The Die Hard System

TLA$^+$ Course

Leslie Lamport

The Problem

Our heros must defuse a bomb by putting exactly 4 gallons of water on a scale.

They have a 3-gallon jug, a 5-gallon jug, and a water faucet.

We model a system as a set of behaviors, where a behavior is a sequence of states.

We represent a state as an assignment of values to variables.

We describe the behaviors by:

An initial predicate $Init$ describing possible initial states.

A next-state action $Next$ specifying possible state changes.

This specifies all behaviors $s_1 \rightarrow s_2 \rightarrow \cdots$ such that

- s_1 satisfies $Init$
- Each step $s_i \rightarrow s_{i+1}$ satisfies $Next$.

How TLC Works

It generates this state / transition graph.

How to Get Blown Up

1. Not know when jug is “full”.

2. Have jug spring a leak.

The TLA$^+$ spec is a model of the real world.

No model is perfectly accurate.

Some models are useful.

Engineering $\neq$ math

Real Engineering = math + ...

TLA$^+$ can help with the math, not with the ...

Other Ways not to Get Blown Up

It's easy to put 3 gallons in jug.

Get someone to run to a store and buy a gallon of water, or of apple juice.

Or 8 pounds of potatos.

If a design is good, it should be easy to model.

The converse isn’t necessarily true.

Exercise: Generalizing $DieHard$

Instead of using 3 & 5 gallon jugs to obtain 4 gallons,

use $J1$ & $J2$ gallon jugs to obtain G gallons.

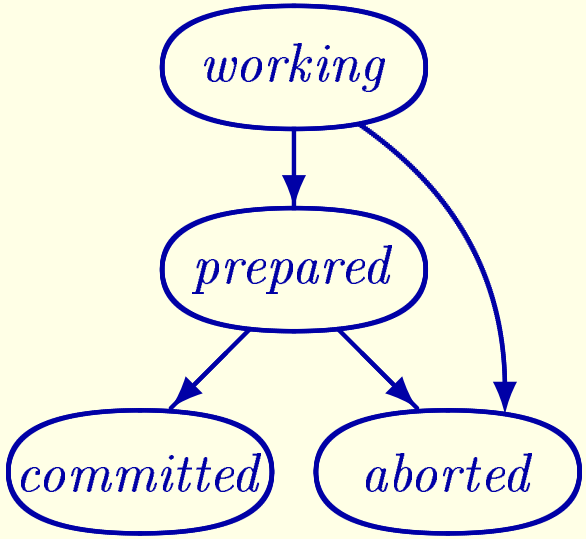
A Specification of Transaction Commit

TLA$^+$ Course

Leslie Lamport

There is a set of Resource Managers (RMs).

Each RM is described by the following state diagram:



The RMs must either all commit or all abort.

How do we declare that RM is a Set?

We don't have to.

TLA$^+$ is based on a set theory in which every value is a set.

Even 42 and "abc" are sets, but the axioms of TLA$^+$ don't say what their elements are.

We don't know if "abc" $\in$ 42 equals TRUE or FALSE.

TLC will report an error if it has to evaluate that expression.

Functions — a.k.a. Arrays

Programming	**Math**
array	function
index set $(0 \,..\, n)$	domain (any set)
$f\,[x]$	$f\,(x)$

TLA$^+$ is math, but uses $f\,[x]$.

Parentheses used for defined operators, as in $Min(m, n)$.

TLA$^+$ Notation for Functions

Used in definition of $TCTypeOK$:

$[S \to T]$

The set of all functions f with domain S such that $\forall\, x \in S : f[x] \in T$.

TLA$^+$ Notation for Functions

Used in definition of $TCInit$:

$$[x \in S \mapsto exp(x)]$$

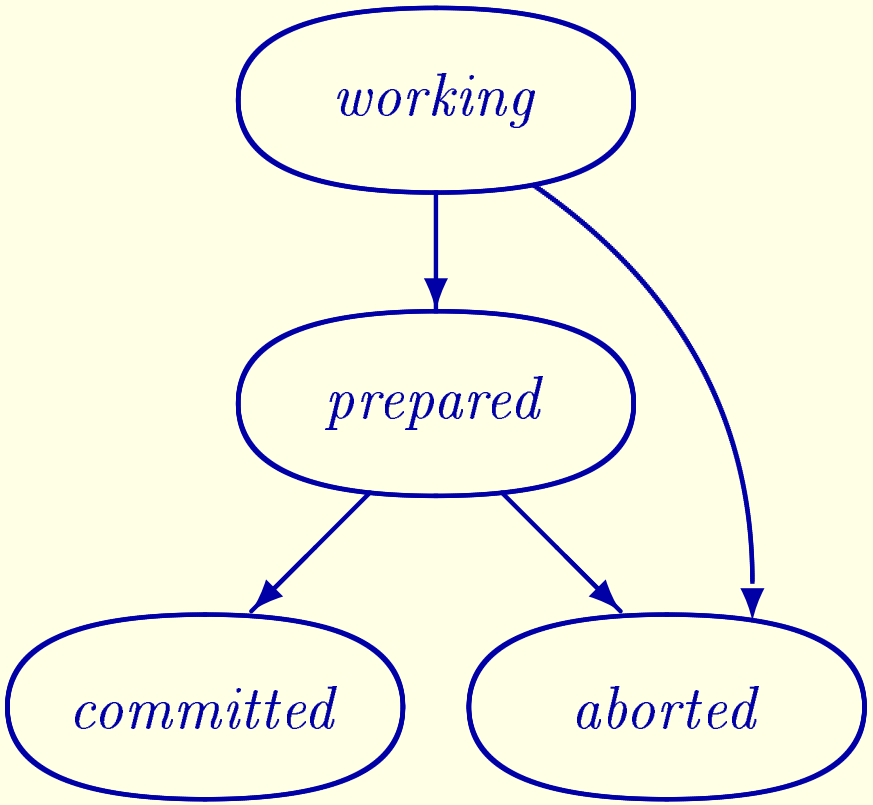
The function f with domain S
such that $\forall\, x \in S : f[x] = e(x)$.

For example:

$$square \triangleq [x \in Nat \mapsto x^2]$$

Defines $square$ to be the function with domain Nat
such that $square[n] = n^2$ for every $n \in Nat$.

Let's look at the next-state relation.



The obvious way to define $Prepare$:

$$Prepare(r) \triangleq \land\ rmState[r] = \text{“working”}$$
$$\land\ rmState[r]' = \text{“prepared”}$$

This specifies the new value of $rmState[r]$.

But what about the new value of $rmState[s]$
for RMs s other than r?

And what about the new value of the domain of $rmState$?

~~$rmState[r]' = \text{“prepared”}$~~

$$rmState' = [s \in RM \mapsto \text{IF } s = r \text{ THEN “prepared”} \text{ ELSE } rmState[s]\ ]$$

Correct, but we need an abbreviation.

$$rmState' = [rmState \text{ EXCEPT } ![r] = \text{“prepared”}]$$

$[f \text{ EXCEPT } ![x] = v]$

When anyone sees this, it's hatred at first sight.

