A Deep Specification for Dropbox

Benjamin C. Pierce
University of Pennsylvania

Clojure/conj
November, 2015

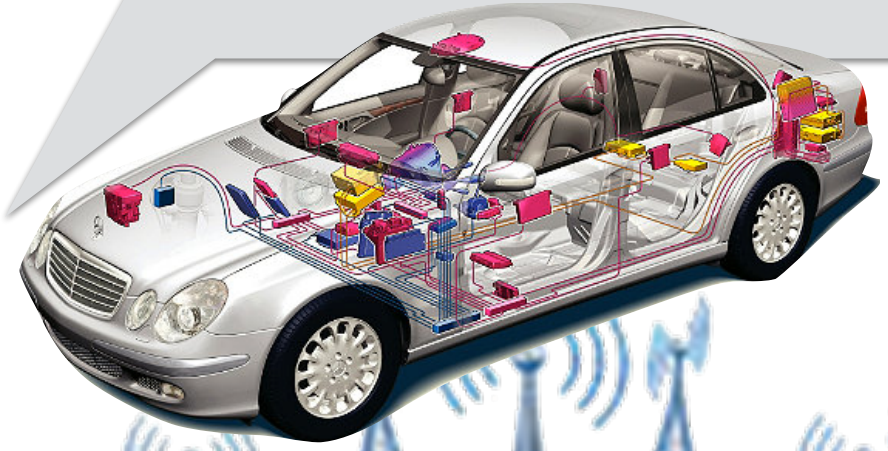
“We can’t build software that works…”

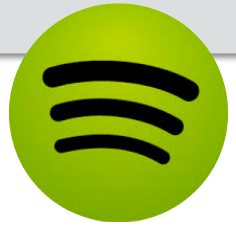
PWNED!

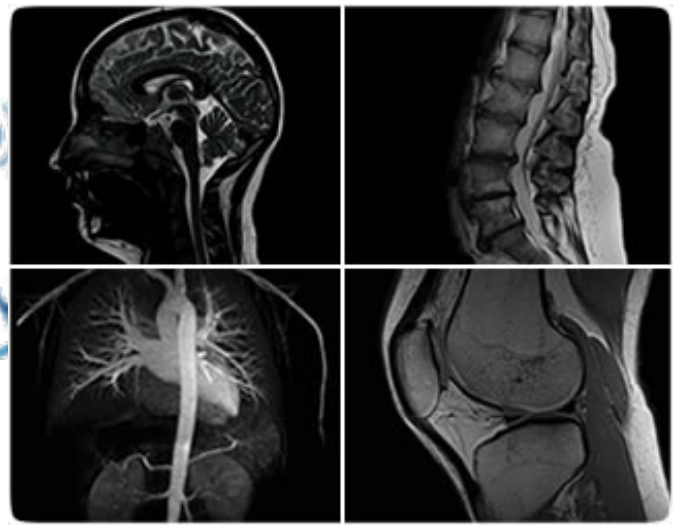
"We can't build software that works…"

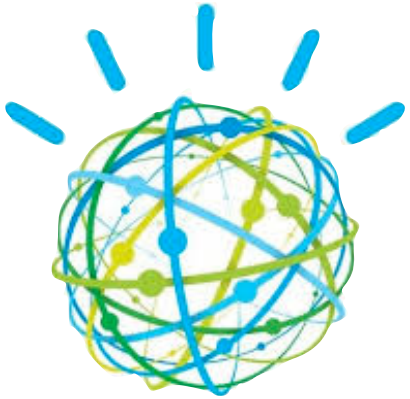
But look at all the software that *does* work!

But look at all the software that *does* work!









How did that happen?

Lots of ways!

Lots of ways!

- Better programming languages
  - Basic *safety guarantees* built in
  - Powerful mechanisms for *abstraction* and *modularity*

Lots of ways!

- Better programming languages
  - Basic *safety guarantees* built in
  - Powerful mechanisms for *abstraction* and *modularity*
- Better software development methodology

Lots of ways!

- Better programming languages
  - Basic *safety guarantees* built in
  - Powerful mechanisms for *abstraction* and *modularity*
- Better software development methodology
- Stable platforms and frameworks

Lots of ways!

- Better programming languages
  - Basic *safety guarantees* built in
  - Powerful mechanisms for *abstraction* and *modularity*
- Better software development methodology
- Stable platforms and frameworks
- Better use of specifications

Lots of ways!

- Better programming languages
  - Basic *safety guarantees* built in
  - Powerful mechanisms for *abstraction* and *modularity*
- Better software development methodology
- Stable platforms and frameworks
- Better use of specifications

I.e., descriptions of what software does (as opposed to how to do it)

Why are specifications useful?

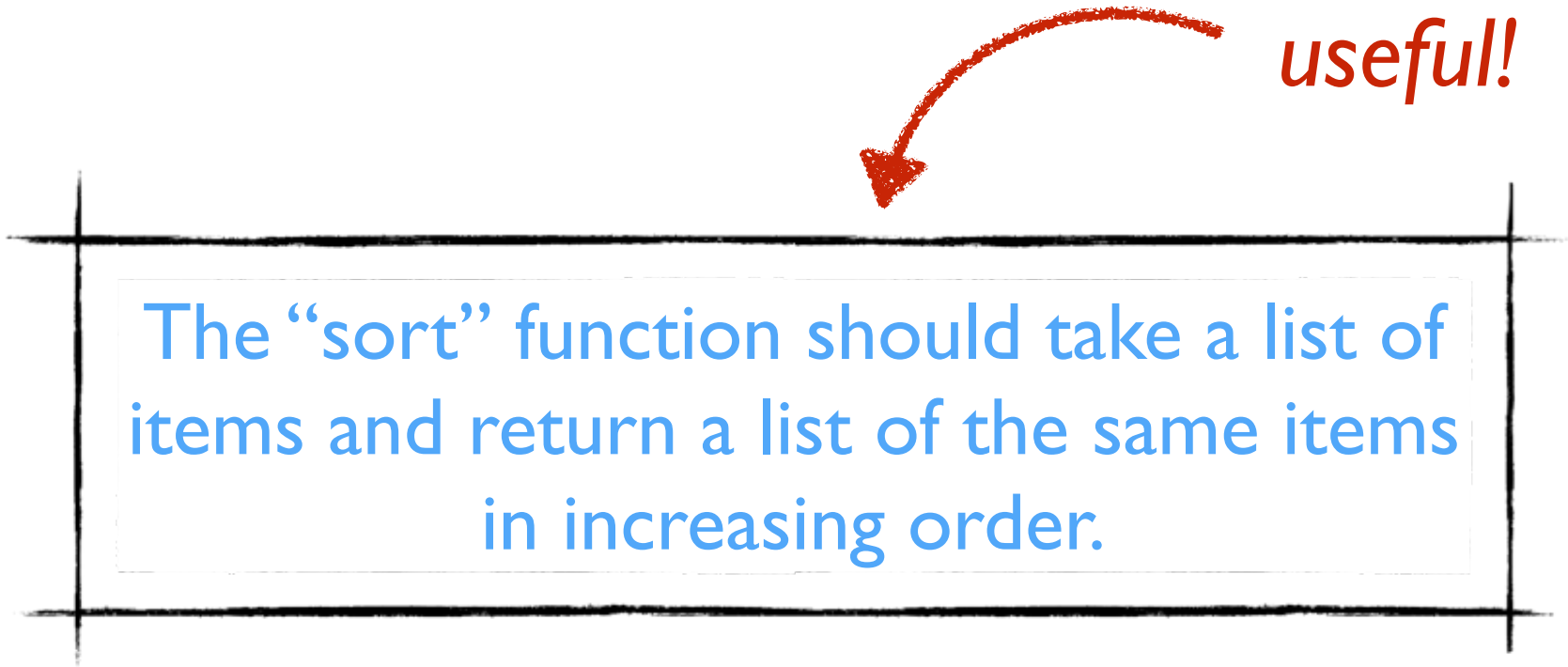
Why are specifications useful?

If you want to build software that works, it is helpful to know what you mean by "works"!

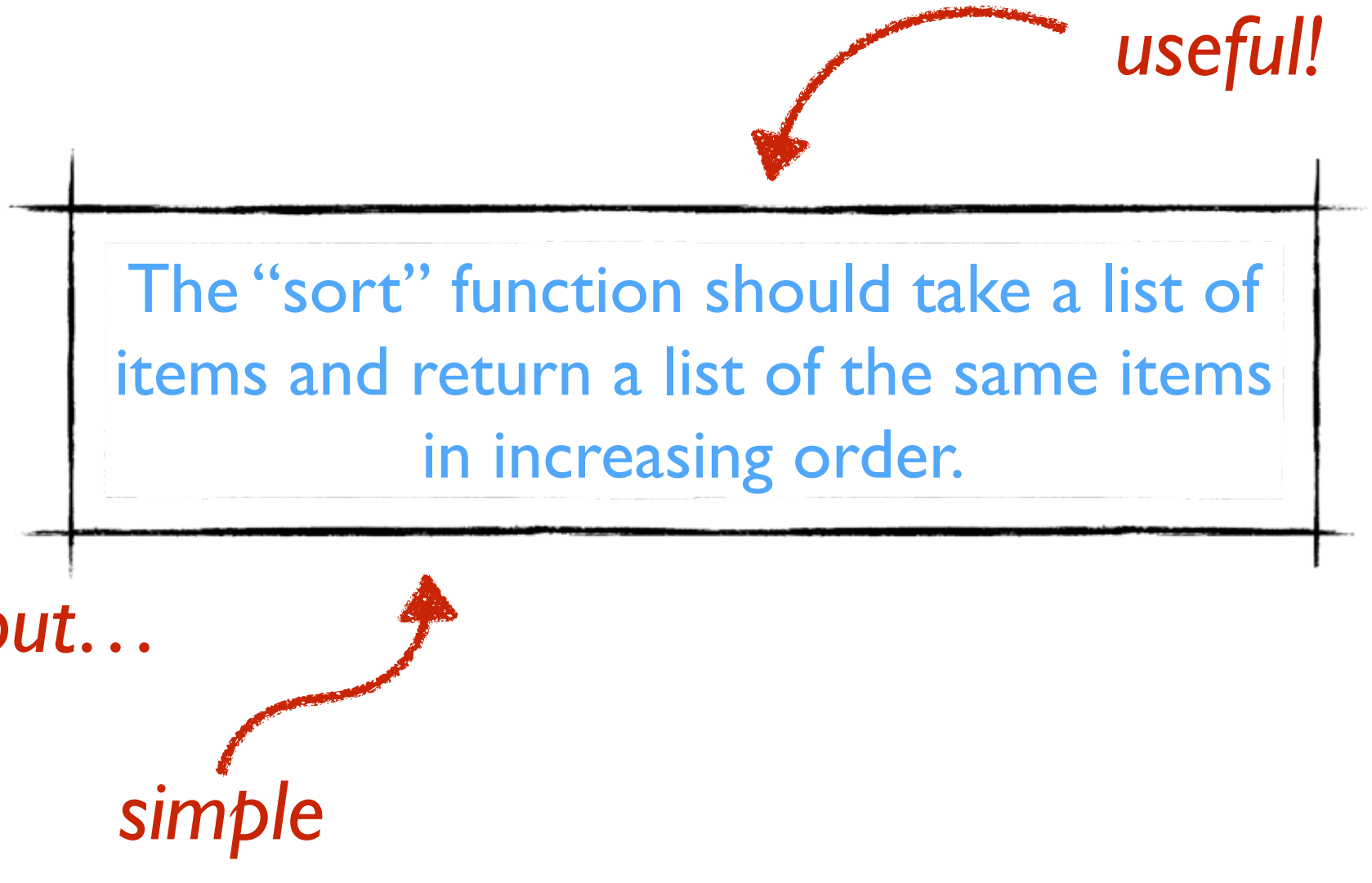
A Specification:

The "sort" function should take a list of items and return a list of the same items in increasing order.

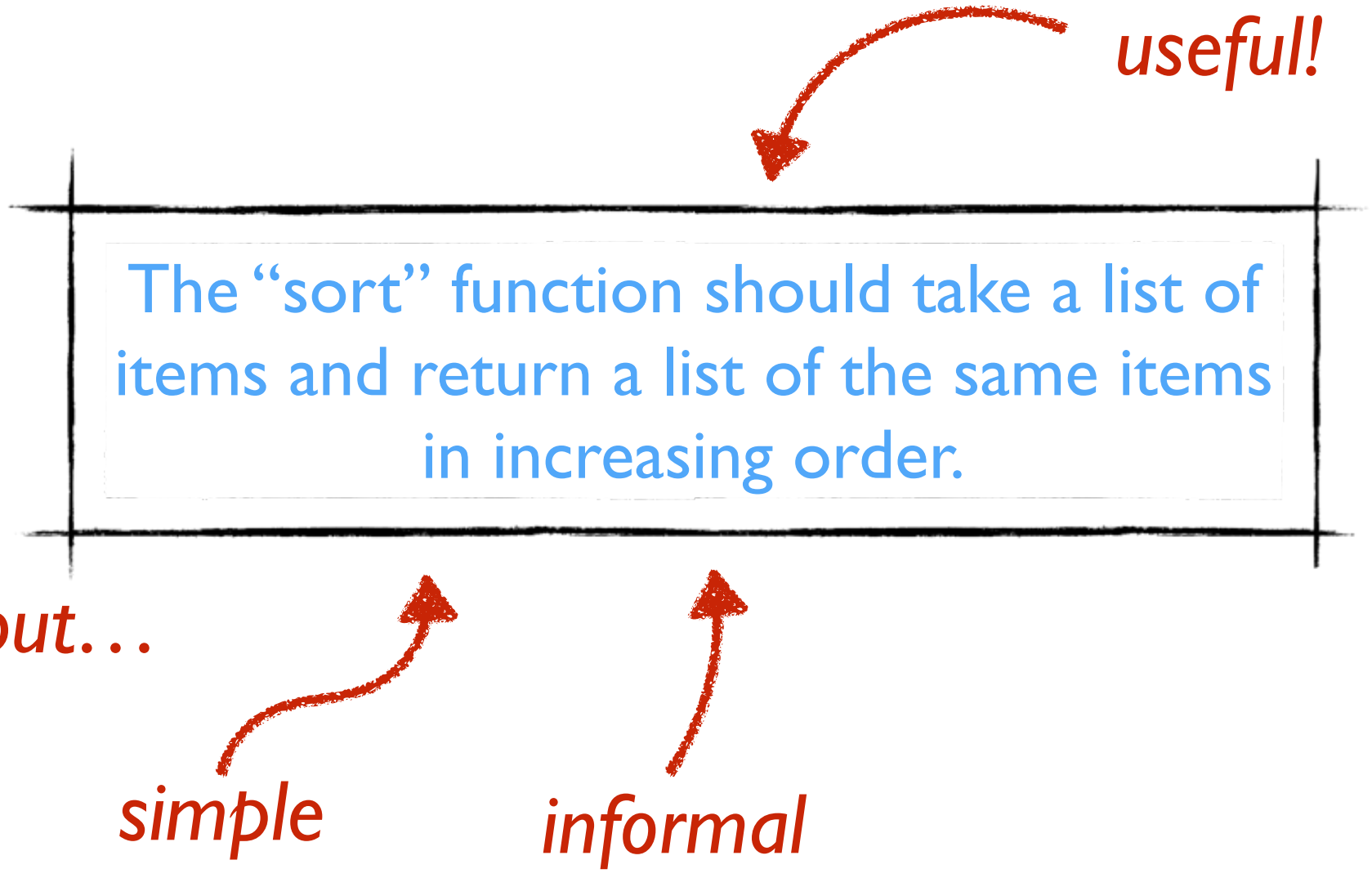
A Specification:



