TRANSACTION COMMIT

This video should be viewed in conjunction with a Web page. To find that page, search the Web for TLA+ Video Course.
This lecture is about matrimony. Actually, it’s the first of three lectures about a problem from the domain of databases called transaction commit. Transaction commit is a very simple problem, but it is about computer systems and not Hollywood action heros. Jim Gray was a computer scientist who, in the words of his Turing award citation, “made seminal contributions to database and transaction processing research.” I had the privilege of knowing Jim for many years. He used to describe transaction commit in terms of a wedding. I learned long ago that if Jim did something, it was the right thing to do. So this lecture begins with a discussion of weddings.

[slide 2]
WEDDINGS
An Old-Fashioned Wedding

Here’s how an old-fashioned wedding goes.
Here’s how an old-fashioned wedding goes.

There’s the bride, let’s call her Anne.
An Old-Fashioned Wedding

Here’s how an old-fashioned wedding goes.
There’s the bride, let’s call her Anne.
There’s the groom, let’s call him Henry.
An Old-Fashioned Wedding

Here’s how an old-fashioned wedding goes.

There’s the bride, let’s call her Anne.
There’s the groom, let’s call him Henry.

And there’s Thomas, the minister.

The minister begins by asking:
An Old-Fashioned Wedding

Here’s how an old-fashioned wedding goes.

There’s the bride, let’s call her Anne.

There’s the groom, let’s call him Henry.

And there’s Thomas, the minister.

The minister begins by asking:  Henry, wilt thou have this woman to thy wedded wife?
A More Modern Wedding

Let’s make it more modern. A 21st century minister might say:
Let’s make it more modern. A 21st century minister might say: **Hank, are you prepared to commit to this relationship?**

To which Henry would reply:
Let’s make it more modern. A 21st century minister might say: Hank, are you prepared to commit to this relationship?

To which Henry would reply:

I’m prepared.

The minister then asks:
Let’s make it more modern. A 21st century minister might say: Hank, are you prepared to commit to this relationship?

To which Henry would reply:

I’m prepared.

The minister then asks:

Anne, are you prepared to commit to this relationship? And Anne replies:
I’m prepared.
The minister then says:
I’m prepared.
The minister then says:

You’re now both in a committed relationship.
What Can Go Wrong

What can go wrong in a wedding?

When the minister asks one of them, say the bride:

[ slide 15 ]
What can go wrong in a wedding?

When the minister asks one of them, say the bride: Are you prepared to commit to this relationship?

She might answer...
What can go wrong in a wedding?

When the minister asks one of them, say the bride: Are you prepared to commit to this relationship?

She might answer No!
What can go wrong in a wedding?

When the minister asks one of them, say the bride: Are you prepared to commit to this relationship?

She might answer No!

The minister would then abort the wedding.
What Can Go Wrong

Here’s another way the wedding can go amiss.
Here’s another way the wedding can go amiss.

Both the groom

I’m prepared.
What Can Go Wrong

Here’s another way the wedding can go amiss.
Both the groom and the bride might say they’re prepared.
What Can Go Wrong

Stop! Anne will regret it.

Here’s another way the wedding can go amiss. Both the groom and the bride might say they’re prepared. But someone else at the wedding might object.
This wedding is aborted.

Here’s another way the wedding can go amiss. Both the groom and the bride might say they’re prepared. But someone else at the wedding might object. The minister could then decide that it was a valid objection and abort the wedding.
What a Wedding Accomplishes

What does a wedding accomplish?

[ slide 24 ]
What does a wedding accomplish?

A wedding begins with the bride and groom possibly unsure if they should be married.
What does a wedding accomplish?

A wedding begins with the bride and groom possibly unsure if they should be married.

It allows them each to decide if they’re prepared to commit to the relationship.
What does a wedding accomplish?

A wedding begins with the bride and groom possibly unsure if they should be married.

It allows them each to decide if they’re prepared to commit to the relationship or if they want the wedding aborted.
What a Wedding Accomplishes

Minister

committed

Henry
committed

Anne

It should finish with them both believing they are in a committed relationship.

[ slide 28 ]
What a Wedding Accomplishes

It should finish with them both believing they are in a committed relationship
Or both believing that the wedding was aborted.
What a Wedding Accomplishes

It should finish with them both believing they are in a committed relationship
Or both believing that the wedding was aborted.

It should be impossible for them to disagree about the outcome.
The Minister

What function does the minister perform?
The Minister

He implements the wedding.

What function does the minister perform?

His job is to implement the wedding.
What function does the minister perform?

His job is to implement the wedding.

He’s part of *how* the wedding works,
The Minister

He implements the wedding.

He’s part of how it works, not what it does.

What function does the minister perform?

His job is to implement the wedding.

He’s part of how the wedding works, not part of what the wedding is supposed to accomplish.

[slide 34]
Specifying a Wedding

We’re going to write a specification of a wedding.
Specifying a Wedding

What a wedding accomplishes, not how it’s performed.

We’re going to write a specification of a wedding.

A specification of what a wedding should accomplish, not how it’s actually performed.
We’re going to write a specification of a wedding.

A specification of what a wedding should accomplish, not how it’s actually performed.

What, not how.
Specifying a Wedding

The specification mentions only the bride and groom,
Specifying a Wedding

The specification mentions only the bride and groom, not the minister.

The specification mentions only the bride and groom, not the minister, who’s part of how, not what.
Here’s the state/transition diagram of each of the two participants: the bride and the groom.
Here’s the state/transition diagram of each of the two participants: the bride and the groom.

Each participant starts in the state of being unsure about what he or she wants to do.
Here’s the state/transition diagram of each of the two participants: the bride and the groom.

Each participant starts in the state of being unsure about what he or she wants to do.

From that state, they can go into either the prepared or the aborted state.
The state/transition diagram of each participant:

From the prepared state,
From the prepared state, they can go to either the committed or aborted state.
The state/transition diagram of each participant:

- **unsure**
- **prepared**
- **committed**
- **aborted**

From the prepared state, they can go to either the committed or aborted state.

They remain forever in either of those two states.

[slide 45]
The state/transition diagram of each participant:

```
unsure

prepared

committed aborted
```

From the prepared state, they can go to either the committed or aborted state.

They remain forever in either of those two states.
We can generalize all this to a really modern wedding.
A Really Modern Wedding

Not limited to a bride and a groom.

We can generalize all this to a really modern wedding.

One that’s not limited to just one bride and one groom.
A Really Modern Wedding

Not limited to a bride and a groom.

Can have any number of participants.

We can generalize all this to a really modern wedding.
One that’s not limited to just one bride and one groom.

We can have any number of participants.
We can generalize all this to a really modern wedding.
One that’s not limited to just one bride and one groom.
We can have any number of participants.
A Really Modern Wedding

We can generalize all this to a really modern wedding. One that’s not limited to just one bride and one groom. We can have any number of participants.
We can generalize all this to a really modern wedding. One that’s not limited to just one bride and one groom. We can have any number of participants.
We can generalize all this to a really modern wedding. One that's not limited to just one bride and one groom. We can have any number of participants.
A Really Modern Wedding

We can generalize all this to a really modern wedding. One that’s not limited to just one bride and one groom. We can have any number of participants.

[slide 54]
We can generalize all this to a really modern wedding.
One that’s not limited to just one bride and one groom.
We can have any number of participants.
We can generalize all this to a really modern wedding. One that’s not limited to just one bride and one groom. We can have any number of participants.
Each participant has the same states as before.

Each participant has the same permitted states and state transitions as before.
Each participant has the same states as before.

Cannot have one participant committed and another aborted.

Each participant has the same permitted states and state transitions as before.

We cannot allow one participant to believe the relationship is committed and another to believe it was aborted.

[slide 58]
In a transaction commit . . .
A Wedding

Henry

Catherine A

Anne B

Jane

Anne C

Catherine H

Catherine P

a wedding
A Database Transaction

Resource Manager

Resource Manager

Resource Manager

Resource Manager

Resource Manager

Resource Manager

Resource Manager

Resource Manager

A wedding is replaced by a database transaction.
A Database Transaction

Is performed by the RMs.

A wedding is replaced by a database transaction.

The transaction is performed by a collection of processes called Resource Managers.
A Database Transaction

Is performed by the RMs.

Can either commit or abort.

a wedding is replaced by a database transaction.

The transaction is performed by a collection of processes called Resource Managers.

The transaction can either commit or abort.
A Database Transaction

Can commit only if all RMs are prepared to commit.

The transaction can commit only if all resource managers are prepared to commit.
A Database Transaction

Can commit only if all RMs are prepared to commit.

Must abort if any RM wants to abort.

The transaction can commit only if all resource managers are prepared to commit.

The transaction must abort if any resource manager wants to abort.
A Database Transaction

All RMs must agree on whether it committed or aborted.

The transaction can commit only if all resource managers are prepared to commit.

The transaction must abort if any resource manager wants to abort.

All resource managers must agree on whether the transaction committed or aborted.
The execution of a transaction commit is just like a really modern wedding, which can have many participants.
The execution of a transaction commit is just like a really modern wedding, which can have many participants.

A resource manager corresponds to a participant in the wedding.
The state/transition diagram of each RM:

The state/transition diagram of each resource manager is the same as that of each participant in a wedding.
The state/transition diagram of each resource manager is the same as that of each participant in a wedding.

Except that the *unsure* state
The state/transition diagram of each resource manager is the same as that of each participant in a wedding.

