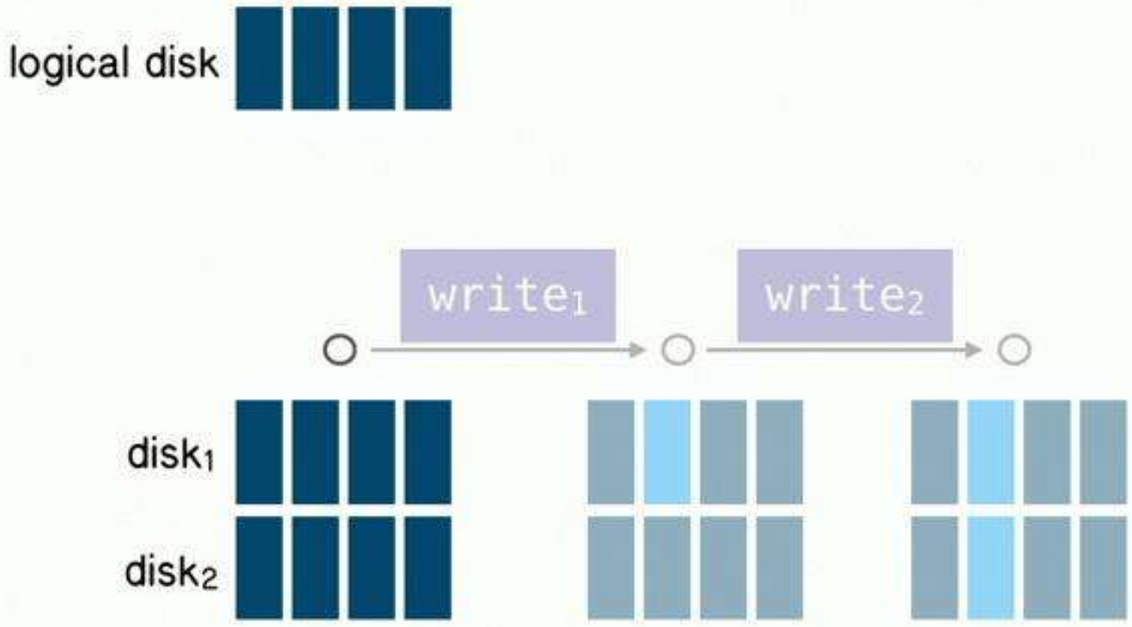
Argosy: Verifying layered storage systems with recovery refinement

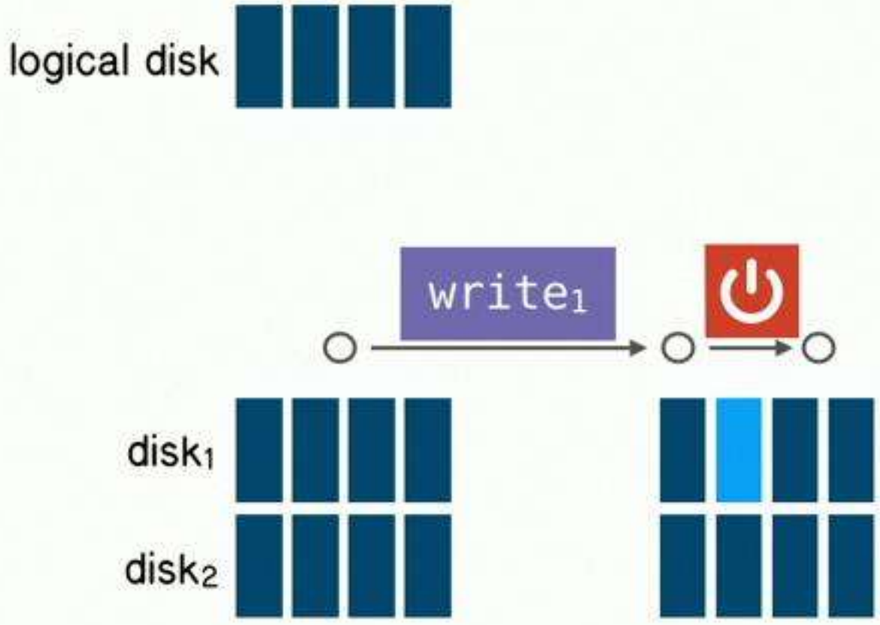
Tej Chajed, Joseph Tassarotti, Frans Kaashoek, Nickolai Zeldovich

MIT

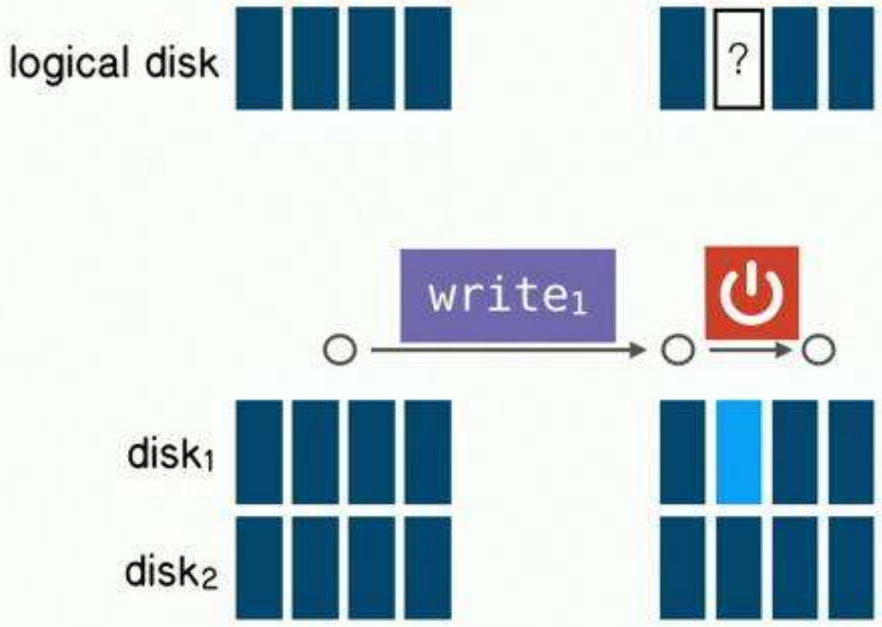
Bob writes a replication system



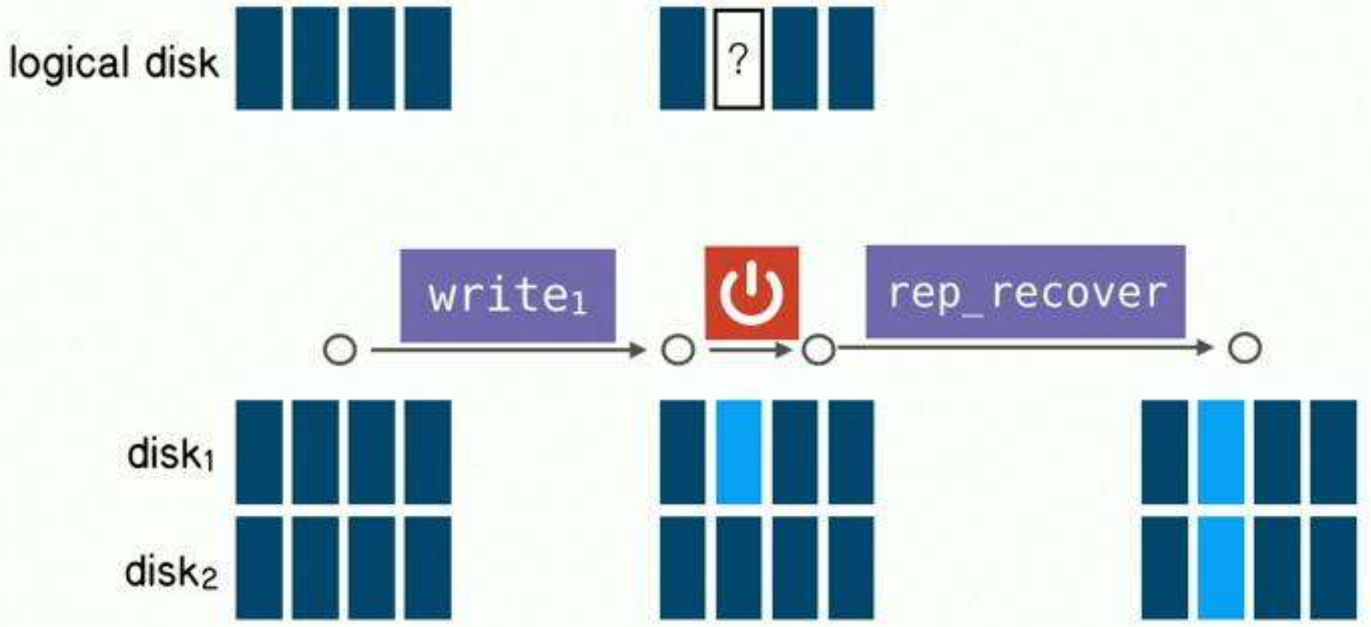
Bob writes a replication system



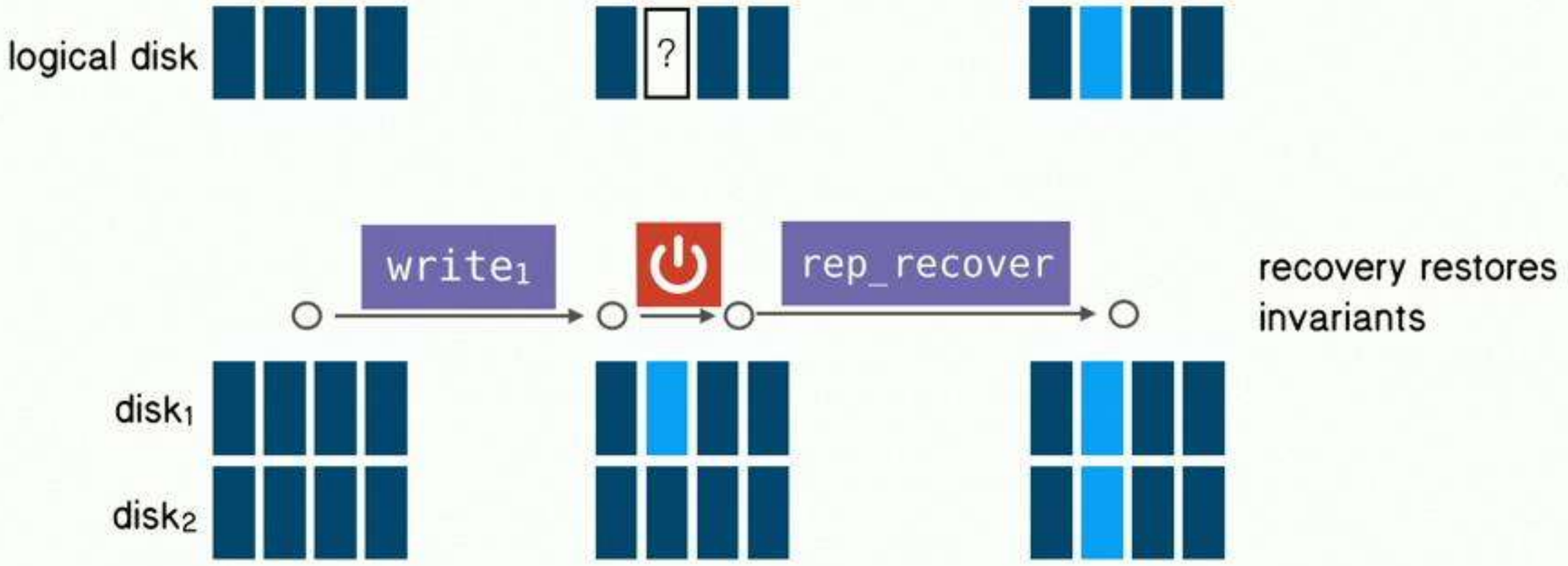
Bob writes a replication system



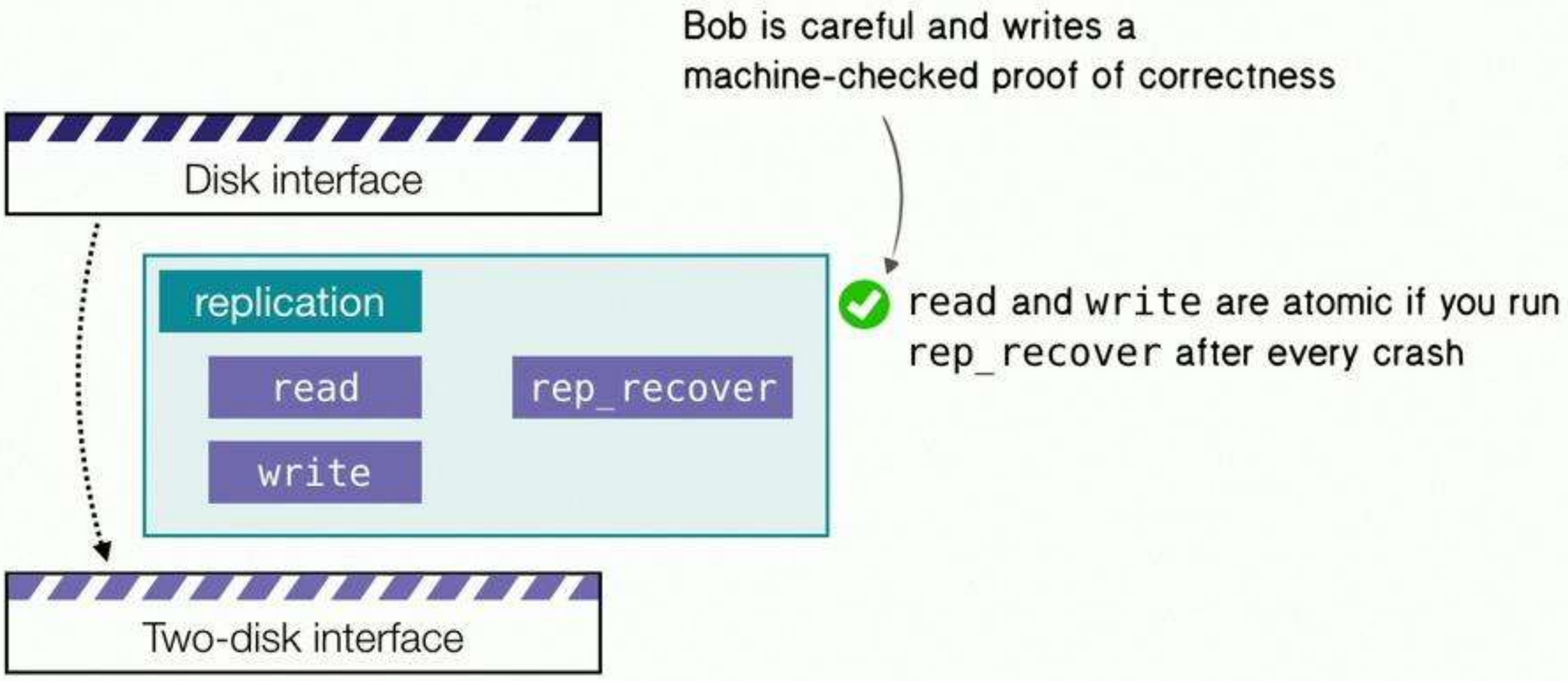
Bob writes a replication system

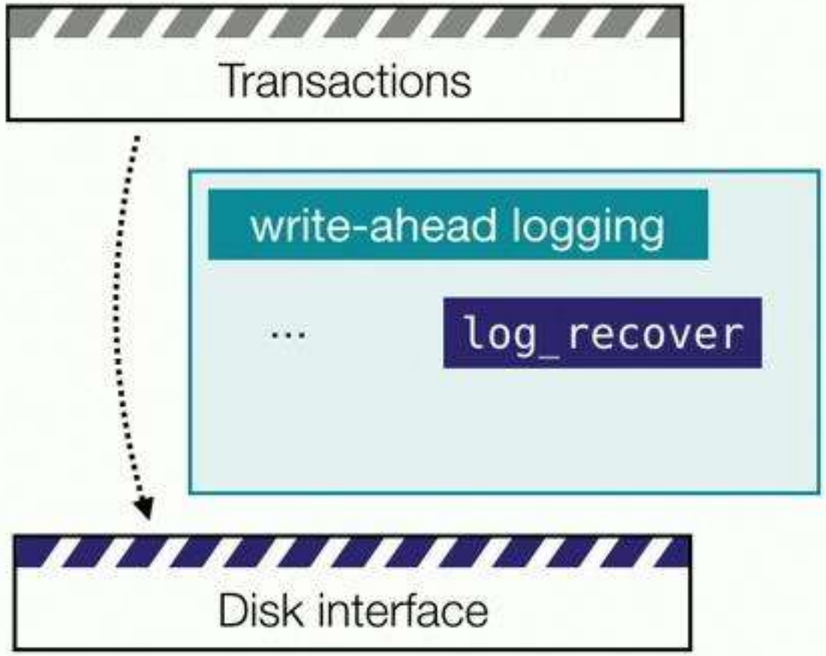


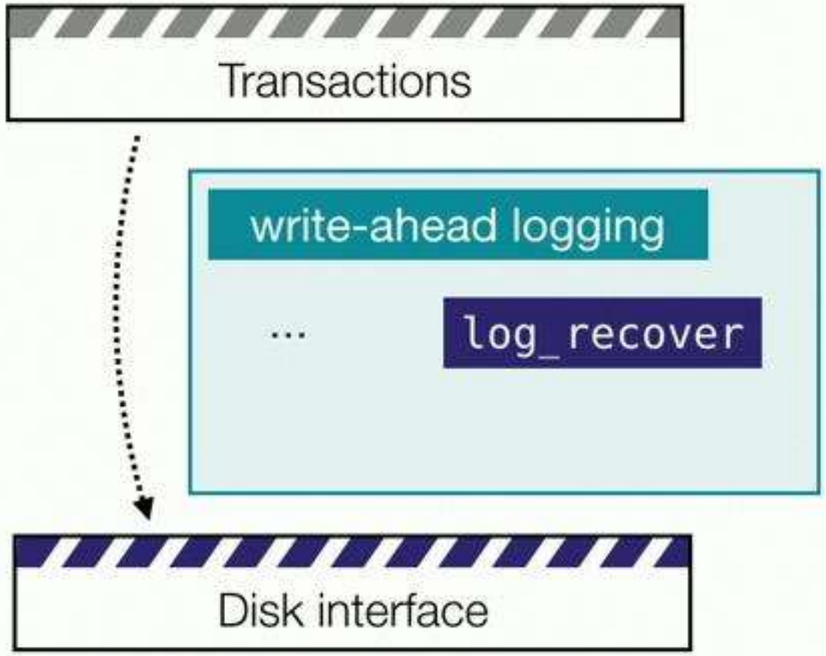
Bob writes a replication system and implements its recovery procedure



