PAXOS COMMIT

This video should be viewed in conjunction with a Web page. To find that page, search the Web for TLA+ Video Course.
In this lecture, we study a specification of Paxos Commit – a fault-tolerant distributed algorithm that implements transaction commit. The spec illustrates most of the TLA+ constructs you don’t already know that you will use in writing specs.

I hope you’ll also study the algorithm itself. I think it’s neat, but then I’m prejudiced, since Jim Gray and I invented it. But that’s up to you. These lectures are about TLA+, not distributed algorithms.
THE ALGORITHM

The Paxos Commit algorithm.
The problem with two-phase commit:

There’s an obvious problem with two-phase commit:
The problem with two-phase commit:

It can hang forever if the TM fails.

There's an obvious problem with two-phase commit: It can hang forever if the transaction manager fails.
The problem with two-phase commit:
   It can hang forever if the TM fails.

A simple engineering solution:

There's an obvious problem with two-phase commit: It can hang forever if the transaction manager fails.

There's a simple engineering solution.
The problem with two-phase commit:

It can hang forever if the TM fails.

A simple engineering solution:

Have a backup TM take over if the TM fails.

There’s an obvious problem with two-phase commit: It can hang forever if the transaction manager fails.

There’s a simple engineering solution.

Have a backup transaction manager take over if the primary transaction manager fails.
The problem with two-phase commit:

It can hang forever if the TM fails.

A simple engineering solution:

Have a backup TM take over if the TM fails.

You can find it in textbooks.

You can find this solution in database textbooks.
The problem with two-phase commit:

It can hang forever if the TM fails.

A simple engineering solution:

Have a backup TM take over if the TM fails.

You can find it in textbooks.

It’s straightforward to implement.

You can find this solution in database textbooks.

It’s straightforward to implement.
The problem with two-phase commit:

It can hang forever if the TM fails.

A simple engineering solution:

Have a backup TM take over if the TM fails.

You can find it in textbooks.

It’s straightforward to implement and test that it works.

You can find this solution in database textbooks.

It’s straightforward to implement and to test that it works.
It’s deployed and works fine

The system is deployed and works fine, and everyone’s happy
It’s deployed and works fine until one day:

The primary TM decides to commit and then pauses.
The backup TM thinks the primary failed and it decides to take over.
The backup TM broadcasts an Abort message.
The primary TM resumes and broadcasts a Commit message.

Some RMs abort and others commit.

SYSTEM FAILURE

The system is deployed and works fine, and everyone’s happy until one day:
It’s deployed and works fine until one day:

The primary TM decides to commit

The system is deployed and works fine, and everyone’s happy until one day:

The primary transaction manager decides to commit
It’s deployed and works fine until one day:

The primary TM decides to commit and then pauses.

The system is deployed and works fine, and everyone’s happy until one day:

The primary transaction manager decides to commit and then pauses for some reason.

Perhaps it’s pre-empted by a higher priority task.
It’s deployed and works fine until one day:

The primary TM decides to commit and then pauses.

The backup TM thinks the primary failed and it decides to take over.

The backup transaction manager thinks the primary failed and it decides to take over.
It’s deployed and works fine until one day:

The primary TM decides to commit and then pauses.

The backup TM thinks the primary failed and it decides to take over.

**The backup TM broadcasts an *Abort* message.**
It’s deployed and works fine until one day:

The primary TM decides to commit and then pauses.

The backup TM thinks the primary failed and it decides to take over.

The backup TM broadcasts an *Abort* message.

The primary TM resumes and broadcasts a *Commit* message.

The backup transaction manager thinks the primary failed and it decides to take over.

The backup transaction manager broadcasts an *Abort* message.

Meanwhile, the primary transaction manager resumes and broadcasts a *Commit* message.
It’s deployed and works fine until one day:

The primary TM decides to commit and then pauses.

The backup TM thinks the primary failed and it decides to take over.

The backup TM broadcasts an *Abort* message.

The primary TM resumes and broadcasts a *Commit* message.

Some RMs abort and others commit.

The backup transaction manager thinks the primary failed and it decides to take over.

The backup transaction manager broadcasts an *Abort* message.

Meanwhile, the primary transaction manager resumes and broadcasts a *Commit* message.

This causes some resource managers to abort and others to commit.
It’s deployed and works fine until one day:

The primary TM decides to commit and then pauses.

The backup TM thinks the primary failed and it decides to take over. **SYSTEM FAILURE**

The backup TM broadcasts an *Abort* message.

The primary TM resumes and broadcasts a *Commit* message.

Some RMs abort and others commit.

Which constitutes a system failure.
Finding fault-tolerant distributed algorithms is hard.
Finding fault-tolerant distributed algorithms is hard.

They’re easy to get wrong
Finding fault-tolerant distributed algorithms is hard.

They’re easy to get wrong, and hard to find errors by testing.
Finding fault-tolerant distributed algorithms is hard.

They’re easy to get wrong, and hard to find errors by testing.

We should get the algorithm right before we code it.
Finding fault-tolerant distributed algorithms is hard. They’re easy to get wrong, and hard to find errors by testing. We should get the algorithm right before we code. Writing and checking a TLA+ spec is the best way I know to do that.
Paxos Commit is a fault-tolerant transaction-commit algorithm described in this 2006 paper by Jim Gray and me.

**Consensus on Transaction Commit**

Jim Gray and Leslie Lamport

*ACM Transactions on Database Systems (TODS)*

Volume 31, issue 1 (March 2006), pages 133–160
Paxos Commit is a fault-tolerant transaction-commit algorithm described in this paper:

**Consensus on Transaction Commit**
Jim Gray and Leslie Lamport

*ACM Transactions on Database Systems (TODS)*
Volume 31, issue 1 (March 2006), pages 133–160

The paper explains the algorithm and specifies it in module *PaxosCommit*.

Paxos Commit is a fault-tolerant transaction-commit algorithm described in this 2006 paper by Jim Gray and me.

The paper explains the algorithm and specifies it in a TLA+ module named *PaxosCommit*.
We’re looking at this module for two reasons:

– To see what a real spec looks like.
– To learn some more TLA+.

You can read the paper if you want to understand the algorithm. This lecture explains only the TLA+ operators you haven’t seen yet that are used in the spec.
We’re looking at this module for two reasons:

– To see what a real spec looks like.

You should read the paper if you want to understand the algorithm.
We’re looking at this module for two reasons:

- To see what a real spec looks like.
- To learn some more TLA+.

You should read the paper if you want to understand the algorithm.

The first is to see what a real spec looks like.

The second is to learn some more TLA+. 
We’re looking at this module for two reasons:

– To see what a real spec looks like.
– To learn some more TLA$^+$. 

You can read the paper if you want to understand the algorithm.

We’re looking at this module for two reasons:

The first is to see what a real spec looks like.
The second is to learn some more TLA$^+$. 

You should read the paper if you want to understand the algorithm.
We’re looking at this module for two reasons:

– To see what a real spec looks like.
– To learn some more TLA⁺.

You can read the paper if you want to understand the algorithm.

This lecture explains only the TLA⁺ operators you haven’t seen yet that are used in the spec.
Stop the video now and:

Download module $PaxosCommit$ to the same folder as $TCommit$.

Download the paper.
Stop the video now and:

Download module *PaxosCommit* to the same folder as *TCommit*.

Download the paper.

**Modules** *TCommit*, *TwoPhase*, *PaxosCommit* used in these lectures differ slightly from the ones in the paper.

Stop the video now and download module *PaxosCommit* to the same folder as module *TCommit*; and download the paper if you want to read it.

The module *PaxosCommit* that we use here, as well as modules *TCommit* and *TwoPhase* used in previous lectures, differ slightly from the ones in the paper.

[slide 33]
THE SPECIFICATION

The Paxos Commit algorithm’s specification
The module begins with an EXTENDS statement that imports the definition of arithmetic operators from the standard \textit{Integers} module.
The module then defines $Maximum(S)$
The largest element of the finite set $S$ of integers

The module then defines $Maximum(S)$ to be the largest element of $S$ if $S$ is a finite set of integers,
The largest element of the finite set $S$ of integers, or $-1$ if $S$ is the empty set.

The module then defines $\text{Maximum}(S)$ to be the largest element of $S$ if $S$ is a finite set of integers, and to equal $-1$ if it's the empty set.

We don’t care what it equals if $S$ is infinite or not a set of numbers.
The definition has this form

\[
\text{Minimum}(S) \triangleq \begin{cases} 
\text{IF } S = \emptyset & \text{THEN } -1 \\
\text{ELSE } & \text{smallest number in } S
\end{cases}
\]
$$\text{Maximum}(S) \triangleq$$

\[
\begin{align*}
\text{IF } S = \{\} & \text{ THEN } -1 \\
\text{ELSE} & \text{ smallest number in } S'
\end{align*}
\]

The definition has this form
The definition has this form

The smallest number in $S$ is written this way
\[ \text{Maximum}(S) \triangleq \]

\if S = \{\} \text{ THEN } -1 \]

\else \text{ CHOOSE } n \in S : n \geq \text{ every element in } S \]

where the CHOOSE expression

[slide 42]
\[ \text{Maximum}(S) \triangleq \]
\[
\begin{align*}
\text{IF } S = \{\} \text{ THEN } & -1 \\
\text{ELSE } & \text{CHOOSE } n \in S : \quad n \geq \text{ every element in } S
\end{align*}
\]

Equals an arbitrarily chosen \( n \) in \( S \)

where the CHOOSE expression equals an arbitrarily chosen value \( n \) in \( S \)
Maximum(S) ≡
IF S = {} THEN  − 1
ELSE CHOOSE n ∈ S : \( n \geq \text{every element in } S \)

Equals an arbitrarily chosen \( n \) in \( S \) satisfying ...
where the CHOOSE expression equals an arbitrarily chosen value \( n \) in \( S \) satisfying the condition that \( n \) is greater-than or equal to every element in \( S \). If \( n \) is finite and nonempty, then there is exactly one such \( n \).