Except that the *unsure* state is traditionally called the *working* state of a resource manager.
The state/transition diagram of each resource manager is the same as that of each participant in a wedding.

Except that the *unsure* state is traditionally called the *working* state of a resource manager.
We now see how transaction commit can be specified in TLA+.
You will first:

– Open the Toolbox.
– Create a new module named TCommit.
– Copy the body of the spec from the web page and paste it into the module.

So you can view the spec while watching the video, you will first
You will first:

- Open the Toolbox.

So you can view the spec while watching the video, you will first

Open the Toolbox.
You will first:

- Open the Toolbox.
- Create a new module named *TCommit*.

So you can view the spec while watching the video, you will first
Open the Toolbox.

Create a new module named *TCommit*.
You will first:

– Open the Toolbox.
– Create a new module named $TCommit$.
– Copy the body of the spec from the web page and paste it into the module.

So you can view the spec while watching the video, you will first

Open the Toolbox.

Create a new module named $TCommit$.

And copy the body of the spec from the web page and paste it into the module.
You will first:

– Open the Toolbox.
– Create a new module named \textit{TCommit}.
– Copy the body of the spec from the web page and paste it into the module.

Stop the video and do this now.

Stop the video and do this now.
The spec has comments.

You’ll see that the spec has lots of comments.
The spec has comments.

I’ll discuss them later.

You’ll see that the spec has lots of comments.

I’ll discuss comments later.
The spec has comments.
I’ll discuss them later.

Now, let’s look at the spec.

You’ll see that the spec has lots of comments.
I’ll discuss comments later.

Now, let’s look at the spec without any comments.
The spec is in module \textit{TCommit}.
The spec is in module \textit{TCommit}.

It begins by declaring \textit{RM} to be a constant.
The spec is in module `TCommit`.

It begins by declaring `RM` to be a constant which means that its value is the same throughout every behavior.
The spec is in module *TCommit*.

It begins by declaring *RM* to be a constant, which means that its value is the same throughout every behavior.

The constant *RM* represents the set of resource managers.
Why is $RM$ a set?

The spec is in module $TCommit$.

It begins by declaring $RM$ to be a constant which means that its value is the same throughout every behavior.

The constant $RM$ represents the set of resource managers.

What tells us $RM$ is a set?
Why is $RM$ a set?

In TLA+, every value is a set.
Why is $RM$ a set?

In TLA+, every value is a set.

Even 42 and “abc” are sets.
Why is $RM$ a set?

In TLA+, every value is a set.

Even 42 and “$abc$” are sets.

But TLA+ doesn’t say what their elements are.
Why is \( RM \) a set?

In TLA+, every value is a set.

Even 42 and “\( abc \)” are sets.

But TLA+ doesn’t say what their elements are.

TLC can’t evaluate \( 42 \in “abc” \).

In TLA+, every value is a set.

Even values like 42 and the string \( abc \) are sets.

But the semantics of TLA+ don’t say what the elements of the sets 42 and \( abc \) are.

So the TLC model checker will report an error if it tries to evaluate an expression like 42 is an element of \( abc \).
CONSTANT $RM$

MODULE $TCommit$

[slide 91]
Next comes the declaration of the spec’s single variable $rmState$. 

[slide 92]
Next comes the declaration of the spec’s single variable \( rmState \).

followed by the type invariant that describes what values we expect \( rmState \) to be able to assume.
Next comes the declaration of the spec’s single variable \( rmState \).

followed by the type invariant that describes what values we expect \( rmState \) to be able to assume.

We prefix standard names like \( TypeOK \) by \( TC \) because in a later video we’ll be talking about two separate specs.
CONSTANT $RM$

VARIABLE $rmState$

$TCTypeOK \triangleq$

$rmState \in \cdots$

An array indexed by RMs.

The value of $rmState$ will be an array indexed by the set of resource managers.
An array indexed by RMs.

\[ \text{rmState}[r] \text{ is the state of RM } r. \]
This is the TLA+ notation
 MODULE TCommit

CONSTANT RM

VARIABLE rmState

\[ TCTypeOK \triangleq \quad \]
\[ \text{rmState} \in [RM \rightarrow \cdots] \]

The set of all arrays indexed by elements of \( RM \)

This is the TLA+ notation for the set of all arrays indexed by elements of \( RM \)
The set of all arrays indexed by elements of $RM$ with values in \ldots.

This is the TLA+ notation for the set of all arrays indexed by elements of $RM$ with values in \ldots.
This is the TLA+ notation for the set of all arrays indexed by elements of \( RM \) with values in the set given by dot dot dot, where dot dot dot is this set whose elements are the four strings working, prepared, committed, and aborted.
This is the TLA+ notation for the set of all arrays indexed by elements of \( RM \) with values in the set given by dot dot dot, where dot dot dot is this set whose elements are the four strings working, prepared, committed, and aborted.
This is the TLA+ notation for the set of all arrays indexed by elements of $RM$ with values in the set given by dot dot dot, where dot dot dot is this set whose elements are the four strings $working$, $prepared$, $committed$, and $aborted$. 

[slide 102]
This is the TLA+ notation for the set of all arrays indexed by elements of $RM$ with values in the set given by dot dot dot. 

where dot dot dot is this set whose elements are the four strings $working$, $prepared$, $committed$, and $aborted$.
This is the TLA+ notation for the set of all arrays indexed by elements of $RM$

with values in the set given by dot dot dot.

where dot dot dot is this set whose elements are the four strings $working$, $prepared$, $committed$, $aborted$. 

[slide 104]
This is the TLA+ notation for the set of all arrays indexed by elements of $RM$ with values in the set given by dot dot dot, where dot dot dot is the set whose elements are the four strings \textit{working}, \textit{prepared}, \textit{committed}, and \textit{aborted}.

[slide 105]
This is the TLA+ notation for the set of all arrays indexed by elements of \( RM \) with values in the set given by dot dot dot. where dot dot dot is this set whose elements are the four strings \( \text{working}, \text{prepared}, \text{committed}, \text{and} \text{aborted} \) which represent the four possible states of a resource manager.
This is the TLA+ notation for the set of all arrays indexed by elements of $RM$ with values in the set given by dot dot dot.

where dot dot dot is this set whose elements are the four strings *working*, *prepared*, *committed*, and *aborted* which represent the four possible states of a resource manager.
The right arrow is typed *dash greater than* in ASCII.
The initial predicate $T CommitInit$ asserts that $rmState$ equals

\[
T CommitTypeOK \triangleq rmState \in [RM \to \{"working", "prepared", "committed", "aborted"\}]
\]

$T CommitInit \triangleq rmState = [r \in RM \mapsto "working"]$
The initial predicate $TCInit$ asserts that $rmState$ equals this expression, which is TLA+ notation for

$$TCInit \triangleq rmState = \left[ r \in RM \mapsto \text{“working”} \right]$$
The array with index set $RM$ such that

\[ \text{rmState} \in [RM \rightarrow \{ \text{"working"}, \text{"prepared"}, \text{"committed"}, \text{"aborted"} \}] \]

The initial predicate $TCHandleInit$ asserts that $\text{rmState}$ equals this expression, which is TLA+ notation for

The array with index set $RM$ such that

\[ [r \in RM \mapsto \text{"working"}][rm] = \text{"working"} \]

for all $rm$ in $RM$

The initial predicate $TCHandleInit$ asserts that $\text{rmState}$ equals this expression, which is TLA+ notation for

The array with index set equal to the set of resource managers such that the array applied to little $rm$ equals the string “working”, for every resource manager little $rm$.  

[slide 111]
The TLA+ syntax for an array expression:

\[
\begin{array}{c}
\text{variable} \in \text{set} \mapsto \text{expression}
\end{array}
\]

This is the TLA+ syntax for an array-valued expression.
The TLA+ syntax for an array expression:

\[
\begin{array}{c}
\text{variable} \in \text{set} \implies \text{expression} \\
\end{array}
\]

This is the TLA+ syntax for an array-valued expression.

Where this maps to symbol is typed \textit{bar dash greater-than} in ASCII.
The TLA+ syntax for an array expression:

$$\left[ \begin{array}{c} \text{variable} \in \text{set} \rightarrow \text{expression} \\ \end{array} \right]$$

This is the TLA+ syntax for an array-valued expression.
Where this \textit{maps to} symbol is typed \textit{bar dash greater-than} in ASCII.

For example, inside square brackets
The TLA+ syntax for an array expression:

\[
\left[ \text{variable} \in \text{set} \mapsto \text{expression} \right]
\]

\[
\left[ i \mapsto i^2 \right]
\]

This is the TLA+ syntax for an array-valued expression. Where this maps to symbol is typed `bar dash greater-than` in ASCII. For example, inside square brackets we put the variable \( i \).
The TLA+ syntax for an array expression:

\[
\begin{array}{c}
\text{variable} \in \text{set} \mapsto \text{expression} \\
[i \in ]
\end{array}
\]

This is the TLA+ syntax for an array-valued expression.

Where this *maps to* symbol is typed \textit{bar dash greater-than} in ASCII.

For example, inside square brackets,
We put the variable \(i\) \textit{element of}
The TLA+ syntax for an array expression:

\[
\begin{align*}
&\left[ \text{variable} \in \text{set} \mapsto \text{expression} \right] \\
&\left[ i \in 1..42 \right]
\end{align*}
\]

This is the TLA+ syntax for an array-valued expression.

Where this maps to symbol is typed \texttt{bar dash greater-than} in ASCII.

