Program logics: reasoning principles for high-assurance software

Introduction

Xavier Leroy
2021-03-04

Collège de France, chair of software sciences
xavier.leroy@college-de-france.fr
How to make sure that software behaves correctly?
Conventional methods

Test

• Run the program on well-chosen inputs.
• Compare observed behaviors with expected behaviors.

Review

• Carefully proofread the code, the tests, the design documents, …

Analysis

• Mathematical study of some aspects of the program: numerical precision, time or space complexity, etc.
• Pencil and paper, or with machine assistance (static analysis tools).
Limitations of testing

*Testing shows the presence, not the absence of bugs.*

(E. W. Dijkstra, 1969)

We test a small number of all possible behaviors of the program. Some bugs trigger very rarely!

**Example (carry propagation in a cryptographic library)**

Add $2 \times ta \times tb$ to $c2:c1:c0$ while “optimizing” carry propagation.

```c
BN_UMULT_LOHI(t0,t1,ta,tb);
t2 = t1+t1; c2 += (t2<t1)?1:0;
t1 = t0+t0; t2 += (t1<t0)?1:0;
c0 += t1; t2 += (c0<t1)?1:0;
c1 += t2; c2 += (c1<t2)?1:0;
```
Limitations of code review

Given enough eyeballs, all bugs are shallow.
(Eric Raymond, 1999)

Reviewers are tired or distracted.

Some codes such as hot fixes are not reviewed much.

Example (the goto fail bug)

```c
if ((err=SSLHashSHA1.update(&hashCtx,&signedParams)) != 0)
    goto fail;
    goto fail;
if ...
...  
fail: return err;
```
Limitations of code analysis

Beware of bugs in the above code;
I have only proved it correct, not tried it.
(Donald E. Knuth, 1977)

Risk of errors in pencil-and-paper analyses and of unsoundness in static analysis tools.

Possible gap between the analysis and the actual program or its actual execution context.

Example (Ariane 501)

Overflow in a conversion 64-bit FP number → 16-bit integer.

An analysis conducted in the context of Ariane 4 proved that the converted quantity, called BH, always fits in 16 bits. The analysis was invalid in the context of Ariane 5.
Deductive verification (also called program proof)

Logical reasoning that establishes properties that hold for all possible executions of the program.

Unlike other “formal methods”, the properties established go all the way up to full functional correctness w.r.t. a specification.

Practical interest:

• Obtaining guarantees stronger than those we can get using testing and review.
• Finding bugs we cannot find by other means.
A program logic provides us with a **specification language** and **reasoning principles** to reason about program behaviors.

Specifications generally consist in **logical assertions** about the program:

- **preconditions**: hypotheses on inputs  
  (function parameters; initial values of variables)
- **postconditions**: guarantees on outputs  
  (function results; final values of variables)
- **invariants**: guarantees on the states at a program point  
  (loop invariants, data structure invariants, …)
Program logics and deductive verification

Program

Assertions

Verification conditions

Proofs: pencil-and-paper, automated, or interactive

OK / alarm
Hunting for bugs:
the example of binary search
Binary search

```java
l = 0; h = a.length - 1;
while (l <= h) {
    m = (l + h) / 2;
    if (a[m] == v) return m;
    if (a[m] < v) h = m - 1; else l = m + 1;
}
return -1;
```
A long history

```java
l = 0; h = a.length - 1;
while (l <= h) {
    m = (l + h) / 2;
    if (a[m] == v) return m;
    if (a[m] < v) h = m - 1; else l = m + 1;
}
return -1;
```

1946  John Mauchly, *Moore School Lectures*
1960  Derrick H. Lehmer publishes the modern algorithm
1986  Jon Bentley, *Programming pearls*, chapter 4
2004  Bug report: `java.util.Arrays.binarySearch()` *will throw an ArrayIndexOutOfBoundsException if the array is large.*
2006  Joshua Bloch, *Nearly All Binary Searches and Mergesorts are Broken.*
The source of the bug: an arithmetic overflow

\[
m = (l + h) / 2;
\]

We have \(0 \leq l \leq h < a.length\).

\(l + h\) can overflow if \(a.length\) is large enough.

In Java, \(l + h\) becomes negative, as well as \(m\), hence \(a[m]\) raises an “out of bounds” exception.

In C, we have a so-called undefined behavior. Often, the program continues with the wrong value of \(m\). Worse things can happen.

A simple fix:

\[
m = l + (h - l) / 2;
\]
A bug that is hard to find

Test:

• We rarely test on very big inputs.
• A 64-bit machine and several Gb of RAM are required to trigger this bug.

Review:

• The formula \( (l + h)/2 \) is so familiar as to raise no suspicion.
• Reviewers are likely to suggest “optimizing” \( l + (h - l)/2 \) as \( (l + h)/2 \).

Analyses:

• A variation interval analysis can detect the problem.
Demo

Deductive verification of binary search using the Frama-C WP tool.
The course and the seminar
Objectives for the course

Understand the principles of program logics and the recent developments in this area.

*Leitmotiv:* which logics for which features of programming languages?

(variables, pointers, concurrency, higher-order, etc)
Demonstrate implementations of program logics in industrial-strength verification tools.

Discuss new verification problems and new ideas to tackle them.
1. The birth of program logics
2. Variables and loops: Hoare logic
3. Pointers and data structures: separation logic
4. Shared-memory concurrency: concurrent separation logic
5. Extensions of separation logic: fractional permissions, ghost state, stored locks, …
6. Logics for weakly-consistent shared memory
7. Logics for functional, higher-order languages
Les logiques de programmes à l’épreuve du réel: tours et détours avec Frama-C/WP

Preuve auto-active de programmes en SPARK

VeriFast: Semi-automated modular verification of concurrent C and Java programs using separation logic

Raisonner à propos du temps en logique de séparation

Protocoles personnalisés en logique de séparation: ressources fantômes et invariants dans la logique Iris

Gillian: Compositional Symbolic Testing and Verification