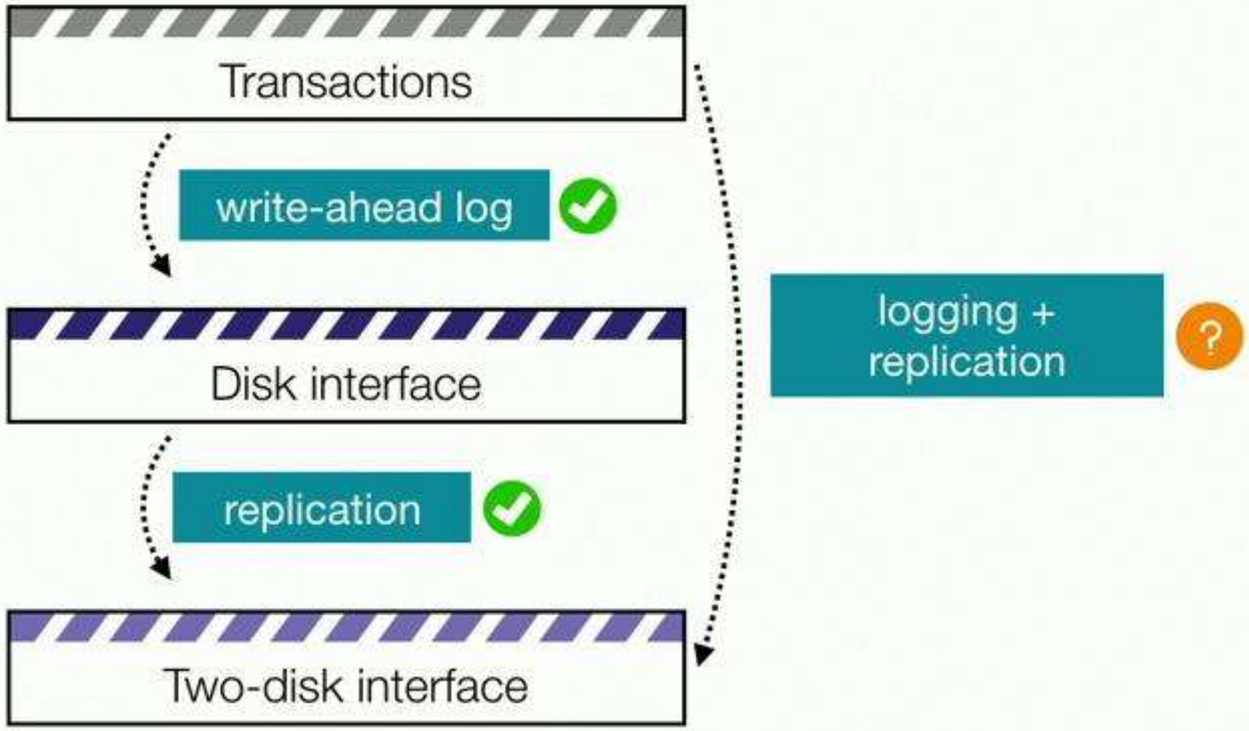
Bob writes a replication system and implements its recovery procedure

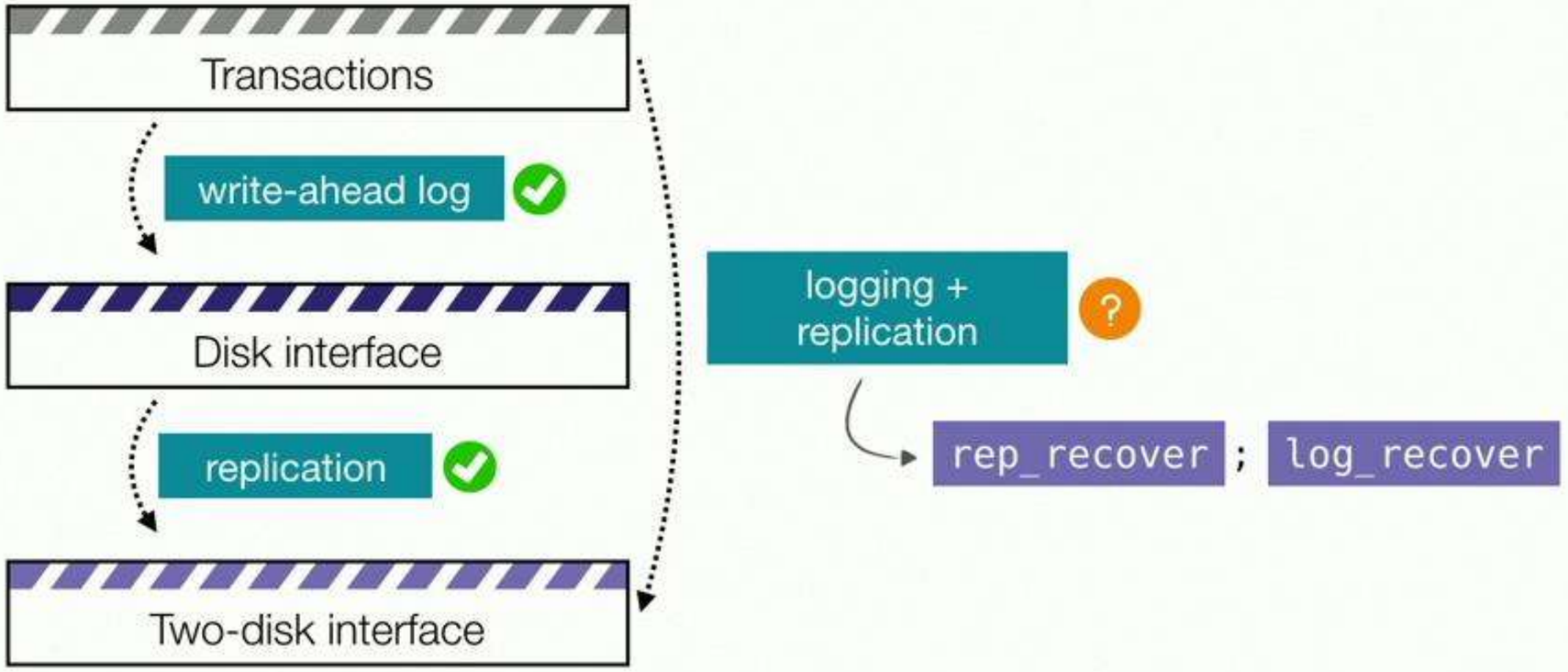

Bob is careful and writes a
machine-checked proof of correctness
Disk interface
replication
read
write
rep_recover
read and write are atomic if you run
rep_recover after every crash
Two-disk interface


Transactions
write-ahead logging
...
log_recover
Disk interface

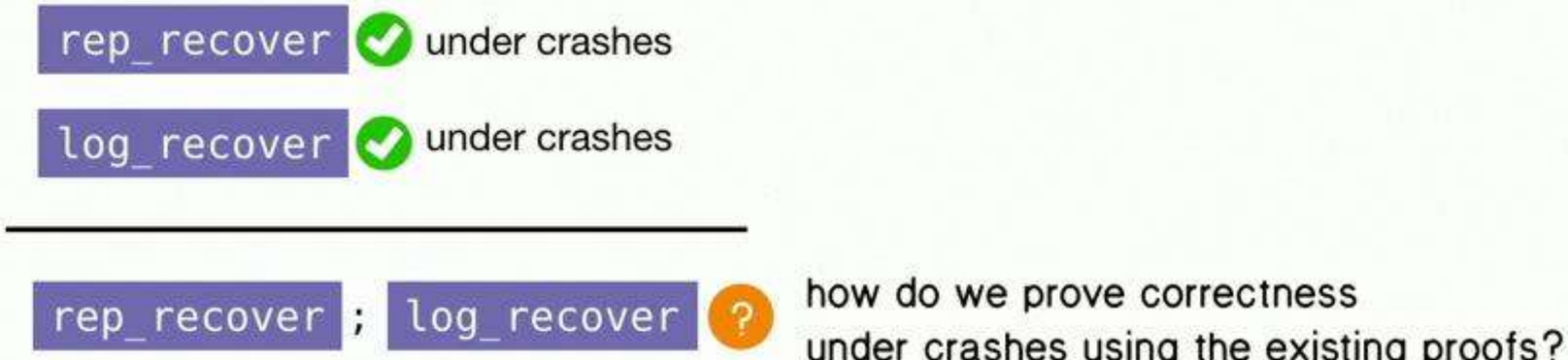


ops are atomic if you run `log_recover` after every crash

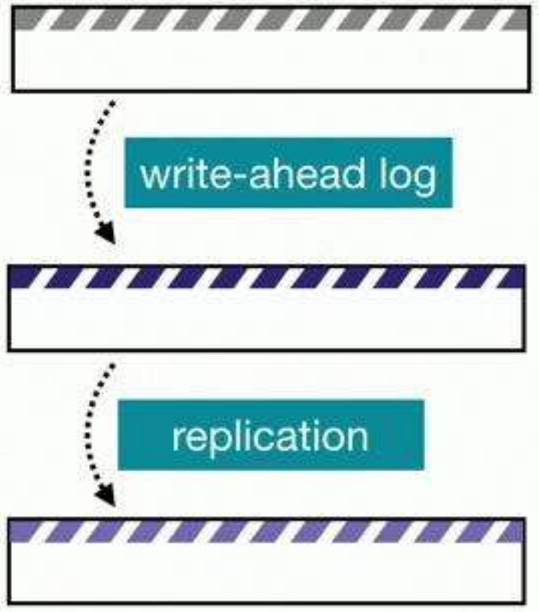

Transactions
write-ahead log
Disk interface
replication
Two-disk interface
logging +
replication
?


Transactions
write-ahead log
Disk interface
replication
Two-disk interface
logging +
replication
?
rep_recover ; log_recover

Challenge: crashes during composed recovery

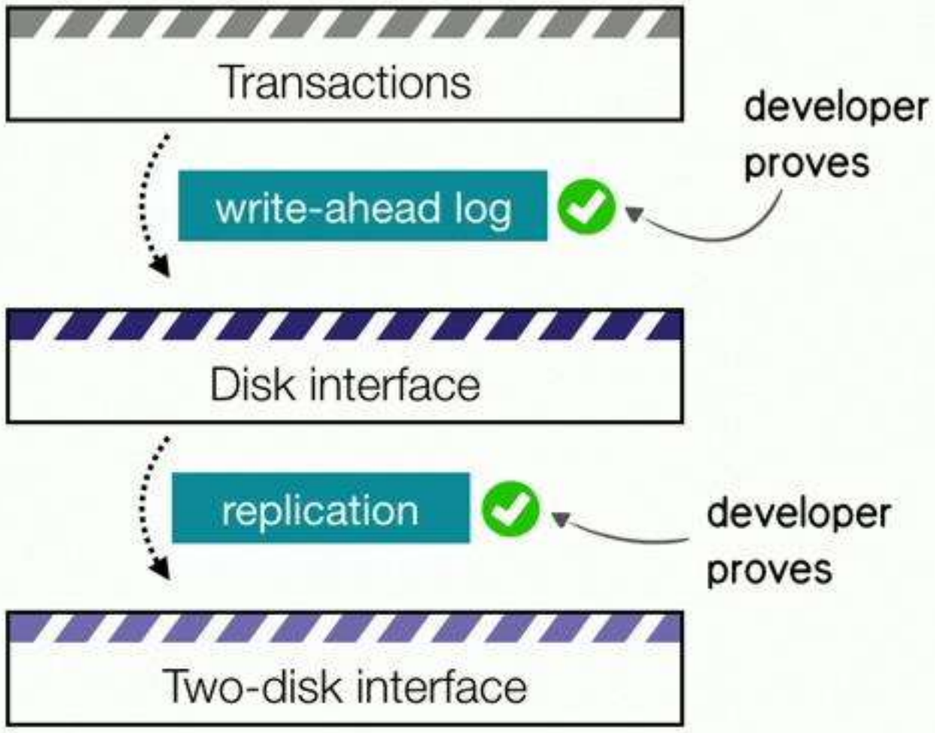


Prior work cannot handle multiple recovery procedures

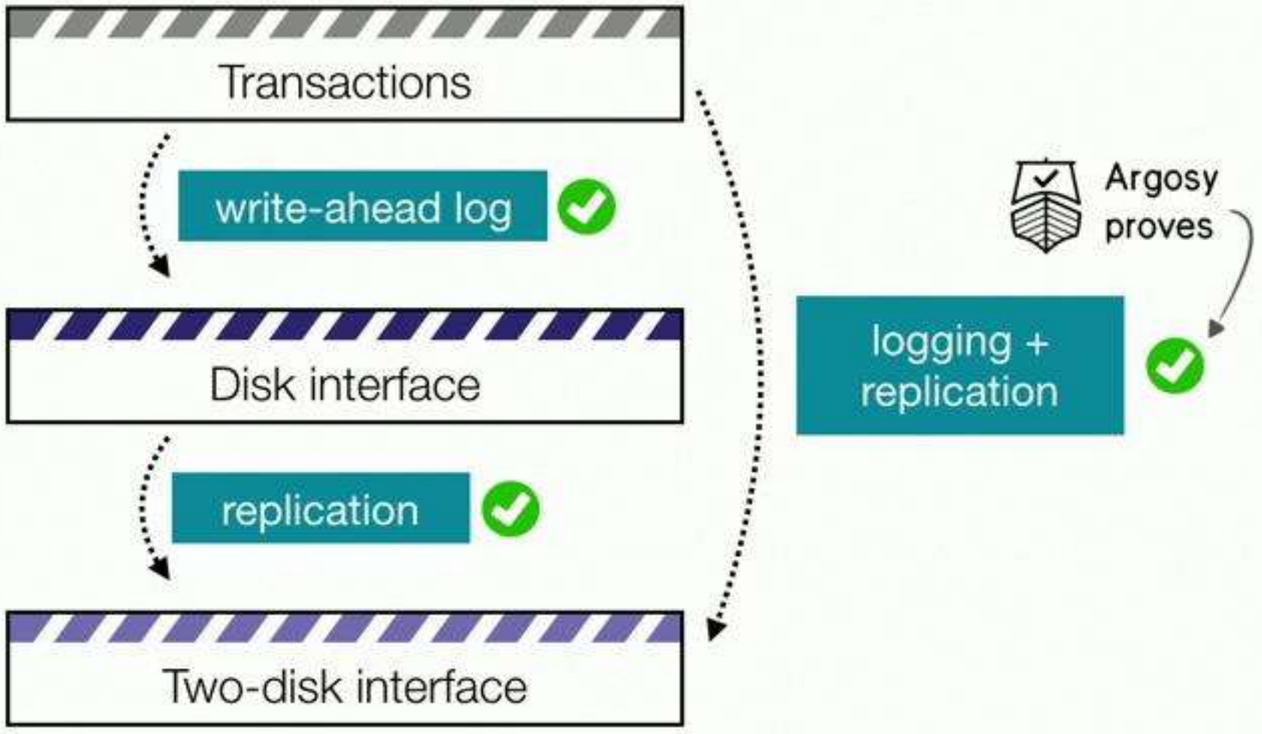


CHL [SOSP '15]	not modular
Yggdrasil [OSDI '16]	single recovery
Flashix [SCP '16]	restricted recovery procedures