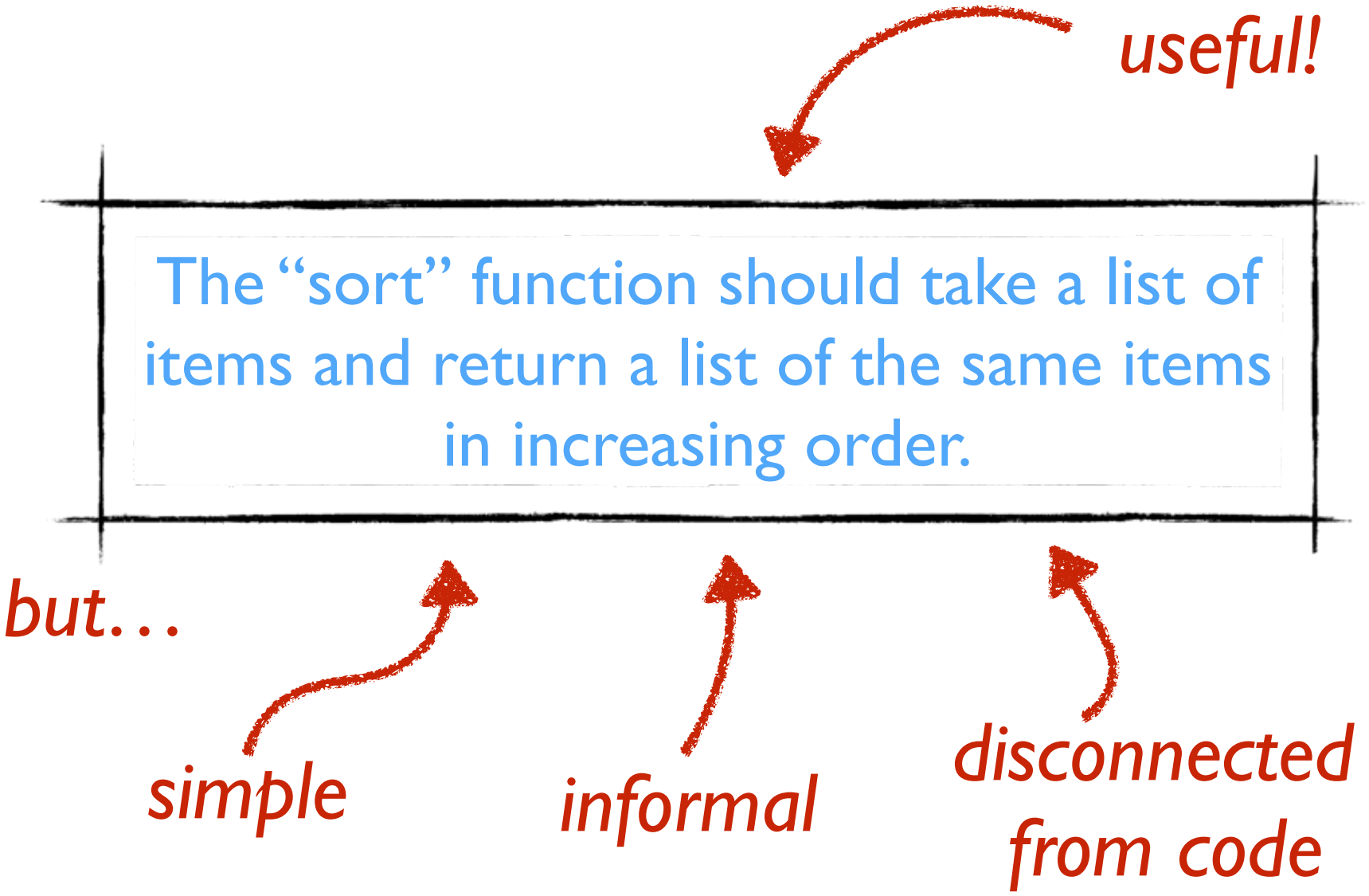
A Specification:

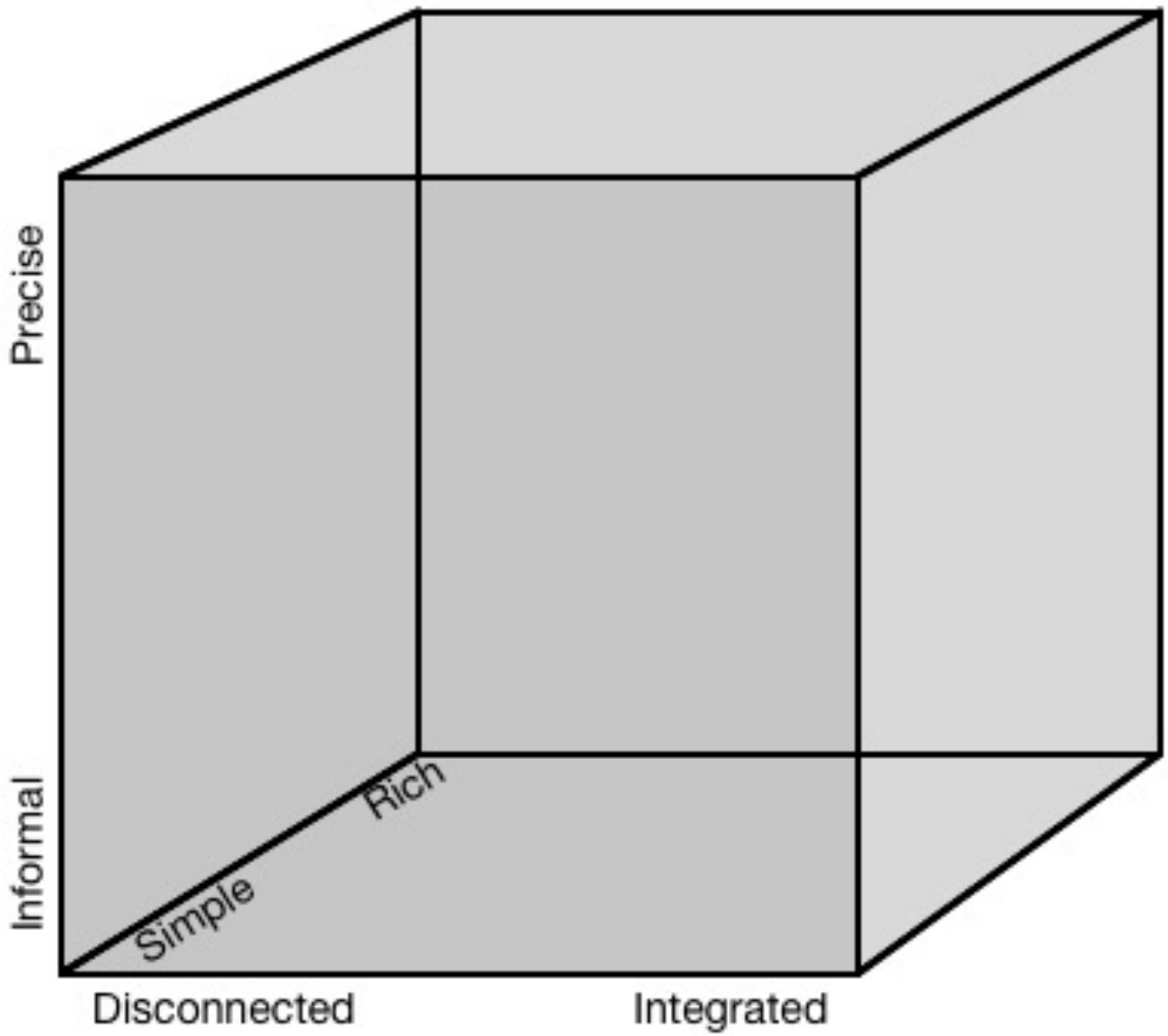


A Specification:



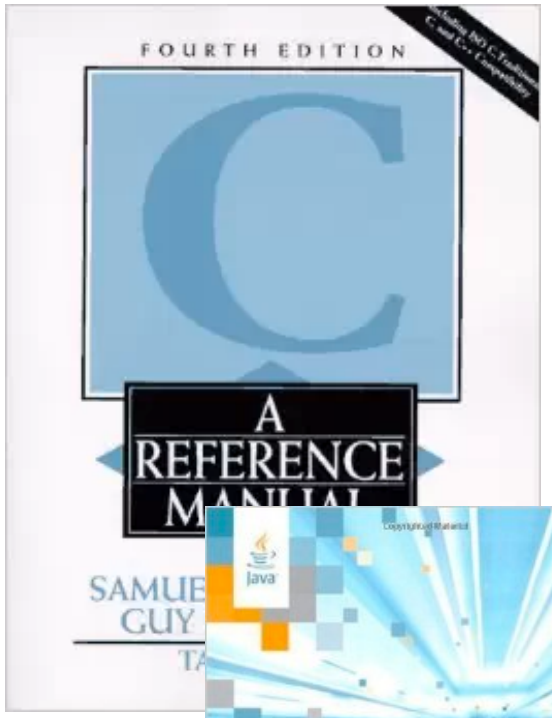
A Specification:




Precise
Informal
Simple
Rich
Disconnected
Integrated

Simple → Rich

- C Language Reference
  - 592 pages
  - also Java (792 pages), C++ (1354 pages, etc.
- x86 CPU reference
  - 1499 pages
- AUTOSAR standardized automotive architecture
  - 3000 pages



AUTOSAR

Informal ⟶ Precise

Formal *specification languages*

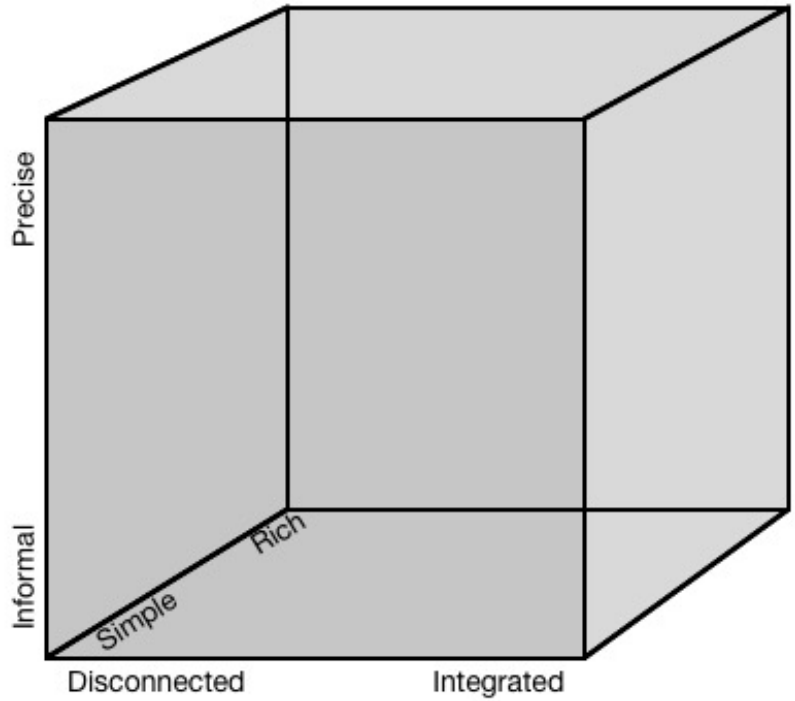
- Z, Alloy, VDM, …
- ACL2
  - x86 instruction set
  - Java virtual machine
- (and many newer ones…)