For example, inside square brackets
We put the variable \( i \) element of \texttt{The set of integers from one through 42}
The TLA+ syntax for an array expression:

\[
\begin{array}{c}
\left[ \text{variable} \in \text{set} \mapsto \text{expression} \right] \\
\left[ i \in 1..42 \mapsto \right]
\end{array}
\]

This is the TLA+ syntax for an array-valued expression.

Where this \textit{maps to} symbol is typed \texttt{bar dash greater-than} in ASCII.

For example, inside square brackets
We put the variable \textit{i} element of The set of integers from one through 42
maps to symbol
The TLA+ syntax for an array expression:

\[
\begin{array}{l}
\text{[ } \text{variable} \in \text{set} \mapsto \text{expression} \text{ ]} \\
\text{[ } i \in 1\ldots42 \mapsto i^2 \text{ ]}
\end{array}
\]

This is the TLA+ syntax for an array-valued expression.

Where this \textit{maps to} symbol is typed \texttt{bar dash greater-than} in ASCII.

For example, inside square brackets
We put the variable \textit{i} element of The set of integers from one through 42 maps to symbol \textit{the expression} \textit{i squared}. 

[slide 119]
The TLA⁺ syntax for an array expression:

\[
\begin{align*}
  [ \ i & \in 1 \ldots 42 \ \mapsto \ i^2 \ ]
\end{align*}
\]

So this definition
The TLA+ syntax for an array expression:

\[
\text{sqr} \triangleq [ i \in 1..42 \rightarrow i^2 ]
\]

So this definition
The TLA⁺ syntax for an array expression:

\[ sqr \triangleq [ \ i \in 1\ldots42 \rightarrow i^2 ] \]

Defines \( sqr \) to be an array with index set \( 1\ldots42 \)

So this definition defines \( s-q-r \) to be an array with index set the set of integers from one through 42
The TLA⁺ syntax for an array expression:

\[ \text{sqr} \triangleq [ \ i \in 1..42 \iff \ i^2 ] \]

Defines \text{sqr} to be an array with index set \( 1..42 \) such that \( \text{sqr}[i] = i^2 \) for all \( i \) in \( 1..42 \).

So this definition defines s-q-r to be an array with index set the set of integers from one through 42 such that s-q-r of \( i \) equals \( i \) squared for all \( i \) in that set.
Let’s look at some different terminology
Let’s look at some different terminology used in programming and math for the same things.
Let’s look at some different terminology used in programming and math for the same things.

What programmers call an array
Let’s look at some different terminology used in programming and math for the same things.

What programmers call an array mathematicians call a function.
Let’s look at some different terminology used in programming and math for the same things.

What programmers call an array mathematicians call a function.

What programmers call the index set of an array
Let’s look at some different terminology used in programming and math for the same things.

What programmers call an array mathematicians call a function.

What programmers call the index set of an array mathematicians call the domain of a function.
Programmers use square brackets for array application.

<table>
<thead>
<tr>
<th>Terminology</th>
<th>Programming</th>
<th>Math</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>function</td>
<td></td>
</tr>
<tr>
<td>index set</td>
<td>domain</td>
<td></td>
</tr>
<tr>
<td>$f[e]$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Programmers use square brackets for array application.
Mathematicians use parentheses for function application.

<table>
<thead>
<tr>
<th><strong>Programming</strong></th>
<th><strong>Math</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>function</td>
</tr>
<tr>
<td>index set</td>
<td>domain</td>
</tr>
<tr>
<td>$f[e]$</td>
<td>$f(e)$</td>
</tr>
<tr>
<td>Terminology</td>
<td>Programming</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td>array</td>
<td>function</td>
</tr>
<tr>
<td>index set</td>
<td>domain</td>
</tr>
<tr>
<td>$f[e]$</td>
<td>$f(e)$</td>
</tr>
</tbody>
</table>

Programmers use square brackets for array application. Mathematicians use parentheses for function application.

In TLA+ we write formulas not programs, so we use the mathematical terminology for
<table>
<thead>
<tr>
<th><strong>Programming</strong></th>
<th><strong>Math</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>function</td>
</tr>
<tr>
<td>index set</td>
<td>domain</td>
</tr>
<tr>
<td>$f[e]$</td>
<td>$f(e)$</td>
</tr>
</tbody>
</table>

Programmers use square brackets for array application. Mathematicians use parentheses for function application.

In TLA+ we write formulas not programs, so we use the mathematical terminology for functions.
Programmers use square brackets for array application. Mathematicians use parentheses for function application.

In TLA+ we write formulas not programs, so we use the mathematical terminology for functions and their domains.
### Terminology

<table>
<thead>
<tr>
<th>Programming</th>
<th>Math</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>function</td>
</tr>
<tr>
<td>index set</td>
<td>domain</td>
</tr>
<tr>
<td>$f[e]$</td>
<td>$f(e)$</td>
</tr>
</tbody>
</table>

Programmers use square brackets for array application. Mathematicians use parentheses for function application.

In TLA+ we write formulas not programs, so we use the mathematical terminology for functions and their domains.

However, TLA+ uses square brackets for function application.
Programmers use square brackets for array application. Mathematicians use parentheses for function application.

In TLA+ we write formulas not programs, so we use the mathematical terminology for functions and their domains.

However, TLA+ uses square brackets for function application to avoid confusing it with another way mathematics uses parentheses.
Many popular programming languages allow only index sets \( 0 \ldots n \).

Many popular programming languages allow arrays only whose index sets consist of the set of integers from 0 to some \( n \).
Many popular programming languages allow only index sets $0 \ldots n$.

Math and TLA$^+$ allow a function to have any set as its domain.

Many popular programming languages allow arrays only whose index sets consist of the set of integers from 0 to some $n$.

Math, and therefore TLA$^+$, allows a function to have any set as its domain.
Many popular programming languages allow only index sets $0 \ldots n$.

Math and TLA+ allow a function to have any set as its domain — for example, the set of all integers.

Many popular programming languages allow arrays only whose index sets consist of the set of integers from 0 to some $n$.

Math, and therefore TLA+, allows a function to have any set as its domain.

Even infinite sets, such as the set of all integers.
Let’s return to the spec

\[
\text{CONSTANT } RM
\]

\[
\text{VARIABLE } \text{rmState}
\]

\[
\text{TCTypeOK } \triangleq \text{ rmState } \in [RM \rightarrow \{ "working" , "prepared" , "committed" , "aborted" \}]
\]

\[
\text{TCInit } \triangleq \text{ rmState } = [r \in RM \mapsto "working"]
\]
Let’s return to the spec and jump down to the definition of the next-state formula $TCNext$.
Let’s return to the spec and jump down to the definition of the next-state formula \( TCNext \)

\[
TCNext \triangleq \exists r \in RM : \text{Prepare}(r) \lor \text{Decide}(r)
\]
Let's return to the spec and jump down to the definition of the next-state formula $TC'Next$
Let’s return to the spec and jump down to the definition of the next-state formula $TCNext$

This formula is true if and only if

$$\exists r \in RM : Prepare(r) \lor Decide(r)$$
There exists $r \in RM : \text{Prepare}(r) \lor \text{Decide}(r)$

Let’s return to the spec and jump down to the definition of the next-state formula $TC_{\text{next}}$

This formula is true if and only if

there exists
There exists $r \in RM : \text{Prepare}(r) \lor \text{Decide}(r)$

There exists

Let’s return to the spec and jump down to the definition of the next-state formula $TC'Next$.

This formula is true if and only if

there exists

Typed \texttt{backslash E} in ASCII.

[slide 146]
There exists some $r$ in the set $RM$.

There exists $r \in RM : Prepare(r) \lor Decide(r)$
There exists some $r$ in the set $RM$ for which this subformula is true.

$$\exists r \in RM : \text{Prepare}(r) \lor \text{Decide}(r)$$

There exists some $r$ in the set $RM$ for which this subformula is true.
∃ r ∈ RM : Prepare(r) ∨ Decide(r)

If RM = { “r1”, “r2”, “r3”, “r4” }
If $RM = \{ \text{"r1"}, \text{"r2"}, \text{"r3"}, \text{"r4"} \}$

then this formula equals

$$\exists r \in RM : \text{Prepare}(r) \lor \text{Decide}(r)$$

There exists some $r$ in the set $RM$ for which this subformula is true.

Suppose $RM$ is a set whose elements are the four strings $r1, r2, r3, \text{ and } r4$.

Then this formula equals
There exists some \( r \) in the set \( RM \) for which this subformula is true.

Suppose \( RM \) is a set whose elements are the four strings \( r_1, r_2, r_3, \) and \( r_4 \).

Then this formula equals the disjunction of the four formulas we get by substituting in
\[ \exists r \in RM : \textcolor{red}{\text{Prepare}(r)} \lor \textcolor{red}{\text{Decide}(r)} \]

If \( RM = \{\ "r1", \ "r2", \ "r3", \ "r4" \} \)

then this formula equals

\[ \lor \]
\[ \lor \]
\[ \lor \]
\[ \lor \]

There exists some \( r \) in the set \( RM \) for which this subformula is true.

Suppose \( RM \) is a set whose elements are the four strings \( r1, r2, r3, \) and \( r4 \).

Then this formula equals the disjunction of the four formulas we get by substituting in the subformula each of those four elements of \( RM \)

[slide 152]
There exists some $r$ in the set $RM$ for which this subformula is true.

Suppose $RM$ is a set whose elements are the four strings $r1$, $r2$, $r3$, and $r4$.