That condition on \( n \) is written this way.

\[
Maximum(S) \triangleq \\
\text{IF } S = \{\} \text{ THEN } -1 \\
\text{ELSE } \text{CHOOSE } n \in S : \forall m \in S : n \geq m
\]
It's a little easier to read with parentheses.
Maximum(S) \triangleq
\begin{align*}
\text{IF } S = \{\} \text{ THEN } & -1 \\
\text{ELSE CHOOSE } n \in S : (\forall m \in S : n \geq m)
\end{align*}

It's a little easier to read with parentheses.

This formula states that for every $m$ in $S$, $n$ is greater than or equal to $m$. 

[slide 47]
**CHOOSE** $v \in S : P$ equals

In general, the expression **CHOOSE** variable $v$ in expression $S$ colon formula $P$ equals
\textbf{CHOOSE } v \in S : P \text{ equals }

\textbf{if } there is a \ v \text{ in } S \text{ for which } P \text{ is true }

In general, the expression \textbf{CHOOSE variable } v \text{ in expression } S \text{ colon formula } P \text{ equals }

If there is at least one value \ v \text{ in the set } S \text{ for which formula } P \text{ is true }
\textbf{CHOOSE} \ v \in \ S : \ P \quad \text{equals}\quad$
\begin{align*}
\textbf{if} & \quad \text{there is a} \ v \ \text{in} \ S \ \text{for which} \ P \ \text{is true} \\
\textbf{then} & \quad \text{some such} \ v
\end{align*}$

In general, the expression \textbf{CHOOSE} \ variable \ v \ \text{in expression} \ S \ \text{colon formula} \ P \ \text{equals} \quad$
\begin{align*}
\text{If there is at least one value} \ v \ \text{in the set} \ S \ \text{for which formula} \ P \ \text{is true} \\
\text{then the expression equals some such} \ v. \\
\text{If there’s more than one, then the semantics of TLA+ don’t specify which one.}
\end{align*}$
CHOOSE $v \in S : P$ equals

if there is a $v$ in $S$ for which $P$ is true
then some such $v$
else a completely unspecified value.

Else, If there is no such $v$, then the value of the CHOOSE expression is completely unspecified.

And TLC will report an error if that’s the case when it tries to evaluate the expression.
\[
\text{CHOOSE } \; v \in S : \; P \quad \text{equals}
\]

\[
\begin{align*}
\text{if} & \quad \text{there is a } v \in S \text{ for which } P \text{ is true} \\
\text{then} & \quad \text{some such } v \\
\text{else} & \quad \text{a completely unspecified value.}
\end{align*}
\]

\[
\text{CHOOSE } \; i \in 1 \ldots 99 : \; \text{TRUE}
\]

Is an unspecified integer between 1 and 99.

Else, If there is no such \(v\), then the value of the \text{CHOOSE} expression is completely unspecified.

And TLC will report an error if that’s the case when it tries to evaluate the expression.

For example: this expression equals an unspecified integer between 1 and 99. We don’t know which one.
CHOOSE \( v \in S : P \) equals
\[
\text{if there is a } v \text{ in } S \text{ for which } P \text{ is true} \\
\text{then some such } v \\
\text{else a completely unspecified value.}
\]

CHOOSE \( i \in 1..99 : \text{TRUE} \)

Is an unspecified integer between 1 and 99.

It might or might not equal 37.

It might equal 37, or it might not; the semantics of TLA+ don’t say.
In math, any expression equals itself.

In math, any expression always equals itself.
In math, any expression equals itself.

\[(\text{CHOOSE } i \in 1 \ldots 99 : \text{TRUE}) = (\text{CHOOSE } i \in 1 \ldots 99 : \text{TRUE})\]

In math, any expression always equals itself.

So this \texttt{CHOOS}E expression always equals itself.
In math, any expression equals itself.

\[(\text{CHOOSE } i \in 1 \ldots 99 : \text{TRUE}) = (\text{CHOOSE } i \in 1 \ldots 99 : \text{TRUE})\]

There is no nondeterminism in a mathematical expression.

In math, any expression always equals itself.

So this \text{CHOOSE} expression always equals itself.

There is no nondeterminism in any mathematical expression, including a \text{CHOOSE} expression.

[slide 56]
In math, any expression equals itself.

\[
(\text{CHOOSE } i \in 1 \ldots 99 : \text{TRUE}) = (\text{CHOOSE } i \in 1 \ldots 99 : \text{TRUE})
\]

There is no nondeterminism in a mathematical expression.

If \( \text{CHOOSE } i \in 1 \ldots 99 : \text{TRUE} \) equals 37 today;
it will equal 37 next week.

If this \( \text{CHOOSE} \) expression equals 37 today, it will still equal 37 next week.
In math, any expression equals itself.

\[(\text{CHOOSE } i \in 1 \ldots 99 : \text{TRUE}) = (\text{CHOOSE } i \in 1 \ldots 99 : \text{TRUE})\]

There is no nondeterminism in a mathematical expression.

If \( \text{CHOOSE } i \in 1 \ldots 99 : \text{TRUE} \) equals 37 today; it will equal 37 next week.

TLC will always get the same number when it evaluates it.

If this \( \text{CHOOSE} \) expression equals 37 today, it will still equal 37 next week.

TLC will always get the same number when it evaluates this expression.

You shouldn’t care what number.
\( x' \in 1 \ldots 99 \)

Allows the value of \( x \) in the next state to be any number in \( 1 \ldots 99 \).

The formula \( x \) prime in the set 1 dot dot 99 allows the value of \( x \) in the next state to be any of the 99 numbers from 1 to 99.
\( x' \in 1 \ldots 99 \)

Allows the value of \( x \) in the next state to be any number in \( 1 \ldots 99 \).

\[ x' = \text{CHOOSE } i \in 1 \ldots 99 : \text{TRUE} \]

Allows the value of \( x \) in the next state to be one particular number.

The formula \( x \) prime in the set 1 dot dot 99 allows the value of \( x \) in the next state to be any of the 99 numbers from 1 to 99.

The formula \( x' \) equals this CHOOSE expression allows the value of \( x \) in the next state to be some particular number between 1 and 99 — perhaps 37.
\( x' \in 1 \ldots 99 \)

Allows the value of \( x \) in the next state to be any number in 1 \ldots 99.

\[
x' = \text{CHOOSE } i \in 1 \ldots 99 : \text{TRUE}
\]

Allows the value of \( x \) in the next state to be one particular number.

The formula \( x \) prime in the set 1 dot dot 99 allows the value of \( x \) in the next state to be any of the 99 numbers from 1 to 99.

The formula \( x' \) equals this CHOOSE expression allows the value of \( x \) in the next state to be some particular number between 1 and 99 — perhaps 37.

There’s no reason why you’d ever want to write something like this.
You should write \[ \text{CHOOSE} \ v \in S : P \]

Only when there’s exactly one \( v \) in \( S \) satisfying \( P \).

You should write this \text{CHOOSE} expression only when there’s exactly one value \( v \) in \( S \) satisfying formula \( P \).
You should write \textbf{CHOOSE} \( v \in S : P \)

Only when there’s exactly one \( v \) in \( S \) satisfying \( P \).
As in the definition of \( \text{Maximum}(S) \).

You should write this \textbf{CHOOSE} expression only when there’s exactly one value \( v \) in \( S \) satisfying formula \( P \).

For example, the way it was used in the definition of \( \text{Maximum} \) of \( S \).
You should write \( \text{CHOOSE } v \in S : P \)

Only when there’s exactly one \( v \) in \( S \) satisfying \( P \).

Or when it’s part of a larger expression whose value doesn’t depend on which \( v \) is chosen.

You should write this \textbf{CHOOSE} expression only when there’s exactly one value \( v \) in \( S \) satisfying formula \( P \).

For example, the way it was used in the definition of \textit{Maximum} of \( S \).

Or when it’s part of a larger expression whose value doesn’t depend on which possible value of \( v \) is chosen.

We’ll see an example of that later.
After defining *Maximum*, the module contains a CONSTANTS statement declaring these four constants.

CONSTANTS \( RM \), \( Acceptor \), \( Majority \), \( Ballot \)
After defining *Maximum*, the module contains a CONSTANTS statement declaring these four constants

*RM* is again the set of resource managers
After defining *Maximum*, the module contains a CONSTANTS statement declaring these four constants

*RM* is again the set of resource managers and *Acceptors* is another a set of processes called acceptors.
After defining *Maximum*, the module contains a CONSTANTS statement declaring these four constants:

*RM* is again the set of resource managers and *Acceptor* is another a set of processes called acceptors.

The constants *Majority* and *Ballot* are sets described in the following statement.