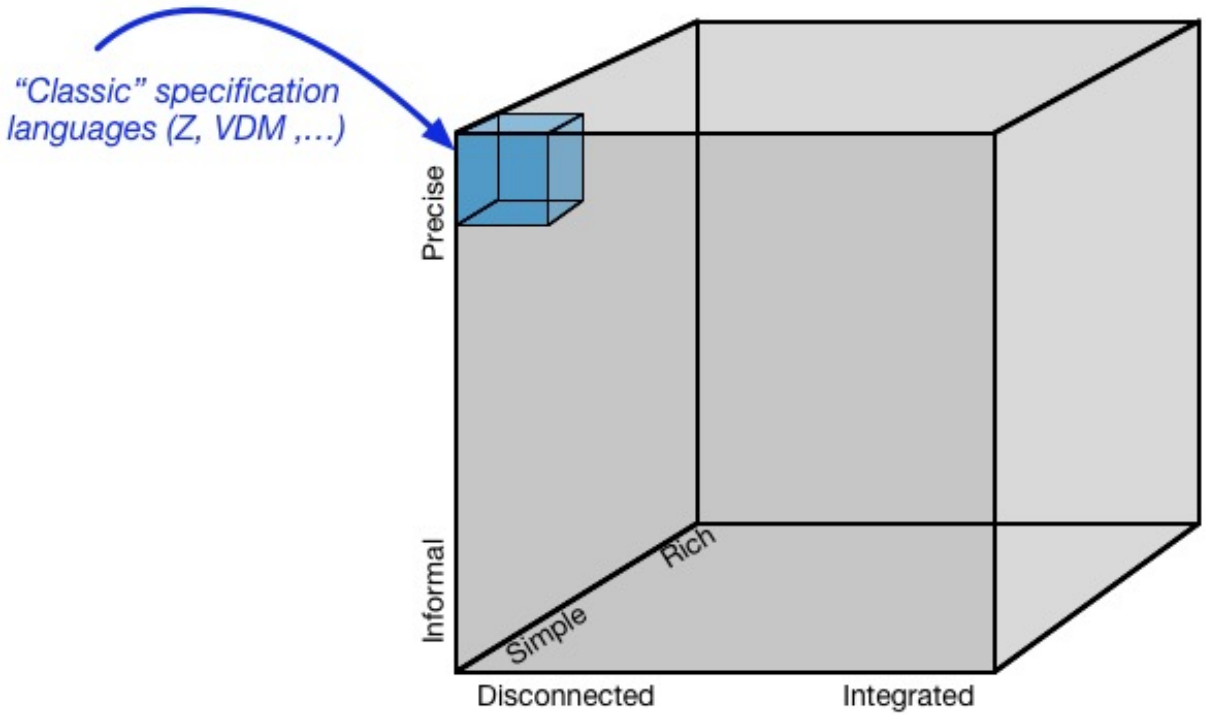
Disconnected ⟶ Integrated

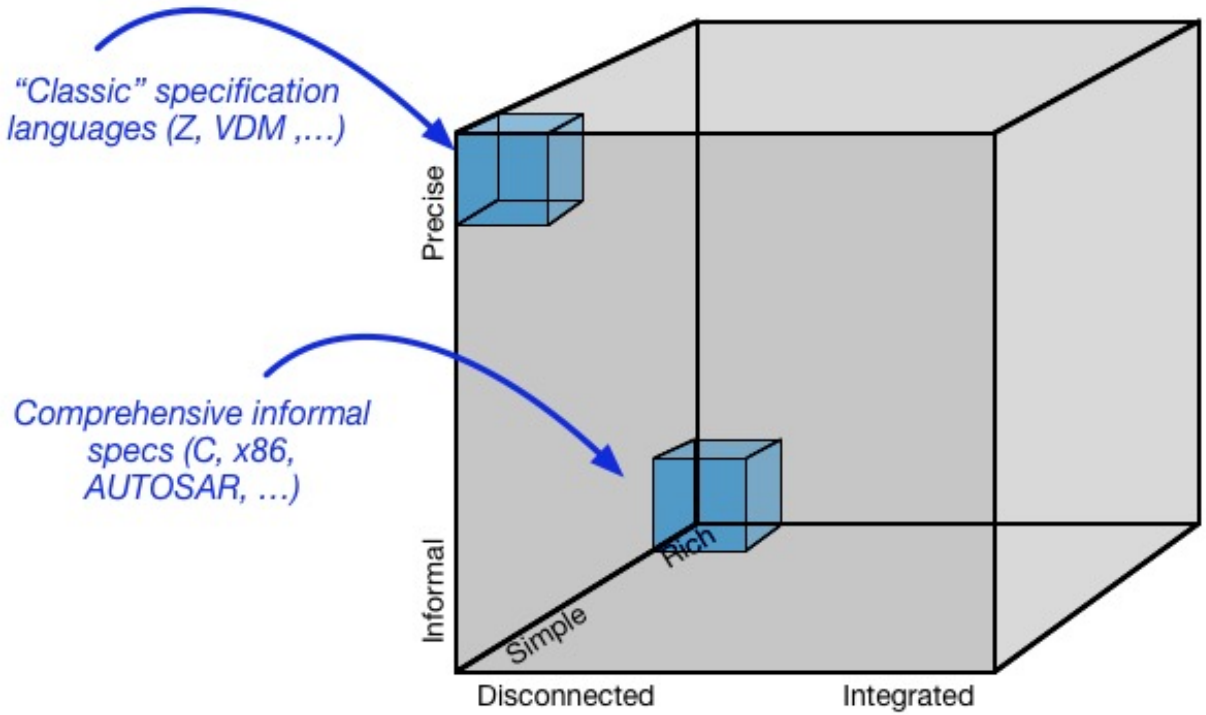
Formal *verification tools*

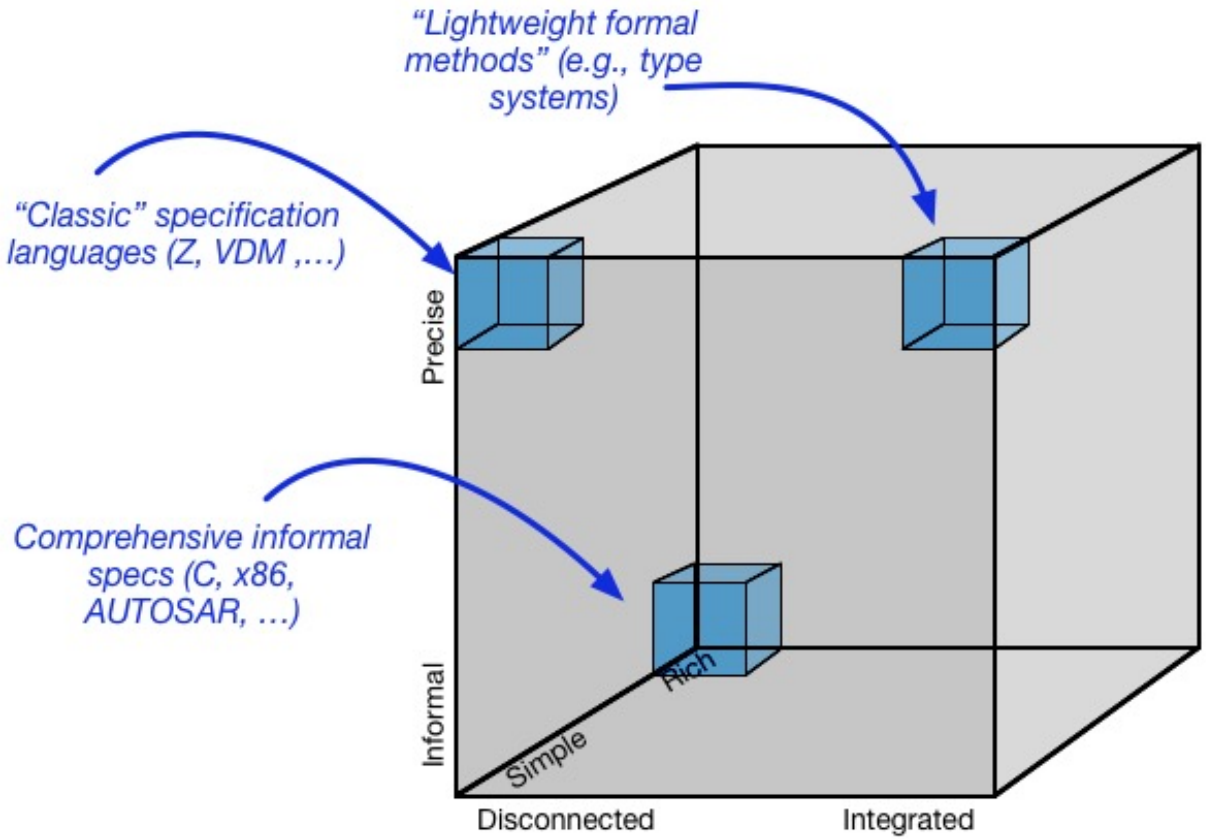
- Human constructs "proof script"; computer checks it
- Capable in principle of establishing connections between arbitrary specifications and code
- Challenging to use at scale, but getting better!

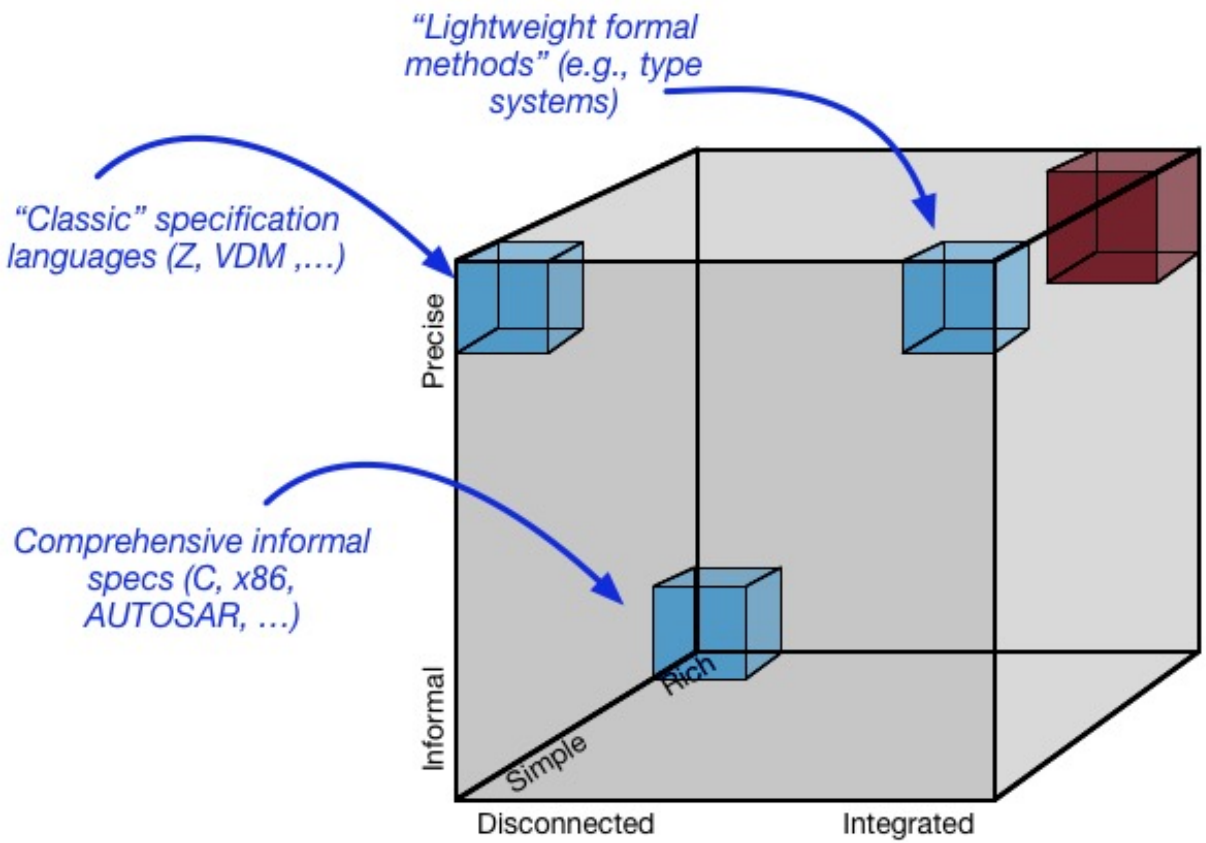
Recap…

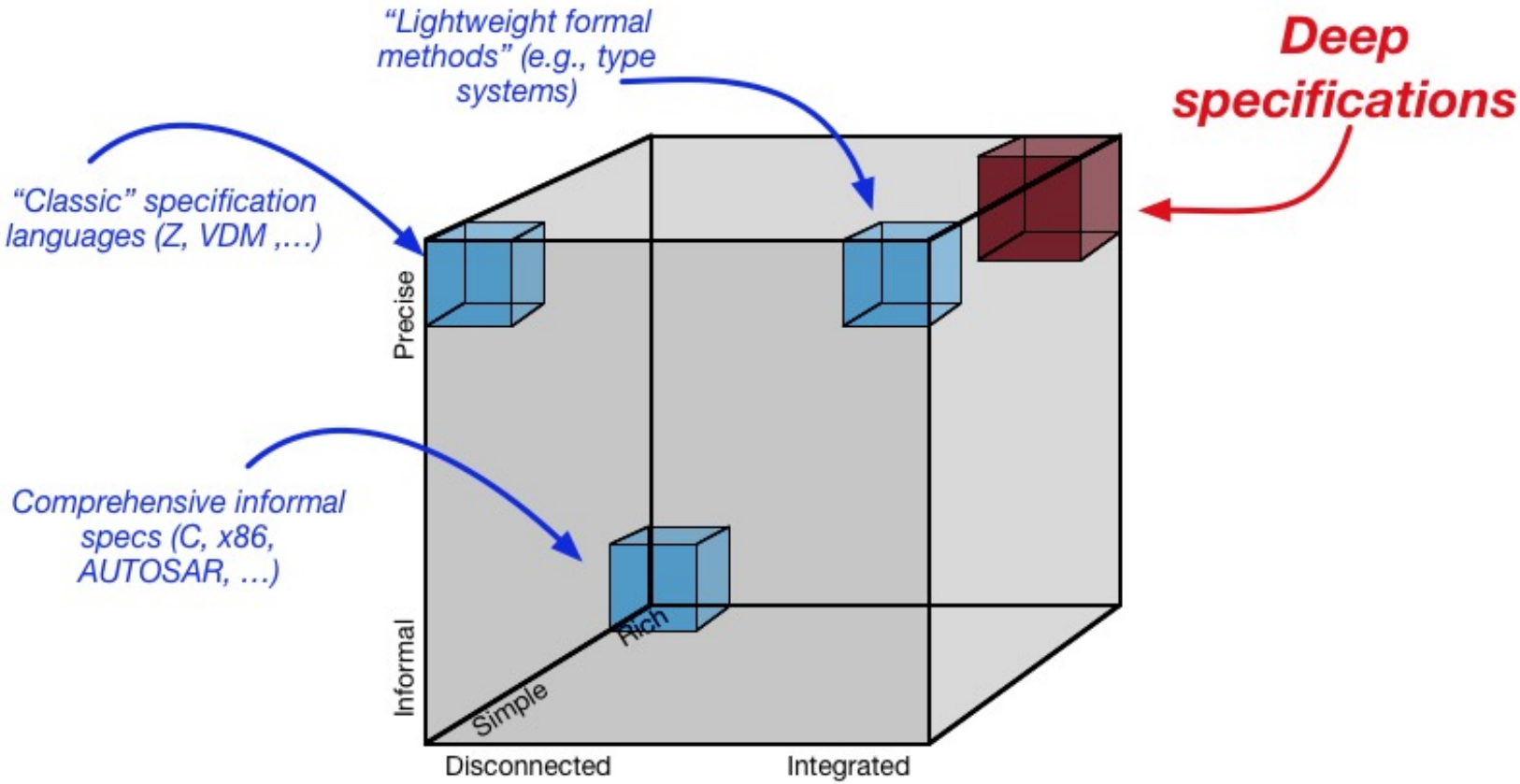



"Classic" specification languages (Z, VDM ,…)
Precise
Informal
Simple
Rich
Disconnected
Integrated


"Classic" specification languages (Z, VDM ,…)
Comprehensive informal specs (C, x86, AUTOSAR, …)
Precise
Informal
Simple
Rich
Disconnected
Integrated


"Lightweight formal methods" (e.g., type systems)
"Classic" specification languages (Z, VDM ,…)
Comprehensive informal specs (C, x86, AUTOSAR, …)
Precise
Informal
Simple
Rich
Disconnected
Integrated


"Lightweight formal methods" (e.g., type systems)
"Classic" specification languages (Z, VDM ,…)
Comprehensive informal specs (C, x86, AUTOSAR, …)
Precise
Informal
Rich
Simple
Disconnected
Integrated

