

Software quality and formal methods: Hoare/Dijkstra approach

Dr. Ivan S. Zapreev

Neat Software Designs

2020-01-24

Global Outline:

- Software Quality
- Programming Languages
- Formal Verification
- Frama-C Framework
- Verification Practice
- Concluding Remarks

Software
quality and
formal
methods:
Hoare/Dijkstra
approach

Dr. Ivan S.
Zapreev

Software
Quality

Programming
Languages

Formal
Verification

Frama-C
Framework

Verification
Practice

Concluding
Remarks

Software Quality

Software Quality: Outline

Software
quality and
formal
methods:
Hoare/Dijkstra
approach

Dr. Ivan S.
Zapreev

Software
Quality

Programming
Languages

Formal
Verification

Frama-C
Framework

Verification
Practice

Concluding
Remarks

- Motivating examples
- Software Development
- Software Verification

Motivating examples: Therac-25

Years 1985–1987: Radiation overdose

- Control software flaw:
 - Race conditions
- Death of 6 (six) cancer patients



Figure 1: Radiation therapy

Motivating examples: Ariane-5

Year 1996: Missile crash

- Control software flaw:
 - 64-bit float to 16-bit integer conversion
- \$137 million Rocket + \$500 million cargo



Figure 2: Space flights

Motivating examples: Toyota

Year 2005: Sudden unintended acceleration

- Control software flaw:
 - Recursion causing stack overflow
- 89 deaths and 57 injuries
- \$1.2 billion compensations



Figure 3: Toyota Camry

Motivating examples: Plenty More

BUGS



BUGS EVERYWHERE

The 12 Software Bugs That Caused Epic Failures: [<link>](#)

Software Development: V-model

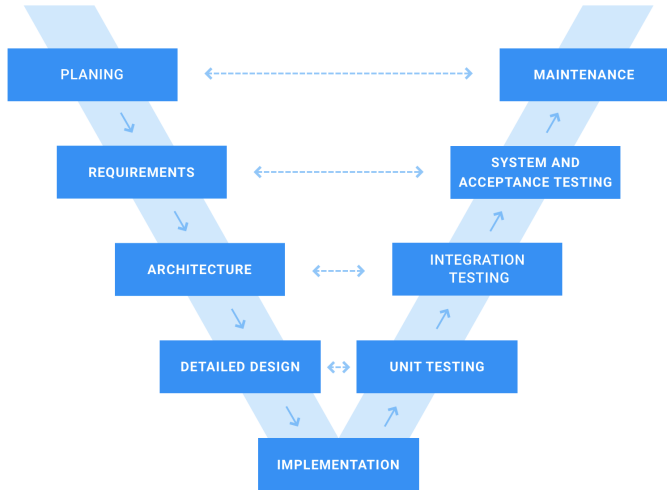


Figure 4: Software development process

Software
quality and
formal
methods:
Hoare/Dijkstra
approach

Dr. Ivan S.
Zapreev

Software
Quality

Programming
Languages

Formal
Verification

Frama-C
Framework

Verification
Practice

Concluding
Remarks

Software Development: V & V

Software
quality and
formal
methods:
Hoare/Dijkstra
approach

Dr. Ivan S.
Zapreev

Software **Validation** and **Verification**.

Formally defined by the International Organization for Standardization, see ISO-9000:2015:

- **Verification** – *“Confirmation, through the provision of objective evidence, that specified requirements have been fulfilled.”*
- **Validation** – *“Confirmation, through the provision of objective evidence, that the requirements for a specific intended use or application have been fulfilled.”*

Software
Quality

Programming
Languages

Formal
Verification

Frama-C
Framework

Verification
Practice

Concluding
Remarks

Software Development: Testing

- **Verification:**

- Are we building the product right?
- Does the system comply with its specification?

- **Validation:**

- Are we building the right product?
- Does the system meet the needs of the customer?