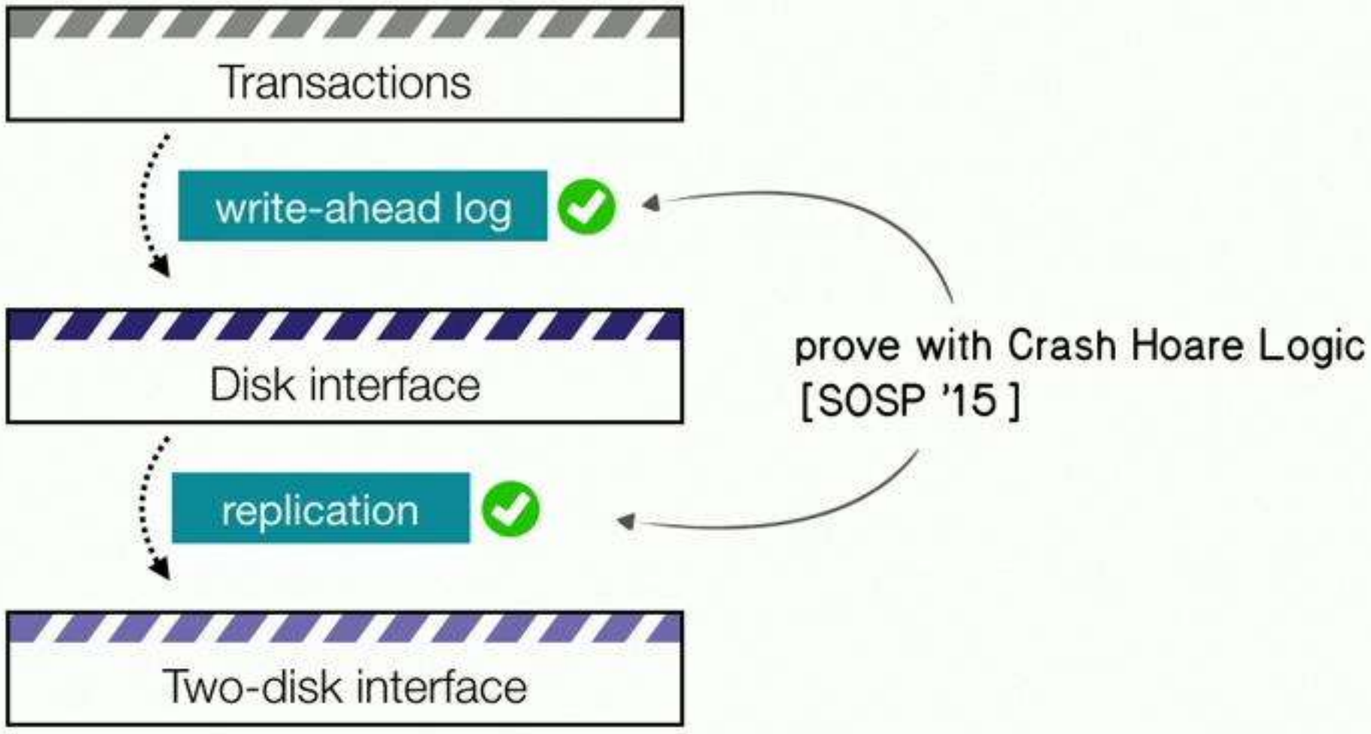
Argosy supports modular recovery proofs



Argosy supports modular recovery proofs



Argosy is compatible with existing techniques



Contributions

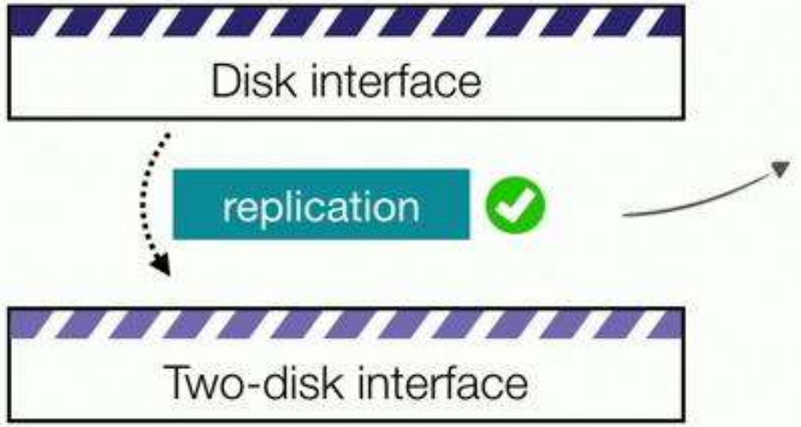
Recovery refinement for modular proofs

CHL for proving recovery refinement

see paper — Verified example: logging + replication

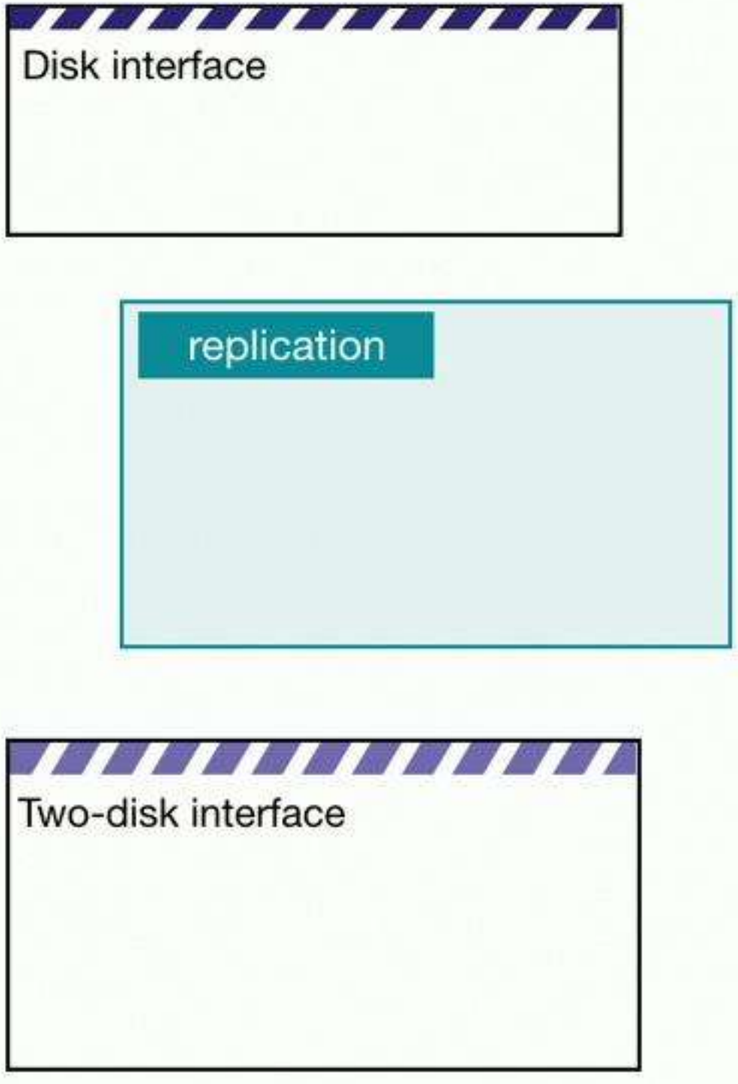
see code — Machine-checked proofs in Coq 

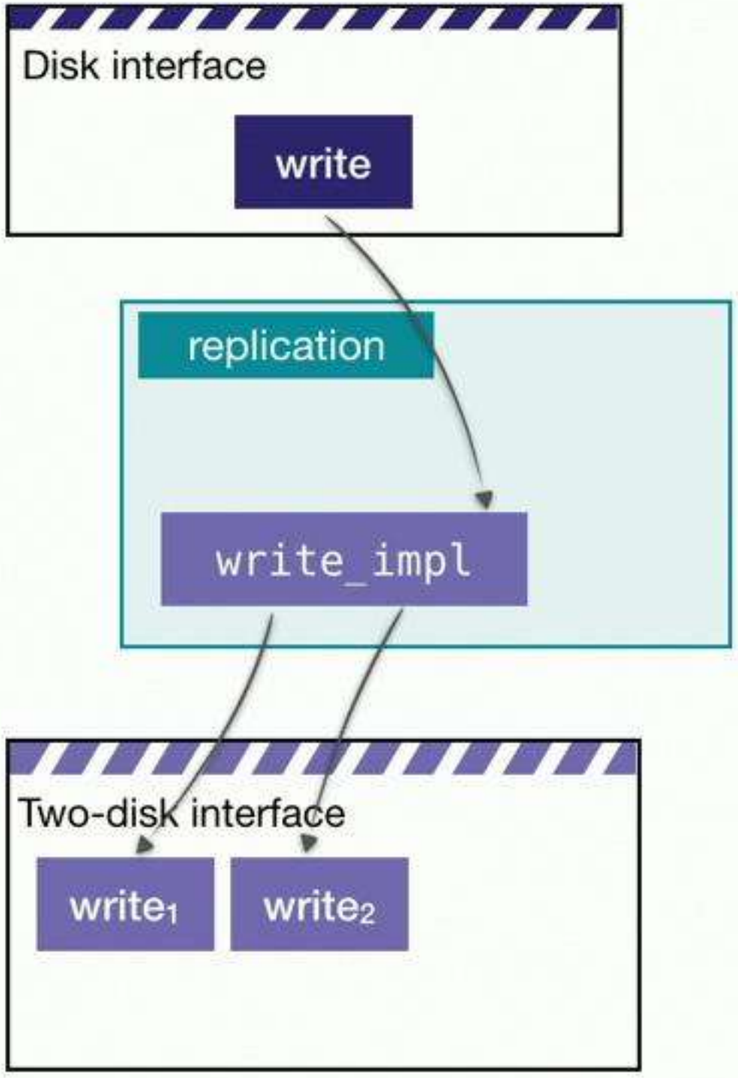
Preview: recovery refinement

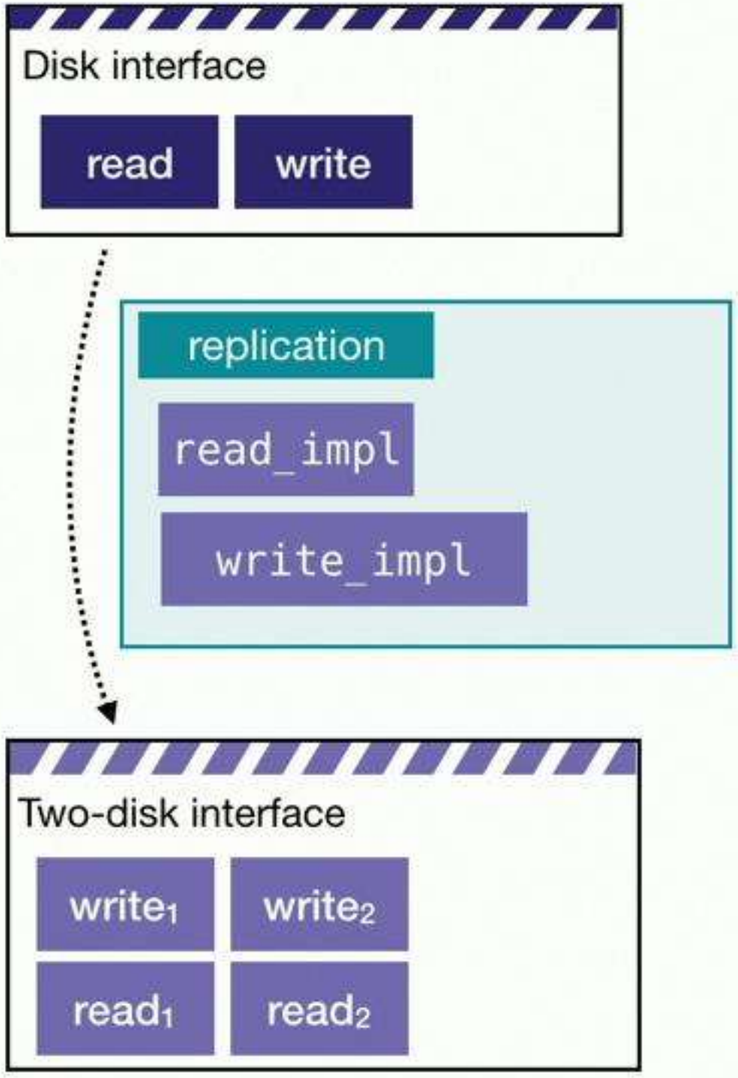


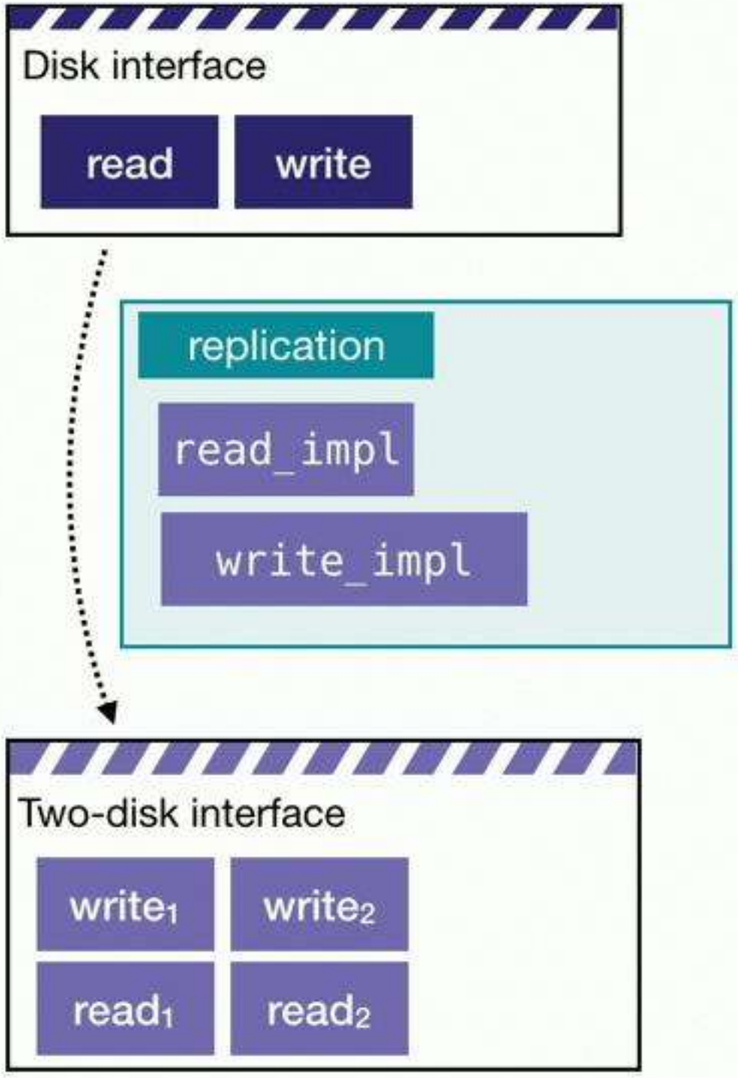
1. Normal execution correctness using *refinement*
2. Crash and recovery correctness using *recovery refinement*

Refinement

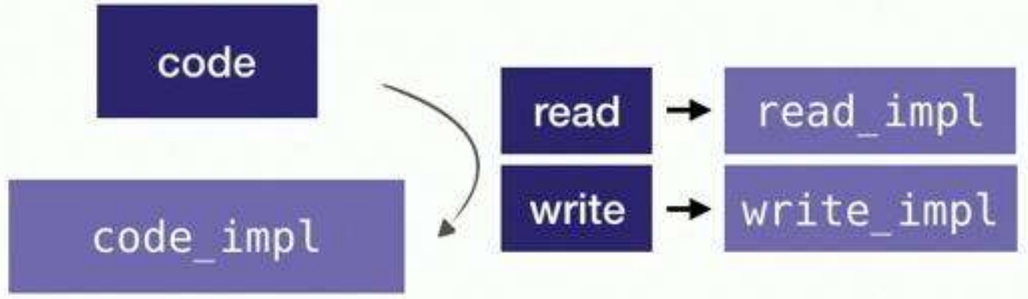

Disk interface
replication
Two-disk interface


Disk interface
write
replication
write_impl
Two-disk interface
write1
write2

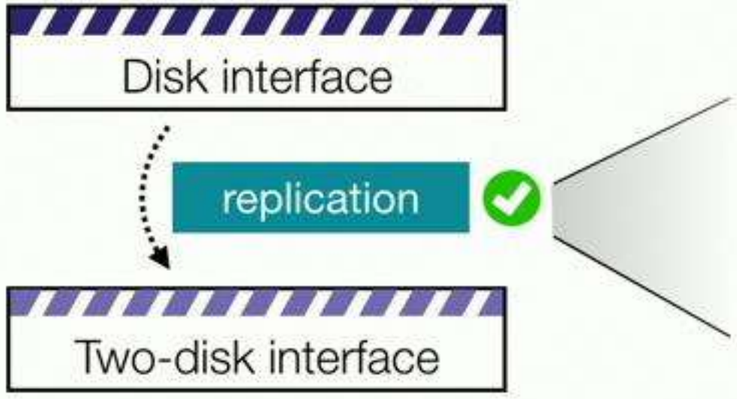

Disk interface
read
write
replication
read_impl
write_impl
Two-disk interface
write1
write2
read1
read2

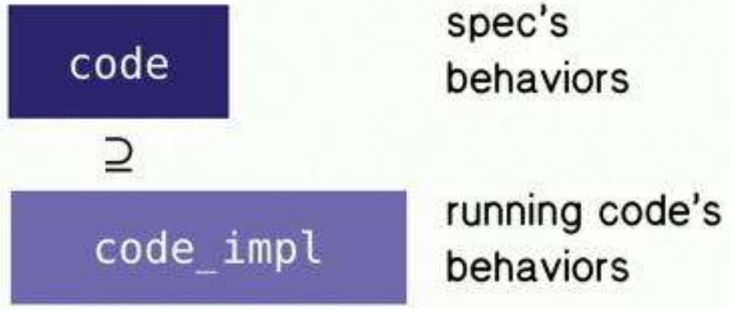


correctness is based on how we use replication:
run code using Disk interface on top of two disks



Correctness: trace inclusion

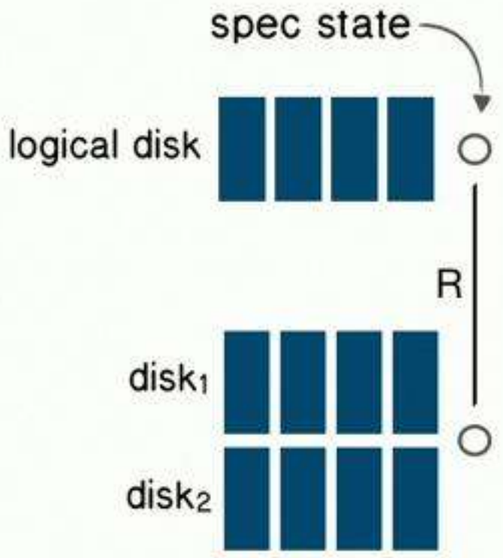




spec's behaviors

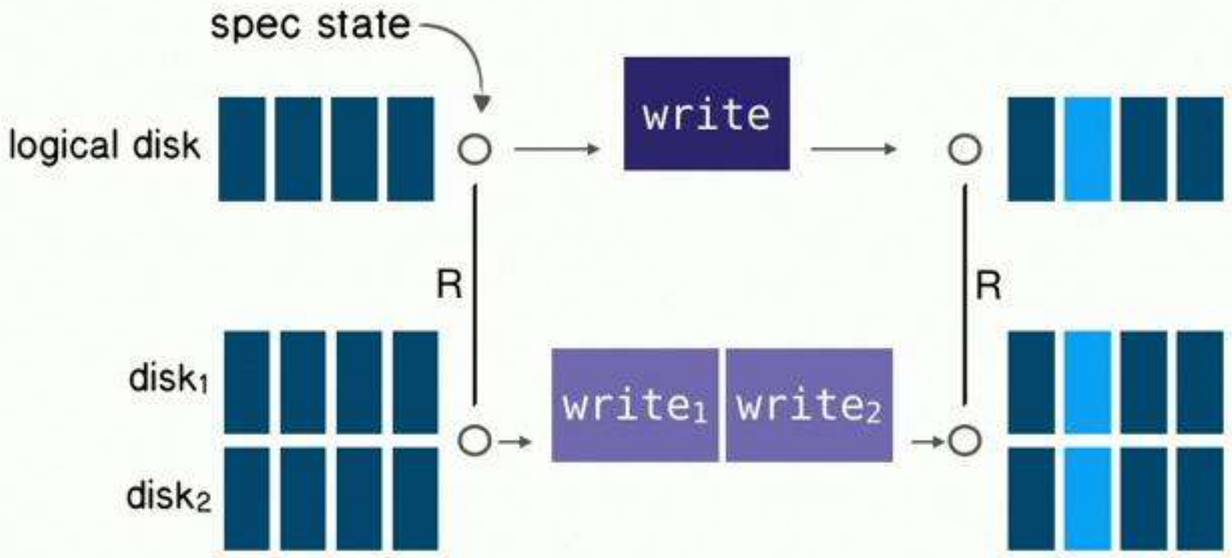
running code's behaviors

Proving correctness with an abstraction relation



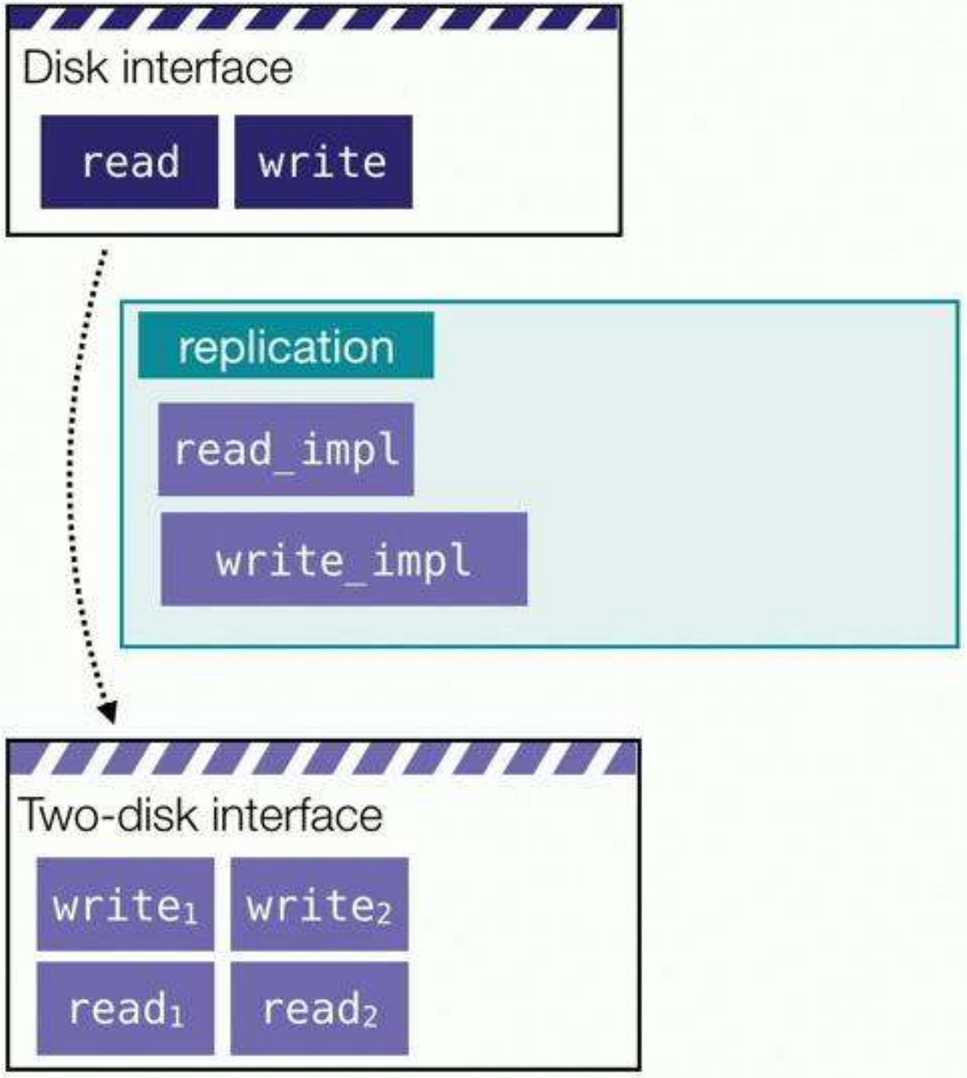
1. developer provides abstraction relation R

