP or bring more confidence for certification if Frama-C can reason on them. We discuss the results in the next section.

1 Special thanks to Yannick Moy from AdaCore

```
type Unsigned is mod 2**32;
subtype uint16 is Unsigned range 0 .. 65535;
type UINT16_ARR is array (Positive range <>) of uint16;
Max : constant := 10_000;
function LinearInterpolation(Xa, Ya, Xb, Yb, X : uint16) return uint16 is
Result : uint16;
begin
if Xa /= Xb then
Result := Ya + (X - Xa) * (Yb - Ya) / (Xb - Xa);
else
Result := Ya;
end if;
return Result;
end LinearInterpolation;
function IntSqrt(number : uint16) return uint16
with Global => null, Contract_Cases =>
(number <= Max => IntSqrt'Result * IntSqrt'Result / 100 + 30 >= number
    and number+10 >= IntSqrt'Result * IntSqrt'Result / 100, number > Max
    => IntSqrt'Result = 1000) is
TabX : UINT16_ARR(1 .. 41) := (0,5,10,25,40,...,8000,8600,9200,10000);
TabY : UINT16_ARR(1 .. 41) := (0,22,32,50,63,...,894,927,959,1000);
begin
for I in 1 .. 40 loop
pragma Loop_Invariant (for all J in 1 .. I => number >= TabX(J));
if number in TabX(I) .. TabX(I+1) then
return LinearInterpolation (TabX(I), TabY(I), TabX(I+1), TabY(I+1),
    number);
end if;
end loop;
return TabY(41);
end IntSqrt;
```

Figure 36: SPARK code for automatic proof with GNATprove

8.3 RESULTS

In this section, we explain the results and why Frama-C failed to scale-up from 8 to 41 values, and what should be done to cope with this type of problems.

8.3.1 *From Frama-C to the SMT solver*

To understand the reason why automatic proof failed for the full table, we have to detail the transformations between the C code through Frama-C, Why3 and the solvers. First, Frama-C transforms the C code and its ACSL contracts using the weakest precondition calculus into VC in the WhyML language. It also introduces additional goals to verify the absence of runtime errors such as overflows. The WhyML output contains all the theories necessary for the proof and is sent to Why3. Then Why3 transforms it into the language of the chosen prover. For our use case, the WhyML transformation contained quantified formulas and had redefined some operators such as `division` using uninterpreted functions.

8.3.2 *The Difficult Goal*

There were 51 goals (verification conditions) to be proved and two of them were not proven. The most difficult goal was about proving that the contract of the post condition in the linear interpolation function had the same behavior as the code. We show it in Figure 37.

```
typedef unsigned short uint16;
typedef unsigned char uint8;
/*@
requires 0 <= Xa <= 10000 && 0 <= Xb <= 10000;
requires 0 <= Ya <= 1000 && 0 <= Yb <= 1000;
requires Yb > Ya && Xb >= Xa;
requires Xa <= X <= Xb;
ensures Xa != Xb ==> \result == (Ya + (X - Xa) * (Yb - Ya) / (Xb - Xa));
ensures Xa == Xb ==> \result == Ya;
assigns \nothing;
*/
uint16 LinearInterpolation(uint16 Xa, uint16 Ya, uint16 Xb, uint16 Yb,
    uint16 X)
{
if (Xa != Xb) {
return(Ya + (X - Xa) * (Yb - Ya) / (Xb - Xa));
} else {
return(Ya);
}
}
```

Figure 37: Annotated interpolation function for Frama-C WP automatic proof

Actually, contracts use mathematical arithmetic (without overflow), but code uses modular arithmetic, where overflows may occur. For our use case, we used a 16-bit unsigned integer to store the returned value of the interpolation.

8.3.3 *Direct Proof with SMT-LIB*

Since 2 goals were not proven with the official Frama-C version, we obtained a new version that could address directly SMT solvers using the SMT-LIB standard [BST+10]. We proved our goals with Colibri, CVC4 and Yices2. We remarked that the SMT-LIB file did not contain quantifiers and did not redefine operators such as `division`. We concluded that this approach scaled and worked better for problems with nonlinear arithmetic such as interpolation functions. Furthermore, some SMT solvers such as Yices2 do not support quantification.

8.3.4 *Experience with the Why3 SMT Output Files*

We wanted to understand what was the impact of the redefined division using uninterpreted functions and of quantified formulas, so we modified manually the SMT request sent to the solver. First, we removed the specific functions

about division and used the standard SMT-LIB `div` operator. Then, the proof succeeded with CVC4 but only if using nonlinear logic containing bit vectors. Disabling bit vectors from that logic resulted in a failure to prove the formula. On the other hand, the quantifier-free SMT output did not need bit vector logic to be proved.

8.3.5 *Abstract Interpretation Combined with Deductive Proof*

Because it is difficult to understand how the SMT solvers proved the difficult goal, we used Astrée to prove the absence of overflow in the returned value of the linear interpolation function. This proof can then be used as hypothesis in Frama-C WP. Astrée could find the dependency between Y_b and Y_a and estimate a precise interval for $(Y_b - Y_a)$. The same was done for $(X_b - X_a)$ and $(X - X_a)$. Thus a precise interval was calculated for Y in $[0, 10000]$, which fits in a 16-bit unsigned integer without overflow.

8.4 METHODOLOGY

In this section, we propose a methodology based on our experience to solve problems using discrete-valued functions such as linear interpolation. Our use case is a simple one and we could have tested it for each value in the domain of validity of the function. However, in practice, there are more complex discrete-valued functions implemented with linear interpolation tables called lookup tables. These functions are often called by other discrete-valued functions. The number of cases to test can be the product of the cardinalities of the domains of the individual functions. We propose to use the methodology shown below in Figure 38 in order to prove those functions.

First, we need to isolate all the functions we want to prove together and annotate the code with contracts specifying the behavior expected from each function. Then, we can try to prove it in Frama-C via Why3. If the proof succeeds, we can stop. Otherwise, we can try to use the direct SMT-LIB output of Frama-C WP with the SMT solvers. As we have seen, this approach removes quantifiers and uses native mathematical operators. If it does not succeed, for some goals (VC) we can try to prove them using abstract interpretation tools. If this method does not succeed, we need to use a proof assistant to prove the difficult goals.

8.5 RELATED WORK

Our work concerns the formal verification of the square root function used for embedded automotive applications and using fixed-point numbers. Embedded software generally needs to be optimized because of the limited power of the micro-controllers. We used deductive proof engines (Frama-C and GNAT-

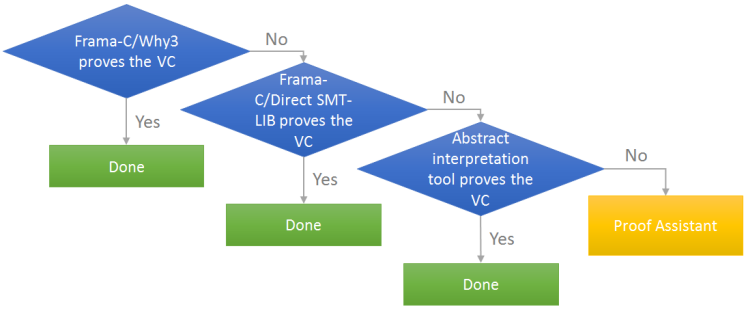


Figure 38: Methodology for proving Discrete-Valued Functions

prove) that create verification obligations discharged mostly automatically by SMT solvers. For our application, we only need to calculate square root for a predefined interval and the precision of our lookup table is enough to satisfy the requirements. We could use other methods such as Newton method but it would be more resources consuming. To the best of our knowledge, a linearly-interpolated fixed-point square root algorithm has not been the subject of formal verification work. In this section, we give a survey on some related work about the correctness proof of square root algorithms for machine representation and for standard mathematical functions in general.

The problem of the specification and validation of standard functions is also discussed in [Kulo7]. Even if a standard for representing floating-point numbers has been defined (IEEE 754), this standard does not provide requirements for the specification of standard functions. This work is a systematic presentation of ideas from other studies about the formal specification and testing of standard mathematical functions. The author does not use automatic proof assistance.

We think that the first floating-point algorithms verifications were motivated by some hardware bugs such as the Pentium FDIV bug discovered in 1994. For example, in 1998 Russinoff used the ACL2 theorem prover to verify the square root algorithm in the K7 microprocessor [Rus98]. Later, in 1999 he also verified the square root microcode of the K5 microprocessor [Rus99]. In 2000, researchers from Intel Corporation verified the square root algorithm used in an Intel processor with the Forte system that combines symbolic trajectory evaluation and theorem proving [AJK+00]. In 2002, IBM presented a research paper about the formal verification of the IBM Power4 processor that uses Chebyshev polynomials to calculate square root [SG02]. The team used the ACL2 theorem prover to mechanically verify the square root algorithm. In 2002, Bertot et al. verified the divide-and-conquer part of Gnu MultiPrecision Library (GMP)'s square root using the Coq proof assistant [BMZ02]. In 2003, Harrison published his work about a square root algorithm verification using HOL Light [Har03]. This particular algorithm used for floating-point numbers was provided by Intel for a new 64-bit architecture called Itanium to replace some less efficient generic libraries. The main benefits of using theorem proving for the verifica-

tion of this algorithm were reliability and re-usability. Actually, its proof involved Diophantine equations that were very tedious and error-prone to do by hand. The author argues that all the proof process should be done in the same tool – the proof assistant – because it uses a strict logical deduction process. In 2011, Shelekhov proposed a specification and verification of square root using PVS [She11]. The paper concludes that synthesis of programs of the standard functions such as *floor*, *isqrt*, and *ilog2* is found to be less tedious than the deductive verification of these programs. In 2016, Oracle presented a research work about the formal verification of a square root implementation [REN+16]. They used ACL2 and interval arithmetic to verify the low-level Verilog descriptions of the floating-point division and square root implementations in the SPARC ISA, and discovered new optimizations (lookup table reductions) while doing so. In 2018, Intel Corporation presented a research paper about the proof of correctness of square root using a digit serial method (DSM) and a theorem prover (HOL-Light) [FBE+18]. A DSM is an algorithm that determines the digits of a real number serially, starting with the leading digit. In 2019, Melquiond et al. presented a paper about the formal verification of the GMP library's algorithm for calculating the square root of a 64-bit integer using Why3 [MRH19]. This algorithm can be seen as a fixed-point arithmetic algorithm that implements Newton method. The authors used the WhyML modeling language to implement GMP's algorithm together with its specification and then the Why3 tool to prove its correctness automatically. The resulting proved WhyML model was then extracted to correct-by-construction C code, which was binary compatible to the one from GMP. The authors reported that this work took a few days. They also used ghost code in WhyML to simplify the verification conditions.

