Introduction to formal verification

Jieung Kim

Aug. 2022

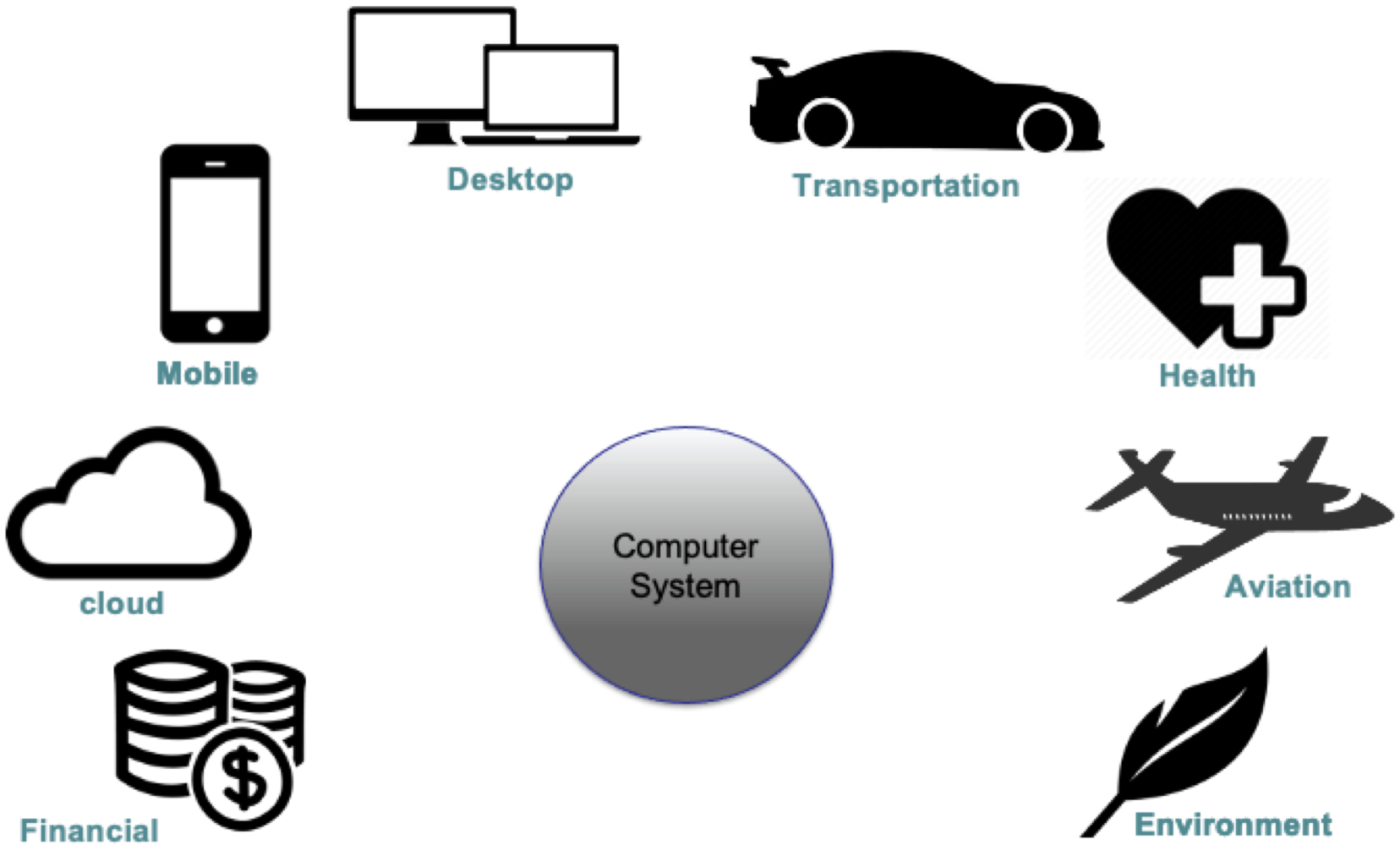
Contents

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

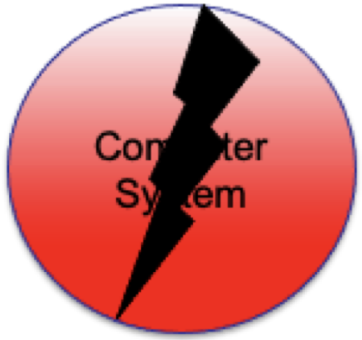
Crash

Accident

Mobile

Life

cloud



Loss

Financial

Environment

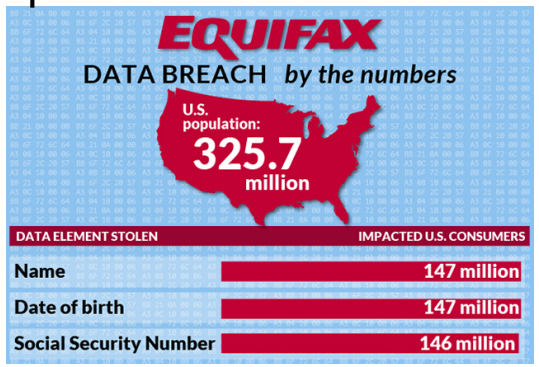
Software failure

Ariane 5 explosion
$370 million

1996

...

50% of American personal record



2018

Recalls More than 150,000 vehicles

2021~2022

Software failure

AUTHOR

HERB KRASNER

CISQ Advisory Board Member and retired Professor of Software Engineering at the University of Texas at Austin.

He can be reached at hkrasner@utexas.edu.

