Introduction to formal verification

TechTalk @ Google
2022-08-11

Jieung Kim
Contents

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

Ariane 5 explosion
$370 million
1996

... 2018

50% of American personal record

Recalls More than 150,000 vehicles
2021~2022
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 $835 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) is $2.08 trillion.
Software failure

$2.08 trillion

$1.89 trillion
Italy GDP (2020)

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

$2.63 trillion
France GDP (2020)
The cost of poor software quality is a significant portion of GDP. Over the years from 2002 to 2020, the cost has increased substantially. In 2002, the cost was $0.06 trillion, representing 0.5% of GDP. By 2016, the cost had increased to $1.1 trillion, accounting for 5.9% of GDP. In 2018, the cost reached $1.95 trillion, which was 9.5% of GDP. By 2020, the cost had further increased to $2.08 trillion, representing 9.95% of GDP.
Software failure

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

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

Total: $2.08 trillion
Reduce operational software failures

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

Applications of our software

Specification of our software (New version)

Our software (New version)

Specification of underlay

HW & underlying software (underlay)

⇒

Specification of our software (New version)

Our software (new version) faithfully implements the specification (new version) based on underlying HW and software specifications

Specification of our software (old version)
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 analysis (Coverity, Infer)</td>
<td>Medium</td>
<td>Medium</td>
<td>low</td>
</tr>
</tbody>
</table>
### Tools for software assurance

<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 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?**
Tools for software assurance

What do we need to know for formal verification?

• It is built on top of lots of underlying theories

• But, verification engineers can only focus on the tiny subset that is actually required for the verification target
Can it actually remove 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.

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 [58]. The struggle to improve their reliability spawned several important lines of research, such as programming abstractions [5, 38, 46], bug-finding tools [27, 39, 55, 56], and formal verification techniques [23, 30, 36, 54].

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

EuroSys ’17
April 23-26, 2017, Belgrade, Serbia
© 2017 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-4938-3/17/04...
DOI: http://dx.doi.org/10.1145/3064176.3064183

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 large single-node implementations, such as the seL4 OS kernel [28] and the CompCert compiler [35]. More recently,
Can it actually remove bugs?

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.
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
Key components

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

- Subject of formal verification

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

- Consists of
  - Core proof kernel (underlying logic)
  - Extended libraries for better expressiveness
Verification tutorial: simple stateless function

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

• Mathematical (functional) specs:

Definition range_sum (start end : nat) : nat :=
  ...
  (end * (end - 1) - start * (start - 1)) / 2
  end.

Program example:

int range_sum (int start, int end) {
  int sum = 0;
  ...
  for (int i = start; i <= end; i++) {
    sum += i;
  }
  return sum;
}
Verification tutorial: simple stateless function

Mathematical (functional) specifications

All possible inputs (start, end)

Low-level Implementation

Generate same output (sum)
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 $A : \text{Type}$.  

Inductive $\text{list} : \text{Type} :=$ 

| nil : list  
| cons : $A \rightarrow \text{list} \rightarrow \text{list}$.  

Program example:

1) With array

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

1) With linked list

```c
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

Low-level Implementation

\[(\text{abs}, \text{args}) \rightarrow (\text{abs}', \text{ret})\]

\[
\begin{align*}
\text{R} & \quad \text{R} \\
((\text{mem, reg}), \text{args}) & \rightarrow \cdots \rightarrow ((\text{mem}', \text{reg}'), \text{ret})
\end{align*}
\]
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

New abstractions by compose multiple modules

Specification of our software

Abducted model by hiding HW and C details

High-level spec

Low-level spec

C code

Specification of underlay

TCB (HW & underlying SW) abstraction

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

Specification of underlay

Abstracted model by hiding HW and C details

C friendly spec for easy correctness proof + C correctness proof
Verification tutorial: modularity

How can we effectively use high expressiveness?

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

How can we reduce the very high cost?

- Abstracted model by hiding HW and C details
- C friendly spec for easy correctness proof + C correctness proof

Specification of our software

High-level spec

High-level spec

High-level spec

Low-level spec

Low-level spec

Low-level spec

Low-level spec

C code

C code

C code

C code

Specification of underlay
Formal verification projects
My formal verification researches

- CertiKOS – Small OS and hypervisor
  - Safety and Liveness of MCS Lock - Layer by Layer. APLAS 2017
  - Certified concurrent abstraction layers. PLDI 2018
  - Building certified concurrent OS kernels. Comm. of ACM 62(10) 2019

- ADO (Atomic Distributed Object) – Distributed system
  - WormSpace: A Modular Foundation for Simple, Verifiable Distributed Systems. SoCC 2019
  - Much ADO about failures: a fault-aware model for compositional verification of strongly consistent distributed systems. Proc. ACM Program. Lang. 5(OOPSLA)
  - Adore: Atomic Distributed Objects with Certified Reconfiguration, PLDI 2022

- pKVM formal verification – Practical hypervisor
Distributed system verification

Distributed system
Distributed system software stack

- Distributed services (e.g., KV store).
- SMR (High-level API)
- Distributed protocols (Low-level API)
Network-based models too complex