The studies about standard mathematical functions and in particular square root specification and validation cited above are all platform-dependent. A new approach proposed by Shilov et al. consisted in a platform-independent verification of standard mathematical functions. In [NDI+18], this approach was applied to the square root function and combines a manual (pen-and-paper) verification of a base case that proves the algorithm's correctness with real numbers to provide a proof-outline for the verification of the algorithm for machine numbers. The function implements Newton method and uses a lookup table for initial approximations. The specification is done in terms of total correctness assertions with use of precise arithmetic and the mathematical square root and the verification is done in Floyd-Hoare style. A proof of correctness of the algorithm is given for a fixed-point arithmetic and for a floating-point arithmetic. The primary purpose of the paper is to make explicit the properties of the machine arithmetic that are sufficient to perform the verification presented in the paper. Computer-aided implementation and validation of the proof using ACL2 was partially done, the complete ACL2 implementation was left for future studies.

8.6 CONCLUSIONS

In this chapter, we presented our experiments with automatic deductive proof of correctness applied to a discrete-valued function calculating a square root by interpolation. We used Frama-C WP and GNATprove to prove the correctness of the function, but we encountered some difficulties with the nonlinear formula of the linear interpolation. Three non-standard approaches worked well for us: the use of bit vectors in SPARK, the direct SMT-LIB quantifier-free output of Frama-C and the static analysis with Astrée. bit vectors are well supported in most modern SMT solvers and are well suited for problems that involve modular arithmetic, but scaling is sometimes difficult. For our use case, SMT requests without quantifiers performed and scaled better because there was no need for bit vectors. Abstract Interpretation analysis gave more confidence in proving that there was no overflow in the linear interpolation calculus. We have proposed a methodology to use a combination of these different methods until the proof is done. We also show that using industrial use cases with off-the-shelf tools does not always scale, but if we work with researchers, we can find a solution and improve the tools.

Using deductive methods is very promising in an industrial context for safety-critical applications. It can replace unit testing as shown in [MLD+13] and thus decrease cost while increasing quality. It is also an intellectual activity that brings more satisfaction for engineers compared to testing.

9

CONCLUSION AND PERSPECTIVES

> *Formal methods will never have a significant impact until they can be used by people who don't understand them.*
>
> — Tom Melham, University of Oxford

In this thesis, we targeted the introduction of formal methods in the automotive embedded software development to bring more robustness and reliability especially for safety-critical applications. Actually, the advent of automated driving and autonomous vehicles can change the current methods and ways of working and will require new ones. The increasing amount of software in the car, along with the complexity of the functionalities it implements, makes its validation challenging. Furthermore, public authorities could require in the future the certification of the software as is already the case in the aviation and railway industries. The liability of the car manufacturers could be engaged in case of fatalities due to software bugs.

To answer these challenges some industries have brought formal methods into their software development process. Their motivation was the exhaustiveness of these methods compared to testing, and the lesser effort necessary to use them compared to thorough testing.

In this final chapter, we first summarize the key elements from our work and review the fulfillment of the research objectives detailed in Chapter 1. We detail in Section 9.2 its concrete results. Section 9.3 presents the future works regarding both the integration of formal methods in the automotive industry and wider scale research directions.

9.1 RESEARCH OBJECTIVES FULFILLMENT

9.1.1 *Research Objective 1: Industrial Applications of Formal Methods*

In order to achieve this objective, we have illustrated the use of formal methods in industries such as aviation and railway in Chapter 4. We based our study on published papers and discussions with industrial companies and academics that have worked with the industry.

In a certification context, these industries are required to provide enough arguments about the safety of their systems. We identified that *model checking* was rather convenient when using a model-based design approach to verify safety properties about the entire system early. *Deductive proof* is also used for particular scenarios such as verifying handwritten C or ADA code but can require a great amount of annotations. Our industrial feedback was that it provided more satisfaction to people used to make tests before because it is a more intellectual activity. Finally, static analysis based on *abstract interpretation* can be used to prove the absence of run-time errors on the final code. This method is recommended by ISO 26262 at all ASIL levels.

9.1.2 *Research Objective 2: Experimental Application on Automotive Use Cases*

Our second objective was experimental. We needed to apply formal methods and tools in order to get an insight on how they can be used in an automotive context and which tools could be accessible for a non-expert engineer.

We used SCADE to model a cruise controller and apply model checking to it. Globally, specifying formal properties with SCADE turned out to be rather easy and accessible for the engineers already familiar with a modeling tool. The proof process worked rather well for this model, which contained only linear arithmetic with Boolean and integer variables. Then we modeled another function using nonlinear arithmetic and floating point-numbers. Thus, we reached the limits of the current industrial model checkers capabilities. For now, they cannot cope with the nonlinear arithmetic and have a limited support for floating point-numbers. We also identified some limitations for proving properties containing long duration timers.

Our nonlinear model contained a square root calculation based on a lookup table and linear interpolation. We used Frama-C WP to put into practice the deductive proof method. This method is also easily accessible to the engineers but we encountered some difficulties with the scaling of the method. As the linear interpolation is a nonlinear function, Frama-C WP had some difficulties to scale with our complete lookup table. It only handled a subset of the table. We noted that using modern SMT solvers to solve this problem was rather efficient instead of those provided with Frama-C. Another difficulty with this method is the amount of annotations necessary for the proof. In an industrial context, a part of these annotations could be generated automatically.

Finally, we used the abstract interpretation tools Astrée and Polyspace Code Prover on handwritten code and on automatically generated code. In the industry, sound static analysis is often seen as producing too many false alarms but we found that these tools have improved. For our code, there were few false alarms that we fixed using the hints in Section 5.3.3. Polyspace has a more user-friendly interface but is much slower than Astrée. On the other hand, Astrée proposes more advanced options and can be fine-tuned to attain zero false alarms.

9.1.3 *Research Objective 3: Methodologies*

To achieve our third objective, we used production specifications, models and code to apply different methodologies with different methods and tools. In order to be used by non-expert engineers, formal methods need to be integrated in tools available for industrial use and with provided user support. Thus, even a beginner can use them but the following prerequisite are important to take into account:

- *Model checking* and *Deductive proof* require us to identify the properties to be proved. Generally, a specification is present in the safety-critical domain and can be used as a source for formal properties. A domain expert is also necessary for the analysis of the invalid properties to point whether the property, the model and/or the specification are wrong and need to be fixed;
- *Abstract interpretation* tools do not need a specification because the properties to be verified are already integrated in the tool. However, the analysis of the alerts can require some knowledge of the code, its internal/external interactions and the programming language's side effects.

9.2 CONCRETE PRODUCTIONS

This thesis permitted to bridge the gap between the industry and academy in three aspects:

- By proposing a new algorithm improving the invariant generation, enabling the automatic proof of properties involving long-running timers, which are present in most embedded software. This algorithm was integrated in JKind on GitHub[1] as proof of concept and tested successfully at Groupe PSA and Collins Aerospace;
- By proposing a new coverage metrics for evaluating the quality of properties (specification) that are proved valid using model checking and mutation techniques. This algorithm was integrated in JKind on GitHub[2] as proof of concept and tested successfully on a representative automotive model;
- By providing researchers a concrete example of the square root function used in the automotive domain to help the improvement of the automatic deductive provers.

1 JKind, branch "invgen-timers" on GitHub: https://github.com/agacek/jkind
2 JKind, branch "mutation" on GitHub: https://github.com/v-todorov/jkind

9.3 FUTURE RESEARCH DIRECTIONS

Our invariant generation technique and methodology using types could also be applied to a more difficult problem: proving properties on nonlinear systems. Modern SMT solvers take into account some nonlinear theories, but it is time consuming to obtain models for complex nonlinear queries. Model checking for nonlinear systems based on invariant generation could be the subject of a future work.

Another subject of a future work can be to continue developing the link between invariant generation and mutation proof. It consists in finding parts of the code that are not covered by the automatically generated invariants and highlight them to give an immediate feedback to the designer who will need to strengthen the specification on those particular parts of the code. It will improve the provability of the specification and its quality.