[slide 68]
CONSTANTS $RM$, $Acceptor$, $Majority$, $Ballot$

This $ASSUME$ statement asserts assumptions being made about the constants.
CONSTANTS $RM$, $Acceptor$, $Majority$, $Ballot$

ASSUME
\[
\begin{align*}
&\land Ballot \subseteq Nat \\
&\land 0 \in Ballot \\
&\land Majority \subseteq \text{subset } Acceptor \\
&\land \forall MS1, MS2 \in Majority : MS1 \cap MS2 \neq \emptyset
\end{align*}
\]

This ASSUME statement asserts assumptions being made about the constants.
CONSTANTS $RM$, $Acceptor$, $Majority$, $Ballot$

ASSUME

$\land Ballot \subseteq Nat$
$\land 0 \in Ballot$
$\land Majority \subseteq \text{subset } Acceptor$
$\land \forall MS1, MS2 \in Majority : MS1 \cap MS2 \neq {}$

This ASSUME statement asserts assumptions being made about the constants.

For example, the second conjunct asserts the assumption that zero is an element of the set $Ballot$. 

[slide 71]
CONSTANTS $RM$, $Acceptor$, $Majority$, $Ballot$

ASSUME
\[ \land Ballot \subseteq Nat \]
\[ \land 0 \in Ballot \]
\[ \land Majority \subseteq \text{SUBSET} \ Acceptor \]
\[ \land \forall MS1, MS2 \in Majority : MS1 \cap MS2 \neq \emptyset \]

These assumptions use some TLA+ notation that you haven’t seen yet.
CONSTANTS $RM$, $Acceptor$, $Majority$, $Ballot$

ASSUME

$\land Ballot \subseteq \textbf{Nat}$

the set of natural numbers

These assumptions use some TLA+ notation that you haven’t seen yet.

$\textbf{Nat}$ is defined in the imported $Integers$ module to be the set of natural numbers (that is, the non-negative integers).
CONSTANTS $RM$, $Acceptor$, $Majority$, $Ballot$

ASSUME

$\land Ballot \subseteq Nat$

$Ballot$ is a subset of $Nat$.

These assumptions use some TLA+ notation that you haven’t seen yet.

$Nat$ is defined in the imported $Integers$ module to be the set of natural numbers (that is, the non-negative integers).

The first conjunct asserts that $Ballot$ is a subset of $Nat$, meaning that every element of $Ballot$ is an element of the set $Nat$ of natural numbers.

[slide 74]
CONSTANTS $RM$, $Acceptor$, $Majority$, $Ballot$

ASSUME

$\land Ballot \subseteq Nat$

\subsetneq
CONSTANTS $RM$, $Acceptor$, $Majority$, $Ballot$

ASSUME

$\land Majority \subseteq \text{SUBSET } Acceptor$

the set of all subsets of $Acceptor$

The subset symbol is typed backslash subset e-q.

SUBSET $Acceptor$ is the set of all subsets of the set $Acceptor$. 
CONSTANTS $RM, Accepter, Majority, Ballot$

ASSUME

$$\land Majority \subseteq \text{SUBSET } Accepter$$

the set of all subsets of $Accepter$

Also called the \textit{powerset of $Accepter$}, written $P(Accepter)$

The subset symbol is typed backslash subset e-q.

\textit{SUBSET } Accepter is the set of all subsets of the set $Accepter$.

Mathematicians call it the powerset of $Accepter$ and write it $P$ of $Accepter$.

[slide 77]
CONSTANTS $RM$, $Acceptor$, $Majority$, $Ballot$

ASSUME

\[ \wedge Majority \subseteq \text{subset } Acceptor \]

The elements of $Majority$ are subsets of $Acceptor$.

The subset symbol is typed backslash subset e-q.

SUBSET $Acceptor$ is the set of all subsets of the set $Acceptor$.

Mathematicians call it the powerset of $Acceptor$ and write it $P$ of $Acceptor$.

The conjunct asserts the assumption that every element of $Majority$ is a subset of the set $Acceptor$.

[slide 78]
CONSTANTS $RM, Acceptor, Majority, Ballot$

ASSUME

\[ \forall MS1, MS2 \in Majority : MS1 \cap MS2 \neq \{\} \]

intersection of $MS1$ and $MS2$

This subexpression is the intersection of the sets $MS1$ and $MS2$. 

[slide 79]
CONSTANTS $RM$, $Acceptor$, $Majority$, $Ballot$

ASSUME

\[ \forall MS_1, MS_2 \in Majority : MS_1 \cap MS_2 \neq \{\} \]

the set of elements in both $MS_1$ and $MS_2$

This subexpression is the intersection of the sets $MS_1$ and $MS_2$.

It’s the set consisting of all elements in both $MS_1$ and $MS_2$. 

[slide 80]
CONSTANTS \(RM, Acceptor, Majority, Ballot\)

ASSUME

\[\forall MS1, MS2 \in Majority : MS1 \cap MS2 \neq \{\}\]

This subexpression is the intersection of the sets \(MS1\) and \(MS2\).

It’s the set consisting of all elements in both \(MS1\) and \(MS2\).

The intersection symbol is typed either backslash intersect or backslash cap.
CONSTANTS $RM$, $Acceptor$, $Majority$, $Ballot$

ASSUME

\[ \forall MS1, MS2 \in Majority : MS1 \cap MS2 \neq \{ \} \]

Any two elements of $Majority$ have an element in common.

This subexpression is the intersection of the sets $MS1$ and $MS2$.

It’s the set consisting of all elements in both $MS1$ and $MS2$.

The intersection symbol is typed either backslash intersect or backslash cap.

The conjunct asserts that every two elements of the set $Majority$ are sets having at least one element in common.

[slide 82]
CONSTANTS $RM$, $Acceptor$, $Majority$, $Ballot$

ASSUME
\[\land Ballot \subseteq Nat\]
\[\land 0 \in Ballot\]
\[\land Majority \subseteq \text{subset } Acceptor\]
\[\land \forall MS1, MS2 \in Majority : MS1 \cap MS2 \neq {}\]

TLC will check these assumptions.
The module next defines \textit{Messages} to be a set consisting of several kinds of records.

\[
\text{Messages} \triangleq \\
[\text{type : \{"phase1a"\}, ins : RM, bal : Ballot \setminus \{0\}] \\
\cup \\
[\text{type : \{"phase1b"\}, ins : RM, mbal : Ballot, bal : Ballot \cup \{-1\}, val : \{"prepared", "aborted", "none"\}, acc : Acceptor}] \\
\cup \\
[\text{type : \{"phase2a"\}, ins : RM, bal : Ballot, val : \{"prepared", "aborted"\}] \\
\cup \\
[\text{type : \{"phase2b"\}, acc : Acceptor, ins : RM, bal : Ballot, val : \{"prepared", "aborted"\}] \\
\cup \\
[\text{type : \{"Commit", "Abort"\}]}
\]
The module next defines $Messages$ to be a set consisting of several kinds of records. The definition contains this expression.
The module next defines *Messages* to be a set consisting of several kinds of records. The definition contains this expression.
The module next defines *Messages* to be a set consisting of several kinds of records. The definition contains this expression.

This *set minus* operator is defined as follows. For any sets $S$ and $T$, 

[slide 87]
The module next defines \textit{Messages} to be a set consisting of several kinds of records. The definition contains this expression.

This \textit{set minus} operator is defined as follows. For any sets $S$ and $T$,

$S$ set-minus $T$ is the set of all elements in $S$ that are not in $T$.

[slide 88]
For example, the integers from 10 to 20 set-minus the integers from 1 to 14 equals the set of integers from 15 to 20.
The set of non-0 elements in \( Balot \). 

For example, the integers from 10 to 20 set-minus the integers from 1 to 14 equals the set of integers from 15 to 20.

So, \( Balot \) set-minus the set containing only 0 is the set of non-zero elements in \( Balot \).
VARIABLES $rmState$, $aState$, $msgs$

$$PCTypeOK \triangleq \quad \wedge \quad rmState \in [RM \rightarrow \{ \text{"working"}, \text{"prepared"}, \text{"committed"}, \text{"aborted"} \}] \quad \wedge \quad aState \in [RM \rightarrow [\text{Acceptor} \rightarrow [mbal : Ballot, \quad bal : Ballot \cup \{ -1 \}, \quad val : \{ \text{"prepared"}, \text{"aborted"}, \text{"none"} \}]]]] \quad \wedge \quad msgs \subseteq Messages$$

The module next declares its variables and defines the type-correctness invariant $PCTypeOK$. 

[slide 91]
The module next declares its variables and defines the type-correctness invariant $PCTypeOK$.

As in the two-phase commit spec, there is a variable $m-s-g-s$ whose value is a set of messages.

[slide 92]
VARIABLES \( rmState, aState, msgs \)

\[ PCTypeOK \triangleq \]
\[
\land \ rmState \in [RM \to \{ \text{"working"}, \text{"prepared"}, \text{"committed"}, \text{"aborted"} \}]
\land \ aState \in [RM \to \{\text{Acceptor} \to [mbal : Ballot, bal : Ballot \cup \{-1\}, val : \{\text{"prepared"}, \text{"aborted"}, \text{"none"} \}]]]
\land \ msgs \subseteq Messages
\]

\[ PCTypeOK \text{ also asserts that the value of the variable } aState \]
VARIABLES $rmState, aState, msgs$

\[ PCTypeOK \triangleq \]
\[ \land rmState \in [RM \rightarrow \{\text{“working”), “prepared”), “committed”), “aborted”)\}] \]
\[ \land aState \in [RM \rightarrow [Acceptor \rightarrow [mbal : Ballot,] \]
\[ bal : Ballot \cup \{-1\}, \]
\[ val : \{\text{“prepared”), “aborted”), “none”)\}]] \]
\[ \land msgs \subseteq Messages \]

$PCTypeOK$ also asserts that the value of the variable $aState$ is a function with domain $RM$
\textbf{VARIABLES} \( rmState, \ aState, \ msgs \)

\( PCTypeOK \ \triangleq \)
\begin{align*}
\land \ rmState & \in [RM \rightarrow \{ \text{“working”}, \ \text{“prepared”}, \ \text{“committed”}, \ \text{“aborted”} \}] &
\land \ aState & \in [RM \rightarrow \{ \text{Acceptor} \rightarrow \{ \text{mbal} : \text{Ballot}, \ \text{bal} : \text{Ballot} \cup \{ -1 \}, \ \text{val} : \{ \text{“prepared”}, \ \text{“aborted”}, \ \text{“none”} \} \}] &
\land \ msgs & \subseteq \text{Messages}
\end{align*}

\( r \in RM \ \text{implies} \ aState[r] \) \text{ is a function with domain } \text{Acceptor}.

