Formal verification of automotive embedded software

Vassil Todorov
LRI, Groupe PSA, Université Paris-Saclay

Frédéric Boulanger
LRI, CentraleSupélec, Université Paris-Saclay

Safouan Taha
LRI, CentraleSupélec, Université Paris-Saclay

ABSTRACT

The ever-increasing complexity of automotive embedded systems and the need for safe advanced driver assistance systems (ADAS) represent a great challenge for car manufacturers. Furthermore, we expect that in the near future, authorities require a software certification in order to get convinced that ADAS are safe enough. Theoretical research and experience show that when using conventional design approaches it is impossible to guarantee high confidence to those systems. The way taken by some industries (e.g. aerospace, railway, nuclear) was by partially using formal verification techniques.

In this paper, we first present a background of the formal verification techniques and how they can contribute to achieve the requirements of some safety standards. Next, we share our experience with the application of those techniques that seem to be mature enough to be used in an industrial context: Static analysis based on Abstract Interpretation, SMT-based software Model checking and Deductive proof. Finally, we make a detailed analysis about our experiments and propose an approach introducing formal methods into the development of automotive embedded software.

CCS CONCEPTS

• **Software and its engineering → Software verification**;

KEYWORDS

Software verification · Formal methods · ADAS · Certification

ACM Reference Format:
Vassil Todorov, Frédéric Boulanger, and Safouan Taha. 2018. Formal verification of automotive embedded software. In *FormaliSE '18: FormaliSE '18: 6th Conference on Formal Methods in Software Engineering , June 2, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3193992.3194003

1 INTRODUCTION

Automotive software has a major role in today's vehicle control and entertainment features. It is developed according to prescription rules defined by the ISO 26262 standard. Verification and validation activities (V&V) have to be performed in order to guarantee the compliance of the software to the standard. These activities are a key concern on complex systems particularly for the safety-critical ones and require a great effort to avoid unexpected behaviors.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
FormaliSE '18, June 2, 2018, Gothenburg, Sweden
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5718-0/18/06…$15.00
https://doi.org/10.1145/3193992.3194003

For many years, the automotive industry has relied on reviews, analysis and tests for V&V activity. The reason was the cost effectiveness of those methods and the fact that the software was not so critical. Actually, the driver had a mechanical control over the steering and the brakes and could stop in case of emergency.

However, in the near future, autonomous vehicles will bring new requirements for higher software correctness and probably a software certification issue. At the same time, software is getting more and more complex. We think that a way to manage the complexity issue and to guarantee a high confidence and integrity in safety-critical software is to use *formal methods*. It will bring a higher reliability, robustness and confidence. Moreover, using formal methods in an early design phase results in a better specification, maintenance and early design validation thus reducing the overall V&V cost.

In this paper, we present our experience in applying formal verification techniques to a software written the way automotive is doing it today. In Section 2 we present a brief overview of formal methods and comment on some tools. In Section 3 we discuss safety standards, tools and cost, which are the factors that could help introducing formal methods in the development process. In Section 4 we give a detailed analysis of our experiments. Our approach for introducing formal methods in the development process is presented in Section 5, before concluding in Section 6.

2 OVERVIEW OF FORMAL METHODS

The first works on formal methods date back to the 1960s. 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 tools are more standardized and stable. Software tools and technology (such as requirements and design methodology) are still changing rapidly.

We shortly discuss here the three main categories of formal methods that provide some guarantee of quality and that are highly automated.

2.1 Abstract Interpretation

Abstract Interpretation is a kind of static analysis introduced in 1977 by Cousot et al. [9] that 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 abstract interpretation is to efficiently compute an approximate or abstract representation of a program and bring *sound* guarantees about dynamic properties. *Sound* means that the approximation is reliable and that 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. False positives can be generated because of the approximation.

This method has been implemented in static analysis tools to check for runtime errors like division-by-zero, out of bound array access, overflows and others. Some tools extend it also to check MISRA coding rules [16], SEI CERT Secure C rules [20] and MITRE Common Weakness Enumeration CWE rules [15]. Astrée from AbsInt, Polyspace Code Prover from MathWorks and TrustInSoft Analyzer from TrustInSoft are commercial tools for sound static analysis. Other tools like Coverity and Klocwork are based on heuristics and do not guarantee to find all runtime errors. They are out of the scope of this paper.