PUBLICATIONS

2020 *Conference papers*

V. Todorov, S. Taha, and F. Boulanger, "Specification Quality Metrics Based on Mutation and Inductive Incremental Model Checking" in Proceedings of the NASA Formal Methods - 12th International Symposium, NFM 2020, Moffett Field, CA, USA, May 11-15, 2020, May 2020, vol. 12229, pp. 187–203, doi: 10.1007/978-3-030-55754-6_11.
https://hal-centralesupelec.archives-ouvertes.fr/hal-02956436

2019 *Journal articles*

V. Todorov, S. Taha, F. Boulanger, and A. Hernandez, "Proving Properties of Discrete-Valued Functions Using Deductive Proof: Application to the Square Root" in Modeling and Analysis of Information Systems, vol. 26, no. 4, pp. 520–533, Dec. 2019, doi: 10.18255/1818-1015-2019-4-520-533.
https://www.mais-journal.ru/jour/article/view/1274/930

V. Todorov, S. Taha, F. Boulanger, and A. Hernandez, "Proving Properties of Discrete-Valued Functions Using Deductive Proof: Application to the Square Root" in System Informatics, no. 14, pp. 45–54, Jul. 2019.
https://system-informatics.ru/en/article/248

Conference papers

V. Todorov, S. Taha, F. Boulanger, and A. Hernandez, "Improved Invariant Generation for Industrial Software Model Checking of Time Properties" in IEEE 19th International Conference on Software Quality, Reliability and Security (QRS), Sofia, Bulgaria, Jul. 2019, pp. 334–341, doi: 10.1109/QRS.2019.00050.
https://hal-centralesupelec.archives-ouvertes.fr/hal-02322576

2018 *Conference papers*

V. Todorov, F. Boulanger, and S. Taha, "Formal Verification of Automotive Embedded Software" in Proceedings of the 6th Conference on Formal Methods in Software Engineering, New York, NY, USA, Nov. 2018, pp. 84–87, doi: 10.1145/3193992.3194003.
https://hal.archives-ouvertes.fr/hal-01768687

BIBLIOGRAPHY

[Abr96] Jean-Raymond Abrial. *The B-book : assigning programs to meanings*. Cambridge [u.a.] Cambridge Univ. Press, 1996.

[Abr18] Jean-Raymond Abrial. On B and Event-B: Principles, Success and Challenges. pages 31–35. 2018.

[AFMW96] Martin Alt, Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. Cache behavior prediction by abstract interpretation. Static Analysis, pages 52–66, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[AJK+00] Mark D. Aagaard, Robert B. Jones, Roope Kaivola, Katherine R. Kohatsu, and Carl-Johan H. Seger. Formal Verification of Iterative Algorithms in Microprocessors. In *Proceedings of the 37th Annual Design Automation Conference*, DAC '00, pages 201–206, New York, NY, USA, 2000. Association for Computing Machinery. event-place: Los Angeles, California, USA.

[And03] Charles André. Semantics of S . S . M . (Safe State Machine). Université de Nice-Sophia Antipolis/CNRS, April 2003.

[Are19] Nikos Arechiga. Specifying Safety of Autonomous Vehicles in Signal Temporal Logic. In *2019 IEEE Intelligent Vehicles Symposium (IV)*, pages 58–63, 2019.

[ARG+20] Omar M. Alhawi, Herbert Rocha, Mikhail R. Gadelha, Lucas C. Cordeiro, and Eddie Batista. Verification and refutation of C programs based on k-induction and invariant inference. *International Journal on Software Tools for Technology Transfer*, May 2020.

[Bar00] John Barnes. The SPARK Way to Correctness is via Abstraction. *Ada Lett.*, XX(4):69–79, December 2000.

[Bar03] John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[BBF+00] Gérard Berry, Amar Bouali, Xavier Fornari, Emmanuel Ledinot, Eric Nassor, and Robert de Simone. Esterel: A Formal Method Applied to Avionic Software Development. *Sci. Comput. Program.*, 36(1):5–25, January 2000.

[BCB18] Jaroslav Bendik, Ivana Cerná, and Nikola Beneš. Recursive Online Enumeration of All Minimal Unsatisfiable Subsets. In Shuvendu K. Lahiri and Chao Wang, editors, *Automated Technology for Verification and Analysis*, pages 143–159, Cham, 2018. Springer International Publishing.

[BCC97] Sergey Berezin, Sérgio Vale Aguiar Campos, and Edmund M. Clarke. Compositional Reasoning in Model Checking. In *Revised Lectures from the International Symposium on Compositionality: The Significant Difference*, COMPOS'97, pages 81–102, Berlin, Heidelberg, 1997. Springer-Verlag.

[BCC+03] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A Static Analyzer for Large Safety-Critical Software. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 196–207, New York, NY, USA, May 2003. Association for Computing Machinery. event-place: San Diego, California, USA.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 193–207. Springer Berlin Heidelberg, 1999.

[BCD+11] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovi'c, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11)*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, July 2011.

[BCD+20] Abderrahmane Brahmi, Marie-Jo Carolus, David Delmas, Mohamed Habib Essoussi, Pascal Lacabanne, Victoria Moya Lamiel, Famantanantsoa Randimbivololona, and Jean Souyris. Industrial use of a safe and efficient formal method based software engineering process in avionics. In *10th European Congress on Embedded Real Time Software and Systems (ERTS 2020)*, Toulouse, France, January 2020.

[BCG83] R. Balzer, Jr. Cheatham, T. E., and C. Green. Software Technology in the 1990's: Using a New Paradigm. *Computer*, 16(11):39–45, November 1983.

[BCHPM04] Yves Bertot, Pierre Castéran, Gérard (informaticien) Huet, and Christine Paulin-Mohring. *Interactive theorem proving and program development : Coq'Art : the calculus of inductive constructions*. Texts

in theoretical computer science. Springer, Berlin, New York, 2004. Données complémentaires http://coq.inria.fr.

[BCM+89] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 1020 States and Beyond. *Information and Computation*, 98:142–170, 1989.

[BD05] Amar Bouali and Bernard Dion. Formal Verification for Model-Based Development. In *SAE Technical Paper 2005-01-0781*, 2005.

[BDE+18] Abderrahmane Brahmi, David Delmas, Mohamed Habib Essoussi, Famantanantsoa Randimbivololona, Abdellatif Atki, and Thomas Marie. Formalise to automate: deployment of a safe and cost-efficient process for avionics software. In *9th European Congress on Embedded Real Time Software and Systems (ERTS 2018)*, Toulouse, France, January 2018.

[Ber19] Ryan Berryhill. Chasing Minimal Inductive Validity Cores in Hardware Model Checking. October 2019.

[BFFFL+11] Ricardo Bedin França, Denis Favre-Felix, Xavier Leroy, Marc Pantel, and Jean Souyris. Towards Formally Verified Optimizing Compilation in Flight Control Software. In *PPES 2011: Predictability and Performance in Embedded Systems*, volume 18 of *OpenAccess Series in Informatics*, pages 59–68, Grenoble, France, March 2011. Schloss Dagstuhl, Leibniz-Zentrum fuer Informatik.

[BG92] Gérard Berry and Georges Gonthier. The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation. *Sci. Comput. Program.*, 19(2):87–152, November 1992.

[BGWC18] Jaroslav Bendik, Elaheh Ghassabani, Michael Whalen, and Ivana Cerná. Online Enumeration of All Minimal Inductive Validity Cores. In Einar Broch Johnsen and Ina Schaefer, editors, *Software Engineering and Formal Methods*, pages 189–204, Cham, 2018. Springer International Publishing.

[BL99] Saddek Bensalem and Yassine Lakhnech. Automatic Generation of Invariants. *Formal Methods in System Design*, 15(1):75–92, July 1999.

[BL05] Mike Barnett and K. Rustan M. Leino. Weakest-precondition of Unstructured Programs. *SIGSOFT Softw. Eng. Notes*, 31(1):82–87, September 2005.

[BM08] Aaron R. Bradley and Zohar Manna. Property-directed incremental invariant generation. *Formal Aspects of Computing*, 20(4):379–405, 2008.

[BMZ02] Yves Bertot, Nicolas Magaud, and Paul Zimmermann. A Proof of GMP Square Root. *Journal of Automated Reasoning*, 29(3-4):225–252, 2002.

[BNSV14] Guillaume Brat, Jorge A. Navas, Nija Shi, and Arnaud Venet. IKOS: A Framework for Static Analysis Based on Abstract Interpretation. In Dimitra Giannakopoulou and Gwen Salaün, editors, *Software Engineering and Formal Methods: 12th International Conference, SEFM 2014, Grenoble, France, September 1-5, 2014. Proceedings*, pages 271–277. Springer International Publishing, Cham, 2014.

[Boc09] Thomas Bochot. *Vérification par Model Checking des commandes de vol : applicabilité industrielle et analyse de contre-exemples*. PhD thesis, 2009. Thèse de doctorat dirigée par Wiels, Virginie et Waeselynck, Hélène Informatique Toulouse, ISAE 2009 2009ESAE0003.

[Bou11] Jean-Louis Boulanger. *Utilisations industrielles des techniques formelles : interprétation abstraite*, volume 1. Paris, hermès science publications-lavoisier edition, June 2011.

[BP13] Timothy Bourke and Marc Pouzet. ZéLus: A Synchronous Language with ODEs. In *Proceedings of the 16th International Conference on Hybrid Systems: Computation and Control*, HSCC '13, pages 113–118, New York, NY, USA, 2013. Association for Computing Machinery. event-place: Philadelphia, Pennsylvania, USA.

[BR02] Thomas Ball and Sriram K. Rajamani. The SLAM Project: Debugging System Software via Static Analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 1–3, New York, NY, USA, 2002. Association for Computing Machinery. event-place: Portland, Oregon.

[Bra11] Aaron R. Bradley. SAT-Based Model Checking without Unrolling. In Ranjit Jhala and David Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation: 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, pages 70–87. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[Bra12] Aaron R. Bradley. Understanding IC3. In Alessandro Cimatti and Roberto Sebastiani, editors, *Theory and Applications of Satisfiability Testing – SAT 2012: 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, pages 1–14. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[Bry86] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.*, 35(8):677–691, August 1986.

