# Formal Methods in Software Engineering

Andrei Arusoaie

2025-2026

# Table of Contents

# Organisation

- Course webpage: `https://edu.info.uaic.ro/metode-formale-inginerie-software/`
- Instructor: Andrei Arusoaie
- Grading:
  - lab works (40%)
  - case study - written report (30%)
  - project (30%)
- **Important**: no final exam! no reexamination!
- Minimum: 50%

# An old issue with software

Program correctness

- Validation: Are we building the right product?
- Correctness: Is the product built correctly?
  - Ensures that the software is free from defects and behaves as specified
  - Confirms that the software operates without errors and adheres to its design specifications
  - Methods: formal methods, unit (or other ways of) testing, code reviews, ...

# Formal Methods

- *Formal methods is that area of computer science that is concerned with the application of mathematical techniques to the design and implementation of computer hardware and (more usually) software.*

  Michael Wooldridge (`https://www.cs.ox.ac.uk/people/michael.wooldridge/teaching/soft-eng/lect06.pdf`)

# What kind of issues formal methods can help with?

- Software bugs
- Security vulnerabilities
- Design errors
- Specification

# Example 1: Ariane 5 Flight 501 Failure

- Successor to Ariane 4; cornerstone for ESA–NASA space station collaboration.
- Study interaction between solar wind and Earth's magnetosphere in unprecedented detail.
- Strategic Aim: Keep Europe competitive in the space industry.
- Ariane 5 exploded at the first launch failed.
- Immediate estimated loss — \$370 million, but the real cost was much higher ($> \$400m$).

# Excerpt from the investigation report

- The internal SRI software exception was caused during execution of a data conversion from 64-bit floating point to 16-bit signed integer value. The floating point number which was converted had a value greater than what could be represented by a 16-bit signed integer. This resulted in an Operand Error. The data conversion instructions (in Ada code) were not protected from causing an Operand Error, although other conversions of comparable variables in the same place in the code were protected.

Full report:
`https://esamultimedia.esa.int/docs/esa-x-1819eng.pdf`

# Ariane 5 Flight 501 Failure — Key Points

- Value on 16-bit bus exceeded representable range $\rightarrow$ `operand_error` flag set.
- Flag alone did not cause failure.
- Detection routine interpreted flag as hardware error $\rightarrow$ sent diagnostic message with flag's software location.
- Other systems misread diagnostic as command to nozzle actuators $\rightarrow$ rocket veered off course and exploded.
- Software bug? Design error? Missing integration testing?

# Example 2: Java Binary Search Function in Arrays Class

```
public static int binarySearch(int[] a, int key) {
    int low = 0;
    int high = a.length - 1;

    while (low <= high) {
        int mid = (low + high) / 2;
        int midVal = a[mid];

        if (midVal < key)
            low = mid + 1;
        else if (midVal > key)
            high = mid - 1;
        else
            return mid; // key found
    }
    return -(low + 1);  // key not found.
}
```

# Example 2: integer overflow in the mid calculation

```
public static int binarySearch(int[] a, int key) {
    int low = 0;
    int high = a.length - 1;

    while (low <= high) {
        int mid = (low + high) / 2; // possible overflow
        int midVal = a[mid];

        if (midVal < key)
            low = mid + 1;
        else if (midVal > key)
            high = mid - 1;
        else
            return mid; // key found
    }
    return -(low + 1);  // key not found.
}
```

# Example 2: . . . 9 years later, the fix came

```
public static int binarySearch(int[] a, int key) {
  int low = 0;
  int high = a.length - 1;

  while (low <= high) {
      int mid = low + (high - low) / 2; // fix
      int midVal = a[mid];

      if (midVal < key)
          low = mid + 1;
      else if (midVal > key)
          high = mid - 1;
      else
          return mid; // key found
  }
  return -(low + 1);  // key not found.
}
```

# Example 3: CVE-2014-0160: Heartbleed

- Buffer overflow in OpenSSL (2012-2014)
- 66% of web servers worldwide vulnerable
- Data Exposed: Passwords, private keys, personal data
- Retrieve 64KB chunks of server memory repeatedly

**4 lines** of missing bounds checking
broke the internet's security

*"The worst vulnerability found since commercial traffic began to flow on the internet"*
– Bruce Schneier, Security Expert