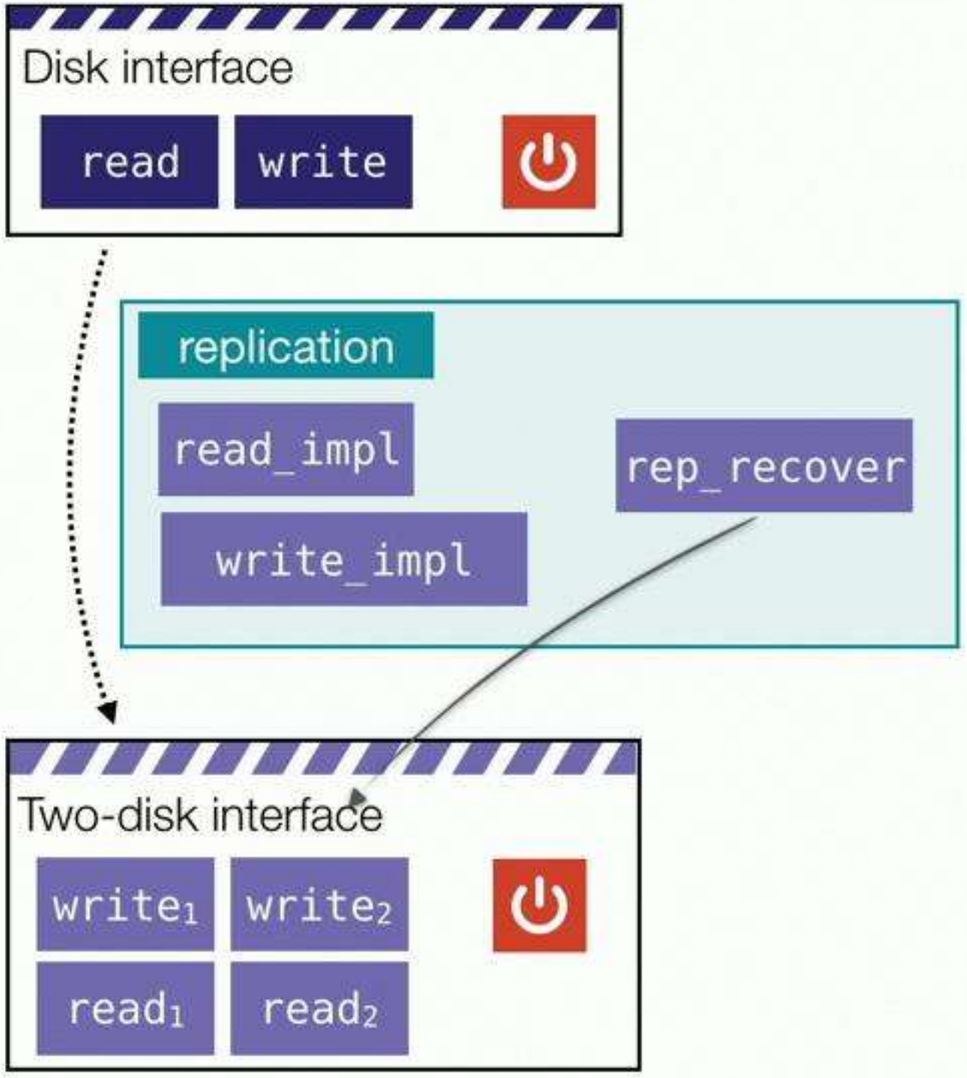
Proving correctness with an abstraction relation

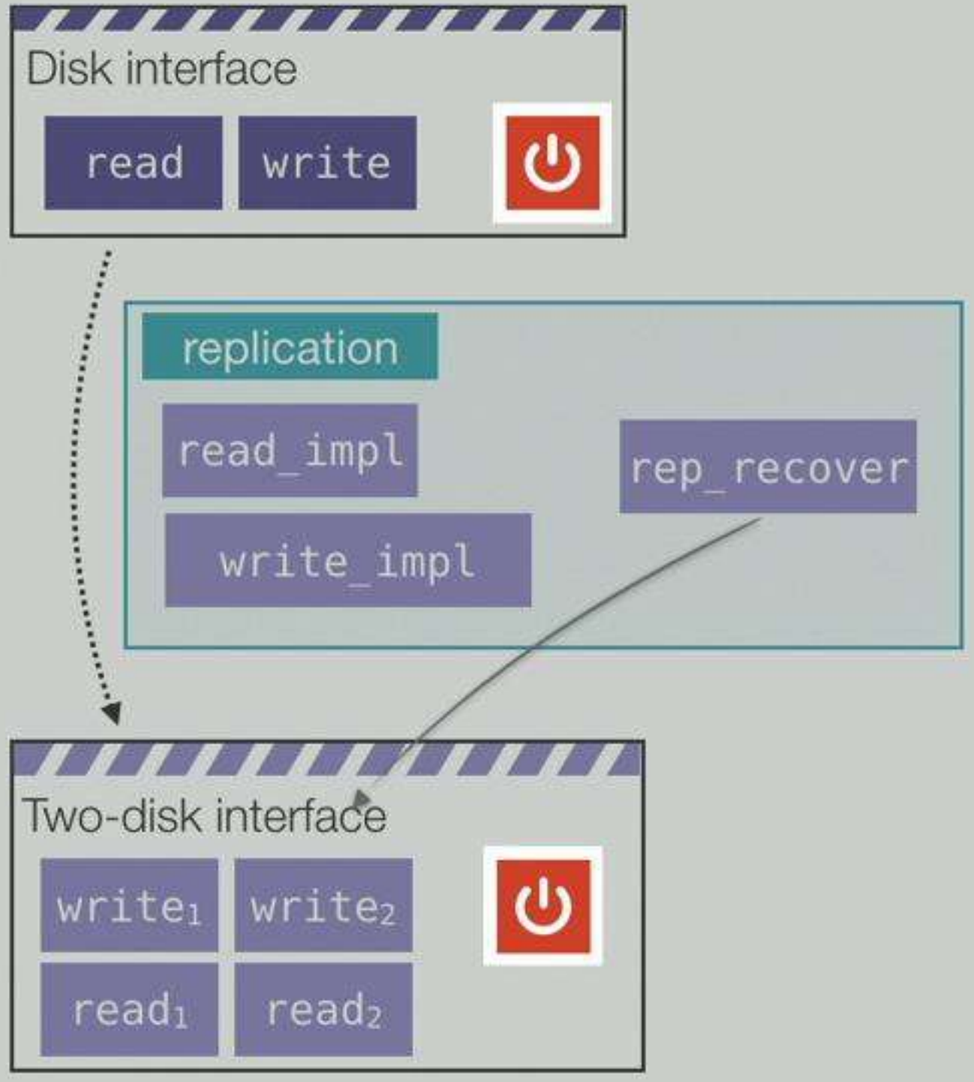


1. developer provides abstraction relation R
2. prove spec execution exists
3. and abstraction relation is preserved

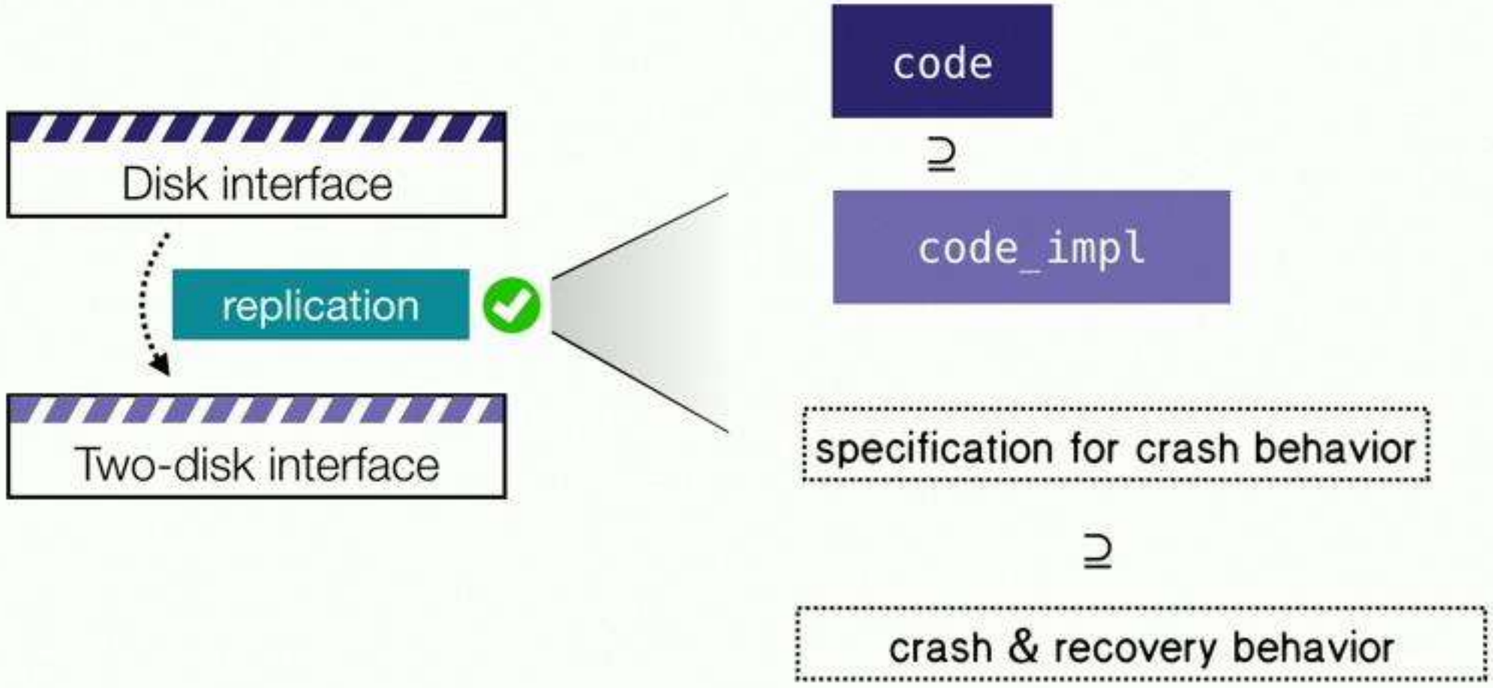
Recovery refinement


Disk interface
read
write
replication
read_impl
write_impl
Two-disk interface
write1
write2
read1
read2

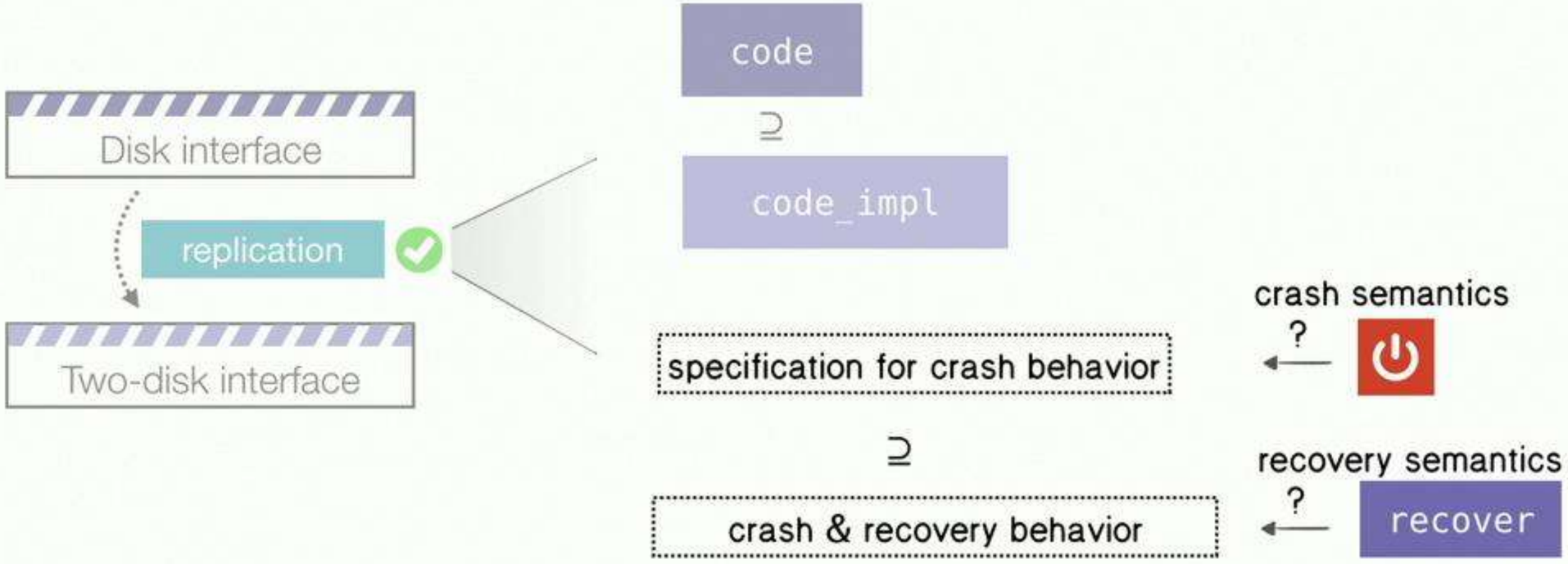

Disk interface
read
write
replication
read_impl
write_impl
rep_recover
Two-disk interface
write1
write2
read1
read2

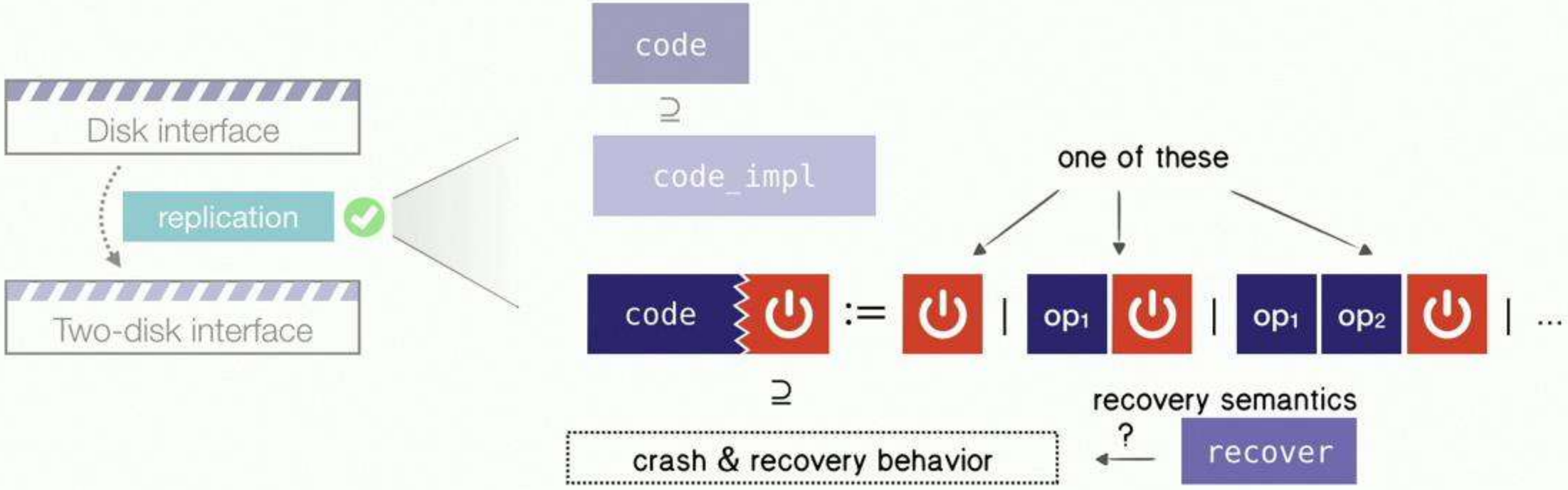

Disk interface
read
write
replication
read_impl
write_impl
rep_recover
Two-disk interface
write1
write2
read1
read2