What does the ! mean? Nothing!

You'll get used to it.

You can also write:

$[f \text{ EXCEPT } ![42] = \text{“abc”},\ ![2] = \text{“d”}]$

$[f \text{ EXCEPT } ![42][-2] = \text{“abc”}]$

If you really hate EXCEPT, you can define

$$Assign(fcn, idx, val) \triangleq fcn' = [fcn \text{ EXCEPT } ![idx] = val]$$

You can then replace

$$rmState' = [rmState \text{ EXCEPT } ![r] = \text{“prepared”}]$$

with

$$Assign(rmState, r, \text{“prepared”})$$

$r1 :>$ “aborted” @@ $r2 :>$ “aborted” @@ $r3 :>$ “working”

$:>$ has higher precedence than @@ , so this equals

($r1 :>$ “aborted”) @@ ($r2 :>$ “aborted”) @@ ($r3 :>$ “working”)

The function with f with domain $\{r1, r2, r3\}$ such that

$f[r1] =$ “aborted”

$f[r2] =$ “aborted”

$f[r3] =$ “working”

:> and @@ defined in the standard TLC module.

When I tell the Toolbox to run a model MODEL_2,
it runs TLC on a module MC that begins:

MODULE *TLC*

EXTENDS $TCommit$, TLC

CONSTANTS $r1$, $r2$, $r3$

Everything you type into the model is put into module MC.
For example, model values become declared constants.

Exercise: Further Generalizing $DieHard$

Write a module $DieHarder$ in which the two jugs are replaced by an arbitrary constant set $Jugs$ of jugs whose sizes are specified by a constant $Sizes$.
The problem is again to obtain G gallons of water.

A Specification of Two Phase Commit

TLA$^+$ Course

Leslie Lamport

Properties

A state is any assignment of values to variables.

A behavior is any sequence of states.

A property is a assertion about behaviors.
It is either true or false on any behavior.

A TLA$^+$ specification is a property, for example:

$$TCInit \wedge \Box[TCNext]_{rmState}$$

Safety Properties

Assert that something bad doesn't happen.

False only if violated at some point in the execution.

Example: Only messages that were sent are received.

Liveness Properties

Assert that something good eventually does happen.

Have to see entire execution to know that it's false.

Example: Every message sent is received.

So far, we have only specified safety properties.

$Init$ and $Next$ specify only what is allowed to happen.

They don't assert that anything must happen

The specs we're writing now allow behaviors that stop at any time — even if additional steps are possible.

This may seem strange, but it's best to completely separate safety and liveness.

We'll see later how to specify liveness.

Records

A record corresponds to a Struct in C.

In TLA$^+$, a record with components $name$ and num is a function with domain $\{$“name”, “num”$\}$.

$rcd.name$ is an abbreviation for rcd[“name”].

$[name \mapsto$ “Fred”, $num \mapsto 42]$

is an abbreviation for

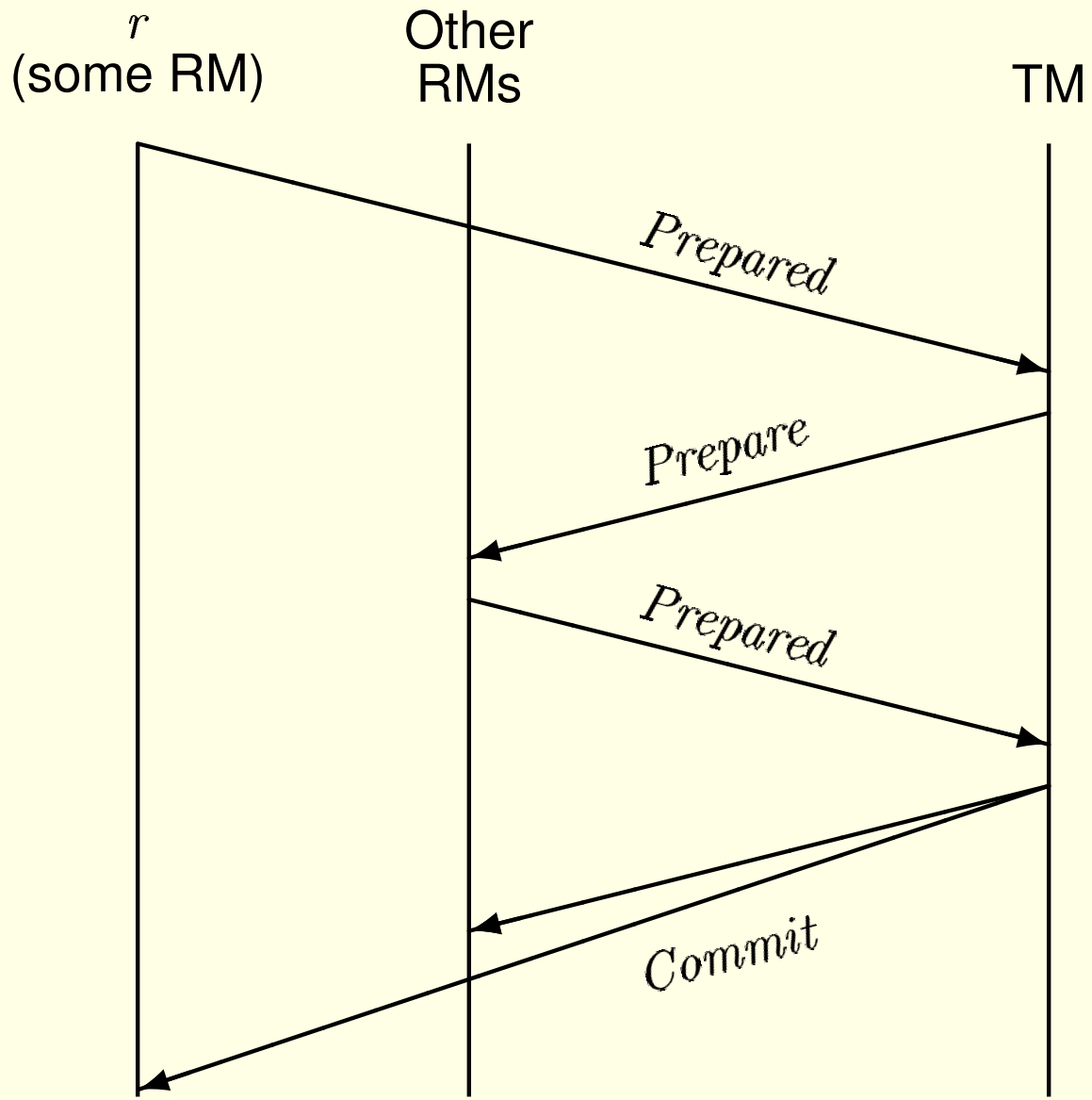
(“name” $:>$ “Fred”) @@ (“num” $:>$ 42)