[BST+10] Clark Barrett, Aaron Stump, Cesare Tinelli, Sascha Boehme, David Cok, David Deharbe, Bruno Dutertre, Pascal Fontaine, Vijay Ganesh, Alberto Griggio, Jim Grundy, Paul Jackson, Albert Oliveras, Sava Krstić, Michal Moskal, Leonardo De Moura, Roberto Sebastiani, To David Cok, and Jochen Hoenicke. The SMT-LIB Standard: Version 2.0. Technical report, 2010.

[BVWW10] Thomas Bochot, Pierre Virelizier, Hélène Waeselynck, and Virginie Wiels. Paths to Property Violation: A Structural Approach for Analyzing Counter-Examples. In *2010 IEEE 12th International Symposium on High Assurance Systems Engineering*, pages 74–83, November 2010.

[BW16] Steffen Beringer and Heike Wehrheim. Verification of AUTOSAR Software Architectures with Timed Automata. In Maurice H. ter Beek, Stefania Gnesi, and Alexander Knapp, editors, *Critical Systems: Formal Methods and Automated Verification: Joint 21st International Workshop on Formal Methods for Industrial Critical Systems and 16th International Workshop on Automated Verification of Critical Systems, FMICS-AVoCS 2016, Pisa, Italy, September 26-28, 2016, Proceedings*, pages 189–204. Springer International Publishing, Cham, 2016.

[Bü17] David Bühler. *EVA, an Evolved Value Analysis for Frama-C : structuring an abstract interpreter through value and state abstractions*. PhD thesis, 2017.

[CC77] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[CC99] Patrick Cousot and Radhia Cousot. Refining Model Checking by Abstract Interpretation. *Automated Software Engineering*, 6(1):69–95, January 1999.

[CC10] Patrick Cousot and Radhia Cousot. A gentle introduction to formal verification of computer systems by abstract interpretation. In J. Esparza, O. Grumberg, and M. Broy, editors, *Logics and Languages for Reliability and Security*, NATO Science Series III: Computer and Systems Sciences, pages 1–29. IOS Press, 2010.

[CCF+05] Patrick Cousot, Radhia Cousot, Jerôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTREÉ

Analyzer. In Mooly Sagiv, editor, *Programming Languages and Systems*, pages 21–30, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[CCIM18] Sylvain Conchon, Albin Coquereau, Mohamed Iguernlala, and Alain Mebsout. Alt-Ergo 2.2. In *SMT Workshop: International Workshop on Satisfiability Modulo Theories*, Oxford, United Kingdom, July 2018.

[CCM09] Géraud Canet, Pascal Cuoq, and Benjamin Monate. A Value Analysis for C Programs. In *Proceedings of the 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, SCAM '09, pages 123–124, Washington, DC, USA, 2009. IEEE Computer Society.

[CE82] Edmund M. Clarke and E. Allen Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, UK, 1982. Springer-Verlag.

[CFJ93] Edmund M. Clarke, Thomas Filkorn, and Somesh Jha. Exploiting Symmetry In Temporal Logic Model Checking. In *Proceedings of the 5th International Conference on Computer Aided Verification*, CAV '93, pages 450–462, Berlin, Heidelberg, 1993. Springer-Verlag.

[CG12] Alessandro Cimatti and Alberto Griggio. Software Model Checking via IC3. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification: 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, pages 277–293. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[CGJ+00] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-Guided Abstraction Refinement. In E. Allen Emerson and Aravinda Prasad Sistla, editors, *Computer Aided Verification*, pages 154–169, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[CGM+12] Darren Cofer, Andrew Gacek, Steven Miller, Michael W. Whalen, Brian LaValley, and Lui Sha. Compositional Verification of Architectural Models. In *Proceedings of the 4th International Conference on NASA Formal Methods*, NFM'12, pages 126–140, Berlin, Heidelberg, 2012. Springer-Verlag.

[CGMT13] Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. IC3 Modulo Theories via Implicit Predicate Abstraction. *CoRR*, abs/1310.6847, 2013.

[CGSS13] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT Solver. In Nir Piterman and Scott A. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 93–107, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[Cha00] Roderick Chapman. Industrial Experience with SPARK. *Ada Lett.*, XX(4):64–68, December 2000.

[Cha14] A. Champion. *Collaboration of formal techniques for the verification of safety properties over transition systems*. Theses, ISAE - Institut Supérieur de l'Aéronautique et de l'Espace, January 2014.

[CHHL13] Darren Cofer, John Hatcliff, Michaela Huhn, and Mark Lawford. Software Certification: Methods and Tools (Dagstuhl Seminar 13051). *Dagstuhl Reports*, 3(1):111–148, 2013.

[CHN12] Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. SMTInterpol: An Interpolating SMT Solver. In Alastair Donaldson and David Parker, editors, *Model Checking Software*, pages 248–254, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[CKKV01] Hana Chockler, Orna Kupferman, Robert P. Kurshan, and Moshe Y. Vardi. A Practical Approach to Coverage in Model Checking. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Computer Aided Verification*, pages 66–78, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[Cla07] K. Claessen. A Coverage Analysis for Safety Property Lists. In *Formal Methods in Computer Aided Design (FMCAD'07)*, pages 139–145, November 2007.

[CM14a] Darren Cofer and Steven Miller. DO-333 Certification Case Studies. In Julia M. Badger and Kristin Yvonne Rozier, editors, *NASA Formal Methods: 6th International Symposium, NFM 2014, Houston, TX, USA, April 29 – May 1, 2014. Proceedings*, pages 1–15. Springer International Publishing, Cham, 2014.

[CM14b] Darren Cofer and Steven P. Miller. Formal Methods Case Studies for DO-333. Technical report, April 2014.

[CMST16] Adrien Champion, Alain Mebsout, Christoph Sticksel, and Cesare Tinelli. The Kind 2 Model Checker. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification: 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, pages 510–517. Springer International Publishing, Cham, 2016.

[Cof10] Darren Cofer. Model Checking: Cleared for Take Off. In Jaco van de Pol and Michael Weber, editors, *Model Checking Software: 17th International SPIN Workshop, Enschede, The Netherlands, September 27-29, 2010. Proceedings*, pages 76–87. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[CPHP87] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: A Declarative Language for Real-time Programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, pages 178–188, New York, NY, USA, 1987. ACM.

[CPP17] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. Scade 6: A Formal Language for Embedded Critical Software Development. In *TASE 2017 - 11th International Symposium on Theoretical Aspects of Software Engineering*, pages 1–10, Nice, France, September 2017.

[CS14] Roderick Chapman and Florian Schanda. Are We There Yet? 20 Years of Industrial Theorem Proving with SPARK. 2014.

[CWM08] Darren Cofer, Michael W. Whalen, and Steven P. Miller. Software Model Checking for Avionics Systems. In *Proceedings of the 27th Digital Avionics Systems Conference (DASC'08)*, St. Paul, MN, October 2008. IEEE.

[Del12] David Delmas. Fan-C, a Frama-C plug-in for data flow verification. 2012.

[DFJ+18] Claire Dross, Guillaume Foliard, Théo Jouanny, Lionel Matias, Stuart Matthews, Jean-Marc Mota, Yannick Moy, Pascal Pignard, and Romain Soulat. Climbing the Software Assurance Ladder - Practical Formal Verification for Reliable Software. Oxford, UK, July 2018.

[DGP+09] David Delmas, Eric Goubault, Sylvie Putot, Jean Souyris, Karim Tekkal, and Franck Védrine. Towards an Industrial Use of FLUCTUAT on Safety-Critical Avionics Software. In *Proceedings of the 14th International Workshop on Formal Methods for Industrial Critical Systems*, FMICS '09, pages 53–69, Berlin, Heidelberg, 2009. Springer-Verlag.

[Dij75] Edsger W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM*, 18(8):453–457, August 1975.

[DKW08] V. D'Silva, D. Kroening, and G. Weissenbacher. A Survey of Automated Techniques for Formal Software Verification. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 27(7):1165–1178, July 2008.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A Machine Program for Theorem-Proving. *Commun. ACM*, 5(7):394–397, July 1962. Place: New York, NY, USA Publisher: Association for Computing Machinery.

[DLS06] S. Demri, F. Laroussinie, and Ph Schnoebelen. A parametric analysis of the state-explosion problem in model checking. *Journal of Computer and System Sciences*, 72(4):547 – 575, 2006.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[DMLK+16] Stéphane Duprat, Victoria MOYA LAMIEL, Florent Kirchner, Loïc Correnson, and David Delmas. Spreading Static Analysis with Frama-C in Industrial Contexts. In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, TOULOUSE, France, January 2016.

[DS07] David Delmas and Jean Souyris. Astrée: From Research to Industry. In *Proceedings of the 14th International Conference on Static Analysis*, SAS'07, pages 437–451, Berlin, Heidelberg, 2007. Springer-Verlag.

[Dut14] Bruno Dutertre. Yices 2.2. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification*, pages 737–744, Cham, 2014. Springer International Publishing.

[EFJ10] J. F. Etienne, S. Fechter, and E. Juppeaux. Using Simulink Design Verifier for Proving Behavioral Properties on a Complex Safety Critical System in the Ground Transportation Domain. In Marc Aiguier, Francis Bretaudeau, and Daniel Krob, editors, *Complex Systems Design & Management: Proceedings of the First International Conference on Complex System Design & Management CSDM 2010*, pages 61–72. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[FBE+18] W. E. Ferguson, J. Bingham, L. Erkök, J. R. Harrison, and J. Leslie-Hurd. Digit Serial Methods with Applications to Division and Square Root. *IEEE Transactions on Computers*, 67(3):449–456, March 2018.