```
S 3
{ "abc":"def" }
```

```
S 1
{ "abc":"def" "foo":"bar" }
```

```
S 2
{ "abc":"def" "foo":"bar" }
```

prepare

```
S 1
time = 2
leader = true
```

```
S 2
time = 2
leader = false
```

```
S 3
time = 1
leader = false
```

commit

ack

```
36
```
Network-based models too complex

Challenges
Network errors

```
{    "abc":"def"
    "foo":"bar"
}
```

```
{    "abc":"def"
    "foo":"bar"
}
```

```
{    "abc":"def"
    "foo":"bar"
}
```

prepare

```
S 1
    time = 2
    leader = true
    {
    "abc":"def"
    "foo":"bar"
    }
```

```
S 2
    time = 2
    leader = false
    {
    "abc":"def"
    "foo":"bar"
    }
```

```
S 3
    time = 1
    leader = false
    {
    "abc":"def"
    }
```

commit

```
S 1
    time = 2
    leader = true
    {
    "abc":"def"
    "foo":"bar"
    }
```

ack
Network-based models too complex

Challenges
- Network errors
- Protocol subtleties

S1
- time = 2
- leader = true
- 
  "abc" : "def"
  "foo" : "bar"

S2
- time = 2
- leader = false
- 
  "abc" : "def"
  "foo" : "bar"

S3
- time = 1
- leader = false
- 
  "abc" : "def"

prepare

commit

ack
Network-based models too complex

Challenges
- Network errors
- Protocol subtleties
- Application (distributed service) bugs

S1
- time = 2
- leader = true
- \{ "abc":"def" , "foo":"bar" \}

S2
- time = 2
- leader = false
- \{ "abc":"def" , "foo":"bar" \}

S3
- time = 1
- leader = false
- \{ "abc":"def" \}

Prepare

Commit

Ack
Network-based models too complex

Distributed system software stack

- Distributed services (e.g., KV store)
- SMR (High-level API)
- Distributed protocols (Low-level API)

- Non-determinism
  - Complex interleaving
  - Network & node errors
- Several protocols and implementations (paxos, raft, chain-replication, etc)
- Lots of verification works done
State machine replication too abstract
State machine replication too abstract

Distributed system software stack

- Distributed services (e.g., KV store).
- SMR (High-level API)
- Distributed protocols (Low-level API)

- Deterministic
- Unified abstraction
- Non-determinism
  - Complex interleaving
  - Network & node errors
- Several protocols and implementations (paxos, raft, chain-replication, etc)
- Lots of verification works done
Partial failure

S1
{
   "abc":"def"
}

S2
{
   "abc":"def"
}

S3
{
   "abc":"def"
}
Partial failure

```
{
  "abc":"def"
  "foo":"bar"
}
S1
```

```
{
  "abc":"def"
}
S2
```

```
{
  "abc":"def"
}
S3
```

Alice

“foo”:”bar”
Partial failure

S1
{  "abc" : "def"  "foo" : "bar" }

S2
{  "abc" : "def" }

S3
{  "abc" : "def" }

Bob

Read
Partial failure
Partial failure is important

**Partial failure is a central reality of distributed computing. [...] Being robust in the face of partial failure requires some expression at the interface level.** *(Jim Waldo. A Note on Distributed Computing. 1994)*

- Unavoidable feature unique to distributed systems
- Influence with all aspects of distributed protocols (e.g., leader election and reconfiguration)
- Can be used for performance optimizations
  - TAPIR (SOSP ’15): Transactions with out-of-order commits
  - Speculator (SOSP ’05): Speculative distributed file system
Partial failure is important

- Deterministic
- Unified abstraction
- Non-determinism
  - Complex interleaving
  - Network & node errors
- Several protocols and implementations (paxos, raft, chain-replication, etc)
- Lots of verification works done
ADO (Atomic distributed object)

Distributed system software stack

- Distributed services (e.g., KV store)
- SMR (High-level API)
- ADO (Atomic distributed object)
- Distributed protocols (Low-level API)

- Deterministic
- Unified abstraction
- Non-determinism
  - Complex interleaving
  - Network & node errors
- Several protocols and implementations (paxos, raft, chain-replication, etc)
- Lots of verification works done

Simple, but non-deterministic abstraction
Covers all protocols
Make connection between two APIs possible
## ADO state

<table>
<thead>
<tr>
<th>“abc”:“def”</th>
<th>“foo”:“bar”</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

### ADO Legend

- **Method**
- **Timestamp**
- **Persistent Log**
- **Entry**
ADO state

