Agenda for today

Review definition of *implements* as subset of traces, with cache example

Too much spec state → history variables, with StatDB example

Abstraction relations ↔ history variables

Premature choice → prophecy variables
Correctness

Definition: External code traces \( \subseteq \) external spec traces

Note: this throws away a lot of information: internal state and actions

Safety: what code **may** do—never anything bad
Liveness: what code **must** do—eventually something good

A sequential procedure (e.g., \( \text{sort}(a) \)) relates initial and final states.
Correctness: code relation \( \subseteq \) spec relation. That is,

  Safety: the spec allows the final state of the code (partial correctness)
  Liveness: termination (total correctness)

Basic idea for proof: **simulation**—code matches spec one action at a time
Simple example: code for memory with cache

Note: a command may change the state, be non-deterministic. You can only assign its return value. An expression is math: no state change, deterministic.

Spec for memory

type \( M \)  =  \( A \rightarrow V \)
var  \( m \)  :  \( M \)

\[
\text{initM}() = \text{var} m' \mid (m'.\text{dom} = A); m := m' \quad m' \text{ arbitrary, defined}
\]

\[
\text{read}(a): V = m(a)
\]

\[
\text{write}(a, v) = m(a) := v \quad m := m(a := v)
\]
Writeback cache code

type $C = A \rightarrow \text{opt } V$

const $cSize = 4096$

var $mc: M$

$c: C$

\[\text{initC}() = \text{var } c' | c'.\text{dom.size} = cSize \]
\[\wedge \forall a \in c'.\text{dom} | c'(a) = mc(a);\]
\[c := c'\]

\[\text{read}(a): V = \text{load}(a); \text{ret } c(a)\]

\[\text{write}(a,v) = \text{load}(a); c(a) := v\]
Internal

\[
\text{read}(a) : V = \text{load}(a); \textbf{ret} \ c(a)
\]
\[
\text{write}(a, v) = \text{load}(a); c(a) := v
\]

\[
\text{load}(a) = \begin{cases} 
\text{if } c(a) \neq \text{None} \text{ then skip} \\
\text{else } \{ \text{flush1}(); c(a) := \text{mc}(a) \}
\end{cases}
\]

\[
\text{flush1}() = \begin{cases} 
\text{var } a \in c.\text{dom}; \\
\text{if } c(a) = \text{mc}(a) \text{ then skip} \\
\text{else } \{ \text{mc}(a) := c(a)\}; \\
\text{c}(a) := \text{None}
\end{cases}
\]

Note: the spec is deterministic, but the code is not.
Abstraction function

It has to take the code state \((m, c)\) to the spec state \(m\).

\[
m = mc + c
\]

function overlay

Writing this out

\[
m = \text{fun } a \Rightarrow \text{if } c(a) \neq \text{None then } c(a) \text{ else } mc(a)
\]

Why code to spec? Because code has many ways to represent the spec as in the cache example.
(Not always true, though. Sometimes the spec is better with more state.)

An abstraction function \(F\) must satisfy two conditions.

**Initial**: If \(t\) is an initial state of \(T\), then \(F(t)\) is an initial state of \(S\).

**Next** (simulation): If \(t\) is a reachable state of \(T\)
and \(t \xrightarrow{\pi} t'\) is a step of \(T\),
then there is a step \(F(t) \xrightarrow{\pi} F(t')\) of \(S\) that has the same trace \(\pi\).
Simulation

If $t$ is a reachable state of $T$ and $t \xrightarrow{\pi} t'$ is a step of $T$, then there is a step $F(t) \xrightarrow{\pi} F(t')$ of $S$ that has the same trace $\pi$.

The diagram commutes

**Theorem:** If there is an abstraction function from $T$ to $S$, then $T$ implements $S$, i.e., every trace of $T$ is a trace of $S$.

Why: Induction on the length of the trace.
Invariant

**Invariant**: a property that is true of all reachable states.

Must show that code simulates spec from every **reachable** state
The invariant bounds the reachable states.
   Usually the code doesn’t simulate the spec from *every* state.