# Example 3: The Fatal Code

```
int dtls1_process_heartbeat(SSL *s) {
    unsigned char *p = &s->s3->rrec.data[0], *pl;
    unsigned short hbtype;
    unsigned int payload;     // User-controlled length!
    unsigned int padding = 16;

    hbtype = *p++;
    n2s(p, payload);          // No bounds check on payload!
    pl = p;

    // DISASTER: memcpy using untrusted length
    memcpy(bp, pl, payload);  // payload <= actual_length?
    // ...
}
```

Source:

https://www.cs.fsu.edu/~baker/opsys/notes/heartbleed.html

# Example 3: Attack Scenario

1. `payload` comes from the attacker. No verification that enough data exists!
2. `n2s(p, payload)` reads those 2 bytes and interprets them as the payload length
3. Attacker sends: "A" – 1 byte length
4. `n2s(p, payload)` reads that the payload has more than 1 byte
5. Server reveals pars of its memory (keys, passwords, data) that were not intended to be sent

# What methods can be used to avoid such issues?

- Code reviews (effective?)
- Testing (unit, integration, system, acceptance, . . . )
- Formal methods
  - Static/dynamic analysis methods
  - Mathematical/Logical specifications + formal reasoning

# Why formal methods?

- Code is written in a programming language, to be executed by a machine
- Programming languages don't come with tools that allow us to reason about programs
- But mathematical/logical languages provide such tools
  - The main idea is to encode programs/systems/protocols and specifications in a mathematical/logical language
  - Then, use mathematical/logical reasoning to prove properties about the program

# Formal Methods - I

... consist of a plethora of instruments/tools

# Formal Methods - I

... consist of a plethora of instruments/tools

(I) Specification Languages:

- Model-based Specification Languages (for defining models and their behaviors)
  - Z (set theory + predicate logic), VDM (system functions and data types), B-Method (state and operations, proof obligations)

# Formal Methods - I

... consist of a plethora of instruments/tools

(I) Specification Languages:

- Model-based Specification Languages (for defining models and their behaviors)
  - Z (set theory + predicate logic), VDM (system functions and data types), B-Method (state and operations, proof obligations)
- Property-based Specification Languages (for specifying the properties and constraints the system must satisfy)
  - Alloy (declarative language for expressing complex structural constraints and behavior in terms of relations)

# B-Method example: Abstract Machine of a Ticket System

```
MACHINE Ticket
VARIABLES serve, next
INVARIANT
        serve:NATURAL & next:NATURAL & serve <= next
INITIALISATION serve, next := 0, 0
OPERATIONS
         ss <-- serv_next =
                PRE serve < next
                THEN ss, serve := serve + 1, serve + 1
                END;
        tt <-- take_ticket =
                tt, next := next, next + 1
END
```

Source:
https://mooc.imd.ufrn.br/course/the-b-method/video/5

# Formal Methods - II

(II) Formal Verification Techniques

- Model Checking (verify properties of a system by exploring all possible states and transitions)
  - Tools: SPIN, NuSMV

# Formal Methods - II

(II) Formal Verification Techniques

- Model Checking (verify properties of a system by exploring all possible states and transitions)
  - Tools: SPIN, NuSMV
- Automated Reasoning (automatically generate proof obligations from programs and prove them using SAT/SMT solvers)
  - Tool for verification: Why3, Spass, Vampire
  - SMT/SAT solvers: Z3, CVC4, Alt-Ergo

# Formal Methods - II

(II) Formal Verification Techniques

- Model Checking (verify properties of a system by exploring all possible states and transitions)
  - Tools: SPIN, NuSMV
- Automated Reasoning (automatically generate proof obligations from programs and prove them using SAT/SMT solvers)
  - Tool for verification: Why3, Spass, Vampire
  - SMT/SAT solvers: Z3, CVC4, Alt-Ergo
- Interactive Program Verification (interactively formulate and prove properties about programs)
  - Tools: Coq, Isabelle, Agda, Idris, ...

# SPIN

```
init {
        // file: ex_1a.pml
        byte i          // initialized to 0 by default
        do              // loop forever
        :: i++          // short for i = i + 1
        od
}
```

Source: https://spinroot.com/spin/Man/1_Exercises.html

# Z3

```
(declare-const p Bool)
(declare-const q Bool)
(declare-const r Bool)
(define-fun conjecture () Bool
        (=> (and (=> p q) (=> q r))
                 (=> p r)))
(assert (not conjecture))
(check-sat)
```

