Software quality and formal methods: Hoare/Dijkstra approach

Dr. Ivan S. Zapreev

Neat Software Designs

2020-01-17

Software Quality

Programming Languages

Scientific Approache

Frama - C

Verification Examples

Concluding remarks

Outline:

- Software Quality
  - Motivating Examples
  - Software Development
  - Formal Verification
- Programming Languages
  - Language generations
  - Imperative programming
  - ANSI-C
- Scientific Approach
  - Various methods
  - Software verification
  - Hoare/Dijkstra approach
- Frama - C
  - Platform description
  - Plugins overview
  - Introduction to ACSL
- Verification Examples
- Concluding remarks

Software Quality

Motivating Examples: Major

- 1985–1987 – *Therac-25:*
  - Radiation therapy overdose
  - Control software flaw:
    - Race conditions
  - Death of 6 (six) cancer patients
- 1996 – *Ariane-5 missile:*
  - Missile crash
  - Control software flaw:
    - 64-bit float to 16-bit int
  - $7 billion development program
  - $500 million cargo
- 2005 – *Toyota Camry:*
  - Sudden unintended acceleration:
  - Control software flaw:
    - Recursion causing stack overflow
  - 89 deaths and 57 injuries
  - $1.2 billion compensations

Motivating Examples: More

The 12 Software Bugs That Caused Epic Failures: <link>

Software Development: V-model

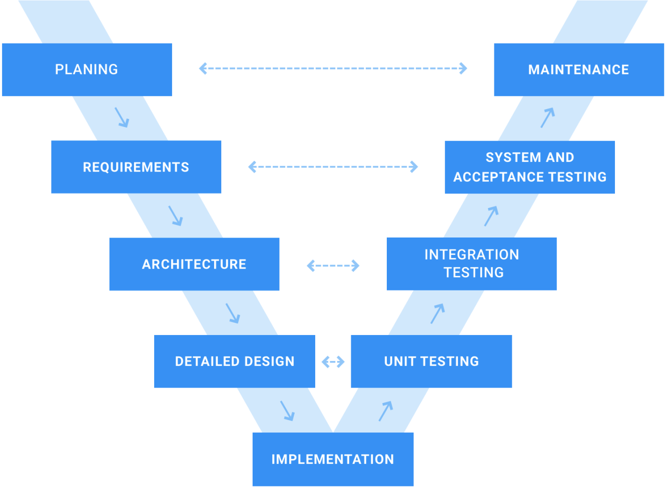


Figure 1: Software development process

Software Development: V & V

Is formally defined in, e.g.: `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 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?

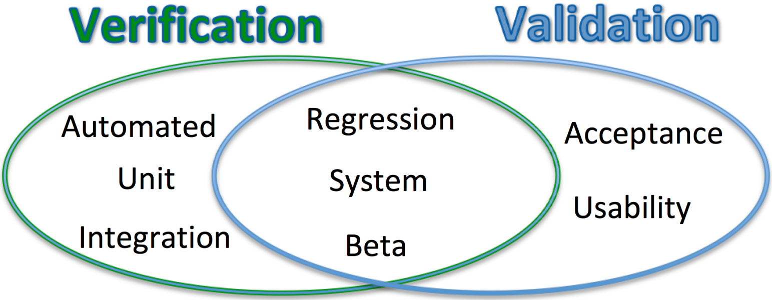


Figure 2: Devision of testing types

Formal Verification

Facts:

- No glabally recognized definition of Formal Methods[1].
- Local attempts to have one[2], e.g.:

  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.

Conclusion:

Formal methods are techniques suitable for Verification.

[1] "Formal Methods for Industrial Critical Systems", S. Gnesi, T. Margaria

[2] "Formal Methods", Michael Collins, CMU

Formal Software Verification

A program shall satisfy a formal specification of its behavior.

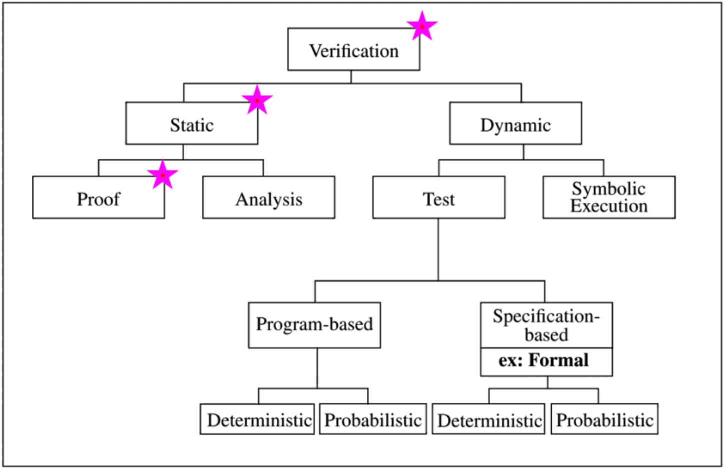
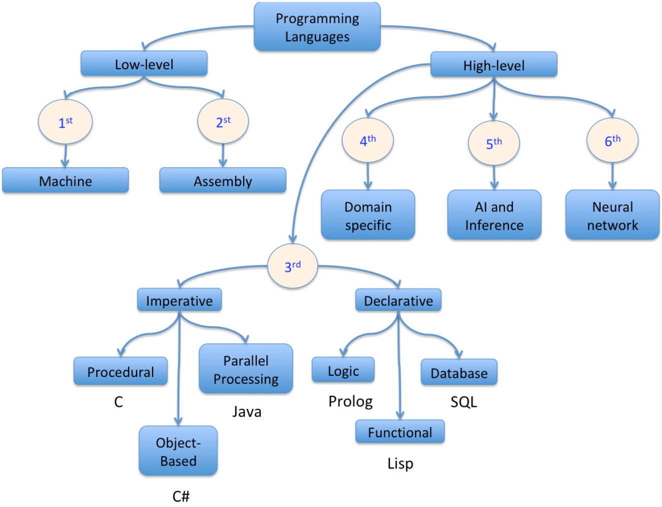