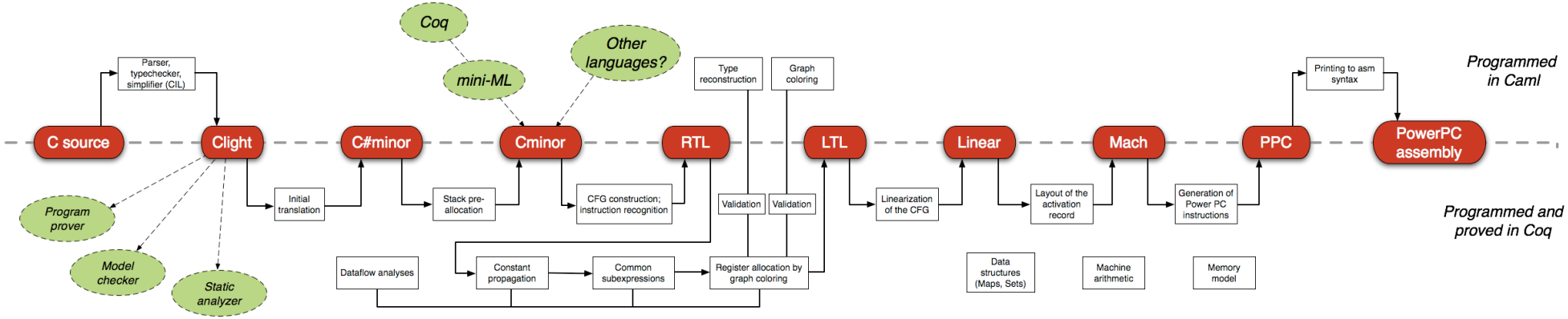

"Lightweight formal methods" (e.g., type systems)
Deep specifications
"Classic" specification languages (Z, VDM ,…)
Comprehensive informal specs (C, x86, AUTOSAR, …)
Precise
Informal
Rich
Simple
Disconnected
Integrated

Deep specifications

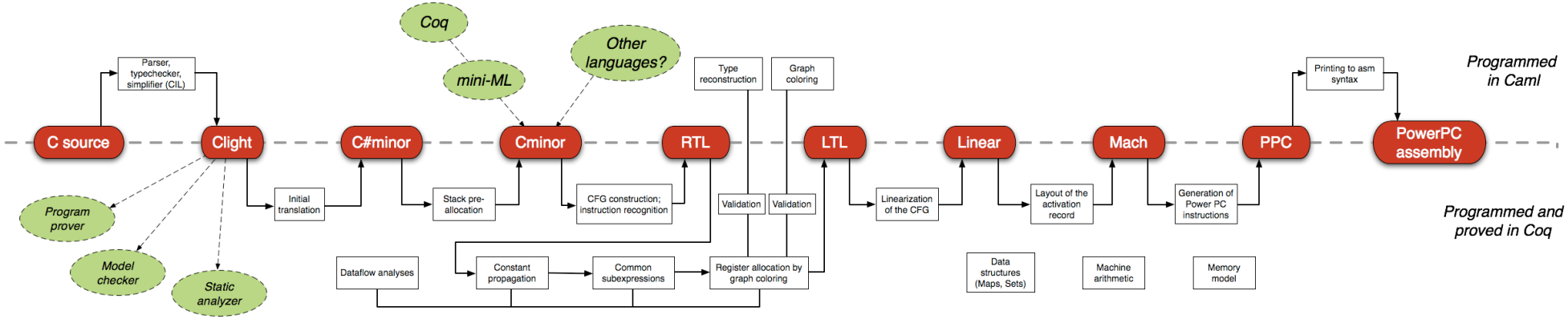
Deep specifications

1. Rich
2. Formal
3. Integrated with code

CompCert C compiler

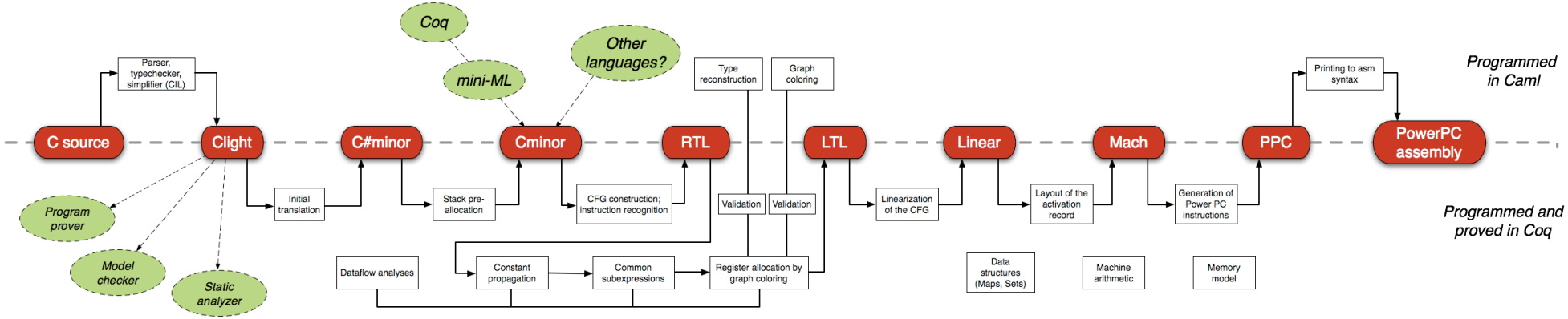


CompCert C compiler



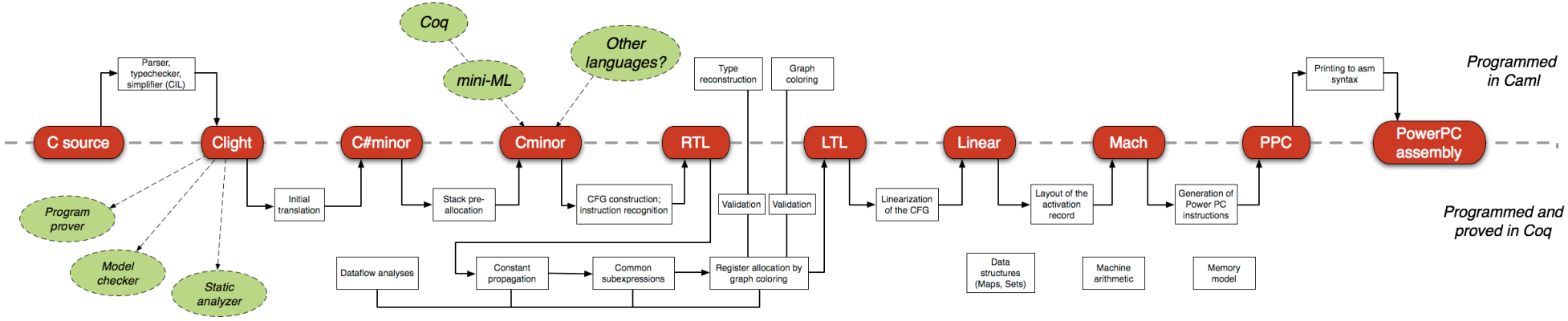
- Accepts most of the ISO C 99 language

CompCert C compiler



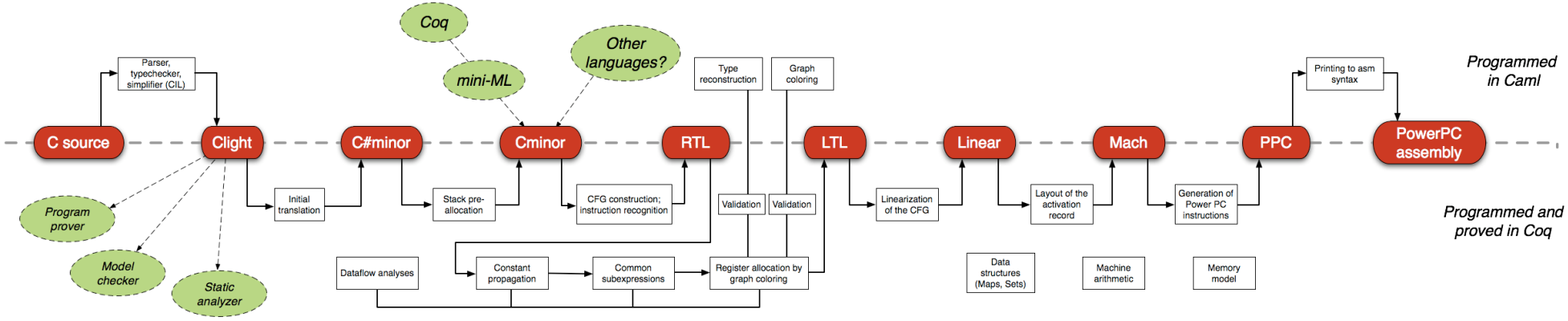
- Accepts most of the ISO C 99 language
- Produces machine code for PowerPC, ARM, and IA32 (x86 32-bit) architectures

CompCert C compiler



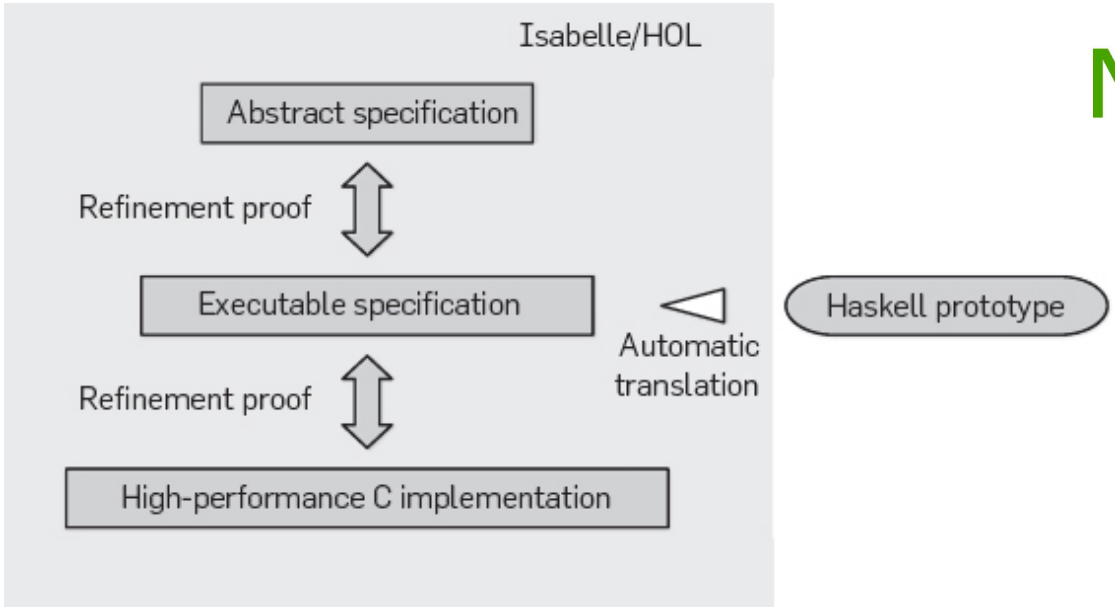
- Accepts most of the ISO C 99 language
- Produces machine code for PowerPC, ARM, and IA32 (x86 32-bit) architectures
- 90% of the performance of GCC (v4, opt. level 1)

CompCert C compiler



- Accepts most of the ISO C 99 language
- Produces machine code for PowerPC, ARM, and IA32 (x86 32-bit) architectures
- 90% of the performance of GCC (v4, opt. level 1)
- Fully verified

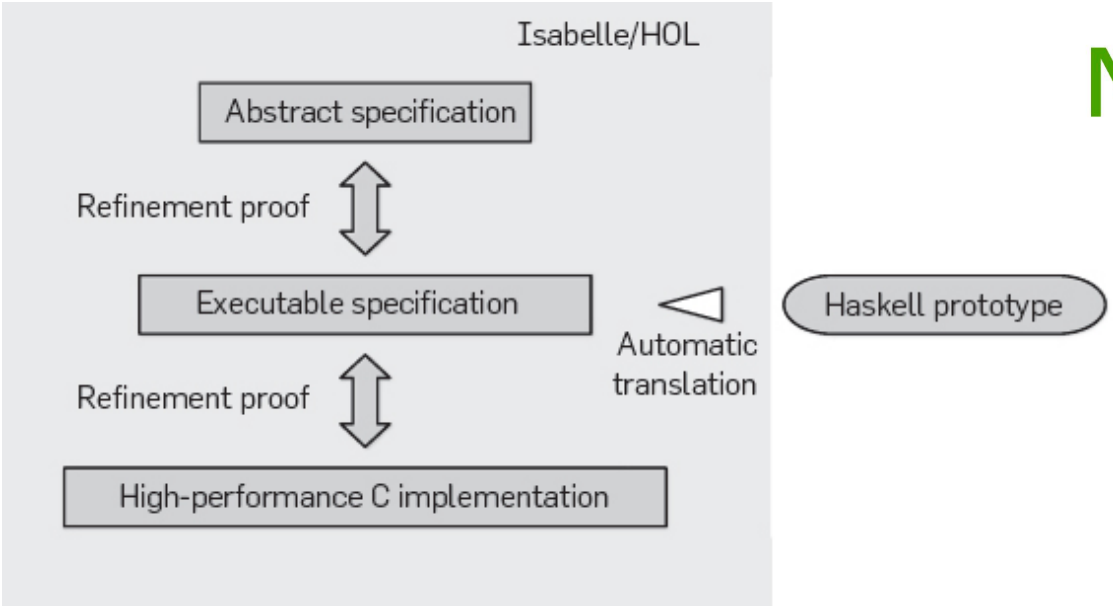
seL4


Isabelle/HOL
Abstract specification
Refinement proof
Executable specification
Automatic translation
Haskell prototype
Refinement proof
High-performance C implementation

NICTA

seL4

NICTA



- Real-world operating-system kernel with an end-to-end proof of implementation correctness and security enforcement

New tools

- Coq
- Isabelle
- ACL2
- …

Powerful
proof assistants and
program logics

- F*
- Dafny
- Boogie
- …

Mostly automatic verifiers
based on SMT solvers

Formal verification of real software

- Verified compilers
  - CompCertTSO, CakeML, Bedrock,…
- Verified operating systems
  - CertiKOS, Ironclad Apps, Jitk, …
- Verified filesystems
  - Fscq, …
- Verified distributed systems
  - Verdi, …
- Verified cryptographic algorithms and protocols
  - SHA, TLS, …

What's happening now?

What's happening now?