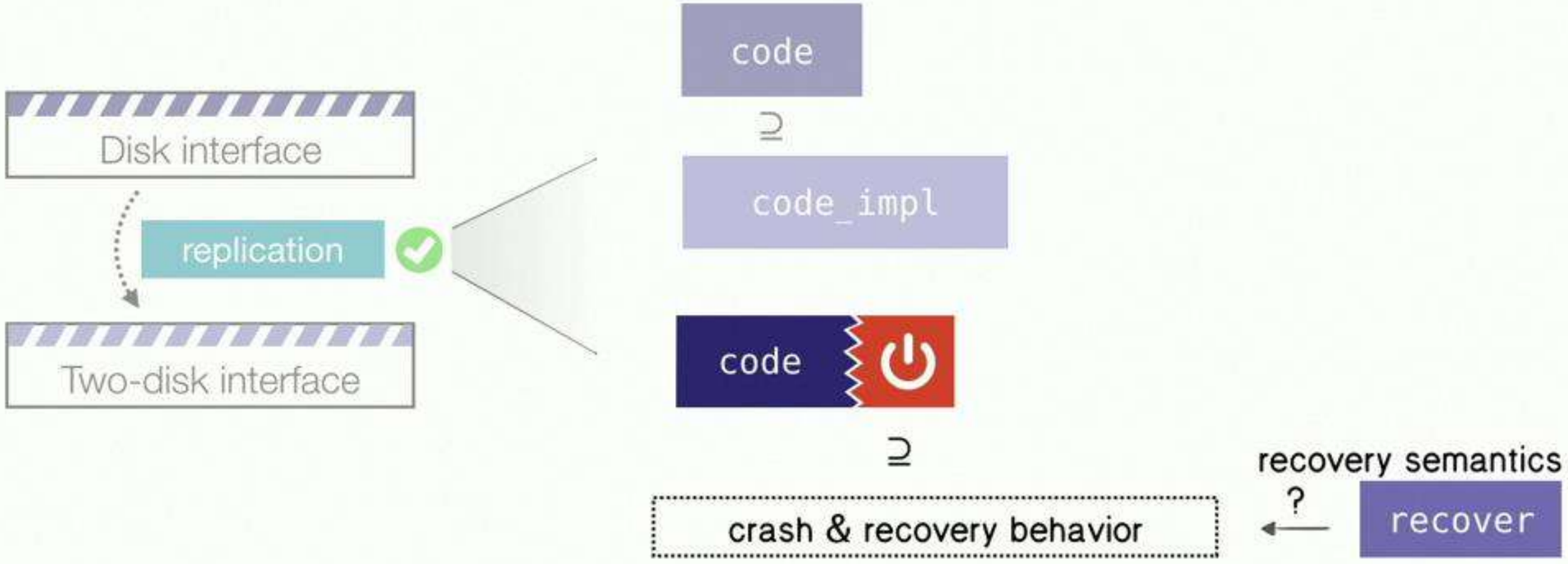
Extending trace inclusion with recovery

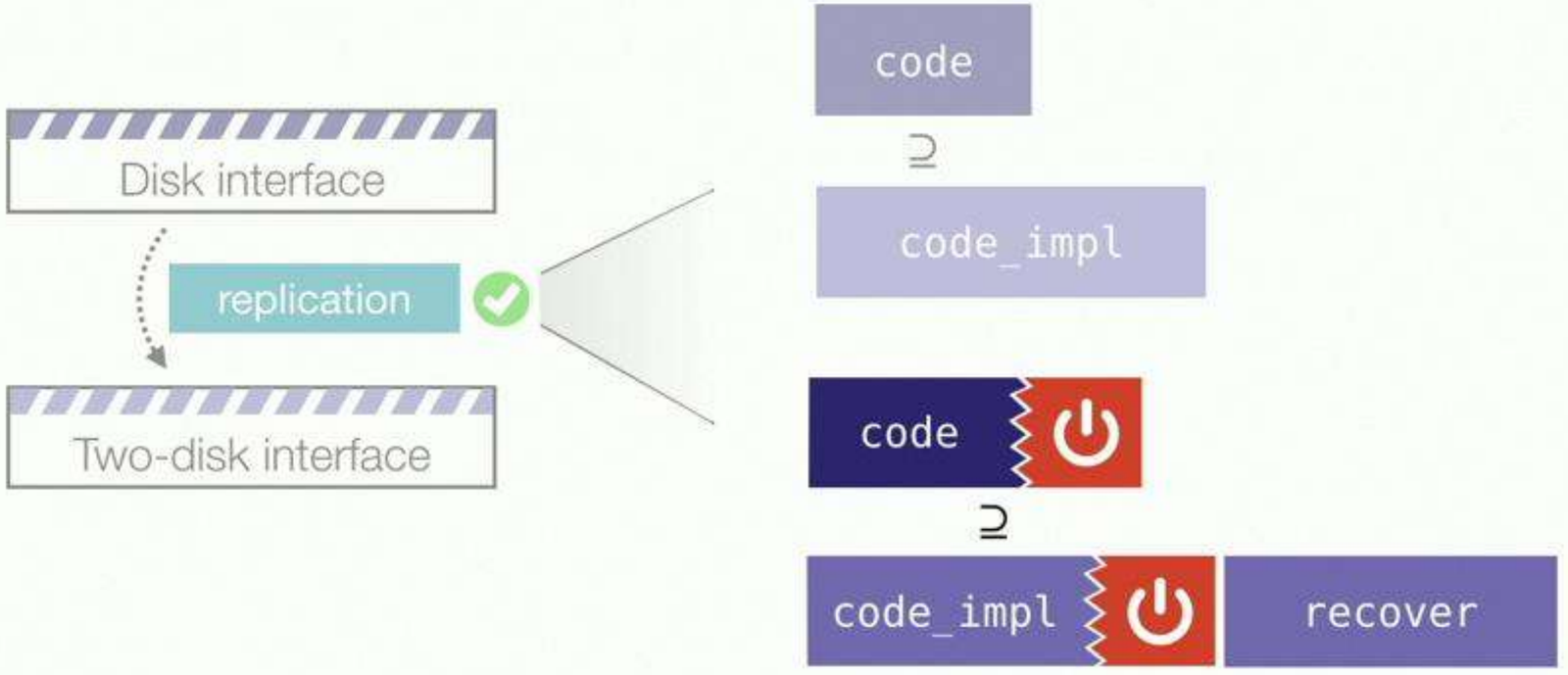


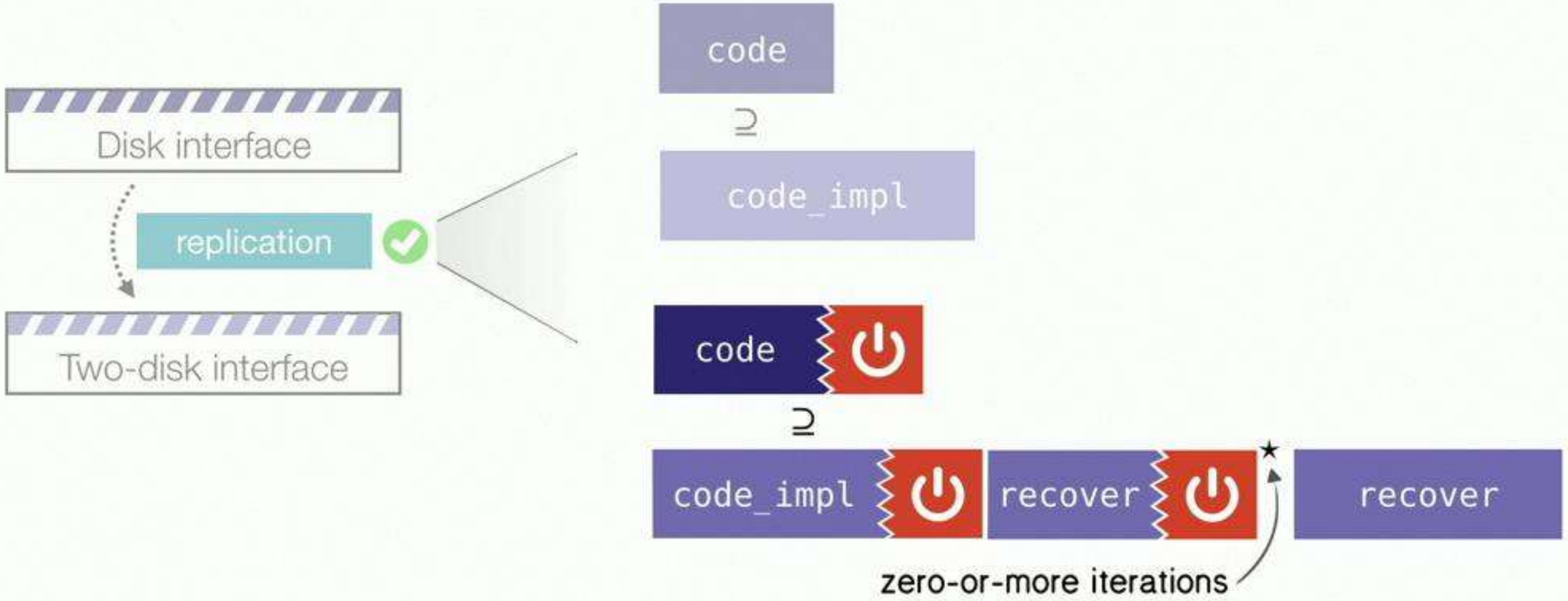
Extending trace inclusion with recovery



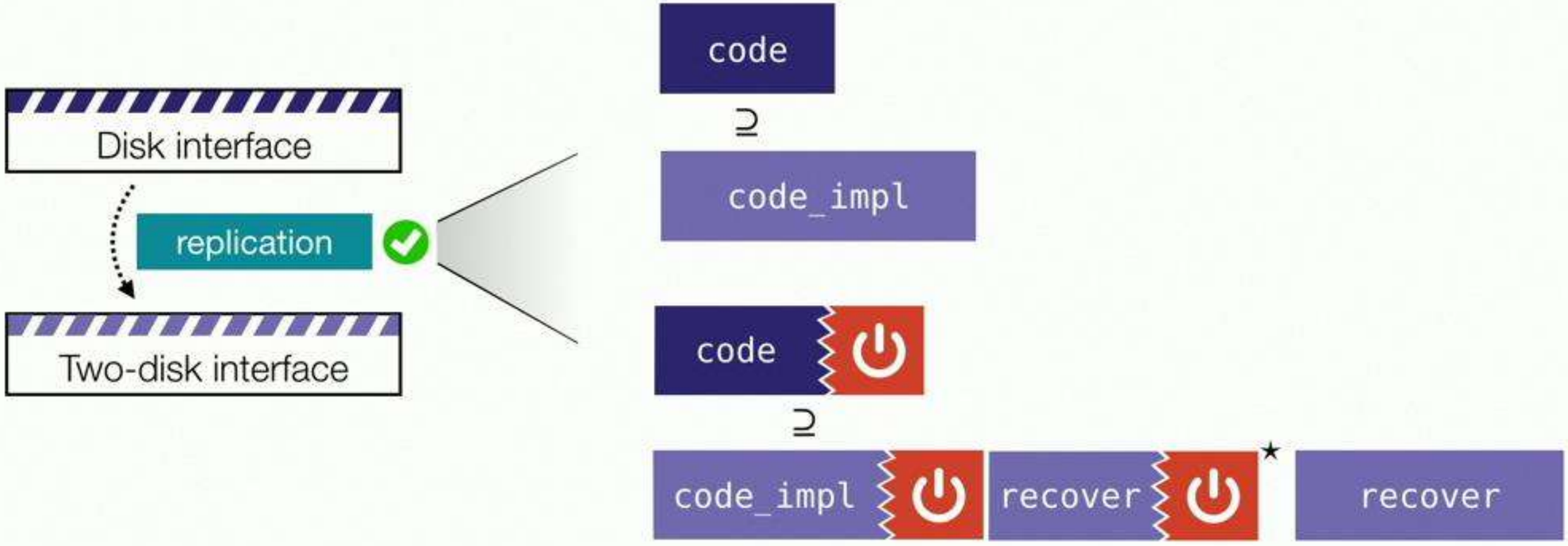

Disk interface
replication
Two-disk interface
code
⊇
code_impl
one of these
code
:=
op1
op1
op2
...
⊇
crash & recovery behavior
recovery semantics
?
recover


Disk interface
replication
Two-disk interface
code
⊇
code_impl
code
⊇
crash & recovery behavior
recovery semantics
?
recover


Disk interface
replication
Two-disk interface
code
⊇
code_impl
code
⊇
code_impl
recover

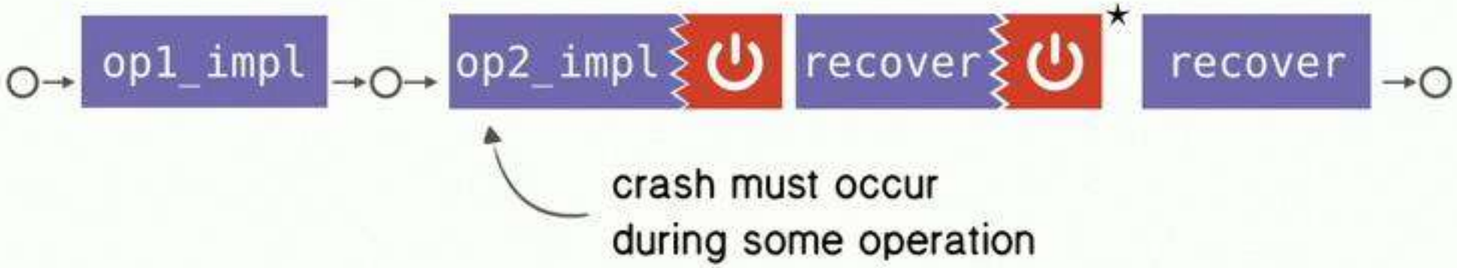

Disk interface
replication
Two-disk interface
code
⊇
code_impl
code
⊇
code_impl
recover
recover
zero-or-more iterations

