Push-Button Verification of File Systems

via Crash Refinement

Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, Xi Wang

University of Washington

File systems are hard to get right

- Complex hierarchical on-disk data structures
- Maintain consistency in the face of crashes

Example: Missing flush in ext4

```
269 +       /* Make sure all replayed data is on permanent storage */
270 +       if (journal->j_flags & JBD2_BARRIER)
271 +               blkdev_issue_flush(journal->j_fs_dev, GFP_KERNEL, NULL);
```

- Bugs are hard to reproduce
- Hard to test fixes

> *It's unlikely this will be necessary [..] but we need this in order to guarantee correctness.*
>
> *File System Developers*

Verification: Effective at eliminating bugs, but costly

- Prove the absence of bugs
  - BilbyFS [ASPLOS 2016]
  - FSCQ [SOSP 2015]

- Requires expertise and are labor intensive

Goal: Push-button verification

- No manual annotation / proof of implementation
- Get a concrete test case for any bug
- Our approach: System design to leverage a state-of-the-art automated theorem prover, Z3.

Push-button verification: Challenges

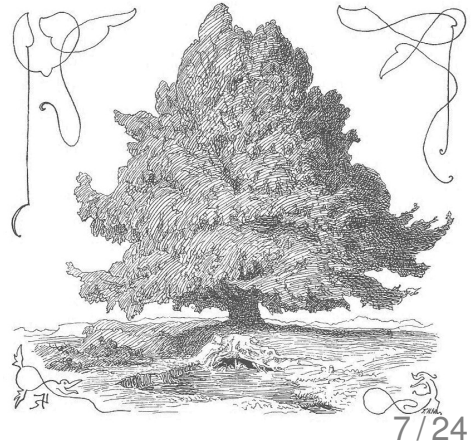
- **Need a formalization of correctness**
  - Needs to capture crash & recovery
  - Needs to be automatically verifiable
- **Too many states to exhaust**
  - Disks are large
  - Many execution paths
  - Non-determinism
- **Prior work focused on bug finding**
  - eXplode [OSDI '06], EXE [CCS '06]
  - Useful, but incomplete

Contributions and outlines

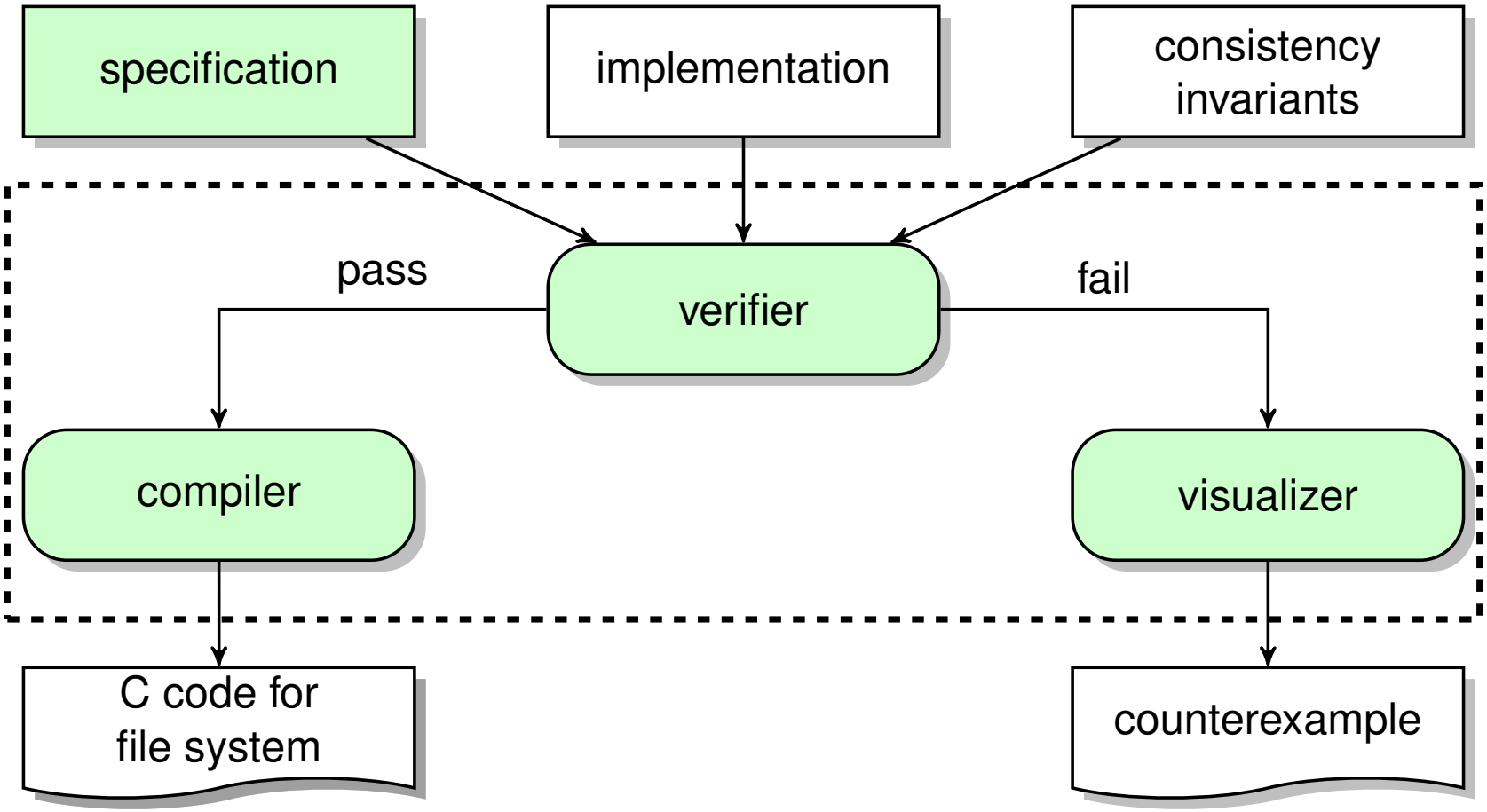
- Yggdrasil ['yg:,drasil:]:
  A toolkit for building verified file systems

- *Crash refinement*
  - A new definition of file-system correctness
  - Enable modularity to scale verification

