DOI: 10.18255/1818-1015-2019-4-520-533

UDC 519.987

Proving Properties of Discrete-Valued Functions Using Deductive Proof: Application to the Square Root

Todorov V., Taha S., Boulanger F., Hernandez A.

Received September 17, 2019
Revised November 19, 2019
Accepted November 27, 2019

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 the automotive embedded software.

Keywords: formal methods, deductive proof, proving discrete-valued functions

For citation: Todorov V., Taha S., Boulanger F., Hernandez A., "Proving Properties of Discrete-Valued Functions Using Deductive Proof: Application to the Square Root ", *Modeling and Analysis of Information Systems*, **26**:4 (2019), 520–533.

On the authors:
Vassil Todorov, orcid.org/0000-0002-2739-499X,
Groupe PSA,
Route de Gisy, 78140 Vélizy-Villacoublay, France, e-mail: vassil.todorov@lri.fr

Safouan Taha, orcid.org/0000-0003-3950-6415, PhD,
CentraleSupélec,
3 Rue Joliot Curie, 91190 Gif-sur-Yvette, France, e-mail: safouan.taha@lri.fr

Frédéric Boulanger, orcid.org/0000-0003-3185-2807, PhD,
CentraleSupélec,
3 Rue Joliot Curie, 91190 Gif-sur-Yvette, France, e-mail: frederic.boulanger@lri.fr

Armando Hernandez, orcid.org/0000-0003-4555-4616,
Groupe PSA,
Route de Gisy, 78140 Vélizy-Villacoublay, France, e-mail: armando.hernandez@mpsa.com

Acknowledgments:
This work was supported by the Groupe PSA, a French multinational manufacturer of automobiles and motorcycles sold under the Peugeot, Citroën, DS, Opel and Vauxhall brands.

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 our 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. Fig. 1 shows this approach. 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.

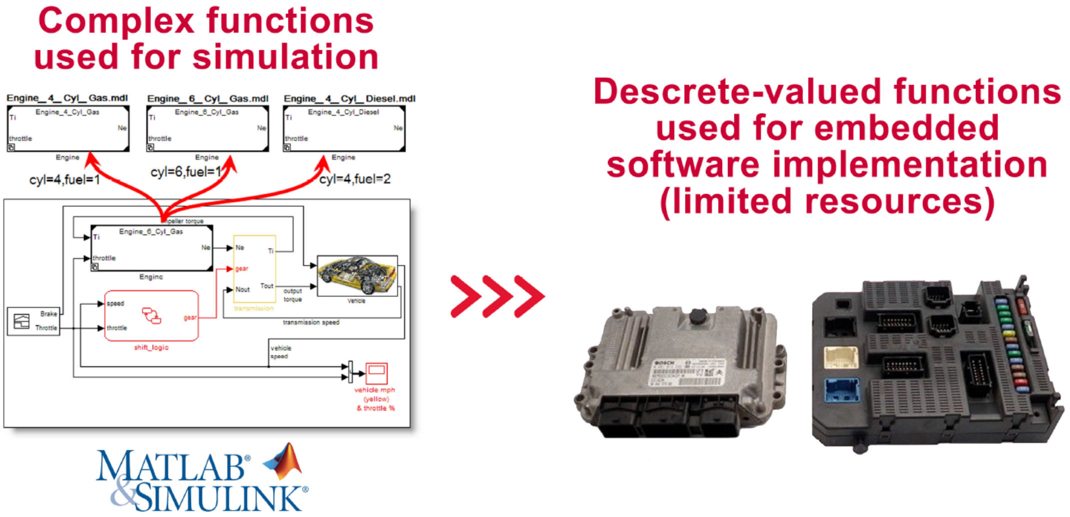

Figure 1. Complex functions for simulation vs. discrete-valued ones for implementation

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 [29], 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 [16] 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 [7]. We discuss the results as well as how other methods such as Abstract interpretation can be combined with deductive proof.