Usually prove $P$ is an invariant by induction on the length of executions.
   $P$ is true initially, and if $P(s)$ and $s \rightarrow s'$ then $P(s')$

The only invariant for the WB cache is the cache size: $c.\text{dom.size} = c\text{Size}$

Hash table code for the memory spec would have a more interesting one:
   If $v$ is in the bucket with hash $h$, then $\text{hash}(v) = h$.

If cache entries have dirty bits the invariant is: $c(a) \neq m(a) \Rightarrow \text{dirty}(a)$
Cache code with dirty

\textbf{type} \( C \) \hspace{0.5cm} = \ A \to \textbf{opt} \ (\text{\textit{val}}: V, \text{\textit{dirty}}: \text{Bool}) \hspace{0.5cm} \text{c defined at some A’s memory in the code}

\textbf{var} \ mc \hspace{0.5cm} : \ M \hspace{0.5cm} \text{memory in the code}

\hspace{1cm} \begin{array}{l}
\textbf{c} \hspace{0.5cm} : \ C \\
\end{array}

\textit{init}C() \hspace{0.5cm} = \hspace{0.5cm} \textbf{var} \ c’ \hspace{0.5cm} | \hspace{0.5cm} c’.\text{dom.size} = \text{cSize} \hspace{0.5cm} \text{c’ defined at cSize A’s}
\hspace{1cm} \begin{array}{l}
\wedge \forall a \in c’.\text{dom} \hspace{0.5cm} | \hspace{0.5cm} c’(a) = (mc(a), \text{false}); \\
\text{c} := c’; \\
\end{array}

\textit{read}(a): V \hspace{0.5cm} = \hspace{0.5cm} \textit{load}(a); \hspace{0.5cm} \textbf{ret} \ c(a).\text{val}

\textit{write}(a, v) = \hspace{0.5cm} \textit{load}(a); \hspace{0.5cm} c(a) := (v, \text{true})
**Internal with dirty**

```plaintext
var mc : M
    c : C
read(a): V = load(a); ret c(a).val
write(a, v) = load(a); c(a) := (v, true)

load(a) = if c(a) ≠ None then skip
else {flush1();
    c(a) := (mc(a), false)}

flush1() = var a ∈ c.dom;
    if c(a).dirty
    then {mc(a) := c(a).val} else skip;
    c(a) := None
```

memory in the code

if a is in the cache, done
if not, make space, and put it there

pick an a in the cache
write a to m if dirty
and take a out of cache
Completeness

The simple abstraction function method always works if the spec avoids

• extra state,
• too few or too many transitions, and
• premature choices.

Why bother? Just don’t make bad choices in the spec?

No. A spec should be written to be as clear as possible to the clients, not to make it easy to prove the code correct.
Extra spec state

Notation: If $q: \text{seq } T$ and $\oplus$ is an operator on $T$, $\text{fold}(\oplus, q)$ is $q_0 \oplus q_1 \oplus \ldots \oplus q_n$

It works for sets too if $\oplus$ is commutative.

$$\text{sum}(q: \text{seq Int}) = \text{fold}(+, q)$$

StatDB. Definitions (from a textbook):

mean = $\frac{\sum_i db(i)}{n}$, variance = $\frac{\sum_i (db(i) - \text{mean})^2}{n} = \frac{\sum_i db(i)^2}{n} - \text{mean}^2$

var $db : \text{seq } V := [ ]$ a multiset

add($v$) = $db := db ++ [v]$

size(): Nat = $db$.size

mean(): opt $V = \text{if } db = [ ] \text{ then } \text{None } \text{ else } \text{sum}(db)/db$.size

var(): opt $V = \text{if } db = [ ] \text{ then } \text{None } \text{ else } \text{sum}(|v \in db \mid | (v - \text{mean}())^2})/db$.size
Efficient code for StatDB

\begin{verbatim}
var count := 0
    sum := 0
    sumSq := 0

add(v) = count := count + 1; sum := sum + v;
    sumSq := sumSq + v^2

mean(): opt V = if count = 0 then None else sum/count

var(): opt V = if count = 0 then None else sumSq/count - mean()^2
\end{verbatim}

But there’s no AF, because the spec has more state than the code—there’s no way we can conjure up all of \textit{db} from the three Nats in the code.

This is not a bad spec; the job of the spec is to be clear, not to be efficient.
History variables

To get an AF, we must add **history variables** to track the extra spec state.