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

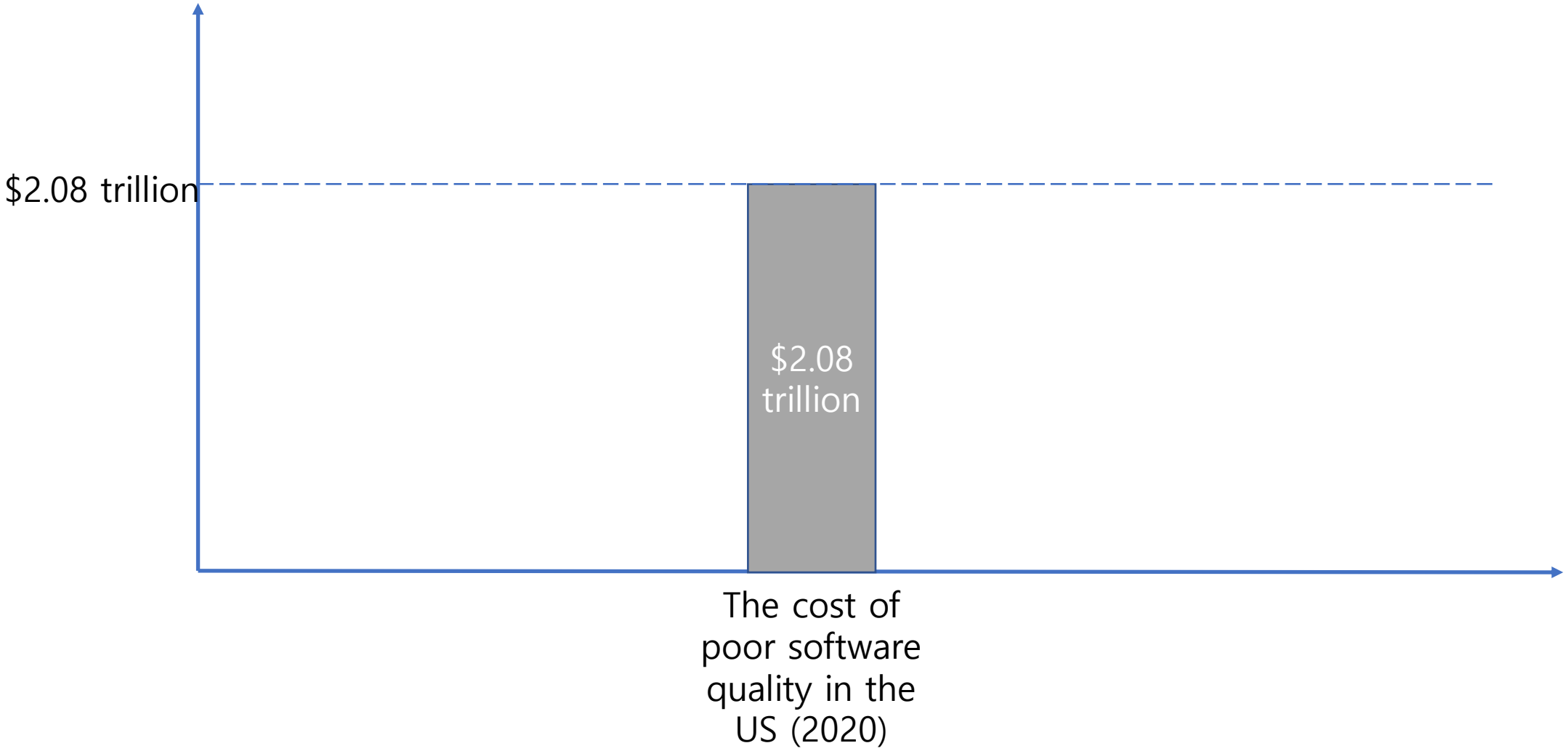
SYNOPSYS®
Silicon to Software™

Software Engineering Institute
Carnegie Mellon

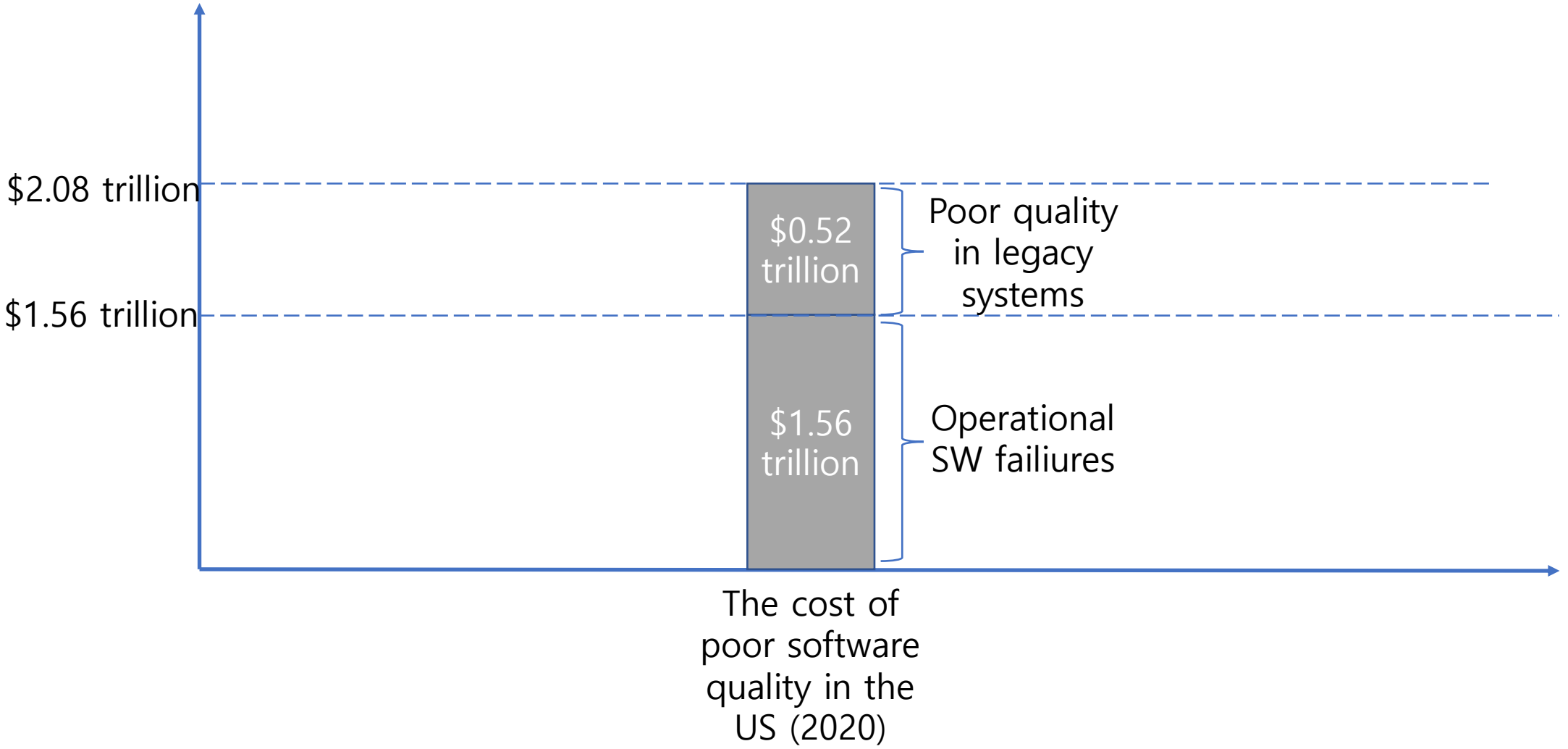
NORTHROP GRUMMAN

- 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)