[FCS18] Pietro Ferrara, Agostino Cortesi, and Fausto Spoto. CIL to Java-Bytecode Translation for Static Analysis Leveraging. In *Proceedings of the 6th Conference on Formal Methods in Software Engineering*, FormaliSE '18, pages 40–49, New York, NY, USA, 2018. Association for Computing Machinery. event-place: Gothenburg, Sweden.

[FFS01] Cormac Flanagan, Cormac Flanagan, and James B. Saxe. Avoiding Exponential Explosion: Generating Compact Verification Conditions. *SIGPLAN Not.*, 36(3):193–205, January 2001.

[FLR17] Kathleen Fisher, John Launchbury, and Raymond Richards. The HACMS program: using formal methods to eliminate exploitable bugs. *Philosophical transactions. Series A, Mathematical, physical, and engineering sciences*, 375(2104):20150401, October 2017.

[FP13] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 – Where Programs Meet Provers. In *ESOP'13 22nd European Symposium on Programming*, volume 7792, Rome, Italy, March 2013. Springer.

[FQ03] Cormac Flanagan and Shaz Qadeer. Assume-Guarantee Model Checking. Technical report, 2003.

[GBPG08] Mihaela Gheorghiu Bobaru, Corina S. Păsăreanu, and Dimitra Giannakopoulou. Automated Assume-Guarantee Reasoning by Abstraction Refinement. In Aarti Gupta and Sharad Malik, editors, *Computer Aided Verification: 20th International Conference, CAV 2008 Princeton, NJ, USA, July 7-14, 2008 Proceedings*, pages 135–148. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[GBW+18] Andrew Gacek, John Backes, Mike Whalen, Lucas Wagner, and Elaheh Ghassabani. The JKind Model Checker. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification*, pages 20–27, Cham, 2018. Springer International Publishing.

[GGW16] Elaheh Ghassabani, Andrew Gacek, and Michael W. Whalen. Efficient Generation of Inductive Validity Cores for Safety Properties. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 314–325, New York, NY, USA, 2016. ACM.

[GGW+17] Elaheh Ghassabani, Andrew Gacek, Michael W. Whalen, Mats P. E. Heimdahl, and Lucas Wagner. Proof-based Coverage Metrics for Formal Verification. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, pages 194–199, Piscataway, NJ, USA, November 2017. IEEE Press. event-place: Urbana-Champaign, IL, USA.

[GKD07] Daniel Große, Ulrich Kühne, and Rolf Drechsler. Estimating Functional Coverage in Bounded Model Checking. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '07, pages 1176–1181, San Jose, CA, USA, 2007. EDA Consortium. event-place: Nice, France.

[GL94] Orna Grumberg and David E. Long. Model Checking and Modular Verification. *ACM Trans. Program. Lang. Syst.*, 16(3):843–871, May 1994. Place: New York, NY, USA Publisher: Association for Computing Machinery.

[GNP18] Dimitra Giannakopoulou, Kedar S. Namjoshi, and Corina S. Păsăreanu. Compositional Reasoning. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 345–383. Springer International Publishing, Cham, 2018.

[GP93] Patrice Godefroid and Didier Pirottin. Refining dependencies improves partial-order verification methods (extended abstract). In Costas Courcoubetis, editor, *Computer Aided Verification*, pages 438–449, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

[GP15] Eric Goubault and Sylvie Putot. A Zonotopic Framework for Functional Abstractions. *Form. Methods Syst. Des.*, 47(3):302–360, December 2015.

[GPC04] Dimitra Giannakopoulou, Corina S. Pasareanu, and Jamieson M. Cobleigh. Assume-Guarantee Verification of Source Code with Design-Level Assumptions. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 211–220, Washington, DC, USA, 2004. IEEE Computer Society.

[Gri87] David Gries. *The Science of Programming*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 1987.

[GWG17] Elaheh Ghassabani, Michael Whalen, and Andrew Gacek. Efficient Generation of All Minimal Inductive Validity Cores. In *Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design*, FMCAD '17, pages 31–38, Austin, TX, November 2017. FMCAD Inc. event-place: Vienna, Austria.

[GWGH19] E. Ghassabani, M. Whalen, A. Gacek, and M. Heimdahl. Inductive Validity Cores. *IEEE Transactions on Software Engineering*, pages 1–1, January 2019.

[HAAS+20] Alexandra Halchin, Yamine Ait-Ameur, Neeraj Kumar Singh, Julien Ordioni, and Abderrahmane Feliachi. Handling B models in the PERF integrated verification framework: Formalised and certified embedding. *Science of Computer Programming*, 196:102477, 2020.

[HAK+16] Ashlie B. Hocking, M. Anthony Aiello, John C. Knight, Shinichi Shiraishi, Masahiro Yamaura, and Nikos Arechiga. Proving Prop-

erties of Simulink Models that Include Discrete Valued Functions. May 2016.

[HAKA16] Ashlie B. Hocking, M. Anthony Aiello, John C. Knight, and Nikos Aréchiga. Proving Critical Properties of Simulink Models. In *Symposium Theme: Mission Resilience with High Assurance Systems Engineering*, Orlando, Florida, USA, 2016.

[HAKA17] Ashlie B. Hocking, M. Anthony Aiello, John C. Knight, and Nikos Aréchiga. Input Space Partitioning to Enable Massively Parallel Proof. In Clark Barrett, Misty Davies, and Temesghen Kahsai, editors, *NASA Formal Methods: 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings*, pages 139–145. Springer International Publishing, Cham, 2017.

[Har97] Luddy Harrison. Can abstract interpretation become a mainstream compiler technology? In Pascal Van Hentenryck, editor, *Static Analysis*, pages 395–395, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[Har03] John Harrison. Formal Verification of Square Root Algorithms. *Formal Methods in System Design*, 22(2):143–153, March 2003.

[HB12] Kryštof Hoder and Nikolaj Bjørner. Generalized Property Directed Reachability. In *Proceedings of the 15th International Conference on Theory and Applications of Satisfiability Testing*, SAT'12, pages 157–171, Berlin, Heidelberg, 2012. Springer-Verlag.

[HCRP91] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[HKAS14] A. B. Hocking, J. Knight, M. A. Aiello, and S. Shiraishi. Proving Model Equivalence in Model Based Design. In *Software Reliability Engineering Workshops (ISSREW), 2014 IEEE International Symposium on*, pages 18–21, November 2014.

[HLR93] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*, Twente, June 1993. Workshops in Computing, Springer Verlag.

[Hoa69] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Commun. ACM*, 12(10):576–580, October 1969.

[Hob15] Chris Hobbs. *Embedded Software Development for Safety-Critical Systems*. Auerbach Publications, USA, 2015.

[HT08] George Edward Hagen and Cesare Tinelli. Scaling Up the Formal Verification of Lustre Programs with SMT-Based Techniques. In *2008 Formal Methods in Computer-Aided Design*, pages 1–9, November 2008.

[Ing19] Félix Ingrand. Recent Trends in Formal Validation and Verification of Autonomous Robots Software. In *IEEE International Conference on Robotic Computing*, Proceeedings of the Third IEEE International Conference on Robotic Computing (IRC), Naples, Italy, February 2019.

[ISO18] ISO. *ISO 26262, Road vehicles - Functional safety*. ISO, Geneva, Switzerland, 2018. Type: Norm.

[JDK+14] Xiaoqing Jin, Jyotirmoy Deshmukh, James Kapinski, Koichi Ueda, and Ken Butts. Challenges of Applying Formal Methods to Automotive Control Systems. 2014 NATIONAL WORKSHOP ON TRANSPORTATION CYBER-PHYSICAL SYSTEMS PROGRAM COMMITTEE, 2014.

[JM97] J. M. Jazequel and B. Meyer. Design by contract: the lessons of Ariane. *Computer*, 30(1):129–130, January 1997.

[KEH+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. SeL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. Association for Computing Machinery. event-place: Big Sky, Montana, USA.

[KGT11] Temesghen Kahsai, Yeting Ge, and Cesare Tinelli. Instantiation-based Invariant Discovery. In *Proceedings of the Third International Conference on NASA Formal Methods*, NFM'11, pages 192–206, Berlin, Heidelberg, 2011. Springer-Verlag.

[KGTW12] Temesghen Kahsai, Pierre-Loïc Garoche, Cesare Tinelli, and Mike Whalen. Incremental Verification with Mode Variable Invariants in State Machines. In *Proceedings of the 4th International Conference on NASA Formal Methods*, NFM'12, pages 388–402, Berlin, Heidelberg, 2012. Springer-Verlag.

[KKP+15] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: A software analysis perspective. *Formal Aspects of Computing*, 27(3):573–609, 2015.

[KMMS16] Nikolai Kosmatov, Claude Marché, Yannick Moy, and Julien Signoles. Static versus Dynamic Verification in Why3, Frama-C and SPARK 2014. In *7th International Symposium on Leveraging Applications*, volume 9952 of *Lecture Notes in Computer Science*, pages 461–478, Corfu, Greece, October 2016. Springer.

[Kss+19] Daniel Kästner, bernard schmidt, maximilian schlund, Laurent Mauborgne, Stephan Wilhelm, and Christian Ferdinand. Analyze This! Sound Static Analysis for Integration Verification of Large-Scale Automotive Software. 2019.

[KT14] Daniel Kroening and Michael Tautschnig. CBMC – C Bounded Model Checker. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 389–391, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[Kul07] V. V. Kuliamin. Standardization and Testing of Implementations of Mathematical Functions in Floating Point Numbers. *Program. Comput. Softw.*, 33(3):154–173, May 2007.