```
"abc":"def"  1
"foo":"bar"  2
"xyz":"123"  4
"cat":"dog"  3
"dot":"cot"  3
```

ADO Legend

- **Method**
- **Timestamp**
- Persistent Log Entry
- Cache Tree Entry
ADO operations

```
"abc":"def" 1
"foo":"bar" 2
"cat":"dog" 3
"xyz":"123" 4
"dot":"cot" 3
```

Pull
- Prepare
- Refine

Invoke
- Local Update

Push
- Commit
ADO operations

ADO

```
"abc":"def" 1
"foo":"bar" 2
```

```
"cat":"dog" 3
"dot":"cot" 3

"xyz":"123" 4
```

Multi-Paxos

S1

```
"abc":"def" 1
"foo":"bar" 2
```

S2

```
"abc":"def" 1
"foo":"bar" 2
"xyz":"123" 4
```

S3

```
"abc":"def" 1
"foo":"bar" 2
"cat":"dog" 3
"dot":"cot" 3
```
ADO operations

ADO

```
<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;abc&quot;</td>
<td>&quot;def&quot;</td>
<td>1</td>
</tr>
<tr>
<td>&quot;foo&quot;</td>
<td>&quot;bar&quot;</td>
<td>2</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;cat&quot;</td>
<td>&quot;dog&quot;</td>
<td>3</td>
</tr>
<tr>
<td>&quot;dot&quot;</td>
<td>&quot;cot&quot;</td>
<td>3</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;xyz&quot;</td>
<td>&quot;123&quot;</td>
<td>4</td>
</tr>
</tbody>
</table>
```

Multi-Paxos

```
<table>
<thead>
<tr>
<th>Node</th>
<th>Key</th>
<th>Value</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>&quot;abc&quot;</td>
<td>&quot;def&quot;</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>&quot;foo&quot;</td>
<td>&quot;bar&quot;</td>
<td>2</td>
</tr>
<tr>
<td>S2</td>
<td>&quot;abc&quot;</td>
<td>&quot;def&quot;</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>&quot;foo&quot;</td>
<td>&quot;bar&quot;</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>&quot;xyz&quot;</td>
<td>&quot;123&quot;</td>
<td>4</td>
</tr>
<tr>
<td>S3</td>
<td>&quot;abc&quot;</td>
<td>&quot;def&quot;</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>&quot;foo&quot;</td>
<td>&quot;bar&quot;</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>&quot;cat&quot;</td>
<td>&quot;dog&quot;</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>&quot;dot&quot;</td>
<td>&quot;cot&quot;</td>
<td>3</td>
</tr>
</tbody>
</table>
```
ADO operations

```
ADO

"abc": "def" 1
"foo": "bar" 2

"cat": "dog" 3
"dot": "cot" 3

"xyz": "123" 4
```

Multi-Paxos

```
S1
5

S2
5

S3
5

"abc": "def" 1
"foo": "bar" 2

"cat": "dog" 3
"dot": "cot" 3

"xyz": "123" 4
```

Prepare

S1
S3
S2
ADO operations

ADO

```
"abc":"def"
1

"foo":"bar"
2

"xyz":"123"
4

"cat":"dog"
3

"dot":"cot"
3
```

Multi-Paxos

```
S1

"abc":"def"
1

"foo": "bar"
2

"xyz": "123"
4

S2

"abc": "def"
1

"foo": "bar"
2

"xyz": "123"
4

S3

"abc": "def"
1

"foo": "bar"
2

"cat": "dog"
3

"dot": "cot"
3
```
Get permission to update and select a starting point in the cache tree.
ADO operations

ADO

