Introduction to formal verification

Jieung Kim

Aug. 2022
Contents
Contents

• Intro - Do we need formal verification?
• Formal verification intro with examples
• Conclusion
Intro –
Do we need formal verification?
Software in the world
Software failure
Software failure

Ariane 5 explosion
$370 million

1996

... 2018 2021~2022

50% of American personal record

Recalls More than 150,000 vehicles

$370 million

Recall

EQUIFAX
DATA BREACH by the numbers

U.S. population: 325.7 million

DATA ELEMENT STOLEN IMPACTED U.S. CONSUMERS
Name 147 million
Date of birth 147 million
Social Security Number 146 million

158,000 TESLA RECALL
Software failure

The Cost of Poor Software Quality in the US: A 2020 Report

The Consortium for Information & Software Quality™ (CISQ™) released new research: The Cost of Poor Software Quality in the US: A 2020 Report

- Unsuccessful IT/Software projects - $260 billion (up from $177.5 billion in 2018)
- Poor quality in legacy systems - $520 billion (down from $635 billion in 2018)
- Operational software failures - $1.56 trillion (up from $1.275 trillion in 2018)
Software failure

The cost of poor software quality in the US (2020) $2.08 trillion
Software failure

The cost of poor software quality in the US (2020):

- $2.08 trillion
- $1.56 trillion

- Poor quality in legacy systems: $0.52 trillion
- Operational SW failures: $1.56 trillion

$1.56 trillion
Software failure

The cost of poor software quality

GDP

2002:
- $0.06 trillion
- 0.5% GDP

2016:
- $10.9 trillion
- 5.9% GDP

2018:
- $18.7 trillion
- 9.5% GDP

2020:
- $20.9 trillion
- 9.95% GDP
Sources of software failure

Bugs are due to

- Lack of **software formal specifications**
- Lack of **underlying models**
  - Lack of formal syntax & semantics of programming languages
  - Lack of formal definitions of program translations from high-level language programs to binary codes
  - Lack of formal definitions on the hardware the programs are running
- **Mismatches** between specifications and programs
How to reduce software failure costs

• Operational software failures
  • Show that the specification is correctly written
  • Show that the software faithfully implements its specification

• Poor quality in legacy system
  • Build a new well structured software that can replace or supplement the legacy system
  • Show that the new specification properly covers the previous specification (of the legacy system)
  • Do the same thing described above
    • Show that the specification is correctly written
    • Show that the software faithfully implements its specification
Reduce operational software failures

Our software faithfully implements the specification based on underlying HW and software specifications.
Improve poor legacy software

- Specifications of underlay
- Our software (New version)
- Specifications of our software (New version)
- Applications of our software

Our software (new version) faithfully implements the specification (new version) based on the specification of HW and underlying software.
## Tools for software assurance

Can those tools entirely tackle previous two challenges?

→ **NO!**

<table>
<thead>
<tr>
<th></th>
<th>Expressiveness level</th>
<th>Assurance level</th>
<th>Cost level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code review</td>
<td>Very high</td>
<td>Very low</td>
<td>Medium</td>
</tr>
<tr>
<td>Testing</td>
<td>Medium</td>
<td>Low</td>
<td>Medium</td>
</tr>
<tr>
<td>Type checker (Java, Haskell, Rust)</td>
<td>Low</td>
<td>High</td>
<td>low</td>
</tr>
<tr>
<td>Static &amp; dynamic analysis (Coverity, Infer)</td>
<td>Medium</td>
<td>Medium</td>
<td>low</td>
</tr>
</tbody>
</table>
## Tools for software assurance