2.2 Model Checking

Model checking is a formal method introduced in the early 80s by two teams: E. M. Clarke, E. A. Emerson [7], and J. P. Queille, J. Sifakis [18]. It determines if a formal model of a system satisfies a correctness property from the specification. It provides a better expressiveness for properties than Abstract Interpretation but is less scalable due to some limitations (e.g. nonlinear arithmetic, combinatorial explosion). The most valuable feature of Model checking for system design engineers is the *counterexample* produced when the property is violated. It shows an execution trace leading to a state in which the property is not verified.

There exist different types of model checkers but for our use cases (checking safety properties on embedded software), the most suitable are the ones that use induction and SMT (Satisfiability Modulo Theories) solvers, for example Kind2 (University of Iowa) [5], JKind (Rockwell Collins) [10], Prover Plugin also known as Design Verifier (Prover Technologies) and GATeL (CEA). They generally translate a model written in a synchronous dataflow formal language such as Lustre [3] into first-order logic formulas which are then checked for satisfiability by the solver. Sometimes, properties are not strong enough to be provable by induction. In this case, it is helpful to strengthen the induction hypotheses with known invariant properties of the system using invariant generation techniques [13].

2.3 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. The proofs are based on Hoare logic and Dijkstra's precondition calculus [11]. This method is generally *more expressive* than abstract interpretation and model checking, but if the property is invalid, there is generally no counterexample to help the engineer. It also needs more human expertise than the other methods.

We can mention some frameworks and tools for deductive proof: SPARK for Ada [6], Frama-C WP for C code [14] and its predecessor CAVEAT that was successfully used at Airbus [21]. The B Method [2] is another deductive method that was used with success in some French railway companies.

3 INTRODUCING FORMAL METHODS IN THE INDUSTRY

Some factors could help introducing formal methods in the industry:

- Recommendation or requirement by the *standards*;
- Availability of supporting *tools* that are easy to use by the engineers;
- An overall *cost* for using them which is less than without using formal methods.

3.1 Standards

For the automotive industry, ISO 26262 is the standard for functional safety management. This standard recommends the use of formal methods for its most critical levels.

In rail transport, EN 50128 recommends the use of formal methods for the less critical levels and highly recommends their use for the most critical levels, making it de facto mandatory.

In the aviation domain particularly, there is a special supplement DO-333 to the DO-178C which modifies, deletes and adds objectives specific to formal methods. Thus using formal methods can replace some heavy duty tasks like MC/DC 100% coverage testing.

The "Common Criteria" software security standard requires the use of formal methods for its highest critical level (EAL7).

Static analysis is also recommended or required by some standards. Performing it with abstract interpretation tools brings better guarantees for the absence of runtime errors.

3.2 Tool support

Using formal methods requires appropriate tools. There has been many tools developed for some particular cases and there is no universal tool covering all types of formal methods. We can also say that the maturity of the tools for each formal method is different. For example, static analysis based on abstract interpretation tools are mature enough to be used, although the precision needs to be improved. Model checking based tools for software are less robust than static analysis tools and need to be improved to cover a larger fragment of programs. Deductive proof tools also need to improve their user assistance in writing formal specifications and invariants, and in giving an explanation when the goals cannot be proved.

3.3 Cost

An important point for the industry when adopting a new technology is the cost. We can argue that using formal methods increases the software quality and thus reduces the overall V&V cost, but it is difficult to measure. Some companies use formal methods in combination with testing, in order to introduce them along an activity that engineers are familiar with. This way, cost reduction can be measured.

4 EXPERIMENTS

In this section, we discuss our experiments with different formal methods to get an idea of which methods can be used at each development stage, if it really scales and if we could find bugs more easily compared to testing. The context is the development of an AUTOSAR embedded application software.

