TLA$^+$ Video Course – Lecture 7

Leslie Lamport

PAXOS COMMIT

This video should be viewed in conjunction with a Web page.
To find that page, search the Web for *TLA+ Video Course*.

The TLA$^+$ Video Course
Lecture 7
Paxos Commit

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.

[slide 2]

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.

[slide 7]

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 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.

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

The backup transaction manager 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.

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

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 it's hard to find that they're wrong 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.

Finding fault-tolerant distributed algorithms is hard.

They’re easy to get wrong , and it’s hard to find that they’re wrong by testing.

It’s important to 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.

Finding fault-tolerant distributed algorithms is hard.

They're easy to get wrong , and it's hard to find that they're wrong by testing.

It's important to get the algorithm right before we code it.

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 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

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

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:

We’re looking at this module for two reasons:

We're looking at this module for two reasons:

– To see what a real spec looks like.

We're looking at this module for two reasons:

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

We're looking at this module for two reasons:

– To see what a real spec looks like.

– To learn some more TLA$^+$.

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$^+$.

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.

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 module $TCommit$; and download the paper if you want to read it.

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.

THE SPECIFICATION

The Paxos Commit algorithm's specification

EXTENDS $Integers$

The module begins with an EXTENDS statement that imports the definition of arithmetic operators from the standard $Integers$ module.

$$Maximum(S) \triangleq$$

The module then defines $Maximum(S)$

$Maximum(S) \triangleq$

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,

$Maximum(S) \triangleq$

The largest element of the finite set S of integers, or -1 if S is the empty set.

The module then defines $Maximum(S)$ 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.

$$
\begin{array}{l}
Maximum(S) \;\triangleq \\
\quad \text{IF } S = \{\} \text{ THEN } -1 \\
\qquad\qquad\quad\;\; \text{ELSE } \text{smallest number in } S
\end{array}
$$

The definition has this form

$$
\begin{array}{l}
Maximum(S) \;\triangleq \\
\quad \text{IF } S = \{\} \text{ THEN } -1 \\
\qquad\qquad\quad \text{ELSE } \text{smallest number in } S
\end{array}
$$

The definition has this form

$$
\begin{array}{l}
Maximum(S) \ \triangleq \\
\quad \text{IF } S = \{\} \text{ THEN } -1 \\
\quad \phantom{\text{IF } S = \{\}} \text{ ELSE } \text{smallest number in } S
\end{array}
$$

The definition has this form

The smallest number in S is written this way

$$
\begin{array}{l}
Maximum(S) \triangleq \\
\quad \text{IF } S = \{\} \text{ THEN } -1 \\
\qquad\qquad\qquad \text{ELSE CHOOSE } n \in S : \; n \geq \text{every element in } S
\end{array}
$$

where the CHOOSE expression

$Maximum(S) \triangleq$

$\quad \text{IF } S = \{\} \text{ THEN } -1$

$\quad\quad\quad\quad \text{ELSE } \boxed{\text{CHOOSE } n \in S :}$ $n \geq$ every element in S

Equals an arbitrarily chosen n in S

where the CHOOSE expression equals an arbitrarily chosen value n in S

$Maximum(S) \triangleq$
$\quad$ IF $S = \{\}$ THEN -1
$\qquad\qquad$ ELSE CHOOSE $n \in S$: $\boxed{n \geq \text{every element in } S}$

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.

$$
\begin{array}{l}
Maximum(S) \triangleq \\
\quad \text{IF } S = \{\} \text{ THEN } -1 \\
\quad \qquad\qquad\qquad \text{ELSE CHOOSE } n \in S : \; \forall\, m \in S : n \geq m
\end{array}
$$

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.

[slide 45]

$$
\begin{array}{l}
Maximum(S) \;\triangleq \\
\quad \text{IF } S = \{\} \text{ THEN } -1 \\
\quad \phantom{\text{IF } S = \{\} } \text{ ELSE } \text{ CHOOSE } n \in S : (\forall\, m \in S : n \geq m)
\end{array}
$$

It’s a little easier to read with parentheses.

$$
\begin{array}{l}
Maximum(S) \ \triangleq \\
\quad \text{IF } S = \{\} \text{ THEN } -1 \\
\qquad\qquad\qquad \text{ELSE CHOOSE } n \in S : \boxed{(\forall\, m \in S : n \geq m)}
\end{array}
$$

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

CHOOSE $v \in S : P$ equals

if there is a v in S for which P is true

In general, the expression CHOOSE variable v in expression S colon formula P equals

If there is at least one value v in the set S for which formula P is true

CHOOSE $v \in S : P$ equals

if there is a v in S for which P is true

then some such v

In general, the expression CHOOSE variable v in expression S colon formula P equals

If there is at least one value v in the set S for which formula P is true

then the expression equals some such v.

If there's more than one, then the semantics of TLA+ don't specify which one.

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.

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.

CHOOSE $i \in 1\,..\,99$: TRUE

Is an unspecified integer between 1 and 99.

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.

For example: this expression equals an unspecified integer between 1 and 99. We don't know which one.

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.

CHOOSE $i \in 1\,..\,99$: 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.

$$(\textsf{CHOOSE } i \in 1\,..\,99 : \textsf{TRUE}) \;=\; (\textsf{CHOOSE } i \in 1\,..\,99 : \textsf{TRUE})$$

In math, any expression always equals itself.

So this CHOOSE expression always equals itself

In math, any expression equals itself.

$$(\text{CHOOSE } i \in 1\,..\,99 : \text{TRUE}) \;=\; (\text{CHOOSE } i \in 1\,..\,99 : \text{TRUE})$$

There is no nondeterminism in a mathematical expression.

In math, any expression always equals itself.

So this CHOOSE expression always equals itself

There is no nondeterminism in any mathematical expression, including a CHOOSE expression.

In math, any expression equals itself.

$$(\text{CHOOSE } i \in 1\,..\,99 : \text{TRUE}) \;=\; (\text{CHOOSE } i \in 1\,..\,99 : \text{TRUE})$$

There is no nondeterminism in a mathematical expression.

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

If this CHOOSE expression equals 37 today, it will still equal 37 next week.