\( PCTypeOK \) also asserts that the value of the variable \( aState \) is a function with domain \( RM \) such that for every \( r \) in \( RM \), \( aState[r] \) is a function with domain \( \text{Acceptor} \).
VARIABLES \( rmState, aState, msgs \)

\[ PCTypeOK \triangleq \]
\[ \land rmState \in [RM \rightarrow \{ \text{“working”}, \text{“prepared”}, \text{“committed”}, \text{“aborted”} \}] \]
\[ \land aState \in [RM \rightarrow [\text{Acceptor} \rightarrow [mbal : \text{Ballot}, \]
\[ \quad bal : \text{Ballot} \cup \{-1\}, \]
\[ \quad val : \{\text{“prepared”}, \text{“aborted”}, \text{“none”}\}]]] \]
\[ \land msgs \subseteq \text{Messages} \]

\( r \in RM \) implies \( aState[r] \) is a function with domain \( \text{Acceptor} \).

\( a \in \text{Acceptor} \) implies \( aState[r][a] \) is a record with three fields.

\( PCTypeOK \) also asserts that the value of the variable \( aState \) is a function with domain \( RM \) such that for every \( r \) in \( RM \), \( aState[r] \) is a function with domain \( \text{Acceptor} \) such that for every \( a \) in the set \( \text{Acceptor} \), \( aState[r][a] \) is a record these three fields.
VARIABLES \( rmState, aState, msgs \)

\[ PCTypeOK \triangleq \]
\[ \land rmState \in [RM \to \{ \text{"working"}, \text{"prepared"}, \text{"committed"}, \text{"aborted"} \}] \]
\[ \land aState \in [RM \to [\text{Acceptor} \to [mbal : \text{Ballot}, \]
\[ \quad bal : \text{Ballot} \cup \{-1\}, \]
\[ \quad val : \{ \text{"prepared"}, \text{"aborted"}, \text{"none"} \}]]] \]
\[ \land msgs \subseteq Messages \]

\( r \in RM \) implies \( aState[r] \) is a function with domain \( \text{Acceptor} \).

\( a \in \text{Acceptor} \) implies \( aState[r][a] \) is a record with three fields.

\( aState[r][a].bal \) is in \( \text{Ballot} \) or equals \(-1\).

\[ PCTypeOK \] also asserts that the value of the variable \( aState \) is a function with domain \( RM \) such that for every \( r \) in \( RM \), \( aState[r] \) is a function with domain \( \text{Acceptor} \) such that for every \( a \) in the set \( \text{Acceptor} \), \( aState[r][a] \) is a record these three fields.

And, for example, \( aState[r][a].bal \) is in the set \( \text{Ballot} \) or equals \(-1\).
There’s nothing new here.

VARIABLES \( rmState, aState, msgs \)

\[ PCTypeOK \triangleq \]
\[ \land \ W \in [RM \rightarrow \{ \text{“working”, “prepared”, “committed”, “aborted”} \}] \]
\[ \land \ M \in [RM \rightarrow [Acceptor \rightarrow [mbal : Ballot, \]
\[ \quad \quad \quad \quad bal : Ballot \cup \{ -1 \}, \]
\[ \quad \quad \quad \quad val : \{ \text{“prepared”, “aborted”, “none”} \}]]] \]
\[ \land \ G \subseteq Messages \]

There’s nothing new here; it’s just a little more complicated than the formulas you’ve seen so far.

That’s true for what follows in the module, up until

[slide 98]
This definition of $Phase_{2a}$, which introduces several new features of TLA+.
This definition of \textit{Phase2a}, which introduces several new features of TLA+.

The first is this \texttt{LET-IN} expression.

\begin{verbatim}
LET mset \triangleq \{ m \in msgs : \land m.type = "phase1b"
\land m.ins = r
\land m.mbal = bal
\land m.acc \in MS \}
maxbal \triangleq \text{Maximum}\{m.bal : m \in mset\}
val \triangleq \text{IF maxbal} = -1
\text{THEN} "aborted"
\text{ELSE} (\text{CHOOSE} m \in mset : m.bal = maxbal).val
\end{verbatim}

\begin{verbatim}
\text{IN} \land \forall ac \in MS : \exists m \in mset : m.acc = ac
\land \text{Send([type \mapsto "phase2a", ins \mapsto r, bal \mapsto bal, val \mapsto val]})
\end{verbatim}
This definition of \textit{Phase2a}, which introduces several new features of TLA+.

The first is this \texttt{LET-IN} expression.
This definition of \textit{Phase2a}, which introduces several new features of TLA+.

The first is this \texttt{LET-IN} expression.

The \texttt{LET} clause makes three definitions local to the let-in expression.
This definition of Phase2a, which introduces several new features of TLA+.

The first is this LET-IN expression.

The LET clause makes three definitions local to the let-in expression.

The defined identifiers can be used only in the expression.
The next TLA+ notation introduced here is

\[ \text{LET } \ mset \overset{\Delta}{=} \ \{ \ m \in \ msgs : \ \land \ m.\text{type} = \text{“phase1b”} \ \\
\land m.\text{ins} = r \\
\land m.\text{mbal} = \text{bal} \\
\land m.\text{acc} \in MS \ \} \]

\[ \maxbal \overset{\Delta}{=} \text{Maximum}\left(\{m.\text{bal} : m \in mset\}\right) \]

\[ \text{val} \overset{\Delta}{=} \text{IF } \maxbal = -1 \ \\
\text{THEN } \text{“aborted”} \ \\
\text{ELSE } (\text{CHOOSE } m \in mset : m.\text{bal} = \maxbal).\text{val} \]

\[ \begin{align*}
\land \forall \ ac \in MS : \exists m \in mset : m.\text{acc} = ac \\
\land \text{Send}\left([\text{type} \mapsto \text{“phase2a”}, \text{ins} \mapsto r, \text{bal} \mapsto \text{bal}, \text{val} \mapsto \text{val}]\right)
\end{align*} \]
The next TLA+ notation introduced here is this set expression. It equals

\[
\{ m \in msgs : \land m.type = \text{"phase1b"} \\
\land m.ins = r \\
\land m.mbal = bal \\
\land m.acc \in MS \}
\]
The subset of \( msgs \) consisting of all its elements \( m \) satisfying this formula.

\[
\{ m \in msgs : \\land m.type = \text{“phase1b”} \\
\land m.ins = r \\
\land m.mbal = bal \\
\land m.acc \in MS \}
\]

The next TLA+ notation introduced here is this set expression. It equals The subset of \( msgs \)
\[ \{ m \in msgs : \land m.type = \text{"phase1b"} \land m.ins = r \land m.mbal = bal \land m.acc \in MS \} \]

The subset of \( msgs \) containing all elements \( m \)

The next TLA+ notation introduced here is this set expression. It equals The subset of \( msgs \) consisting of all its elements \( m \)
The subset of \( msgs \) containing all elements \( m \) satisfying this formula.

\[
\{ m \in msgs : \land m.\text{type} = \text{“phase1b”} \\
\land m.\text{ins} = r \\
\land m.\text{mbal} = bal \\
\land m.\text{acc} \in MS \}
\]

The next TLA+ notation introduced here is this set expression. It equals The subset of \( msgs \) consisting of all its elements \( m \) satisfying this formula.
The LET-IN expression also introduces another set notation.
The **LET-IN** expression also introduces another set notation.

\[ \{ m.\text{bal} : m \in mset \} \]
The set of all \( m.\text{bal} \)

The \texttt{LET-IN} expression also introduces another set notation.

This expression equals the set of all elements of the form \( m.\text{bal} \)
The set of all $m.bal$ with $m$ in $mset$.

The **LET-IN** expression also introduces another set notation.

This expression equals the set of all elements of the form $m.bal$ for all $m$ in the set $mset$. 

[slide 112]
Two Set Constructors

These are two different set constructors.
Two Set Constructors

\{ v \in S : P \}

These are two different set constructors.

The first has the form variable \( v \) in set \( S \) colon formula \( P \).
Two Set Constructors

\[ \{ v \in S : P \} \]

the subset of \( S \) consisting of all \( v \) satisfying \( P \)

These are two different set constructors.

The first has the form variable \( v \) in set \( S \) colon formula \( P \).

It’s the subset of \( S \) consisting of all values \( v \) for which the formula \( P \) is true.
Two Set Constructors

\[ \{ v \in S : P \} \]

the subset of \( S \) consisting of all \( v \) satisfying \( P \)

\[ \{ n \in Nat : n > 17 \} \]

These are two different set constructors.

The first has the form variable \( v \) in set \( S \) colon formula \( P \).

It’s the subset of \( S \) consisting of all values \( v \) for which the formula \( P \) is true.

