Argosy: Verifying layered storage systems with recovery refinement

Tej Chajed, Joseph Tassarotti, Frans Kaashoek, Nickolai Zeldovich

MIT
logical disk

\begin{itemize}
  \item disk_1
  \item disk_2
\end{itemize}

Bob writes a replication system
Bob writes a replication system
logical disk

write

disk₁

Bob writes a replication system

disk₂
Bob writes a replication system
logical disk

\[ \text{write}_1 \rightarrow \text{rep\_recover} \]

disk\_1

disk\_2

Bob writes a replication system and implements its recovery procedure
Bob writes a replication system and implements its recovery procedure.
Bob is careful and writes a machine-checked proof of correctness.

Replication
- read
- write
- rep_recover

Read and write are atomic if you run rep_recover after every crash.
write-ahead logging

... log_recover

ops are atomic if you run log_recover after every crash
Transactions

- write-ahead log

Disk interface

- replication

Two-disk interface

logging + replication
Challenge: crashes during composed recovery

rep_recover \(\checkmark\) under crashes

log_recover \(\checkmark\) under crashes

rep_recover ; log_recover ?

how do we prove correctness under crashes using the existing proofs?
Prior work cannot handle multiple recovery procedures

CHL [SOSP ’15] not modular

Yggdrasil [OSDI ’16] single recovery

Flashix [SCP ’16] restricted recovery procedures
Argosy supports modular recovery proofs

- Transactions
  - write-ahead log
  - Disk interface
    - replication
  - Two-disk interface

developer proves
Argosy supports modular recovery proofs

Transactions

write-ahead log

Disk interface

replication

Two-disk interface

logging + replication

Argosy proves
Argosy is compatible with existing techniques

Transactions

write-ahead log 🔴

Disk interface

replication 🔴

Two-disk interface

prove with Crash Hoare Logic [SOSP '15]
Contributions

Recovery refinement for modular proofs
CHL for proving recovery refinement

see paper  Verified example: logging + replication

see code  Machine-checked proofs in Coq
Preview: recovery refinement

1. Normal execution correctness using *refinement*
2. Crash and recovery correctness using *recovery refinement*
Refinement
Disk interface

replication

Two-disk interface
correctness is based on how we use replication:
run code using Disk interface on top of two disks

Disk interface
- read
- write

replication
- read_impl
- write_impl

Two-disk interface
- write_1
- write_2
- read_1
- read_2

code
- read
- write

code_impl
- read_impl
- write_impl
Correctness: trace inclusion

Disk interface ➔ replication ➔ Two-disk interface

code ➔ code_impl

spec's behaviors ⊇ running code's behaviors
Proving correctness with an abstraction relation

1. developer provides abstraction relation R
Proving correctness with an abstraction relation

1. developer provides abstraction relation $R$
2. prove spec execution exists
3. and abstraction relation is preserved
Recovery refinement
Extending trace inclusion with recovery

Disk interface

replication

Two-disk interface

code

\exists

code_impl

specification for crash behavior

\exists

crash & recovery behavior
Extending trace inclusion with recovery

Disk interface

replication

Two-disk interface

code

\[
\Rightarrow
\]

code_impl

specification for crash behavior

\[
\Rightarrow
\]

crash & recovery behavior

crash semantics

? 

recovery semantics

? 

recover

19
Disk interface → replication
Two-disk interface

code

code_impl

code

summary

crash & recovery behavior

recovery semantics

recover
Disk interface

replication

Two-disk interface

code

2

code_impl

code

2

code_impl

recover

recover

zero-or-more iterations
Trace inclusion, with recovery
Proving trace inclusion, with recovery
Proving trace inclusion, with recovery

crash must occur during some operation
Proving trace inclusion, with recovery
Proving trace inclusion, with recovery
Proving trace inclusion, with recovery
Recovery refinement

non-crash execution

\[ \text{op} \rightarrow \text{op_impl} \rightarrow \text{op} \]