In math, any expression equals itself.

$$(\text{CHOOSE } i \in 1\,..\,99 : \text{TRUE}) \;=\; (\text{CHOOSE } i \in 1\,..\,99 : \text{TRUE})$$

There is no nondeterminism in a mathematical expression.

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

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

If this 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.

$x' \in 1\,..\,99$

Allows the value of x in the next state to be any number in $1\,..\,99$.

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.

$x' \in 1 \,..\, 99$

Allows the value of x in the next state to be any number in $1 \,..\, 99$.

$x' = \text{CHOOSE } i \in 1 \,..\, 99 : \text{TRUE}$

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.

$x' \in 1\,..\,99$

Allows the value of x in the next state to be any number in $1\,..\,99$.

$\cancel{x' = \text{CHOOSE } i \in 1\,..\,99 : \text{TRUE}}$

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 CHOOSE $v \in S : P$

Only when there's exactly one v in S satisfying P.

You should write this CHOOSE expression only when there's exactly one value v in S satisfying formula P.

You should write CHOOSE $v \in S : P$

Only when there's exactly one v in S satisfying P.

As in the definition of $Maximum(S)$.

You should write this 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 $Maximum$ of S.

You should write 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 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 $Maximum$ 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.

CONSTANTS RM, $Acceptor$, $Majority$, $Ballot$

After defining $Maximum$, the module contains a CONSTANTS statement declaring these four constants

CONSTANTS $\boxed{RM}$, $Acceptor$, $Majority$, $Ballot$

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

CONSTANTS RM, $\boxed{Acceptor}$, $Majority$, $Ballot$

the set of acceptor processes

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.

CONSTANTS RM, $Acceptor$, $\boxed{Majority, Ballot}$

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.

CONSTANTS RM, $Acceptor$, $Majority$, $Ballot$

This ASSUME statement asserts assumptions being made about the constants.

$\textsc{constants}\ RM,\ Acceptor,\ Majority,\ Ballot$

$\textsc{assume}$

$$\begin{aligned}
&\land\ Ballot \subseteq Nat \\
&\land\ 0 \in Ballot \\
&\land\ Majority \subseteq \textsc{subset}\ Acceptor \\
&\land\ \forall\, MS1, MS2 \in Majority : MS1 \cap MS2 \neq \{\}
\end{aligned}$$

This ASSUME statement asserts assumptions being made about the constants.

CONSTANTS RM, $Acceptor$, $Majority$, $Ballot$

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

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 $Ballot$.

CONSTANTS RM, $Acceptor$, $Majority$, $Ballot$

ASSUME

$$\begin{aligned}
&\land\ Ballot \subseteq Nat \\
&\land\ 0 \in Ballot \\
&\land\ Majority \subseteq \text{SUBSET}\ Acceptor \\
&\land\ \forall\, MS1, MS2 \in Majority : MS1 \cap MS2 \neq \{\}
\end{aligned}$$

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

CONSTANTS RM, $Acceptor$, $Majority$, $Ballot$

ASSUME

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

the set of natural numbers

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).

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.

CONSTANTS RM, $Acceptor$, $Majority$, $Ballot$

ASSUME

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

`\subseteq`

The subset symbol is typed backslash subset e-q.

CONSTANTS RM, $Acceptor$, $Majority$, $Ballot$

ASSUME

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

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, $Acceptor$, $Majority$, $Ballot$

ASSUME

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

the set of all subsets of $Acceptor$

Also called the *powerset of* $Acceptor$, written $\mathcal{P}(Acceptor)$

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 $Acceptor$.

CONSTANTS RM, $Acceptor$, $Majority$, $Ballot$

ASSUME

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

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 $Acceptor$.

CONSTANTS RM, $Acceptor$, $Majority$, $Ballot$

ASSUME

$$\land \forall MS1, MS2 \in Majority : \underbrace{\boxed{MS1 \cap MS2}}_{\text{intersection of } MS1 \text{ and } MS2} \neq \{\}$$

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

CONSTANTS RM, $Acceptor$, $Majority$, $Ballot$

ASSUME

$$\land \forall MS1, MS2 \in Majority : \boxed{MS1 \cap MS2} \neq \{\}$$

the set of elements in both $MS1$ and $MS2$

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

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

CONSTANTS RM, $Acceptor$, $Majority$, $Ballot$

ASSUME

$$\land \forall MS1, MS2 \in Majority : MS1 \boxed{\cap} MS2 \neq \{\}$$

\intersect

\cap

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

$$\land \forall MS1, MS2 \in Majority : MS1 \cap MS2 \neq \{\}$$

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.

CONSTANTS RM, $Acceptor$, $Majority$, $Ballot$

ASSUME

$$\begin{array}{l} \land\ Ballot \subseteq Nat \\ \land\ 0 \in Ballot \\ \land\ Majority \subseteq \text{SUBSET } Acceptor \\ \land\ \forall\, MS1, MS2 \in Majority : MS1 \cap MS2 \neq \{\} \end{array}$$

TLC will check these assumptions.

TLC will check all these assumptions.

$$
\begin{aligned}
&Messages \triangleq \\
&\quad [type : \{\text{"phase1a"}\}, ins : RM, bal : Ballot \setminus \{0\}] \\
&\qquad \cup \\
&\quad [type : \{\text{"phase1b"}\}, ins : RM, mbal : Ballot, bal : Ballot \cup \{-1\}, \\
&\quad\; val : \{\text{"prepared"}, \text{"aborted"}, \text{"none"}\}, acc : Acceptor] \\
&\qquad \cup \\
&\quad [type : \{\text{"phase2a"}\}, ins : RM, bal : Ballot, val : \{\text{"prepared"}, \text{"aborted"}\}] \\
&\qquad \cup \\
&\quad [type : \{\text{"phase2b"}\}, acc : Acceptor, ins : RM, bal : Ballot, \\
&\quad\; val : \{\text{"prepared"}, \text{"aborted"}\}] \\
&\qquad \cup \\
&\quad [type : \{\text{"Commit"}, \text{"Abort"}\}]
\end{aligned}
$$