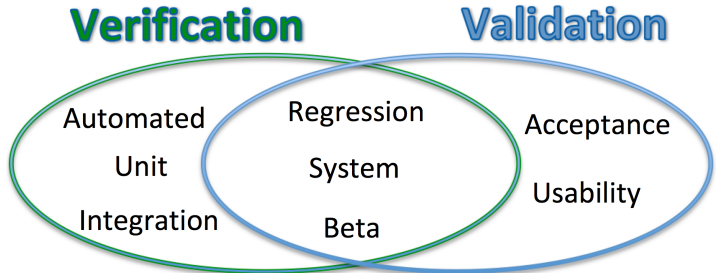


Figure 5: Types of testing

Software Verification: Formal

Software
quality and
formal
methods:
Hoare/Dijkstra
approach

Dr. Ivan S.
Zapreev

Software
Quality

Programming
Languages

Formal
Verification

Frama-C
Framework

Verification
Practice

Concluding
Remarks

Formal verification¹:

"Is the act of proving or disproving the correctness of intended algorithms underlying a system with respect to a certain formal specification or property, using formal methods of mathematics."

Formal methods²:

"Formal methods are techniques used to model complex systems as mathematical entities."

"By building a rigorous model of a complex system, it is possible to verify the system's properties in a more thorough fashion than empirical testing."

¹"Formal Verification" on Wikipedia

²"Formal Methods", Michael Collins, CMU

Software
quality and
formal
methods:
Hoare/Dijkstra
approach

Dr. Ivan S.
Zapreev

Software
Quality

Programming
Languages

Formal
Verification

Frama-C
Framework

Verification
Practice

Concluding
Remarks

Programming Languages

Programming Languages: Outline

Software
quality and
formal
methods:
Hoare/Dijkstra
approach

Dr. Ivan S.
Zapreev

Software
Quality

Programming
Languages

Formal
Verification

Frama-C
Framework

Verification
Practice

Concluding
Remarks

- Language Generations
- Declarative vs. Imperative
- What is ANSI-C?

Language Generations

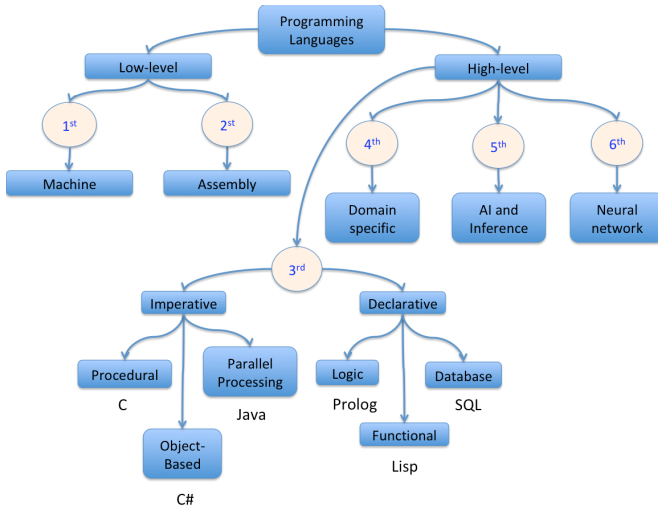


Figure 6: Generations of Programming languages

Declarative vs. Imperative: Main

Software
quality and
formal
methods:
Hoare/Dijkstra
approach

Dr. Ivan S.
Zapreev

Software
Quality

Programming
Languages

Formal
Verification

Frama-C
Framework

Verification
Practice

Concluding
Remarks

Consider a problem of multiplying all array elements by 2:

- **Declarative** – Specifies **what** to achieve:

```
//Declarative `JavaScript`  
var arr_dbl = arr.map((x) => x * 2)
```

- **Imperative** – Defines the **how** steps:

```
//Imperative `JavaScript`  
var arr_dbl = []  
for (let i = 0; i < arr.length; i++) {  
    arr_dbl.push(arr[i] * 2)  
}
```


