PROVING PROPERTIES OF DISCRETE-VALUED FUNCTIONS

Using Deductive Proof:

Application to the Square Root

TEAM MEMBERS:

CB.EN.U4CSE22522: KANISKA D

CB.EN.U4CSE22544: SHREYA J V

CB.EN.U4CSE22546: SRINIDHI K

ABSTRACT

This paper discusses the need for formal verification in the development of automotive embedded systems, especially with the increasing role of Advanced Driver Assistance Systems (ADAS). The authors suggest that formal proof methods can contribute to establishing safety properties and facilitate certification processes, similar to practices in other critical industries such as aerospace, railway, and nuclear.
The paper focuses on a practical case: verifying a square root calculation function that uses linear interpolation. The authors employ deductive proof techniques to demonstrate correctness but highlight limitations encountered with existing tools. They propose approaches to overcome these limitations and successfully complete the proof. The suggested methods can be applied to similar challenges often found in automotive embedded software development.

Problem description:

The problem addressed in this paper is the difficulty of proving the properties of discrete-valued functions using deductive proof. Deductive proof is a formal method of reasoning that uses logical deduction to prove the correctness of a statement. However, it can be difficult to apply deductive proof to discrete-valued functions, which are functions that take on a finite number of values.

Research Objectives:

The research objectives of this paper are to:

- Propose a method for representing discrete-valued functions in a way that is amenable to deductive proof.
- Develop a deductive proof technique for proving the properties of discrete-valued functions.
- Apply the proposed method and technique to the problem of proving the correctness of a square root function.

HOARE TRIPLE:

$P \{Q\} R$

If the preconditions (P) are true before executing the program (Q), then the postconditions (R) will be true after the program has executed, provided that the program (Q) terminates successfully.

WEAKEST PRECONDITION:

Weakest Precondition is the process of proving the correctness of the program.

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

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.

Tools for deductive reasoning

Atelier B

It supports the B-Method, which is a formal method based on formal specification languages.

Caveat

It is a static analysis tool designed to help verify safety critical software. It operates on ANSI C programs.

Frama-C WP:

It employs the Weakest Precondition (WP) calculus to automatically generate verification conditions and perform formal verification on C code.

Linear Interpolation

Linear interpolation is a method for estimating values that lie between two known values. It is commonly used in various fields, including mathematics, computer graphics, computer-aided design, and signal processing. The idea behind linear interpolation is to assume that the relationship between two known values is linear and then estimate the value that lies between them.

Given two known points (x1,y1) and (x2,y2), where x1/x2 linear interpolation allows you to estimate the value of y at a point x that lies between x1 and x2.

The formula for linear interpolation is:

$$y=y1+(x-x1)[(y2-y1)/(x2-x1)]$$

In this formula:

- x is the point at which you want to estimate y.
- x1 and x2 are the x-coordinates of the known points.
- y1 and y2 are the corresponding y-coordinates of the known points.

The formula essentially computes the equation of the straight line passing through the points (x1,y1) and (x2,y2) and then evaluates the y-coordinate at the given x.

```
/*@
requires 0 <= Xa <= 10000 && 0 <= Xb <= 10000;
requires 0 <= Ya <= 1000 && 0 <= Yb <= 1000;
behavior a:
assumes Xa != Xb;
ensures \result == (Ya + (X - Xa) * (Yb - Ya) / (Xb - Xa));
behavior b:
assumes Xa == Xb;
ensures \result == Ya;
complete behaviors
disjoint behaviors;
assigns \nothing;
*/

if Xa != Xb then
    Result := Ya + (X - Xa) * (Yb - Ya) / (Xb - Xa);
else
    Result := Ya;
end if;
return Result;
```

requires: This clause specifies the preconditions that must be satisfied for the function to be called.

requires 0 <= Xa <= 10000 && 0 <= Xb <= 10000: Ensures that the value of Xa,Xb is within the range [0, 10000].

requires 0 <= Ya <= 1000 && 0 <= Yb <= 1000: Ensures that the value of Ya,Yb is within the range [0, 1000].

behavior a: This behavior is associated with the case when Xa!=Xb

assumes Xa != Xb: assumption that Xa != Xb

ensures \result == (Ya + (X - Xa) * (Yb - Ya) / (Xb - Xa)): Ensures the result should be equal to the value calculated using linear interpolation based on the provided input values (Xa, Xb, Ya, Yb) and the input value X.

behavior a: This behavior is associated with the case when Xa==Xb

assumes Xa == Xb: Assumption that Xa == Xb

ensures \result == Ya: The result of the function should be equal to the value of Ya.

complete behaviors: This clause indicates that behavior_a and behavior_b, together form a complete set of possible behaviors.

disjoint behaviors: When two behaviors are disjoint, it means that the conditions under which each behavior is specified do not overlap

assigns \nothing: The function does not modify any global variables.

```
/*@ 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;
*/
```

```
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
```

1. **assigns \nothing;** The function does not modify any global variables.
2. **behavior in_range:** This behavior is associated with the case when number is less than or equal to 10000.

