Proving properties of Discrete-Valued Functions using Deductive Proof: Application to the Square Root

Vassil Todorov[1], Safouan Taha[2], Frédéric Boulanger[2] and Armando Hernandez[1]

[1] Groupe PSA, France
[2] LRI, CentraleSupélec, Université Paris-Saclay, France

Abstract. For many years, automotive embedded systems have been validated only by testing. In the near future, Advanced Driver Assistance Systems (ADAS) will take a greater part in the car's software design and development. Furthermore, their increasing critical level may lead authorities to require a certification for those systems. We think that bringing formal proof in their development can help establishing safety properties and get an efficient certification process. Other industries (e.g. aerospace, railway, nuclear) that produce critical systems requiring certification also took the path of formal verification techniques. One of these techniques is *deductive proof*. It can give a higher level of confidence in proving critical safety properties and even avoid unit testing.
In this paper, we chose a production use case: a function calculating a square root by linear interpolation. We use deductive proof to prove its correctness and show the limitations we encountered with the off-the-shelf tools. We propose approaches to overcome some limitations of these tools and succeed with the proof. These approaches can be applied to similar problems, which are frequent in automotive embedded software.

Keywords: Formal Methods · Deductive Proof · Proving Discrete-Valued Functions

1 Introduction and Motivation

Today, the automotive industry relies mostly on a model-based approach for developing embedded software. It consists in connecting common library blocks (operators) to design and simulate a model of the behavior to be produced. It uses a higher level of abstraction than the code. Code with the behavior of the model is then produced automatically. The most commonly used tools for software design are Simulink, from the MathWorks, and Scade, from ANSYS.

The main advantage of this approach is that models can be simulated and debugged before code generation. Thus, some of the errors are found and fixed earlier in the design process. On the other hand, simulation shares many common points with testing and cannot prove that the calculation is correct. Furthermore,

the implementation of a model on a specific hardware can bring behaviors that have not been seen before at design stage.

For the rest of the study we take as example a function calculating a square root. During the design stage, the simulation can use a standard implementation of this function. However, in the implementation, we replace it with an optimized version because of hardware constraints. Our example is a discrete-valued function implementing the square root calculation, which uses a linear interpolation table. In automotive applications, as on-board computers have limited power, discrete-valued functions are frequently used in the implementation to avoid complex calculations.

In the near future, we expect that authorities will require a certification for highly critical software in self-driving cars. Our motivation is to provide proofs of correctness for production code using formal methods.

In a previous paper [17], we summarized some experiments about applying tools that use formal methods to industrial software. In this paper, 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 [12] 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 [5]. We discuss the results and how other methods such as Abstract interpretation can be combined with deductive proof.

2 Deductive methods

2.1 Preliminaries

The foundations of the proof of logical properties on an imperative language program were put forward by C. A. R. Hoare [11] in 1969. Based on the precise semantic 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 WP (*Weakest Precondition*) from Dijkstra [8]. The principle

consists in automatically calculating the most general property WP(S,P) holding before a statement S such that 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.

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 [16,2,10]. In order to automate the process, all modern tools based on WP are using automatic theorem provers as back-end. We can cite, for example Alt-Ergo [6], Colibri[3], CVC4 [3], Yices2 [9], Z3 [7].

2.2 Tools for deductive reasoning

As we are interested in tools used by the industry, we present here only those that are mostly used today: Atelier B[4], Caveat [15], Frama-C WP and GNATprove.

Atelier B. Atelier B is a tool supporting the B method, which is a formal methodology to specify, build and implement software systems. The B method was originally developed in the 1980s by Jean-Raymond Abrial [1] and is based on first-order logic, set theory, abstract machine theory and refinement theory. This method is suitable for a new development. As we reused existing C code, we did not use this method.