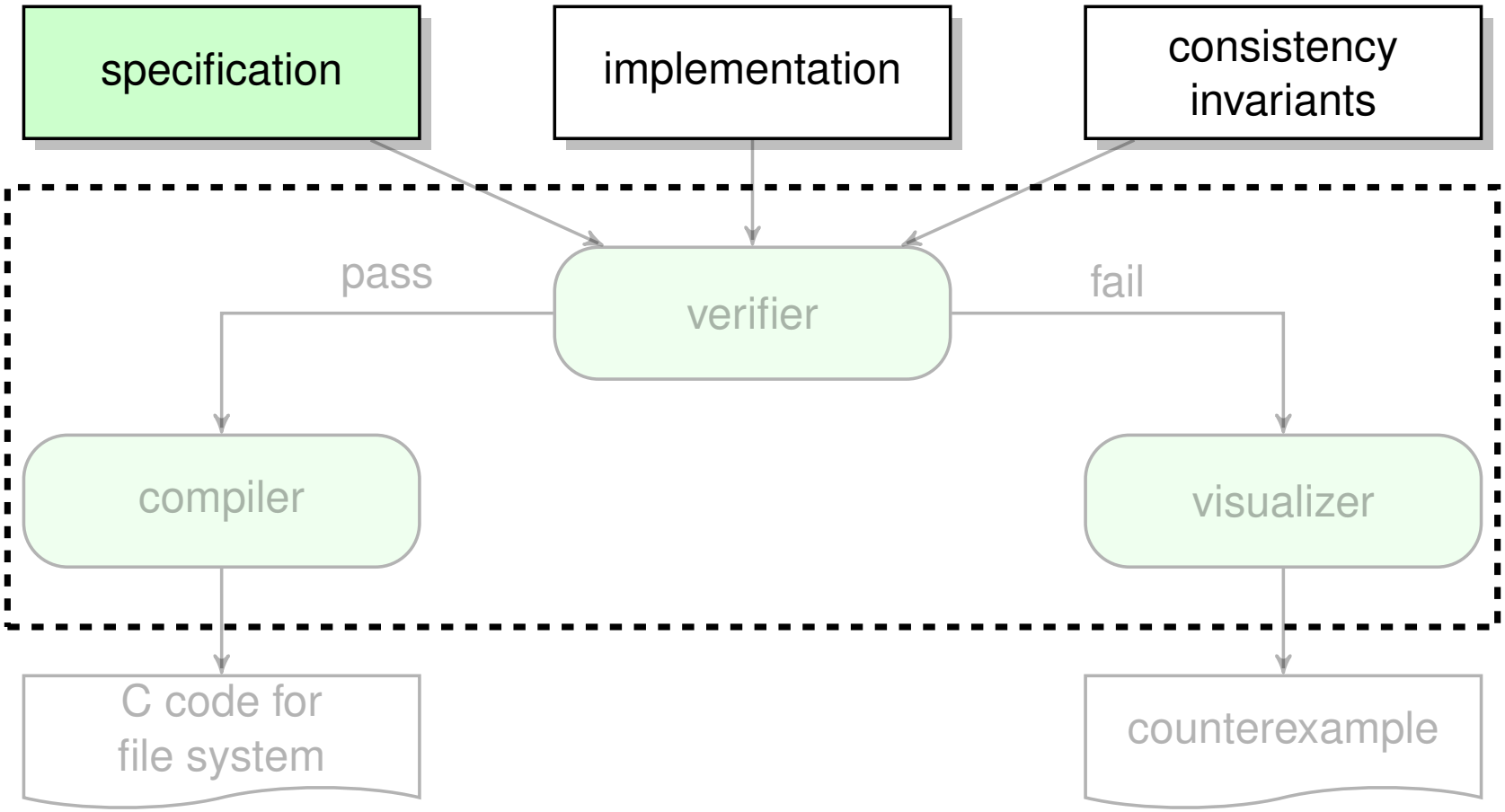
- Case study: The Yxv6 file system
  - Similar to ext3 and xv6, but guarantees functional correctness and crash safety



Yggdrasil Overview (Green – trusted components)



Yggdrasil Overview (Green – trusted components)

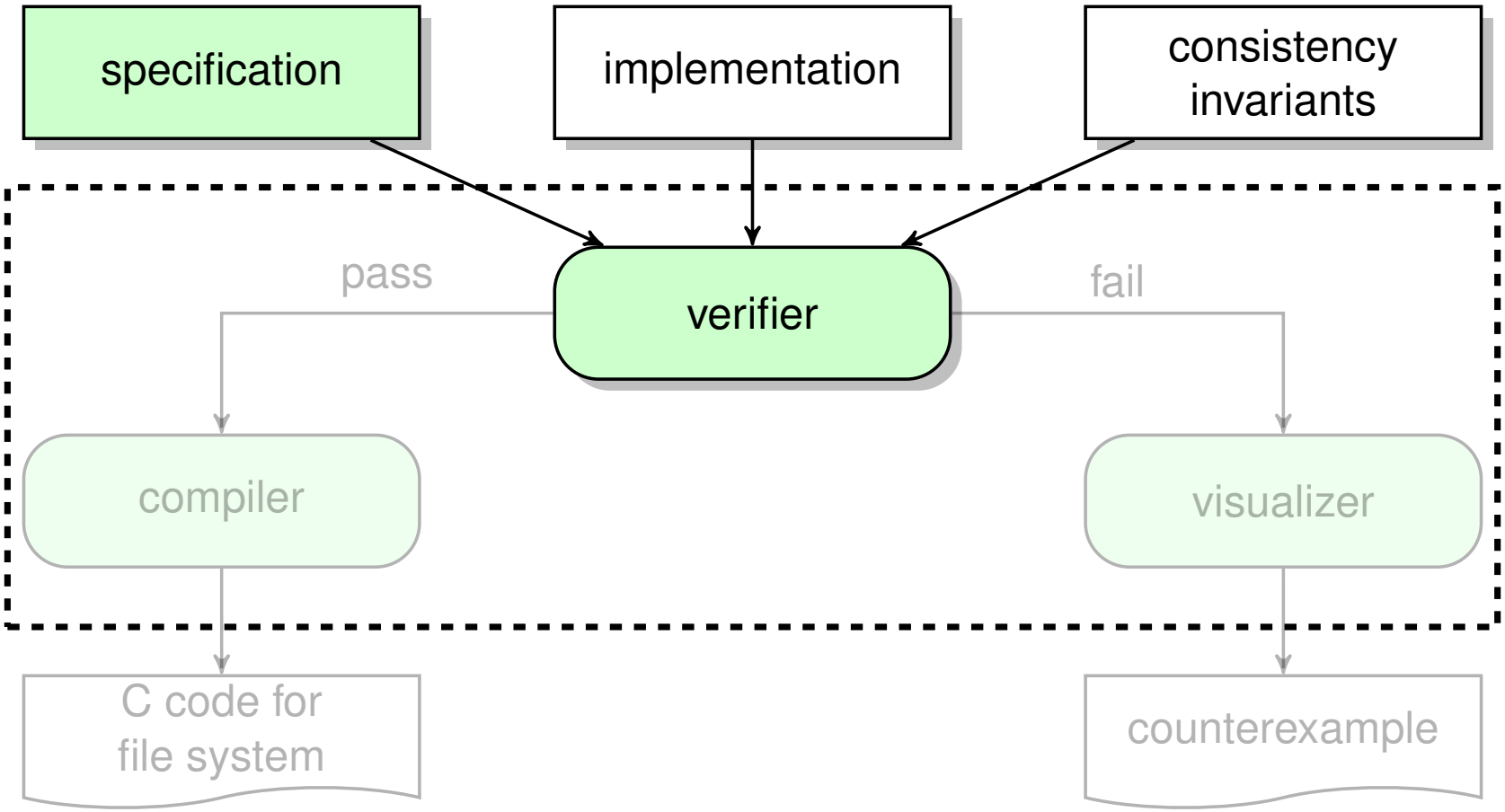


Yggdrasil Overview (Green – trusted components)