1. Deductive methods

1.1. Preliminaries

The foundations of the proof of logical properties on an imperative language program were put forward by C. A. R. Hoare [15] 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 [10]. 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 [27, 3, 13]. 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 [8], Colibri[1], CVC4 [4], Yices2 [11], Z3 [9].

1.2. 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[2], Caveat [22], Frama-C WP and GNATprove.

[1] Colibri: `http://smtcomp.sourceforge.net/2018/systemDescriptions/COLIBRI.pdf`

[2] Atelier B: `https://www.atelierb.eu`

1.2.1. 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 [2] 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.

1.2.2. 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[3] language in a deductive manner. It implements the Weakest Precondition calculus and targets multiple automatic solvers via the Why3 platform[4].

1.2.3. 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.

2. Environment

We present in Fig. 2 our environment. 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 (VCs) 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 VCs 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.

[3] ACSL specification language: https://frama-c.com/acsl.html

[4] Why3: http://why3.lri.fr/

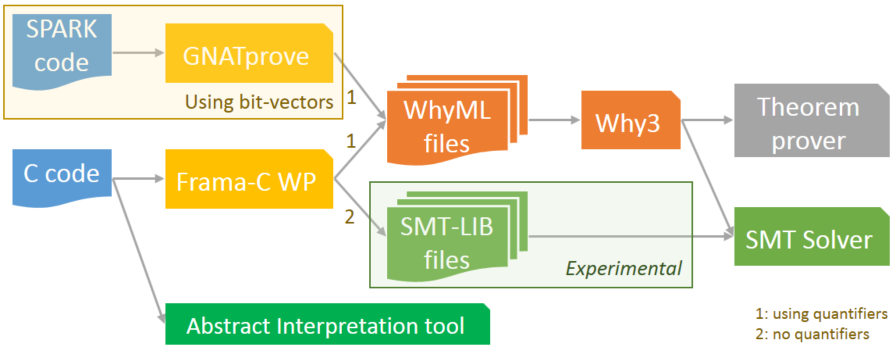

Figure 2. Environment for deductive proof on C and SPARK code

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.

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 proceeded in two steps. First, we proved a simplified version of the code using only eight values in the interpolation table (Fig. 3) 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 4.).

The code of our main function is given in Fig. 4. 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.

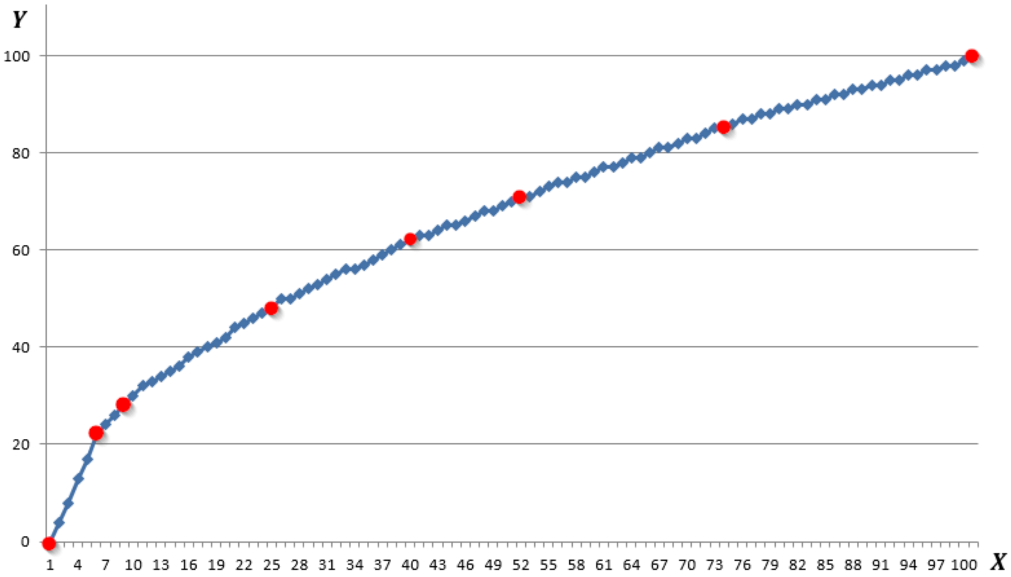

Figure 3. Square root calculation in [0, 1.00] by linear interpolation from eight known values

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.

```
/*@ 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 4. Annotated square root function for Frama-C WP automatic proof

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.

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[5] to see whether it would scale better. Fig. 5 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.