For example, this expression
Two Set Constructors

\[ \{ v \in S : P \} \]

the subset of \( S \) consisting of all \( v \) satisfying \( P \)

\[ \{ n \in Nat : n > 17 \} = \{ 18, 19, 20, \ldots \} \]

the set of all natural numbers greater than 17

These are two different set constructors.
The first has the form variable \( v \) in set \( S \) colon formula \( P \).
It’s the subset of \( S \) consisting of all values \( v \) for which the formula \( P \) is true.
For example, this expression equals the set of all natural numbers greater than 17.

[slide 117]
Two Set Constructors

\{ v \in S : P \}

the subset of \( S \) consisting of all \( v \) satisfying \( P \)

\{ e : v \in S \}

The second constructor has the form expression \( e \) colon variable \( v \) in set \( S \).
Two Set Constructors

\{ v \in S : P \}

the subset of \( S \) consisting of all \( v \) satisfying \( P \)

\{ e : v \in S \}

the set of all \( e \) for \( v \) in \( S \)

The second constructor has the form expression \( e \) colon variable \( v \) in set \( S \).

It’s the set consisting of all values assumed by the expression \( e \) when \( v \) is an element of \( S \).
Two Set Constructors

\{ v \in S : P \}\)

the subset of \( S \) consisting of all \( v \) satisfying \( P \)

\{ e : v \in S \}\)

the set of all \( e \) for \( v \) in \( S \)

\{ n^2 : n \in \text{Nat} \}\)

The second constructor has the form expression \( e \) colon variable \( v \) in set \( S \).

It’s the set consisting of all values assumed by the expression \( e \) when \( v \) is an element of \( S \).

For example, this expression
Two Set Constructors

\{ v \in S : P \}\]

the subset of \( S \) consisting of all \( v \) satisfying \( P \)

\{ e : v \in S \}\]

the set of all \( e \) for \( v \) in \( S \)

\{ n^2 : n \in Nat \} = \{ 0, 1, 4, 9, \ldots \}

the set of all squares of natural numbers

The second constructor has the form expression \( e \) colon variable \( v \) in set \( S \).

It’s the set consisting of all values assumed by the expression \( e \) when \( v \) is an element of \( S \).

For example, this expression equals the set of all squares of natural numbers.

[slide 121]
There’s one more thing I’d like to point out about this expression.

\[
\begin{align*}
\text{LET } mset & \triangleq \{ m \in msgs : \wedge m.type = \text{“phase1b”} \\
& \quad \wedge m.ins = r \\
& \quad \wedge m.mbal = bal \\
& \quad \wedge m.acc \in MS \} \\
\maxbal & \triangleq \text{Maximum}(\{ m.bal : m \in mset \}) \\
\val & \triangleq \text{IF } \maxbal = -1 \\
& \quad \text{THEN } \text{“aborted”} \\
& \quad \text{ELSE } (\text{CHOOSE } m \in mset : m.bal = \maxbal).\val \\
\text{IN} \quad \forall ac \in MS : \exists m \in mset : m.acc = ac \\
& \quad \wedge \text{Send}([\text{type } \mapsto \text{“phase2a”, } \ins \mapsto r, \bal \mapsto \bal, \val \mapsto \val])
\end{align*}
\]
(\text{CHOOSE } m \in \text{mset} : m.\text{bal} = \text{maxbal}).\text{val}

There’s one more thing I’d like to point out about this expression.
Choice of \( m \) need not be unique.

\[
(\text{CHOOSE } m \in \text{mset} : m.\text{bal} = \text{maxbal}).\text{val}
\]

There’s one more thing I’d like to point out about this expression.

This \textit{CHOOSE} expression can allow more than one possible choice for \( m \).
All choices of $m$ have same value of $m.val$.

There’s one more thing I’d like to point out about this expression.

This `CHOOSE` expression can allow more than one possible choice for $m$.

In any reachable state of the algorithm, all possible choices of $m$ have the same value of $m.val$. 

[slide 125]
Paxos Commit is not an easy algorithm to understand, and this is probably its most subtle part.

I don’t know how to write a clearer precise description of this step of the algorithm.

If you understand the algorithm, then when you get used to the math, I think you’ll find this definition as elegant as I do.
\[
\text{Phase} 1b(\text{acc}) \triangleq \\
\exists m \in \text{msgs} : \\
\quad \land m.\text{type} = \text{"phase} 1a\text{"} \\
\quad \land a\text{State}[m.\text{ins}][\text{acc}].\text{mbal} < m.\text{bal} \\
\quad \land a\text{State}' = [a\text{State EXCEPT } ![m.\text{ins}][\text{acc}].\text{mbal} = m.\text{bal}] \\
\quad \land \text{Send}([\text{type} \mapsto \text{"phase} 1b\text{"}, \\
\quad \quad \quad \quad \quad \text{ins} \mapsto m.\text{ins}, \\
\quad \quad \quad \quad \quad \text{mbal} \mapsto m.\text{bal}, \\
\quad \quad \quad \quad \quad \text{bal} \mapsto a\text{State}[m.\text{ins}][\text{acc}].\text{bal}, \\
\quad \quad \quad \quad \quad \text{val} \mapsto a\text{State}[m.\text{ins}][\text{acc}].\text{val}, \\
\quad \quad \quad \quad \quad \text{acc} \mapsto \text{acc}]) \\
\quad \land \text{UNCHANGED } r\text{mState}
\]
\[ aState' = [aState \ \text{EXCEPT} \ ![m.ins][acc].mbal = m.bal] \]

The next new construct is in this definition.

In this subformula.
The next new construct is in this definition.

In this subformula.

\[ aState' = [aState \ \text{EXCEPT} \ ![m.ins][acc].mbal = m.bal] \]
The next new construct is in this definition.

In this subformula, you haven’t seen this form of \texttt{EXCEPT} expression. It’s an abbreviation for

\[aState \ EXCEPT \ ![m.ins][acc].mbal = m.bal\]
\[ aState \text{ EXCEPT } ![m.ins] \]

\[ aState \text{ EXCEPT } ![m.ins] = \]

\textit{aState} EXCEPT its value on \textit{m.ins} equals
\[aState \ \text{EXCEPT} \ ![m.ins][acc]\]
\[aState \ \text{EXCEPT} \ ![m.ins] =
\[aState[m.ins] \ \text{EXCEPT} \ ![acc] =

\text{\textit{aState} EXCEPT its value on \textit{m.ins} equals} \\
\text{\textit{aState} of \textit{m.ins} EXCEPT its value on a-c-c equals}
\[ aState \text{ EXCEPT } !(m.\text{ins})[acc].mbal \]

\[ aState \text{ EXCEPT } !(m.\text{ins}) = \]

\[ aState[m.\text{ins}] \text{ EXCEPT } ![acc] = \]

\[ aState[m.\text{ins}][acc] \text{ EXCEPT } !.mbal = \]

\textit{aState \textbf{EXCEPT} its value on } m.\text{ins} \textbf{equals} \\
\textit{aState \textbf{of} } m.\text{ins} \textbf{EXCEPT} its value on \textbf{a-c-c} \textbf{equals} \\
\textit{aState \textbf{of} } m.\text{ins} \textbf{of} \textbf{a-c-c \textbf{EXCEPT} its} \textbf{m-bal component \textbf{equals}}

[slide 133]
\[
[aState \ EXCEPT \ ![m.ins][acc] . m_{bal} = m_{bal} ] \\
[aState \ EXCEPT \ ![m.ins] = \\
[aState[m.ins] \ EXCEPT \ ![acc] = \\
[aState[m.ins][acc] \ EXCEPT \ ![m_{bal} = m_{bal} ]] 
\]

\textit{aState} \ EXCEPT \ its value on \textit{m.ins} equals \\
\textit{aState} \ of \textit{m.ins} \ EXCEPT \ its value on \textit{a-c-c} equals \\
\textit{aState} \ of \textit{m.ins} \ of \textit{a-c-c} \ EXCEPT \ its \textit{m-bal} component equals \textit{m} \textit{dot} \textit{bal}. \\
\textit{Whew}. \\
[\text{slide 134}]
If you stop and decipher this, you’ll see that
If you stop and decipher this, you’ll see that this formula corresponds to

\[ a_{State}' = [a_{State} \text{ EXCEPT } !(m.\text{ins})[acc].mbal = m.bal] \]
If you stop and decipher this, you’ll see that this formula corresponds to this programming-language statement.

So you just have to remember this idiom and not try to figure out the EXCEPT expression. That’s what I do.
\(\begin{align*}
\text{Phase2b}(\text{acc}) & \triangleq \\
& \land \exists m \in \text{msgs} : \\
& \quad \land m.\text{type} = \text{"phase2a"} \\
& \quad \land \text{aState}[m.\text{ins}][\text{acc}].\text{mbal} \leq m.\text{bal} \\
& \quad \land \text{aState}' = [\text{aState \ EXCEPT } ![m.\text{ins}][\text{acc}].\text{mbal} = m.\text{bal}, \\
& \qquad ![m.\text{ins}][\text{acc}].\text{bal} = m.\text{bal}, \\
& \qquad ![m.\text{ins}][\text{acc}].\text{val} = m.\text{val}] \\
& \quad \land \text{Send}([\text{type} \mapsto \text{"phase2b"}, \text{ins} \mapsto m.\text{ins}, \text{bal} \mapsto m.\text{bal}, \\
& \qquad \text{val} \mapsto m.\text{val}, \text{acc} \mapsto \text{acc}]) \\
& \quad \land \text{UNCHANGED} \ \text{rmState}
\end{align*}\)
This definition contains another generalization of the `EXCEPT` construct.

\[ aState' = [aState \text{ EXCEPT } ![m.ins][acc].mbal = m.bl, \\
            ![m.ins][acc].bal = m.bl, \\
            ![m.ins][acc].val = m.val] \]
This definition contains another generalization of the \texttt{EXCEPT} construct.

\[ aState' = [aState \ \texttt{EXCEPT} \ [m.ins][acc].mbal = m.bal, \\
[m.ins][acc].bal = m.bal, \\
[m.ins][acc].val = m.val] \]
This definition contains another generalization of the EXCEPT construct.