Caveat and Frama-C WP. Caveat and Frama-C WP are tools for deductive reasoning on C programs. Caveat was introduced at Airbus in 2002 to replace unit tests by unit proof and thus obtain a cost reduction and quality improvement over this part of the development process. The tool with its back-end Alt-Ergo were certified and recognized by the aviation certification authorities. Caveat analyzed C programs (with some restrictions in terms of language constructs) and had its own specification language based on a first order logic. Frama-C is the academic open source tool developed by the same team as Caveat. Its WP module verifies properties written in the ACSL[5] language in a deductive manner. It implements the Weakest Precondition calculus and targets multiple automatic solvers via the Why3 platform[6].

[3] Colibri: `http://smtcomp.sourceforge.net/2018/systemDescriptions/COLIBRI.pdf`
[4] Atelier B: `https://www.atelierb.eu`
[5] ACSL specification language: `https://frama-c.com/acsl.html`
[6] Why3: `http://why3.lri.fr/`

GNATprove. GNATprove is a tool for deductive reasoning over SPARK (based on Ada) programs. Like Frama-C, it uses the Why3 platform but SPARK supports bit-vector data types. A bit-vector is an array data type for compactly storing bits. Most modern SMT-solvers support a theory of bit-vectors, which can help solving problems using this data type. Furthermore, for properties that are not valid, GNATprove can obtain a counterexample from the SMT solver.

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

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 fix-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 present an example for the calculation in the [0, 1.00] subrange using eight known values in Fig. 1.

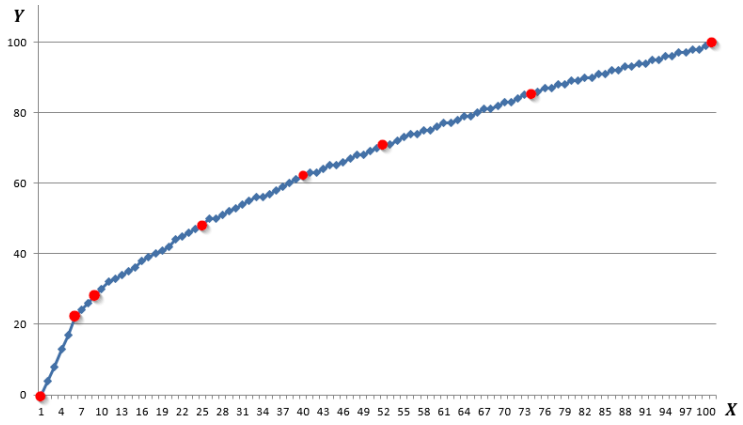

Fig. 1. Square root calculation in [0, 1.00] by linear interpolation from eight values

We proceeded in two steps. First, we proved a simplified version of the code using only eight values in the interpolation table (Fig. 1). 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. At 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 4). Then we rewrote the function in SPARK[7] to see whether it would scale better. The main difference between C and SPARK is that we can specify a `bit-vector` data type in SPARK. For our use case, it helped the solver to reason using modular arithmetic. Most SMT solvers used as back-end via 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. We also analyzed our complete C code with Astrée [13] from AbsInt, a static analysis tool using abstract interpretation, to prove some difficult goals and provide useful hypotheses to Frama-C WP.

Our first proof on the simplified code succeeded with Frama-C. Extending the table to 41 values, as in the real code, did not succeed. On the other hand, SPARK succeeded with the full table of 41 values.

4 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.

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 verification conditions (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 redefined some operators such as `division` using uninterpreted functions.

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 Fig. 2.

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.

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 [4]. 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

[7] Special thanks to Yannick Moy from AdaCore.

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.

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.

Abstract interpretation. 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.

5 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

```
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);
  }
}
```

Fig. 2. Annotated interpolation function for Frama-C WP automatic proof

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 Fig. 3 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 (VCs) 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.