4.1 Abstract Interpretation

This section illustrates the use of abstract interpretation for static analysis of about 300 000 lines of code. We checked a variety of runtime related errors using the MathWorks Polyspace Code Prover

2016b and AbsInt Astrée 17.04i abstract interpretation tools. We noticed that there was a reasonable number of alarms that could be analyzed by the engineers. Comparing the two tools, Polyspace has done a great effort on the user interface and the automation of the project creation and analysis but there is a restricted number of parameters to control. For instance, the user cannot guide the analyzer to choose a specific abstract domain for a portion of the code. On the other hand, Astrée is less automated (the entry point must be set manually, stubs must be defined explicitly) but accepts annotations (the user can select an abstract domain for some portion of the code, or for some variables). Astrée has a great number of options but their default values were not the optimal choice for our software, and we needed some expertise from AbsInt to get a working configuration. Finally, configuring Polyspace and Astrée with similar options, we noticed that the analysis with Polyspace took much more time than with Astrée.

4.2 Model checking

This section illustrates the use of model checking to verify safety properties of a real Cruise Controller function. Initially, this function was developed using textual requirements (high and low-level software requirements) and handwritten C code. Model checking C code has been studied in [19] but none of the model checkers were able to handle C source code for embedded systems out of the box. The tools were made for a specific subset of the C language. That is why we decided to model our requirements and verify which model checking technique works best for our use case. We chose Ansys Scade Suite R17.2 as modeling tool for its formally defined language based on Lustre. Thus the translation from Scade to Lustre was almost automatic. We developed a little tool based on an XSLT transformation called *Scade2Lustre* which transformed our Scade model into Lustre code. This transformation preserves the original semantics as we used the textual version of Scade nodes. Prover Plugin (commercially called Scade Design Verifier) is the native model checker of Scade Suite and we compared its results with other Lustre model checkers. We can note that defining properties in Scade is easy because they are defined using the same library blocks as for the model.

As reported in [8] writing good formal properties shares many similarities with writing good requirements and is as much art as science. This report mentions that properties that cut across an entire system often find the most errors and that the best source of formal properties is found in the safety-related requirements of the system. We formalized our properties from the safety-related requirements, extending them to all system high-level requirements (HLR) concerning the deactivation of the function. We want to prove that the cruise controller will be deactivated when any of the specified conditions occurs.

Once we had modeled the cruise controller and its properties in Scade, we checked them with Prover Plugin, which reported a bug we had previously found during tests. It took us only a few seconds to find it, instead of hours of writing a particular test scenario. Results show that induction-based model checking is working fine for inductive properties and small models. *Invariant generation* is a very useful technique that helps the induction strengthen the property, but can sometimes take a lot of memory and time. Some model checkers verify all the properties at the same time (incremental verification), so the proof of an easy property can help proving a difficult one. PDR/IC3 [12] is essential and efficient for proving properties on big models, properties that consider a long period of time, and properties involving constants that are not exactly the same as in the code. We can summarize that Prover Plugin worked fine with the standard problem but not with properties involving a long period of time or constants that did not match the values in the model (lack of PDR/IC3). Kind2 is excellent for long duration properties and can take into account some Scade operators. JKind is on average the fastest for short duration properties and properties involving constants that are not exactly the same as in the code.

4.3 Deductive methods

We applied deductive methods to a function that computes the square root from 0.00 to 100.00 by linear integer interpolation between two known points using the following formula:

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

We want to prove that the result is correct for a given precision (here between -0.3 and 0.1) using contracts and the weakest precondition calculus implemented in Frama-C WP and SPARK. We proceeded in two steps. Firstly, we tried to prove it with less values in the interpolation table and finally we extended it to the full table.

The proof with less values succeeded after taking into account some precision issues. Next, we tried to prove the complete table present in the code. Although the values were correct, the automatic proof using Alt-Ergo, CVC4, Z3 and other SMT solvers did not succeed to prove the post condition of the interpolation function. Alt-Ergo and CVC4 were unable to give an answer and Z3 gave up after 5 minutes (Timeout). We submitted the problem to the Frama-C development team, and they found a solution in providing us with a new development version of Frama-C with a special SMT solver called Colibri that succeeded to prove the complete code. Generally speaking, we remarked that the proof success depends much on the SMT solver and even on its version as it relies on heuristics. For instance, we had to change our code annotations when changing only the version of the Alt-Ergo SMT solver.