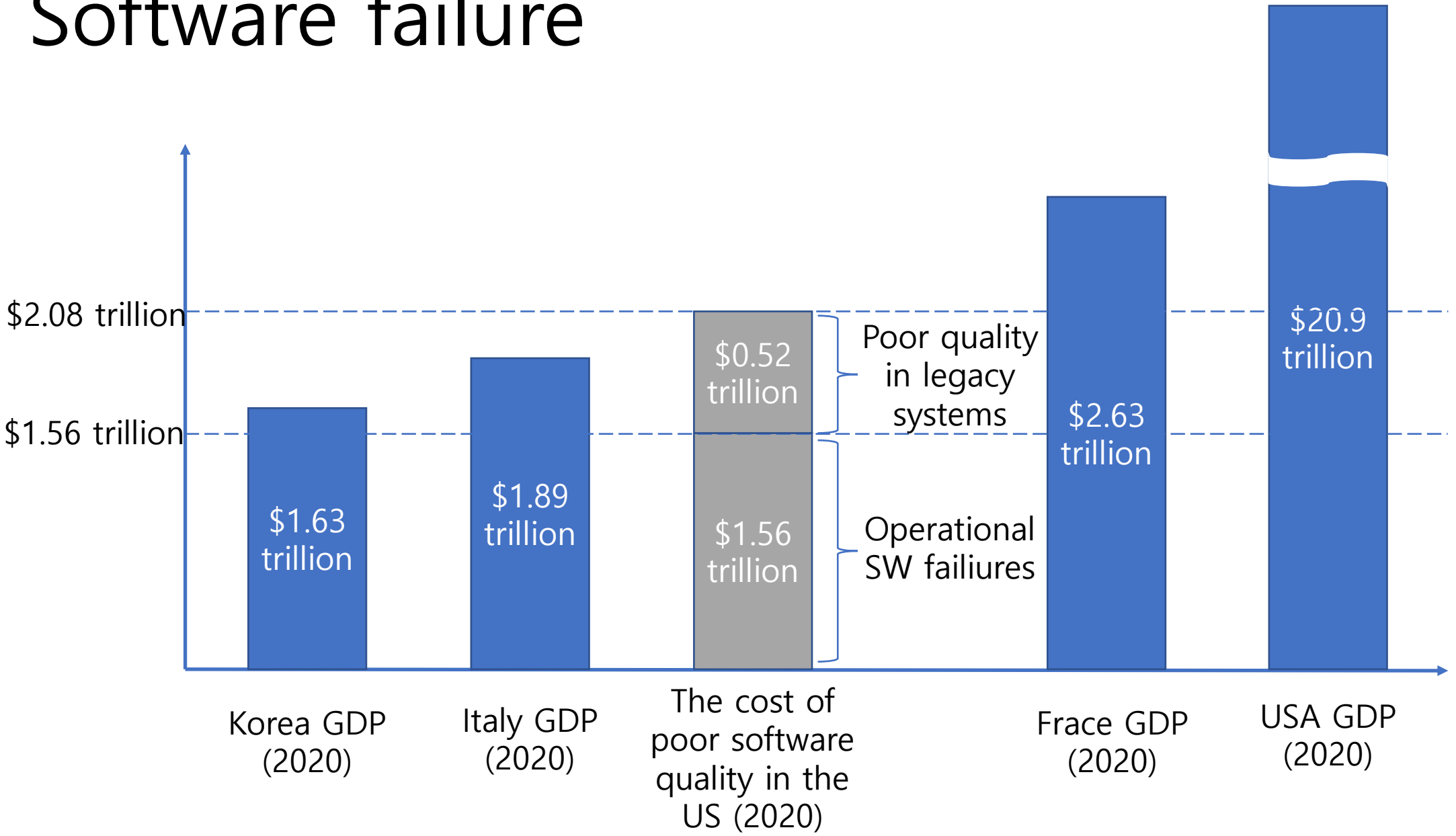
Software failure



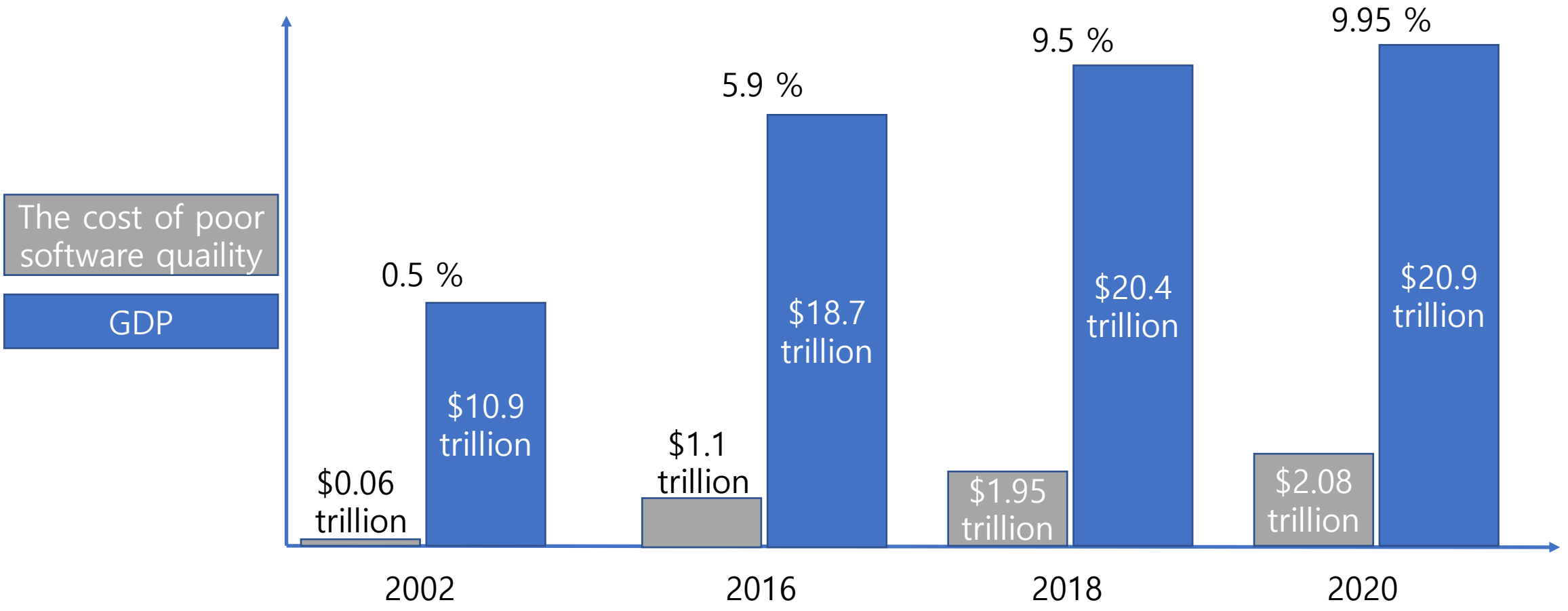
Software failure



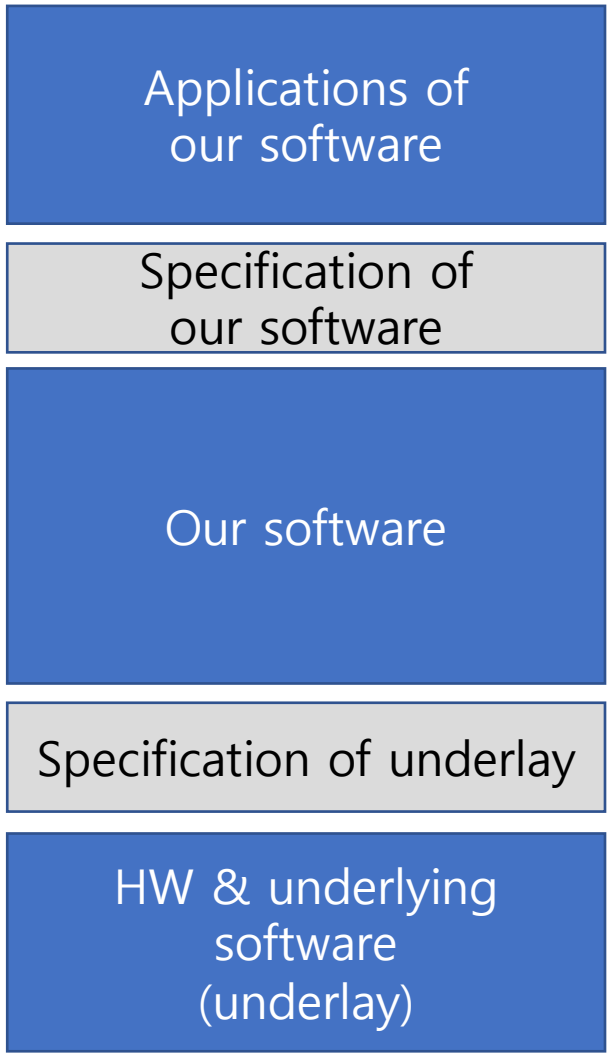
Software failure



Software failure



Sources of software failure



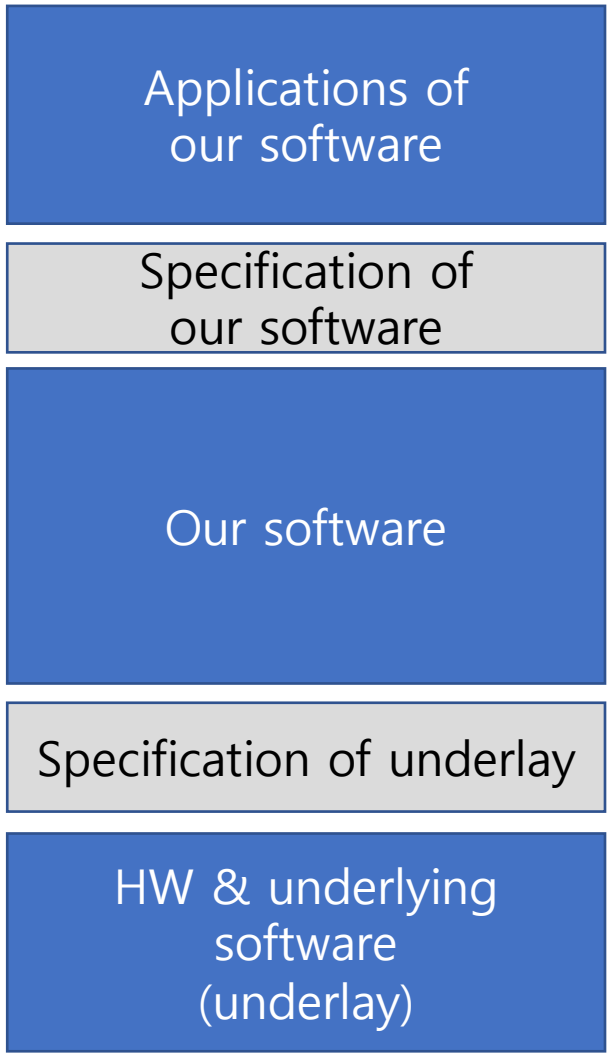
Bugs are due to

- 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

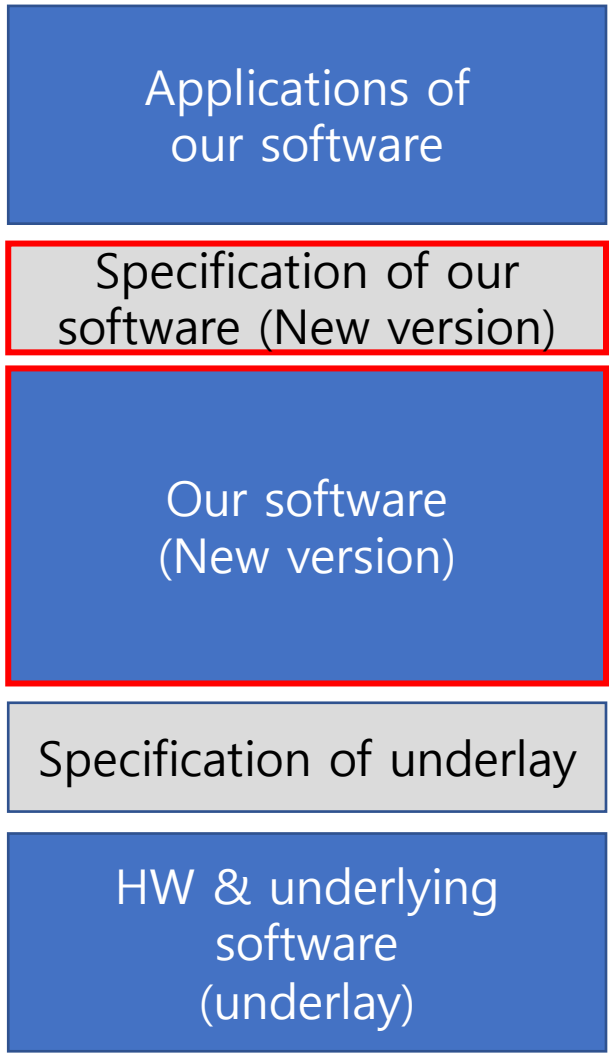
- Operational software failures
  - Show that the specification is correctly written
  - Show that the software faithfully implements its specification
- Poor quality in legacy system
  - Build a new well structured software that can replace or supplement the legacy system
  - Show that the new specification 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



Specification of our software (New version) ⊇ Specification of our software (old version)

Our software (new version) faithfully implements the specification (new version) based on the specification of HW and unerlying software

Tools for software assurance

	Expressiveness level	Assurance level	Cost level
Code review	Very high	Very low	Medium
Testing	Medium	Low	Medium
Type checker (Java, Haskell, Rust)	Low	High	low