Then this formula equals the disjunction of the four formulas we get by substituting in the subformula each of those four elements of $RM$.
\[ \exists r \in RM : \text{Prepare}(r) \lor \text{Decide}(r) \]

If \( RM = \{ "r1", "r2", "r3", "r4" \} \)

then this formula equals

\[ \lor \text{Prepare}("r1") \lor \text{Decide}("r1") \]
\[ \lor \text{Prepare}("r2") \lor \text{Decide}("r2") \]
\[ \lor \]
\[ \lor \]

There exists some \( r \) in the set \( RM \) for which this subformula is true.

Suppose \( RM \) is a set whose elements are the four strings \( r1, r2, r3, \) and \( r4 \).

Then this formula equals the disjunction of the four formulas we get by substituting in the subformula each of those four elements of \( RM \)
\[ \exists r \in RM : \text{Prepare}(r) \lor \text{Decide}(r) \] 

If \( RM = \{ "r1", "r2", "r3", "r4" \} \) 
then this formula equals 

\[ \lor \text{Prepare}("r1") \lor \text{Decide}("r1") \] 
\[ \lor \text{Prepare}("r2") \lor \text{Decide}("r2") \] 
\[ \lor \text{Prepare}("r3") \lor \text{Decide}("r3") \] 
\[ \lor \]

There exists some \( r \) in the set \( RM \) for which this subformula is true.

Suppose \( RM \) is a set whose elements are the four strings \( r1, r2, r3, \) and \( r4 \).

Then this formula equals the disjunction of the four formulas we get by substituting in the subformula each of those four elements of \( RM \)
\[ \exists r \in RM : \boxed{\text{Prepare}(r) \lor \text{Decide}(r)} \]

If \( RM = \{ "r1", "r2", "r3", "r4" \} \)

then this formula equals

\[ \lor \text{Prepare}("r1") \lor \text{Decide}("r1") \]
\[ \lor \text{Prepare}("r2") \lor \text{Decide}("r2") \]
\[ \lor \text{Prepare}("r3") \lor \text{Decide}("r3") \]
\[ \lor \text{Prepare}("r4") \lor \text{Decide}("r4") \]

There exists some \( r \) in the set \( RM \) for which this subformula is true.

Suppose \( RM \) is a set whose elements are the four strings \( r1, r2, r3, \) and \( r4 \).

Then this formula equals the disjunction of the four formulas we get by substituting in the subformula each of those four elements of \( RM \)
\[ \exists r \in RM : Prepare(r) \lor Decide(r) \]

\[ \exists \] declares \( r \) local to formula.

The \textit{exists} declares the identifier \( r \) to be local to this formula. We can replace \( r \) by any other identifier.
∃xyz ∈ RM : Prepare(xyz) ∨ Decide(xyz)

∃ declares r local to formula.

r ← xyz doesn’t change meaning

The exists declares the identifier r to be local to this formula. We can replace r by any other identifier.

For example xyz, without changing the meaning of the formula.
\[ \exists xyz \in RM : \text{Prepare}(xyz) \lor \text{Decide}(xyz) \]

\[ \exists \text{ declares } r \text{ local to formula.} \]

\[ r \leftarrow xyz \text{ doesn't change meaning if } xyz \text{ not declared or defined.} \]

The \textit{exists} declares the identifier \( r \) to be local to this formula. We can replace \( r \) by any other identifier.

For example \( xyz \), without changing the meaning of the formula.

But \( xyz \) must not already be declared or defined at this point in the spec. TLA\(^*\) does not allow defining or declaring a symbol that already has a meaning.
Let’s now return to the spec and move back up

\[
TC\text{Next} \triangleq \exists r \in RM : \text{Prepare}(r) \lor \text{Decide}(r)
\]
Let’s now return to the spec and move back up to the definitions of Prepare and Decide,
$\text{Prepare}(r) \triangleq$

Let’s now return to the spec and move back up to the definitions of $\text{Prepare}$ and $\text{Decide}$, starting with $\text{Prepare}$
Let's now return to the spec and move back up to the definitions of \( \text{Prepare} \) and \( \text{Decide} \), starting with \( \text{Prepare} \).

Recall the state/transition graph of a resource manager.
Let's now return to the spec and move back up to the definitions of Prepare and Decide, starting with Prepare.

Recall the state/transition graph of a resource manager.

*Prepare* of $r$ describes the *working* to *prepared* step of resource manager $r$. 

[slide 164]
This step can be taken only when the current state of resource manager $r$ is \textit{working}, so $\text{rmState}$ of $r$ equal to the string \textit{working} must be true.
\[ \text{Prepare}(r) \triangleq \land \text{rmState}[r] = \text{"working"} \]

This step can be taken only when the current state of resource manager \( r \) is \textit{working}, so \textit{rmState} of \( r \) equal to the string \textit{working} must be true.

The step must change the value of \textit{rmState} of \( r \) to the string \textit{prepared}.
This step can be taken only when the current state of resource manager $r$ is \textit{working}, so $rmState$ of $r$ equal to the string \textit{working} must be true.

The step must change the value of $rmState$ of $r$ to the string \textit{prepared}.

Most people think that condition is expressed like this.
This step can be taken only when the current state of resource manager \( r \) is \textit{working}, so \( \text{rmState} \) of \( r \) equal to the string \textit{working} must be true.

The step must change the value of \( \text{rmState} \) of \( r \) to the string \textit{prepared}.

Most people think that condition is expressed like this.

Stop the video and figure out why this is wrong.
The value of \( rmState'[r] \) in the new state is "prepared".

You have to learn to see what a formula says, not what you think it should say.
The value of `rmState'[r] in the new state is "prepared".

You have to learn to see what a formula says, not what you think it should say.

What does this formula actually say?
The value of \( rmState'[r] \) in the new state is "prepared".
\( rmState'[r] = \text{"prepared"} \)

What does this formula say?

The value of \( rmState[r] \) in the new state is \text{"prepared"}.

What does it say about the value of \( rmState[s] \) in the new state for an RM \( s \) with \( s \neq r \)?

What does it say about the value of \( rmState \) of \( s \) in the new state for a resource manager \( s \) different from \( r \)?
$\text{rmState}'[r] = \text{“prepared”}$

What does this formula say?

The value of $\text{rmState}[r]$ in the new state is “prepared”.

What does it say about the value of $\text{rmState}[s]$ in the new state for an RM $s$ with $s \neq r$?

**Nothing!**

What does it say about the value of $\text{rmState}$ of $s$ in the new state for a resource manager $s$ different from $r$?

Absolutely nothing!
\( \text{rmState}'[r] = \text{"prepared"} \)

The spec can’t just say what the new value of \( \text{rmState} \) of \( r \) is.
The spec can’t just say what the new value of \( rmState \) of \( r \) is. It must say what the new value of the entire function \( rmState \) is. That value must be a function with domain \( RM \). And we know how to write such a function.
\[ rmState'[r] = \text{"prepared"} \]

\[ rmState' = [ s \in RM \mapsto \cdots ] \]

It looks like this, where we have to replace the dot dot dot dot
\( rmState'[r] = \text{"prepared"} \)

\[ rmState' = [ s \in RM \mapsto \cdots ] \]

↑

the new value of \( rmState[s] \)

It looks like this, where we have to replace the dot dot dot with an expression that specifies the new value of \( rmState \) of \( s \) for each resource manager \( s \).
If $s$ is resource manager $r$, then the value of $rmState$ of $s$ in the new state should be the string *prepared*.
If $s$ is resource manager $r$, then the value of $rmState$ of $s$ in the new state should be the string "prepared"

Any other resource manager $s$ should have the same value of $rmState$ in the new state as in the old state.
\[ \text{If } s \text{ is resource manager } r, \text{ then the value of } rmState \text{ of } s \text{ in the new state should be the string } \text{prepared} \]

Any other resource manager \( s \) should have the same value of \( rmState \) in the new state as in the old state.

This is correct, but it’s too long-winded.
\[
[s \in RM \mapsto \begin{cases}
\text{IF } s = r \text{ THEN } \text{"prepared"}
\vspace{0.5em}
\text{ELSE } \text{rmState}[s]
\end{cases}]
\]

If \(s\) is resource manager \(r\), then the value of \(rmState\) of \(s\) in the new state should be the string \textit{prepared}.

Any other resource manager \(s\) should have the same value of \(rmState\) in the new state as in the old state.

This is correct, but it’s too long-winded.

\textit{We need a shorter way to write this expression.}
\[ s \in RM \mapsto \text{IF } s = r \text{ THEN } \text{“prepared”} \text{ ELSE } rmState[s] \]

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

TLA+ provides this \text{EXCEPT} construct. Everyone hates it.
[s ∈ RM → IF s = r THEN “prepared” ELSE rmState[s]]

[rmState EXCEPT ![r] = “prepared”]

TLA+ provides this **EXCEPT** construct.
Everyone hates it.

What does the exclamation point (usually read as bang) mean? It means nothing.
\[ s \in R M \mapsto \text{IF } s = r \text{ THEN } \text{"prepared" ELSE } \text{rmState}[s] ]

\[ \text{rmState EXCEPT } ![r] = \text{"prepared"} ]

meaningless syntax

TLA+ provides this \text{EXCEPT} construct. Everyone hates it.

What does the exclamation point (usually read as bang) mean? It means nothing.

It’s just syntax.