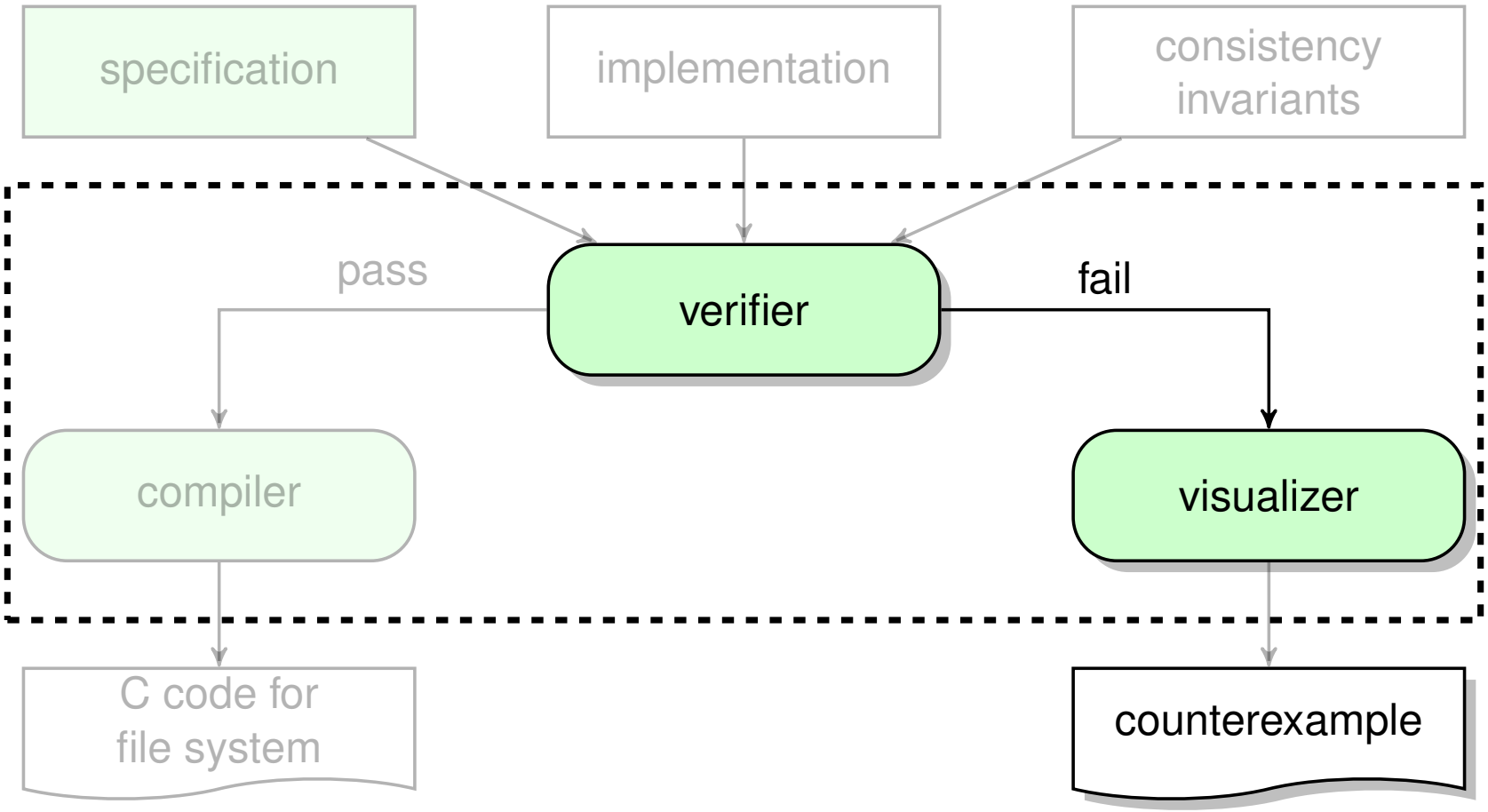
specification | implementation | consistency invariants

```
class TxnDisk(BaseSpec):
    def begin_tx(self):
        self._txn = []

    def write_tx(self, bid, data):
        self._cache = self._cache.update(bid, data)
        self._txn.append((bid, data))

    def commit_tx(self):
        with self._mach.transaction():
            for bid, data in self._txn:
                self._disk = self._disk.update(bid, data)
```

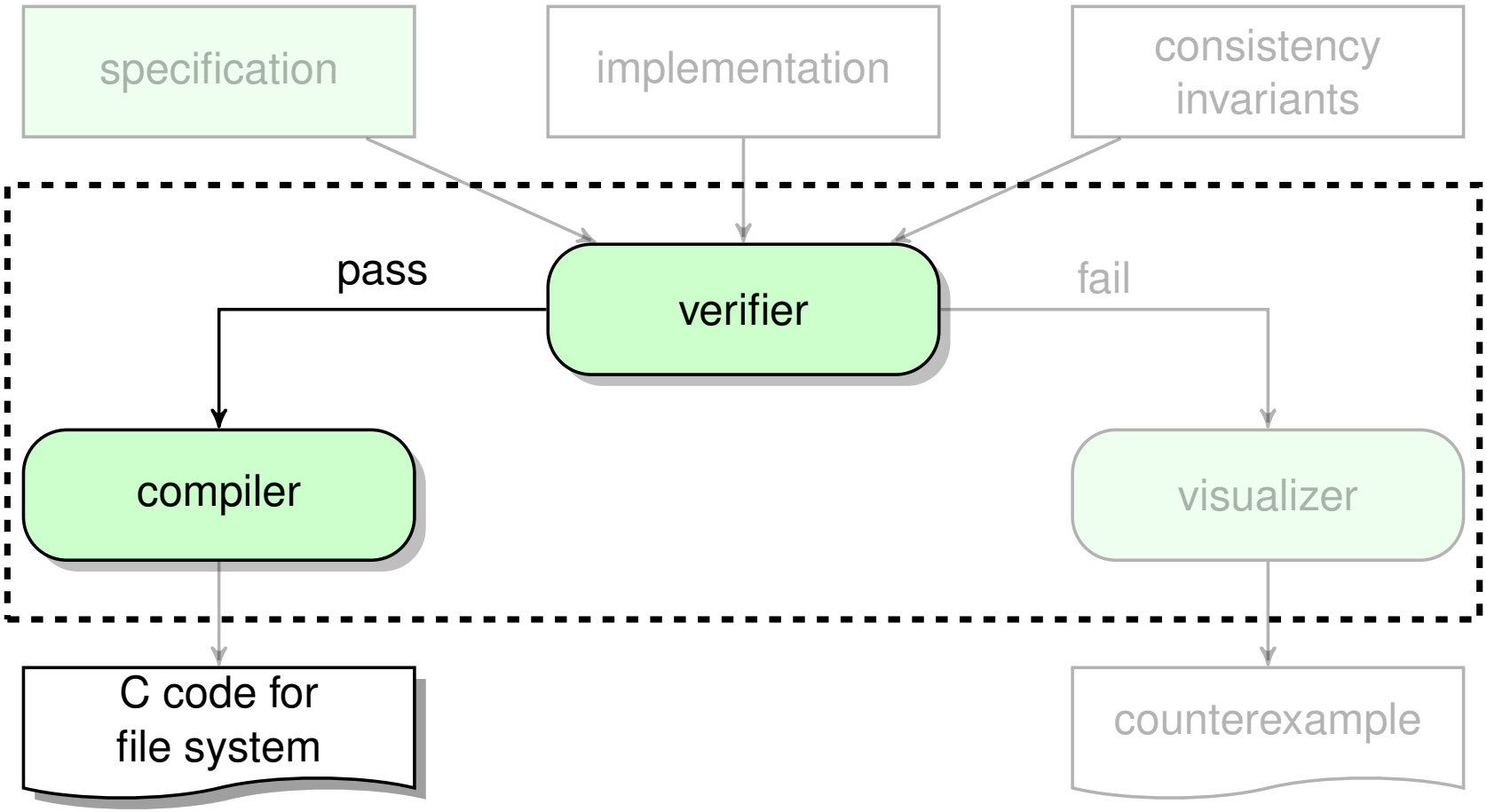
Yggdrasil Overview (Green – trusted components)