$[name : \{\text{“Fred”}, \text{“Ted”}, \text{“Ned”}\},\ num : 0\,..\,99]$

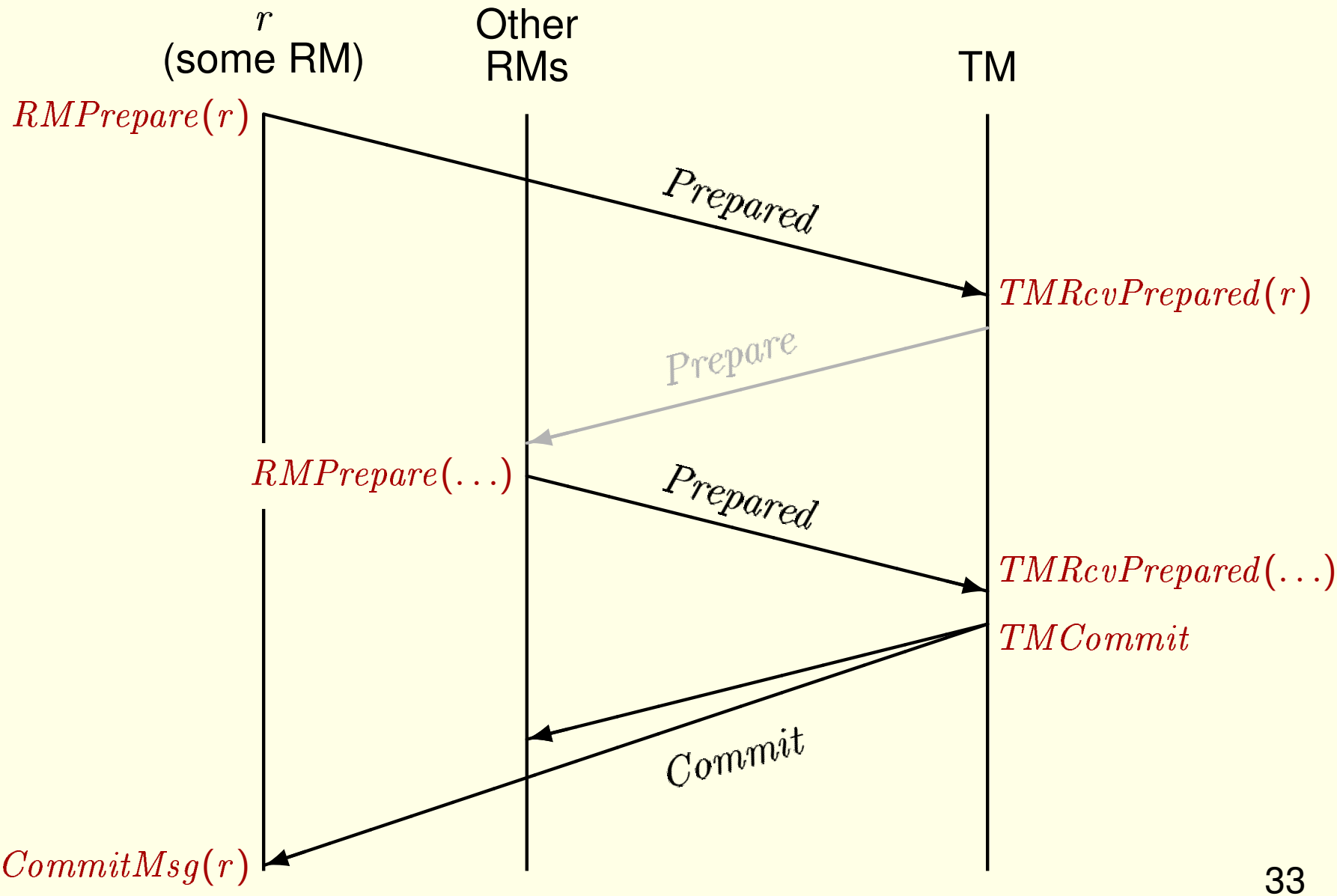
The set of all records $[name \mapsto a,\ num \mapsto b]$ such that

$(a \in \{\text{“Fred”}, \text{“Ted”}, \text{“Ned”}\})\ \wedge\ (b \in 0\,..\,99)$

The Algorithm – When the Transaction Commits



The Algorithm – When the Transaction Commits



UNCHANGED

UNCHANGED exp an abbreviation for $exp' = exp$

UNCHANGED $\langle rmState, tmState, msgs \rangle$

$\equiv$ $\langle rmState, tmState, msgs \rangle' = \langle rmState, tmState, msgs \rangle$

by definition of UNCHANGED

$\equiv$ $\langle rmState', tmState', msgs' \rangle = \langle rmState, tmState, msgs \rangle$

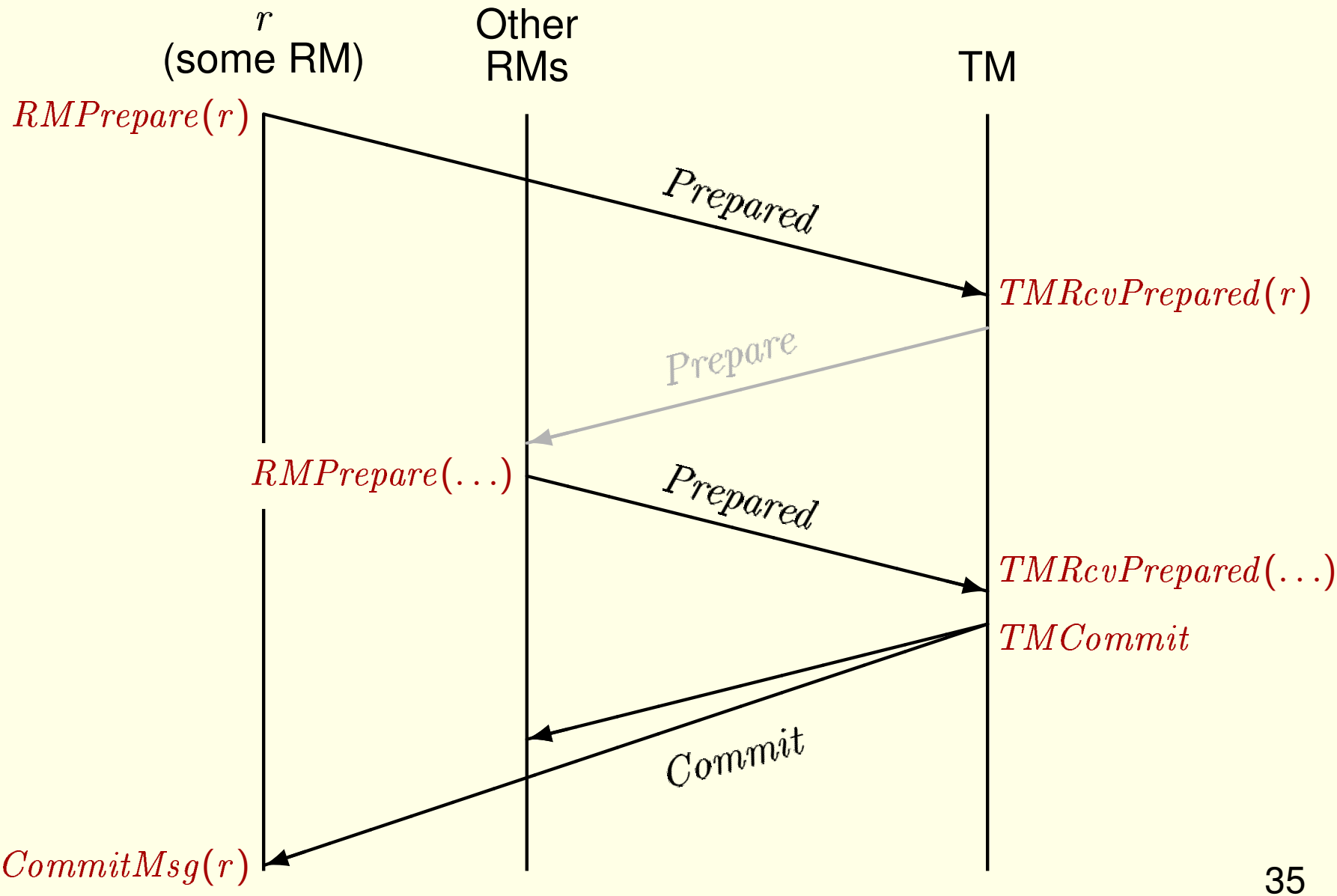
priming exp means priming all variables in it