Declarative vs. Imperative: Test

Software
quality and
formal
methods:
Hoare/Dijkstra
approach

Dr. Ivan S.
Zapreev

Software
Quality

Programming
Languages

Formal
Verification

Frama-C
Framework

Verification
Practice

Concluding
Remarks

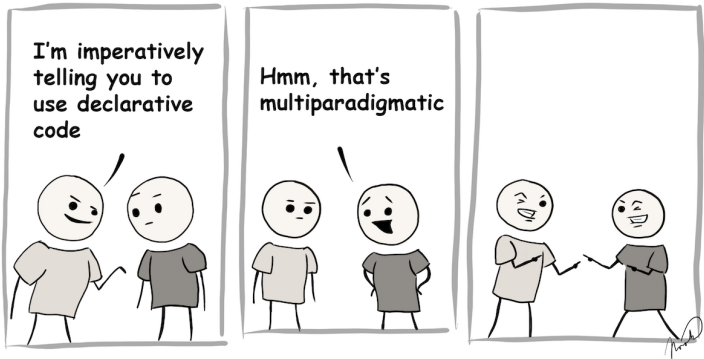


Figure 7: If you laugh, it means you've passed

What is ANSI-C: An old C

C language:

C is an *imperative procedural* language.

Procedural language:

Is an imperative language in which the program is built from one or more subroutines commonly known as *functions*.

Defining ANSI-C:

ANSI-C is a common name for two equivalent standards:

- C89 – American National Standards Institute (ANSI)
- C90 – International Organization for Standardization (ISO)

Software
quality and
formal
methods:
Hoare/Dijkstra
approach

Dr. Ivan S.
Zapreev

Software
Quality

Programming
Languages

Formal
Verification

Frama-C
Framework

Verification
Practice

Concluding
Remarks

Formal Verification

Formal Verification: Outline

Software
quality and
formal
methods:
Hoare/Dijkstra
approach

Dr. Ivan S.
Zapreev

Software
Quality

Programming
Languages

Formal
Verification

Frama-C
Framework

Verification
Practice

Concluding
Remarks

- Verification Goal
- Hoare Approach
- Dijkstra Extension

Verification Goal: Global

A *program* shall satisfy a formal specification of its *behavior*.

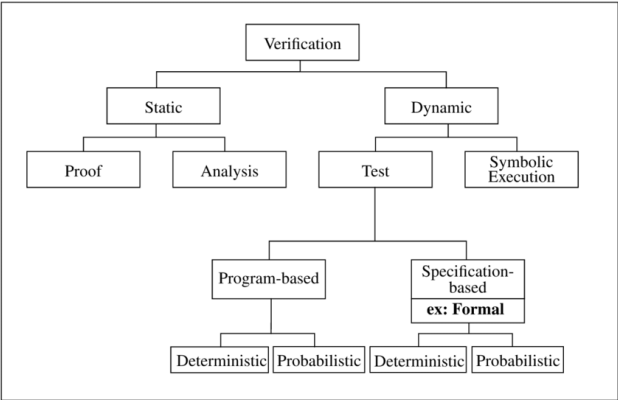


Figure 8: Verification methods

Verification Goal: Proving

Mathematically prove conformance to formal specifications.

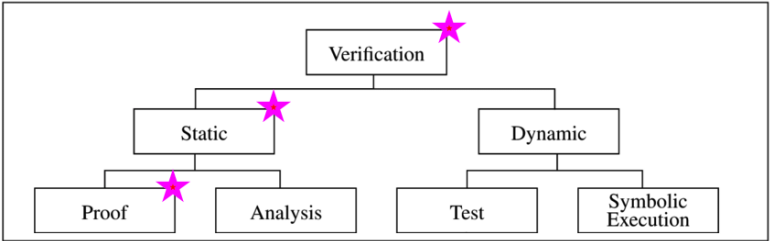


Figure 9: Formal correctness proving

Consider:

The Hoare/Dijkstra approach for proving correctness of *imperative programs*.

Hoare Approach³

Hoare triples: $\{P\} C \{Q\}$

C - code; P - pre-condition; Q - post-condition;

Axioms, e.g. *Skip* and *Assign*:

$$\overline{\{P\} \text{skip} \{P\}} \text{ and } \overline{\{P[E/V]\} V := E \{P\}}$$

Where E is any expression and V is any variable.