<table>
<thead>
<tr>
<th>Tool Description</th>
<th>Expressiveness level</th>
<th>Assurance level</th>
<th>Cost level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code review</td>
<td>Very high</td>
<td>Very low</td>
<td>Medium</td>
</tr>
<tr>
<td>Testing</td>
<td>Medium</td>
<td>Low</td>
<td>Medium</td>
</tr>
<tr>
<td>Type checker (Java, Haskell, Rust)</td>
<td>Low</td>
<td>High</td>
<td>low</td>
</tr>
<tr>
<td>Static &amp; dynamic analysis (Coverity, Infer)</td>
<td>Medium</td>
<td>Medium</td>
<td>low</td>
</tr>
<tr>
<td>Formal verification (Z3, Adga, Coq)</td>
<td>Medium ~ High</td>
<td>High ~ Very high</td>
<td>Medium ~ Very high</td>
</tr>
</tbody>
</table>
## Tools for software assurance

<table>
<thead>
<tr>
<th>Practical</th>
<th>Expressiveness level</th>
<th>Assurance level</th>
<th>Cost level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code review</td>
<td>Very high</td>
<td>Very low</td>
<td>Medium</td>
</tr>
<tr>
<td>Testing</td>
<td>Medium</td>
<td>Low</td>
<td>Medium</td>
</tr>
<tr>
<td>Type checker (Java, Haskell, Rust)</td>
<td>Low</td>
<td>High</td>
<td>low</td>
</tr>
<tr>
<td>Static &amp; dynamic analysis (Coverity, Infer)</td>
<td>Medium</td>
<td>Medium</td>
<td>low</td>
</tr>
<tr>
<td>Formal verification (Z3, Adga, Coq)</td>
<td>Medium ~ High</td>
<td>High ~ Very high</td>
<td>Medium ~ Very high</td>
</tr>
</tbody>
</table>

How can we effectively use high expressiveness? How can we avoid very high cost?
An Empirical Study on the Correctness of Formally Verified Distributed Systems

Pedro Fonseca  Kaiyuan Zhang  Xi Wang  Arvind Krishnamurthy
University of Washington
{pfonseca, kaiyuanz, xi, arvind}@cs.washington.edu

Abstract
Recent advances in formal verification techniques enabled the implementation of distributed systems with machine-checked proofs. While results are encouraging, the importance of distributed systems warrants a large scale evaluation of the results and verification practices.

This paper thoroughly analyzes three state-of-the-art, formally verified implementations of distributed systems: IronFleet, Verdi, and Chapar. Through code review and testing, we found a total of 16 bugs, many of which produce serious consequences, including crashing servers, returning incorrect results to clients, and invalidating verification guarantees. These bugs were caused by violations of a wide-range of assumptions on which the verified components relied. Our results revealed that these assumptions referred to a small fraction of the trusted computing base, mostly at the interface of verified and unverified components. Based on our observations, we have built a testing toolkit called PK, which focuses on testing these parts and is able to automate the detection of 13 (out of 16) bugs.

1. Introduction
Distributed systems, complex and difficult to implement correctly, are notably prone to bugs. This is partially because developers find it challenging to reason about the combination of concurrency and failure scenarios. As a result, distributed systems bugs pose a serious problem for both service providers and end users, and have critically caused service interruptions and data losses. The struggle to improve their reliability spawned several important lines of research, such as programming abstractions, bug-finding tools, and formal verification techniques.

Formal verification, in particular, offers an appealing approach because it provides a strong correctness guarantee of the absence of bugs under certain assumptions. Over the last few decades, the dramatic advances in formal verification techniques have allowed these techniques to scale to complex systems. They were successfully applied to build large single-node implementations, such as the seL4 OS kernel and the CompCert compiler. More recently, they enabled the verification of complex implementations of distributed protocols, including IronFleet, Verdi, and Chapar, which are known to be non-trivial to implement correctly.

At a high level, verifying these distributed system implementations follows the workflow shown in Figure 1. First, developers describe the desired behavior of the system in a high-level specification, which is often manually reviewed and trusted to be correct. Developers also need to model the primitives, such as system calls provided by the OS, on which the implementation relies upon; we refer to this as the shim layer. Finally, developers invoke auxiliary tools (e.g., scripts) to communicate with a verifier and print results. The specification, the shim layer, and auxiliary tools, as well as the components they glue together, are part of the trusted computing base (TCB). If the verification check passes, it guarantees the correctness of the implementation, assuming the TCB is correct.
Tools for software assurance

Formal verification can guarantee the correctness of target software module.