$\equiv$ $(rmState' = rmState) \land (tmState' = tmState) \land (msgs' = msgs)$

two tuples are equal iff all their components are

Note that tuples are enclosed in $\langle \ldots \rangle$, typed «...».

The Algorithm – When the Transaction Commits



THEOREM $TPSpec \Rightarrow TC!TCSpec$

Asserts that any behavior satisfying $TPSpec$ of module $TwoPhase$ also satisfies $TCSpec$ of module $TCommit$.

But how can that make sense?

A state in a behavior of $TPSpec$ assigns values to $rmState$, $tmState$, $tmPrepared$, and $msgs$.

A state in a behavior of $TCSpec$ assigns a value only to $rmState$.

Remember: A state is an assignment of values to variables.

What variables?

All possible variables.

(There are infinitely many.)

Consider this state:

$$rmState = [r \in \{\text{“r1”}, \text{“r2”}, \text{“r3”}\} \mapsto \text{“working”}]$$

$$tmState = \text{“ouch”}$$

$$msgs = 3.14$$

$$numberOfCustomersInTimbuktuStarbucks = \sqrt{-37}$$

$$foo = \{\text{“bar”}\}$$

$$\vdots$$

Consider this state:

$$rmState = [r \in \{\text{“r1”}, \text{“r2”}, \text{“r3”}\} \mapsto \text{“working”}]$$
$$tmState = \text{“ouch”}$$
$$msgs = 3.14$$
$$numberOfCustomersInTimbuktuStarbucks = \sqrt{-37}$$
$$foo = \{\text{“bar”}\}$$
$$\vdots$$

It satisfies $TCInit$ iff $RM = \{\text{“r1”}, \text{“r2”}, \text{“r3”}\}$.

Remember: $TCInit \triangleq rmState = [r \in RM \mapsto \text{“working”}]$

Because formula $TCSpec$ contains only the variable $rmState$, whether or not a behavior satisfies $TCSpec$ depends only on the value assigned to $rmState$ by each of its states.

All the other variables can have any values in any of its states.

In a behavior satisfying formula TCSpec,
foo could have the value $\{$"bar"$\}$ in the first state,
the value $2^{48976553}$ in the second state,
the value $[a \mapsto 22, b \mapsto \{13, \{13\}, \{\{13\}\}\}]$ in the third state, ...

The theorem

THEOREM $TPSpec \Rightarrow TC!TCSpec$

makes sense since every behavior assigns values to the variables in $TPSpec$ and in $TC!TCSpec$.

Another Way to Look at This.

Let a *specification state* (also called a *system state*) be an assignment of values to the spec's variables.

The formula $TPSpec \Rightarrow TC!TCSpec$ asserts that:

> For every sequence of $TwoPhase$ states satisfying $TPSpec$, removing assignments to all variables except $rmState$ yields a sequence of $TCommit$ states satisfying $TCSpec$.

Note: The "*two-phase commit specification*" sometimes means the formula $TPSpec$, and sometimes means the entire module $TwoPhase$.

Is This Theorem True?

A behavior satisfying $TPSpec$ can have steps that leave $rmState$ unchanged – in particular, steps allowed by the three TM ... actions.

The next-state relation $TCNext$ of $TCommit$ does not allow such steps.

How can the theorem be true?

Remember: $TCSpec \triangleq TCInit \wedge \Box[TCNext]_{rmState}$

$TCInit \wedge \Box[TCNext]_{rmState}$

asserts that the initial state (of the behavior) satisfies $TCInit$.

$TCInit \wedge \Box[TCNext]_{rmState}$

asserts that every step satisfies $[TCNext]_{rmState}$.

$[TCNext]_{rmState}$

is an abbreviation for $TCNext \vee (\text{UNCHANGED } rmState)$.

The formula $TCSpec$ asserts of a behavior that:

– The initial state satisfies $TCInit$.

– Every step satisfies $TCNext$ or is a stuttering step (one that leaves $rmState$ unchanged).

Every TLA$^+$ spec we write allows stuttering steps.

I can give you several reasons why:

1. The state describes everything we can observe about the system at any instant, so a step that doesn't change the state is equivalent to nothing having happened.
2. . . .

The best reason:

THEOREM $TPSpec \Rightarrow TC!TCSpec$

Implementation is implication.

It's simple.

Simplicity is what specification is all about.

We only have to consider infinite behaviors.

A terminating execution is described by a behavior ending with infinitely many stuttering steps.

A behavior represents a possible history of the entire universe.

A behavior satisfying $TPSpec$ represents a history of a universe containing the two-phase commit algorithm.

The universe doesn't stop just because the algorithm does.

$$TPSpec \triangleq TPInit \land \Box[TPNext]_{\langle rmState, tmState, tmPrepared, msgs \rangle}$$

$TPSpec$ allows stuttering steps (ones that leave $rmState$, $tmState$, $tmPrepared$, and $msgs$ unchanged).

That's why it didn't matter if $TMRcvPrepared(r)$ allowed stuttering steps.

All our specs so far allow behaviors ending in infinite stuttering because they specify only safety, so they allow halting at any time.

Where's the Concurrency?