\[ \text{R} \rightarrow \text{R} \]

crash and recovery execution

\[ \text{op} \rightarrow \text{op_impl} \rightarrow \text{recover} \rightarrow \text{op} \rightarrow \text{recover} \]

\[ \text{R} \rightarrow \text{R} \rightarrow \text{R} \]
Recovery refinement

non-crash execution

\[ \text{op} \rightarrow R \rightarrow \text{op_impl} \rightarrow O \]

crash and recovery execution

\[ \text{op} \rightarrow R \rightarrow \text{op_impl} \rightarrow \text{recover} \rightarrow O \]

\[ O \rightarrow R \rightarrow \text{recover} \rightarrow O \]

![Trace inclusion]

implies

specification behavior \( \supset \)
running code behavior
Composition theorem
Kleene algebra for transition relations
Kleene algebra for transition relations

<table>
<thead>
<tr>
<th>expression</th>
<th>matching transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$o_1o_2$</td>
<td>$o_1 \rightarrow o_2$</td>
</tr>
<tr>
<td>$\text{op}$</td>
<td>$\text{op} \rightarrow \text{op}$</td>
</tr>
<tr>
<td>$r^*$</td>
<td>$r \rightarrow r \rightarrow r \rightarrow \ldots$</td>
</tr>
</tbody>
</table>
Theorem: recovery refinements compose

Transactions

- write-ahead log
- log_recover

Disk interface

- replication
- rep_recover

Two-disk interface
Theorem: recovery refinements compose

If

Transactions

Disk interface

write-ahead log
... log_recover

replication
... rep_recover

Two-disk interface

then

Transactions

logging + replication
... rep_recover; log_recover

Two-disk interface
Goal: prove composed recovery correct

```plaintext
rep_recover ✓ under crashes
log_recover ✓ under crashes

rep_recover ; log_recover ?
```
Goal: prove composed recovery correct
Using Kleene algebra for reasoning

\[(\text{rep} \mid \text{rep} \log \text{rep})^* \text{rep} \log\]

after de-nesting \((p \mid q)^* = p^*(qp^*)^*\)
Using Kleene algebra for reasoning

\[(\text{rep} \cdot \text{rep} \log \text{rep})^* \text{rep} \text{log}\]

After de-nesting \((p \mid q)^* = p^*(qp^*)^*\)

\[= \text{rep} \cdot \text{rep} \log \text{rep} \cdot \text{rep} \cdot \text{rep} \cdot \text{rep}^* \cdot \text{rep} \cdot \text{log}\]

After sliding \((pq)^* p = p(qp)^*\)

\[= \text{rep} \cdot \text{rep} \cdot \text{rep} \cdot \text{log} \cdot \text{rep} \cdot \text{rep}^* \cdot \text{rep}^* \text{log}\]
After rewrite both proofs apply

rep invariants restored

replication proof
After rewrite both proofs apply

rep invariants restored

behaves like

replication proof
After rewrite both proofs apply

rep invariants restored

behaves like

replication proof

write-ahead log proof

log invariants restored
Kleene algebra also helps with refinement
Kleene algebra also helps with refinement
An anecdote about modularity
An anecdote about modularity

Using Crash Hoare Logic for Certifying the FSCQ File System
Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich
MIT CSAIL

Abstract
FSCQ is the first file system with crash-safety guarantees (using the Coq proof assistant) whose specification and whose implementation provably avoids bugs.

Verifying a high-performance crash-safe file system using a tree specification
Haogang Chen,‡ Tej Chajed, Alex Konradi,‡ Stephanie Wang,§ Atalay İleri, Adam Chlipala, M. Frans Kaashoek, Nickolai Zeldovich
MIT CSAIL

ABSTRACT
DFSCQ is the first file system that (1) provides a precise specification for fsync and fdatasync, which allow applications to achieve high performance and crash safety; and (2) provides a machine-checked proof that its implementation meets this specification. DFSCQ's specification captures the behavior of sophisticated optimizations, including locking.