Inference rules, e.g. *Composition* and *Conditional*:

$$\frac{\{P\} S_1 \{R\}, \{R\} S_2 \{Q\}}{\{P\} S_1; S_2 \{Q\}} \text{ and } \frac{\{B \wedge P\} S \{Q\}, \{\neg B \wedge P\} T \{Q\}}{\{P\} \text{ if } B \text{ then } S \text{ else } T \text{ elseif } \{Q\}}$$

Partial correctness: If P holds before executing C then Q holds afterwards, **ONLY** if C terminates.

³“An Axiomatic Basis for Computer Programming”, Tony Hoare, 1969.

Dijkstra Extension⁴

The *weakest pre-condition calculus* for $\{P\} C \{Q\}$

- Explains how C transforms P into Q ;
- Gives a predicate transform semantics for proofs;

Backward reasoning: ($W.P.$)

- Based on Q and C calculate the *weakest pre-condition* \hat{P}
- If $P \implies \hat{P}$, then $\{P\} C \{Q\}$ holds

Forward reasoning: ($S.P.$)

- Based on P and C calculate the *strongest post-condition* \hat{Q}
- If $\hat{Q} \implies Q$, then $\{P\} C \{Q\}$ holds

⁴“Guarded commands, non-determinacy and formal derivation of programs”, Edsger Dijkstra, 1975

Software
quality and
formal
methods:
Hoare/Dijkstra
approach

Dr. Ivan S.
Zapreev

Software
Quality

Programming
Languages

Formal
Verification

**Frama-C
Framework**

Verification
Practice

Concluding
Remarks

Frama-C Framework

Frama-C Framework: Outline

Software
quality and
formal
methods:
Hoare/Dijkstra
approach

Dr. Ivan S.
Zapreev

Software
Quality

Programming
Languages

Formal
Verification

Frama-C
Framework

Verification
Practice

Concluding
Remarks

- Framework Description
- Plugins Overview
- What is ACSL?

Framework Description

Frama-C is a:

- Plug-in-based
- Open-source
- Cross-platform

framework for ANSI-C source-code analysis:

- Browse unfamiliar code
- Static code analysis
- Dynamic code analysis
- Code transformations
- Certification of critical software

You can easily build upon the existing plug-ins to implement your own analysis.

Plugins Overview: Main

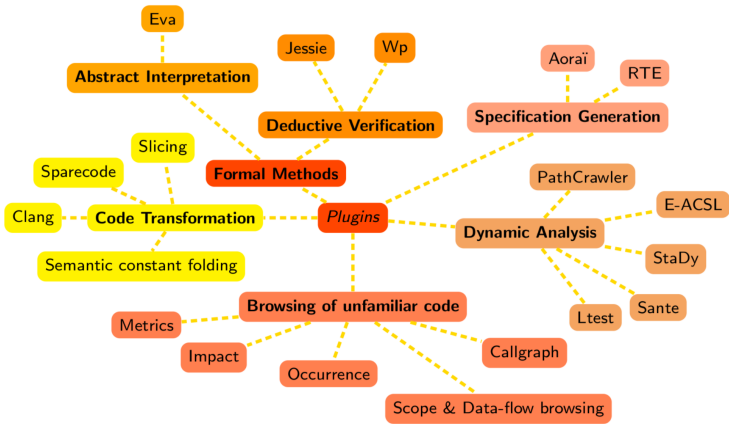


Figure 10: Frama-C plugins

Plugins Overview: WP

WP – *weakest precondition* for ACSL specs of ANSI-C programs.