[slide 184]
\[ s \in R M \mapsto \text{IF } s = r \text{ THEN "prepared" ELSE } \text{rmState}[s] \]

\[ \text{rmState EXCEPT ![r] = "prepared"}] \]

You’ll get used to it.

TLA+ provides this EXCEPT construct. Everyone hates it.

What does the exclamation point (usually read as bang) mean? It means nothing.

It’s just syntax. But you’ll get used to it.
\[ \text{Prepare}(r) \triangleq \land \text{rmState}[r] = \text{"working"} \]
\[ \land \text{rmState}' = [\text{rmState} \ \text{EXCEPT} \ ![r] = \text{"prepared"}] \]

So, here’s the complete definition of \textit{Prepare}.
Now for the definition of $Decide$. It describes possible steps in which resource manager $r$ reaches a committed or aborted state.
Now for the definition of $Decide$. It describes possible steps in which resource manager $r$ reaches a committed or aborted state.

It’s the disjunction of two formulas.
$\text{Decide}(r) \triangleq \bigvee \text{Describes a } \text{prepared} \rightarrow \text{committed} \text{ step.}$

Now for the definition of $\text{Decide}$. It describes possible steps in which resource manager $r$ reaches a $\text{committed}$ or $\text{aborted}$ state.

It’s the disjunction of two formulas.

The first describes a step in which resource manager $r$ goes from the $\text{prepared}$ state to the $\text{committed}$ state.
$\text{Decide}(r) \triangleq \bigvee \land rm\text{State}[r] = \text{“prepared”}$

Such a step can occur only if $r$ is in the \textit{prepared} state.
Such a step can occur only if \( r \) is in the \textit{prepared} state.

\( r \) can commit only if every resource manager is in the \textit{prepared} or \textit{committed} state.
$Decide(r) \equiv \bigvee \land rmState[r] = \text{“prepared”} \\
\land canCommit$

Such a step can occur only if $r$ is in the \textit{prepared} state.

$r$ can commit only if every resource manager is in the \textit{prepared} or \textit{committed} state.

This condition is written in a formula named \textit{canCommit}, whose definition we’ll look at later.

[slide 192]
And in the new state, \( r \) is \textit{committed} and the state of every other resource manager remains the same.
\[
\text{Decide}(r) \equiv \bigvee \land \text{rmState}[r] = \text{"prepared"} \\
\land \text{canCommit} \\
\land \text{rmState}' = [\text{rmState EXCEPT !}[r] = \text{"committed"}] \\
\bigvee
\]

And in the new state, \( r \) is \textit{committed} and the state of every other resource manager remains the same.

This is expressed with our friend \textit{EXCEPT}.
\[ Decide(r) \triangleq \bigvee \land rmState[r] = \text{"prepared"} \]
\[ \land canCommit \]
\[ \land rmState' = [rmState \ EXCEPT \ ![r] = \text{"committed"}] \]
\[ \bigvee \text{ Describes steps that abort.} \]

And in the new state, \( r \) is committed and the state of every other resource manager remains the same.

This is expressed with our friend \( \text{EXCEPT} \).

The second disjunction describes possible transitions to the aborted state.
\[ \text{Decide}(r) \overset{\triangle}{=} \bigvee \land \text{rmState}[r] = \text{"prepared"} \\]
\[ \land \text{canCommit} \\]
\[ \land \text{rmState}' = [\text{rmState} \text{ EXCEPT } ![r] = \text{"committed"}] \\]
\[ \lor \land \text{rmState}[r] \in \{\text{"working"}, \text{"prepared"}\} \]

\(r\) can abort from the \textit{working} or \textit{prepared} state, so \text{rmState} of \(r\) must be an element of the set consisting of the two strings \textit{working} and \textit{prepared}.
\[ \text{Decide}(r) \triangleq \lor \land \text{rmState}[r] = \text{“prepared”} \]
\[ \land \text{canCommit} \]
\[ \land \text{rmState}' = [\text{rmState EXCEPT ![} r ] \land \text{“committed”}] \]
\[ \lor \land \text{rmState}[r] \in \{ \text{“working”, “prepared”} \} \]
\[ \land \text{no RM is committed} \]

\( r \) can abort from the \textit{working} or \textit{prepared} state, so \text{rmState} of \( r \) must be an element of the set consisting of the two strings \textit{working} and \textit{prepared}.

\( r \) can abort only when no other resource manager is committed.
\[ \text{Decide}(r) \triangleq \bigvee \land \text{rmState}[r] = \text{“prepared”} \]
\[ \land \text{canCommit} \]
\[ \land \text{rmState}' = [\text{rmState}\ \text{EXCEPT} \ ! [r] = \text{“committed”}] \]
\[ \bigvee \land \text{rmState}[r] \in \{ \text{“working”, “prepared”} \} \]
\[ \land \text{notCommitted} \]

\(r\) can abort from the \textit{working} or \textit{prepared} state, so \text{rmState} of \(r\) must be an element of the set consisting of the two strings \textit{working} and \textit{prepared}.

\(r\) can abort only when no other resource manager is committed.

This condition is written as formula \textit{notCommitted}, whose definition we’ll look at later.

[slide 198]
And the state of \( r \) changes to \textit{aborted}, while the state of all other resource managers remain the same.
\[ Decide(r) \triangleq \begin{align*} &\lor \land rmState[r] = \text{“prepared”} \\ &\land \text{canCommit} \\ &\land rmState' = [rmState \ except \ ![r] = \text{“committed”}] \\ &\lor \land rmState[r] \in \{\text{“working”, “prepared”}\} \\ &\land \text{notCommitted} \\ &\land rmState' = [rmState \ except \ ![r] = \text{“aborted”}] \end{align*} \]

And the state of \( r \) changes to \textit{aborted}, while the state of all other resource managers remain the same.

We now look at the definitions of \textit{canCommit} and \textit{notCommitted}, but first a digression.
\[ \exists r \in RM : \text{Prepare}(r) \lor \text{Decide}(r) \]

Remember that this formula asserts:
\[ \exists r \in RM : \text{Prepare}(r) \lor \text{Decide}(r) \]

There exists \( r \) in \( RM \) for which this subformula is true.

Remember that this formula asserts:
there exists some \( r \) in the set \( RM \) for which this subformula is true.
\[ \exists r \in RM : Prepare(r) \lor Decide(r) \]

If \( RM = \{ "r1", "r2", "r3", "r4" \} \)

Remember that this formula asserts:
there exists some \( r \) in the set \( RM \) for which this subformula is true.

If \( RM \) is this set of four elements,
\[ \exists r \in RM : Prepare(r) \lor Decide(r) \]

If \( RM = \{ "r1", "r2", "r3", "r4" \} \)
then the formula equals
\[ \lor Prepare("r1") \lor Decide("r1") \]
\[ \lor Prepare("r2") \lor Decide("r2") \]
\[ \lor Prepare("r3") \lor Decide("r3") \]
\[ \lor Prepare("r4") \lor Decide("r4") \]

Remember that this formula asserts:
there exists some \( r \) in the set \( RM \) for which this subformula is true.

If \( RM \) is this set of four elements, then the \textit{exists} formula equals this disjunction of four formulas.
$\exists r \in RM : \text{Prepare}(r) \lor \text{Decide}(r)$

Remember that this formula asserts: there exists some $r$ in the set $RM$ for which this subformula is true.

If $RM$ is this set of four elements, then the $exists$ formula equals this disjunction of four formulas.

There is a dual to this formula in which the $exists$ symbol is replaced by
Remember that this formula asserts:
there exists some $r$ in the set $RM$ for which this subformula is true.

If $RM$ is this set of four elements, then the exists formula equals this disjunction of four formulas.

There is a dual to this formula in which the exists symbol is replaced by this forall symbol.
\( \forall r \in RM : \text{Prepare}(r) \lor \text{Decide}(r) \)
This formula asserts that:

\[ \forall r \in RM : \text{Prepare}(r) \lor \text{Decide}(r) \]
\[ \forall r \in RM : \text{Prepare}(r) \lor \text{Decide}(r) \]

**For all** \( r \) in \( RM \), this subformula is true.

Typed \textit{backslash} \textit{A} in ASCII.

This formula asserts that:

**for all** \( r \) in the set \( RM \), this subformula is true.
\forall r \in RM : \text{Prepare}(r) \lor \text{Decide}(r)

If \( RM = \{ "r1", "r2", "r3", "r4" \} \)
then the formula equals

\[ \wedge \text{Prepare("r1")} \lor \text{Decide("r1")} \]
\[ \wedge \text{Prepare("r2")} \lor \text{Decide("r2")} \]
\[ \wedge \text{Prepare("r3")} \lor \text{Decide("r3")} \]
\[ \wedge \text{Prepare("r4")} \lor \text{Decide("r4")} \]

Typed backslash \( A \) in ASCII.

This formula asserts that:
for all \( r \) in the set \( RM \), this subformula is true.

If \( RM \) is this set of four elements,
then the \textit{forall} formula equals this \textit{conjunction} of four formulas.

Now to the definitions of \textit{canCommit} and \textit{notCommitted}.
Remember that $canCommit$ should assert that
canCommit \triangleq \textit{every RM is prepared or committed}

Remember that canCommit should assert that every resource manager is in the prepared or committed state.
Remember that $canCommit$ should assert that every resource manager is in the \textit{prepared} or \textit{committed} state.