If you want, you can try to figure out what this EXCEPT expression means when I tell you that

\[ aState \text{ EXCEPT } !\text{[m.ins][acc].mbal} = m.bal, \]
\[ !\text{[m.ins][acc].bal} = m.bal, \]
\[ !\text{[m.ins][acc].val} = m.val \]
This definition contains another generalization of the `EXCEPT` construct.

If you want, you can try to figure out what this `EXCEPT` expression means when I tell you that this subformula describes the same change to `aState` as

```
\[ aState' = [aState \ EXCEPT \ !\text{[m.ins][acc].mbal} = m.bal, \\
              !\text{[m.ins][acc].bal} = m.bal, \\
              !\text{[m.ins][acc].val} = m.val] \]
```
\[ aState' = [aState \text{ EXCEPT} ![m.ins][acc].mbal = m.bal, \\
! [m.ins][acc].bal = m.bal, \\
! [m.ins][acc].val = m.val] \]

\[
\begin{align*}
\text{aState}[m.ins][acc].mbal &= m.bal; \\
\text{aState}[m.ins][acc].bal &= m.bal; \\
\text{aState}[m.ins][acc].val &= m.val;
\end{align*}
\]

executing this sequence of three program statements.

Notice the correspondence between the parts of the \text{EXCEPT} expression
\[ aState' = [aState \ \text{EXCEPT} \begin{align*}
&![m.\text{ins}][acc].mbal = m.\text{bal,} \\
&![m.\text{ins}][acc].\text{bal} = m.\text{bal,} \\
&![m.\text{ins}][acc].\text{val} = m.\text{val}] \\
\end{align*} \]

\begin{align*}
\text{aState}[m.\text{ins}][acc].mbal & = m.\text{bal;} \\
\text{aState}[m.\text{ins}][acc].\text{bal} & = m.\text{bal;} \\
\text{aState}[m.\text{ins}][acc].\text{val} & = m.\text{val;} \\
\end{align*}

executing this sequence of three program statements.

Notice the correspondence between the parts of the EXCEPT expression and the program statements.
\[ aState' = [aState \ \text{EXCEPT} \begin{align*}
&[m.\text{ins}][acc].mbal = m.\text{bal}, \\
&[m.\text{ins}][acc].bal = m.\text{bal}, \\
&[m.\text{ins}][acc].val = m.\text{val} \] 
\]

\[
\begin{align*}
aState[m.\text{ins}][acc].mbal & = m.\text{bal}; \\
aState[m.\text{ins}][acc].bal & = m.\text{bal}; \\
aState[m.\text{ins}][acc].val & = m.\text{val}; 
\end{align*}
\]

executing this sequence of three program statements.

Notice the correspondence between the parts of the \text{EXCEPT} expression and the program statements.
executing this sequence of three program statements.

Notice the correspondence between the parts of the EXCEPT expression and the program statements.
executing this sequence of three program statements.

Notice the correspondence between the parts of the `EXCEPT` expression and the program statements.
CHECKING THE SPEC

Checking the Specification
Open *PaxosCommit* in the Toolbox

Open module *PaxosCommit* in the Toolbox

[slide 149]
Open *PaxosCommit* in the Toolbox and create a new model.
Open \textit{PaxosCommit} in the Toolbox and create a new model.

You have to enter the initial and next-state formulas.
Open *PaxosCommit* in the Toolbox and create a new model.

You have to enter

and

Open module *PaxosCommit* in the Toolbox and create a new model.

You have to enter the initial and next-state formulas and the values of the constants.

[slide 152]
Open *PaxosCommit* in the Toolbox and create a new model.

You have to enter

The initial and next-state formulas are named

[slide 153]
Open *PaxosCommit* in the Toolbox and create a new model.

You have to enter

![Initial predicate and next-state relation](image1)

and

![What is the model?](image2)

The initial and next-state formulas are named *PCInit* and *PCNext*.
Open $PaxosCommit$ in the Toolbox and create a new model.

You have to enter

The initial and next-state formulas are named $PCInit$ and $PCNext$.

Now for the values assigned to the constants.
We normally start with a tiny model

We normally start with a tiny model
We normally start with a tiny model, but we’ll skip that.
We normally start with a tiny model, but we’ll skip that.

Instead, we’ll use the smallest model that could reveal an error in the algorithm.

We normally start with a tiny model but we’ll skip that.

Instead, we’ll use a model which, if you understand the algorithm, you’ll see is the smallest one that could reveal a non-trivial error.
Ballot $\leftarrow$  
Acceptor $\leftarrow$ \{a1, a2, a3\} a set of model values  
Majority $\leftarrow$  
RM $\leftarrow$

We normally start with a tiny model but we’ll skip that. Instead, we’ll use a model which, if you understand the algorithm, you’ll see is the smallest one that could reveal a non-trivial error.

We assign a set of three model values to Acceptor,
We normally start with a tiny model but we’ll skip that.

Instead, we’ll use a model which, if you understand the algorithm, you’ll see is the smallest one that could reveal a non-trivial error.

We assign a set of three model values to $\text{Accept}or$, and a set of two model values to $RM$. 

[slide 160]
Ballot $\leftarrow \{0, 1\}$
Acceptor $\leftarrow \{a_1, a_2, a_3\}$
Majority $\leftarrow$
RM $\leftarrow \{r_1, r_2\}$

We assign this set of two numbers to *Ballot*,

[slide 161]
We assign this set of two numbers to \textit{Ballot}, and this set of sets of acceptors to \textit{Majority}.
We assign this set of two numbers to $Ballot$, and this set of sets of acceptors to $Majority$.

This is an ordinary assignment,
Ballot ← \{0, 1\}
Acceptor ← \{a_1, a_2, a_3\}
Majority ← \{\{a_1, a_2\}, \{a_1, a_3\}, \{a_2, a_3\}\}
RM ← \{r_1, r_2\}

We assign this set of two numbers to *Ballot*, and this set of sets of acceptors to *Majority*.

This is an ordinary assignment, because the model values $a_1$, $a_2$, and $a_3$ are declared in the assignment of a set of model values to *Acceptor*.

[slide 164]
Ballot $\leftarrow \{0, 1\}$
Acceptor $\leftarrow \{a_1, a_2, a_3\}$
Majority $\leftarrow \{\{a_1, a_2\}, \{a_1, a_3\}, \{a_2, a_3\}\}$
RM $\leftarrow \{r_1, r_2\}$

We assign this set of two numbers to Ballot, and this set of sets of acceptors to Majority.

This is an ordinary assignment, because the model values $a_1$, $a_2$, and $a_3$ are declared in the assignment of a set of model values to Acceptor.
Ballot $\leftarrow \{0, 1\}$
Acceptor $\leftarrow \{a_1, a_2, a_3\}$
Majority $\leftarrow \{\{a_1, a_2\}, \{a_1, a_3\}, \{a_2, a_3\}\}$
RM $\leftarrow \{r_1, r_2\}$

This can be a symmetry set, because $r_1$ and $r_2$ aren’t used elsewhere.

The set we assigned to $RM$ can be a symmetry set because its elements aren’t used elsewhere.
Ballot $\leftarrow \{0, 1\}$
Acceptor $\leftarrow \{a_1, a_2, a_3\}$
Majority $\leftarrow \{\{a_1, a_2\}, \{a_1, a_3\}, \{a_2, a_3\}\}$
RM $\leftarrow \{r_1, r_2\}$

But what about this set?

The set we assigned to $RM$ can be a symmetry set because its elements aren't used elsewhere.

But what about the set we assigned to $Acceptor$?
Ballot ← \{0, 1\}
Acceptor ← \{a_1, a_2, a_3\}
Majority ← \{\{a_1, a_2\}, \{a_1, a_3\}, \{a_2, a_3\}\}
RM ← \{r_1, r_2\}

But what about this set?

\(a_1, a_2, a_3\) are used here.

The set we assigned to \(RM\) can be a symmetry set because its elements aren't used elsewhere.

But what about the set we assigned to \(Acceptor\)?

Its elements are used in the value assigned to \(Majority\).
Ballot $\leftarrow \{0, 1\}$
Acceptor $\leftarrow \{a_1, a_2, a_3\}$
Majority $\leftarrow \{\{a_1, a_2\}, \{a_1, a_3\}, \{a_2, a_3\}\}$
RM $\leftarrow \{r_1, r_2\}$

This use is OK because the expression is symmetric in $a_1, a_2, a_3$.

The set we assigned to $RM$ can be a symmetry set because its elements aren’t used elsewhere.

But what about the set we assigned to $Acceptor$?

Its elements are used in the value assigned to $Majority$.

But this use is OK because the expression they appear in is symmetric in the elements of the set we assigned to $Acceptor$.
Ballot $\leftarrow \{0, 1\}$
Acceptor $\leftarrow \{a_1, a_2, a_3\}$
Majority $\leftarrow \{\{a_1, a_2\}, \{a_1, a_3\}, \{a_2, a_3\}\}$
RM $\leftarrow \{r_1, r_2\}$

This use is OK because the expression is symmetric in $a_1$, $a_2$, $a_3$.

Interchanging any two of these elements leaves the expression unchanged.