```ada
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 5. SPARK code for automatic proof with GNATprove

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 [18] 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

[5] Special thanks to Yannick Moy from AdaCore

for certification if Frama-C can reason on them. We discuss the results in the next section.

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.

4.1. 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 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 had redefined some operators such as `division` using uninterpreted functions.

4.2. 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 Fig. 6.

```
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 6. 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.

4.3. 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 [5]. 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.

4.4. 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.

4.5. 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 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. 7 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

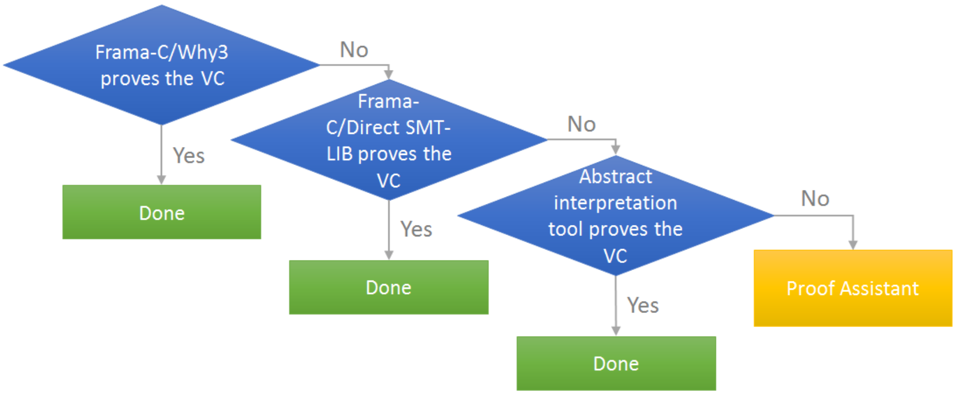
Frama-C/Why3 proves the VC
No
Yes
Done
Frama-C/Direct SMT-LIB proves the VC
No
Yes
Done
Abstract interpretation tool proves the VC
No
Yes
Done
Proof Assistant

Figure 7. Methodology for proving Discrete-Valued Functions

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.

6. 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 GNATprove) 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 [17]. 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 paper 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 [24]. Later, in 1999 he also verified the square root microcode of the K5 microprocessor [23]. 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 [1]. In 2002, IBM presented a research paper about the formal verification of the IBM Power4 processor that uses Chebyshev polynomials to calculate square root [25]. 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 GMP's square root using the Coq proof assistant [6]. In 2003, Harrison published his work about a square root algorithm verification using HOL Light [14]. 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 verification 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 [26]. 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 [21]. 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) [12]. 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 [19]. 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 [28], 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.

Conclusions

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 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 tests as shown in [20] and thus decrease cost while increasing quality. It is also an intellectual activity that brings more satisfaction for engineers compared to testing.

References

[1] Aagaard M. D., Jones R. B., Kaivola R., Kohatsu K. R., Seger C.-J. H., "Formal Verification of Iterative Algorithms in Microprocessors", *Proceedings of the 37th Annual Design Automation Conference (DAC 2000)*, 2000, 201–206.

[2] Hoare A., Chapron P., Abrial J. R., *The B-book: Assigning Programs to Meanings*, Cambridge University Press, New York, NY, USA, 1996.

[3] Barnett M., Leino K. R. M., "Weakest-Precondition of Unstructured Programs", *Softw. Eng. Notes*, **31**:1 (2005), 82–87.