The module next defines $Messages$ to be a set consisting of several kinds of records.

$$
\begin{array}{l}
Messages \triangleq \\
\quad [type : \{\text{"phase1a"}\}, ins : RM, bal : \boxed{Ballot \setminus \{0\}}] \\
\qquad \cup \\
\quad [type : \{\text{"phase1b"}\}, ins : RM, mbal : Ballot, bal : Ballot \cup \{-1\}, \\
\quad\; val : \{\text{"prepared"}, \text{"aborted"}, \text{"none"}\}, acc : Acceptor] \\
\qquad \cup \\
\quad [type : \{\text{"phase2a"}\}, ins : RM, bal : Ballot, val : \{\text{"prepared"}, \text{"aborted"}\}] \\
\qquad \cup \\
\quad [type : \{\text{"phase2b"}\}, acc : Acceptor, ins : RM, bal : Ballot, \\
\quad\; val : \{\text{"prepared"}, \text{"aborted"}\}] \\
\qquad \cup \\
\quad [type : \{\text{"Commit"}, \text{"Abort"}\}]
\end{array}
$$

The module next defines $Messages$ to be a set consisting of several kinds of records. The definition contains this expression.

$Ballot \setminus \{0\}$

The module next defines $Messages$ to be a set consisting of several kinds of records. The definition contains this expression.

$Ballot \boxed{\setminus} \{0\}$

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,

$Ballot \setminus \{0\}$

$S \setminus T$ is the set of elements in S not in T.

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,

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

$Ballot \setminus \{0\}$

$S \setminus T$ is the set of elements in S not in T.

$(10\,..\,20) \setminus (1\,..\,14) \;=\; 15\,..\,20$

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.

$Ballot \setminus \{0\}$

The set of non-0 elements in $Ballot$.

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, $Ballot$ set-minus the set containing only 0 is the set of non-zero elements in $Ballot$.

$\text{VARIABLES } rmState, aState, msgs$

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

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

$\text{VARIABLES } rmState,\ aState,\ \boxed{msgs}$

$$
\begin{array}{l}
PCTypeOK \triangleq \\
\quad \land\ rmState \in [RM \to \{\text{"working"}, \text{"prepared"}, \text{"committed"}, \text{"aborted"}\}] \\
\quad \land\ aState \in [RM \to [Acceptor \to [\begin{array}[t]{@{}l@{\ }l}
mbal & : Ballot, \\
bal & : Ballot \cup \{-1\}, \\
val & : \{\text{"prepared"}, \text{"aborted"}, \text{"none"}\}]]]
\end{array} \\
\quad \land\ \boxed{msgs \subseteq Messages}
\end{array}
$$

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.

VARIABLES $rmState$, $\boxed{aState}$, $msgs$

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

$PCTypeOK$ also asserts that the value of the variable $aState$

VARIABLES $rmState, aState, msgs$

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

$PCTypeOK$ also asserts that the value of the variable $aState$ is a function with domain RM

VARIABLES $rmState$, $aState$, $msgs$

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

$r \in RM$ implies $aState[r]$ is a function with domain $Acceptor$.

$PCTypeOK$ 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 $Acceptor$

VARIABLES $rmState$, $aState$, $msgs$

$$
\begin{aligned}
&PCTypeOK \triangleq \\
&\quad \land\ rmState \in [RM \rightarrow \{\text{"working"}, \text{"prepared"}, \text{"committed"}, \text{"aborted"}\}] \\
&\quad \land\ aState \in [RM \rightarrow [Acceptor \rightarrow \boxed{\begin{array}{l}[mbal : Ballot, \\ \ bal : Ballot \cup \{-1\}, \\ \ val : \{\text{"prepared"}, \text{"aborted"}, \text{"none"}\}]\end{array}}]] \\
&\quad \land\ msgs \subseteq Messages
\end{aligned}
$$

$r \in RM$ implies $aState[r]$ is a function with domain $Acceptor$.

$a \in Acceptor$ 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 $Acceptor$ such that for every a in the set $Acceptor$, $aState[r][a]$ is a record these three fields

VARIABLES $rmState$, $aState$, $msgs$

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

$r \in RM$ implies $aState[r]$ is a function with domain $Acceptor$.

$a \in Acceptor$ implies $aState[r][a]$ is a record with three fields.

$aState[r][a].bal$ is in $Ballot$ 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 $Acceptor$ such that for every a in the set $Acceptor$, $aState[r][a]$ is a record these three fields

And, for example, $aState[r][a].bal$ is in the set $Ballot$ or equals -1.

VARIABLES $rmState,\ aState,\ msgs$

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

There's nothing new here.

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

$$
\begin{array}{l}
Phase2a(bal,\ r) \triangleq \\
\quad \land \neg \exists\, m \in msgs : \begin{array}[t]{l} \land\ m.type = \text{“phase2a”} \\ \land\ m.bal = bal \\ \land\ m.ins = r \end{array} \\
\quad \land \exists\, MS \in Majority : \\
\qquad \textsc{let}\ mset \triangleq \{m \in msgs : \begin{array}[t]{lll} \land\ m.type & = & \text{“phase1b”} \\ \land\ m.ins & = & r \\ \land\ m.mbal & = & bal \\ \land\ m.acc & \in & MS \quad \} \end{array} \\
\qquad\qquad maxbal \triangleq Maximum(\{m.bal : m \in mset\}) \\
\qquad\qquad val \triangleq \textsc{if}\ maxbal = -1 \\
\qquad\qquad\qquad\quad \textsc{then}\ \text{“aborted”} \\
\qquad\qquad\qquad\quad \textsc{else}\ (\textsc{choose}\ m \in mset : m.bal = maxbal).val \\
\qquad \textsc{in}\quad \land\ \forall\, ac \in MS : \exists\, m \in mset : m.acc = ac \\
\qquad\qquad\ \land\ Send([type \mapsto \text{“phase2a”},\ ins \mapsto r,\ bal \mapsto bal,\ val \mapsto val]) \\
\quad \land\ \textsc{unchanged}\ \langle rmState,\ aState \rangle
\end{array}
$$

This definition of $Phase2a$, which introduces several new features of TLA+.