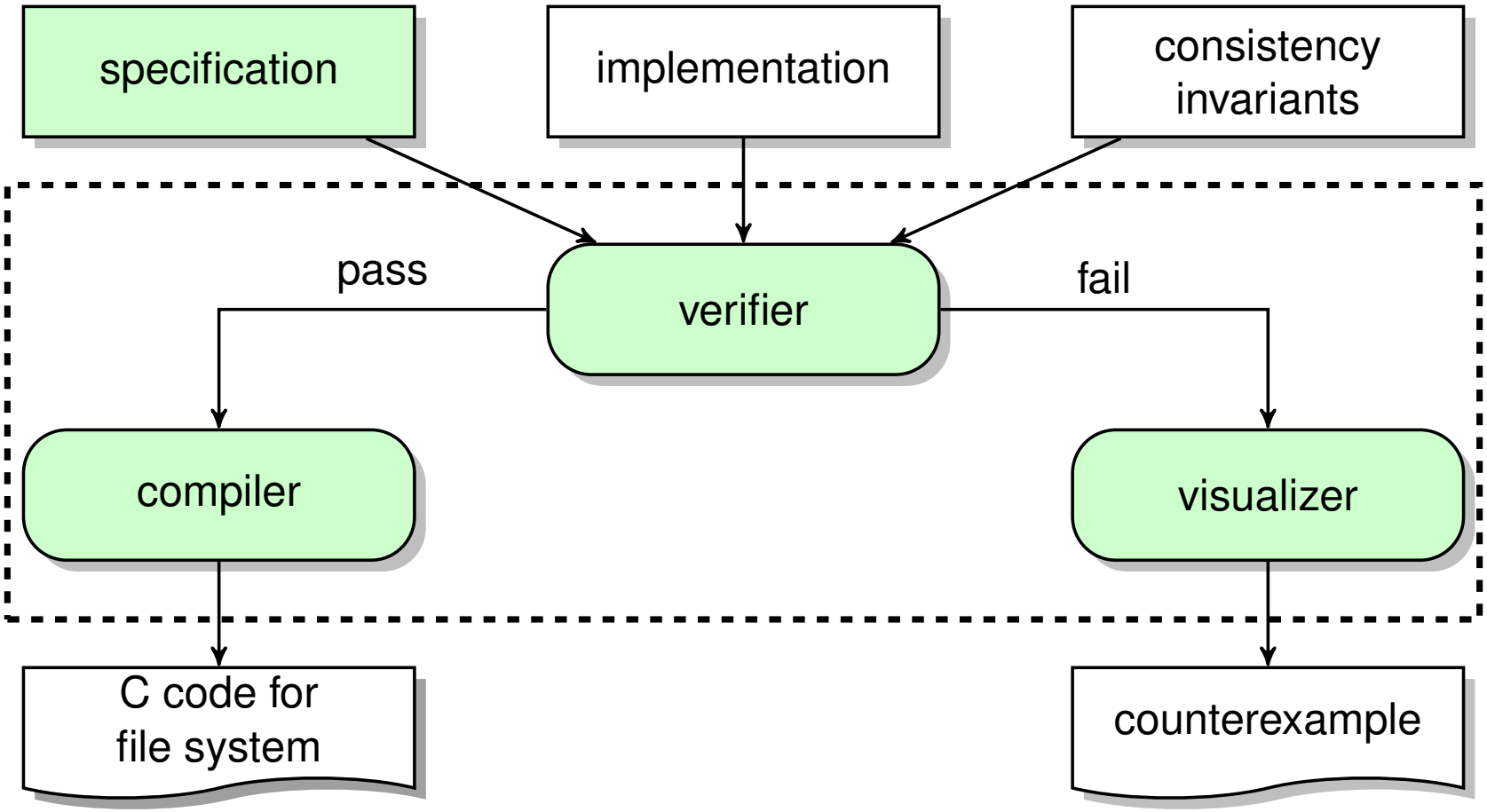
Yggdrasil Overview (Green – trusted components)



Yggdrasil Overview (Green – trusted components)



Yggdrasil Overview (Green – trusted components)