Static & dynamic alaysis (Coverity, Infer)	Medium	Medium	low

Can those tools entirely tackle previous two challenges?

→ **NO!**

Tools for software assurance

practical

	Expressiveness level	Assurance level	Cost level
Code review	Very high	Very low	Medium
Testing	Medium	Low	Medium
Type checker (Java, Haskell, Rust)	Low	High	low
Static & dynamic alaysis (Coverity, Infer)	Medium	Medium	low
Formal verificaiton (Z3, Adga, Coq)	Medium ~ High	High ~ Very high	Medium ~ Very high

Tools for software assurance

practical

	Expressiveness level	Assurance level	Cost level
Code review	Very high	Very low	Medium
Testing	Medium	Low	Medium
Type checker (Java, Haskell, Rust)	Low	High	low
Static & dynamic alaysis (Coverity, Infer)	Medium	Medium	low
Formal verificaiton (Z3, Adga, Coq)	Medium ~ High	High ~ Very high	Medium ~ Very high

How can we effectively u se high expressiveness?

How can we avoid very high cost?

Tools for software assurance

- Does formal verification actually remove software bugs?

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.

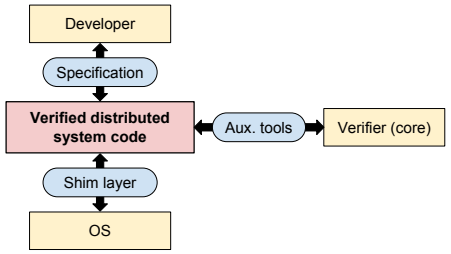
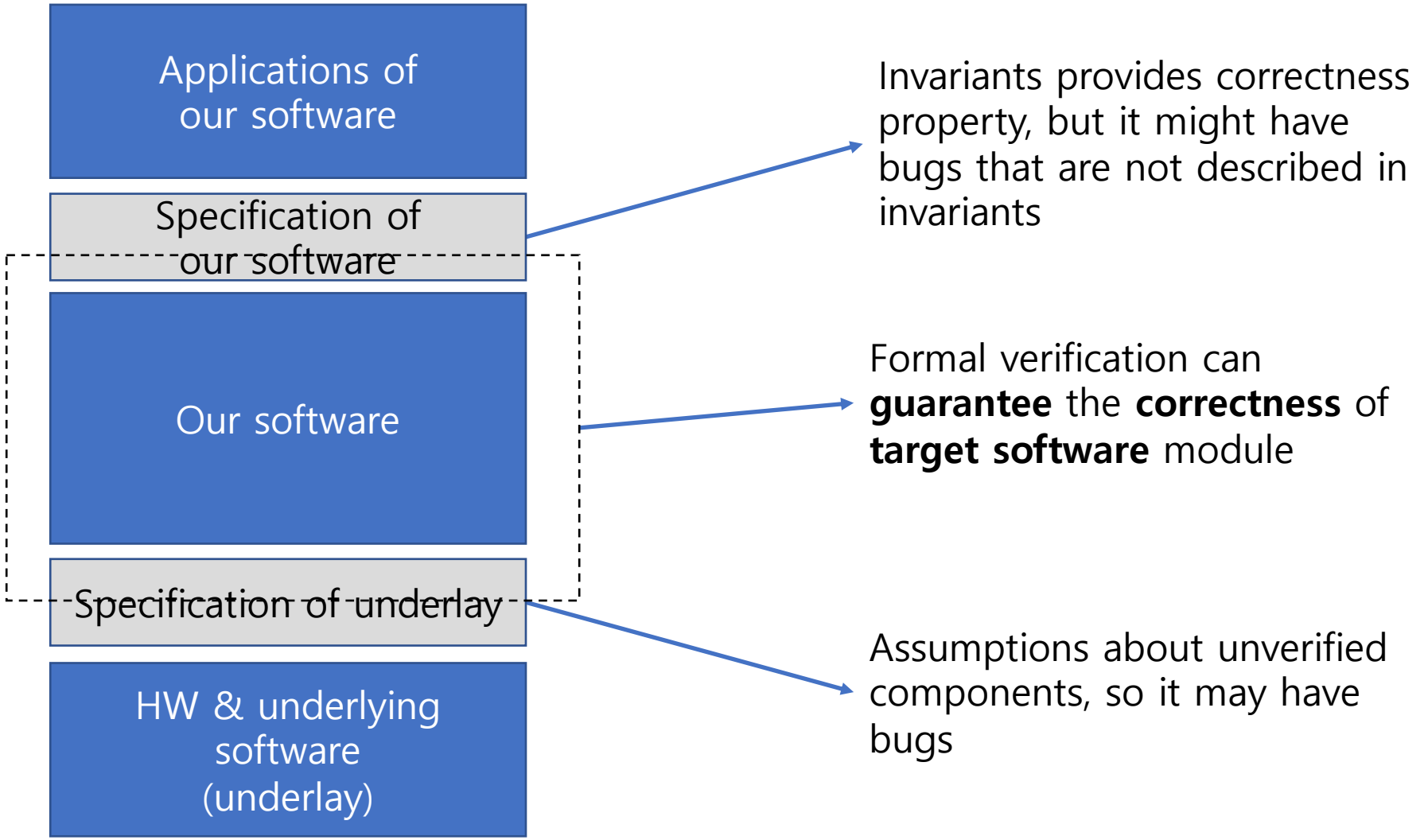


Figure 1: An overview of the workflow to verify a distributed system implementation.