In this case, we just add \( db \), the entire state of the spec, and the spec’s action

\[
db := db ++ [v] \text{ in add.}
\]

Adding all the spec state always works, but often you can add less.

The history variables are not allowed to affect the ordinary variables, so it’s obvious that the code with history variables has the *same* traces.

So instead of \( code \subseteq spec \), we have \( code = code + history \subseteq spec \).

The AF proof needs an invariant that relates \( db \) to the rest of the state.

\[
\begin{align*}
\text{count} &= db.\text{size} \\
\text{sum} &= \text{fold}(+, db) \\
\text{sumSq} &= \text{fold}(+, db \circ \text{square})
\end{align*}
\]
Abstraction relations

Another way: generalize AFs to *abstraction relations*. \( \approx \) is an AR if:

**Initial**: If \( t \) is an initial state of \( T \), \( S \) has an initial state \( s \) such that \( t \approx s \).

**Next** (simulation): If \( t \) and \( s \) are reachable, \( t \approx s \) and \( t \xrightarrow{\pi} t' \) is a \( T \) step, then there exists an \( s' \) such that \( s \xrightarrow{\pi} s' \) is an \( S \) step, and \( t' \approx s' \).

\[ T \text{ and } S \text{ run in parallel. } T \text{ drives the execution, doing what it wants.} \]
\[ S \text{ follows along, producing the same external trace.} \]
\[ \text{The two conditions guarantee that } S \text{ can always do this.} \]
**AR for StatDB**

Here’s the AR for StatDB

\[
\begin{align*}
\text{count} &= \text{db.size} \\
\text{sum} &= \text{fold}(+, \text{db}) \\
\text{sumSq} &= \text{fold}(+, \text{db} \circ \text{square})
\end{align*}
\]

It’s just the same as the invariant we had for the history variables.
Taking several steps in the spec

If \( t \) and \( s \) are reachable states of \( T \) and \( S \) with \( t \approx s \), and \( t \xrightarrow{\pi} t' \) is a step of \( T \), then there’s an \textit{execution fragment} (sequence of steps) of \( S \) from \( s \) to some \( s' \) with the same trace (one \( \pi \) step, others internal), and with \( t' \approx s' \).

\[
\begin{align*}
  s_0 \Rightarrow \cdots \Rightarrow s_i \xrightarrow{\pi} s_{i+1} \Rightarrow \cdots \Rightarrow s'
\end{align*}
\]

Usually there are many internal code steps for each spec step, e.g., machine instructions.

Sometimes you want internal spec steps, such as the \textit{drop} action in the async messaging spec.
Premature choice and prophecy variables

Two realistic examples of premature choice in specs.

**Reliable two-party channel spec**

\[
\textbf{var} \ ch \ : \ \textbf{seq} \ Msg
\]

\[
\text{send}(msg) = ch := ch ++ [msg]
\]

\[
\text{rcv}(addr) = \textbf{if} \ msg = ch.\text{head} \ \textbf{then} \{ch := ch.\text{tail}; \ \textbf{ret} \ msg\}
\]

At a crash, maybe drop some messages.

\[
\text{crash} = \textbf{var} \ \text{keep} \subseteq ch.\text{dom}; \ ch := \text{keep}.\text{sort} \circ ch
\]

\[
\begin{array}{c}
0 \\ 1 \\ 3
\end{array} \rightarrow
\begin{array}{c}
0 \\ 1 \\ 2 \\ 3
\end{array}
\]

\[
\begin{array}{c}
a \\ b \\ c \\ d
\end{array} \rightarrow
\begin{array}{c}
0 \\ 1 \\ 2 \\ 3
\end{array}
\]

\[
\begin{array}{c}
0 \\ 1 \\ b \\ d
\end{array}
\]

\[
\text{keep}.\text{sort} \quad ch \quad ch'
\]
**Code uses async messaging**

```plaintext
var ch : set AsyncMsg

send(asm) = ch := ch \cup \{asm\}
rcv() = \textbf{var} asm \in ch \mid \textbf{ret} asm
doesn’t drop from \textit{ch}

The drop action is internal, and non-deterministic.

\texttt{drop()} = \textbf{var} asm \in ch; ch := ch - \{asm\}
\texttt{can drop any time}
```

In real code (such as TCP) a message can be lost long after a crash.