Trace inclusion, with recovery



Proving trace inclusion, with recovery

Proving trace inclusion, with recovery

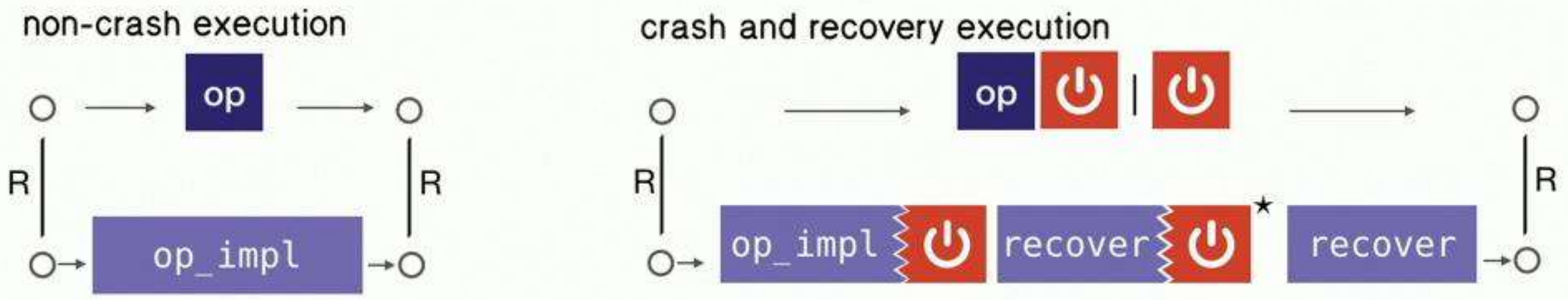


Proving trace inclusion, with recovery

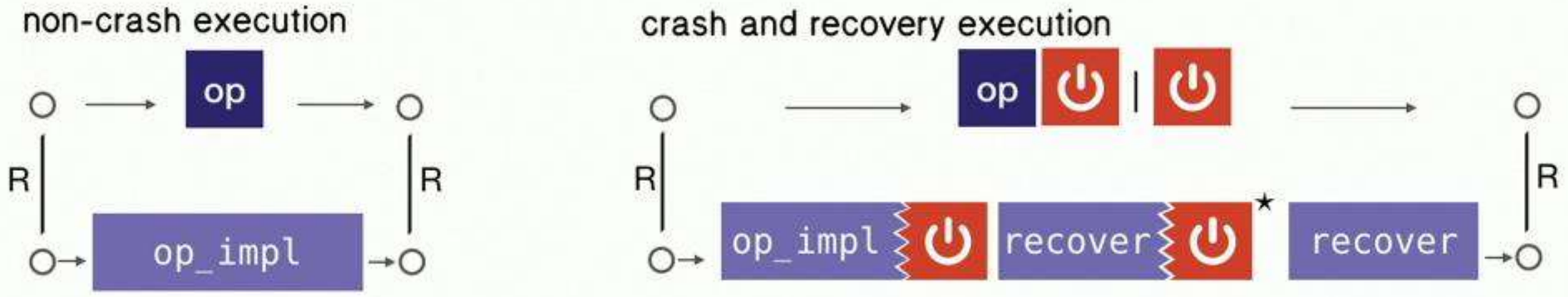
Proving trace inclusion, with recovery

Proving trace inclusion, with recovery

Recovery refinement



Recovery refinement



implies →

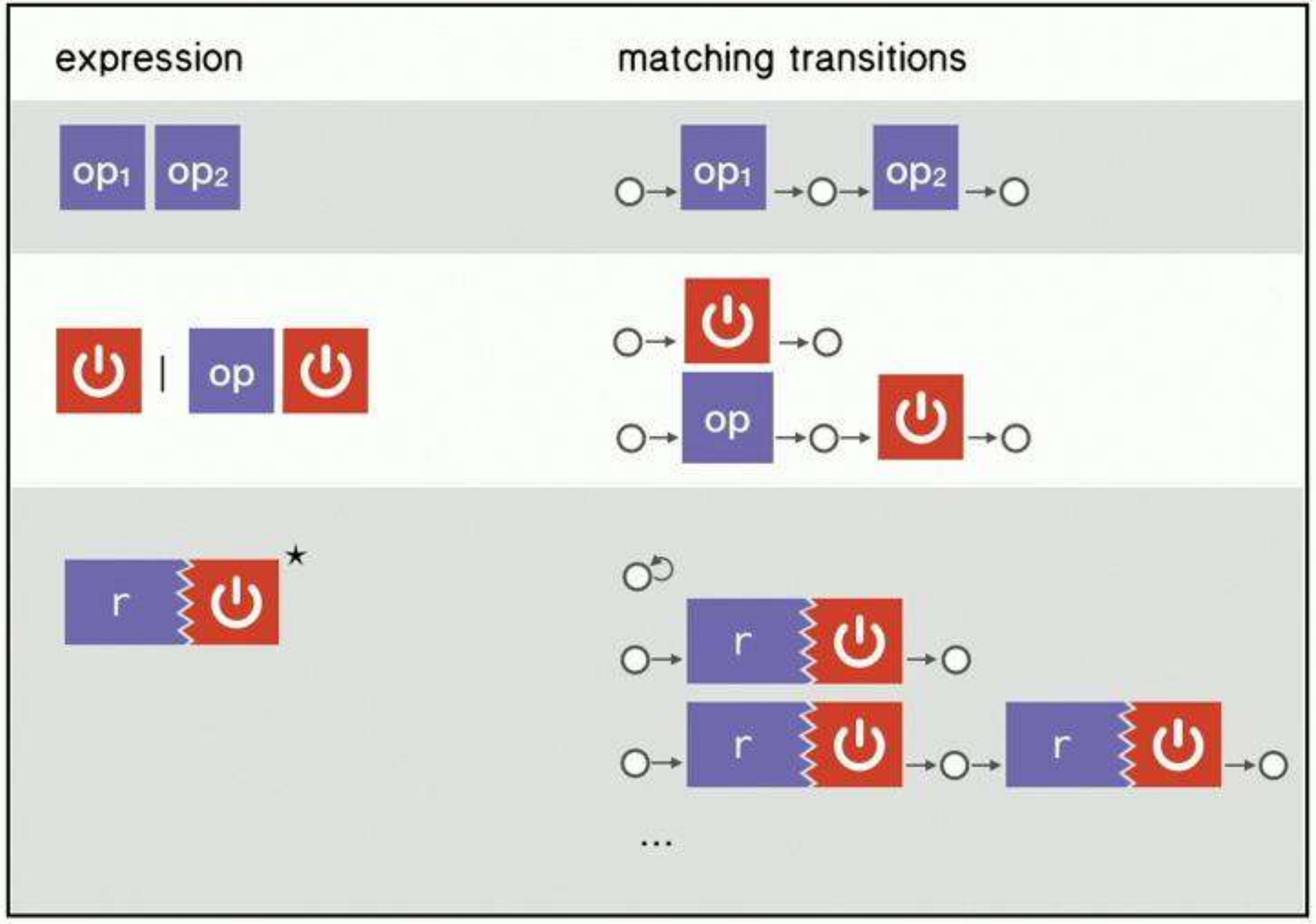
Trace inclusion

specification behavior
⊇
running code behavior

Composition theorem

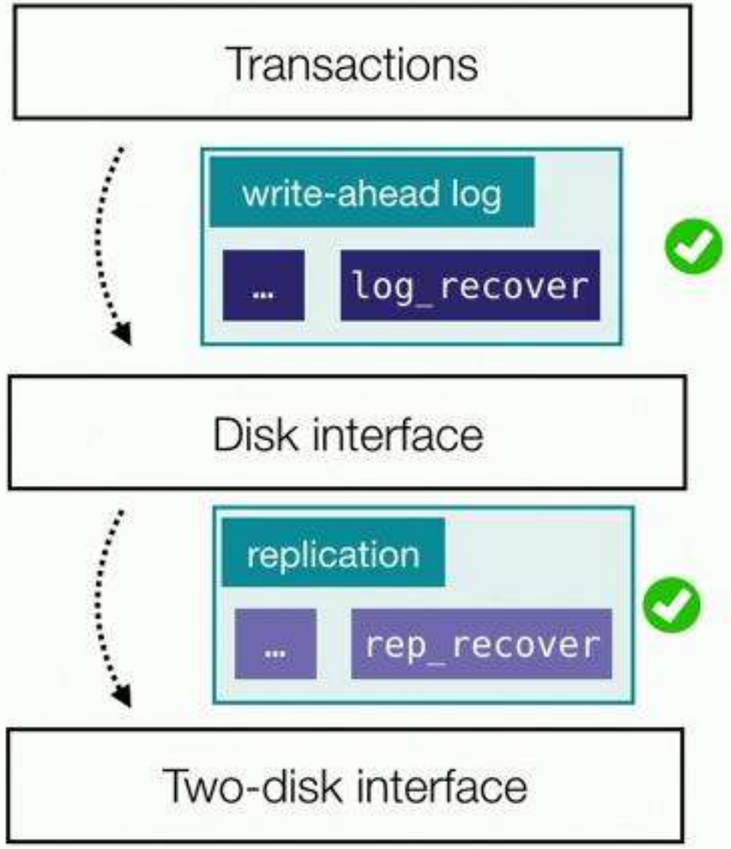
Kleene algebra for transition relations

Kleene algebra for transition relations

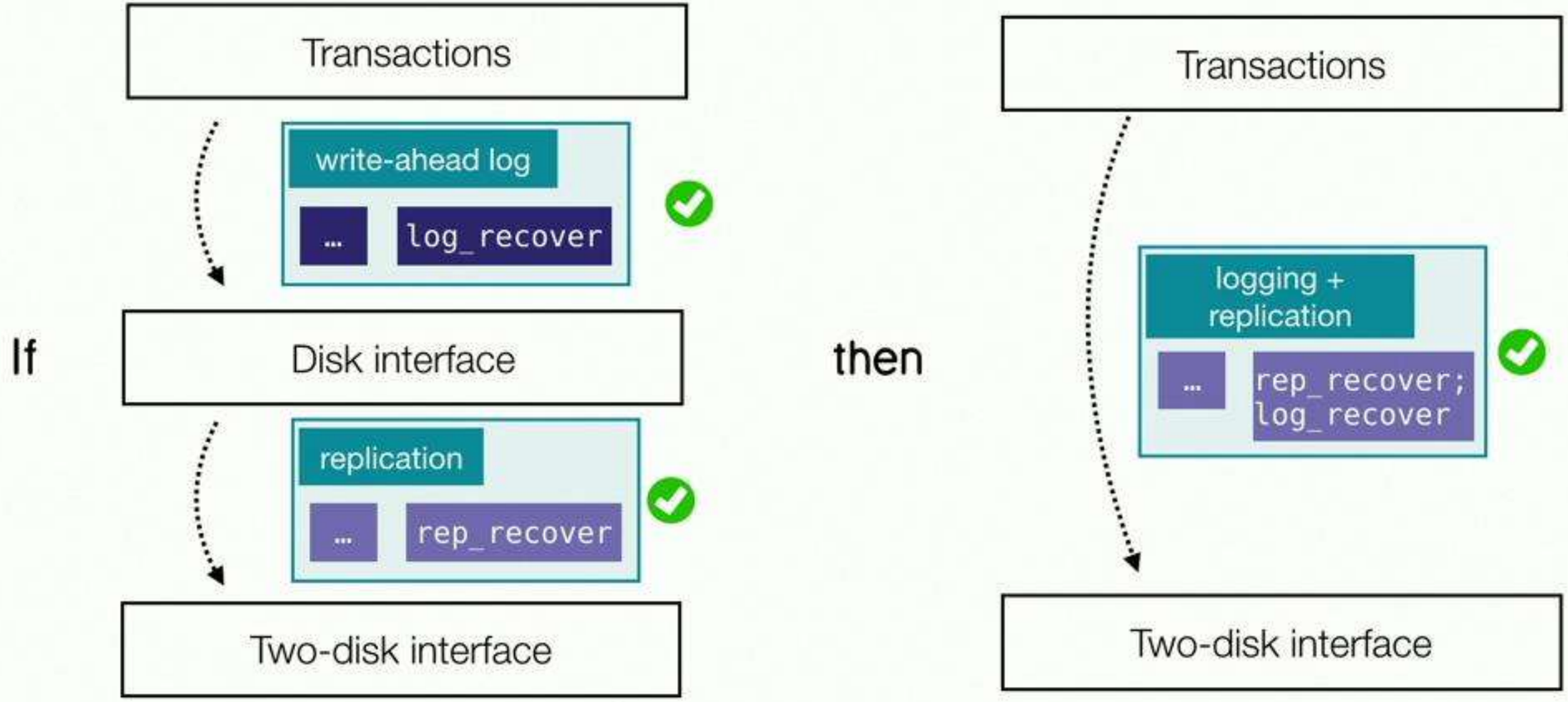


Theorem: recovery refinements compose

If

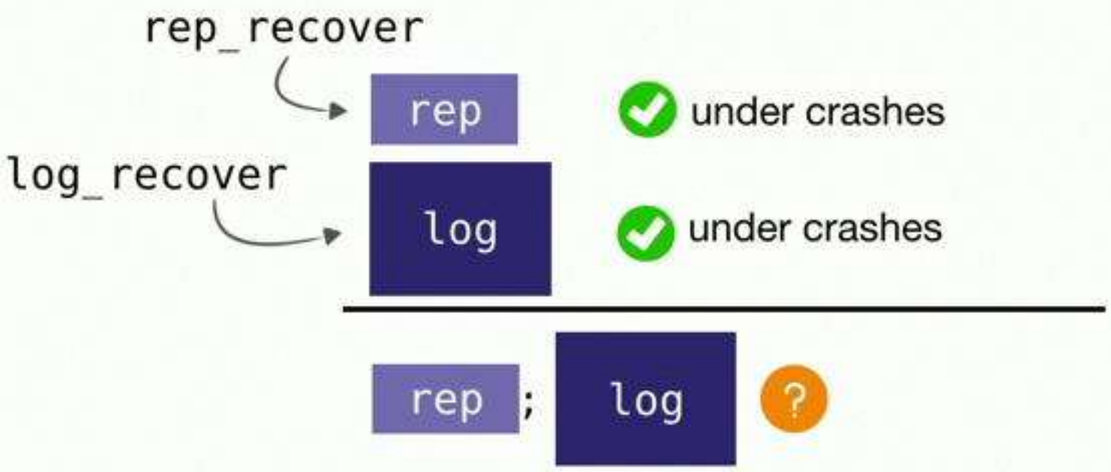


Theorem: recovery refinements compose

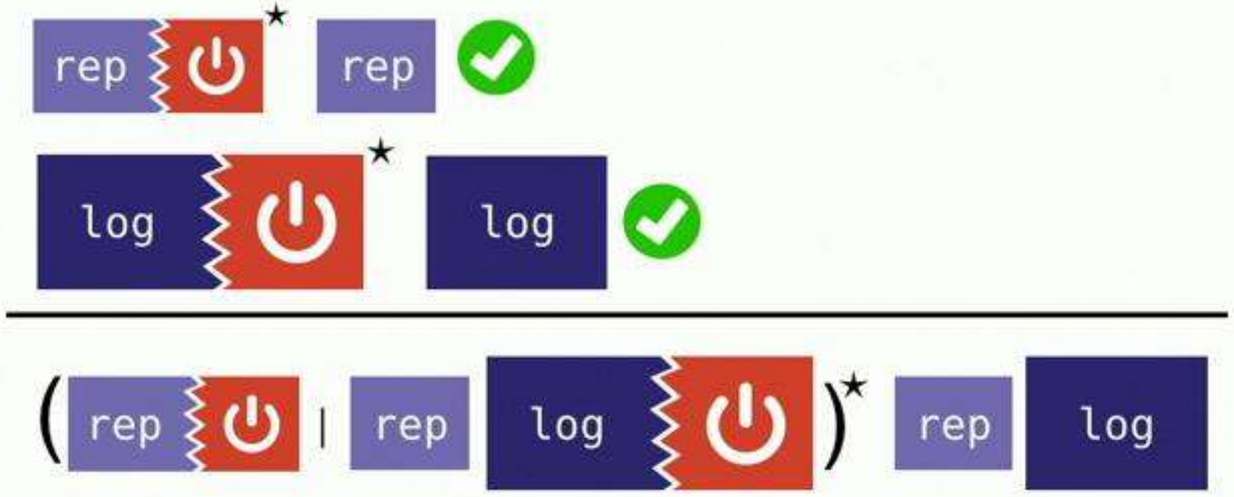


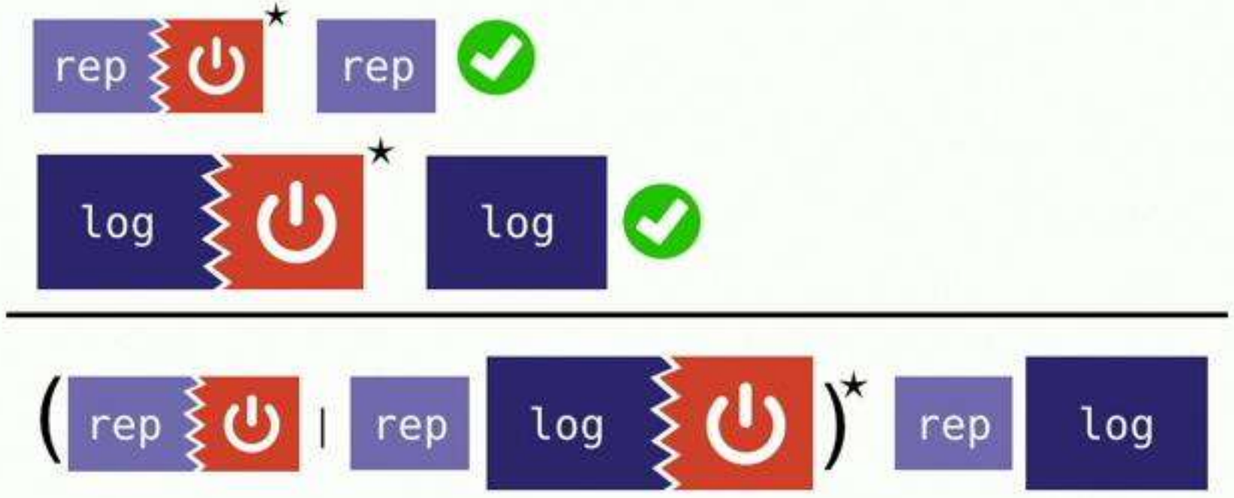
Goal: prove composed recovery correct

Goal: prove composed recovery correct




rep
rep
log
log


rep
rep
log
log
(
rep
|
rep
log
)*
rep
log



how to re-use recovery proofs here?

Using Kleene algebra for reasoning

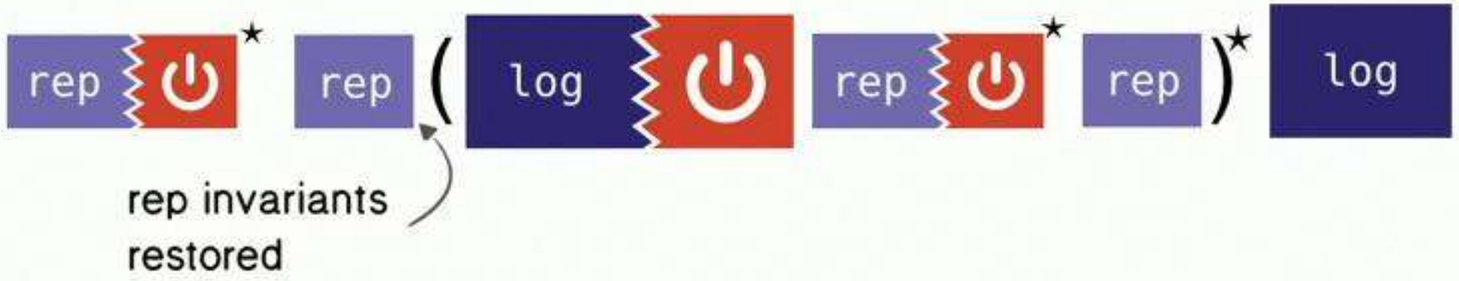
after *de-nesting* $(p \mid q)^\star = p^\star(qp^\star)^\star$

Using Kleene algebra for reasoning

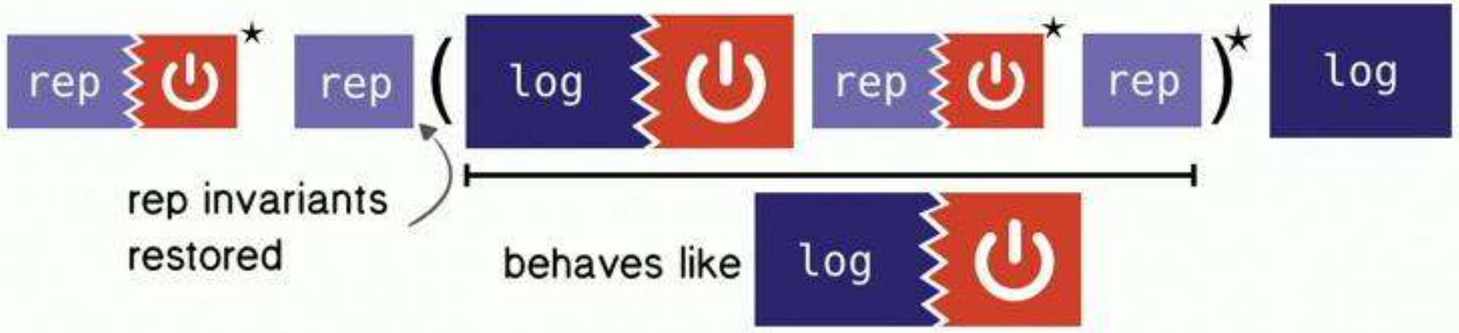
after *de-nesting* $(p \mid q)^\star = p^\star(qp^\star)^\star$