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

This definition of $Phase2a$, which introduces several new features of TLA+.

The first is this LET-IN expression.

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

This definition of $Phase2a$, which introduces several new features of TLA+.

The first is this LET-IN expression.

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

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.

$$
\begin{array}{l}
\text{LET } mset \triangleq \{m \in msgs : \land\ m.type = \text{"phase1b"} \\
\qquad\qquad\qquad\qquad\qquad\quad \land\ m.ins = r \\
\qquad\qquad\qquad\qquad\qquad\quad \land\ m.mbal = bal \\
\qquad\qquad\qquad\qquad\qquad\quad \land\ m.acc \in MS \quad \} \\
\qquad maxbal \triangleq Maximum(\{m.bal : m \in \boxed{mset}\}) \\
\qquad val \triangleq \text{IF } \boxed{maxbal} = -1 \\
\qquad\qquad\qquad \text{THEN } \text{"aborted"} \\
\qquad\qquad\qquad \text{ELSE } (\text{CHOOSE } m \in \boxed{mset} : m.bal = \boxed{maxbal}).val \\
\text{IN} \quad \land\ \forall\, ac \in MS : \exists\, m \in \boxed{mset} : m.acc = ac \\
\qquad\ \ \land\ Send([type \mapsto \text{"phase2a"},\ ins \mapsto r,\ bal \mapsto bal,\ val \mapsto \boxed{val}])
\end{array}
$$

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.

$$
\begin{array}{ll}
\text{LET} & mset \triangleq \{m \in msgs : \begin{array}[t]{l} \land\ m.type \ \ = \text{"phase1b"} \\ \land\ m.ins \ \ \ = r \\ \land\ m.mbal = bal \\ \land\ m.acc \ \ \in MS \end{array} \quad \} \\
 & maxbal \triangleq Maximum(\{m.bal : m \in mset\}) \\
 & val \triangleq \text{IF } maxbal = -1 \\
 & \qquad\qquad \text{THEN } \text{"aborted"} \\
 & \qquad\qquad \text{ELSE } (\text{CHOOSE } m \in mset : m.bal = maxbal).val \\
\text{IN} & \quad \land\ \forall\, ac \in MS : \exists\, m \in mset : m.acc = ac \\
 & \quad \land\ Send([type \mapsto \text{"phase2a"},\ ins \mapsto r,\ bal \mapsto bal,\ val \mapsto val])
\end{array}
$$

The next TLA+ notation introduced here is

$$\{m \in msgs : \begin{array}{l} \wedge\ m.type \ = \text{"phase1b"} \\ \wedge\ m.ins \ \ \ = r \\ \wedge\ m.mbal = bal \\ \wedge\ m.acc \ \ \in MS \end{array} \}$$

The next TLA+ notation introduced here is this set expression. It equals

$$\{m \in \boxed{msgs} : \begin{array}{l} \land\, m.type \;\; = \text{“phase1b”} \\ \land\, m.ins \;\;\; = r \\ \land\, m.mbal = bal \\ \land\, m.acc \;\;\, \in MS \end{array} \}$$

The subset of $msgs$

The next TLA+ notation introduced here is this set expression. It equals The subset of $msgs$

$$\{\boxed{m} \in msgs : \begin{array}{l} \land\, m.type \;\; = \text{"phase1b"} \\ \land\, m.ins \;\;\; = r \\ \land\, m.mbal = bal \\ \land\, m.acc \;\;\; \in MS \end{array} \}$$

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

$$\{m \in msgs : \boxed{\begin{aligned} &\land m.type &&= \text{"phase1b"} \\ &\land m.ins &&= r \\ &\land m.mbal &&= bal \\ &\land m.acc &&\in MS \end{aligned}}\}$$

The subset of $msgs$ containing all elements m satisfying this formula.

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.

$$
\begin{array}{ll}
\text{LET} & mset \triangleq \{m \in msgs : \begin{array}[t]{l} \land\ m.type = \text{"phase1b"} \\ \land\ m.ins = r \\ \land\ m.mbal = bal \\ \land\ m.acc \in MS \quad \} \end{array} \\
 & maxbal \triangleq Maximum(\{m.bal : m \in mset\}) \\
 & val \triangleq \text{IF } maxbal = -1 \\
 & \qquad\qquad \text{THEN } \text{"aborted"} \\
 & \qquad\qquad \text{ELSE } (\text{CHOOSE } m \in mset : m.bal = maxbal).val \\
\text{IN} & \quad \land\ \forall\, ac \in MS : \exists\, m \in mset : m.acc = ac \\
 & \quad \land\ Send([type \mapsto \text{"phase2a"},\ ins \mapsto r,\ bal \mapsto bal,\ val \mapsto val])
\end{array}
$$

The LET-IN expression also introduces another set notation.

$$\{m.bal : m \in mset\}$$

The LET-IN expression also introduces another set notation.

$$\{\boxed{m.bal} : m \in mset\}$$

The set of all $m.bal$

The LET-IN expression also introduces another set notation.

This expresion equals the set of all elements of the form $m.bal$

$$\{m.bal : \boxed{m \in mset}\}$$

The set of all $m.bal$ with m in $mset$.

The LET-IN expression also introduces another set notation.

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

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.

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

$\{n^2 : n \in Nat\}$

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.

$$
\begin{array}{ll}
\text{LET} & mset \triangleq \{m \in msgs : \begin{array}{l} \land\ m.type = \text{"phase1b"} \\ \land\ m.ins = r \\ \land\ m.mbal = bal \\ \land\ m.acc \in MS \end{array} \} \\
 & maxbal \triangleq Maximum(\{m.bal : m \in mset\}) \\
 & val \triangleq \text{IF } maxbal = -1 \\
 & \qquad\qquad \text{THEN } \text{"aborted"} \\
 & \qquad\qquad \text{ELSE } (\text{CHOOSE } m \in mset : m.bal = maxbal).val \\
\text{IN} & \land\ \forall\, ac \in MS : \exists\, m \in mset : m.acc = ac \\
 & \land\ Send([type \mapsto \text{"phase2a"},\ ins \mapsto r,\ bal \mapsto bal,\ val \mapsto val])
\end{array}
$$

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

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

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

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

Choice of m need not be unique.