In $TwoPhase$, we model an execution of the two-phase commit algorithm by a sequence of global states.

But in a distributed system, each process has its own state.

You can't observe any global state.

How can this work?

At a low enough level of detail, an execution of any digital system can be modeled as a set of atomic events.

An atomic event might be a message arriving at a computer, or some bit flipping in a processor.

Atomic means that any two events act as if they occurred in some order.

If the two events are independent, then performing them in either order has the same effect.

We can therefore model an execution as an ordered sequence of events.

A system is described by the set of all it possible executions.

We can therefore model a system as a set of sequences of events.

We want a model that has the fewest events (largest grain of atomicity) that is useful for our purpose.

For example, we'd rather represent the receipt of a message as a single event, not as thousands of separate bit-flipping events.

$TCommit$ is a spec with few events; it's useful for describing what two-phase commit accomplishes.

To describe how two-phase commit works, we use $TwoPhase$, which has more events.

How do we describe sequences of events?

The way it has been done for 60 years: by a state machine that generates them. (From Mealy/Moore machines in the ’50s to Nancy Lynch’s I/O automata and beyond.)

Instead of having a state change generate an event, it’s simpler to let the state change *be* the event.

So TLA$^+$ just has states, not events.

Instead of events, we have state changes – which we call *steps*.

All methods of modeling systems that work in practice are based on atomic events.

All of them in which correctness is a property of an individual execution are isomorphic.

The simplest models describe an execution as a sequence of states.

I know three methods not based on atomic events. (I invented two of them.)

You should ignore them.

Paxos Commit

TLA$^+$ Course

Leslie Lamport

CHOOSE

CHOOSE $x \in S : P(x)$ equals an arbitrary value v such that $P(v)$ is true – if there is one.

Otherwise, its value is not specified. (But it has a value.)

For example, CHOOSE $n \in 0 \,..\, 42 : n > 13$ equals some unspecified value in $14 \,..\, 42$.

There's no nondeterminism.

If CHOOSE $n \in 0 \,..\, 42 : n > 13$ equals 17 now, then it equals 17 every time it's evaluated.

There are two ways in which you should use CHOOSE .

The first is when it's in an expression that has the same value no matter what value the CHOOSE chooses.

For example, there's only one value that can be chosen by the CHOOSE in the definition of $Maximum$:

$$Maximum(S) \triangleq$$
IF $S = \{\}$ THEN -1
ELSE CHOOSE $n \in S : (\forall\, m \in S : n \geq m)$

The parentheses aren't necessary.

The other way is rare; I may mention it later.

The Paxos Commit Algorithm

Paxos is an algorithm for getting processes to agree on a single value, even in the presence of failures. It works in a series of numbered rounds called *ballots*.

Paxos commit executes a separate instance of Paxos for each RM r to get agreement on whether r prepares or aborts.

There are three kinds of roles:

RMs.

Acceptors, which cast votes in ballots to determine what value is chosen in each Paxos instance.

Leaders, one for each ballot number > 0.
(Each RM is the ballot 0 leader for its instance of Paxos.)

SUBSET

SUBSET S is the set of all subsets of the set S.

In other words,

$T \in$ SUBSET S is equivalent to $T \subseteq S$.

For example, SUBSET $\{2, 4, 6\}$ equals:

$\{\ \{\},\ \{2\},\ \{4\},\ \{6\},\ \{2, 4\},\ \{2, 6\},\ \{4, 6\},\ \{2, 4, 6\}\ \}$

The TLA$^+$ Set Constructors

$\{x \in S : P(x)\}$

The subset of S consisting of all elements x of S for which $P(x)$ is true.

$\{n \in 2\,..\,25 : n \,\%\, 5 = 1\} \quad = \quad \{6, 11, 16, 21\}$

$\{e(x) : x \in S\}$

The set of all values $e(x)$ such that x is an element of S

$\{n^2 + 1 : n \in 1\,..\,4\} \quad = \quad \{2, 5, 10, 17\}$

Can also write $\{e(x, y) : x \in S, y \in T\}$ etc.

Review of Day 1

TLA$^+$ Course

Leslie Lamport

A specification comprises one or more modules.

EXTENDS $Integers$

Imports the standard module $Integers$ that defines

$+$ $-$ $*$ ^ (exponentiation) % $\div$ (integer division)

$>$ $<$ $\geq$ $\leq$ $..$ Nat

Definitions

$Init \triangleq \ldots$ $Min(m, n) \triangleq \ldots$

IF ... THEN ... ELSE

LET ... IN ...

We model a system as a set of behaviors, where a behavior is a sequence of states.

We represent a state as an assignment of values to variables.

We describe the behaviors by:

An initial predicate $Init$ describing possible initial states.

A next-state action $Next$ specifying possible state changes.

This specifies all behaviors $s_1 \rightarrow s_2 \rightarrow \cdots$ such that

- s_1 satisfies $Init$
- Each step $s_i \rightarrow s_{i+1}$ satisfies $Next$.

The set of behaviors is described by a single temporal logic formula $Init \wedge \Box[Next]_{\langle v_1,\ldots,v_n\rangle}$

Notation for functions (arrays)

$f[x]$ $[S \to T]$ $[x \in S \mapsto exp(x)]$ $[f \text{ EXCEPT } ![x] = v]$

$(d_1 :> v_1) \texttt{@@} \ldots \texttt{@@} (d_n :> v_n)$

Notation for records

A record is a function with domain a set of strings.

$rcd.name$

$[name_1 \mapsto val_1, \ldots, name_n \mapsto val_n]$

$[name_1 : set_1, \ldots, name_n : set_n]$

UNCHANGED $\langle v_1, \ldots, v_n \rangle$

Equivalent to $(v_1' = v_1) \land \ldots \land (v_n' = v_n)$

$[A]_v$ equals $A \lor (\text{UNCHANGED } v)$

Stuttering steps.

$TC \triangleq \text{INSTANCE } TCommit$

Imports definitions from $TCommit$ with renaming.

Safety Properties

Assert that something bad doesn't happen.

Liveness Properties

Assert that something good eventually does happen.

THEOREM $TPSpec \Rightarrow TC!TCSpec$

Implementation is implication.

CHOOSE $x \in S : P(x)$

SUBSET S

Finite Sequences

A tuple is another name for a finite sequence.

Sequences are functions.

The sequence $\langle -3, \text{“xyz”}, \{0, 2\}\rangle$ is the function:

$(1 :> -3) \mathbin{@@} (2 :> \text{“xyz”}) \mathbin{@@} (3 :> \{0, 2\})$

The sequence $\langle 1, 4, 9, \ldots, N^2\rangle$ is the function:

$[i \in 1 \,..\, N \mapsto i^2]$

Operators Defined in the Standard $Sequences$ Module

$Head(seq) \triangleq seq[1]$

$Tail(\langle s_1, \ldots, s_n \rangle)$ equals $\langle s_2, \ldots, s_n \rangle$.