Remember, this means that interchanging any two elements of that set leaves the expression unchanged.
Remember, this means that interchanging any two elements of that set leaves the expression unchanged.

For example, if we interchange $a_1$ and $a_3$ in the expression,
Ballot $\leftarrow \{0, 1\}$
Acceptor $\leftarrow \{a_1, a_2, a_3\}$
Majority $\leftarrow \{\{a_1, a_2\}, \{a_1, a_3\}, \{a_2, a_3\}\}$
RM $\leftarrow \{r_1, r_2\}$

For example, interchanging $a_1 \leftrightarrow a_3$ in

\[
\{\{a_1, a_2\}, \{a_1, a_3\}, \{a_2, a_3\}\}
\]

produces

\[
\{\{a_3, a_2\}, \{a_3, a_1\}, \{a_2, a_1\}\}
\]

Remember, this means that interchanging any two elements of that set leaves the expression unchanged.

For example, if we interchange $a_1$ and $a_3$ in the expression, we get this expression.
Ballot $\leftarrow \{0, 1\}$
Acceptor $\leftarrow \{a_1, a_2, a_3\}$
Majority $\leftarrow \{\{a_1, a_2\}, \{a_1, a_3\}, \{a_2, a_3\}\}$
RM $\leftarrow \{r_1, r_2\}$

These two sets are equal:

$$\{\{a_1, a_2\}, \{a_1, a_3\}, \{a_2, a_3\}\}$$

$$\{\{a_3, a_2\}, \{a_3, a_1\}, \{a_2, a_1\}\}$$

Remember, this means that interchanging any two elements of that set leaves the expression unchanged.

For example, if we interchange $a_1$ and $a_3$ in the expression, we get this expression.

And these two expressions are equal because they describe sets with the same three elements.
Ballot $\leftarrow \{0, 1\}$
Acceptor $\leftarrow \{a_1, a_2, a_3\}$
Majority $\leftarrow \{\{a_1, a_2\}, \{a_1, a_3\}, \{a_2, a_3\}\}$
RM $\leftarrow \{r_1, r_2\}$

These two sets are equal:

\[
\begin{align*}
\{\{a_1, a_2\}, \{a_1, a_3\}, \{a_2, a_3\}\} \\
\{\{a_3, a_2\}, \{a_3, a_1\}, \{a_2, a_1\}\}
\end{align*}
\]

Remember, this means that interchanging any two elements of that set leaves the expression unchanged.

For example, if we interchange $a_1$ and $a_3$ in the expression, we get this expression.

And these two expressions are equal because they describe sets with the same three elements one

[slide 174]
These two sets are equal:

\[
\{\{a_1, a_2\}, \{a_1, a_3\}, \{a_2, a_3\}\}
\]

\[
\{\{a_3, a_2\}, \{a_3, a_1\}, \{a_2, a_1\}\}
\]

Remember, this means that interchanging any two elements of that set leaves the expression unchanged.

For example, if we interchange \(a_1\) and \(a_3\) in the expression, we get this expression.

And these two expressions are equal because they describe sets with the same three elements: one two
Remember, this means that interchanging any two elements of that set leaves the expression unchanged.

For example, if we interchange $a_1$ and $a_3$ in the expression, we get this expression.

And these two expressions are equal because they describe sets with the same three elements one two three
Ballot $\leftarrow \{0, 1\}$
Acceptor $\leftarrow \{a1, a2, a3\}$
Majority $\leftarrow \{\{a1, a2\}, \{a1, a3\}, \{a2, a3\}\}$
RM $\leftarrow \{r1, r2\}$

It’s OK to use elements of a symmetry set

In general, it’s OK to use elements of a symmetry set
Ballot $\leftarrow \{0, 1\}$
Acceptor $\leftarrow \{a_1, a_2, a_3\}$
Majority $\leftarrow \{\{a_1, a_2\}, \{a_1, a_3\}, \{a_2, a_3\}\}$
RM $\leftarrow \{r_1, r_2\}$

It’s OK to use elements of a symmetry set in an expression assigned to another constant

In general, it’s OK to use elements of a symmetry set in an expression assigned to another constant
Ballot $\leftarrow \{0, 1\}$
Acceptor $\leftarrow \{a_1, a_2, a_3\}$
Majority $\leftarrow \{\{a_1, a_2\}, \{a_1, a_3\}, \{a_2, a_3\}\}$
RM $\leftarrow \{r_1, r_2\}$

It's OK to use elements of a symmetry set in an expression assigned to another constant if the expression is symmetric.

In general, it's OK to use elements of a symmetry set in an expression assigned to another constant if the expression is symmetric.
Ballot $\leftarrow \{0, 1\}$
Acceptor $\leftarrow \{a1, a2, a3\}$
Majority $\leftarrow \{\{a1, a2\}, \{a1, a3\}, \{a2, a3\}\}$
RM $\leftarrow \{r1, r2\}$

It’s OK to use elements of a symmetry set in an expression assigned to another constant if the expression is symmetric in the elements of the symmetry set.

In general, it’s OK to use elements of a symmetry set in an expression assigned to another constant if the expression is symmetric in the elements of the symmetry set.
Ballot $\leftarrow \{0, 1\}$
Acceptor $\leftarrow \{a_1, a_2, a_3\}$
Majority $\leftarrow \{\{a_1, a_2\}, \{a_1, a_3\}, \{a_2, a_3\}\}$
RM $\leftarrow \{r_1, r_2\}$

There’s one additional condition for symmetry sets.

There’s just one additional condition a symmetry set must satisfy that I can now explain.
There’s one additional condition for symmetry sets.

Elements of a symmetry set

There’s just one additional condition a symmetry set must satisfy that I can now explain.

Elements of a symmetry set,
There’s one additional condition for symmetry sets.

Elements of a symmetry set, or a constant assigned elements of a symmetry set.

There’s just one additional condition a symmetry set must satisfy that I can now explain.

Elements of a symmetry set, or a constant that’s assigned elements of a symmetry set.
There's one additional condition for symmetry sets.

Elements of a symmetry set, or a constant assigned elements of a symmetry set may not appear in a CHOOSE expression.

There's just one additional condition a symmetry set must satisfy that I can now explain.

Elements of a symmetry set, or a constant that's assigned elements of a symmetry set may not appear in a CHOOSE expression.
Ballot $\leftarrow \{0, 1\}$
Acceptor $\leftarrow \{a_1, a_2, a_3\}$
Majority $\leftarrow \\{\{a_1, a_2\}, \{a_1, a_3\}, \{a_2, a_3\}\}$
RM $\leftarrow \{r_1, r_2\}$

There’s one additional condition for symmetry sets.

Elements of a symmetry set, or a constant assigned elements of a symmetry set may not appear in a \texttt{CHOOSE} expression.

In the \textit{PaxosCommit} spec, elements of a symmetry set don’t appear in a \texttt{CHOOSE} because
There’s one additional condition for symmetry sets.

Elements of a symmetry set, or a constant assigned elements of a symmetry set may not appear in a `CHOOSE` expression.

In the `PaxosCommit` spec, elements of a symmetry set don’t appear in a `CHOOSE` because they can appear only in these assignments and there’s no `CHOOSE` there.
There's one additional condition for symmetry sets.

Elements of a symmetry set, or a constant assigned elements of a symmetry set may not appear in a CHOOSE expression.

In the PaxosCommit spec, elements of a symmetry set don't appear in a CHOOSE because they can appear only in these assignments and there's no CHOOSE there.

To verify that a constant which is assigned elements of a symmetry set doesn't appear in a CHOOSE expression,
We must check that these constants don’t appear in a `CHOOSE` expression of the spec.

In the *PaxosCommit* spec, elements of a symmetry set don’t appear in a `CHOOSE` because they can appear only in these assignments and there’s no `CHOOSE` there.

To verify that a constant which is assigned elements of a symmetry set doesn’t appear in a `CHOOSE` expression, we must check that these constants don’t appear in any `CHOOSE` expression in the spec.
Ballot $\leftarrow \{0, 1\}$

**Acceptor** $\leftarrow \{a_1, a_2, a_3\}$

**Majority** $\leftarrow \\{\{a_1, a_2\}, \{a_1, a_3\}, \{a_2, a_3\}\}$

**RM** $\leftarrow \{r_1, r_2\}$

We must check that these constants don’t appear in a `CHOOSE` expression of the spec.

They don’t.

You can check that they don’t.
Assign these values in the model, with *Acceptor* and *RM* being symmetry sets.

You can check that they don’t.

Assign these values in the model, letting *Acceptor* and *RM* be symmetry sets.
What to Check

We should check that the algorithm is correct.

We’ll see in a later video, how to check that it implements transaction commit.
What to Check

There are two invariants we can check:

- The type correctness invariant \( \text{PCT~typeOK} \)
- Invariant \( \text{TCConsistent} \) imported from module \( \text{TCommit} \)

Add them and run TLC on the model.

We should check that the algorithm is correct.

We’ll see in a later video, how to check that it implements transaction commit.

For now, there are two invariants we can check:
What to Check

There are two invariants we can check:

– The type correctness invariant $PCTypeOK$

The type correctness invariant $PCTypeOK$ that we looked at earlier
What to Check

There are two invariants we can check:

– The type correctness invariant \( PCT_{TypeOK} \)

– Invariant \( TCC_{Consistent} \) imported from module \( TCommit \)

The type correctness invariant \( PCT_{TypeOK} \) that we looked at earlier

and the invariant \( TCC_{Consistent} \), which is imported with an INSTANCE statement from module \( TCommit \).
What to Check

There are two invariants we can check:

– The type correctness invariant \( PCTypeOK \)
– Invariant \( TCConsistent \) imported from module \( TCommit \)

Add them and run TLC on the model.