[Kur08] R. P. Kurshan. Verification Technology Transfer. In Orna Grumberg and Helmut Veith, editors, *25 Years of Model Checking*, pages 46–64. Springer-Verlag, Berlin, Heidelberg, 2008.

[LA04] C. Lattner and V. Adve. LLVM: a compilation framework for lifelong program analysis transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86, 2004.

[LAC+17] Lucas Wagner, Alain Mebsout, Cesare Tinelli, Darren Cofer, and Konrad Slind. Qualification of a Model Checker for Avionics Software Verification. 2017.

[LBCG16] Jing Liu, John D. Backes, Darren Cofer, and Andrew Gacek. From Design Contracts to Component Requirements Verification. In Sanjai Rayadurgam and Oksana Tkachuk, editors, *NASA Formal Methods: 8th International Symposium, NFM 2016, Minneapolis, MN, USA, June 7-9, 2016, Proceedings*, pages 373–387. Springer International Publishing, Cham, 2016.

[LDPM17] Thierry Lecomte, David Deharbe, Etienne Prun, and Erwan Mottin. Applying a Formal Method in Industry: A 25-Year Trajectory. In Simone Cavalheiro and José Fiadeiro, editors, *Formal Methods: Foundations and Applications*, pages 70–87, Cham, 2017. Springer International Publishing.

[LM10] K. Rustan M. Leino and Michał Moskal. Usable Auto-Active Verification. Technical report, Research in Software Engineering Microsoft Research, Redmond, WA, USA, November 2010.

[LMR+98] Ph Lacan, Jordi Monfort, Le Vinh Quy Ribal, Alina Deutsch, and Anne Gonthier. The software reliability verification process: The ariane 5 example. 1998.

[LSP07] Thierry Lecomte, Thierry Servat, and Guilhem Pouzancre. Formal Methods in Safety-Critical Railway Systems. 2007.

[Mau04] Laurent Mauborgne. Astrée: Verification of Absence of Runtime Error. In Renè Jacquart, editor, *Building the Information Society: IFIP 18th World Computer Congress Topical Sessions 22–27 August 2004 Toulouse, France*, pages 385–392. Springer US, Boston, MA, 2004.

[MB05] Bruno Marre and Benjamin Blanc. Test Selection Strategies for Lustre Descriptions in GATeL. *Electron. Notes Theor. Comput. Sci.*, 111(C):93–111, January 2005.

[MBF18] Alin Mihalache, Fabrice Bedoucha, and Yann-Mikaël Foll. CO-EXISTENCE OF CRITICAL AND NON-CRITICAL SOFTWARE MODULES USING PARTITIONING AND SUPERVISION. In *Congrès Lambda Mu 21, " Maîtrise des risques et transformation numérique : opportunités et menaces "*, Reims, France, October 2018.

[McM93] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, USA, 1993.

[McM05] K. L. McMillan. Applications of Craig Interpolants in Model Checking. In Nicolas Halbwachs and Lenore D. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 1–12, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[McM06] Kenneth L. McMillan. Lazy Abstraction with Interpolants. In Thomas Ball and Robert B. Jones, editors, *Computer Aided Verification*, pages 123–136, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[Mil09] Steven P. Miller. Bridging the Gap Between Model-Based Development and Model Checking. In Stefan Kowalewski and Anna Philippou, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 443–453, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[MLD+13] Yannick Moy, Emmanuel Ledinot, Herve Delseny, Virginie Wiels, and Benjamin Monate. Testing or Formal Verification: DO-178C Alternatives and Industrial Experience. *IEEE Softw.*, 30(3):50–57, May 2013.

[Moy09] Yannick Moy. [Frama-c-discuss] Frama-C vs Ada/SPARK, November 2009.

[Moy10] Yannick Moy. Static analysis is not just for finding bugs. 23:5–8, 2010.

[MP95] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag New York, Inc., New York, NY, USA, 1995.

[MR05] Laurent Mauborgne and Xavier Rival. Trace Partitioning in Abstract Interpretation Based Static Analyzers. In Mooly Sagiv, editor, *Programming Languages and Systems*, pages 5–20, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[MRH19] Guillaume Melquiond and Raphaël Rieu-Helft. Formal Verification of a State-of-the-Art Integer Square Root. In Sylvie Boldo and Martin Langhammer, editors, *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*, pages 183–186, Kyoto, Japan, June 2019.

[MSS03] João P. Marques Silva and Karem A. Sakallah. Grasp—A New Search Algorithm for Satisfiability. In Andreas Kuehlmann, editor, *The Best of ICCAD: 20 Years of Excellence in Computer-Aided Design*, pages 73–89. Springer US, Boston, MA, 2003.

[MWGH16] A. Murugesan, M. W. Whalen, E. Ghassabani, and M. P. E. Heimdahl. Complete Traceability for Requirements in Satisfaction Arguments. In *2016 IEEE 24th International Requirements Engineering Conference (RE)*, pages 359–364, September 2016.

[NDI+18] Nikolay V. Shilov, Dmitry A. Kondratyev, Igor S. Anureev, Eugene V. Bodin, and Alexei V. Promsky. Platform-independent Specification and Verification of the Standard Mathematical Square Root Function. *Modeling and Analysis of Information Systems*, 25(6):637–666, 2018.

[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 R. Nigel Horspool, editor, *Compiler Construction*, pages 213–228, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[noa15] Tech.AD. Conference Proceedings, Berlin, February 2015.

[noa20] Infer Static Analyzer, May 2020. Page Version ID: 954772395.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. LNCS 2283. Springer, 1 edition, 2002.

[OBC18] Julien ORDIONI, Nicolas BRETON, and Jean-Louis Colaço. HLL v.2.7 Modelling Language Specification. Other STF-16-01805, RATP, May 2018.

[Our15] Alain Ourghanlian. Evaluation of static analysis tools used to assess software important to nuclear power plant safety. *Nuclear Engineering and Technology*, 47(2):212–218, March 2015.

[PDBC+15] Marielle Petit-Doche, Nicolas Breton, Roméo Courbis, Yoann Fonteneau, and Matthias Güdemann. Formal Verification of Industrial Critical Software. In *Formal Methods for Industrial Critical Systems - 20th International Workshop, FMICS 2015, Oslo, Norway, June 22-23, 2015 Proceedings*, pages 1–11, 2015.

[PDH99] Corina S. Păsăreanu, Matthew B. Dwyer, and Michael Huth. Assume-Guarantee Model Checking of Software: A Comparative Case Study. In Dennis Dams, Rob Gerth, Stefan Leue, and Mieke Massink, editors, *Theoretical and Practical Aspects of SPIN Model Checking: 5th and 6th International SPIN Workshops Trento, Italy, July 5, 1999 Toulouse, France, September 21 and 24, 1999 Proceedings*, pages 168–183. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.

[Pnu77] Amir Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57, USA, 1977. IEEE Computer Society.

[Pnu89] A. Pnueli. In Transition from Global to Modular Temporal Reasoning about Programs. In *Logics and Models of Concurrent Systems*, pages 123–144. Springer-Verlag, Berlin, Heidelberg, 1989.

[Poh10] Klaus Pohl. *Requirements Engineering: Fundamentals, Principles, and Techniques*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[QS82] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In Mariangiola Dezani-Ciancaglini and Ugo Montanari, editors, *International Symposium on Programming: 5th Colloquium Turin, April 6–8, 1982 Proceedings*, pages 337–351. Springer Berlin Heidelberg, Berlin, Heidelberg, 1982.

[REN+16] D. L. Rager, J. Ebergen, D. Nadezhin, A. Lee, C. K. Chau, and B. Selfridge. Formal verification of division and square root implementations, an Oracle report. In *2016 Formal Methods in Computer-Aided Design (FMCAD)*, pages 149–152, October 2016.

[RSB+99] Famantanantsoa Randimbivololona, Jean Souyris, Patrick Baudin, Anne Pacalet, Jacques Raguideau, and Dominique Schoen. Applying Formal Proof Techniques to Avionics Software: A Pragmatic Approach. In *Proceedings of the Wold Congress on Formal Methods in the Development of Computing Systems-Volume II*, FM '99, pages 1798–1815, London, UK, UK, 1999. Springer-Verlag.

[Rus95] John Rushby. Formal Methods and their Role in the Certification of Critical Systems. Technical Report SRI-CSL-95-1, Computer Science Laboratory, SRI International, Menlo Park, CA, March 1995. Also available as NASA Contractor Report 4673, August 1995, and to be issued as part of the FAA Digital Systems Validation Handbook (the guide for aircraft certification).

[Rus98] David M. Russinoff. A Mechanically Checked Proof of IEEE Compliance of the Floating Point Multiplication, Division and Square Root Algorithms of the AMD-K7™ Processor. *LMS Journal of Computation and Mathematics*, 1:148–200, 1998.

[Rus99] David M. Russinoff. A Mechanically Checked Proof of Correctness of the AMD K5 Floating Point Square Root Microcode. *Formal Methods in System Design*, 14(1):75–125, January 1999.

[SAB08] N. V. Shilov, I. S. Anureev, and E. V. Bodin. Generation of correctness conditions for imperative programs. *Programming and Computer Software*, 34(6):307–321, November 2008.

[Sai00] Hassen Saidi. Model Checking Guided Abstraction and Analysis. In Jens Palsberg, editor, *Static Analysis*, pages 377–396, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[SAP+05] Sayanlan Das, Ansuman Banerjee, Prasenjit Basu, Pallab Dasgupta, P. P. Chakrabarti, Chunduri Rama Mohan, and L. Fix. Formal methods for analyzing the completeness of an assertion suite against a high-level fault model. In *18th International Conference on VLSI Design held jointly with 4th International Conference on Embedded Systems Design*, pages 201–206, January 2005.