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

Tools for software assurance



Tools for software assurance

What do we need to know for formal verification?

- It is built on top of lots of unerlying theories
- Verification engineers can only focus on the subset that is act ually required for the verfication target

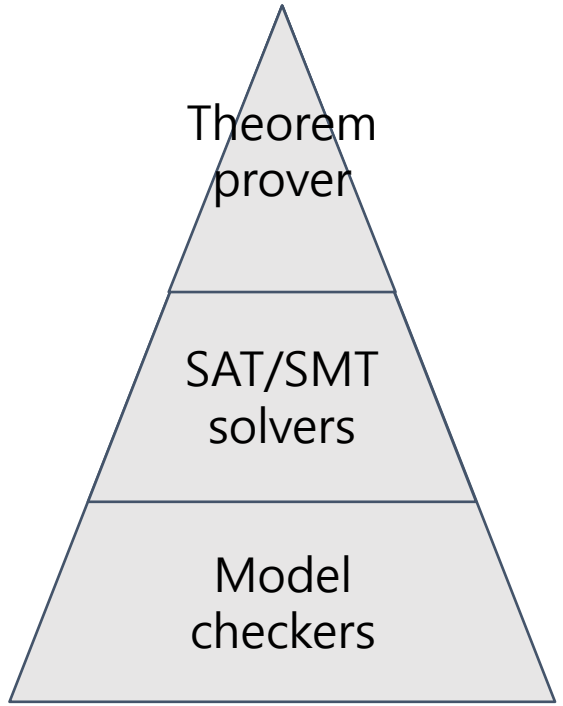
Formal verification intro with examples

Formal verification

Definition

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

Formal verification Hierarchy

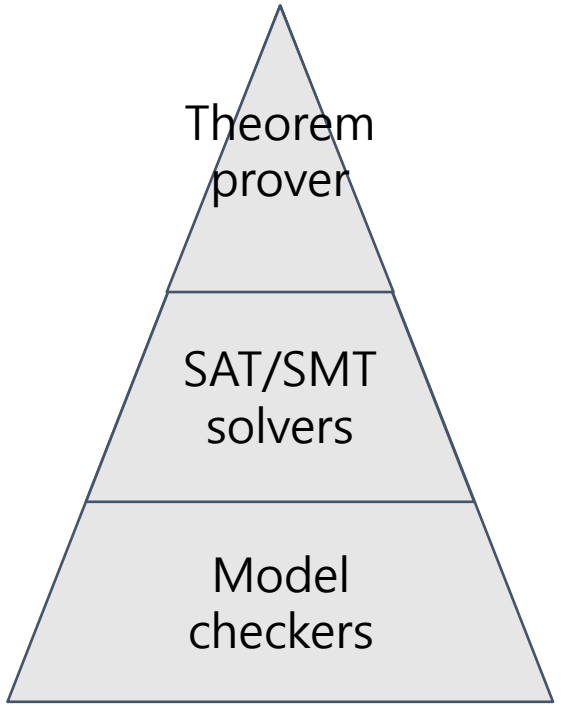


Formal verification hierarchy with different types of proof checkers

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

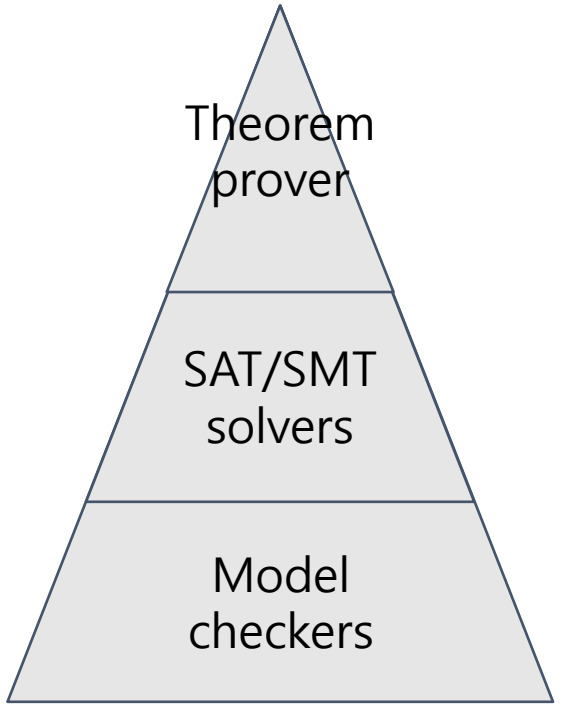


Formal verification hierarchy with different types of proof checkers

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



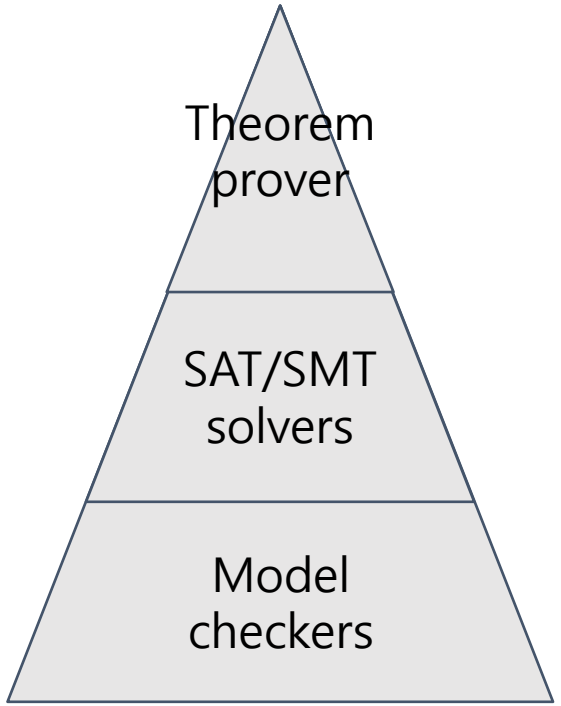
Formal verification hierarchy with different types of proof checkers

Formal verification Hierarchy

Theorem provers

- Supports a **very expressiveness specificatio n 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