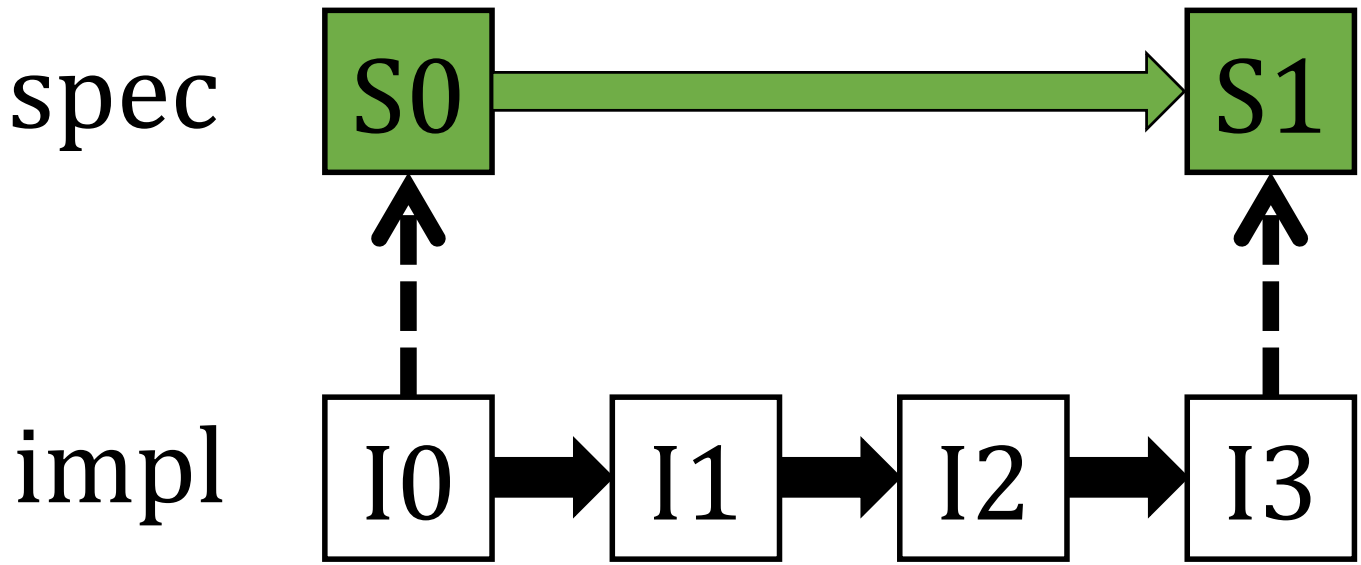


Summary of the Yggdrasil toolkit

- Easy to use, no complex logic required
- Useful test-cases for bugs
- Limitations
  - No concurrency
  - Unverified Python to C compiler and FUSE

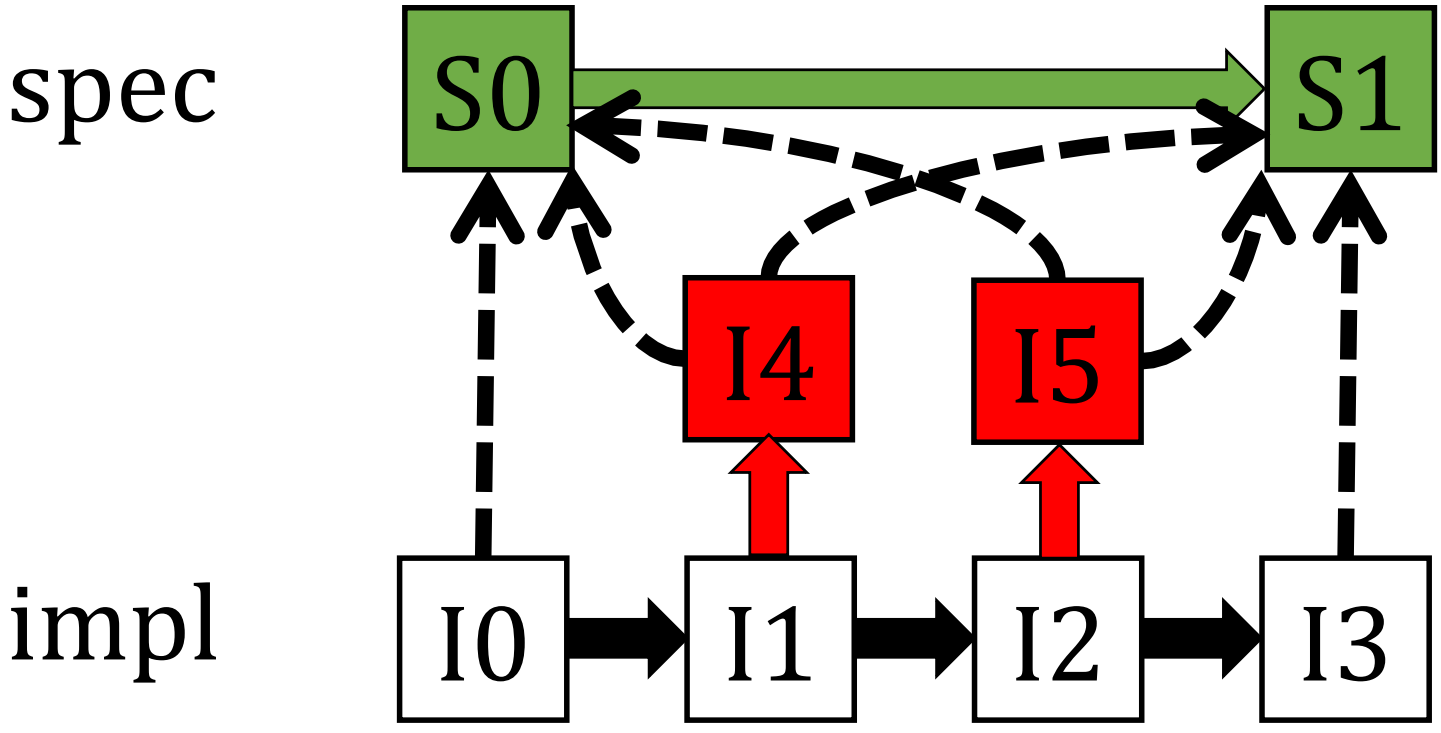
Straw-man approach to verify FS

- Model *FS* as a state machine with a set of operations {`mknod`, `rename`, etc.}.



- Limitation: Doesn't capture crashes

Crash refinement: Intuition



- Formalize this intuition
  1. Capture crashes explicitly with a *crash schedule*
  2. Use the *crash schedule* to define correctness

Crash refinement ½: Crash schedule

- Explicit *crash-schedule*
  - A set of boolean variables
  - Captures crashes and disk reorderings

Crash refinement ½: Crash schedule

- Explicit *crash-schedule*
  - A set of boolean variables
  - Captures crashes and disk reorderings

Operation	Schedule	Disk state
write($a1, v1$)		
write($a2, v2$)		

Crash refinement ½: Crash schedule

- Explicit *crash-schedule*
  - A set of boolean variables
  - Captures crashes and disk reorderings

	Operation	Schedule	Disk state
⇒	write($a1, v1$)		
	write($a2, v2$)		

Crash refinement ½: Crash schedule

- Explicit *crash-schedule*
  - A set of boolean variables
  - Captures crashes and disk reorderings

	Operation	Schedule	Disk state
⇒	write($a1, v1$)	$\{b1\}$	
	write($a2, v2$)		

Crash refinement ½: Crash schedule

- Explicit *crash-schedule*
  - A set of boolean variables
  - Captures crashes and disk reorderings

	Operation	Schedule	Disk state
$\Rightarrow$	write($a1, v1$)	$\{b1\}$	$d\,[a1 \mapsto \mathit{if}\ b1\ \mathit{then}\ v1\ \mathit{else}\ \mathrm{old}(a1)]$
	write($a2, v2$)		

Crash refinement ½: Crash schedule

- Explicit *crash-schedule*
  - A set of boolean variables
  - Captures crashes and disk reorderings