[Sch19] Florian Schanda. SPARK in an automotive context, Frama-C & SPARK Day 2019, Paris, June 2019.

[SD07] Jean Souyris and David Delmas. Experimental Assessment of Astrée on Safety-Critical Avionics Software. pages 479–490. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[SFF04] Jean Souyris and Denis Favre-Felix. Proof of properties in avionics. In *Building the Information Society, IFIP 18th World Computer*

Congress, Topical Sessions, 22-27 August 2004, Toulouse, France, pages 527–535, 2004.

[SG02] Jun Sawada and Ruben Gamboa. Mechanical Verification of a Square Root Algorithm Using Taylor's Theorem. In Mark D. Aagaard and John W. O'Leary, editors, *Formal Methods in Computer-Aided Design*, pages 274–291, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[She11] V. I. Shelekhov. Verification and synthesis of addition programs under the rules of correctness of statements. *Automatic Control and Computer Sciences*, 45(7):421–427, December 2011.

[SWDD09] Jean Souyris, Virginie Wiels, David Delmas, and Hervé Delseny. Formal Verification of Avionics Software Products. In *Proceedings of the 2Nd World Congress on Formal Methods*, FM '09, pages 532–546, Berlin, Heidelberg, 2009. Springer-Verlag.

[Tin10] Cesare Tinelli. Foundations of Satisfiability Modulo Theories. In Anuj Dawar and Ruy de Queiroz, editors, *Logic, Language, Information and Computation: 17th International Workshop, WoLLIC 2010, Brasilia, Brazil, July 6-9, 2010. Proceedings*, pages 58–58. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[Tip95] Frank Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[TNW+10] Max Thalmaier, Minh D. Nguyen, Markus Wedler, Dominik Stoffel, Jörg Bormann, and Wolfgang Kunz. Analyzing k-Step Induction to Compute Invariants for SAT-Based Property Checking. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 176–181, New York, NY, USA, 2010. Association for Computing Machinery. event-place: Anaheim, California.

[TTBH19a] Vassil Todorov, Safouan Taha, Frédéric Boulanger, and Armando Hernandez. Improved Invariant Generation for Industrial Software Model Checking of Time Properties. In *2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS)*, pages 334–341, Sofia, Bulgaria, July 2019. IEEE.

[TTBH19b] Vassil Todorov, Safouan Taha, Frédéric Boulanger, and Armando Hernandez. Proving Properties of Discrete-Valued Functions Using Deductive Proof: Application to the Square Root. *System Informatics*, (14):45–54, July 2019.

[Tur49] Alan M. Turing. Checking a Large Routine. pages 67–69, 1949.

[WA85] W W Wadge and E A Ashcroft. LUCID: The data flow programming language. 1985.

[WCM+08] Michael Whalen, Darren Cofer, Steven Miller, Bruce H. Krogh, and Walter Storm. Integration of Formal Analysis into a Model-based Software Development Process. In *Proceedings of the 12th International Conference on Formal Methods for Industrial Critical Systems*, FMICS'07, pages 68–84, Berlin, Heidelberg, 2008. Springer-Verlag.

[WDD+12] V. Wiels, R. Delmas, D Doose, P.L. Garoche, J. Cazin, and G. Durrieu. Formal Verification of Critical Aerospace Software. *AerospaceLab*, (4):p. 1–8, May 2012.

[WH16] D. Watzenig and M. Horn. *Automated Driving: Safer and More Efficient Future Driving*. Springer International Publishing, 2016.

[WLBF09] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal Methods: Practice and Experience. *ACM Comput. Surv.*, 41(4):19:1–19:36, October 2009.

[YLK16] Huafeng Yu, Chung-Wei Lin, and BaekGyu Kim. Automotive Software Certification: Current Status and Challenges. *SAE Int. J. Passeng. Cars – Electron. Electr. Syst.*, 9:74–80, 2016.

[ZM03] Lintao Zhang and Sharad Malik. Validating SAT Solvers Using an Independent Resolution-Based Checker: Practical Implementations and Other Applications. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1*, DATE '03, page 10880, USA, 2003. IEEE Computer Society.

[ZZWF] Liang Zou, Naijun Zhan, Shuling Wang, and Martin Fränzle. Formal Verification of Simulink/Stateflow Diagrams. In *Automated Technology for Verification and Analysis, ATVA 2015*, pages 464–481. Springer.

LIST OF FIGURES

LIST OF TABLES

LISTINGS

LIST OF DEFINITIONS

LIST OF THEOREMS AND LEMMAS

LIST OF ACRONYMS

ABS	Anti-lock Braking System. 5
ACC	Adaptive Cruise Control. 13
ACSL	ANSI/ISO C Specification Langage. 40, 48, 100, 101, 103
ADAS	Advanced Driver-Assistance Systems. 11, 13
ASIL	Automotive Safety Integrity Level. 16, 22, 25, 31, 50, 110
ASW	Application Software. 9
ATP	Automatic Train Protection. 47
AUTOSAR	AUTomotive Open System ARchitecture. 9–12, 17, 58, 62–64
BMC	Bounded Model Checking. 36, 60, 85, 89, 93–95
BSW	Basic Software. 9
CBTC	Communication-Based Train Control. 47
CDCL	Conflict-Driven Clause Learning. 35
CEA	French Alternative Energies and Atomic Energy Commission. 44, 45, 48, 122
CEGAR	Counterexample Guided Abstraction Refinement. 34
CIL	Common Intermediate Language. 44
CoDDA	Compilable Design Description Assistant. 48
COI	Cone Of Influence. 35, 70
DAL	Design Assurance Level. 24
DCSL	Design Contract Specification Language. 48
DPLL	Davis–Putnam–Logemann–Loveland. 35
DV	Design Verifier. 46
ECU	Electronic Control Unit. 5, 8–10, 68, 99

EDF	French Electric Utility Company. 44
EUROCAE	European Organisation for Civil Aviation Equipment. 24
FCS	Flight Control System. 47
FOL	First-Order Logic. 43, 48
FSC	Functional Safety Concept. 57, 61, 62
GMP	Gnu MultiPrecision Library. 106, 107
HC	Highway Chauffeur. 14
HLL	High Level Language. 46, 47
HLR	High-Level Requirements. 56, 68, 69
HOL	Higher-Order Logic. 43
IVC	Inductive Validity Cores. 39, 74, 76, 77, 83–91, 95–97
KCG	SCADE Suite Compilator. 12, 43
LKA	Lane Keeping Assist. 13
LLR	Low-Level Requirements. 56, 68
LLVM	Low Level Virtual Machine. 44
LOC	Lines Of Code. 2
LTL	Linear Temporal Logic. 88, 89
LUT	Lookup table. 99
MBD	Model-Based Design. 11, 17, 59, 63, 99
MC/DC	Modified Condition/Decision Coverage. 56, 62, 88
MISRA	Motor Industry Software Reliability Association. 8, 11
NaN	Not a Numer. 29
NASA	National Aeronautics and Space Administration. 44

OBDD	Ordered Binary Decision Diagrams. 33, 35
PD	Pedestrian Detection. 13
PDR	Property-Directed Reachability. 37, 38, 67, 72–75, 84, 85
QM	Quality Management. 16, 25
RTCA	Radio Technical Commission for Aeronautics. 24
RTE	Runtime Environment. 9
SAE	Society of Automotive Engineers. 13
SCADE	Safety Critical Application Development Environment. 46, 47, 49, 53, 54, 58, 62, 67, 68, 73, 74, 76, 77, 83, 84, 89, 96, 110
SCADE DV	SCADE Design Verifier. 46, 47, 60–62, 68, 72, 74, 81
SIL	Safety Integrity Level. 23, 25
SLDV	Simulink Design Verifier. 46, 47
SMT	Satisfiability Modulo Theories. 35–40, 50, 55, 61, 68, 72, 74, 77, 83, 84, 88, 89, 92, 95–97, 100, 102, 104–106, 108, 110, 112
TJC	Traffic Jam Chauffeur. 14
TSC	Technical Safety Concept. 57
V2I	Vehicle-to-Infrastructure. 13
V2V	Vehicle-to-Vehicle. 13
VC	Verification Conditions. 100, 103, 105
WCET	Worst-Case Execution Time. 17, 29, 45
WP	Weakest Precondition. 39, 40

ACKNOWLEDGMENTS

We must find time to stop and thank the people who make a difference in our lives.

— John F. Kennedy

First and foremost, I would like to express my gratitude to my academic advisors Frédéric Boulanger and Safouan Taha, for their advice and devotion during the last five years and for the things I have learned from them. No less, I would like to thank my industrial tutor Armando Hernandez without whom this rich experience may not have been possible. Gérard Berry gave me the inspiration and motivation to love formal methods during its passionate lectures in Collège de France and I am very grateful to him for that.

I would like to thank as well my colleagues at Groupe PSA, my manager François Gouzonnat who was behind me during the difficult moments and Mihai Socoliuc for inventing and supporting this executive PhD.

Likewise, I cannot forget the great support of my family who has witnessed challenges I faced: my father Atanas, my mother Rayna, my sisters Sonya and Biliana, my wife Victoria and my daughters Anna and Elena. Thank you for your patience and understanding. I'm particularly grateful to my aunt Magdalina who first initiated me into computer programming and who left us in 2019. You were a great teacher and aunt, I miss you so much.