[4] Barrett C., Conway C. L., Deters M., Hadarean L., Jovanovi'c D., King T., Reynolds A., Tinelli C., "Cvc4.", *Computer Aided Verification. CAV 2011. LNCS*, **6806** (2011), 171–177.

[5] Barrett C. et al., "The SMT-LIB Standard: Version 2.0.", *Tech. rep.*, 2010.

[6] Bertot Y., Magaud N., Zimmermann P., "A Proof of GMP Square Root", *Journal of Automated Reasoning*, **29**:3-4 (2002), 225–252.

[7] Chapman R., "Industrial Experience with SPARK", *ACM SIGAda Ada Letters*, **20**:4 (2000), 64–68.

[8] Conchon S., Coquereau A., Iguernlala M., Mebsout A., "Alt-Ergo 2.2", *SMT Workshop: International Workshop on SMT. Oxford, United Kingdom*, 2018.

[9] De Moura L., Bjørner N., "Z3: An Efficient SMT Solver", *TACAS'08/ETAPS'08, Springer-Verlag, Berlin, Heidelberg*, **4963** (2008), 337–340.

[10] Dijkstra E. W., "Guarded Commands, Nondeterminacy and Formal Derivation of Programs", *ACM*, **18**:8 (1975), 453–457.

[11] Dutertre B., "Yices 2.2", *International Conference on Computer Aided Verification. Springer, Cham*, 2014, 737–744.

[12] Ferguson W. E., Bingham J., Erkök L., Harrison J. R., Leslie-Hurd J., "Digit Serial Methods with Applications to Division and Square Root", *IEEE Transactions on Computers*, **67**:3 (2017), 449–456.

[13] Flanagan C., Flanagan C., Saxe J. B., "Avoiding Exponential Explosion: Generating Compact Verification Conditions", *ACM SIGPLAN Not.*, **36**:3 (2001), 193–205.

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

[15] Hoare C. A. R., "An Axiomatic Basis for Computer Programming", *Communications of the ACM*, **12**:10 (1969), 576–580.

[16] Kirchner F., Kosmatov N., Prevosto V., Signoles J., Yakobowski B., "Frama-C: A Software Analysis Perspective", *Formal Aspects of Computing*, **27**:3 (2015), 573–609.

[17] Kuliamin V. V., "Standardization and Testing of Implementations of Mathematical Functions in Floating Point Numbers", *Programming and Computer Software*, **33**:3 (2007), 154–173.

[18] Mauborgne L., "Astrée: Verification of Absence of Runtime Error", *In: Jacquart R. (eds) Building the Information Society. IFIP International Federation for Information Processing*, **156** (2004), 385–392.

[19] Melquiond G., Rieu-Helft R., "Formal Verification of a State-of-the-Art Integer Square Root", *IEEE 26th Symposium on Computer Arithmetic (ARITH), Kyoto, Japan*, 2019, 183–186.

[20] 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), 50-57.

[21] Rager D. L., Ebergen J., Nadezhin D., Lee A., Chau C. K., Selfridge B., "Formal Verification of Division and Square Root Implementations, an Oracle Report.", *Formal Methods in Computer-Aided Design (FMCAD)*, 2016, 149–152.

[22] Randimbivololona F., Souyris J., Baudin P., Pacalet A., Raguideau J., Schoen D., "Applying Formal Proof Techniques to Avionics Software: A Pragmatic Approach", *In: Wing J. M., Woodcock J., Davies J. (eds) — Formal Methods. FM 1999*, **1709** (1999), 1798–1815.

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

[24] Russinoff D. M., "A Mechanically Checked Proof of IEEE Compliance of the Floating Point Multiplication, Division and Square Root Algorithms of the AMD-K7$^{\text{TM}}$ Processor", *LMS J. Comput. Math. (UK)*, **1** (1998), 148–200.

[25] Sawada J., Gamboa R., "Mechanical Verification of a Square Root Algorithm Using Taylor's Theorem", *LNCS. Formal Methods in Computer-Aided Design. FMCAD 2002.*, **2517** (2002), 274–291.

[26] Shelekhov V. I., "Verification and Synthesis of Addition Programs under the Rules of Correctness of Statements", *Automatic Control and Computer Sciences*, **45**:7 (2011), 421–427.