This formula asserts that for every resource manager $r$, the value of $rmState$ of $r$ is either the string \textit{prepared} or the string \textit{committed}.
$notCommitted \overset{\Delta}{=} \text{no RM is committed}$

Remember that $notCommitted$ should assert that no resource manager is committed.
Remember that \textit{notCommitted} should assert that no resource manager is committed.

This formula asserts that, for every resource manager \( r \), the value of \( rmState[r] \) doesn’t equal the string \textit{committed}. 

\[
\text{notCommitted} \triangleq \forall r \in RM : rmState[r] \neq \text{“committed”}
\]
Let’s take another look at the definition of $\text{Decide}$. 

\[
\text{Decide}(r) \triangleq \bigvee \left( \bigwedge rmState[r] = \text{"prepared"} \right) \\
\bigwedge \text{canCommit} \\
\bigwedge rmState' = [rmState \text{ EXCEPT } ![r] = \text{"committed"}] \\
\bigvee \left( \bigwedge rmState[r] \in \{ \text{"working"}, \text{"prepared"} \} \\
\bigwedge \text{notCommitted} \\
\bigwedge rmState' = [rmState \text{ EXCEPT } ![r] = \text{"aborted"}] \right)
\]
Let's take another look at the definition of \textit{Decide}.

Replacing \textit{canCommit} by its definition doesn't change the meaning of \textit{Decide} of \( r \).

\[
\text{Decide}(r) \ \overset{\triangleq}{=} \ \lor \ \land \ rmState[r] = \text{“prepared”} \\
\land \text{canCommit} \\
\land \ rmState' = [\text{rmState EXCEPT } ![r] = \text{“committed”}] \\
\lor \ \land \ rmState[r] \in \{\text{“working”, “prepared”}\} \\
\land \text{notCommitted} \\
\land \ rmState' = [\text{rmState EXCEPT } ![r] = \text{“aborted”}] 
\]
Let’s take another look at the definition of \textit{Decide}.

Replacing \textit{canCommit} by its definition doesn’t change the meaning of \textit{Decide} of \( r \).

Here’s the definition of \textit{canCommit} again.
Let’s take another look at the definition of $\text{Decide}$. 

Replacing $\text{canCommit}$ by its definition doesn’t change the meaning of $\text{Decide}$ of $r$.

Here’s the definition of $\text{canCommit}$ again.

Replacing $\text{canCommit}$ by its definition yields this formula.

\[
\text{Decide}(r) \overset{\Delta}{=} \bigvee \bigwedge \text{rmState}[r] = \text{"prepared"}
\land \forall s \in R M : \text{rmState}[s] \in \{\text{"prepared"}, \text{"committed"}\}
\land \text{rmState}' = [\text{rmState EXCEPT ![} r \text{]} = \text{"committed"}]
\land \text{rmState}[r] \in \{\text{"working"}, \text{"prepared"}\}
\land \text{notCommitted}
\land \text{rmState}' = [\text{rmState EXCEPT ![} r \text{]} = \text{"aborted"}]
\]

\[
\text{canCommit} \overset{\Delta}{=} \forall r \in R M : \text{rmState}[r] \in \{\text{"prepared"}, \text{"committed"}\}
\]
We have to change the bound variable $r$ used in the definition of \textit{canCommit}.
We have to change the bound variable $r$ used in the definition of $canCommit$ to some other variable like $s$.
We have to change the bound variable \( r \) used in the definition of \( \text{canCommit} \) to some other variable like \( s \) to avoid a name conflict with this \( r \).
We have to change the bound variable \( r \) used in the definition of \( \text{canCommit} \) to some other variable like \( s \) to avoid a name conflict with this \( r \).

Similarly, we can replace \( \text{notCommitted} \).
We have to change the bound variable \( r \) used in the definition of \( \text{canCommit} \) to some other variable like \( s \) to avoid a name conflict with this \( r \).

Similarly, we can replace \( \text{notCommitted} \) with its definition.
Definitions provide a simple and powerful way of hierarchically decomposing formulas to make them easier to read.

\[ Decide(r) \triangleq \bigvee \bigwedge \text{rmState}[r] = \text{"prepared"} \]
\[ \land \forall s \in RM : \text{rmState}[s] \in \{ \text{"prepared"}, \text{"committed"} \} \]
\[ \land \text{rmState}' = [\text{rmState } \text{EXCEPT } ![r] = \text{"committed"}] \]
\[ \land \forall s \in RM : \text{rmState}[s] \neq \text{"committed"} \]
\[ \land \text{rmState}' = [\text{rmState } \text{EXCEPT } ![r] = \text{"aborted"}] \]

Definitions provide a simple and powerful way of hierarchically decomposing formulas to make them easier to read.
\[ \text{Decide}(r) \triangleq \forall \land \text{rmState}[r] = \text{"prepared"} \land \forall s \in RM : \text{rmState}[s] \in \{ \text{"prepared"}, \text{"committed"} \} \land \text{rmState}' = [\text{rmState} \ \text{EXCEPT} \ ![r] = \text{"committed"}] \land \forall s \in RM : \text{rmState}[s] \neq \text{"committed"} \land \text{rmState}' = [\text{rmState} \ \text{EXCEPT} \ ![r] = \text{"aborted"}] \]

\text{Decide}(r) \text{ depends on } \]

Whether a \text{Decide} of \( r \) step is possible and what it can do depends on
\[
\text{Decide}(r) \triangleq \vee \wedge \text{rmState} \[ r \] = \text{“prepared”}
\wedge \forall s \in \text{RM} : \text{rmState} \[ s \] \in \{ \text{“prepared”}, \text{“committed”} \}
\wedge \text{rmState'} = [\text{rmState} \text{ EXCEPT } ![r] = \text{“committed”}]
\vee \wedge \text{rmState} \[ r \] \in \{ \text{“working”}, \text{“prepared”} \}
\wedge \forall s \in \text{RM} : \text{rmState} \[ s \] \neq \text{“committed”}
\wedge \text{rmState'} = [\text{rmState} \text{ EXCEPT } ![r] = \text{“aborted”}]
\]

\text{Decide}(r) \text{ depends on the states of all the resource managers.}

Whether a \text{Decide} of \( r \) step is possible and what it can do depends on the states of all the resource managers.
\[
\text{Decide}(r) \triangleq \bigvee \land \text{rmState}[r] = \text{"prepared"}
\land \forall s \in \text{RM} : \text{rmState}[s] \in \{ \text{"prepared"}, \text{"committed"} \}
\land \text{rmState}' = [\text{rmState} \ \text{EXCEPT} \ ![r] = \text{"committed"}]
\land \forall s \in \text{RM} : \text{rmState}[s] \neq \text{"committed"}
\land \text{rmState}' = [\text{rmState} \ \text{EXCEPT} \ ![r] = \text{"aborted"}]
\]

\text{Decide}(r) \text{ depends on the states of all the resource managers.}

How can this be implemented?

Whether a \textit{Decide} of \textit{r} step is possible and what it can do depends on the states of all the resource managers.

How can this be implemented?
Decide\((r)\) depends on the states of all the resource managers.

How can this be implemented?

What programming language allows a single step to examine the states of a whole set of processes?
\[ \text{Decide}(r) \overset{\Delta}{=} \bigvee \bigwedge r\text{mState}[r] = \text{"prepared"} \\
\bigwedge \forall s \in \text{RM} : r\text{mState}[s] \in \{\text{"prepared"}, \text{"committed"} \} \\
\bigwedge r\text{mState}' = [r\text{mState} \text{ EXCEPT } ![r] = \text{"committed"}] \\
\bigvee \bigwedge r\text{mState}[r] \in \{\text{"working"}, \text{"prepared"} \} \\
\bigwedge \forall s \in \text{RM} : r\text{mState}[s] \neq \text{"committed"} \\
\bigwedge r\text{mState}' = [r\text{mState} \text{ EXCEPT } ![r] = \text{"aborted"}] \]

\text{Decide}(r) \text{ depends on the states of all the resource managers.}

We don’t care.

Whether a \text{Decide} \text{ of } r \text{ step is possible and what it can do depends on the states of all the resource managers.}

How can this be implemented?

What programming language allows a single step to examine the states of a whole set of processes?

\text{We don’t care.}
\[ Decide(r) \triangleq \bigvee \land \text{rmState}[r] = \text{“prepared”} \]
\[ \land \forall s \in RM : \text{rmState}[s] \in \{\text{“prepared”}, \text{“committed”}\} \]
\[ \land \text{rmState}' = [\text{rmState} \setminus ! [r] = \text{“committed”}] \]
\[ \bigvee \land \text{rmState}[r] \in \{\text{“working”}, \text{“prepared”}\} \]
\[ \land \forall s \in RM : \text{rmState}[s] \neq \text{“committed”} \]
\[ \land \text{rmState}' = [\text{rmState} \setminus ! [r] = \text{“aborted”}] \]

*Decide*(\(r\)) depends on the states of all the resource managers.

We don’t care.

We’re writing a spec of **what** transaction commit should do,
\[ \text{Decide}(r) \triangleq \bigvee r mState[r] = \text{“prepared”} \]
\[ \land \forall s \in RM : r mState[s] \in \{ \text{“prepared”}, \text{“committed”} \} \]
\[ \land r mState' = [r mState \ \text{EXCEPT} \ ! [r] = \text{“committed”}] \]
\[ \land \land [r mState[r] \in \{ \text{“working”}, \text{“prepared”} \} ] \land [r mState[s] \neq \text{“committed”}] \land r mState' = [r mState \ \text{EXCEPT} \ ! [r] = \text{“aborted”}] \]

\[ \text{Decide}(r) \] depends on the states of all the resource managers.