I would like to thank the VALS research group at LRI for the exceptional discussions we had over lunch and coffee breaks, especially Sylvain, Fatiha, Sylvie, Christine, Jean-Christophe, Chantal, Delphine, Claude, Guillaume, Andrei, Benoît, Frédéric V. and Burkhart.

For all the knowledge not present in official papers and all the amazing discussions we had during conferences all over the world, I would like to thank: Patrick Cousot, Xavier Leroy, Gilles Dowek, Nicolas Halbwachs, Jean Souyris, David Delmas, Hervé Delseny, Emmanuel Ledinot, Dillon Pariente, Christine La Porte, Nicolas Valot, Jean-Marie Cottin, Jérémie Kirsch, David Lesens, Yannick Moy, Jean-Louis Boulanger, Marc Pouzet, Timothy Bourke, David Mentré, Denis Cousineau, Adrien Champion, Elodie Bernard, Roberto Giacobazzi, Andrew Gacek, Michael Whalen, César Muñoz, Hamza Bourbouh, Bruno Dutertre, Danielle Stewart, Moshe Vardi, Joost-Pieter Katoen, Marta Kwiatkowska, Margus Veanes, Nikolaj Bjørner, Nikolay Shilov, Andrey Palyanov, Mikhail Lavrentiev, Igor Anureev, Sergey Staroletov, Bin Fang, Milen Petrov and many others

as well as the people from the CEA LIST team – Florent Kirchner, Julien Signoles, Loïc Correnson, François Bobot, Franck Vedrine, Nikolay Kosmatov.

My grateful thanks are also extended to my past teachers during Bachelor and Master studies at the University Pierre and Marie Curie especially to Christian Queinnec, one of my best teachers, Emmanuel Saint-James, Anne Derieux, Fabrice Kordon, Guy Pujolle, Olivier Fourmaux and many others.

I appreciate the amount of time spent by Gérard Berry and Cesare Tinelli for reviewing this work, but also Sylvie Putot, Pascale Le Gall, Fabrice Kordon and Sylvain Conchon for being part of the final jury.

DECLARATION OF AUTHORSHIP

I, Vassil Todorov, declare that this thesis titled, "Automotive embedded software design using formal methods" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Gif-sur-Yvette, France, December 2020

Vassil Todorov

RÉSUMÉ ÉTENDU

Contexte. Cette thèse s'intéresse à l'introduction des méthodes formelles dans le processus de développement logiciel embarqué automobile. Nous observons ces dernières années une part croissante des fonctions d'assistance à la conduite dont le niveau de criticité et la complexité sont de plus en plus élevées. Leur vérification et leur validation nécessitera un niveau d'exigence que le test seul ne pourrait assurer.

Objectifs de la thèse. L'objectif de cette thèse est d'améliorer de façon significative le processus de construction et surtout de vérification des applications de l'informatique à l'automobile. Très réduites avant l'an 2000, ces applications jouent maintenant un rôle fondamental dans presque toutes les fonctions d'une voiture, allant du contrôle global du véhicule aux nouvelles aides à la conduite en passant par l'information au conducteur et le divertissement des passagers. Les voitures évoluent aussi vers une autonomie accrue et les processus de certification qu'on trouve depuis longtemps en avionique et ferroviaire finiront forcément par arriver dans l'automobile pour garantir la sécurité de ces systèmes. Nous proposons d'utiliser des méthodes formelles pour contribuer à leur robustesse et à leur certification.

Etat de l'art. En première partie de la thèse nous présentons le contexte automobile à travers AUTOSAR, l'ingénierie des exigences, le cycle de développement en V et la généralisation de la conception à base de modèles. Nous étudions ensuite les standards de certification. Enfin, nous présentons les différentes méthodes formelles et en particulier celles qui sont adaptées au domaine automobile : interprétation abstraite ; model checking simple ou symbolique, par BDDs ou SAT/SMT ; les augmentations plus récentes du model-checking par génération d'invariants, abstraction, etc. Nous décrivons ensuite les principes des méthodes déductives plus ou moins automatisées. Nous terminons cette partie en présentant quelques applications industrielles.

Contributions. En première partie, nous proposons des méthodologies utilisables par des ingénieurs non-experts basées sur des cas d'utilisation concrets pour l'application des méthodes formelles au domaine automobile. Au niveau du code, nous proposons d'utiliser l'interprétation abstraite pour détecter les erreurs d'exécution et la preuve déductive pour vérifier la correction des fonctions de librairie. Au niveau du modèle, nous proposons d'utiliser le model checking inductif basé sur SAT/SMT pour prouver les propriétés critiques ainsi que vérifier la couverture des propriétés prouvées. En effet, le model checking classique proposé à l'origine ne passe plus à l'échelle pour la vérification des logiciels embarqués modernes.
Ensuite, nous nous intéressons à l'amélioration de l'état de l'art. Nous proposons une nouvelle méthode de génération d'invariants permettant de prouver à l'aide de model-checking inductif avec solveurs SMT, des propriétés qui font intervenir des timers sur de longues durées. L'idée est double : d'abord introduire des invariants sous la forme d'inégalités simples concernant ces timers et les autres variables, ensuite éviter une explosion de ces introductions par une analyse des dimensions de ces variables au sens physique (m, s^{-1}, ms^{-1}) : les couplages n'ont pas d'utilité pour des dimensions indépendantes. Cette contribution a permis de prouver rapidement des propriétés qui menaient auparavant à des timeouts.
Nous avons également constaté que pour valoriser les méthodes formelles dans un contexte de certification, il est important de pouvoir donner la couverture du modèle par les propriétés prouvées. Nous proposons un nouvel algorithme plus précis que ceux qui existaient, grâce à l'utilisation de la mutation. Cet algorithme est également efficace grâce à l'usage du mode incrémental des solveurs SMT. En plus d'aider à fournir une traçabilité plus précise que les méthodes existantes, notre méthode permet aussi d'améliorer la spécification.
Enfin, nous proposons une étude de la preuve déductive sur une fonction calculée par table d'interpolation, cas que l'on retrouve relativement souvent dans l'automobile. Le passage à l'échelle de la preuve déductive a été difficile et il a fallu utiliser l'interprétation abstraite pour compléter la preuve, faisant ainsi collaborer deux méthodes formelles.

Conclusion. Nous terminons ce document en rappelant les objectifs et les différents apports de cette thèse. Nous proposons aussi quelques idées pour des travaux futurs en lien avec les apports de la thèse.

PhD thesis defended on December 9th, 2020

Gérard Berry	Professor Emeritus at Collège de France
Cesare Tinelli	Professor at University of Iowa
Sylvie Putot	Professor at Ecole Polytechnique
Pascale Le Gall	Professor at CentraleSupélec
Fabrice Kordon	Professor at Sorbonne Université
Sylvain Conchon	Professor at Université Paris-Saclay
Frédéric Boulanger	Professor at CentraleSupélec
Safouan Taha	Associate Professor at CentraleSupélec
Armando Hernandez	Senior software expert at Groupe PSA

université PARIS-SACLAY
ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre: Intégration de méthodes formelles dans la conception des fonctions logicielles automobiles

Mots clés: Génie logiciel, Vérification de logiciel, Méthodes formelles, Model checking, Interprétation abstraite, Preuve déductive

Résumé: La part croissante des fonctions d'assistance à la conduite, leur criticité, ainsi que la perspective d'une certification de ces fonctions, rendent nécessaire leur vérification et leur validation avec un niveau d'exigence que le test seul ne peut assurer.

Depuis quelques années déjà d'autres domaines comme l'aéronautique ou le ferroviaire sont soumis à des contextes équivalents. Pour répondre à certaines contraintes ils ont localement mis en place des méthodes formelles. Nous nous intéressons aux motivations et aux critères qui ont conduit à l'utilisation des méthodes formelles dans ces domaines afin de les transposer sur des scénarios automobiles et identifier le périmètre potentiel d'application.

Dans cette thèse, nous présentons nos études de cas et proposons des méthodologies pour l'usage de méthodes formelles par des ingénieurs non-experts. Le model checking inductif pour un processus de développement utilisant des modèles, l'interprétation abstraite pour démontrer l'absence d'erreurs d'exécution du code et la preuve déductive pour des cas de fonctions critiques de librairie.

Enfin, nous proposons de nouveaux algorithmes pour résoudre les problèmes identifiés lors de nos expérimentations. Il s'agit d'une part d'un générateur d'invariants et d'une méthode utilisant la sémantique des données pour traiter efficacement des propriétés comportant du temps long, et d'autre part d'un algorithme efficace pour mesurer la couverture du modèle par les propriétés en utilisant des techniques de mutation.

Title: Automotive embedded software design using formal methods

Keywords: Software engineering, Software verification, Formal methods, Model checking, Abstract interpretation, Deductive proof

Abstract: The growing share of driver assistance functions, their criticality, as well as the prospect of certification of these functions, make their verification and validation necessary with a level of requirement that testing alone cannot ensure.

For several years now, other industries such as aeronautics and railways have been subject to equivalent contexts. To respond to certain constraints, they have locally implemented formal methods. We are interested in the motivations and criteria that led to the use of formal methods in these industries in order to transpose them to automotive scenarios and identify the potential scope of application.

In this thesis, we present our case studies and propose methodologies for the use of formal methods by non-expert engineers. Inductive model checking for a model-driven development process, abstract interpretation to demonstrate the absence of run-time errors in the code and deductive proof for critical library functions.

Finally, we propose new algorithms to solve the problems identified during our experiments. These are, firstly, an invariant generator and a method using the semantics of data to process properties involving long-running timers in an efficient way, and secondly, an efficient algorithm to measure the coverage of the model by the properties using mutation techniques.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France