	Operation	Schedule	Disk state
	write($a1, v1$)	$\{b1\}$	$d\,[a1 \mapsto \mathit{if}\ b1\ \mathit{then}\ v1\ \mathit{else}\ \mathrm{old}(a1)]$
$\Rightarrow$	write($a2, v2$)		

Crash refinement ½: Crash schedule

- Explicit *crash-schedule*
  - A set of boolean variables
  - Captures crashes and disk reorderings

	Operation	Schedule	Disk state
	write($a1, v1$)	$\{b1\}$	$d\,[a1 \mapsto \mathit{if}\ b1\ \mathit{then}\ v1\ \mathit{else}\ \mathrm{old}(a1)]$
$\Rightarrow$	write($a2, v2$)	$\{b1, b2\}$	

Crash refinement ½: Crash schedule

- Explicit *crash-schedule*
  - A set of boolean variables
  - Captures crashes and disk reorderings

	Operation	Schedule	Disk state
	write($a1, v1$)	$\{b1\}$	$d\,[a1 \mapsto \mathit{if}\ b1\ \mathit{then}\ v1\ \mathit{else}\ \mathrm{old}(a1)]$
$\Rightarrow$	write($a2, v2$)	$\{b1, b2\}$	$d \begin{bmatrix} a1 \mapsto \mathit{if}\ b1\ \mathit{then}\ v1\ \mathit{else}\ \mathrm{old}(a1) \\ a2 \mapsto \mathit{if}\ b2\ \mathit{then}\ v2\ \mathit{else}\ \mathrm{old}(a2) \end{bmatrix}$

Crash refinement ½: Crash schedule

- Explicit *crash-schedule*
  - A set of boolean variables
  - Captures crashes and disk reorderings

Operation	Schedule	Disk state
write($a1, v1$)	$\{b1\}$	$d\,[a1 \mapsto \mathit{if}\ b1\ \mathit{then}\ v1\ \mathit{else}\ \mathrm{old}(a1)]$
write($a2, v2$)	$\{b1, b2\}$	$d \begin{bmatrix} a1 \mapsto \mathit{if}\ b1\ \mathit{then}\ v1\ \mathit{else}\ \mathrm{old}(a1) \\ a2 \mapsto \mathit{if}\ b2\ \mathit{then}\ v2\ \mathit{else}\ \mathrm{old}(a2) \end{bmatrix}$

- Note: this program can produce 4 possible states

Crash refinement 2/2: Definition

1. Augment each *op* in *FS* with an explicit *crash schedule*: $op(disk, \mathbf{inp}, \mathbf{sched}) \rightarrow disk$

2. For each op $\in$ *FS*, prove:

$$\forall disk, \mathbf{inp}, \mathbf{sched}_{impl}.\ \exists \mathbf{sched}_{spec}.$$
$$op_{spec}(disk, \mathbf{inp}, \mathbf{sched}_{spec}) = op_{impl}(disk, \mathbf{inp}, \mathbf{sched}_{impl})$$

- Z3 is good at solving this particular form

Crash refinement 2/2: Definition

1. Augment each *op* in *FS* with an explicit *crash schedule*: $op(disk, \mathbf{inp}, \mathbf{sched}) \rightarrow disk$

2. Prove:

implementation states

$$\forall disk, \mathbf{inp}, \mathbf{sched}_{impl}. \ \exists \mathbf{sched}_{spec}.$$
$$op_{spec}(disk, \mathbf{inp}, \mathbf{sched}_{spec}) = op_{impl}(disk, \mathbf{inp}, \mathbf{sched}_{impl})$$

- Z3 is good at solving this particular form

Crash refinement 2/2: Definition

1. Augment each *op* in *FS* with an explicit *crash schedule*: $op(disk, \mathbf{inp}, \mathbf{sched}) \rightarrow disk$

2. For each op $\in$ *FS*:

   specification states

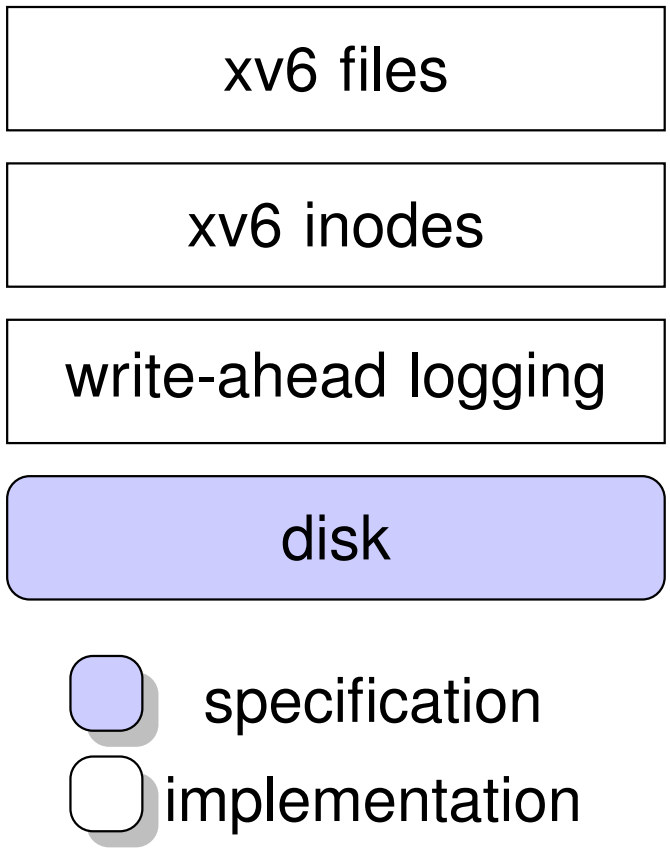
   $$\forall disk, \mathbf{inp}, \mathbf{sched}_{impl}.\ \boxed{\exists \mathbf{sched}_{spec}.}$$
   $$op_{spec}(disk, \mathbf{inp}, \mathbf{sched}_{spec}) = op_{impl}(disk, \mathbf{inp}, \mathbf{sched}_{impl})$$