We don’t care.

We’re writing a spec of what transaction commit should do, not how it’s implemented.

We’re writing a spec of what transaction commit should accomplish, not how it’s implemented.

The next video describes a protocol for implementing it.
Let’s take one more look at the original definition of *Decide*.

\[
Decide(r) \triangleq \lor \land \text{rmState}[r] = \text{“prepared”} \\
\land \text{canCommit} \\
\land \text{rmState’} = [\text{rmState} \ \text{EXCEPT} \ ![r] = \text{“committed”}] \\
\lor \land \text{rmState}[r] \in \{ \text{“working”, “prepared”} \} \\
\land \text{notCommitted} \\
\land \text{rmState’} = [\text{rmState} \ \text{EXCEPT} \ ![r] = \text{“aborted”}] 
\]
Let's take one more look at the original definition of $Decide$.

$Decide$ of $r$ is defined to be a disjunction of two formulas.
Let’s take one more look at the original definition of $Decide$.

$Decide$ of $r$ is defined to be a disjunction of two formulas.

We could give a different name to each of these formulas, say $DecideC$ of $r$ and $DecideA$ of $r$. 

[slide 235]
And in the definition of $TC_{Next}$

$$\begin{align*}
DecideC(r) & \triangleq \land rmState[r] = \text{"prepared"} \\
& \land canCommit \\
& \land rmState' = [rmState \ \text{EXCEPT} \ ![r] = \text{"committed"}] \\
DecideA(r) & \triangleq \land rmState[r] \in \{ \text{"working"}, \text{"prepared"} \} \\
& \land notCommitted \\
& \land rmState' = [rmState \ \text{EXCEPT} \ ![r] = \text{"aborted"}] \\
TC_{Next} & \triangleq \exists r \in RM : Prepare(r) \lor Decide(r)
\end{align*}$$
Define \( DecideC(r) \) as:

\[
DecideC(r) \triangleq \land \land \land \land \land rmState[r] = \text{"prepared"} \\
\land canCommit \\
\land rmState' = [rmState \ EXCEPT ![r] = \text{"committed"}] \\
\land DecideC(r)
\]

Define \( DecideA(r) \) as:

\[
DecideA(r) \triangleq \land \land \land \land \land rmState[r] \in \{ \text{"working"}, \text{"prepared"} \} \\
\land notCommitted \\
\land rmState' = [rmState \ EXCEPT ![r] = \text{"aborted"}] \\
\land DecideA(r)
\]

\( TCNext \triangleq \exists r \in RM : Prepare(r) \lor Decide(r) \)

And in the definition of \( TCNext \) replace \( Decide \) of \( r \)
\[ \text{Decide}_C(r) \equiv \land \text{rmState}[r] = \text{“prepared”} \]
\[ \land \text{canCommit} \]
\[ \land \text{rmState}' = [\text{rmState EXCEPT !}[r] = \text{“committed”}] \]
\[ \text{Decide}_A(r) \equiv \land \text{rmState}[r] \in \{ \text{“working”, “prepared”} \} \]
\[ \land \text{notCommitted} \]
\[ \land \text{rmState}' = [\text{rmState EXCEPT !}[r] = \text{“aborted”}] \]

\[ \text{TCNext} \equiv \exists r \in \text{RM} : \text{Prepare}(r) \lor \text{Decide}_C(r) \lor \text{Decide}_A(r) \]

And in the definition of \text{TCNext} replace \text{Decide of } r by the disjunction of \text{Decide}_C of } r and \text{Decide}_A of } r
There are many ways to decompose a next-state formula into subformulas.

\[ \text{DecideC}(r) \triangleq \wedge \text{rmState}[r] = \text{"prepared"} \]
\[ \quad \wedge \text{canCommit} \]
\[ \quad \wedge \text{rmState}' = [\text{rmState EXCEPT } ![r] = \text{"committed"}] \]
\[ \text{DecideA}(r) \triangleq \wedge \text{rmState}[r] \in \{\text{"working"}, \text{"prepared"}\} \]
\[ \quad \wedge \text{notCommitted} \]
\[ \quad \wedge \text{rmState}' = [\text{rmState EXCEPT } ![r] = \text{"aborted"}] \]
\[ \text{TCNext} \triangleq \exists r \in \text{RM} : \text{Prepare}(r) \lor \text{DecideC}(r) \lor \text{DecideA}(r) \]

And in the definition of TCNext replace Decide of r by the disjunction of DecideC of r and DecideA of r.

There are lots of different ways to decompose a next-state formula into subformulas.
CHECKING THE SPEC
In the Toolbox, create a new model for the \textit{TCommit} spec.
In the Toolbox, create a new model for the $TCommit$ spec.

The Toolbox reports 3 errors.

[Image of a software interface showing 3 errors detected]

In the Toolbox, create a new model for the $TCommit$ spec.

The Toolbox reports that it found three errors in the model.
In the Toolbox, create a new model for the \textit{TCommit} spec.

The Toolbox reports 3 errors.

Click here

[slide 243]
In the Toolbox, create a new model for the \textit{TCommit} spec.

The Toolbox reports 3 errors.

Click here for a list of errors.

In the Toolbox, create a new model for the \textit{TCommit} spec.

The Toolbox reports that it found three errors in the model.

Clicking here, raises this report.

[slide 244]
In the Toolbox, create a new model for the $TCommit$ spec.

The Toolbox reports 3 errors.

In the Toolbox, create a new model for the $TCommit$ spec.

The Toolbox reports that it found three errors in the model.

Clicking here, raises this report.

These two errors occur . . .

[slide 245]
They're indicated by these little red Xs. Since we didn't use the default names `Init` and `Next` for the initial-state and next-state formulas, you have to enter those names. Enter them now.

[slide 246]
They’re indicated by these little red Xs.
here.

They’re indicated by these little red Xs.

Since we didn’t use the default names \textit{Init} and \textit{Next} for the initial-state and next-state formulas, you have to enter those names.
here.

They’re indicated by these little red Xs.

Since we didn’t use the default names $Init$ and $Next$ for the initial-state and next-state formulas, you have to enter those names .

Enter them now.

[slide 249]
This error tells us that the model has to provide a value for the declared constant $RM$. 

[slide 250]
This error tells us that the model has to provide a value for the declared constant $RM$. 
Go to the *What is the model?* area.
Go to the *What is the model?* area.

And double-click on $RM$. 

[slide 253]
We now tell the Toolbox what value the model should assign to $RM$. 
We now tell the Toolbox what value the model should assign to $RM$.

Make sure *Ordinary assignment* is selected.
We now tell the Toolbox what value the model should assign to $RM$.

Make sure *Ordinary assignment* is selected.

And enter the value here.
You should usually start with the smallest possible model, which in this case means letting the set $RM$ have only a single element.

But this spec is so simple, let’s make it a set of three elements. The actual elements don’t matter.
You should usually start with the smallest possible model, which in this case means letting the set $RM$ have only a single element.

But this spec is so simple, let’s make it a set of three elements. The actual elements don’t matter.

We could let it be a set of 3 integers.
But I prefer to use strings, such as r1, r2, and r3.
But I prefer to use strings, such as r1, r2, and r3.

Type this value and click *Finish*
We first check that the spec is type correct
We first check that the spec is type correct by checking that $TCTypeOK$ is an invariant.
We first check that the spec is type correct by checking that $TCTypeOK$ is an invariant.

Add the invariant $TCTypeOK$ to the model.
A behavior satisfying the spec should terminate when all RMs have committed or aborted.

A behavior satisfying the spec should terminate when all resource managers have committed or aborted.
A behavior satisfying the spec should terminate when all RMs have committed or aborted.

As in _SimpleProgram_, we have to tell TLC not to check for deadlock.
A behavior satisfying the spec should terminate when all RMs have committed or aborted.

As in *Simple Program*, we have to tell TLC not to check for deadlock.

So, uncheck this box
A behavior satisfying the spec should terminate when all RMs have committed or aborted.

As in *SimpleProgram*, we have to tell TLC not to check for deadlock.

So, uncheck this box and click on the green arrow to run TLC.
TLC should find no errors.
Always be suspicious of success.
Be Suspicious of Success

Always be suspicious of success.

Check the statistics of the TLC run.
Be Suspicious of Success

Always be suspicious of success.

Check the statistics of the TLC run.

Did TLC find a reasonable number of states that can be reached by behaviors?
Always be suspicious of success.

Check the statistics of the TLC run.

Did TLC find a reasonable number of states that can be reached by behaviors?

The coverage section reports how many times different subactions of the next-state formula were used to generate new states.
You can double click on a line to see what subaction it refers to.
Be Suspicious of Success

You can double click on a line to see what subaction it refers to.

A count of zero means that the subaction wasn’t used, which usually means there’s an error in the spec.
Check the invariance of conditions that should be invariant.

You should check the invariance of conditions that should be invariant.
Check the invariance of conditions that should be invariant.

Such a condition for $TCommit$ is:

You should check the invariance of conditions that should be invariant.

One such condition for the $TCommit$ spec is the following.
Check the invariance of conditions that should be invariant.

Such a condition for $TCommit$ is:

```
It’s impossible for one RM to have aborted and another RM to have committed.
```

You should check the invariance of conditions that should be invariant.

One such condition for the $TCommit$ spec is the following.

```
It’s impossible for one resource manager to have aborted and another resource manager to have committed.
```
Check the invariance of conditions that should be invariant.