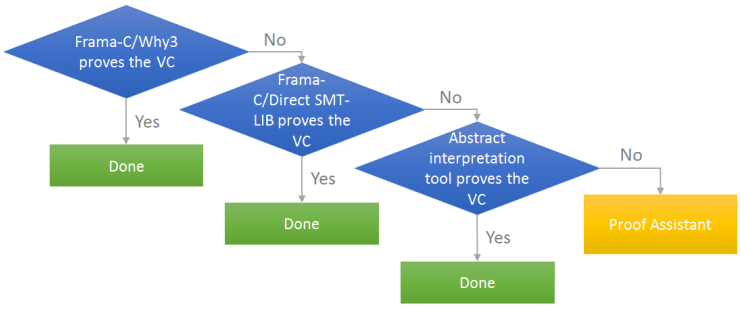

Fig. 3. Methodology for proving Discrete-Valued Functions

6 Conclusions and Future work

In this paper, we have presented our experiments with automatic deductive proof of correctness of a discrete-valued function calculating a square root by interpolation. We used Frama-C WP and GNATprove to prove the function correct but 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 tests as shown in [14] and thus decrease cost while increasing quality. It is also an intellectual activity that brings more satisfaction for engineers compared to testing.

References

1. Abrial, J.R.: The B-book : assigning programs to meanings (1996)
2. Barnett, M., Leino, K.R.M.: Weakest-precondition of Unstructured Programs. SIGSOFT Softw. Eng. Notes **31**(1), 82–87 (2005)
3. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanovi'c, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) (CAV'11). LNCS, vol. 6806, pp. 171–177. Springer (Jul 2011)
4. Barrett, C., Stump, A., Tinelli, C., Boehme, S., Cok, D., Deharbe, D., Dutertre, B., Fontaine, P., Ganesh, V., Griggio, A., Grundy, J., Jackson, P., Oliveras, A., Krstić, S., Moskal, M., Moura, L.D., Sebastiani, R., Cok, T.D., Hoenicke, J.: The SMT-LIB Standard: Version 2.0. Tech. rep. (2010)
5. Chapman, R.: Industrial Experience with SPARK. Ada Letters **XX**(4) (2000)
6. Conchon, S., Coquereau, A., Iguernlala, M., Mebsout, A.: Alt-Ergo 2.2. In: SMT Workshop: International Workshop on SMT. Oxford, United Kingdom (Jul 2018)
7. De Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. TACAS'08/ETAPS'08, Springer-Verlag, Berlin, Heidelberg (2008)
8. Dijkstra, E.W.: Guarded Commands, Nondeterminacy and Formal Derivation of Programs. Commun. ACM **18**(8), 453–457 (Aug 1975)
9. Dutertre, B.: Yices 2.2. In: Biere, A., Bloem, R. (eds.) Computer Aided Verification. pp. 737–744. Springer International Publishing, Cham (2014)
10. Flanagan, C., Flanagan, C., Saxe, J.B.: Avoiding Exponential Explosion: Generating Compact Verification Conditions. SIGPLAN Not. **36**(3), 193–205 (2001)
11. Hoare, C.A.R.: An Axiomatic Basis for Computer Programming (Oct 1969)
12. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A software analysis perspective. Formal Aspects of Computing **27**(3) (2015)
13. Mauborgne, L.: Astrée: Verification of Absence of Runtime Error. In: Jacquart, R. (ed.) Building the Information Society: IFIP 18th World Computer Congress Topical Sessions 22–27 August 2004 Toulouse, France, pp. 385–392. Springer (2004)
14. Moy, Y., Ledinot, E., Delseny, H., Wiels, V., Monate, B.: Testing or Formal Verification: DO-178c Alternatives and Industrial Experience. IEEE Soft. **30**(3) (2013)
15. Randimbivololona, F., Souyris, J., Baudin, P., Pacalet, A., Raguideau, J., Schoen, D.: 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. Springer-Verlag, London (1999)
16. Shilov, N.V., Anureev, I.S., Bodin, E.V.: Generation of correctness conditions for imperative programs. Programming and Computer Software **34**(6), 307–321 (2008)
17. Todorov, V., Boulanger, F., Taha, S.: Formal Verification of Automotive Embedded Software. In: Proceedings of the 6th Conference on Formal Methods in Software Engineering. pp. 84–87. FormaliSE '18, ACM, New York, NY, USA (2018)