after *sliding* $(pq)^\star p = p(qp)^\star$

After rewrite both proofs apply

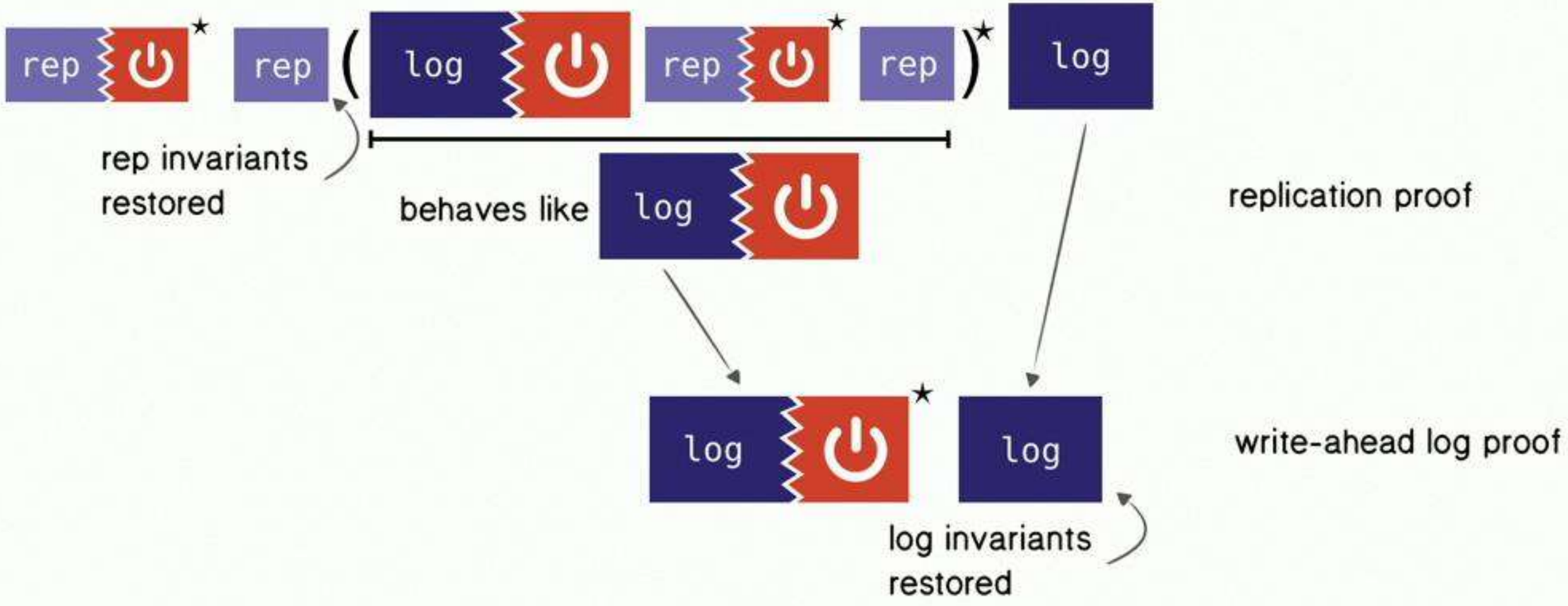


After rewrite both proofs apply



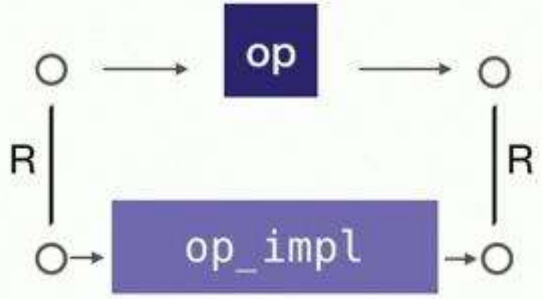
replication proof

After rewrite both proofs apply



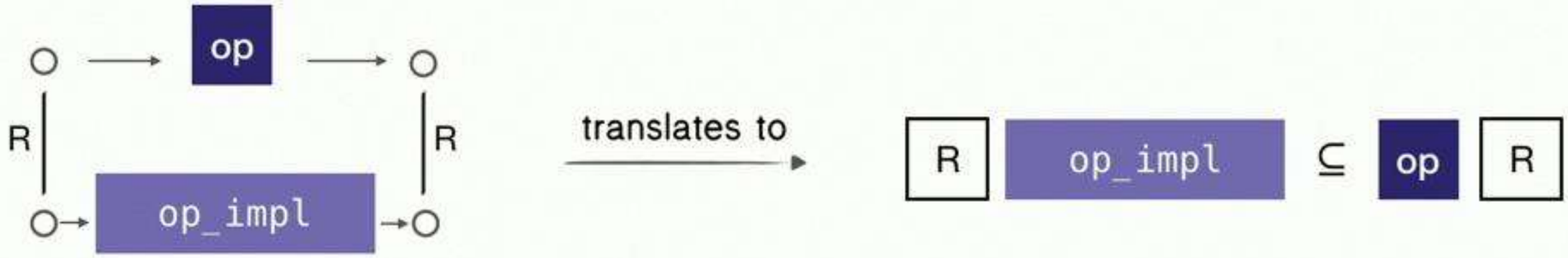
Kleene algebra also helps with refinement

See paper



Kleene algebra also helps with refinement

See paper



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 v
(using the Coq proof assistant
its specification and whose
FSCQ provably avoids bugs
systems, such as performing
barriers or forgetting to zero
happens at an inopportune ti

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 log-

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

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 v
(using the Coq proof assistant
its specification and whose
FSCQ provably avoids bugs
systems, such as performing
barriers or forgetting to zero
happens at an inopportune ti

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 `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 log-