1 INTRODUCTION
File systems achieve high I/O performance and crash safety by implementing sophisticated optimizations to increase disk throughput. These optimizations include deferring writing buffered data to persistent storage, grouping many transactions into a single I/O operation, checksumming journal entries, and bypassing the write-ahead log when writing to persistent storage.
An anecdote about modularity

Using Crash Hoare Logic for Certifying the FSCQ File System
Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich
MIT CSAIL

Abstract
FSCQ is the first file system that uses a tree specification and whose FSCQ provably avoids bugs in
systems, such as performing barriers or forgetting to zero
happens at an inopportune time.

Verifying a high-performance crash-safe file system
using a tree specification
(my advisor)
Haogang Chen, Tej Chajed, Alex Konradi, Stephanie Wang, Atalay İleri,
Adam Chlipala, M. Frans Kaashoek, Nickolai Zeldovich
MIT CSAIL

ABSTRACT
DFSCQ is the first file system that (1) provides a precise
specification for fsync and fsasync, which allow applic-
ations to achieve high performance and crash safety, and
(2) provides a machine-checked proof that its implementa-
tion meets this specification. DFSCQ's specification captures
the behavior of sophisticated optimizations, including lock-

1 INTRODUCTION
File systems achieve high I/O performance and crash safety
by implementing sophisticated optimizations to increase disk
throughput. These optimizations include deferring writing
buffered data to persistent storage, grouping many trans-
actions into a single I/O operation, checksumming journal
entries, and bypassing the write-ahead log when writing to

35
def atomic_save(data, path):
    write_all(data, tmp)
    rename(tmp, path)

    # runs on crash
    def recover():
        fs_recover()
        unlink(tmp)
```python
def atomic_save(data, path):
    write_all(data, tmp)
    rename(tmp, path)

# runs on crash
def recover():
    fs_recover()
    unlink(tmp)
```

This is non-modular and makes the proof much harder

Proving this code correct took 1500 lines of proof code!
Argosy so far

Recovery refinement implies Trace inclusion for recovery

implies (composition theorem)

Modular proofs of multiple layers
Argosy so far

program → Recovery refinement

implies

(implies (composition theorem)

Trace inclusion for recovery

Modular proofs of multiple layers
Crash Hoare Logic
Hoare Logic

"Hoare triple" {P} code {Q}

precondition postcondition
Hoare Logic

"Hoare triple"  
\{P\} code \{Q\}

precondition  
postcondition

\[ s \rightarrow \text{code} \rightarrow s' \]

\text{if } P(s) \text{ then } Q(s') \]
Crash Hoare Logic

"crash specification"

\{P\} code \{Q\} \{Qc\}

precondition  postcondition  crash invariant

\[
\begin{align*}
\text{s} & \rightarrow \text{code} \rightarrow \text{s}' \\
: = & \quad \text{if } P(s) \text{ then } Q(s') \\
\text{s} & \rightarrow \text{code} \rightarrow \text{s}' \\
& \quad \text{if } P(s) \text{ then } Qc(s')
\end{align*}
\]
Crash Hoare Logic

"recovery specification" \{P\} code \(\circ\) recover \{Q\} \{Q_r\}

precondition                        postcondition         recovery postcondition

s \rightarrow \text{code} \rightarrow s' \quad \text{if } P(s) \text{ then } Q(s')

:=

s \rightarrow \text{code} \text{ recover} \text{ recover} \rightarrow s' \quad \text{if } P(s) \text{ then } Q_r(s')
Crash Hoare Logic

```
{P} code {Q} {Qc}
```