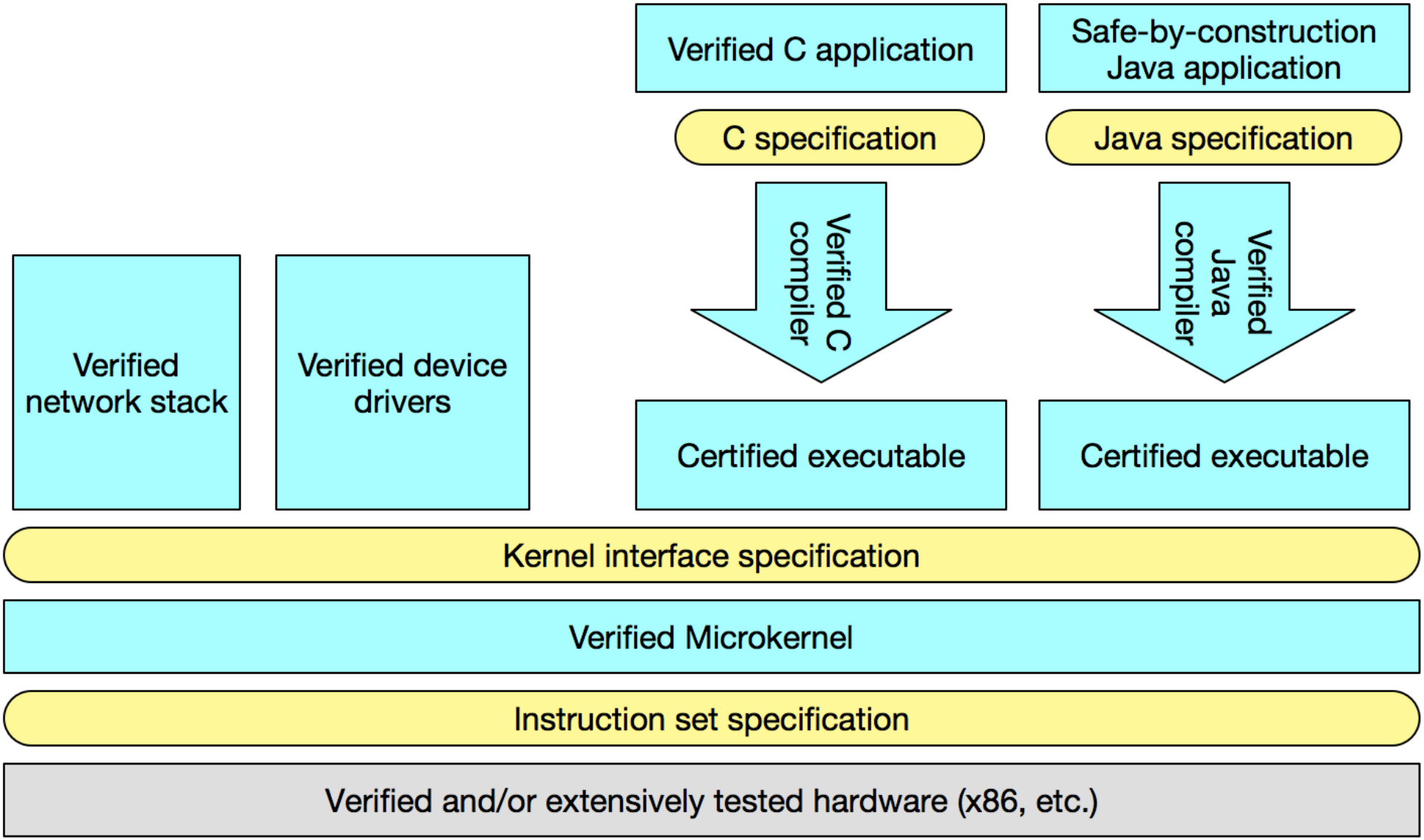
Stephanie Weirich
University of Pennsylvania

Steve Zdancewic
University of Pennsylvania

Andrew Appel
Princeton

Zhong Shao
Yale

Adam Chlipala
MIT



A zero-vulnerability software stack

"But what if I don't *want* to do formal verification?"

Expressive type systems

Classical type systems:

- Highly successful "lightweight formal methods"
- Designed into programming languages, not separate tools
- Limited expressiveness, but "always on" security types

New developments:

- Component types / module systems
- Generalized abstract datatypes
- Session types
- Lightweight dependent types
- …

“But what if I don’t *like* types?”

Another way to use specifications

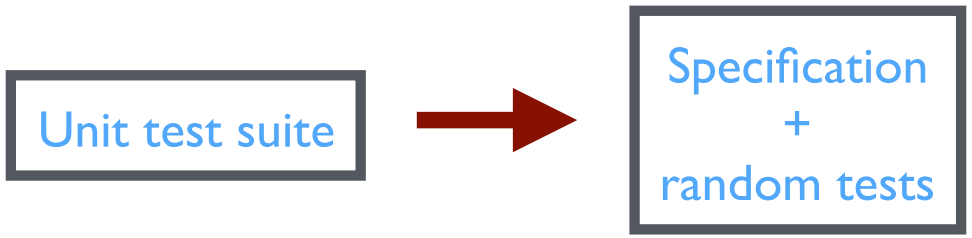
Idea: Use *random testing* to quickly check correspondence between systems and specs

- Good for debugging both code and spec!
- Pretty Good Assurance™ for cheap!

Specification-based random testing

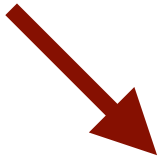
Key ideas

- Write specification as a set of **executable** properties
- Generate many **random inputs** and check whether properties return True
- When a counterexample is found, "**shrink**" it to produce a minimal failing case



Unit test suite

```
sort [1,2,3] => [1,2,3]
sort [3,2,1] => [1,2,3]
sort [] => []
sort [1] => [1]
sort [2,1,3,2] => [1,2,2,3]
...
```



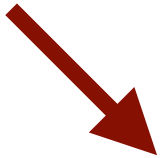
Specification

```
prop_ordered xs = ordered (sort xs)
    where ordered []         =  True
          ordered [x]        =  True
          ordered (x:y:xs)   =  x <= y && ordered (y:xs)

prop_permutation xs = permutation xs (sort xs)
    where permutation xs ys = null (xs \\ ys) && null (ys \\ xs)
```

Unit test suite

```
sort [1,2,3] => [1,2,3]
sort [3,2,1] => [1,2,3]
sort [] => []
sort [1] => [1]
sort [2,1,3,2] => [1,2,2,3]
...
```



Specification

```
prop_ordered xs = ordered (sort xs)
    where ordered []         =  True
          ordered [x]        =  True
          ordered (x:y:xs)   =  x <= y && ordered (y:xs)

prop_permutation xs = permutation xs (sort xs)
    where permutation xs ys = null (xs \\ ys) && null (ys \\ xs)
```

QuickCheck

1999—invented by Koen Claessen and John Hughes, for Haskell

2006—Quviq founded, marketing Erlang version

Many extensions, ports to many other languages (including test.check in **Clojure**! :-)

Finding deep bugs for Ericsson, Volvo Cars, Basho, etc…

A Deep Specification for Dropbox

A Deep Specification for Dropbox

with

John Hughes
Thomas Arts
QuviQ AB

Why specify Dropbox?

Many synchronization services…

- Dropbox, Google Drive, OneDrive, Owncloud, SpiderOak, Sugarsync, Box.net, Seafile, Pulse, Wuala, Teamdrive, Cloudme, Cx, Amazon cloud service, …

…with *many* users…

- Dropbox: >400M
- Google Drive, MS OneDrive: >240M

…executing complex distributed algorithms over large amounts of precious data

Many synchronization services…

- Dropbox, Google Drive, OneDrive, Owncloud, SpiderOak, Sugarsync, Box.net, Seafile, Pulse, Wuala, Teamdrive, Cloudme, Cx, Amazon cloud service, …

…with *many* users…

- Dropbox: >400M
- Google Drive, MS OneDrive: >240M

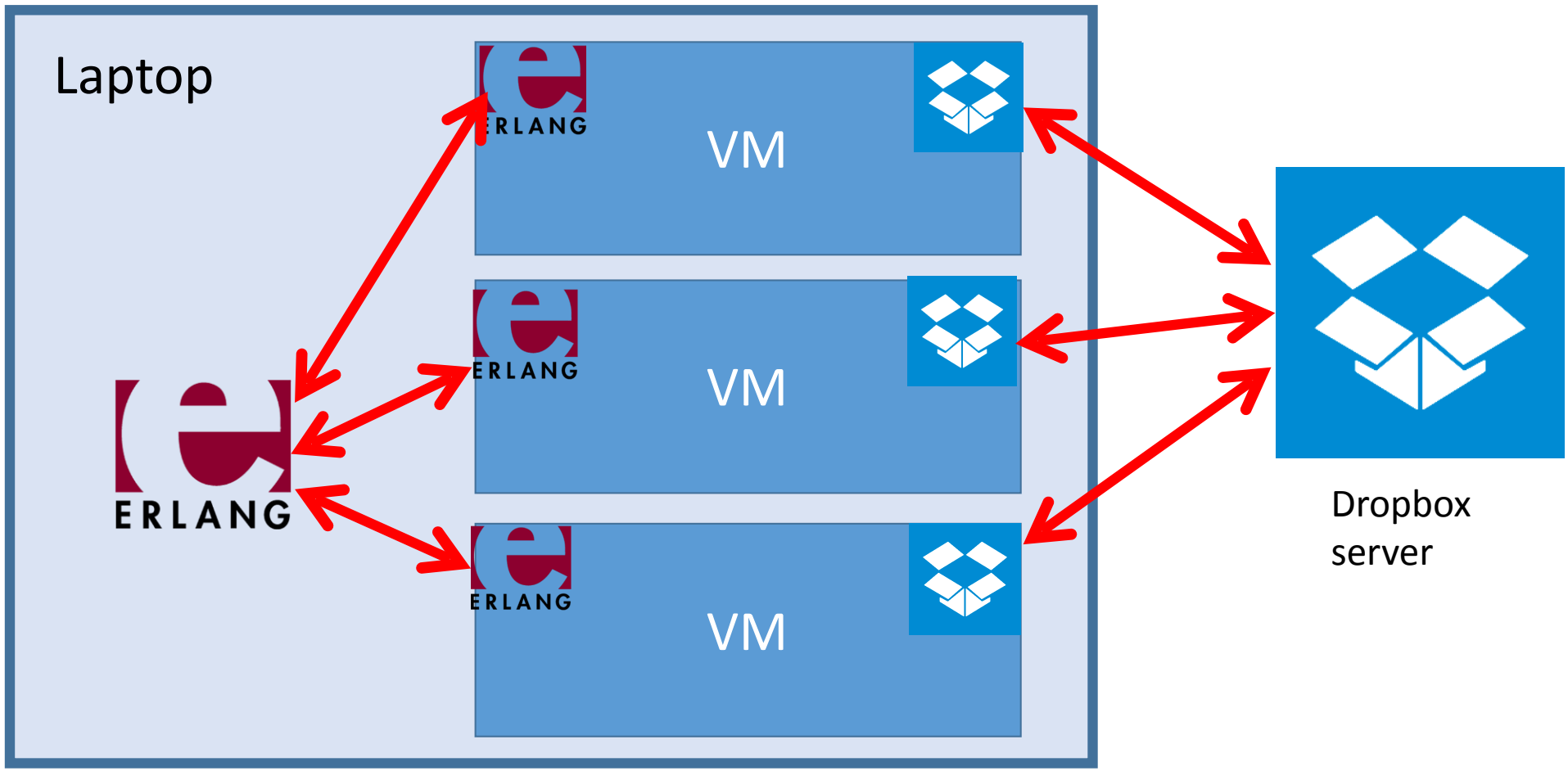
…executing complex distributed algorithms over large amounts of precious data

What could go wrong…?