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.

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

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 $m.val$.

$$
\begin{array}{l}
Phase2a(bal, r) \triangleq \\
\quad \land \neg \exists m \in msgs : \land m.type = \text{"phase2a"} \\
\qquad\qquad\qquad\qquad\quad \land m.bal = bal \\
\qquad\qquad\qquad\qquad\quad \land m.ins = r \\
\quad \land \exists MS \in Majority : \\
\qquad \text{LET } mset \triangleq \{m \in msgs : \land m.type = \text{"phase1b"} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \land m.ins = r \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \land m.mbal = bal \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \land m.acc \in MS \quad \} \\
\qquad\qquad maxbal \triangleq Maximum(\{m.bal : m \in mset\}) \\
\qquad\qquad val \triangleq \text{IF } maxbal = -1 \\
\qquad\qquad\qquad\quad \text{THEN } \text{"aborted"} \\
\qquad\qquad\qquad\quad \text{ELSE } (\text{CHOOSE } m \in mset : m.bal = maxbal).val \\
\qquad \text{IN } \quad \land \forall ac \in MS : \exists m \in mset : m.acc = ac \\
\qquad\qquad\quad \land Send([type \mapsto \text{"phase2a"}, ins \mapsto r, bal \mapsto bal, val \mapsto val]) \\
\quad \land \text{UNCHANGED } \langle rmState, aState \rangle
\end{array}
$$

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.

$$
\begin{array}{l}
Phase1b(acc) \triangleq \\
\quad \exists\, m \in msgs : \\
\qquad \land\ m.type = \text{"phase1a"} \\
\qquad \land\ aState[m.ins][acc].mbal < m.bal \\
\qquad \land\ aState' = [aState \ \text{EXCEPT}\ ![m.ins][acc].mbal = m.bal] \\
\qquad \land\ Send([\begin{array}[t]{@{}l@{\ }l@{}} type & \mapsto \text{"phase1b"}, \\ ins & \mapsto m.ins, \\ mbal & \mapsto m.bal, \\ bal & \mapsto aState[m.ins][acc].bal, \\ val & \mapsto aState[m.ins][acc].val, \\ acc & \mapsto acc]) \end{array} \\
\qquad \land\ \text{UNCHANGED}\ rmState
\end{array}
$$

The next new construct is in this definition.

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

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.

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

The next new construct is in this definition.

In this subformula. you haven't seen this form of EXCEPT expression. It's an abbreviation for

[slide 130]

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

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

$aState$ EXCEPT its value on $m.ins$ equals

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

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

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

$aState$ EXCEPT its value on $m.ins$ equals

$aState$ of $m.ins$ EXCEPT its value on a-c-c equals

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

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

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

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

$aState$ EXCEPT its value on $m.ins$ equals

$aState$ of $m.ins$ EXCEPT its value on a-c-c equals

$aState$ of $m.ins$ of a-c-c EXCEPT its m-bal component equals

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

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

$aState$ EXCEPT its value on $m.ins$ equals
$aState$ of $m.ins$ EXCEPT its value on a-c-c equals
$aState$ of $m.ins$ of a-c-c EXCEPT its m-bal component equals m dot bal.
Whew.

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

$$\begin{array}{l}[aState \text{ EXCEPT } ![m.ins] = \\ \quad [aState[m.ins] \text{ EXCEPT } ![acc] = \\ \qquad [aState[m.ins][acc] \text{ EXCEPT } !.mbal = m.bal]]]\end{array}$$

If you stop and decipher this, you'll see that

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

If you stop and decipher this, you'll see that
this formula corresponds to

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

```
aState[m.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.

$$
\begin{array}{l}
Phase2b(acc) \triangleq \\
\quad \land \exists\, m \in msgs : \\
\qquad \land\ m.type = \text{“phase2a”} \\
\qquad \land\ aState[m.ins][acc].mbal \leq m.bal \\
\qquad \land\ aState' = [aState \text{ EXCEPT } ![m.ins][acc].mbal = m.bal, \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad ![m.ins][acc].bal \quad = m.bal, \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad ![m.ins][acc].val \quad = m.val\,] \\
\qquad \land\ Send([type \mapsto \text{“phase2b”},\ ins \mapsto m.ins,\ bal \mapsto m.bal, \\
\qquad\qquad\qquad val \mapsto m.val,\ acc \mapsto acc]) \\
\quad \land \text{ UNCHANGED } rmState
\end{array}
$$

This definition contains

$$
\begin{aligned}
aState' = [aState \text{ EXCEPT } & ![m.ins][acc].mbal = m.bal, \\
& ![m.ins][acc].bal \quad = m.bal, \\
& ![m.ins][acc].val \quad = m.val\,]
\end{aligned}
$$

This definition contains another generalization of the EXCEPT construct.

no pause

$$
\begin{aligned}
aState' = [aState \text{ EXCEPT } & ![m.ins][acc].mbal = m.bal, \\
& ![m.ins][acc].bal \quad = m.bal, \\
& ![m.ins][acc].val \quad = m.val\,]
\end{aligned}
$$

This definition contains another generalization of the EXCEPT construct.

no pause

$$
\begin{array}{rl}
[aState \text{ EXCEPT } & ![m.ins][acc].mbal = m.bal, \\
& ![m.ins][acc].bal \quad = m.bal, \\
& ![m.ins][acc].val \quad = m.val\,]
\end{array}
$$

This definition contains another generalization of the EXCEPT construct.

no pause

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

$$\begin{aligned} aState' = [aState \text{ EXCEPT } & ![m.ins][acc].mbal = m.bal, \\ & ![m.ins][acc].bal \quad = m.bal, \\ & ![m.ins][acc].val \quad = m.val\,] \end{aligned}$$

This definition contains another generalization of the EXCEPT construct.

no pause

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

$$
\begin{aligned}
aState' = [aState \text{ EXCEPT } & ![m.ins][acc].mbal = m.bal, \\
& ![m.ins][acc].bal \quad = m.bal, \\
& ![m.ins][acc].val \quad = m.val\,]
\end{aligned}
$$