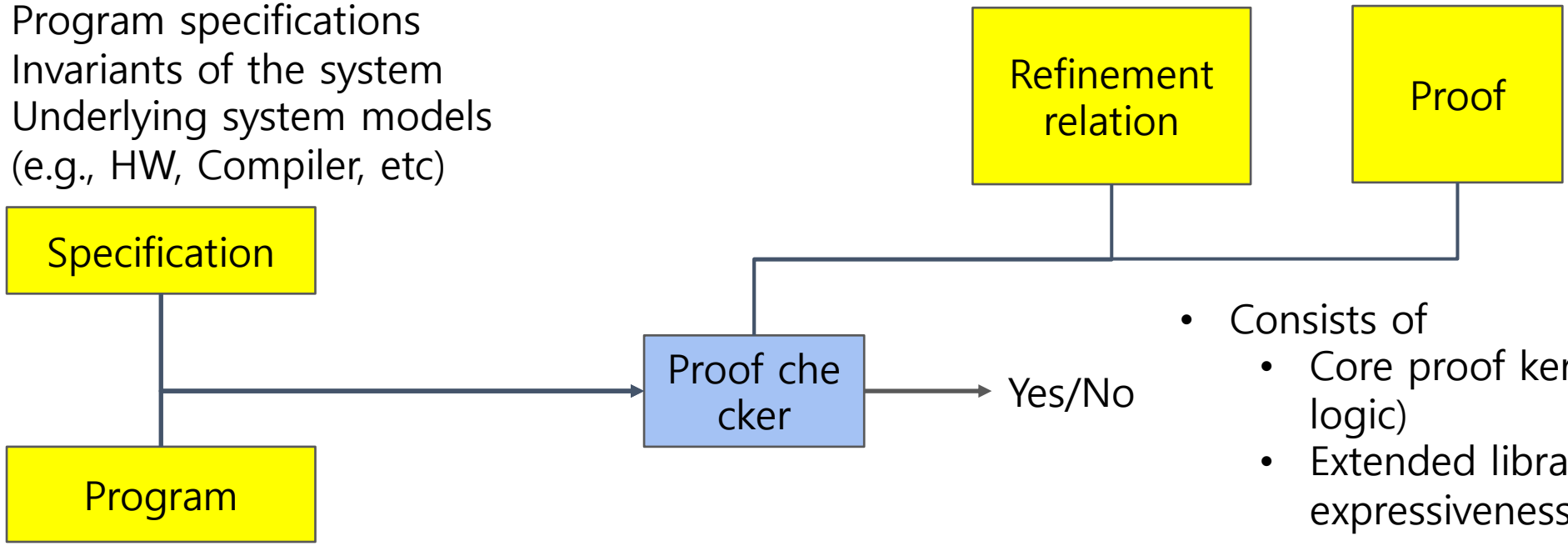


Formal verification hierarchy with different types of proof checkers

Key components

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

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



- Consists of
  - Core proof kernel (underlying logic)
  - Extended libraries for better expressiveness

- Subject of formal verificaiton

Working on formal verification

Working on formal verificaiton is similar to how we make a software

.

- Formal specification and design proofs
  - How can we provide a good abstract model for program ➔ Find the algorithm
    - Good: simple but correct
  - How can we design proofs with lower human efforts ➔ With low complexity

  ➔ **Problem solving / System design**

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

  ➔ Properly express them with pro gramming languages

  ➔ **Coding**

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"

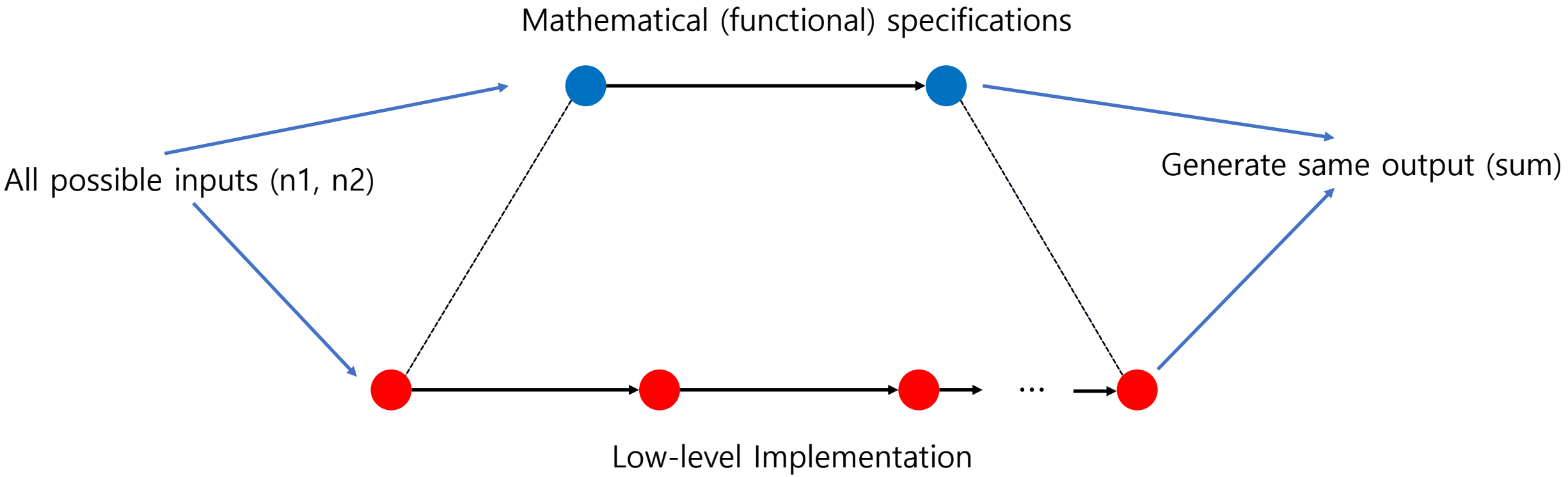
- Mathematical (functional) specs:

```
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



Verification tutorial: abstract state

Software usually facilitates hardware states, memory and registers.

Mathematical state could be much simplier than those physical states.

Mathmatical (functional) list:

```
Variable A : Type.