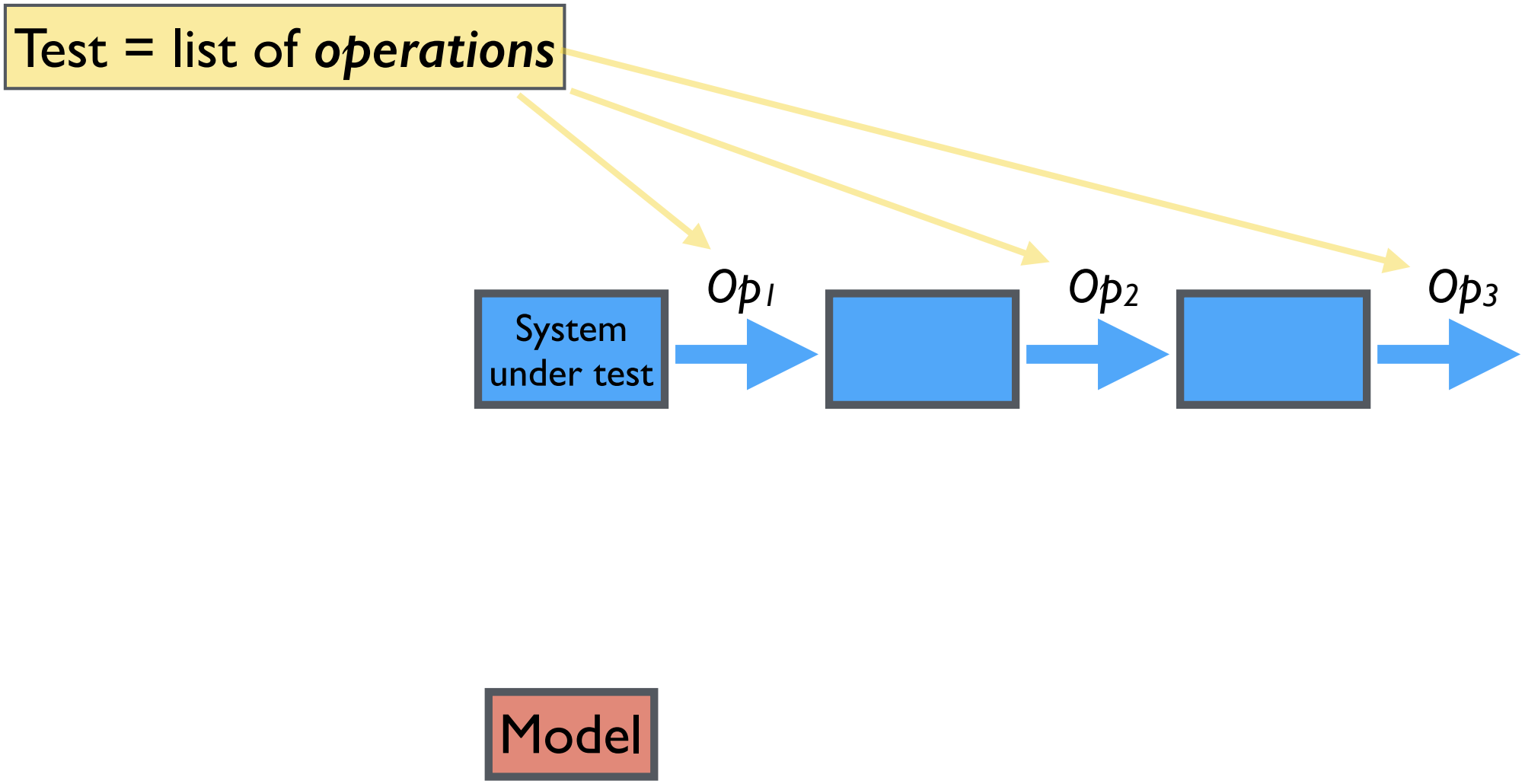
Goals

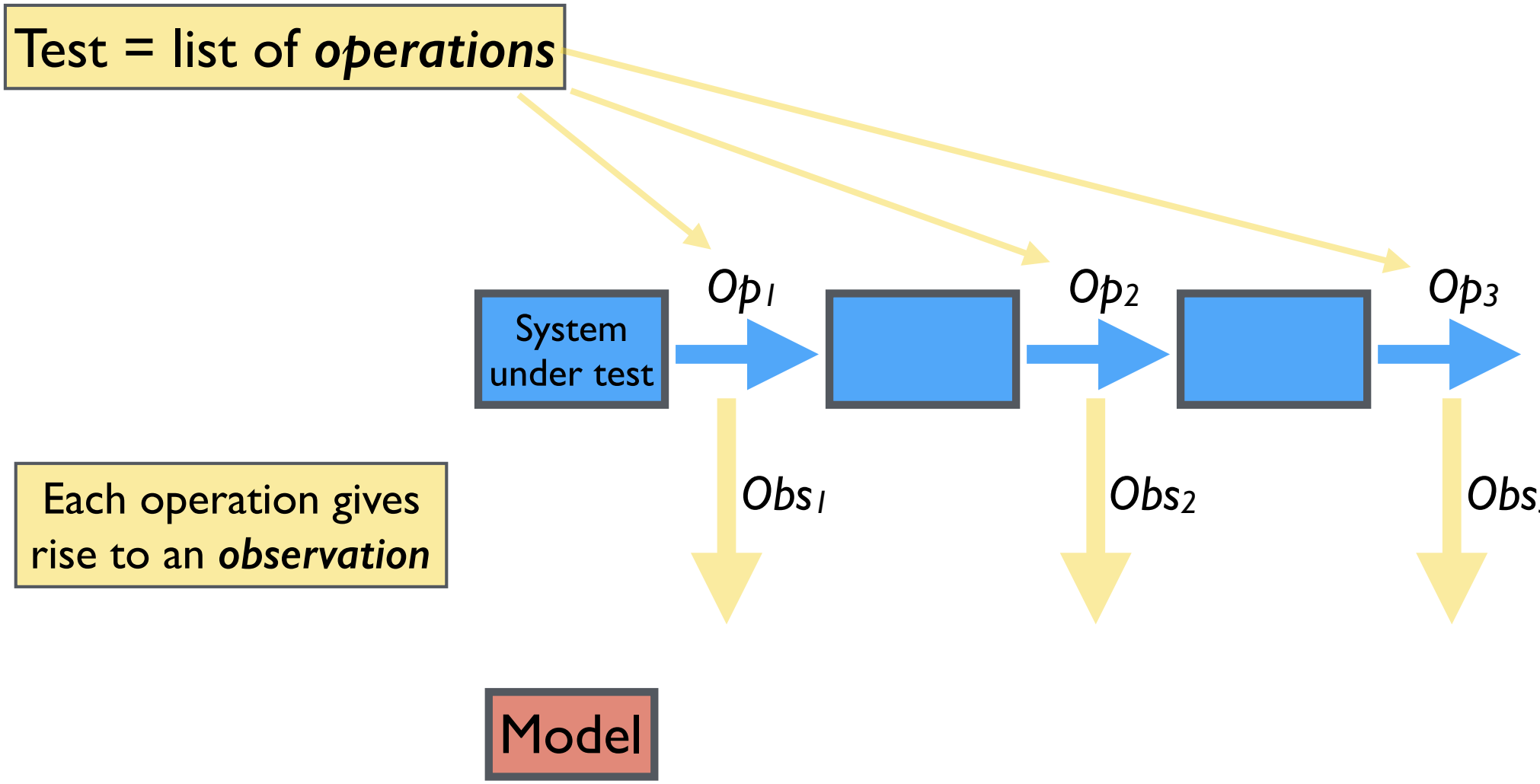
- Give a *precise specification* of the core behavior of a synchronization service
  - Phrased from the perspective of *users*
  - Applicable to a variety of different synchronizers
- *Validate* it against Dropbox's observed behavior
  - Using Erlang QuickCheck

Test Setup

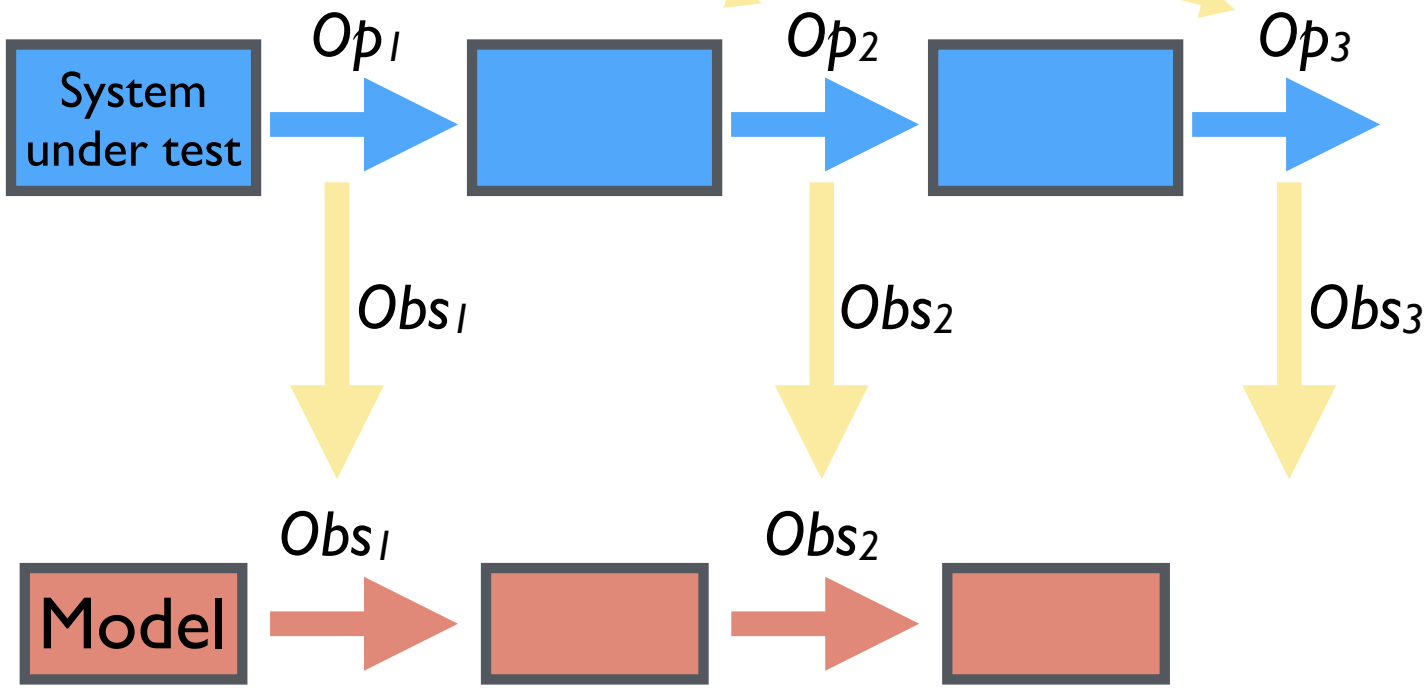



System
under test
Model


Test = list of *operations*
System under test
Op_1
Op_2
Op_3
Model

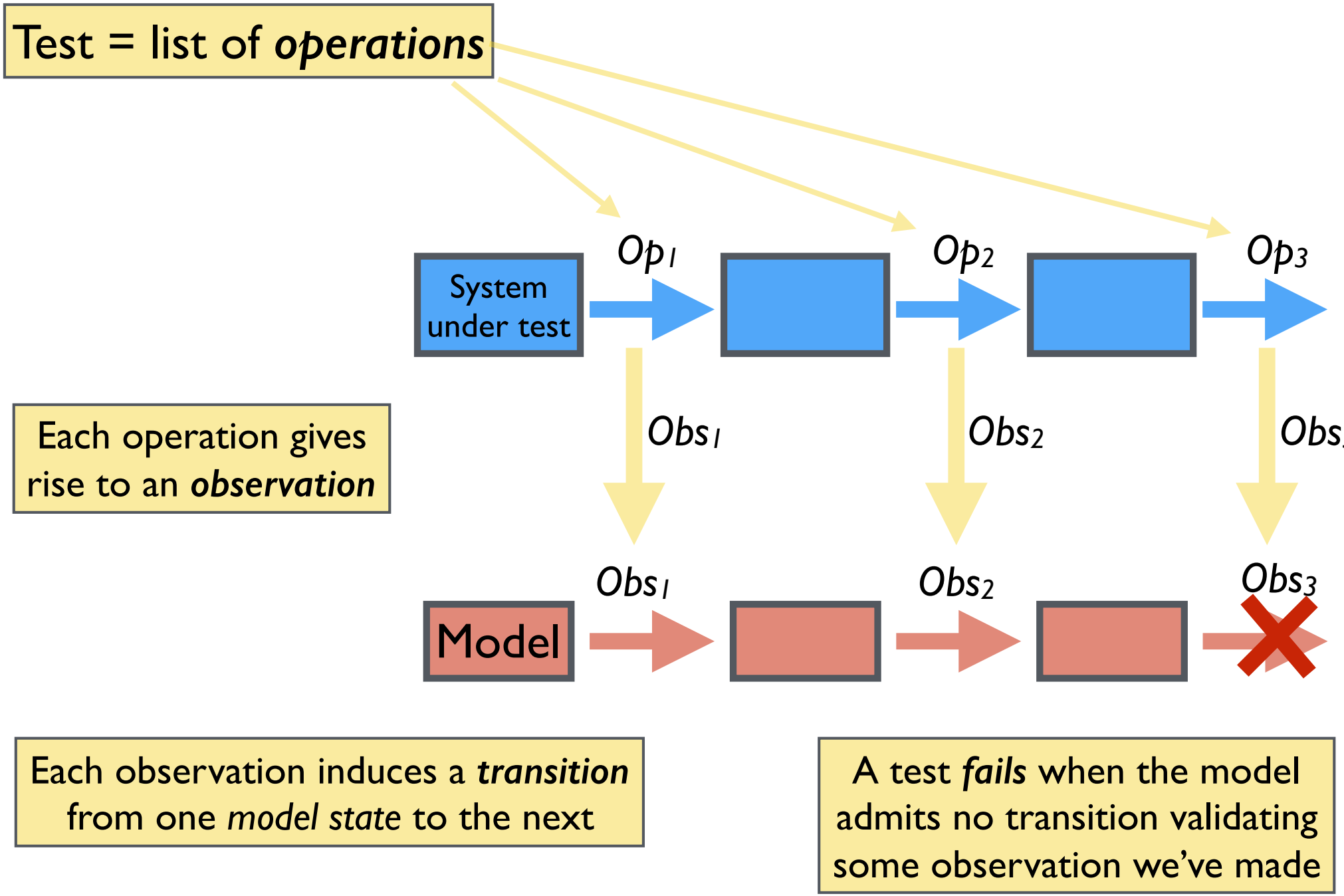

Test = list of *operations*
System under test
Op_1
Op_2
Op_3
Obs_1
Obs_2
Obs_3
Each operation gives rise to an *observation*
Model

Test = list of ***operations***



Each operation gives rise to an ***observation***

Each observation induces a ***transition*** from one *model state* to the next


Test = list of *operations*
System under test
Op_1
Op_2
Op_3
Each operation gives rise to an ***observation***
Obs_1
Obs_2
Obs_3
Model
Obs_1
Obs_2
Obs_3
Each observation induces a ***transition*** from one *model state* to the next
A test ***fails*** when the model admits no transition validating some observation we've made

Basic Specification

If $Op_1 \ldots Op_n$ is some sequence of operations and $Obs_1 \ldots Obs_n$ are the observations we make when we run them, then

is a valid sequence of transitions of the model.

“What operations and observations do we need?”

First try…

First try…

Operations	Observations
READ$_N$	READ$_N$ ⟶ "*current value*"

First try…

Operations	Observations
READ$_N$	READ$_N$ ⟶ "*current value*"
WRITE$_N$ ("*new value*")	WRITE$_N$ ("*new value*") ⟶ "*old value*"

First try…

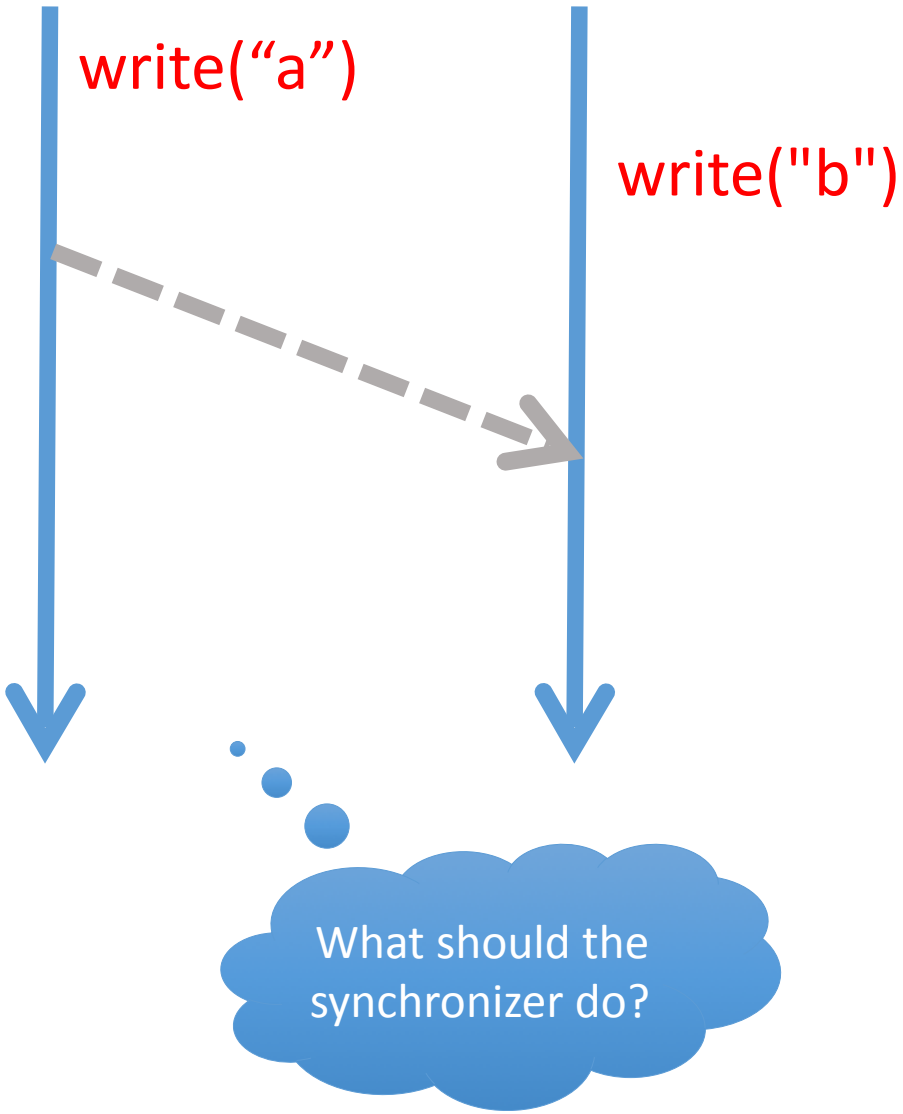
Operations	Observations
READ$_N$	READ$_N$ ⟶ "*current value*"
WRITE$_N$ ("*new value*")	WRITE$_N$ ("*new value*") ⟶ "*old value*"

Use special value ⊥ for "no file"