Figure 3: Verification methods

Programming Languages

Language generations



Programming Languages

Low-level

High-level

1st

2st

Machine

Assembly

4th

5th

6th

Domain specific

AI and Inference

Neural network

3rd

Imperative

Declarative

Procedural

C

Parallel Processing

Java

Object-Based

C#

Logic

Prolog

Database

SQL

Functional

Lisp

Figure 4: Generations of Programming languages

Imperative programming: Main

- *Imperative* – Describes computation in terms of statements that change a program state. Imperative JavaScript example:

```
function sum(a, b) {
  return a + b;
}
console.log( sum(5, 3) );
```

- *Declarative* – Expresses what to accomplish without specifying concrete steps. Declarative JavaScript example:

```
const sum = a => b => a + b;
console.log( sum (5) (3) );
```

Procedural language – is an imperative language in which the program is built from one or more subroutines.

Imperative programming: Test

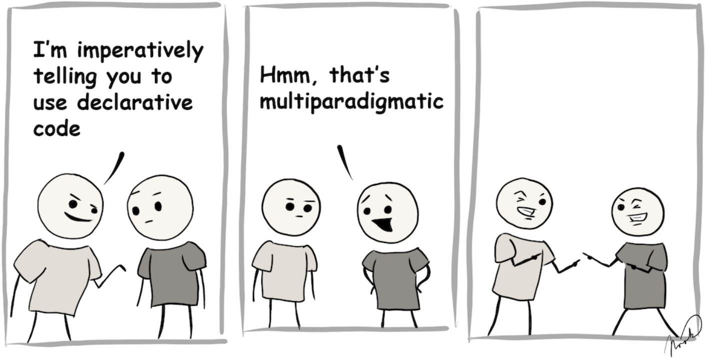

I'm imperatively telling you to use declarative code
Hmm, that's multiparadigmatic

Figure 5: If you laugh, it means you've passed

ANSI-C

An *imperatice procedral* language.

Scientific Approache

Overview

Formal Methods

Hoare/Dijkstra approach

Frama - C

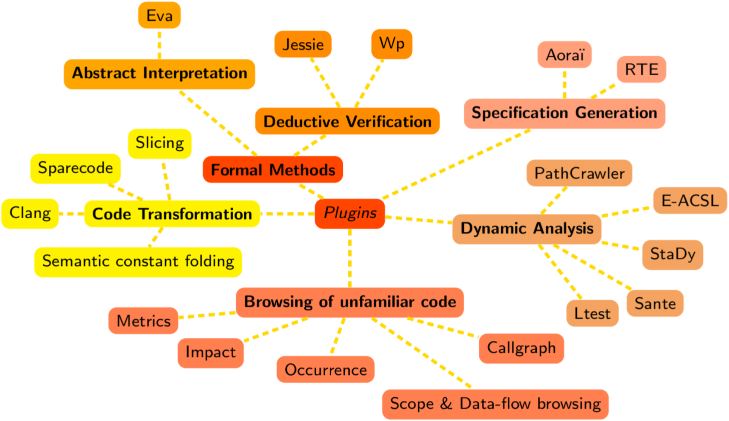
General info

A plugin-based open-source cross-platform framework for `C` source-code analysis:

- Browsing 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



Plugins

Formal Methods

Abstract Interpretation

Eva

Deductive Verification

Jessie

Wp

Specification Generation

Aoraï

RTE

Dynamic Analysis

PathCrawler

E-ACSL

StaDy

Sante

Ltest

Code Transformation

Slicing

Sparecode

Clang

Semantic constant folding

Browsing of unfamiliar code

Metrics

Impact

Occurrence

Scope & Data-flow browsing

Callgraph

Figure 6: Frama-C plugins

Weakest Precondition plugin

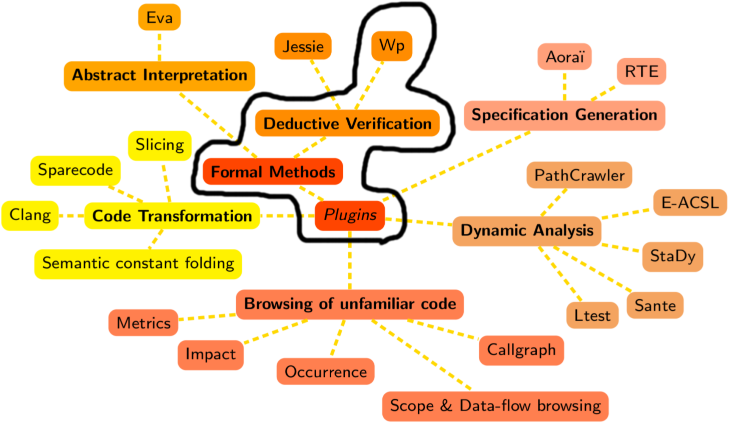


Figure 7: Frama-C WP plugin

Verification Process

ACSL: General

Verification Examples

Concluding remarks