Inductive list : Type :=
  | nil : list
  | cons : A -> list -> list.
```

Program example:

1) With array

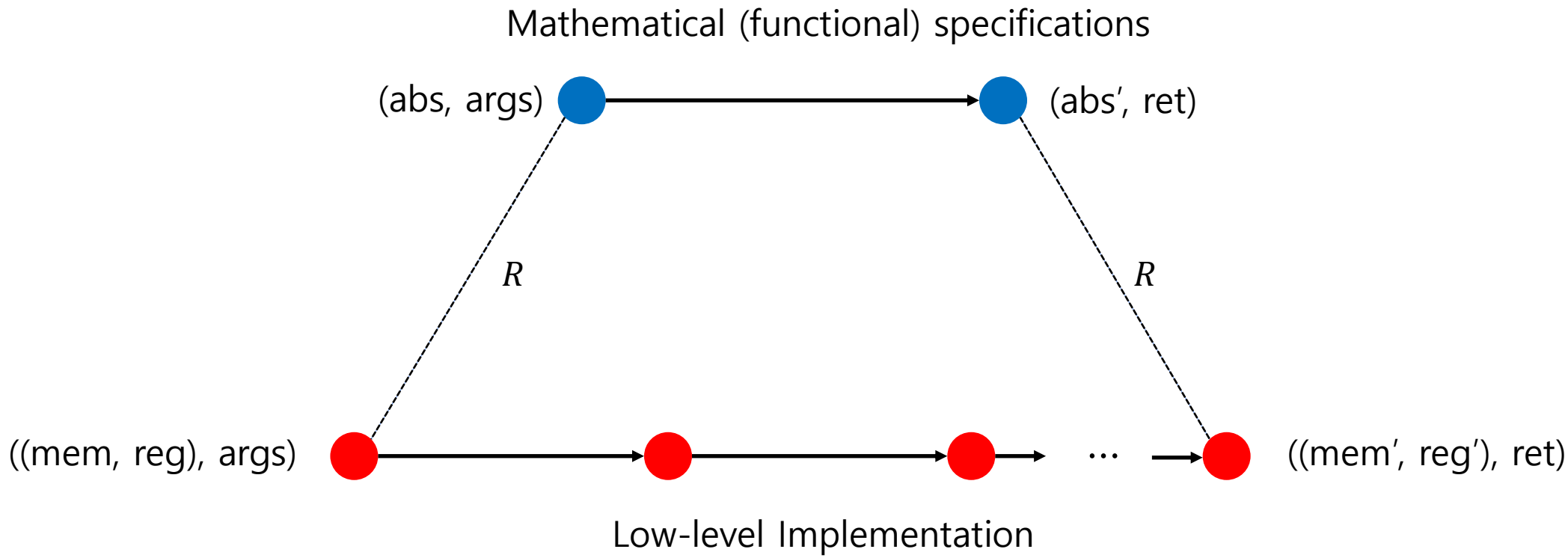
```
int array_list[kMaxLength];
```

1) With linked list

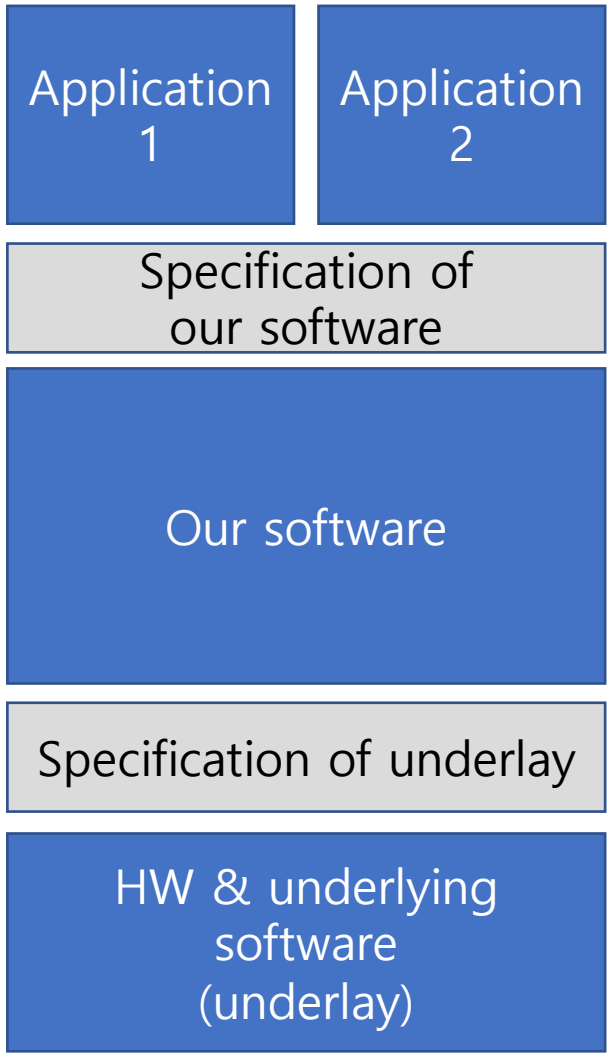
```
struct Node {
    int data;
    Node* next;
    Node* prev;
};
```

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

Verification tutorial: abstract state

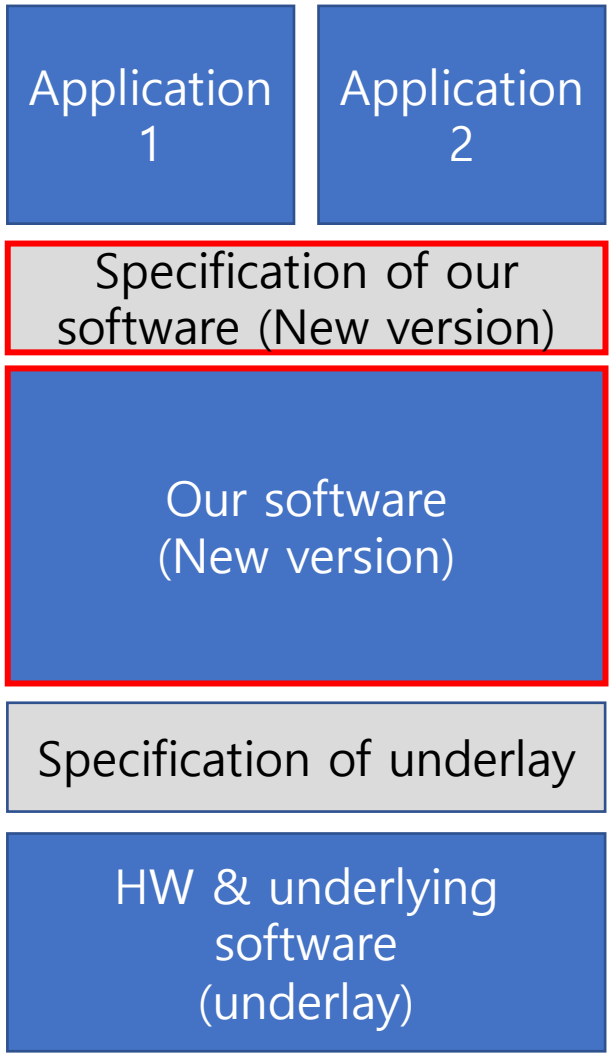


Verification tutorial: modularity



- 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

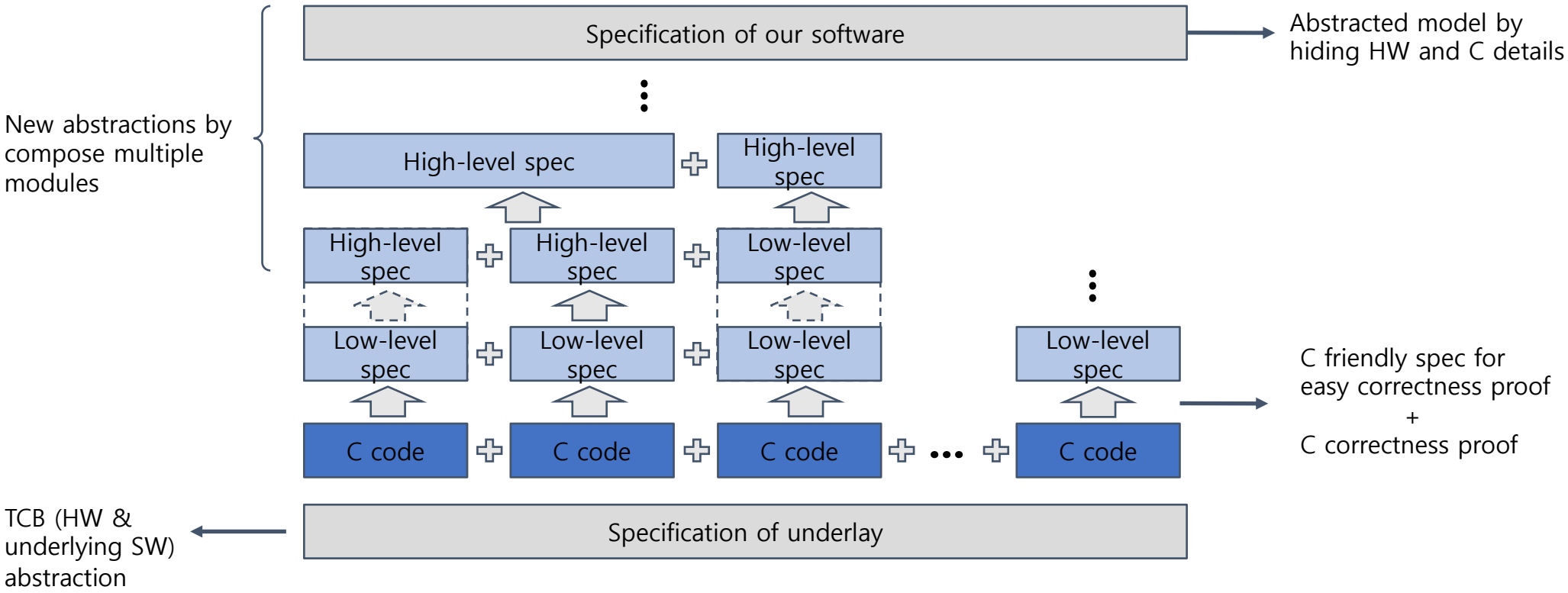
Verification tutorial: modularity



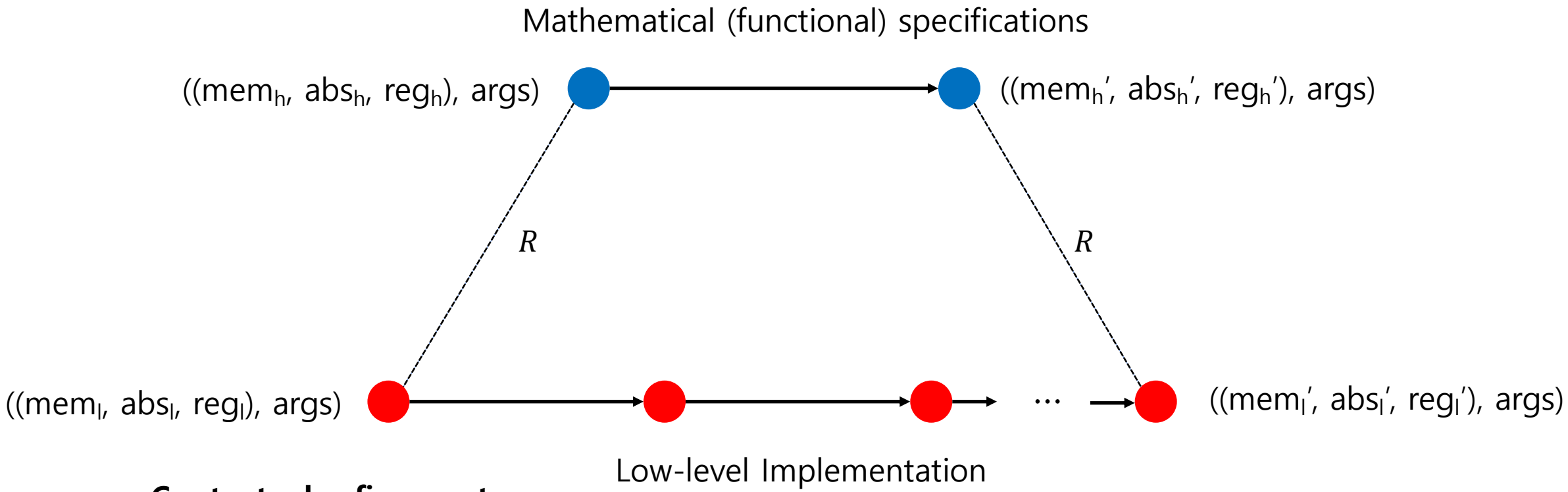
- Provide the total correctness (including memory accesses)
- Connect other verification results