The type correctness invariant \( PCTypeOK \) that we looked at earlier and the invariant \( TCConsistent \), which is imported with an INSTANCE statement from module \( TCommit \).

Add these invariants to the What to check part of the model and run TLC on the model.

[slide 195]
TLC takes 30 seconds to run the model on my laptop using two cores.

TLC takes about 30 seconds to run the model on my laptop using two cores.
TLC takes 30 seconds to run the model on my laptop using two cores. It reports no error and finds 120 thousand distinct states.

If we change the model to assign Ballot the set \{0, 1, 2\} instead of \{0, 1\}, TLC runs for 1 hour on a 128 core machine and finds 220 million states.
TLC takes 30 seconds to run the model on my laptop using two cores. It reports no error and finds 120 thousand distinct states.

If we change the model to assign \textit{Ballot} the set \{0, 1, 2\} instead of \{0, 1\}, TLC runs for 1\frac{1}{2} hours on a 128 core machine and finds 220 million states.

Execution time and space grow exponentially with the size of the model.

TLC takes about 30 seconds to run the model on my laptop using two cores. It reports no error and finds about 120 thousand distinct states.

If we change the model to assign \textit{Ballot} a set of three numbers instead of two,
TLC takes 30 seconds to run the model on my laptop using two cores. It reports no error and finds 120 thousand distinct states.

If we change the model to assign \textit{Ballot} the set \{0, 1, 2\} instead of \{0, 1\}, TLC runs for 1\frac{1}{2} hours on a 128 core machine and finds 220 million states.

TLC runs for about one and a half hours on a 128 core machine and finds about 220 million states.

We use very small models because
TLC takes 30 seconds to run the model on my laptop using two cores. It reports no error and finds 120 thousand distinct states.

If we change the model to assign $Ballot$ the set $\{0, 1, 2\}$ instead of $\{0, 1\}$, TLC runs for $1\frac{1}{2}$ hours on a 128 core machine and finds 220 million states.

Execution time and space grow exponentially with the size of the model.

TLC runs for about one and a half hours on a 128 core machine and finds about 220 million states.

We use very small models because execution time and space grow exponentially with the size of the model.
What good is checking such small models?

To answer that question, make this change to value the model assigns to Majority.

Delete this element of an element of the set.

The expression is no longer symmetric in $a_1, a_2,$ and $a_3$. 

[slide 201]
What good is checking such small models?

Make this change to the model.

\[ \text{Majority} \leftarrow \{\{a_1, a_2\}, \{a_1, a_3\}, \{a_2, a_3\}\} \]

What good is checking such small models?

To answer that question, make this change to value the model assigns to \textit{Majority}.
What good is checking such small models?

Make this change to the model.

Majority ← \{\{a_1, a_2\}, \{a_1, a_3\}, \{a_2, a_3\}\}

To answer that question, make this change to value the model assigns to Majority.

Delete this element of an element of the set.

[slide 203]
What good is checking such small models?

Make this change to the model.

\[ \text{Majority} \leftarrow \{ \{a_1, a_2\}, \{a_1, a_3\}, \{ a_3\} \} \]

What good is checking such small models?

To answer that question, make this change to value the model assigns to \textit{Majority}.

Delete this element of an element of the set.

The expression is no longer symmetric in \( a_1, a_2, \) and \( a_3 \).

[slide 204]
What good is checking such small models?

Make this change to the model.

\[
\text{Majority} \leftarrow \{\{a_1, a_2\}, \{a_1, a_3\}, \{a_3\}\}
\]

So we have to change the assignment to \textit{Acceptor}
What good is checking such small models?

Make this change to the model.

$$\text{Majority} \leftarrow \{\{a_1, a_2\}, \{a_1, a_3\}, \{a_3\}\}$$

So we have to change the assignment to $\text{Acceptor}$.

So it’s no longer a symmetry set.
What good is checking such small models?

Make this change to the model.

$$\text{Majority} \leftarrow \{\{a_1, a_2\}, \{a_1, a_3\}, \{a_3\}\}$$

So we have to change the assignment to $\text{Accept}$. So it’s no longer a symmetry set.
What good is checking such small models?

Make this change to the model.

$\text{Majority} \leftarrow \{\{a_1, a_2\}, \{a_1, a_3\}, \{a_3\}\}$

If you run TLC on the model, it will complain that the assumption is violated.

So we have to change the assignment to $\text{Accept}$.

So it’s no longer a symmetry set.

Now if you run TLC on the model, it will complain that this assumption is violated.
What good is checking such small models?

Make this change to the model.

```
Majority ← {{a1, a2}, {a1, a3}, { a3}}
```

```
ASSUME
  /
  \ Ballot \subseteqq Nat
  /
  \ 0 \in Ballot
  /
  \ Majority \subseteqq SUBSET Acceptor
  /
  \ A MS1, MS2 \in Majority : MS1 \cap MS2 # {}
```

So we have to change the assignment to *Acceptor*

So it’s no longer a symmetry set.

Now if you run TLC on the model, it will complain that this assumption is violated

**Because**

[slide 209]
What good is checking such small models?

Make this change to the model.

```
Majority ← \{a1, a2\}, \{a1, a3\}, \{a3\}
```

```
ASSUME
/
\Balot \subsetneqq \Nat
\0 \in \Balot
\Majority \subsetneqq \SUBSET \Accept
\A \MS1, \MS2 \in \Majority : \MS1 \cap \MS2 ≠ \emptyset
```

So we have to change the assignment to *Acceptor*

So it’s no longer a symmetry set.

Now if you run TLC on the model, it will complain that this assumption is violated.

Because **this assertion is no longer true. So, we have to comment it out.**
What good is checking such small models?

Make this change to the model.

\[
\text{Majority} \leftarrow \{\{a_1, a_2\}, \{a_1, a_3\}, \{a_3\}\}
\]

\begin{verbatim}
ASSUME
    \(\forall\ \text{Ballot} \subseteqq \text{Nat}\)
    \(\forall\ 0 \in \text{Ballot}\)
    \(\forall\ \text{Majority} \subseteqq \text{SUBSET} \text{ Acceptor}\)
    \(\forall\ \forall\ A \text{ MS}_1, \text{ MS}_2 \in \text{Majority} : \text{ MS}_1 \cap \text{ MS}_2 \neq \emptyset\)
\end{verbatim}

So we have to change the assignment to \textit{Acceptor}.

So it’s no longer a symmetry set.

Now if you run TLC on the model, it will complain that this assumption is violated. Because this assertion is no longer true. So, we have to comment it out.
What good is checking such small models?

Make this change to the model.

\[ \text{Majority} \leftarrow \{\{a_1, a_2\}, \{a_1, a_3\}, \{a_3\}\} \]

So we have to change the assignment to \textit{Acceptor}

So it’s no longer a symmetry set.

Now if you run TLC on the model, it will complain that this assumption is violated. Because this assertion is no longer true. So, we have to comment it out.
What good is checking such small models?

Make this change to the model.

\[
\text{Majority} \leftarrow \{\{a1, a2\}, \{a1, a3\}, \{a3\}\}
\]

ASSUME

slash Ballot subseteq Nat
slash 0 in Ballot
slash Majority subseteq SUBSET Acceptor
slash* slash A MS1, MS2 in Majority : MS1 cap MS2 # {}

So we have to change the assignment to *Accept*or*

So it’s no longer a symmetry set.

Now if you run TLC on the model, it will complain that this assumption is violated
Because this assertion is no longer true. So, we have to comment it out.

[slide 213]
Run TLC on the model.

Because the assumption is not satisfied, the algorithm is incorrect for this value of Majority.

TLC reports that invariant TCConsistent is violated, and it produces a 14-state error trace.

Even a very small model can catch an error in an algorithm.

Run TLC on the model.

Because the assumption is not satisfied, the algorithm is incorrect for this changed value of Majority.

TLC reports that invariant TCConsistent is violated and it produces a minimal-length 14-state error trace.

The Paxos commit algorithm is correct.
Run TLC on the model.

Because the assumption is not satisfied, the algorithm is incorrect for this value of \textit{Majority}.

Run TLC on the model.

Because the assumption is not satisfied, the algorithm is incorrect for this changed value of \textit{Majority}.
Run TLC on the model.

Because the assumption is not satisfied, the algorithm is incorrect for this value of $Majority$.

TLC reports that invariant $TCC_{Consistent}$ is violated.
Run TLC on the model.

Because the assumption is not satisfied, the algorithm is incorrect for this value of *Majority*.

TLC reports that invariant *TCConsistent* is violated, and it produces a 14-state error trace.

Run TLC on the model.

Because the assumption is not satisfied, the algorithm is incorrect for this changed value of *Majority*.

TLC reports that invariant *TCConsistent* is violated and it produces a minimal-length 14-state error trace.

The Paxos commit algorithm is correct.

[slide 217]
Run TLC on the model.

Because the assumption is not satisfied, the algorithm is incorrect for this value of $Majority$.

TLC reports that invariant $TCConsistent$ is violated, and it produces a 14-state error trace.

Even a very small model can catch an error in an algorithm.

But this example shows that even a very small model can catch an error in a real algorithm.
You’ve now learned enough of the TLA+ language to start writing your own specs. However, before you do that, you should know more about what TLA+ specs mean. In particular, you should understand what it means for the Paxos Commit algorithm to implement the transaction-commit spec. That’s the topic of the next lecture.
End of Lecture 7

PAXOS COMMIT