Finally, we experimented with SPARK (a subset of the Ada language) and the proof of the complete code succeeded. The hint was in using bit vectors over the integers, as they are known to work better with CVC4 and Z3. One of the advantages of SPARK was the *counterexample* returned when the contract checking failed.

5 OUR VISION FOR DEPLOYING FORMAL METHODS IN THE AUTOMOTIVE INDUSTRY

We noticed that most automotive software engineers are not familiar with formal methods and their potential. We think that a progressive introduction in the software engineering process could be successful by identifying at each stage the class of problems that could be solved using such techniques. These use cases can be used to show what kinds of problems can be solved and how.

System requirements are handwritten and can be inconsistent. If we formalize the system requirements we can use model checking like in SpeAR 2.0 [1] to write formal properties and check their logical entailment and consistency. Logical entailment proves that the formalized properties are consequences of the set of captured assumptions and requirements. Logical consistency aims at identifying conflicting assumptions and requirements. Thus safety requirements can be checked using model checking on high level specifications.

At the software design level, we need to have a specific architectural design supporting formal methods i.e. knowing that complex algorithms and nonlinear arithmetic are difficult to prove, we need to isolate or abstract them as soon as possible. Abstraction can be obtained using contracts, allowing for modular and compositional approaches such as those proposed in CoCoSpec [4].

It is important to introduce contracts at the code level, and this activity can be partly generated from the Autosar XML files that specify the interfaces between components. Another way is to reuse and transfer the contracts defined at model level to the source code level, assuming that one Scade node is transformed into one C-language function.

As we have no guarantee to avoid runtime errors even if we use automatic code generation, we suggest using abstract interpretation at the code level to eliminate all potential runtime errors.

We can also introduce tools like Frama-C E-ACSL to check failures at runtime and model-based testing tools like MaTeLo to get additional guarantees of software correctness.

Lastly, we suggest that in the future autonomous vehicle architecture, which will probably implement difficult to prove machine learning algorithms, there should be a *supervisor module* integrating the safety-critical software. This supervisor could be formally verified using automatic proof techniques if this is taken into account before its design.

6 CONCLUSION

In this paper, we shared our experience about the practical use of formal methods. All the experiments quickly confirmed bugs that had already been found by testing, but with great effort and cost because of the late discovery. We cannot show them for confidentiality reasons. We are convinced that we need to use formal methods to bring better guarantees for the future assisted and autonomous driving systems and reduce the overall V&V cost.

Some of these methods, such as static analysis based on abstract interpretation, are mature enough to be used today and can guarantee that source code is free of runtime errors. We discovered that even automatically generated code can present runtime errors difficult to find only by testing. Advanced academic techniques such as PDR/IC3 are not yet commercialized, but we found some use cases for which they worked better than the commercialized ones. Frama-C WP experienced some difficulties in proving our code automatically. Nevertheless, it detected wrong values in a interpolation table that were not found during classical test sessions. This method is very promising because it can replace unit tests as shown in [17] and provide additional guarantees. We think that formal tools could be specialized to get more information from the designers. This could give rise to new heuristics, enabling the verification of larger applications.

Formal methods are implemented in sophisticated tools, which can contain bugs. These tools need to be stable and preserve compatibility with previous versions. Many tools coming from the academia are developed by Ph.D. students and postdoctoral researchers, and are sometimes not maintained after the end of the project. On the commercial side, there are few tools and their algorithms are not well documented, so it is difficult to know how they work and how to compare them. Both academia and software companies need realistic industrial benchmarks and use cases, which are difficult to obtain because of confidentiality constraints. We need to work together.