  - Different application developers hope to verify their own software
  - HW & underlying software developers hope to provide formal verification
- Provide modular extension or rebuilding of our software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

Verification tutorial: modularity

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



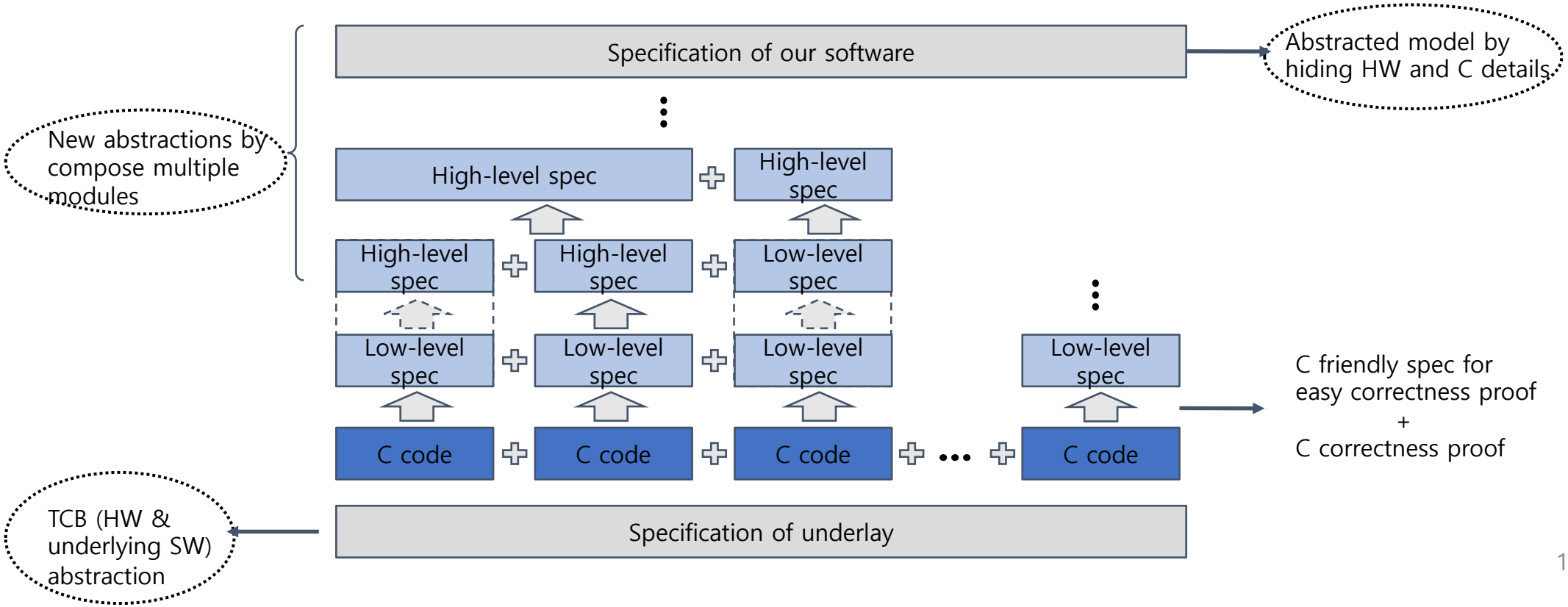
Verification tutorial: modularity



- **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

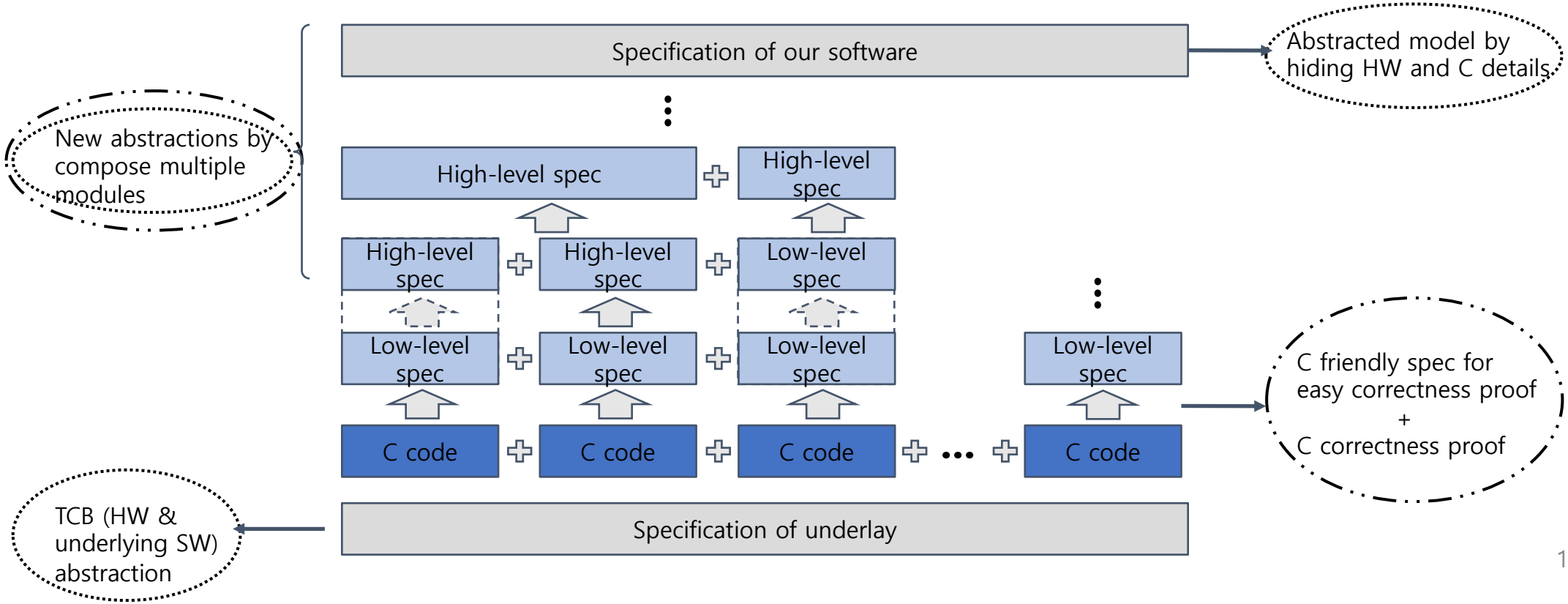
How can we effectively u
se high expressiveness?



Verification tutorial: modularity

How can we effectively u se high expressiveness?

How can we reduce the v ery high cost?



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