```
aState[m.ins][acc].mbal = m.bal;
aState[m.ins][acc].bal  = m.bal;
aState[m.ins][acc].val  = m.val;
```

executing this sequence of three program statements.

Notice the correspondence between the parts of the EXCEPT expression

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

```
aState[m.ins][acc].mbal = m.bal;
aState[m.ins][acc].bal  = m.bal;
aState[m.ins][acc].val  = m.val;
```

executing this sequence of three program statements.

Notice the correspondence between the parts of the EXCEPT expression and the program statements.

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

```
aState[m.ins][acc].mbal = m.bal;
aState[m.ins][acc].bal  = m.bal;
aState[m.ins][acc].val  = m.val;
```

executing this sequence of three program statements.

Notice the correspondence between the parts of the EXCEPT expression and the program statements.

$$
\begin{aligned}
aState' = [aState \text{ EXCEPT } & ![m.ins][acc].mbal = m.bal, \\
& \boxed{![m.ins][acc].bal \quad = m.bal}, \\
& ![m.ins][acc].val \quad = m.val\,]
\end{aligned}
$$

```
aState[m.ins][acc].mbal = m.bal;
aState[m.ins][acc].bal  = m.bal;
aState[m.ins][acc].val  = m.val;
```

executing this sequence of three program statements.

Notice the correspondence between the parts of the EXCEPT expression and the program statements.

$$
\begin{aligned}
aState' = [aState \text{ EXCEPT } & ![m.ins][acc].mbal = m.bal, \\
& ![m.ins][acc].bal \quad = m.bal, \\
& \boxed{![m.ins][acc].val \quad = m.val}]
\end{aligned}
$$

```
aState[m.ins][acc].mbal = m.bal;
aState[m.ins][acc].bal  = m.bal;
aState[m.ins][acc].val  = m.val;
```

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

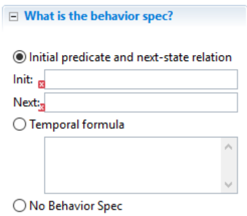
Open module $PaxosCommit$ in the Toolbox

Open $PaxosCommit$ in the Toolbox and create a new model.

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

Open $PaxosCommit$ in the Toolbox and create a new model.

You have to enter

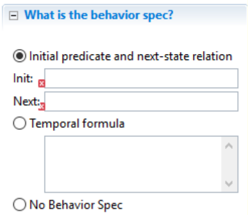


Open module $PaxosCommit$ in the Toolbox and create a new model.

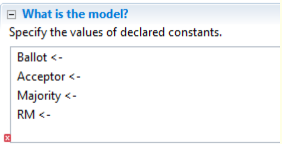
You have to enter the initial and next-state formulas

Open $PaxosCommit$ in the Toolbox and create a new model.

You have to enter



and

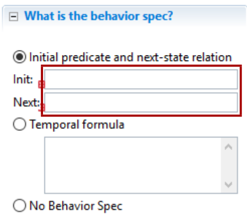


Open module $PaxosCommit$ in the Toolbox and create a new model.

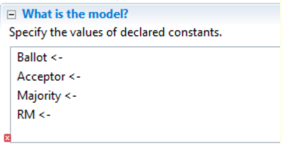
You have to enter the initial and next-state formulas and the values of the constants.

Open $PaxosCommit$ in the Toolbox and create a new model.

You have to enter



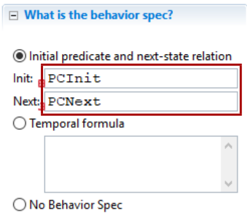
and



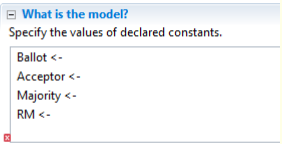
The initial and next-state formulas are named

Open $PaxosCommit$ in the Toolbox and create a new model.

You have to enter



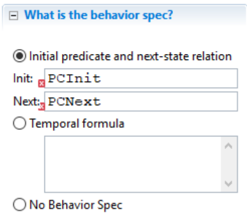
and



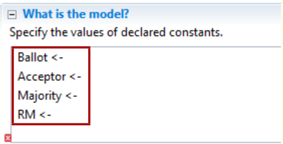
The initial and next-state formulas are named $PCInit$ and $PCNext$.

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

You have to enter



and



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

Now for the values assigned to the constants.

Ballot ←
Acceptor ←
Majority ←
RM ←

We normally start with a tiny model

We normally start with a tiny model

Ballot ←
Acceptor ←
Majority ←
RM ←

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

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

Ballot ←
Acceptor ←
Majority ←
RM ←

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 $\leftarrow$ $\boxed{\texttt{\{a1, a2, a3\}}}$ 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 $Acceptor$,

Ballot ←
Acceptor ← `{a1, a2, a3}`
Majority ←
RM ← $\boxed{\texttt{\{r1, r2\}}}$ a set of model values

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 $Acceptor$, and a set of two model values to RM.

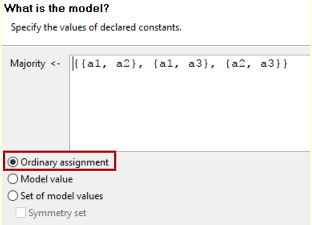
Ballot ← $\boxed{\texttt{\{0, 1\}}}$
Acceptor ← `{a1, a2, a3}`
Majority ←
RM ← `{r1, r2}`

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