Invariants provides correctness property, but it might have bugs that are not described in invariants.

Assumptions about unverified components, so it may have bugs.

Applications of our software

Specification of our software

Our software

Specification of underlay

HW & underlying software (underlay)
Tools for software assurance

What do we need to know for formal verification?
• It is built on top of lots of underlying theories
• Verification engineers can only focus on the subset that is actually required for the verification target
Formal verification intro with examples
Formal verification

Definition

The act of proving the correctness of software with respect to a certain formal specification using mathematics
Formal verification Hierarchy

Formal verification hierarchy with different types of proof checkers.
Formal verification Hierarchy

Model checkers
• Usually **supports a specification language (logic)** of limited expressiveness
  • Compared to e.g., theorem proving languages
• Verification is **fully automated**
• Focus is typically on **specification and verification of (concurrent) systems**
  • Hard to show the mismatch between specifications and programs
  • With the limited setting
  • E.g., Fixed number of threads with statically configured
Formal verification Hierarchy

**SMT solvers**
- SMT (Satisfiability Modulo Theories) is a generalization of the SAT problem
- A form of the constraint satisfaction problem
  - Which extends first-order logic w. additional theories
  - E.g., real numbers, integers, and theories of various data structures (lists, arrays, bit vectors, etc)
- Fully automated, but expressiveness is limited
  - Hard to fully guarantee correctness under all circumstances
  - E.g., OS will not crash with any applications
Formal verification Hierarchy

Theorem provers
• Supports a very expressiveness specification language (logic), such as follows
  • Classical higher order logic
  • Constructive type theory
  • Proof system
  • Mechanized support for performing proofs
• Normally requires manual effort

→ This is the thing that I worked and am working on
Key components

• Mathematical notations for
  • Program specifications
  • Invariants of the system
  • Underlying system models (e.g., HW, Compiler, etc)

• Subject of formal verification

Proof checker

Program

• Proofs for
  • Program meet specifications
  • Specifications are consistent (i.e., all Invariants are well-defined)

Refinement relation

Proof

• Consists of
  • Core proof kernel (underlying logic)
  • Extended libraries for better expressiveness
Working on formal verification

Working on formal verification is similar to how we make a software.

- Formal specification and design proofs
  - How can we provide a good abstract model for program
    - Good: simple but correct
  - How can we design proofs with lower human efforts

→ Problem solving / System design

- Proofs (especially with tools)
  - How can we actually do the proof?
    - Hand-written proof - Whiteboard coding
    - Proof with tools - Programming with IDE

→ Coding

→ Find the algorithm
→ With low complexity
→ Properly express them with programming languages
Working on formal verification

• Formal verification usually requires a large proofs
• Proofs are hard to do it with manual checks or simple checker
• We use tools
  • Proof checker + libraries to use the check easily
  • They are domain specific programming languages for formal verification
  • Model checks: Spin, UPPAAL, etc
  • SAT/SMT solvers: CVC4, Yices, Z3, etc
  • Theorem provers: ACL2, Coq, Isabelle/HOL, Lean, Adga, etc
Coq - interactive theorem prover

http://coq.inria.fr/

- Rich (pure) **functional programming language**
- Rich **logical language** with capability of writing proofs
  - User writes proofs
  - Coq makes sure every step is correct
  - (Proved to be correct) program can be extracted to Ocaml, Haskell, Scheme...
Coq - interactive theorem prover

How to learn it?