- Z3 is good at solving this particular form

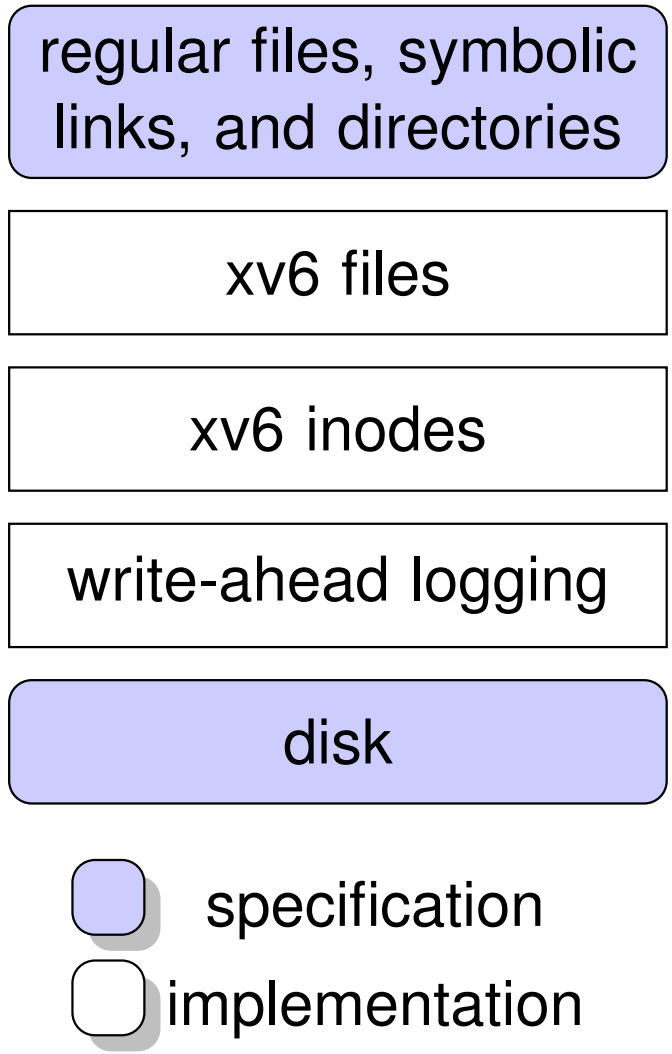
Crash refinement summary

- Amenable to automatic verification using Z3
- Enables modular, scalable verification
- Example: Decouple logical / physical data layout
  - Verify a simple layout first (ex. one inode per block)
  - Prove a separate crash-refinement for efficient layout
- Example: Stacking of layered abstractions

Verifying multiple layers: Straw-man approach

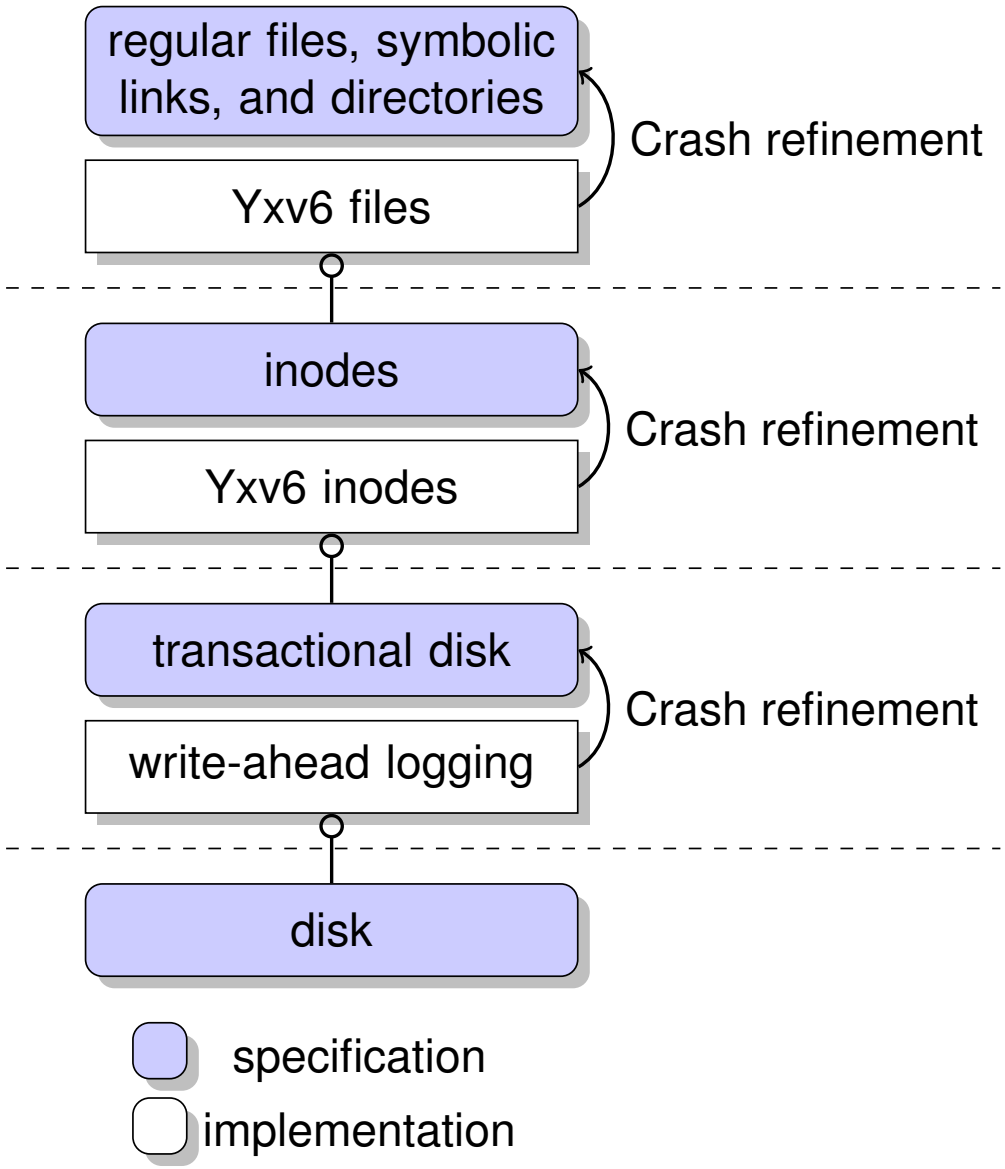


Verifying multiple layers: Straw-man approach



Yxv6 file system: Stack of layered abstractions

- Each layer has a specification
- Each layer builds upon a lower layer specification
- Limit verification to a single layer at a time



Implementation using Python and Z3

- Two Yxv6 variants
  - Yxv6+sync: similar to xv6, FSCQ and ext4+sync
  - Yxv6+group_commit: an optimized Yxv6+sync

component	specification	implementation	consistency inv
Yxv6	250	1,500	5
infrastructure	–	1,500	–
FUSE stub	–	250	–

- Also built: YminLFS, Ycp and Ylog

- No manual proofs!

Yxv6 evaluation

1. How long does it take to verify?
2. Is the implementation actually correct?
3. What is the development effort for Yxv6?
4. Is the performance of Yxv6 reasonable?