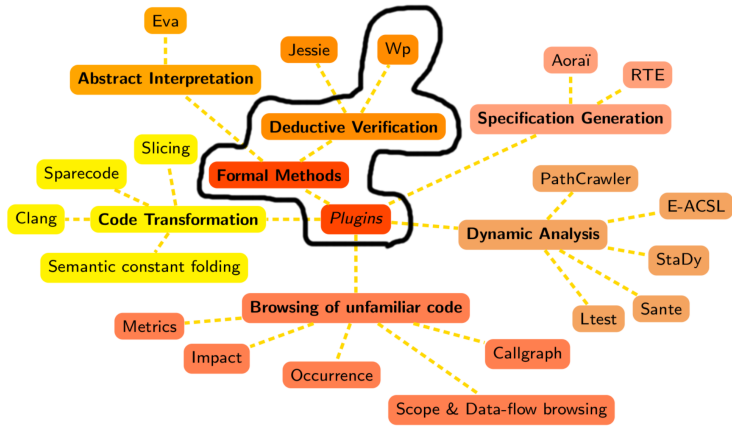


Figure 11: Frama-C WP plugin

What is ACSL: General

In short:

- ACSL – ANSI/ISO C Specification Language
- Allows to formally specify properties of C programs

An example function property specification:

```
/*@ ensures \result >= x && \result >= y;  
    ensures \result == x || \result == y;  
*/  
  
int max (int x, int y) {  
    return (x > y) ? x : y;  
}
```

A general function contract consists of:

- post-conditions - ensures
- pre-conditions - requires

What is ACSL: Pointers

ACSL allows to reason about, e.g.:

- Pointers
- Arrays
- Termination

Consider pointers:

```
/*@ requires \valid(x) && \valid(y);  
    ensures *x <= *y;  
    */  
void max_ptr (int *x, int *y) {  
    if(*x > *y) {  
        int tmp = *x;  
        *x = *y;  
        *y = tmp;  
    }  
}
```

What is ACSL: Completeness

Is the following max_ptr implementation correct?

```
/*@ requires \valid(x) && \valid(y);  
        ensures *x <= *y;  
        */  
void max_ptr (int *x, int *y) {  
    *x = *y = 0;  
}
```

We need to make our specification **complete**:

```
/*@ requires \valid(x) && \valid(y);  
        ensures *x <= *y;  
        ensures (*x == \old(*x) && *y == \old(*y)) ||  
                (*x == \old(*y) && *y == \old(*x));  
        */  
void max_ptr(int *x, int *y);
```


What is ACSL: The whole spec.

The complete ACSL specification v1.4 has 93 pages:

https://frama-c.com/download/acsl_1.4.pdf



Figure 12: Feel free to explore

Software
quality and
formal
methods:
Hoare/Dijkstra
approach

Dr. Ivan S.
Zapreev

Software
Quality
Programming
Languages

Formal
Verification

Frama-C
Framework

**Verification
Practice**

Concluding
Remarks

Verification Practice

Verification Practice: Outline

Software
quality and
formal
methods:
Hoare/Dijkstra
approach

Dr. Ivan S.
Zapreev

Software
Quality

Programming
Languages

Formal
Verification

Frama-C
Framework

Verification
Practice

Concluding
Remarks

- Verification Examples
- Verification Outcomes
- Experience Summary

Verification Examples: abs(.)

Software
quality and
formal
methods:
Hoare/Dijkstra
approach

Dr. Ivan S.
Zapreev

Software
Quality
Programming
Languages

Formal
Verification

Frama-C
Framework

Verification
Practice

Concluding
Remarks

Consider a primitive integer absolute value computation:

```
/*@  
    ensures \result >= 0;  
*/  
int abs(int val) {  
    if(val < 0) return -val;  
    return val;  
}
```

The verification shall return **OK**, right?