```
Ballot ← {0, 1}
Acceptor ← {a1, a2, a3}
Majority ← {{a1, a2}, {a1, a3}, {a2, a3}}
RM ← {r1, r2}
```

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

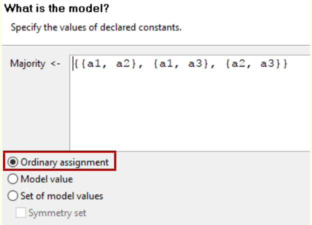
Ballot ← {0, 1}
Acceptor ← {a1, a2, a3}
Majority ← $\boxed{\texttt{\{\{a1, a2\}, \{a1, a3\}, \{a2, a3\}\}}}$
RM ← {r1, r2}



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 ← $\boxed{\texttt{\{a1, a2, a3\}}}$
Majority ← {{a1, a2}, {a1, a3}, {a2, a3}}
RM ← {r1, r2}



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 a1, a2, and a3 are declared in the assignment of a set of model values to $Acceptor$.

Ballot ← `{0, 1}`
Acceptor ← `{a1, a2, a3}`
Majority ← `{{a1, a2}, {a1, a3}, {a2, a3}}`
RM ← `{r1, r2}`

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 $a1$, $a2$, and $a3$ are declared in the assignment of a set of model values to $Acceptor$.

Ballot $\leftarrow$ `{0, 1}`
Acceptor $\leftarrow$ `{a1, a2, a3}`
Majority $\leftarrow$ `{{a1, a2}, {a1, a3}, {a2, a3}}`
RM $\leftarrow$ `{r1, r2}`

This can be a symmetry set, because `r1` and `r2` aren't used elsewhere.

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

Ballot ← `{0, 1}`
Acceptor ← $\boxed{\texttt{\{a1, a2, a3\}}}$
Majority ← `{{a1, a2}, {a1, a3}, {a2, a3}}`
RM ← `{r1, r2}`

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 $\leftarrow$ `{0, 1}`
Acceptor $\leftarrow$ `{a1, a2, a3}`
Majority $\leftarrow$ `{{a1, a2}, {a1, a3}, {a2, a3}}`
RM $\leftarrow$ `{r1, r2}`

But what about this set?

`a1`, `a2`, `a3` 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 $\leftarrow$ `{a1, a2, a3}`
Majority $\leftarrow$ `{{a1, a2}, {a1, a3}, {a2, a3}}`
RM $\leftarrow$ `{r1, r2}`

This use is OK because the expression is symmetric in `a1`, `a2`, `a3`.

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 $\leftarrow$ `{a1, a2, a3}`
Majority $\leftarrow$ `{{a1, a2}, {a1, a3}, {a2, a3}}`
RM $\leftarrow$ `{r1, r2}`

This use is OK because the expression is symmetric in `a1`, `a2`, `a3` .

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.

Ballot ← `{0, 1}`
Acceptor ← `{a1, a2, a3}`
Majority ← `{{a1, a2}, {a1, a3}, {a2, a3}}`
RM ← `{r1, r2}`

For example, interchanging `a1` ↔ `a3` in

`{{a1, a2}, {a1, a3}, {a2, a3}}`

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

For example, if we interchange $a1$ and $a3$ in the expression,

Ballot $\leftarrow$ `{0, 1}`
Acceptor $\leftarrow$ `{a1, a2, a3}`
Majority $\leftarrow$ `{{a1, a2}, {a1, a3}, {a2, a3}}`
RM $\leftarrow$ `{r1, r2}`

For example, interchanging `a1` $\leftrightarrow$ `a3` in

`{{a1, a2}, {a1, a3}, {a2, a3}}`

produces $\updownarrow$ $\updownarrow$ $\updownarrow$ $\updownarrow$

`{{a3, a2}, {a3, a1}, {a2, a1}}`

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

For example, if we interchange $a1$ and $a3$ in the expression, we get this expression.

Ballot $\leftarrow$ `{0, 1}`
Acceptor $\leftarrow$ `{a1, a2, a3}`
Majority $\leftarrow$ `{{a1, a2}, {a1, a3}, {a2, a3}}`
RM $\leftarrow$ `{r1, r2}`

These two sets are equal:

`{{a1, a2}, {a1, a3}, {a2, a3}}`

`{{a3, a2}, {a3, a1}, {a2, a1}}`

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

For example, if we interchange $a1$ and $a3$ in the expression, we get this expression.

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

Ballot ← {0, 1}
Acceptor ← {a1, a2, a3}
Majority ← {{a1, a2}, {a1, a3}, {a2, a3}}
RM ← {r1, r2}

These two sets are equal:

{{a1, a2}, {a1, a3}, {a2, a3}}

{{a3, a2}, {a3, a1}, {a2, a1}}

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

For example, if we interchange $a1$ and $a3$ in the expression, we get this expression.

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

Ballot $\leftarrow$ {0, 1}
Acceptor $\leftarrow$ {a1, a2, a3}
Majority $\leftarrow$ {{a1, a2}, {a1, a3}, {a2, a3}}
RM $\leftarrow$ {r1, r2}

These two sets are equal:

{{a1, a2}, {a1, a3}, {a2, a3}}

{{a3, a2}, {a3, a1}, {a2, a1}}

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

For example, if we interchange $a1$ and $a3$ in the expression, we get this expression.

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

Ballot $\leftarrow$ `{0, 1}`
Acceptor $\leftarrow$ `{a1, a2, a3}`
Majority $\leftarrow$ `{{a1, a2}, {a1, a3}, {a2, a3}}`
RM $\leftarrow$ `{r1, r2}`

These two sets are equal:

`{{a1, a2}, {a1, a3}, {a2, a3}}`

`{{a3, a2}, {a3, a1}, {a2, a1}}`

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

For example, if we interchange $a1$ and $a3$ 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

[slide 176]

Ballot ← `{0, 1}`
Acceptor ← $\boxed{\texttt{\{a1, a2, a3\}}}$
Majority ← `{{a1, a2}, {a1, a3}, {a2, a3}}`
RM ← `{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 ← `{0, 1}`
Acceptor ← `{a1, a2, a3}`
Majority ← $\boxed{\texttt{\{\{a1, a2\}, \{a1, a3\}, \{a2, a3\}\}}}$
RM ← `{r1, r2}`

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 ← `{0, 1}`
Acceptor ← `{a1, a2, a3}`
Majority ← `{{a1, a2}, {a1, a3}, {a2, a3}}`
RM ← `{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 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 $\leftarrow$ `{a1, a2, a3}`
Majority $\leftarrow$ `{{a1, a2}, {a1, a3}, {a2, a3}}`
RM $\leftarrow$ `{r1, r2}`

There's one additional condition for symmetry sets.

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

Ballot $\leftarrow$ `{0, 1}`
Acceptor $\leftarrow$ `{`$\boxed{\texttt{a1, a2, a3}}$`}`
Majority $\leftarrow$ `{{a1, a2}, {a1, a3}, {a2, a3}}`
RM $\leftarrow$ `{`$\boxed{\texttt{r1, r2}}$`}`

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,

Ballot ← `{0, 1}`
$\boxed{\text{Acceptor}}$ ← `{a1, a2, a3}`
$\boxed{\text{Majority}}$ ← `{{a1, a2}, {a1, a3}, {a2, a3}}`
$\boxed{\text{RM}}$ ← `{r1, r2}`

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

Ballot ← `{0, 1}`
Acceptor ← `{a1, a2, a3}`
Majority ← `{{a1, a2}, {a1, a3}, {a2, a3}}`
RM ← `{r1, r2}`

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 ← `{0, 1}`
Acceptor ← `{a1, a2, a3}`
Majority ← `{{a1, a2}, {a1, a3}, {a2, a3}}`
RM ← `{r1, r2}`

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