- $\circ$ (concatenation, typed $\backslash\circ$)

  $\langle 3, 2, 1 \rangle \circ \langle \text{“a”}, \text{“b”} \rangle = \langle 3, 2, 1, \text{“a”}, \text{“b”} \rangle$

$Append(seq, e) \triangleq seq \circ \langle e \rangle$

$Len(seq)$ equals the length of sequence seq.

$Seq(S)$ is the set of all sequences with elements in S.

Exercise: Define $Remove(i, seq)$ to equal the sequence obtained by removing the i^{th} element from sequence seq

Alternating bit protocol (ABP) is a simple protocol for sending a sequence of data values from transmitter A to receiver B using a pair of lossy FIFO channels.

A sends to B messages containing a data part and a one-bit sequence number – i.e., a value that is 0 or 1. B sends to A messages containing a single acknowledgment bit that equals 0 or 1.

A sends a value by continually resending a message with that data value and the same sequence number, until it receives an acknowledgment from B with that sequence number. Then, A starts transmitting the next data value with the complement of the previous sequence number.

When B first receives a message that has sequence number 0, it accepts its data value as the value sent by A and starts sending the message 0. It keeps doing so, ignoring messages with sequence number 0, until it receives a message with sequence number 1. It then accepts that message's data value and starts sending 1. And so on.

This means that A may still receive messages with sequence number 0 when it is already transmitting messages with sequence number 1. (And vice versa.) It ignores such messages and continues transmitting its current message.

The protocol is initialized in a state in which A is sending a (bogus) message with sequence number 1 that B has already received , and there are no messages in transit.

Liveness

TLA$^+$ Course

Leslie Lamport

Enabled

An action A is *enabled* in a state s iff there exists a state t such that $s \to t$ is an A step.

Consider action A of $ABSpec$

$$\begin{array}{rcl} A & \triangleq & \land \ AVar = BVar \\ & & \land \ \exists\, d \in Data : AVar' = \langle d, 1 - AVar[2] \rangle \\ & & \land \ BVar' = BVar \end{array}$$

A is enabled iff $AVar = BVar$ and $Data \neq \{\}$.

Weak Fairness

Weak fairness of action A asserts of a behavior:

> If A remains continuously enabled,
> then an A step eventually occurs.

Or equivalently:

> A cannot remain enabled forever
> without any further A step occurring.

Or equivalently:

> A cannot remain enabled forever
> without infinitely many A steps occurring.

Weak fairness of A is written as the temporal formula $\mathrm{WF}_{vars}(A)$, where $vars$ is the tuple of all the spec's variables.

Don't worry about the $vars$ now.

$\mathrm{WF}_{vars}(Next)$ asserts that the system doesn't stop if it can take a step (isn't deadlocked / halted).

Leads-To ($\rightsquigarrow$)

$P \rightsquigarrow Q$ asserts of a behavior:

> If P is true in any state, then Q is true in that state or a later state.

Or more informally:

> Any time P is true, Q must be true then or later.

TRUE $\rightsquigarrow Q$ asserts of a behavior that Q is true in infinitely many states.

Strong Fairness

Strong fairness of action A asserts of a behavior:

> If A is repeatedly enabled,
> then an A step eventually occurs.

Even if it's repeatedly disabled.

Or equivalently:

> A cannot be enabled in infinitely many states
> without any further A step occurring.

Or equivalently:

> A cannot be enabled in infinitely many states
> without infinitely many A steps occurring.

Strong fairness of A is written as the temporal formula $\mathrm{SF}_{vars}(A)$, where $vars$ is the tuple of all the spec's variables.

Why the *vars*

$\mathrm{WF}_{vars}(A)$ asserts weak fairness of $A \wedge (vars' \neq vars)$.

An $A \wedge (vars' \neq vars)$ step is a non-stuttering A step. So $\mathrm{WF}_{vars}(A)$ asserts of a behavior:

> If a non-stuttering A step remains continuously enabled, then a non-stuttering A step eventually occurs.

There's no way to tell if a stuttering A step occurred.

Eventually ($\Diamond$)

$\Diamond P$ asserts of a behavior that there is some state satisfying P.

There need not be more than one such state.

Used to assert that a terminating system does something.

Silly Expressions

What does $1/0$ mean?

Some people say that $1/0$ is illegal and we shouldn't be allowed to write it.

That's stupid.

If $Real$ is the set of all real numbers, then this formula is true for all x:

$$(x \in Real) \wedge (x \neq 0) \Rightarrow (x * 1/x = 1)$$

In particular, it's true for $x = 0$:

$$(0 \in Real) \wedge (0 \neq 0) \Rightarrow (0 * 1/0 = 1)$$

Of course we should we be able to write this true statement.

It's also true for $x =$ "abc":

$$(\text{“abc”} \in Real) \wedge (\text{“abc”} \neq 0) \Rightarrow (\text{“abc”} * 1/\text{“abc”} = 1)$$

We should be able to write $1/$"abc" too.

The expressions $1/0$ and $1/$"abc" have values; we just don't know what those values are.

In fact, the definition of $/$ and the semantics of TLA$^+$ don't specify those values.

I call such expressions "silly". TLC will report an error if it has to evaluate a silly expression.

$(\text{“abc”} \in Real) \land (\text{“abc”} \neq 0) \Rightarrow (\text{“abc”} * 1 / \text{“abc”} = 1)$

Is this subformula false?

The semantics of TLA$^+$ don't specify if “abc” is a real number, or if it equals 0.

Even though the entire formula is true, TLC will report an error if it tries to evaluate it because it can't evaluate the subformula $(\text{“abc”} \in Real) \land (\text{“abc”} \neq 0)$.

It also reports an error if it tries to evaluate

$\{\text{“abc”}, 0\}$

because it doesn't know how many elements the set has.

The kind of error that would be caught by type checking is caught by TLC finding a silly expression – almost always on a tiny model.

Specifications are not programs.

Types may be good for writing programs.

The types used in programming languages add complexity and reduce expressiveness of a specification language.

Functions versus Operators

A function is an ordinary value – that is, a set.

TLA$^+$ doesn't specify what the elements of a function are.

This is a legal expression:

$$\langle f, f[42] \rangle$$

It's silly unless f is a function whose domain contains 42.

An operator that takes one or more arguments is not a value.

This is a syntactically illegal expression:

$$\langle\, Min,\ Min(42, -7)\,\rangle$$

$f \triangleq [n \in Nat \mapsto n^2 - 1]$

defines $f[42]$ to equal $42^2 - 1$

leaves $f[-1]$ unspecified.

$Op(n) \triangleq n^2 - 1$

defines $Op(42)$ to equal $42^2 - 1$

defines $Op(-1)$ to equal $(-1)^2 - 1$

defines $Op(\text{“abc”})$ to equal $(\text{“abc”})^2 - 1$

a silly expression

Function

- An ordinary value.
- Can be the value of a variable.
- But is defined only on its domain (a set).