Verification Examples: Issue #1

NOP - the verification results are inconclusive:

The screenshot displays the Frama-C WP (Wp) interface. The top pane shows a C code snippet for a function `abs` with an `ensures` property. The code is as follows:

```
/*@ ensures
  (\old(val) == 0 => \result == 0) ^
  (\old(val) > 0 => \result == \old(val)) ^
  (\old(val) < 0 => \result == -\old(val));
*/
int abs(int val)
{
  int __retres;
  if (val < 0) {
    /*@ assert rte: signed_overflow: -2147483647 <= val; */
    __retres = - val;
    goto return_label;
  }
  __retres = val;
  return_label: return __retres;
}
```

The bottom pane shows the verification results. The table below represents the data shown in the interface:

Module	Goal	Model	Qed	Script	Alt-Ergo 2.3.0
abs	Post-condition	Typed		—	
abs	Assertion 'rte,signed_overflow'	Typed	—	—	

Figure 13: WP detects a possible overflow

Verification Examples: Issue #2

Extend the specification with a pre-condition:

```
/*@ requires INT_MIN < val;  
    ensures \result >= 0;  
    */  
int abs(int val) {  
    if(val < 0) return -val;  
    return val;  
}
```

The verification is **OK**, but the spec is NOT complete:

```
/*@ requires INT_MIN < val;  
    ensures \result >= 0;  
    */  
int abs(int val) {  
    return 1;  
}
```

Verification Examples: Final?

An explicit `\result` value specification makes it complete:

```
/*@ requires INT_MIN < val;  
    ensures (val == 0 ==> \result == 0) &&  
           (val > 0 ==> \result == val) &&  
           (val < 0 ==> \result == -val);  
*/  
int abs(int val) {  
    if(val < 0) return -val;  
    return val;  
}
```

What if the implementation was wrong?

Would we be able to identify the root-cause?

Verification Examples: Faulty

Consider a lengthy and potentially buggy implementation:

```
/*@ requires INT_MIN < val;  
    ensures (val == 0 ==> \result == 0) &&  
           (val > 0 ==> \result == val) &&  
           (val < 0 ==> \result == -val);  
*/  
int abs(int val) {  
    if(val == 0) {  
        return 0;  
    } else {  
        if(val < 0) {  
            return val;  
        } else {  
            return -val;  
        }  
    }  
}
```


Verification Examples: Issue #3

The verification is inconclusive, the prover has failed!

The screenshot shows the Frama-C IDE interface. The top pane displays a C program with annotations. The bottom pane shows the 'Messages (2)' tab with a table of verification results.

```

/*@ requires -2147483647 - 1 < val;
   ensures
      (\old(val) == 0 => \result == 0) ^
      (\old(val) > 0 => \result == \old(val)) ^
      (\old(val) < 0 => \result == -\old(val));
*/
int abs(int val)
{
    int __retres;
    if (val == 0) {
        __retres = 0;
        goto return_label;
    }
    else {
        if (val < 0) {

```

Module	Goal	Model	Qed	Script	Alt-Ergo 2.3.0	C
abs	Post-condition	Typed	—	—	✂	
abs	Assertion 'rte,signed_overflow'	Typed	●	—		

Figure 14: What is the actual reason?

Verification Examples: Split

What if we split the post-condition from:

```
/*@ requires INT_MIN < val;  
    ensures (val == 0 ==> \result == 0) &&  
           (val > 0 ==> \result == val) &&  
           (val < 0 ==> \result == -val);  
*/
```

into separate statements:

```
/*@ requires INT_MIN < val;  
    ensures (val == 0 ==> \result == 0);  
    ensures (val > 0 ==> \result == val);  
    ensures (val < 0 ==> \result == -val);  
*/
```

and then run verification again.

Verification Examples: Insights

This gives us insights into what could be wrong:

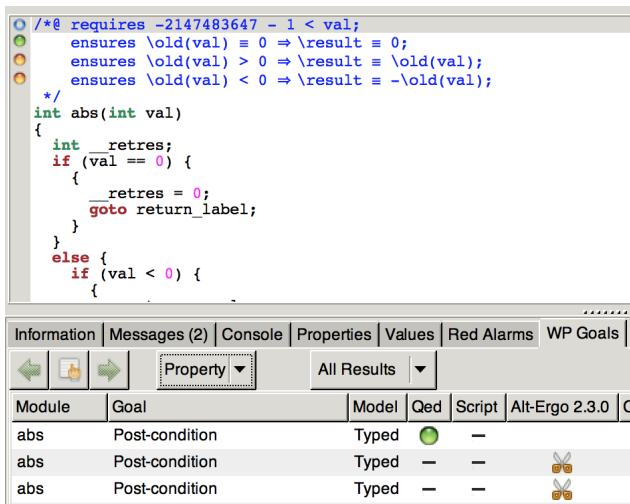


Figure 15: Finding the root-causes

Verification Examples: Bugs

Now we can now look into the code and identify bugs:

```
/*@ requires INT_MIN < val;
    ensures (val == 0 ==> \result == 0);
    ensures (val > 0 ==> \result == val);
    ensures (val < 0 ==> \result == -val);
*/
int abs(int val) {
    if(val == 0) {
        return 0;           //OK
    } else {
        if(val < 0) {
            return val;     //BUG #1, should return -val
        } else {
            return -val;    //BUG #2, should return val
        }
    }
}
```

Verification Examples: Issue #4

Fixing BUG #1 turns the corresponding post-conditions green!

```

/*@ requires -2147483647 - 1 < val;
   ensures \old(val) == 0 => \result == 0;
   ensures \old(val) > 0 => \result == \old(val);
   ensures \old(val) < 0 => \result == -\old(val);
*/
int abs(int val)
{
    int __retres;
    if (val == 0) {
        __retres = 0;
        goto return_label;
    }
    else {
        if (val < 0) {
            /*@ assert __retres signed overflow: 2147483647

```

```

main.c
31 /*@
32 requires INT_MIN < val;
33 ensures (val == 0 ==> \result == 0);
34 ensures (val > 0 ==> \result == val);
35 ensures (val < 0 ==> \result == -val);
36 */
37 int abs(int val) {
38     if(val == 0) {
39         return 0; //OK
40     } else {
41         if(val < 0) {
42             return -val; //OK
43         } else {
44             return -val; //BUG #2, should return val
45         }
46     }
47 }
48

```

Information Messages (2) Console Properties Values Red Alarms WP Goals

Property All Results

Module	Goal	Model	Qed	Script	Alt-Ergo 2.3.0	CVC4 1.7	CVC4 1.7 (counterexamples)	Cc
abs	Post-condition	Typed	●	—				
abs	Post-condition	Typed	—	—				
abs	Post-condition	Typed	●	—				
abs	Assertion 'rte,signed_overflow'	Typed	●	—				

Figure 16: Sequential issue resolution

Verification Examples: Issue #5

Fixing BUG #2 yields an **OK** verification result!

```
/*@ requires -2147483647 - 1 < val;
ensures \old(val) == 0 => \result == 0;
ensures \old(val) > 0 => \result == \old(val);
ensures \old(val) < 0 => \result == -\old(val);
*/
int abs(int val)
{
    int __retres;
    if (val == 0) {
        {
            __retres = 0;
            goto return_label;
        }
    }
    else {
        if (val < 0) {
            {

```

```
main.c
31 /*@
32 requires INT_MIN < val;
33 ensures (val == 0 ==> \result == 0);
34 ensures (val > 0 ==> \result == val);
35 ensures (val < 0 ==> \result == -val);
36 */
37 int abs(int val) {
38     if (val == 0) {
39         return 0; //OK
40     } else {
41         if (val < 0) {
42             return -val; //OK
43         } else {
44             return val; //OK
45         }
46     }
47 }
```

Information | Messages (2) | Console | Properties | Values | Red Alarms | WP Goals

Property

All Results

Module	Goal	Model	Qed	Script	Alt-Ergo 2.3.0	CVC4 1.7	CVC4 1.7 (counterexamples)	Coc
abs	Post-condition	Typed	●	—				
abs	Post-condition	Typed	●	—				
abs	Post-condition	Typed	●	—				
abs	Assertion 'rte,signed_overflow'	Typed	●	—				

Figure 17: Now we are all fine

Verification Outcomes

If the verification result is **OK**:

- The program satisfies the specification, BUT
- Is the specification correct/complete?