```
Ballot <- {0, 1}
Acceptor <- {a1, a2, a3}
Majority <- {{a1, a2}, {a1, a3}, {a2, a3}}
RM <- {r1, r2}
```

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.

Ballot ← `{0, 1}`

$\boxed{\text{Acceptor}}$ ← `{a1, a2, a3}`

$\boxed{\text{Majority}}$ ← `{{a1, a2}, {a1, a3}, {a2, a3}}`

$\boxed{\text{RM}}$ ← `{r1, r2}`

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,

Ballot $\leftarrow$ `{0, 1}`
$\boxed{\text{Acceptor}}$ $\leftarrow$ `{a1, a2, a3}`
$\boxed{\text{Majority}}$ $\leftarrow$ `{{a1, a2}, {a1, a3}, {a2, a3}}`
$\boxed{\text{RM}}$ $\leftarrow$ `{r1, r2}`

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.

$$\begin{array}{l} \text{Ballot} \leftarrow \texttt{\{0, 1\}} \\ \boxed{\text{Acceptor}} \leftarrow \texttt{\{a1, a2, a3\}} \\ \boxed{\text{Majority}} \leftarrow \texttt{\{\{a1, a2\}, \{a1, a3\}, \{a2, a3\}\}} \\ \boxed{\text{RM}} \leftarrow \texttt{\{r1, r2\}} \end{array}$$

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.

Ballot $\leftarrow$ `{0, 1}`
Acceptor $\leftarrow$ `{a1, a2, a3}`
Majority $\leftarrow$ `{{a1, a2}, {a1, a3}, {a2, a3}}`
RM $\leftarrow$ `{r1, r2}`

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:

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 $PCTypeOK$

– Invariant $TCConsistent$ imported from module $TCommit$

The type correctness invariant $PCTypeOK$ that we looked at earlier

and the invariant $TCConsistent$, 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.

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.

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.

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 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 $Ballot$ 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 $Ballot$ 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?

What good is checking such small models?

What good is checking such small models?

Make this change to the model.

Majority $\leftarrow$ `{{a1, a2}, {a1, a3}, {a2, a3}}`

What good is checking such small models?

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

What good is checking such small models?

Make this change to the model.

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

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.

What good is checking such small models?

Make this change to the model.

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

What good is checking such small models?

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

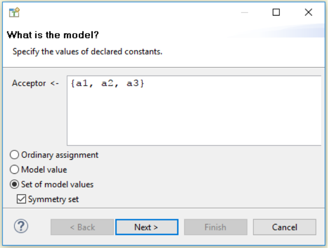
Delete this element of an element of the set.

The expression is no longer symmetric in a1, a2, and a3.

What good is checking such small models?

Make this change to the model.

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

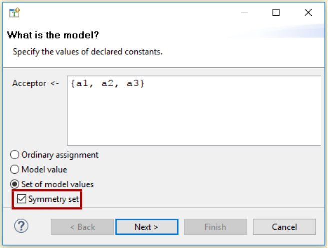


So we have to change the assignment to $Acceptor$

What good is checking such small models?

Make this change to the model.

Majority $\leftarrow$ `{{a1, a2}, {a1, a3}, {a3}}`



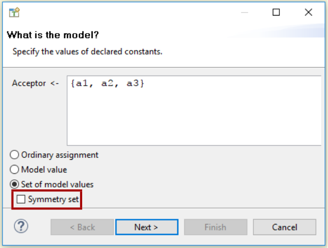
So we have to change the assignment to $Acceptor$

So it’s no longer a symmetry set.

What good is checking such small models?

Make this change to the model.

Majority $\leftarrow$ `{{a1, a2}, {a1, a3}, {a3}}`



So we have to change the assignment to $Acceptor$

So it's no longer a symmetry set.

What good is checking such small models?

Make this change to the model.

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

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

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

What good is checking such small models?

Make this change to the model.

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

```
ASSUME
  /\ Ballot \subseteq Nat
  /\ 0 \in Ballot
  /\ Majority \subseteq 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

What good is checking such small models?

Make this change to the model.

Majority ← {{a1, a2}, {a1, a3}, {a3}}

```
ASSUME
  /\ Ballot \subseteq Nat
  /\ 0 \in Ballot
  /\ Majority \subseteq 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 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.

Majority ← {{a1, a2}, {a1, a3}, {a3}}

```
ASSUME
  /\ Ballot \subseteq Nat
  /\ 0 \in Ballot
  /\ Majority \subseteq 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 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.

Majority ← {{a1, a2}, {a1, a3}, {a3}}

```
ASSUME
  /\ Ballot \subseteq Nat
  /\ 0 \in Ballot
  /\ Majority \subseteq 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 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.

Majority ← {{a1, a2}, {a1, a3}, {a3}}

```
ASSUME
  /\ Ballot \subseteq Nat
  /\ 0 \in Ballot
  /\ Majority \subseteq 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 this assertion is no longer true. So, we have to comment it out.

Run TLC on the model.

Run TLC on the model.

Run TLC on the model.

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

Run TLC on the model.

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

Run TLC on the model.

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

TLC reports that invariant $TCConsistent$ is violated

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

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.

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