- **assumes number <= 10000;** Assumption that number is less than or equal to 10000.
- **ensures number-30 <= (\result)*(\result)/100 <= number+10;** Ensures that the result of the function, when squared and divided by 100, falls within the range [number-30, number+10].

3. **behavior out_of_range:** This behavior is associated with the case when **number** is greater than 10000.

- **assumes number > 10000;** Assumption that number is greater than 10000.
- **ensures \result == 1000;** Ensures that the result of the function is 1000.

4. **complete behaviors;** Specifies that the behaviors cover all possible cases of input values that is either in_range or out_of_range.
5. **disjoint behaviors;** Specifies that the in_range and out_of_range behaviors are disjoint, meaning they do not overlap.

```
/*@
loop invariant 0 <= i <= 40 && number >= TabX[i];
loop assigns i;
loop variant 40-i;
 */

function IntSqrt(number: Integer) return Integer is
begin
for I in 1 .. 40 loop pragma Loop_Invariant (for all J in 1 .. I => TabX(J) <=
number);
 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;
```

1. **/*@ loop invariant 0 <= i <= 40 && number >= TabX[i];**: This loop invariant specifies conditions that must be true at the beginning and end of each iteration of the loop. It ensures that **i** is within the range of 0 to 40, and the value of **number** is greater than or equal to **TabX[i]**.
2. **loop assigns i;**: Indicates that the loop modifies the loop variable **i**.
3. **loop variant 40-i;**: Specifies a loop variant, which is a quantity that decreases on each iteration and is used to ensure termination. Here, the loop variant is **40 - i**, indicating that the loop will terminate when **i** becomes greater than 40.
4. **for I in 1 .. 40 loop**: Initiates a loop that iterates over the range from 1 to 40 (inclusive) with the loop variable **I**.
5. **pragma Loop_Invariant (for all J in 1 .. I => number >= TabX(J));**: This pragma provides additional loop invariants, stating that for all **J** in the range from 1 to **I**, the value of **number** should be greater than or equal to **TabX(J)**. This is an assertion that the loop maintains this condition.
6. **if number in TabX(I) .. TabX(I+1) then**: Checks if the value of **number** falls within the range defined by **TabX(I)** and **TabX(I+1)**.
7. **return LinearInterpolation(TabX(I), TabY(I), TabX(I+1), TabY(I+1), number);**: If the condition in the **if** statement is true, the function returns the result of a linear interpolation between the points **(TabX(I), TabY(I))** and **(TabX(I+1), TabY(I+1))** based on the value of **number**.
8. **end loop;**: Marks the end of the loop.
9. **return TabY(41);**: If the loop completes without satisfying the condition in the **if** statement, the function returns the value at index 41 in the **TabY** array.

SUMMARY:

This research paper addresses the challenging problem of proving properties of discrete-valued functions using deductive proof methods. The authors recognize that while deductive proof is a powerful tool for establishing correctness in mathematical and computational contexts, it encounters difficulties when applied to functions that have a finite number of possible output values.The primary justification for this research lies in the critical importance of ensuring the correctness of functions in safety-critical systems, like those found in automotive embedded software. Deductive proof methods offer a high level of confidence in the properties of these functions, which is crucial for avoiding catastrophic failures and ensuring the safety of users.The research objectives outlined in the paper is to apply these methodologies to address a concrete problem – proving the correctness of a square root function.

CONCLUSION:

The paper discusses experiments on automatic deductive proof for a discrete square root interpolation function using Frama-C WP and GNATprove. Challenges arose with the nonlinear formula of linear interpolation, leading to successful resolutions through three non-standard methods: bit-vectors in SPARK, direct SMT-LIB quantifier-free output in Frama-C, and static analysis with Astrée. While bit-vectors are effective for modular arithmetic, SMT requests without quantifiers proved more scalable for the given case.
Abstract Interpretation analysis enhanced confidence in proving the absence of overflow. The proposed methodology advocates combining these methods iteratively until successful proof. The study highlights challenges in scaling off-the-shelf tools for industrial cases but suggests collaboration with researchers for solutions. Deductive methods show promise in industrial settings, potentially replacing unit tests to decrease costs and improve quality, providing intellectual satisfaction for engineers compared to testing.