Yxv6 evaluation 1/4: Verification time

Let's see how many days it takes to verify Yxv6+sync

Yxv6 evaluation 2/4: Correctness

- Passed Linux testing project's fsstress
- Passed SibylFS [SOSP '15] Posix compliance test
  - Except for incomplete features (ex. hard links, acl)
- Passed manual crash and inspection tests
- Self hosting its development on Linux

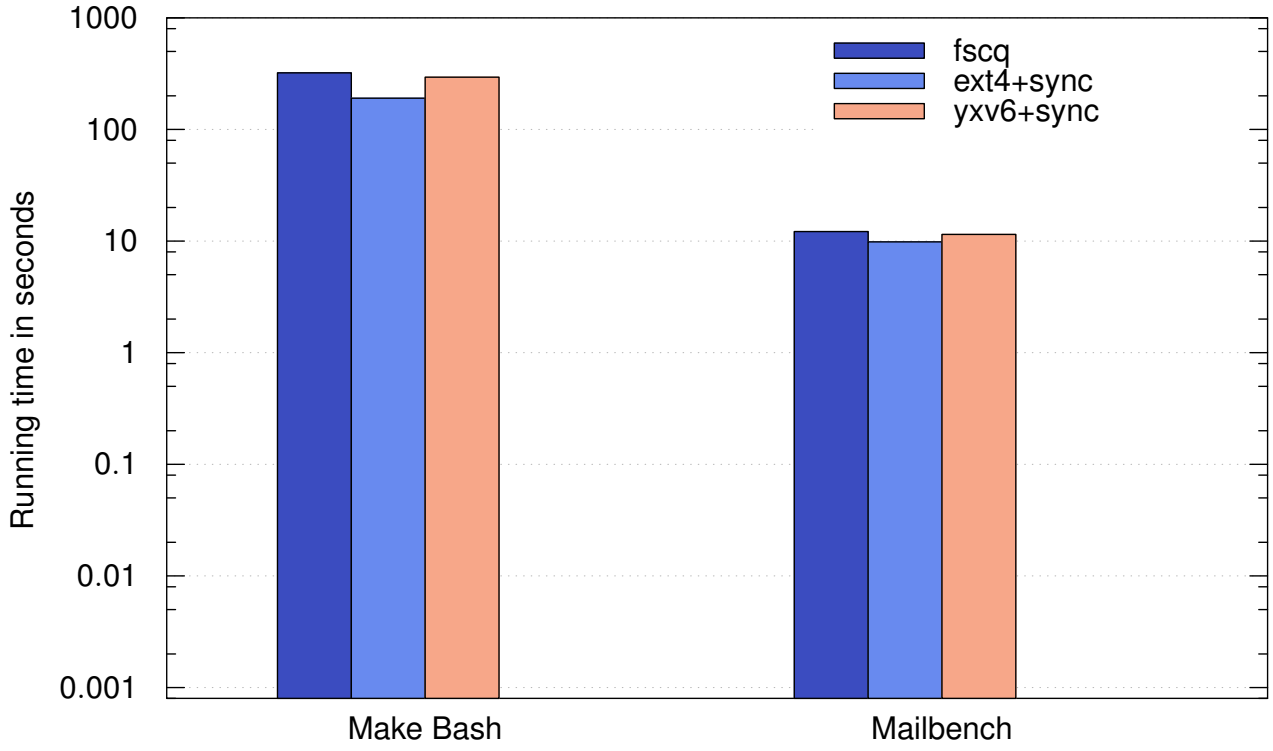
Yxv6 evaluation 3/4: Development effort

- 4 months exploring ways to scale verification
- 2-3 months building Yxv6+sync until self hosting
- Past 6 months: Experiment with optimizations

Yxv6 evaluation 4/4: Runtime performance

- Yxv6+sync similar to FSCQ and ext4+sync

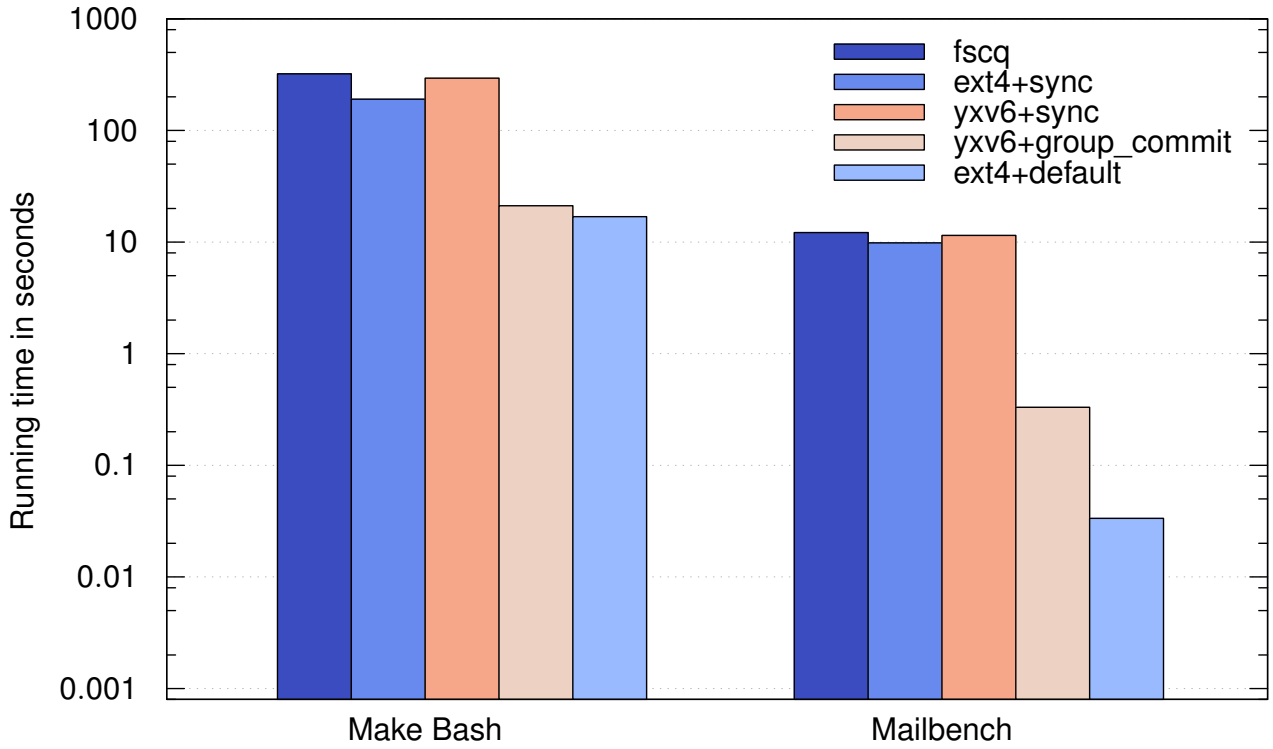
- Yxv6+group_commit
  - 3–150× faster than ext4+sync
  - Within 10× of ext4+default



Yxv6 evaluation 4/4: Runtime performance

- Yxv6+sync similar to FSCQ and ext4+sync

- Yxv6+group_commit
  - 3–150× faster than ext4+sync
  - Within 10× of ext4+default



Yxv6 evaluation 1/4: Verification time revisited

Yxv6 evaluation 1/4: Verification time revisited

- Yxv6+sync takes about a minute to verify
- Yxv6+group_commit
  - Larger log, longer verification time.
  - 1.6 hours using 24 cores

Conclusion

- Push-button verification is feasible for FS
  - No manual proofs on implementation
  - Generate test-cases for bugs
- Design systems to enable automatic, modular verification
- Towards integration of verification into daily development