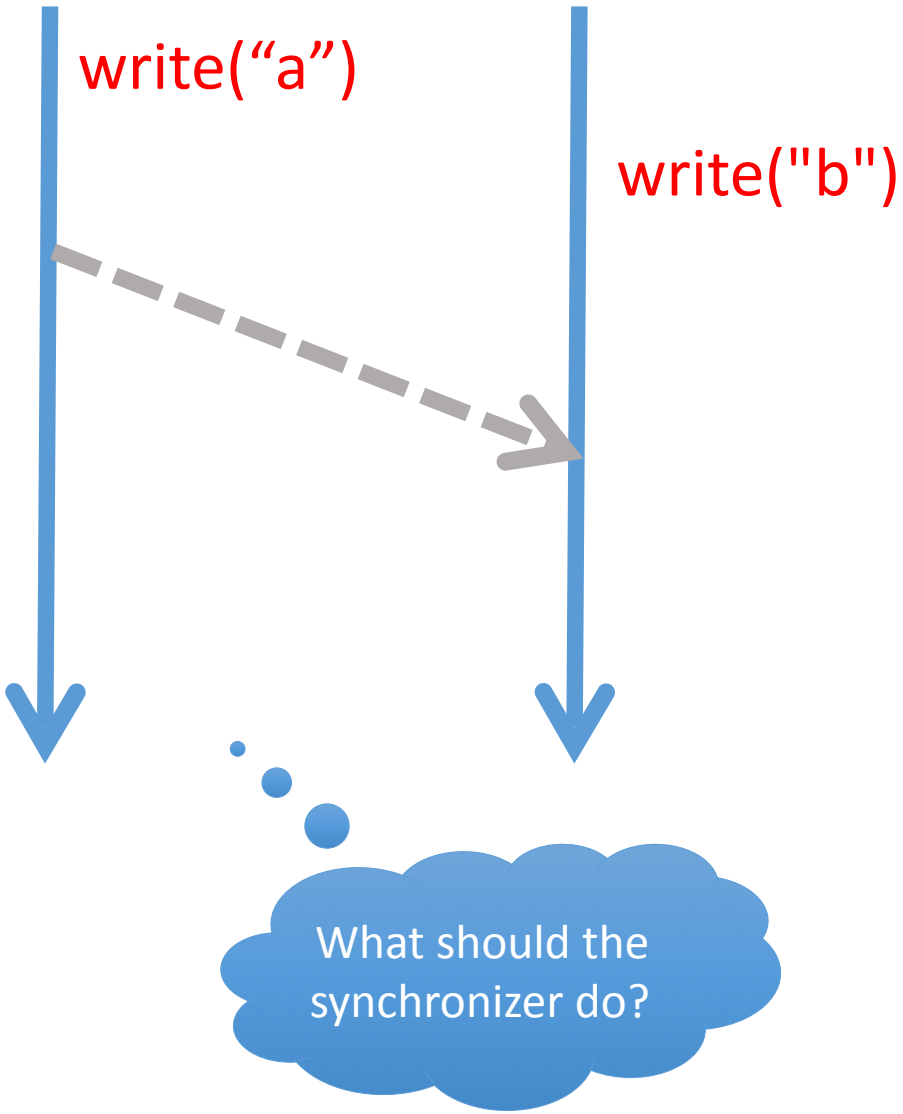
READ$_N$ ⟶ ⊥ means that the file is missing

WRITE$_N$ (⊥) means delete the file

Challenge #1: conflicts



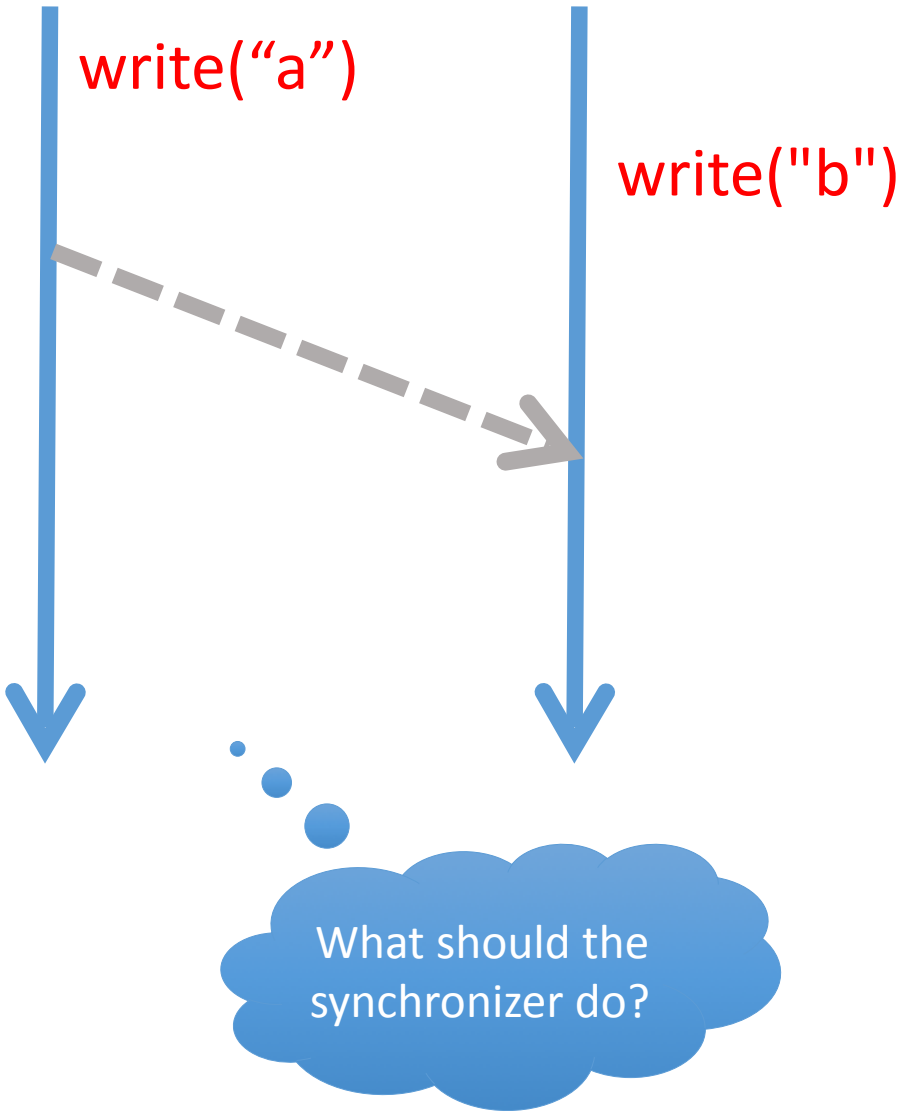
Challenge #1: conflicts



Dropbox’s answer:

The “earlier” value wins; other values are moved to *conflict files* in the same directory.

Challenge #1: conflicts



Dropbox’s answer:
The “earlier” value wins; other values are moved to *conflict files* in the same directory.

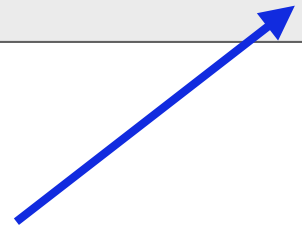
However, these conflict files may not appear for a little while!

Second try…

Operations	Observations
READ$_N$	READ$_N$ ⟶ "*current value*"
WRITE$_N$ ("*new value*")	WRITE$_N$ ("*new value*") ⟶ "*old value*"
STABILIZE	STABILIZE ⟶ ("*value*", {"*conflict values*"})

Second try…

Operations	Observations
READ$_N$	READ$_N$ ⟶ "*current value*"
WRITE$_N$ ("*new value*")	WRITE$_N$ ("*new value*") ⟶ "*old value*"
STABILIZE	STABILIZE ⟶ ("*value*", {"*conflict values*"})



Same value in the file on all clients

Second try…

Operations	Observations
READ$_N$	READ$_N$ ⟶ "*current value*"
WRITE$_N$ ("*new value*")	WRITE$_N$ ("*new value*") ⟶ "*old value*"
STABILIZE	STABILIZE ⟶ ("*value*", {"*conflict values*"})

Same value in the file on all clients

Same set of values in conflict files on all clients

Challenge #2: Background operations

- The Dropbox client communicates with the test harness by observing what it writes to the filesystem.

But…

- The Dropbox client *also* communicates with the Dropbox servers!
  - Timing of these communications is unpredictable

Challenge #2: Background operations

- The Dropbox client communicates with the test harness by observing what it writes to the filesystem.

But…

- The Dropbox client *also* communicates with the Dropbox servers!
  - Timing of these communications is unpredictable

Invisible, unpredictable activity → Nondeterminism!

Approach

- Model the whole system state *including the (invisible) state of the server*
- Add "conjectured actions" to the observed ones

UP$_N$	node *N* uploads its value to the server
DOWN$_N$	node *N* is refreshed by the server

Final version:

Operations	Observations
READ$_N$	READ$_N$ ⟶ "*current value*"
WRITE$_N$ ("*new value*")	WRITE$_N$ ("*new value*") ⟶ "*old value*"
STABILIZE	STABILIZE ⟶ ("*value*", {"*conflict values*"})
	UP$_N$
	DOWN$_N$

For example…

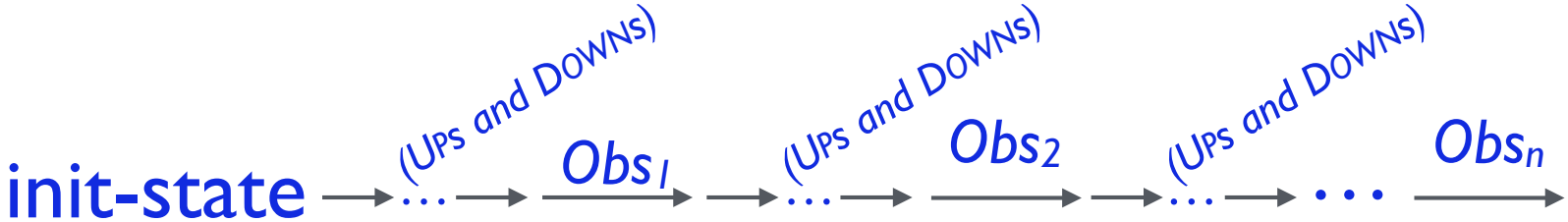
Final version:

Operations	Observations
READ$_N$	READ$_N$ ⟶ "*current value*"
WRITE$_N$ ("*new value*")	WRITE$_N$ ("*new value*") ⟶ "*old value*"
STABILIZE	STABILIZE ⟶ ("*value*", {"*conflict values*"})
	UP$_N$
	DOWN$_N$

For example…

Final specification

If $Op_1 \ldots Op_n$ is some sequence of operations and $Obs_1 \ldots Obs_n$ are the observations we make when we run them, then we can add UP/DOWN "observations" to yield an *explanation* such that



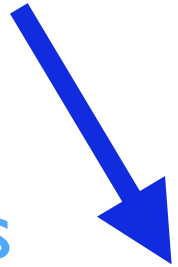
is a valid sequence of transitions of the model.

Test

Client 1	Client 2
WRITE ‘a’	
	WRITE ‘b’
READ	
	WRITE ‘c’
STABILIZE	

Test

Client 1	Client 2
WRITE ‘a’	
	WRITE ‘b’
READ	
	WRITE ‘c’
STABILIZE	



Observations

Client 1	Client 2
WRITE ‘a’ $\rightarrow \perp$	
	WRITE ‘b’ $\rightarrow$ ‘a’
READ $\rightarrow$ ‘b’	
	WRITE ‘c’ $\rightarrow$ ‘b’
STABILIZE $\rightarrow$ (‘c’, $\emptyset$)	

Test

Client 1	Client 2
WRITE ‘a’	
	WRITE ‘b’
READ	
	WRITE ‘c’
STABILIZE	

Observations

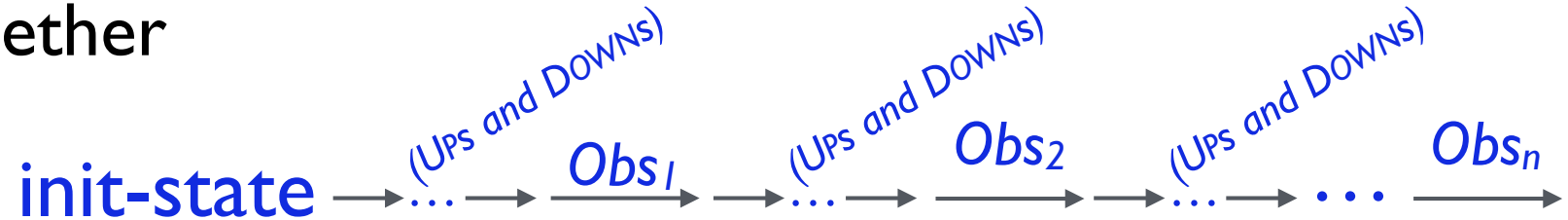
Client 1	Client 2
WRITE ‘a’ → ⊥	
	WRITE ‘b’ → ‘a’
READ → ‘b’	
	WRITE ‘c’ → ‘b’
STABILIZE → (‘c’, ∅)	

Explanation

Client 1	Client 2
WRITE ‘a’ → ⊥	
UP	
	DOWN
	WRITE ‘b’ → ‘a’
	UP
DOWN	
	WRITE ‘c’ → ‘b’
	UP
READ → ‘b’	
DOWN	
STABILIZE → (‘c’, ∅)	

Using the specification for testing

1. Generate a random sequence of operations $Op_1 \ldots Op_n$
2. Apply them to the system under test, yielding observations $Obs_1 \ldots Obs_n$
3. Calculate *all* ways of interleaving Up and Down observations with $Obs_1 \ldots Obs_n$ and, for each one, check whether



   is a valid sequence of transitions of the model
4. If the answer is "no" for every possible interleaving, we have found a failing test; otherwise, repeat

Model states

- *Stable value* (i.e., the one on the server)
- *Conflict set* (only ever grows)
- For each node:
  - Current *local value*
  - "FRESH" or "STALE"
  - "CLEAN" or "DIRTY"

Model states

- *Stable value* (i.e., the one on the server)
- *Conflict set* (only ever grows)
- For each node:
  - Current *local value*
  - "FRESH" or "STALE"
  - "CLEAN" or "DIRTY"

i.e., has the global value changed since this node's last communication with the server

Model states

- *Stable value* (i.e., the one on the server)
- *Conflict set* (only ever grows)
- For each node:
  - Current *local value*
  - "FRESH" or "STALE"
  - "CLEAN" or "DIRTY"

i.e., has the global value changed since this node's last communication with the server

i.e., has the local value been written since this node was last refreshed by the server