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

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

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

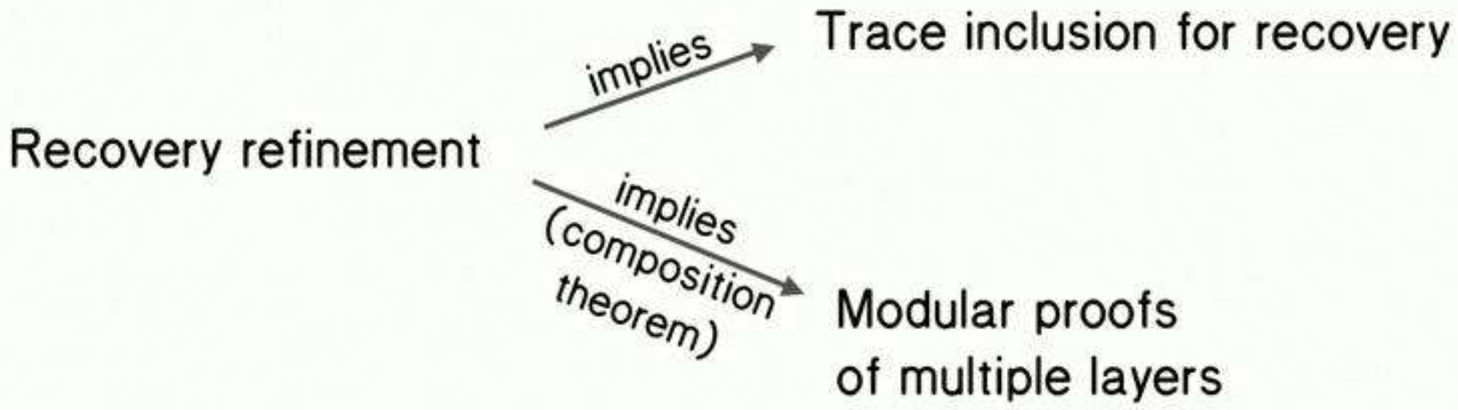
```
def atomic_save(data, path):
  write_all(data, tmp)
  rename(tmp, path)

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

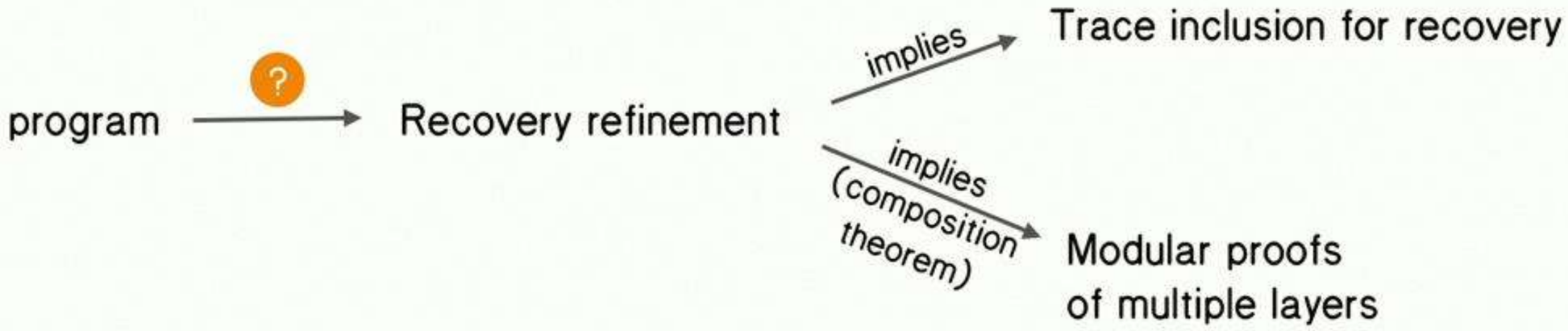
this is non-modular and makes the proof much harder

Proving this code correct took 1500 lines of proof code!

Argosy so far



Argosy so far



Crash Hoare Logic

Hoare Logic

"Hoare triple" {P} code {Q}

precondition postcondition

Hoare Logic

"Hoare triple" {P} code {Q}

precondition postcondition

:=

if P(s) then Q(s')

Crash Hoare Logic

"crash specification" {P} code {Q} {Qc}

precondition postcondition crash invariant

:= if P(s) then Q(s')

if P(s) then Qc(s')

Crash Hoare Logic

"recovery specification" {P} code ↺ recover {Q} {Qr}

precondition — postcondition — recovery postcondition

if P(s) then Q(s')

:=

if P(s) then Qr(s')

Crash Hoare Logic

"crash specification" {P} code {Q} {Qc}

precondition postcondition crash invariant

:= if P(s) then Q(s')

if P(s) then Qc(s')

Crash Hoare Logic

"recovery specification" {P} code ↺ recover {Q} {Qr}

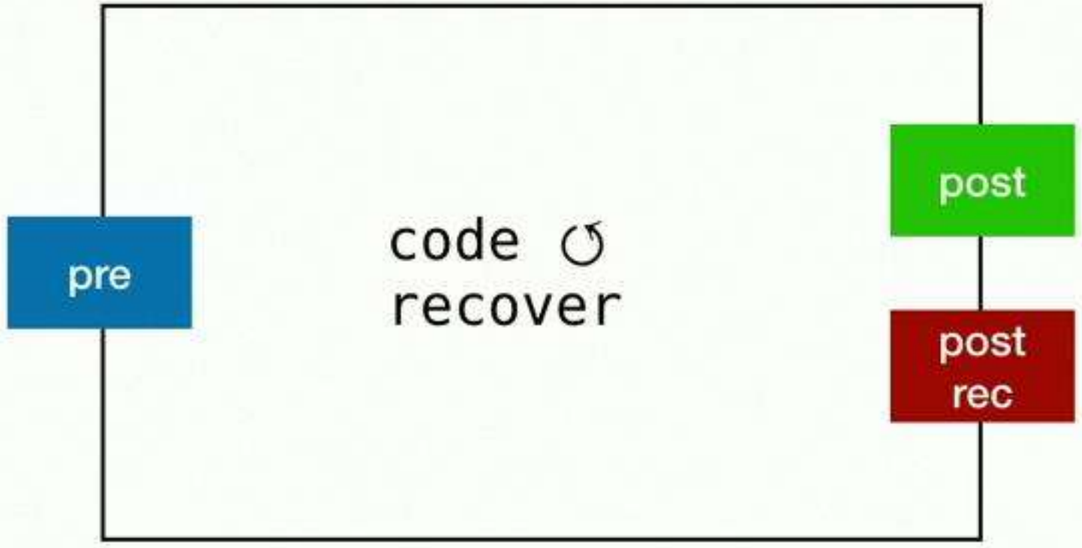
precondition — postcondition — recovery postcondition

if P(s) then Q(s')

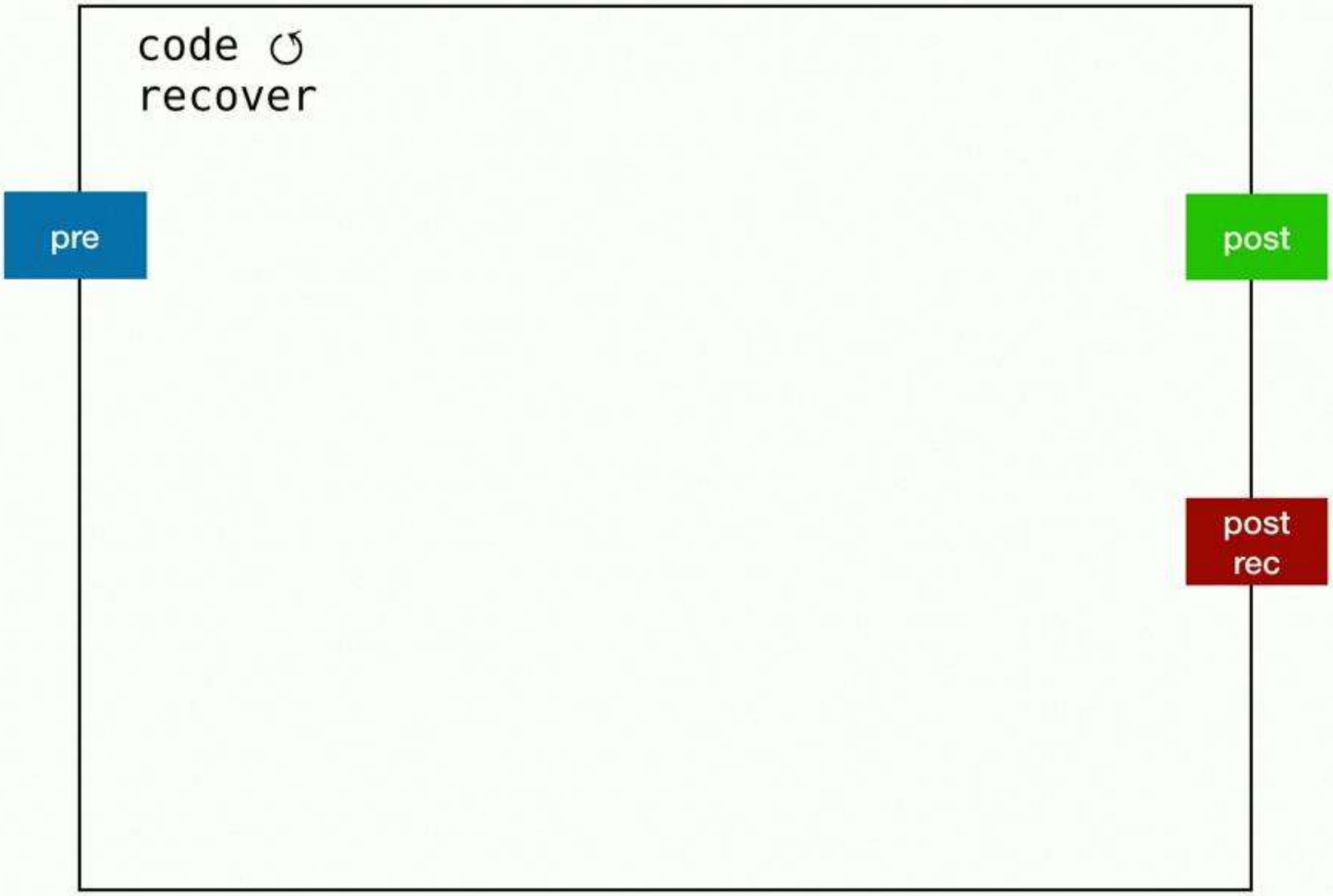
:=

if P(s) then Qr(s')

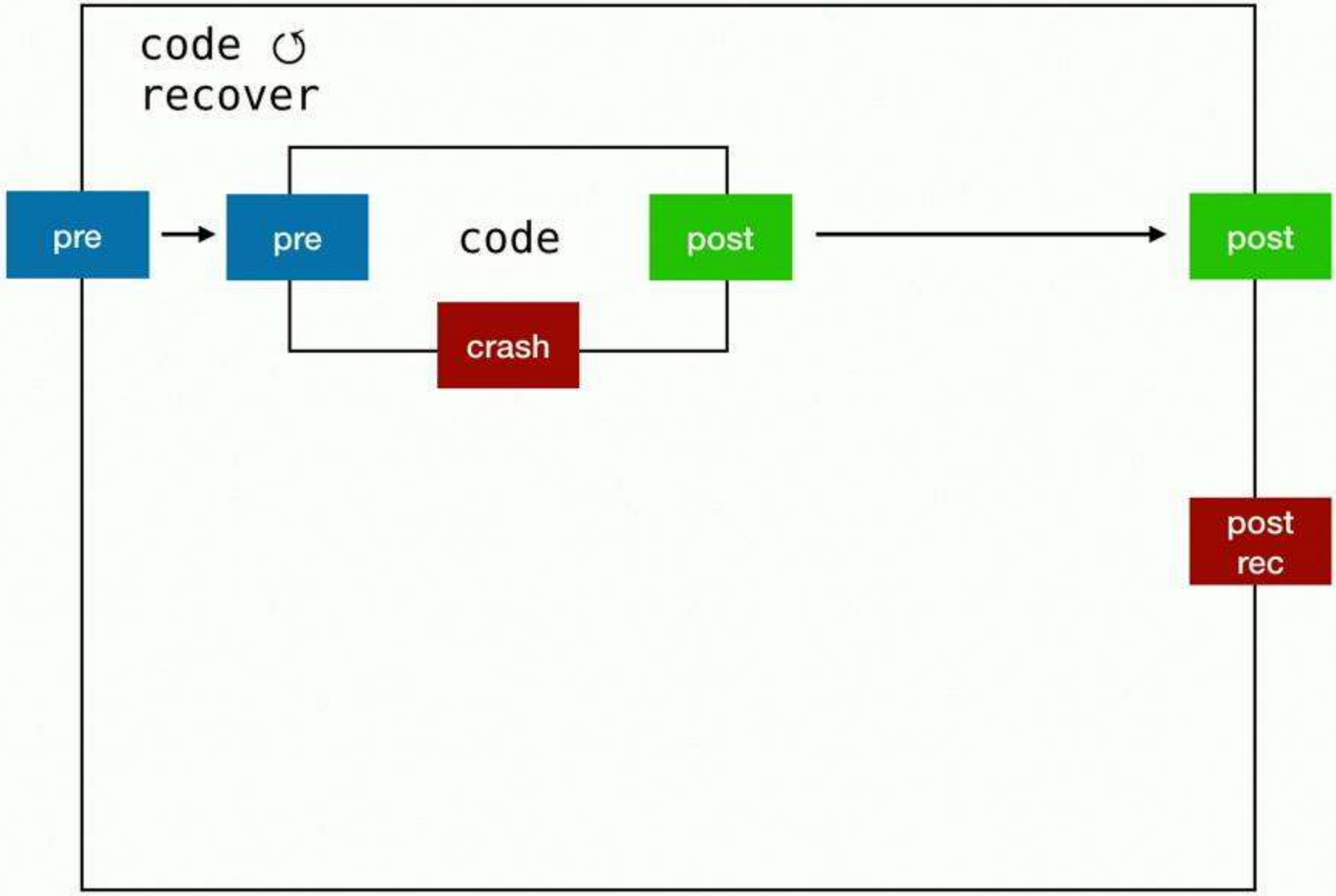
{pre} code ↺ recover {post} {post rec}



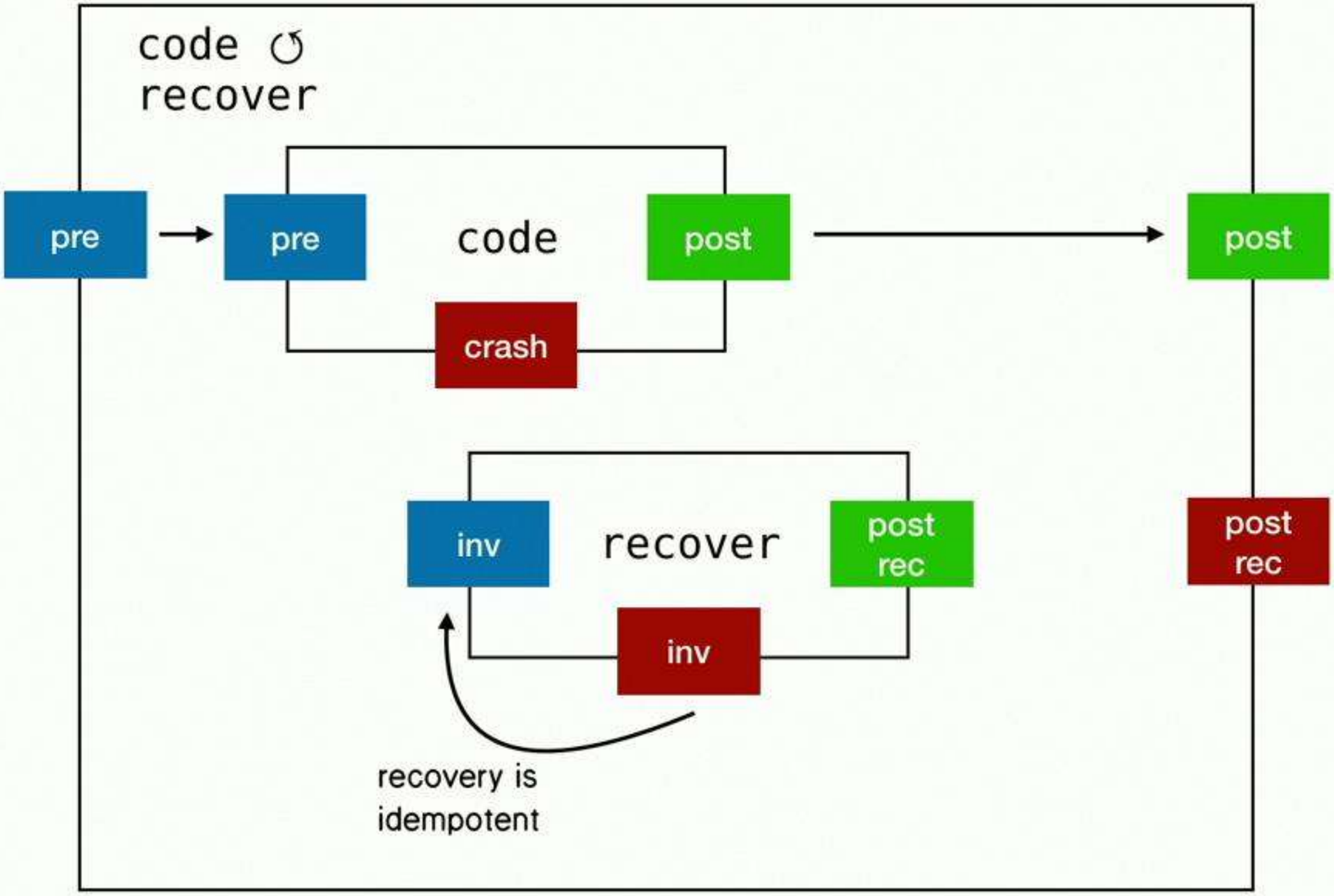
theorem in CHL to prove recovery specs from crash specs



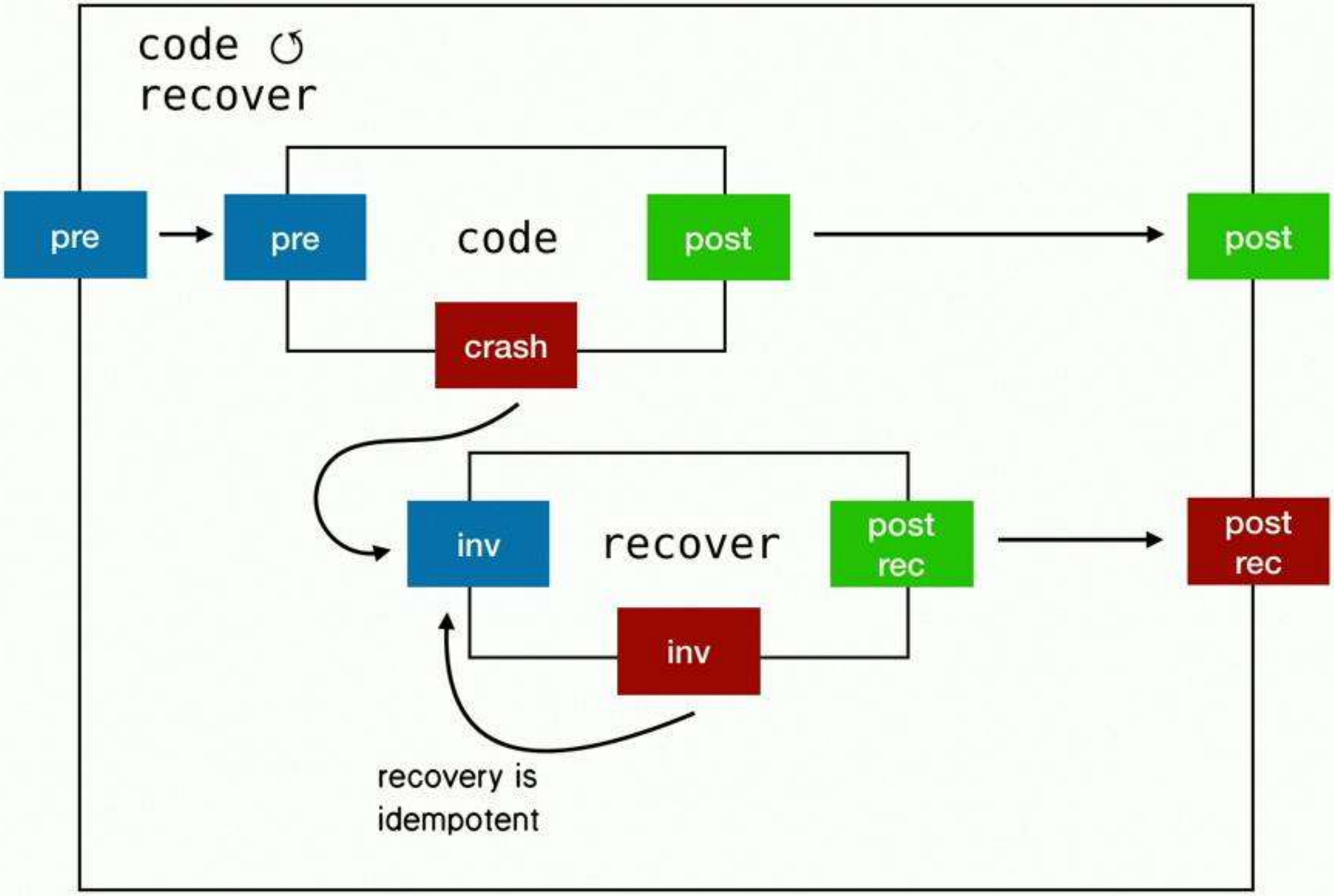
theorem in CHL to prove recovery specs from crash specs



theorem in CHL to prove recovery specs from crash specs



theorem in CHL to prove recovery specs from crash specs



Argosy connects CHL to recovery refinement

Come up with *abstraction relation*

Prove a *refinement specification* for every operation

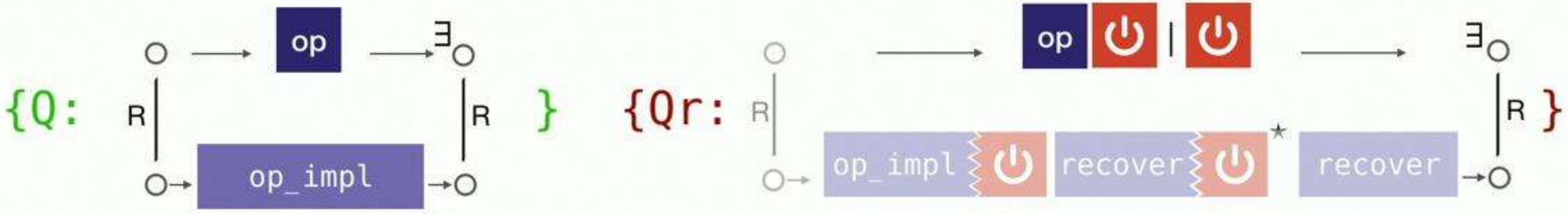
Gives recovery refinement for implementation

Recovery refinement as a CHL spec

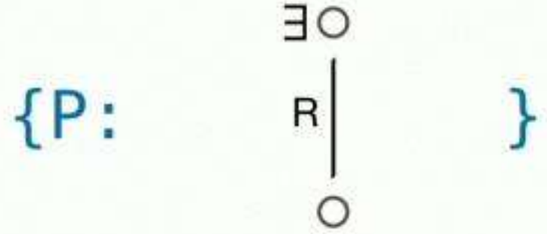
$$\{P\}\ \ \mathtt{op\_impl}\ \circlearrowleft\ \mathtt{recover}\ \ \{Q\}\ \ \{Qr\}$$

Recovery refinement as a CHL spec

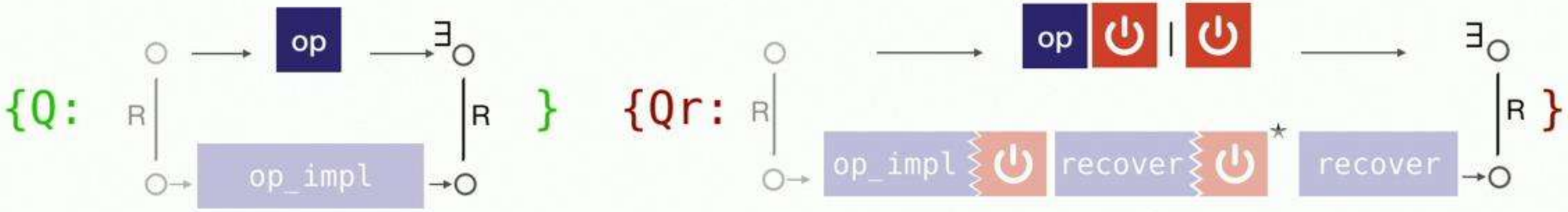
op_impl ↺ recover

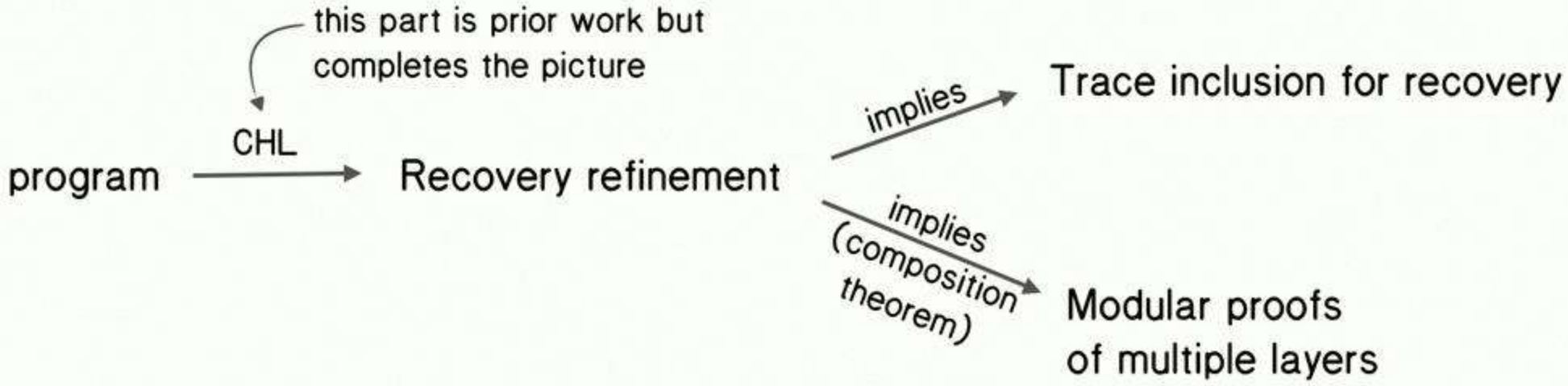


Recovery refinement as a CHL spec



op_impl ↺ recover




this part is prior work but
completes the picture
CHL
program
Recovery refinement
implies
Trace inclusion for recovery
implies
(composition
theorem)
Modular proofs
of multiple layers

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

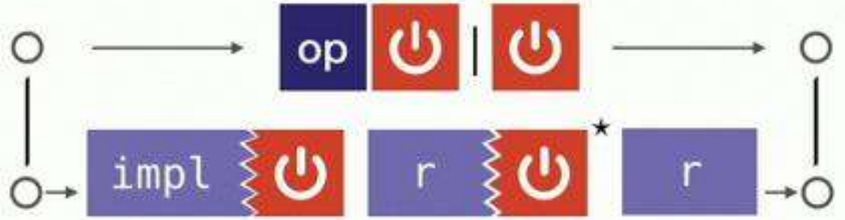
Argosy: modular proofs of layered storage systems

Kleene algebra

Argosy: modular proofs of layered storage systems

Kleene algebra

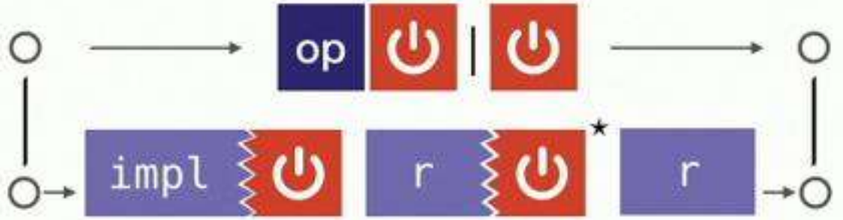
recovery refinement



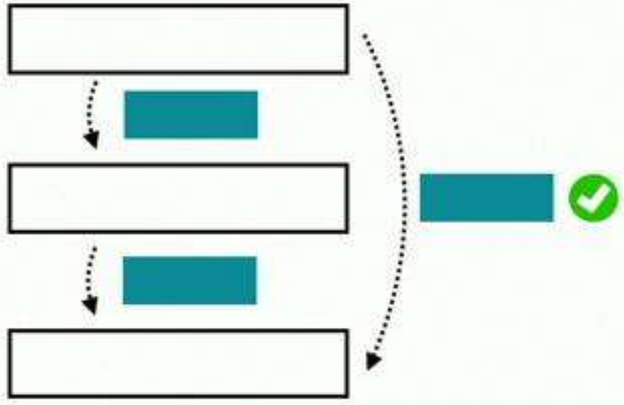
Argosy: modular proofs of layered storage systems

Kleene algebra

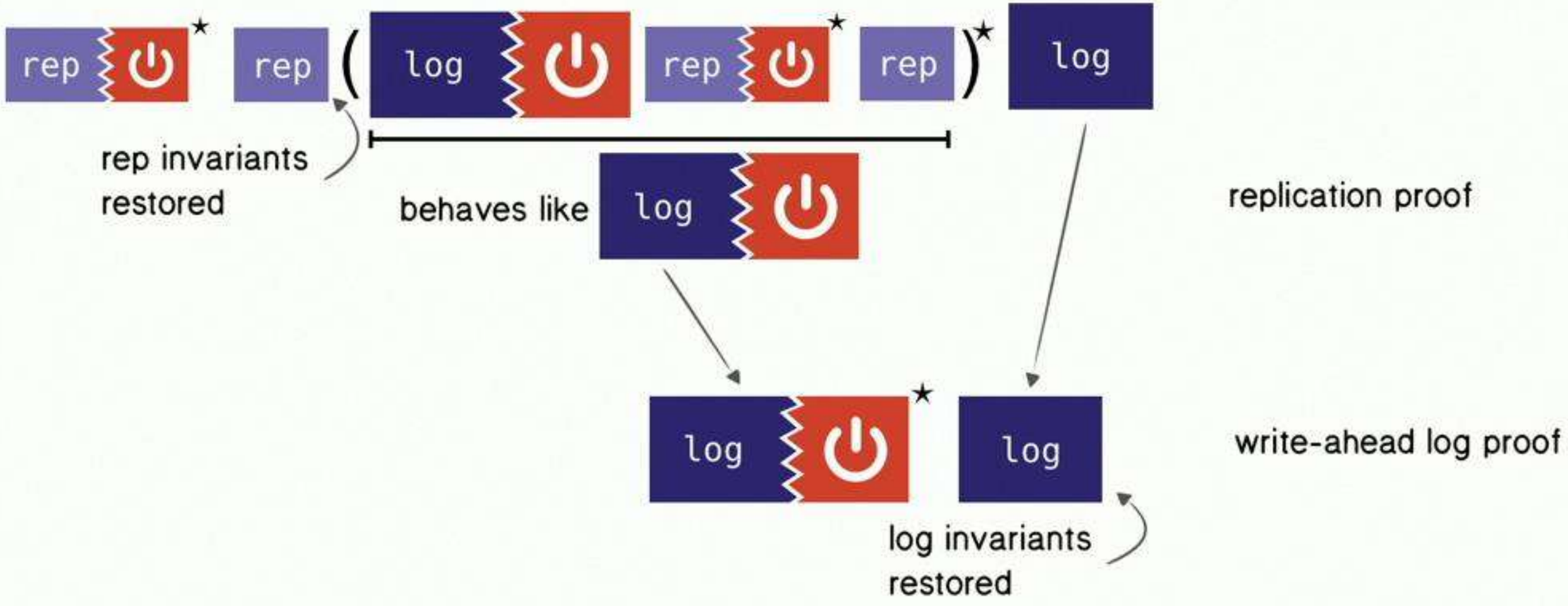
recovery refinement



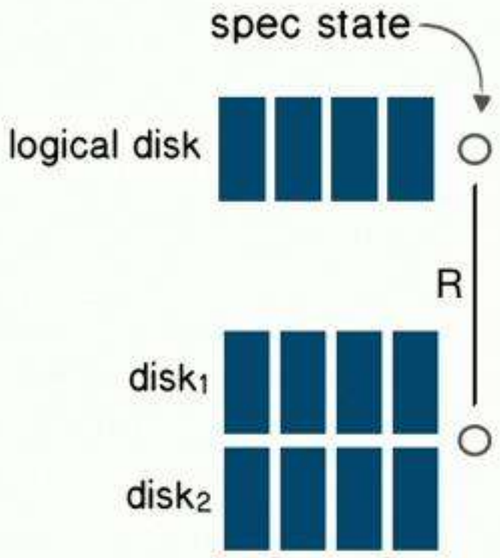
modular proofs



After rewrite both proofs apply



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)
```