REFERENCES

[1] Aaron W. Fifarek, Lucas G. Wagner, Erika R. Hoffman, Benjamin D. Rodes, M. Anthony Aiello, and Jennifer A. Davis. 2017. SpeAR v2.0: Formalized Past LTL Specification and Analysis of Requirements. In *NFM 2017.*

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

[3] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. 1987. LUSTRE: A Declarative Language for Real-time Programming. In *POPL '87.* ACM, 178–188. https://doi.org/10.1145/41625.41641

[4] Adrien Champion, Arie Gurfinkel, Temesghen Kahsai, and Cesare Tinelli. [n. d.]. CoCoSpec: A Mode-Aware Contract Language for Reactive Systems. In *SEFM 2016.* Springer. http://dx.doi.org/10.1007/978-3-319-41591-8_24

[5] Adrien Champion, Alain Mebsout, Christoph Sticksel, and Cesare Tinelli. 2016. The Kind 2 Model Checker. In *CAV 2016, Toronto, ON, Canada.* Springer, 510–517. http://dx.doi.org/10.1007/978-3-319-41540-6_29

[6] Roderick Chapman. 2000. Industrial Experience with SPARK. *Ada Letters* XX, 4 (2000). https://doi.org/10.1145/369264.369270

[7] Edmund M. Clarke and E. Allen Emerson. 1982. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logic of Programs Workshop.* http://dl.acm.org/citation.cfm?id=648063.747438

[8] Darren Cofer and Steven P. Miller. 2014. *Formal Methods Case Studies for DO-333.* Technical Report. http://ntrs.nasa.gov/search.jsp?R=20140004055

[9] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL '77.* ACM, 238–252. https://doi.org/10.1145/512950.512973

[10] Andrew Gacek, John Backes, Mike Whalen, Lucas Wagner, and Elaheh Ghassabani. 2018. The JKind Model Checker. http://arxiv.org/pdf/1712.01222

[11] David Gries. 1987. *The Science of Programming.* Springer-Verlag.

[12] Kryštof Hoder and Nikolaj Bjørner. 2012. Generalized Property Directed Reachability. In *Proceedings of SDAT'12 (SAT'12).* Springer-Verlag, 157–171. https://doi.org/10.1007/978-3-642-31612-8_13

[13] Temesghen Kahsai, Pierre-Loïc Garoche, Cesare Tinelli, and Mike Whalen. 2012. Incremental Verification with Mode Variable Invariants in State Machines. In *Proceedings of the 4th International Conference on NASA Formal Methods (NFM'12).* Springer-Verlag, Berlin, Heidelberg, 388–402. https://doi.org/10.1007/978-3-642-28891-3_35

[14] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2015. Frama-C: A software analysis perspective. *Formal Aspects of Computing* 27, 3 (2015). https://doi.org/10.1007/s00165-014-0326-7

[15] Robert A. Martin and Sean Barnum. 2008. Common Weakness Enumeration (CWE) Status Update. *Ada Letters* XXVIII, 1 (April 2008), 88–91. https://doi.org/10.1145/1387830.1387835

[16] Motor Industry Software Reliability Association. 2008. *MISRA-C:2004 : guidelines for the use of the C language in critical systems* (2nd ed. ed.). MIRA Ltd.

[17] Yannick Moy, Emmanuel Ledinot, Herve Delseny, Virginie Wiels, and Benjamin Monate. 2013. Testing or Formal Verification: DO-178C Alternatives and Industrial Experience. *IEEE Soft.* 30, 3 (2013). https://doi.org/10.1109/MS.2013.43

[18] J. P. Queille and J. Sifakis. 1982. Specification and verification of concurrent systems in CESAR. In *International Symposium on Programming, Turin.* Springer. https://doi.org/10.1007/3-540-11494-7_22

[19] Bastian Schlich and Stefan Kowalewski. 2009. Model Checking C Source Code for Embedded Systems. *Int. J. Softw. Tools Technol. Transf.* 11, 3 (June 2009), 187–202. https://doi.org/10.1007/s10009-009-0106-5

[20] Robert C. Seacord. 2008. *The CERT C Secure Coding Standard.* Addison-Wesley.

[21] Jean Souyris and Denis Favre-Felix. 2004. Proof of properties in avionics. In *Building the Information Society, IFIP 18th World Computer Congress, Toulouse, France.* https://doi.org/10.1007/978-1-4020-8157-6_48