Operator with Arguments

- An abbreviation (a macro).
- Defined on all values (more than in any set).
- Cannot be the value of a variable.

Recursion

Something that is computed by executing a loop is defined mathematically using recursion.

A mathematician might rigorously define $f(n)$ to equal

$$1^2 + 2^2 + \cdots + n^2$$

by writing

$$f(n) \triangleq \text{IF } n = 0 \text{ THEN } 0 \text{ ELSE } n^2 + f(n-1)$$

This is normally not allowed in TLA$^+$ because f can't be used before it has been defined.

The statement RECURSIVE $f(\_)$ allows such a use of f.

The definition can be written as:

RECURSIVE $f(\_)$

$f(n) \triangleq$ IF $n = 0$ THEN 0 ELSE $n^2 + f(n-1)$

You can use CHOOSE to loop through an arbitrary finite set.

For example, this defines $Cardinality(S)$ to be the number of elements in a finite set S:

RECURSIVE $Cardinality(\_)$

$Cardinality(S) \triangleq$

IF $S = \{\}$ THEN 0

ELSE $1 + Cardinality(S \setminus \{\text{CHOOSE } x \in S : \text{TRUE}\})$

Recursive Functions

Instead of defining an operator f to write $1^2 + \cdots + n^2$ as $f(n)$, we can define a function f to write it as $f[n]$.

The definition is

$$f[n \in Nat] \triangleq \text{IF } n = 0 \text{ THEN } 0 \text{ ELSE } n^2 + f[n-1]$$

No RECURSIVE declaration is needed.

If you haven't programmed in a functional language like ML or F♯, learning to use recursion instead of looping may take some time.

Because math is so expressive, TLA$^+$ often allows you to avoid recursion the required function languages.

The standard recursive definition of Len:

RECURSIVE $Len(\_)$

$$Len(seq) \triangleq \text{IF } seq = \langle\,\rangle \text{ THEN } 0 \text{ ELSE } 1 + Len(Tail(seq))$$

Here's the definition from the $Sequences$ module:

$$\text{CHOOSE } n \in Nat : (\text{DOMAIN } seq) = 1\,..\,n$$

Like any powerful tool, recursion should be used carefully

Here's Ackermann's function:

RECURSIVE $A(\_, \_)$

$$A(m, n) \triangleq \text{IF } m = 0 \text{ THEN } n + 1 \text{ ELSE IF } m > 0 \land n = 0 \text{ THEN } A(m - 1, 1) \text{ ELSE } A(m - 1, A(m, n - 1))$$

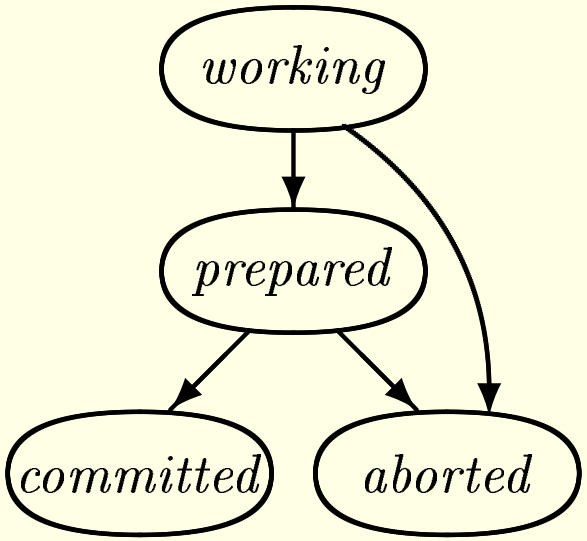
It's interesting because $A(3, 3) = 61$ and $A(4, 3)$ is too large to represent as a 32-bit Java integer.

Refinement Mappings

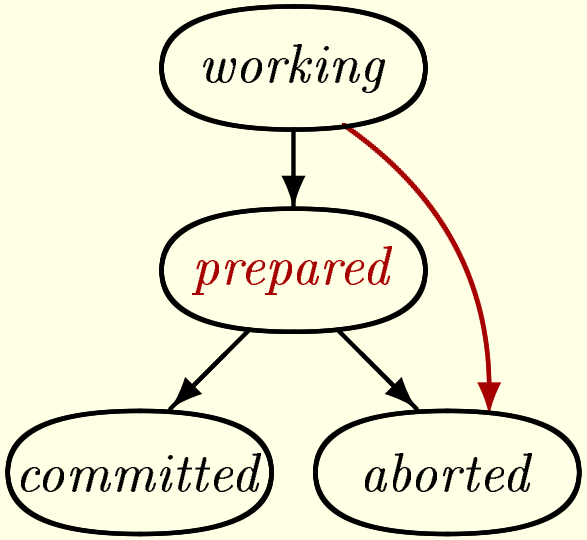
Let's make a silly modification to $TCommit$.

Suppose there are two special RMs named Mo and $Curly$.

We require that if Mo becomes $prepared$ when $Curly$ has not already been $prepared$, then no further $working \rightarrow aborted$ step can be taken.



We require that if Mo becomes $prepared$ when $Curly$ has not already been $prepared$, then no further $working \rightarrow aborted$ step can be taken.



To write the new specification $MCCommit$, we must add information to the state so an RM can determine if a $working \rightarrow aborted$ transition is allowed.

One way is to add a variable $whoFirst$ that initially equals $NoOne$ and is modified by:

$$
\begin{array}{l}
Prepare(r) \triangleq \\
\quad \land \ whoFirst' = \text{IF } (r \in \{Mo, Curly\}) \land (whoFirst = NoOne) \\
\qquad\qquad\qquad\qquad \text{THEN } r \\
\qquad\qquad\qquad\qquad \text{ELSE } whoFirst \\
\quad \land \ \ldots
\end{array}
$$

A $working \to aborted$ step can then be taken only if $whoFirst \neq Mo$.

We can easily write $MCTCommit$ with the variables $rmState$ and $whoFirst$.

Let $MCTwoPhase$ specify some implementation of two-phase commit satisfying the $Mo\text{-}Curly$ requirement, so $MCTwoPhase$ implements $MCTCommit$.

Exactly what does that mean?

$MCTwoPhase$ should require that the variable $rmState$ behave the way $MCCommit$ says it should. But $MCCommit$ also describes the behavior of $whoFirst$.

$whoFirst$ was added to $MCCommit$ only to specify how $rmState$ can change. Spec $MCTwoPhase$ doesn't have to say anything about $whoFirst$.

$MCTwoPhase$ implements $MCTCommit$ iff:

> For every behavior satisfying $MCTwoPhase$, there is a way of assigning values to $whoFirst$ in each state of the behavior that makes it satisfy $MCTCommit$.

For TLC to check this, we have to tell it what value to assign to $whoFirst$ in each state.

We do this by telling it to assign to $whoFirst$ in each state the value of an expression that can contain any of the variables $rmState, tmState, \ldots$ of $MCTwoPhase$.