[27] Shilov N. V., Anureev I. S., Bodin E. V., "Generation of Correctness Conditions for Imperative Programs", *Programming and Computer Software*, **34**:6 (2008), 307–321.

[28] Шилов Н. В., Кондратьев Д. А., Ануреев И. С., Бодин Е. В., Промский А. В., "Платформенно-независимая спецификация и верификация стандартной математической функции квадратного корня", *Моделирование и анализ информационных систем*, **25**:6 (2018), 637-666; [Shilov N. V., Kondratyev D. A., Anureev I. S., Bodin E. V., Promsky A. V., "Platform-Independent Specification and Verification of the Standard Mathematical Square Root Function", *Modeling and Analysis of Information Systems*, **25**:6 (2018), 637-666, (in Russian).]

[29] Todorov V., Boulanger F., Taha S., "Formal Verification of Automotive Embedded Software", *Proceedings of the 6th Conference on Formal Methods in Software Engineering. ACM, New York, USA*, 2018, 84–87.

Тодоров В., Таха С., Буланже Ф., Эрнандес А., "Доказательство свойств дискретных функций с помощью дедуктивного доказательства: приложение к квадратному корню", *Моделирование и анализ информационных систем*, **26**:4 (2019), 520–533.

DOI: 10.18255/1818-1015-2019-4-520-533

Аннотация. В течение многих лет автомобильные встраиваемые системы проверялись только тестированием. В ближайшем будущем усовершенствованные системы помощи водителю (ADAS) будут играть большую роль в дизайне и разработке программного обеспечения автомобиля. Кроме того, увеличение их критического уровня может привести к тому, что власти потребуют сертификации этих систем. Мы думаем, что привнесение формальных доказательств в их развитие может помочь обеспечить выполнение свойств безопасности и получить эффективный процесс сертификации. Другие отрасли (например, аэрокосмическая, железнодорожная, ядерная), которые создают критические системы, требующие сертификации, также могут быть заинтересованы в развитии формальных методов проверки. Одним из этих методов является дедуктивное доказательство. Это может дать более высокий уровень уверенности в доказательстве критических свойств безопасности и даже избежать модульное тестирование. В этой статье мы выбрали вариант прикладного использования: функцию, вычисляющую квадратный корень с помощью линейной интерполяции. Мы используем дедуктивное доказательство, чтобы доказать его правильность и показать ограничения, с которыми мы сталкиваемся при работе с готовыми инструментами. Мы предлагаем подходы для преодоления некоторых ограничений, связанных с этими инструментами,чтобы преуспеть с доказательством. Эти подходы могут быть применены к аналогичным проблемам, которые часто встречаются в автомобильном встроенном программном обеспечении.

Ключевые слова: формальные методы, дедуктивное доказательство, доказательство дискретных функций

Об авторах:
Васил Тодоров, orcid.org/0000-0002-2739-499X,
Groupe PSA,
Route de Gisy, 78140 Vélizy-Villacoublay, Франция, e-mail: vassil.todorov@lri.fr

Сафуан Таха, orcid.org/0000-0003-3950-6415, PhD,
CentraleSupélec,
3 Rue Joliot Curie, 91190 Gif-sur-Yvette, Франция, e-mail: safouan.taha@lri.fr

Фредерик Буланже, orcid.org/0000-0003-3185-2807, PhD,
CentraleSupélec,
3 Rue Joliot Curie, 91190 Gif-sur-Yvette, Франция, e-mail: frederic.boulanger@lri.fr

Армандо Эрнандес, orcid.org/0000-0003-4555-4616,
Groupe PSA,
Route de Gisy, 78140 Vélizy-Villacoublay, Франция, e-mail: armando.hernandez@mpsa.com

Благодарности:
Эта работа была поддержана Groupe PSA, французским многонациональным производителем автомобилей и мотоциклов, которые продаются под брендами Peugeot, Citroën, DS, Opel и Vauxhall.