```
"abc": "def" 1
"foo": "bar" 2
"xyz": "123" 4
"cat": "dog" 3
"dot": "cot" 3

"xyz": "123" 4
```

Multi-Paxos

<table>
<thead>
<tr>
<th></th>
<th>S1 5</th>
<th>S2 5</th>
<th>S3 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&quot;abc&quot;: &quot;def&quot; 1</td>
<td>&quot;abc&quot;: &quot;def&quot; 1</td>
<td>&quot;abc&quot;: &quot;def&quot; 1</td>
</tr>
<tr>
<td>2</td>
<td>&quot;foo&quot;: &quot;bar&quot; 2</td>
<td>&quot;foo&quot;: &quot;bar&quot; 2</td>
<td>&quot;foo&quot;: &quot;bar&quot; 2</td>
</tr>
<tr>
<td>4</td>
<td>&quot;xyz&quot;: &quot;123&quot; 4</td>
<td>&quot;xyz&quot;: &quot;123&quot; 4</td>
<td>&quot;xyz&quot;: &quot;123&quot; 4</td>
</tr>
<tr>
<td>5</td>
<td>&quot;bee&quot;: &quot;gee&quot; 5</td>
<td>&quot;bad&quot;: &quot;cot&quot; 5</td>
<td>&quot;bad&quot;: &quot;cot&quot; 5</td>
</tr>
</tbody>
</table>

"bad": "cot"
ADO operations

Invoking a Method

Add a new entry to the cache tree.
ADO operations

Multi-Paxos

Invoking a Method
Add a new entry to the cache tree.
ADO operations

ADO

```
"abc": "def" 1
"foo": "bar" 2
```

```
"cat": "dog" 3
"xyz": "123" 4
"bee": "gee" 5
"bad": "cot" 5
```

Multi-Paxos

```
S1
5
"abc": "def" 1
"foo": "bar" 2
"xyz": "123" 4
"bee": "gee" 5
"bad": "cot" 5
```

```
S2
5
"abc": "def" 1
"foo": "bar" 2
"xyz": "123" 4
"bee": "gee" 5
```

```
S3
5
"abc": "def" 1
"foo": "bar" 2
"xyz": "123" 4
"bee": "gee" 5
```

Commit
ADO operations

ADO

```
<table>
<thead>
<tr>
<th></th>
<th>“abc”:“def”</th>
<th>“foo”:“bar”</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>S2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>S3</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th></th>
<th>“cat”:“dog”</th>
<th>“dot”:“cot”</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>S2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th></th>
<th>“xyz”:“123”</th>
<th>“bee”:“gee”</th>
<th>“bad”:“cot”</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>4</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>S2</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>S3</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
```

Multi-Paxos

```
<table>
<thead>
<tr>
<th></th>
<th>“abc”:“def”</th>
<th>“foo”:“bar”</th>
<th>“xyz”:“123”</th>
<th>“bee”:“gee”</th>
<th>“bad”:“cot”</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>S2</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>S3</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
```

Commit
ADO operations

Multi-Paxos

S1
1. "abc"="def"
2. "foo"="bar"
3. "xyz"="123"
4. "bee"="gee"
5. "bad"="cot"

S2
1. "abc"="def"
2. "foo"="bar"
3. "xyz"="123"
4. "bee"="gee"

S3
1. "abc"="def"
2. "foo"="bar"
3. "xyz"="123"
4. "bee"="gee"

Push
Move committed methods into the log and prune stale states from the tree.
ADO operations

Multi-Paxos

Push
Move committed methods into the log and prune stale states from the tree.
ADO operations

**ADO**
- Push

```
"abc": "def"
1
"foo": "bar"
2
"xyz": "123"
4
"bee": "gee"
5
```

---

Multi-Paxos

```
S1

"abc": "def"
1
"foo": "bar"
2
"xyz": "123"
4
"bee": "gee"
5
"bad": "cot"
5
```

```
S2

"abc": "def"
1
"foo": "bar"
2
"xyz": "123"
4
"bee": "gee"
5
```

```
S3

"abc": "def"
1
"foo": "bar"
2
"xyz": "123"
4
"bee": "gee"
5
```

---

**Push**

Move committed methods into the log and prune stale states from the tree.
Connection with distributed protocols
Connection with distributed protocols
Distributed applications
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

• ADO: formal verification project example
  • Distributed system formal verification
  • Unified and modular program abstractions for distributed systems