• **Software foundation**
  • Self-study material
  • Mainly maintained by University of Pennsylvania
  • A course material for junior/senior and graduate students in several universities

• We will taste it with several [Demo]s
  • Boolean and natural number / their operators
    • “and”, “or”, “not”, “add”, and “subtract”
    • Properties of those operators
  • More complex examples in software foundation
Verification tutorial: pure function

“given two positive numbers, find sum of all numbers between two”

• Mathematical (functional) specs:

Definition range_sum (n1 n2 : nat) : nat :=
let (start, end) :=
  match (decide (n1 > n2)) with
  | left_ => (n2, n1)
  | right_ => (n1, n2)
in
  (end * (end - 1)
   - start * (start - 1)) / 2
end.

Program example:

• int range_sum (int n1, int n2) {
    int start = n1 > n2 ? n2 : n1;
    int end = n1 > n2 ? n1 : n2;
    int sum = 0;
    for (int i = start; i <= end; i++) {
        sum += i;
    }
    return sum;
}
Verification tutorial: pure function

Mathematical (functional) specifications

All possible inputs (n1, n2)

Generate same output (sum)

Low-level Implementation
Verification tutorial: abstract state

Software usually facilitates hardware states, memory and registers. Mathematical state could be much simpler than those physical states.

Mathematical (functional) list:

Variable \texttt{A} : Type.

Inductive \texttt{list} : Type :=
| nil : list |
| cons : A \rightarrow list \rightarrow list. |

Program example:

1) With array

\texttt{int array_list[kMaxLength];}

1) With linked list

\texttt{struct Node \{ int data; Node* next; Node* prev; \};}

Refinement relation (R): how mathematical list is related to the low-level structure.
Verification tutorial: abstract state

Mathematical (functional) specifications

(abs, args) \rightarrow (abs', ret)

((mem, reg), args) \rightarrow \cdots \rightarrow ((mem', reg'), ret)

Low-level Implementation

\textbf{R} \quad \textbf{R}

Verification tutorial: modularity

- Provide the total correctness (including memory accesses)
- Connect other verification results
  - Different application developers hope to verify their own software
  - HW & underlying software developers hope to provide formal verification
Verification tutorial: modularity

- Provide the total correctness (including memory accesses)
- Connect other verification results
  - Different application developers hope to verify their own software
  - HW & underlying software developers hope to provide formal verification
- Provide modular extension or rebuilding of our software and verification
Verification tutorial: modularity

Decompose the entire software into multiple sub components, verifying them, and combine their proofs together.
Verification tutorial: modularity

Mathematical (functional) specifications

\[(\text{mem}_h, \text{abs}_h, \text{reg}_h), \text{args})\] \rightarrow \[(\text{mem}_h', \text{abs}_h', \text{reg}_h'), \text{args})\]

Low-level Implementation

\[(\text{mem}_l, \text{abs}_l, \text{reg}_l), \text{args})\] \rightarrow \[(\text{mem}_l', \text{abs}_l', \text{reg}_l'), \text{args})\]

- **Contextual refinement**
  - Compositional approach to compositional verification of concurrent objects.
  - Combined with several program logics, it can show consistency between the object implementation and its abstract specification.
Verification tutorial: modularity

How can we effectively use high expressiveness?

Specification of our software

New abstractions by compose multiple modules

High-level spec

Low-level spec

C code

TCB (HW & underlying SW) abstraction

Abstracted model by hiding HW and C details

C friendly spec for easy correctness proof + C correctness proof

Specification of underlay
Verification tutorial: modularity

How can we effectively use high expressiveness?

- New abstractions by compose multiple modules
- TCB (HW & underlying SW) abstraction

Specification of our software

- High-level spec
  - High-level spec
  - Low-level spec
  - C code

Abstracted model by hiding HW and C details

- C friendly spec for easy correctness proof
  - C code
++

How can we reduce the very high cost?

- Specification of underlay
  - Low-level spec
++
  - C code
Conclusion
Conclusion

• Formal verification can reduce the cost for the poor software
  • Operational software failure cost
  • Cost due to poor legacy systems

• Formal verification
  • What is formal verification
  • Formal verification key concept
  • Modularity in formal verification