- **precondition**: $P(s)$
- **postcondition**: $Q(s')$
- **crash invariant**: $Qc(s')$

```
\[ s \rightarrow \text{code} \rightarrow s' \]
\[ := \text{if } P(s) \text{ then } Q(s') \]
```

```
\[ s \rightarrow \text{code} \rightarrow \text{power} \rightarrow s' \]
\[ \text{if } P(s) \text{ then } Qc(s') \]
```
Crash Hoare Logic

"recovery specification" \{P\} code ∩ recover \{Q\} \{Q_r\}

precondition          postcondition          recovery postcondition

\[ s \rightarrow \text{code} \rightarrow s' \]

if \( P(s) \) then \( Q(s') \)

:=

\[ s \rightarrow \text{code} \rightarrow \text{recover} \rightarrow \text{recover} \rightarrow s' \]

if \( P(s) \) then \( Q_r(s') \)
{pre} code \odot recover {post} {post rec}
theorem in CHL to prove recovery specs from crash specs

code ⨯ recover

pre

post

post rec
theorem in CHL to prove recovery specs from crash specs

code \circ recover

pre \rightarrow pre \rightarrow code \rightarrow crash \rightarrow post \rightarrow post

post rec
Theorem in CHL to prove recovery specs from crash specs

code ⋆ recover

pre → pre → code → post → post

recover

inv → crash → post rec → post rec

recovery is idempotent
A theorem in CHL to prove recovery specs from crash specs.

Diagram:
- `pre` to `pre`
- `pre` to `code`
- `code` to `post`
- `post` to `post`
- `crash` to `inv`
- `inv` to `recover`
- `recover` to `post rec`
- `post rec` to `post rec`

Recovery is idempotent.
Argosy connects CHL to recovery refinement

Come up with *abstraction relation*

Prove a *refinement specification* for every operation

Gives recovery refinement for implementation
Recovery refinement as a CHL spec

\{P\} op_impl ⪆ recover \{Q\} \{Qr\}
Recovery refinement as a CHL spec

\{ P: \exists R \} \; \{ Q: \exists R \} \; \{ Q_r: \exists R \}

\text{op_impl} \; \ominus \; \text{recover}

\text{op}

\text{op_impl}

\text{op} \; \text{recover} \; \text{recover}
Recovery refinement as a CHL spec

\[
\{ P: \quad R \} \\

\text{op_impl} \circ \text{recover}
\]

\[
\{ Q: \quad R \} \\
\text{op_impl}
\]

\[
\{ Qr: \quad R \} \\
\text{op_impl} \circ \text{recover} \circ \text{recover}
\]
program \rightarrow CHL \rightarrow \text{Recovery refinement} \rightarrow \text{Trace inclusion for recovery}

\text{implies}

\rightarrow \text{Modular proofs of multiple layers}

\text{implies (composition theorem)}
Argosy is implemented and verified in Coq

3,200 lines for framework

4,000 lines for verified example (logging + replication)

Example extracts to Haskell and runs

github.com/mit-pdos/argosy
Future work
Concurrency

Extending Concurrent Separation Logic [originally 2007]

Implemented using Iris [originally POPL 2015]
Better story for running code

Currently extract to Haskell

Performance problems (esp. for concurrency)

New plan: import Go into Coq
Usability for students

Argosy spun off from course infrastructure

Now want to backport improvements
Argosy: modular proofs of layered storage systems

Kleene algebra

(rep | rep log)*
Argosy: modular proofs of layered storage systems

Kleene algebra

recovery refinement

$$(\text{rep} \lor \text{rep log})^*$$

impl $r$ $r^*$
Argosy: modular proofs of layered storage systems

Kleene algebra

(repl | repl log)*

recovery refinement

impl repl r r

modular proofs
After rewrite both proofs apply

- Rep invariants restored
- Behaves like
- Log invariants restored
- Replication proof
- Write-ahead log proof
Proving correctness with an abstraction relation

1. developer provides abstraction relation R
def atomic_save(data, path):
    write_all(data, tmp)
    rename(tmp, path)

# runs on crash
def recover():
    fs_recover()
    unlink(tmp)