- An \textit{asm} for a message sent before a crash can stay in the network.
- If the \textit{asm} is dropped, so is the \textit{msg}, since there’s no retransmission.
- If the \textit{asm} is received, so is the \textit{msg}.
Spec without premature choice

To fix this, mark in-flight messages:

```plaintext
type MK = (m: Msg, mark: Bool) marked message
var ch : seq MK channel

send(msg) = ch := ch ++ [(msg, false)]
rcv(addr) = if mk = ch.head then {ch := ch.tail; ret mk.m}
drop = if mk = ch.head ∧ mk.mark then ch := ch.tail

crash = ch := ch ◦ (fun mk ⇒ (mk.m,true)) mark all mk’s in flight
```
Consensus spec

\begin{verbatim}
var agreed : opt V := None

allow(v) = var a ∈ {v, None};
    if agreed = None then agreed := a

result(): opt V = var a ∈ {agreed, None}; ret a
\end{verbatim}

data value to agree on

optionally accept v

optionally return agreed

This spec chooses the value to agree on as soon as the value is allowed.

\textit{result} may return \textit{None} even after the choice is made because in distributed code maybe not all the participants know what the outcome is.
**Consensus without premature choice**

Code saves allowed values, and the processes chat to choose a value. The following spec has that form. It has a new `allow`, and adds `agree`.

```plaintext
var agreed : opt V := None          data value to agree on
var allowed : set V := {}           allowed values

allow(v) = allowed := allowed ∪ {v}  remember v
result(): opt V = var a ∈ {agreed, None}; ret a  optionally return agreed
agree() = var a ∈ allowed;
          if agreed = None then agreed := a
```

*Note:* if *result* couldn’t return *None* even after *agree*, these specs would be different, because the second would allow the behavior

```
allow(1); result() = None; allow(2); result() = 1
```

and the first would not.
Prophecy variables

Abstraction functions can still do the job here, but we need an auxiliary variable that looks into the future, as a history variable looks into the past.

A system $TP$ ($T$ with Prophecy) with a *prophecy variable* has the same traces as the original system $T$. 
## Rules for prophecy variables

<table>
<thead>
<tr>
<th><strong>History variable ( h )</strong></th>
<th><strong>Prophecy variable ( p )</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Every <em>initial</em> state has at least one value for ( h ).</td>
<td>1. <em>Every</em> state has at least one value for ( p ).</td>
</tr>
<tr>
<td>2. No existing step is disabled by new guards involving ( h ).</td>
<td>2. No existing step is disabled <em>in the backward direction</em> by ( p ):</td>
</tr>
</tbody>
</table>
| | If \( t \overset{\pi}{\Rightarrow} t' \) is a step and \((t', p')\) a state, there’s a \( p \) and a step of \( TP \)
| | \( (t, p) \overset{\pi}{\Rightarrow} (t', p') \) |
| 3. Assigning to a vanilla variable (including a return value) doesn’t depend on \( h \). | 3. Same: \( p \) *can* affect what actions are enabled, but not how an action changes a vanilla variable. |
| 4. If \( t \) is an initial state of \( T \) and \((t, p)\) is a state of \( TP \), it’s initial. |
How to write premature-choice specs

Most people find prophecy variables hard to grasp.

If you have a spec $S$ that makes a premature choice (not likely), deal with it at the highest possible level, as above:

- write another spec $S'$ without premature choice, and
- use a prophecy variable (or sheer willpower) to show that $S' = S$.

Then proving lower level code won’t involve prophecy variables.
Digression on naming actions

There are (at least) two ways to identify the actions.

- The way the diagram and the code names an action by an identifier such as `read`, together with the values of all the arguments and results.
  
  Thus \( \text{read}(a) \): 3 or \( \text{read}(a, 3) \).

- The other way only talks about state, and it makes the calling sequence explicit with state variables for the arguments and results.
  
  So memory has variables `doRead`, `doWrite`, \( a, v \).
  
  The action named `read(b, u)` corresponds to the steps
  
  \[
  a := b, \quad \text{doRead} := \text{true};
  \]
  \[
  v := u, \quad \text{doRead} := \text{false}
  \]

  That is, memory sets `doRead := false` when the result is ready.