Source: https://microsoft.github.io/z3guide/docs/logic/propositional-logic

# Why3

```
module MaxAndSum

  use int.Int
  use ref.Ref
  use array.Array

  let max_sum (a: array int) (n: int) : (int, int)
    requires { n = length a }
    requires { forall i. 0 <= i < n -> a[i] >= 0 }
    returns  { sum, max -> sum <= n * max }
  = let sum = ref 0 in
    let max = ref 0 in
    for i = 0 to n - 1 do
      invariant { !sum <= i * !max }
      if !max < a[i] then max := a[i];
      sum := !sum + a[i]
    done;
    !sum, !max

end
```

Source: https://www.why3.org/doc/whyml.html

# Coq

```
Require Import List.
Import ListNotations.

Theorem length_append :
  forall (l1 l2 : list nat),
    length (l1 ++ l2) = length l1 + length l2.
Proof.
  induction l1.
  - simpl. reflexivity.
  - intros l2.
    simpl.
    rewrite IHl1.
    reflexivity.
Qed.
```

# Formal Methods - III

(III) Program Analysis

- Static Analysis
  - Analyse the source code of a program without executing it
  - type checking, abstract interpretation, data flow analysis
  - Tools: SonarQube, Clang, ...

# Formal Methods - III

(III) Program Analysis

- Static Analysis
  - Analyse the source code of a program without executing it
  - type checking, abstract interpretation, data flow analysis
  - Tools: SonarQube, Clang, ...
- Dynamic Analysis
  - Analyse the source code of a program during its execution
  - profiling, testing, runtime verification, symbolic execution
  - Valgrind, KLEE, ...

# Clang static analysis

```
int function(int b) {
  int a, c;
  switch (b) {
  case 1: a = b / 0; break;
  case 4: c = b - 4;
    a = b/c; break;
  }
  return a;
}

int main() { return 0; }
```

Source: https://llvm.org/devmtg/2020-09/slides/Using_the_clang_static_ananalyzer_to_find_bugs.pdf

# Formal Methods - IV

(IV) Other methods (incomplete list)

- Refinement: from specification to implementation
- Correct-by-construction: build systems using formal methods at each stage of development
- Design-by-contract: formally specify the rights and obligations of each methods
- Verification-aided development: tools that allow you to write code and verify it during development (Dafny)
- Model-Driven development: combine modeling languages with formal methods to automatically generate code

# Where do we use formal methods today?

- Aerospace (https://link.springer.com/chapter/10.1007/978-3-642-34281-3_2)
- Defense (https://www.darpa.mil/news-events/2023-03-27)
- Nuclear powerplants (https://www-pub.iaea.org/MTCD/Publications/PDF/PUB1851_web.pdf)
- Automotive (Scania: https://link.springer.com/chapter/10.1007/978-3-030-03427-6_14)
- Operating Systems (seL4: https://docs.sel4.systems/projects/l4v/)

# IDE features

IDEs have incorporated several features from formal methods research to enhance software development. Examples:

- static analysis (type mismatches, null pointer dereference, unreachable code, useless conditions, etc.)
- open for plugin integration of model checkers and other tools
- automated refactoring
- coverage testing
- linters

# Drawbacks of Formal Methods

- Don't set very high expectations: formal methods are difficult!
- The understanding of background concepts and how to use them requires plenty of effort.
- Formal methods are expensive; they are mostly used where they are unavoidable (critical systems in general).

# ... but not using them causes trouble

- Ariane 5 Rocket, conversion from 64-bit to 16-bit: https://www.brookings.edu/articles/formal-methods-as-a-path-toward-better-cybersecurity/
- Therac-25 Radiation Therapy, lethal doses of radation due to race conditions: https://www.tricentis.com/blog/real-life-examples-of-software-development-failures
- Mars Climate Orbiter, metric vs imperial unit conversion: https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=920593
- Knight Capital Group Trading, lost $440 million in 45 minutes due to a software error in their trading algorithm: https://link.springer.com/article/10.1007/s10270-023-01124-2

# Why formal methods work?

- They are based on mathematical and logical reasoning
- They are precise; ambiguities are eliminated
- Code can be formally specified
- Formal methods forces a deep understanding of software

# Misconceptions

- Formal methods eliminates existing techniques (e.g., testing). No, they must complement existing techniques.
- You need a PhD to use formal methods :-) No, you don't.
- Formal methods completely eliminate code vulnerabilities. No, it all depends on the correctness of the specification.

# Conclusions

- We are going to see very little compared to what exists in this field
- Focus is on both theoretical and practical sides
- Don't get scared, it is normal if looks difficult
- Our goal is to understand, not to go through many things which we do not understand