Modeling the operations

READ $\rightarrow V$

Precondition: $LocalVal_N = V$

Effect: none

WRITE $V_{new} \rightarrow V_{old}$

Precondition: $LocalVal_N = V_{old}$

Effect: $LocalVal_N \leftarrow V_{new}$

$Clean?_N \leftarrow$ DIRTY

Modeling the operations

$\textsc{Stabilize} \rightarrow (V, C)$

Precondition: $ServerVal = V$

$Conflicts = C$

for all N, $Fresh?_N = \textsc{fresh}$

$Clean?_N = \textsc{clean}$

Effect: none

Modeling the operations

DOWN

Precondition: $Fresh?_N = \text{STALE}$

$Clean?_N = \text{CLEAN}$

Effect: $LocalVal_N \leftarrow ServerVal$

$Fresh?_N \leftarrow \text{FRESH}$

Modeling the operations

UP

Precondition: $Clean?_N = \text{DIRTY}$

Effect: $Clean?_N \leftarrow \text{CLEAN}$

if $Fresh?_N = \text{FRESH}$ then

if $LocalVal_N \neq ServerVal$ then

$Fresh?_{N'} \leftarrow \text{STALE}$ for all $N' \neq N$

$ServerVal \leftarrow LocalVal_N$

else

if $LocalVal_N \notin \{ServerVal, \perp\}$ then

$Conflicts \leftarrow Conflicts \cup \{LocalVal_N\}$

Surprises…

Surprise: Dropbox can (briefly) delete a newly created file…

Client 1	Client 2
WRITE ‘a’ → ⊥	
WRITE ⊥ → ‘a’	
	WRITE ‘b’ → ‘a’
WRITE ‘c’ → ⊥	
READ → ⊥	

Surprise: Dropbox can (briefly) delete a newly created file…

	Client 1	Client 2
Create file	WRITE ‘a’ → ⊥	
	WRITE ⊥ → ‘a’	
		WRITE ‘b’ → ‘a’
	WRITE ‘c’ → ⊥	
	READ → ⊥	

Surprise: Dropbox can (briefly) delete a newly created file…

	Client 1	Client 2
Create file	WRITE ‘a’ → ⊥	
Delete it	WRITE ⊥ → ‘a’	
		WRITE ‘b’ → ‘a’
	WRITE ‘c’ → ⊥	
	READ → ⊥	

Surprise: Dropbox can (briefly) delete a newly created file…

Client 1	Client 2	
Create file: WRITE ‘a’ → ⊥		
Delete it: WRITE ⊥ → ‘a’		
	WRITE ‘b’ → ‘a’	Observe creation
WRITE ‘c’ → ⊥		
READ → ⊥		

Surprise: Dropbox can (briefly) delete a newly created file…

	Client 1	Client 2	
Create file	WRITE ‘a’ → ⊥		
Delete it	WRITE ⊥ → ‘a’		
		WRITE ‘b’ → ‘a’	Observe creation
Create it again	WRITE ‘c’ → ⊥		
	READ → ⊥		

Surprise: Dropbox can (briefly) delete a newly created file…

	Client 1	Client 2	
Create file	WRITE ‘a’ → ⊥		
Delete it	WRITE ⊥ → ‘a’		
		WRITE ‘b’ → ‘a’	Observe creation
Create it again	WRITE ‘c’ → ⊥		
File is gone!	READ → ⊥		

Surprise: Dropbox can (briefly) delete a newly created file…

	Client 1	Client 2	
Create file	WRITE ‘a’ → ⊥		
Delete it	WRITE ⊥ → ‘a’		
		WRITE ‘b’ → ‘a’	Observe creation
Create it again	WRITE ‘c’ → ⊥		
File is gone!	READ → ⊥		

Timing is critical!

Surprise: Dropbox can (briefly) delete a newly created file…

	Client 1	Client 2	
Create file	WRITE ‘a’ → ⊥		
Delete it	WRITE ⊥ → ‘a’		
		WRITE ‘b’ → ‘a’	Observe creation
Create it again	WRITE ‘c’ → ⊥		
File is gone!	READ → ⊥		

Timing is critical! → Add **SLEEP** operations in tests

Surprise: Dropbox can (permanently) re-create a deleted file…

Client 1
WRITE ‘b’ $\rightarrow \perp$
WRITE $\perp \rightarrow$ ‘b’
READ $\rightarrow$ ‘b’

(other clients idle)

Surprise: Dropbox can (permanently) re-create a deleted file…

Create file

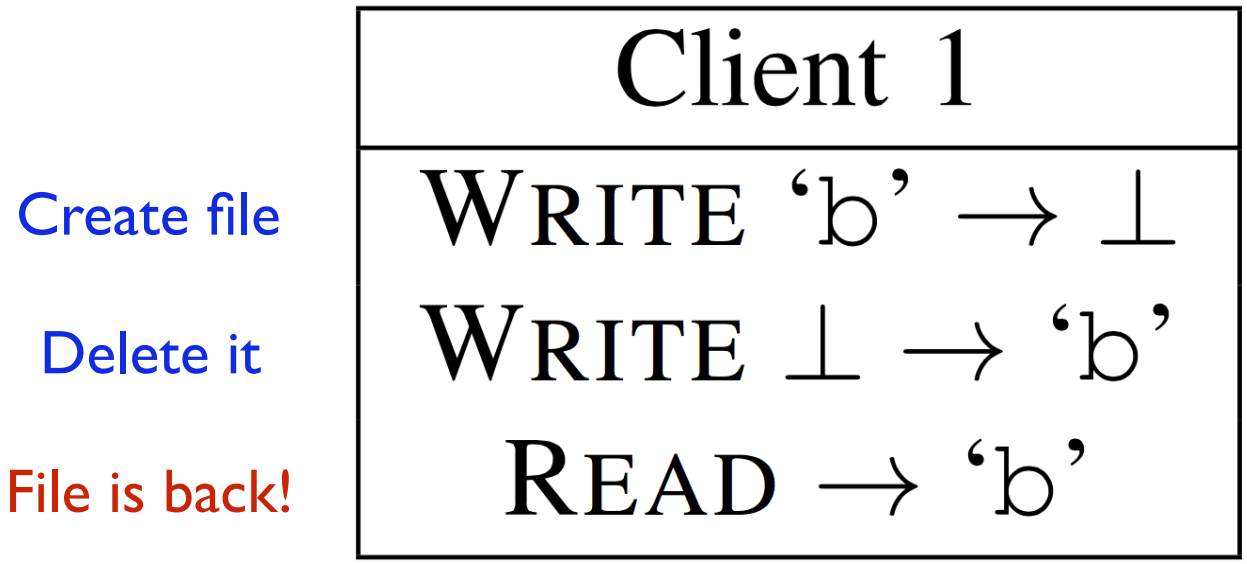
Client 1
WRITE ‘b’ → ⊥
WRITE ⊥ → ‘b’
READ → ‘b’

(other clients idle)

Surprise: Dropbox can (permanently) re-create a deleted file…

	Client 1	
Create file	WRITE ‘b’ → ⊥	(other clients idle)
Delete it	WRITE ⊥ → ‘b’	
	READ → ‘b’	

Surprise: Dropbox can (permanently) re-create a deleted file…



(other clients idle)

(Again, timing is critical)

Surprise: Dropbox can lose data

Client 1	Client 2
WRITE ‘a’ → ‘b’ READ → ‘a’	WRITE ‘b’ → ⊥
STABILIZE → { (‘a’, ∅), (‘b’, ∅) }	

Surprise: Dropbox can lose data

Client 1	Client 2
	WRITE ‘b’ → ⊥
WRITE ‘a’ → ‘b’ READ → ‘a’	
STABILIZE →	
{ (‘a’, ∅),	(‘b’, ∅) }

Create file

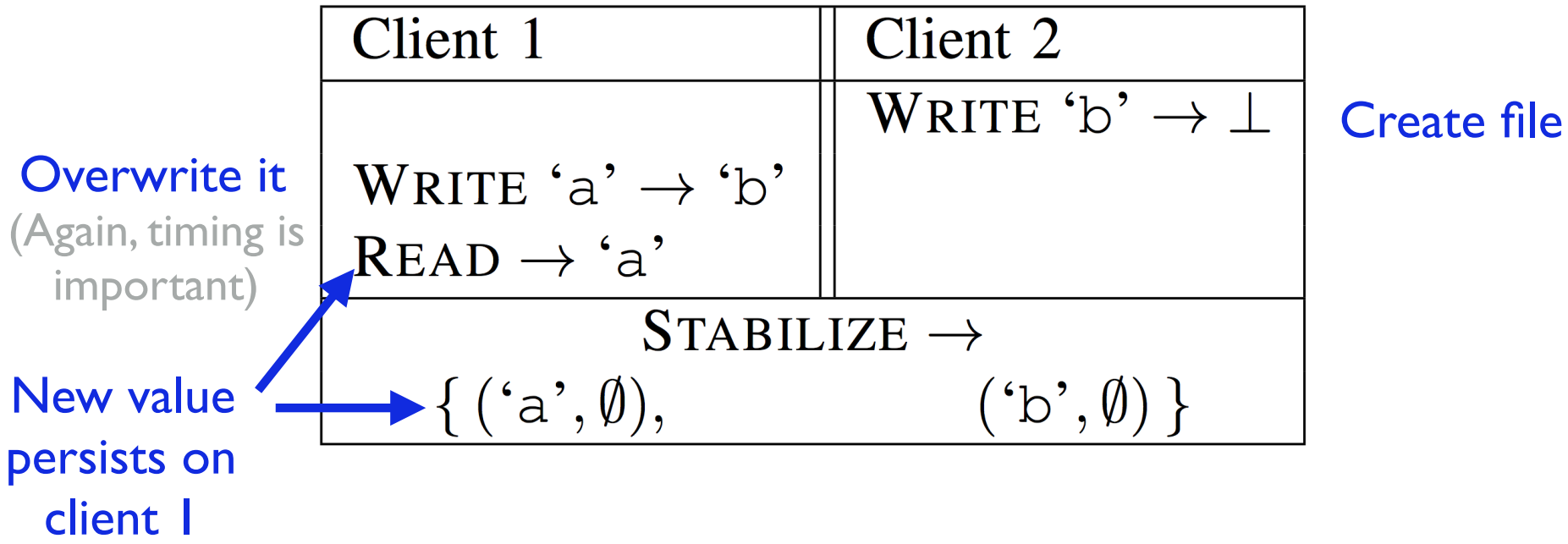
Surprise: Dropbox can lose data

Overwrite it
(Again, timing is important)

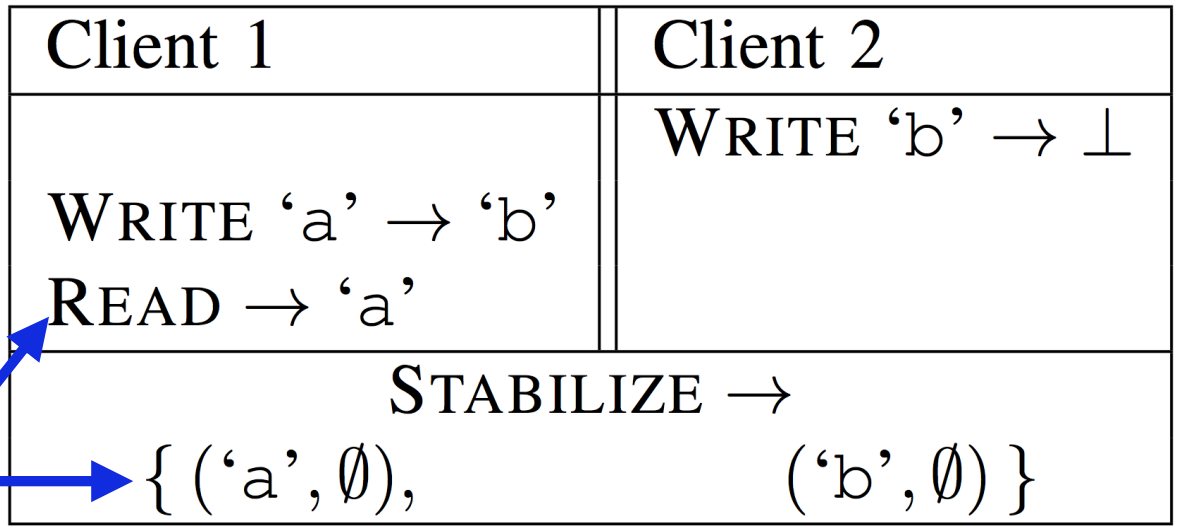
Client 1	Client 2
WRITE ‘a’ → ‘b’ READ → ‘a’	WRITE ‘b’ → ⊥
STABILIZE → { (‘a’, ∅), (‘b’, ∅) }	

Create file

Surprise: Dropbox can lose data



Surprise: Dropbox can lose data



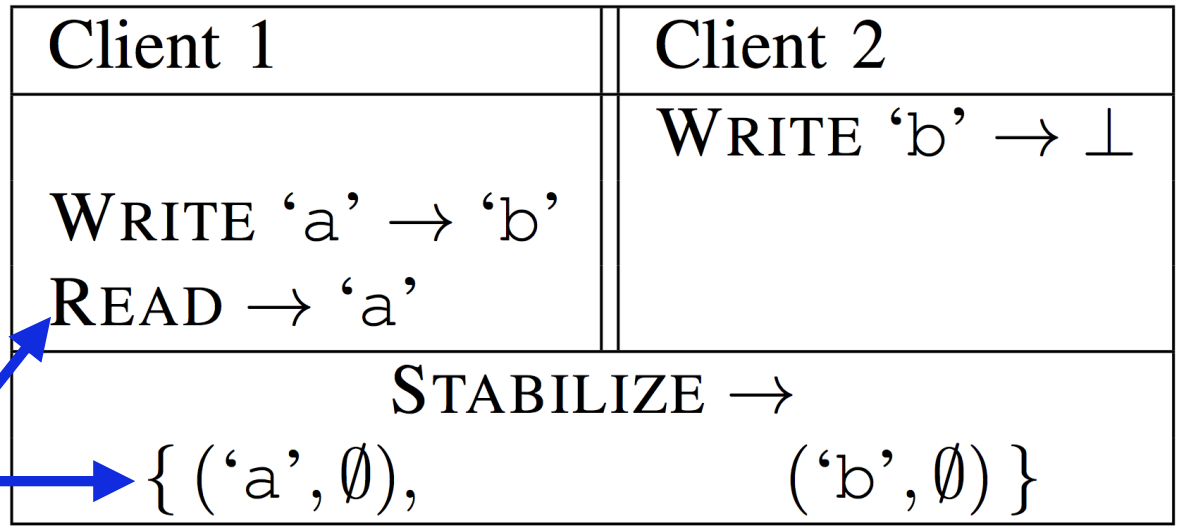
Overwrite it
(Again, timing is important)

New value persists on client 1

Create file

Old value persists on client 2!

Surprise: Dropbox can lose data



Create file

Overwrite it
(Again, timing is important)

New value persists on client 1

Old value persists on client 2!

Client 1 believes it is still Fresh, so if we later write a new value on client 2, it will silently overwrite client 1's value and no conflict file will be created

Work in progress!

- More details:
  - Draft paper available from my webpage
- Next steps:
  - Add directories
  - Test your favorite synchronizer :-)

Thank you!

(Any questions?)