Let's call that expression $f(rmState, tmState, \ldots)$.

We expect to be able to find the expression $f(rmState, tmState, \ldots)$ because the state of $MCTwoPhase$ must determine if a $working \to aborted$ step is permitted.

Let $MCTCSpec$ and $MCTPSpec$ be the specification formulas of $MCTCommit$ and $MCTwoPhase$, respectively. Corresponding to the INSTANCE statement and THEOREM of $TwoPhase$, we have:

$$MC \triangleq \text{INSTANCE } MCTCommit \text{ WITH } whoFirst \leftarrow f(rmState, tmState, \ldots)$$

$$\text{THEOREM } MCTPSpec \Rightarrow MC!MCTCSpec$$

$MC!MCTCSpec$ is formula $MCTCSpec$ with the expression $f(rmState, tmState, \ldots)$ substituted everywhere for the variable $whoFirst$.

We then let TLC check the temporal property $MC!MCTCSpec$.

We call $rmState$ a *visible* (or *interface*) variable and $whoFirst$ an *internal* variable of $MCTCommit$.

In general, a specification S may have visible variables $x_1, \ldots, x_m$ and internal variables $y_1, \ldots, y_n$.

To show that a lower-level spec T with variables $z_1, \ldots, z_p$ implements S, we have to express each y_i as an expression $f_i(z_1, \ldots, z_p)$.

The expressions $f_1, \ldots, f_n$ are called a *refinement mapping*.

Sometimes, one spec implements another spec, but there's no refinement mapping that shows it.

For example, $MCTCommit$ is implemented by the spec obtained from $TwoPhase$ by removing the parts of $TPNext$ that allow a $working \to aborted$ step.

It's possible to add auxiliary variables to the lower-level spec with which you can define a refinement mapping.

That's what you'll have to do if a customer gives you the high-level spec and says *implement it*.

But in real life, that's not what happens.

You get to write both the high-level and the low-level specs.

So, you can write the high-level spec so it doesn't introduce state that isn't implemented by the low-level spec.

A sufficiently low-level implementation of a spec won't contain any of the high-level variables.

For example, a spec of the AB protocol that accurately models the code won't have high-level variables like $AVar$ and $BVar$ that change atomically.

All the high-level variables are considered to be internal.

In that case, we say that the low-level spec implements the high-level one *under the refinement mapping*.

You're not going to write a TLA$^+$ spec at the code level.

You're going to write code at the code level.

The higher-level TLA$^+$ variables are going to be implemented by program state – variables, objects, etc.

Correctness of the code means that it implements the TLA$^+$ spec under a refinement mapping that defines the values of the TLA$^+$ variables in terms of the program state.

The program state is too complicated to write that refinement mapping formally, but you should understand it and document it informally.

The refinement mapping defines what it means for the code to implement the TLA$^+$ spec, making it possible to test that it does.

An Observation About How We Write Specs

$$SmallToBig \triangleq \text{LET } poured \triangleq \ldots \text{ IN } \land\ big' = big + poured \land\ small' = small - poured$$

Is this equivalent to $big' - big = poured$?

Suppose $big = 0$ and $poured = 3$.

$big' = 0 + 3$ implies $big' = 3$

Does $big' - 0 = 3$ also imply $big' = 3$?

What does “abc” $-$ 0 equal?

We don’t know.

It might equal 3.

If it does, then $big' - 0 = 3$ implies $big' = 3$ or $big' =$ “abc” or . . . .

So $big' - big = poured$ is not equivalent to $big' = big + poured$.

Fortunately, we naturally write $big' = big + poured$ because we think of $big' = \ldots$ as the C assignment statement `big = ...`

Writing Your Own Specs

Thanks to Chris Newcombe & Andrew Helwer

TLA$^+$ will not help you design a system.

Writing a TLA$^+$ spec will reveal if you have a design or just a lot of wishful thinking.

If you have a design, writing the spec will help you understand the design.

Model checking can find errors in your understanding.

Eventually, using TLA$^+$ will improve your thinking, which will improve your designs.

That won't happen right away.

For your first specs, specify *how* not *what*.

Specify the design.

Check properties: invariants and perhaps liveness properties.

Later on, you can think about specifying *what* by writing higher-level specs.

When you first try to write a non-trivial spec, you will probably not know where to start.

The first thing to ask is, what is the spec for?

A spec is written for a purpose.

Here are some possible purposes: To explain a design to someone. To make sure all the designers are working on the same design. To find bugs.

If the purpose is to find bugs, you should ask: what class of bugs?

A spec is an abstract model.

You have to choose the right level of abstraction.

You want an abstraction that has the details needed to reveal that class of bugs, but hides as many other details as possible.

I suggest writing down a single correct execution as a sequence of actions like A *sends message* B *to* C.

Use Nondeterminism to Simplify

In a real transaction commit protocol, a process will abort when certain conditions hold. The properties you want to check probably don't depend on those conditions. You can write a simpler spec in which the process nondeterministically decides whether to abort.

A distributed system should satisfy its safety properties if all timeouts are modeled as nondeterministic actions that don't depend on the passage of time.

You can't use CHOOSE to express nondeterminism.

~~$x' = \text{CHOOSE } v \in S : P$~~

$\exists\, v \in S : P \ \wedge\ (x' = v)$

Use CHOOSE only in an expression whose value doesn't depend on which possible value the CHOOSE chooses.

Or in the idiom:

$$c \triangleq \text{CHOOSE } v : v \notin S$$

where S is a constant.

Decide how to decompose the next-state action. This involves separating the subactions into categories, such as: User actions. Internal system actions. Environment actions that the system has no control over. Actions of human operators.

Remember that you're not just specifying the system. You're specifying the entire universe in which the system operates.

The specification of parts of the universe that you're not implementing will probably be much more abstract than the specification of the system's actions.

Don’t worry about re-use or modularity.

You’re writing a few hundred lines of spec, not many thousands of lines of code.

Cut-and-paste works fine for re-use.

Definitions provide all the modularity you need.

You may have trouble expressing non-trivial concepts in an unfamiliar language. For example: specify a directed graph and what it means for the graph to be connected.

If you can't figure something out after giving it a good try, ask for help. Post a question to the Outlook TLA Plus group. I and I hope Dave, Cheng, and Andrew will be monitoring the group and responding. Soon, there should be other experienced users who can answer it.

You should also join the public TLA+ Google group, where you can ask questions that you can sanitize not to reveal any Microsoft confidential information.

How to model check a spec with infinitely many reachable states.

- Use a state constraint.

- Override definitions.

  For example, use a model that redefines Nat to equal $0 \ldots 3$.

Be suspicious of success.

Always check the coverage statistics.

If you're not specifying liveness, check invariants asserting that a state that should be reachable isn't reached.

Check every property you can think of that should be true.

Add errors to the spec and make sure they're caught.