If the verification result is **NOK**⁵:

- An *incorrect implementation*:
 - Find counter-example via test generation;
- A *wrong specification*:
 - Complete spec. and proof analysis;
 - Change/extend the specification;
- A *prover's failure*:
 - Alternative provers;
 - Interactive proof assistants;

⁵This includes a failed verification attempt, e.g. a time out

Experience Summary

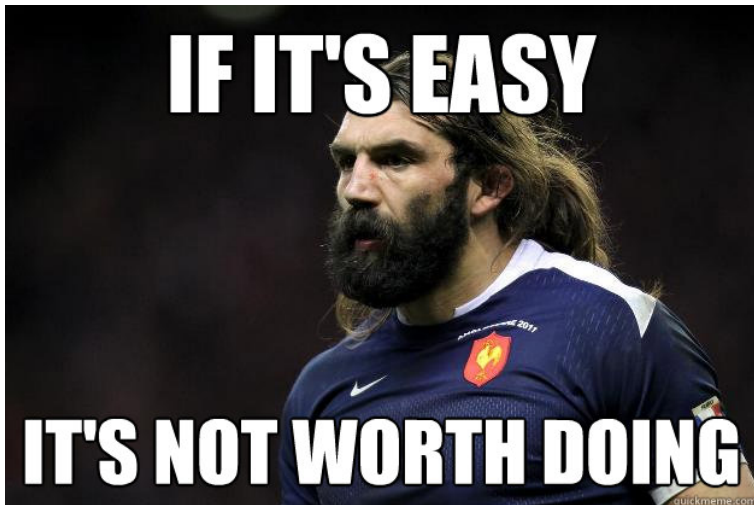


Figure 18: It is not so easy but it helps!

Software
quality and
formal
methods:
Hoare/Dijkstra
approach

Dr. Ivan S.
Zapreev

Software
Quality

Programming
Languages

Formal
Verification

Frama-C
Framework

Verification
Practice

Concluding
Remarks

Concluding Remarks

Concluding Remarks

We have looked into:

- Software quality and software engineering
- Programming language classification
- Formalization of software verification
- Hoare/Dijkstra approach to formal proving
- Frama-C a platform for ANSI-C code analysis
- Experienced practical program verification

We can conclude that:

- Formal software verification is useful
- It is not yet fully automated
- There is a lot more to learn about it!

Thank you and are there any questions?

I appreciate your time!

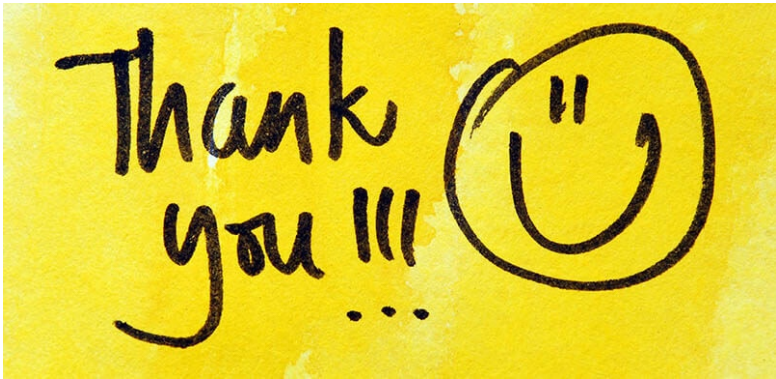


Figure 19: It was great to give you a talk!

More useful links:

- **ACSL Mini-Tutorial:**
<https://frama-c.com/download/acsl-tutorial.pdf>
- **ACSL-tutorial:**
<https://frama-c.com/download/acsl-tutorial.pdf>
- **ACSL-by-Example:**
<https://www.cs.umd.edu/class/spring2016/cmsc838G/frama-c/ACSL-by-Example-12.1.0.pdf>
- **Frama-C website:** <https://frama-c.com/>
- **Frama-C v20.0 manual:** <https://frama-c.com/download/user-manual-20.0-Calcium.pdf>
- **Frama-C WP tutorial:** <https://allan-blanchard.fr/publis/frama-c-wp-tutorial-en.pdf>