Such a condition for $T_{Commit}$ is:

It’s impossible for one RM to have aborted and another RM to have committed.

Expressed by formula $T_{CC}Consistent$. 

You should check the invariance of conditions that should be invariant.

One such condition for the $T_{Commit}$ spec is the following.

It’s impossible for one resource manager to have aborted and another resource manager to have committed.

This condition is expressed by formula $T_{CC}Consistent$ that’s defined in the module as follows. 

[slide 278]
Add the invariant $TCConsistent \triangleq$.

Run TLC on the model. TLC should find no error.

For all $r_1$ and $r_2$ in $RM$ it is the case that:

[slide 279]
\[ TC\text{Consistent} \overset{\Delta}{=} \forall r_1, r_2 \in RM : \]

For all \( r_1 \) and \( r_2 \) in \( RM \) it is the case that:
$TC_{\text{Consistent}} \triangleq$
\[ \forall r1, r2 \in RM : \]

An abbreviation for $\forall r1 \in RM : \forall r2 \in RM :$

For all $r1$ and $r2$ in $RM$ it is the case that:

This is an abbreviation for:

For all $r1$ in $RM$ it’s the case that for all $r2$ in $RM$ it’s the case that:

[slide 281]
\( TCC_{\text{Consistent}} \triangleq \forall r_1, r_2 \in RM : \neg \)

For all \( r_1 \) and \( r_2 \) in \( RM \) it is the case that:

This is an abbreviation for:
For all \( r_1 \) in \( RM \) it’s the case that for all \( r_2 \) in \( RM \) it’s the case that:

It is not true that

[slide 282]
\[ \text{TCConsistent} \triangleq \forall r_1, r_2 \in RM : \neg \]

written ! in C

This negation operator is written as exclamation point in C
This negation operator is written as exclamation point in C

In TLA+ its written as tilde.
\[ TC\text{Consistent} \triangleq \]
\[
\forall r_1, r_2 \in RM : \neg
\]

This negation operator is written as exclamation point in C.

In TLA+ its written as tilde.

So, \emph{TCConsistent} asserts that for all \( r_1 \) and \( r_2 \) in \( RM \) it’s not true that
\[ TC\text{Consistent} \overset{\triangle}{=} \]
\[ \forall r_1, r_2 \in RM : \neg \wedge \text{rmState}[r_1] = \text{"aborted"} \]
\[ \wedge \text{rmState}[r_2] = \text{"committed"} \]

This negation operator is written as exclamation point in C
In TLA+ its written as tilde.
So, \( TC\text{Consistent} \) asserts that for all \( r_1 \) and \( r_2 \) in \( RM \) it’s not true that \( \text{rmState} \) of \( r_1 \) equals aborted and \( \text{rmState} \) of \( r_2 \) equals committed.
Add the invariant \( TCConsistent \).

\[
TCConsistent \triangleq \\
\forall r_1, r_2 \in RM : \neg \land \text{rmState}[r_1] = \text{"aborted"} \\
\land \text{rmState}[r_2] = \text{"committed"}
\]

Add the invariant \( TCConsistent \) to the model.
Add the invariant \( TCConsistent \).

Run TLC on the model.

\[
TCConsistent \quad \overset{\Delta}{=} \\
\forall r_1, r_2 \in RM : \neg \land rmState[r_1] = \text{“aborted”} \\
\land rmState[r_2] = \text{“committed”}
\]

Add the invariant \( TCConsistent \) to the model.

And run TLC on the model.
\[ TCConsistent \overset{\triangle}{=} \]
\[ \forall r_1, r_2 \in RM : \neg \land rmState[r_1] = \text{"aborted"} \land rmState[r_2] = \text{"committed"} \]

Add the invariant \( TCConsistent \).

Run TLC on the model.

TLC should find no error.

Add the invariant \( TCConsistent \) to the model.

And run TLC on the model.

TLC should find no error.
A PARSING NOTE
The scope of $\forall$ and $\exists$ extends as far as possible.

The scope of *forall* and *exists* extends as far as possible.
The scope of $\forall$ and $\exists$ extends as far as possible.

The expression

\[ \forall x \in S : \ldots \]

The scope of \textit{forall} and \textit{exists} extends as far as possible.

For example, this expression

[slide 292]
The scope of $\forall$ and $\exists$ extends as far as possible.

The expression

$$\forall x \in S : \ldots$$

extends to the end of its enclosing expression unless explicitly ended.

The scope of \textit{forall} and \textit{exists} extends as far as possible.

For example, this expression extends to the end of its enclosing expression unless explicitly ended.
The scope of $\forall$ and $\exists$ extends as far as possible.

The expression

$$(\forall x \in S : \ldots)$$

extends to the end of its enclosing expression unless explicitly ended

– by parentheses

The scope of *forall* and *exists* extends as far as possible.

For example, this expression extends to the end of its enclosing expression unless explicitly ended

by enclosing parentheses
The scope of $\forall$ and $\exists$ extends as far as possible.

The expression

$$[r \in T \mapsto \forall x \in S : \ldots]$$

extends to the end of its enclosing expression unless explicitly ended

– by parentheses

The scope of \textit{forall} and \textit{exists} extends as far as possible.

For example, this expression extends to the end of its enclosing expression unless explicitly ended

by enclosing parentheses

or similar brackets or braces

[slide 295]
The scope of \( \forall \) and \( \exists \) extends as far as possible.

The expression

\[
\land \\
\land \forall x \in S : \ldots \\
\land
\]

extends to the end of its enclosing expression unless explicitly ended

– by parentheses

– or by the end of a list item

or by the end of a conjunction or disjunction list item
The scope of $\forall$ and $\exists$ extends as far as possible.

The expression
\[
\land
(\forall x \in S : \ldots)
\land
\]
extends to the end of its enclosing expression unless explicitly ended
– by parentheses
– or by the end of a list item (which adds implicit parentheses)

or by the end of a conjunction or disjunction list item
which adds implicit parentheses
∀ x ∈ S : ....
∧ ∀ x ∈ T : ....

This expression

For example, this expression
\[ \forall x \in S : (\ldots \land \forall x \in T : \ldots) \]

This expression is parsed like this

For example, this expression is parsed as if these parentheses were added,
\[ \forall x \in S : (\ldots \land \forall x \in T : \ldots) \]

This expression is parsed like this, which is the same as this.

For example, this expression is parsed as if these parentheses were added, which is easier to read if we indent the second line.
∀ x ∈ S : ( ... \\
∧ ∀ x ∈ T : ... )

This expression is parsed like this which is the same as this.

The expression is illegal because \( x \) is declared here

So the expression is illegal because this \( x \), which is declared in the inner forall
\[ \forall [x] \in S : (\ldots \land \forall x \in T : \ldots) \]

This expression is parsed like this which is the same as this.

The expression is illegal because \( x \) is declared here when it’s already declared here.

So the expression is illegal because this \( x \), which is declared in the inner forall is already declared in the outer forall. And in TLA+ it’s illegal to redefine an identifier that’s already declared.
Let’s now look at comments in TLA+.
TLA+ has two kinds of comments.

TLA+ provides two kinds of comments.
TLA+ has two kinds of comments.

\[ x' = x + 1 \quad \text{\textit{An end of line comment.}} \]

TLA+ provides two kinds of comments.

An end of line comment
TLA+ has two kinds of comments.

\[ x' = x + 1 \quad \text{\color{red}{\textit{An end of line comment.}}} \]

TLA+ provides two kinds of comments. 

An end of line comment \textit{begins with backslash asterisk.}
TLA+ has two kinds of comments.

\[ x' = x + 1 \ \star \ \text{An end of line comment.} \]

\[ x' = x + (\star \text{This is a silly place for a comment } \star) \ 1 \]

TLA+ provides two kinds of comments.

An end of line comment begins with backslash asterisk.

Other comments
TLA+ has two kinds of comments.

\[ x' = x + 1 \text{ \texttt{ /* An end of line comment.}} \]

\[ x' = x + (\texttt{ /* This is a silly place for a comment */ }) 1 \]

TLA+ provides two kinds of comments.

An end of line comment begins with backslash asterisk.

Other comments are enclosed by these delimiters.
\[ x' = x + 1 \]  

(* This is a boxed comment. *)  

(* It looks very nice when *)  

(* it’s pretty-printed. *)  

(* ********************************************)

Boxed comments like this

[slide 309]
\[ x' = x + 1 \]

(* This is a boxed comment. *)
(* It looks very nice when *)
(* it’s pretty-printed. *)

This is a boxed comment. It looks very nice when it’s pretty-printed.

Boxed comments like this look nice when they’re pretty-printed.
$x' = x + 1$

(* This is a boxed comment. *)

(* It looks very nice when *)

(* it’s pretty-printed. *)

(* ******************************)

Typing boxed comments is easy with Toolbox editor commands

Boxed comments like this look nice when they’re pretty-printed.

It’s easy to type boxed comments using the Toolbox’s editing commands.
Typing Boxed Comments

To find out how to type boxed comments,
Typing Boxed Comments

See the Toolbox’s Help.

To find out how to type boxed comments, See the Toolbox’s Help pages.
Typing Boxed Comments

To find out how to type boxed comments, see the Toolbox’s Help pages.

To do that, click help
Typing Boxed Comments

To find out how to type boxed comments, see the Toolbox’s Help pages. To do that, click help then Dynamic Help.
Typing Boxed Comments

To find out how to type boxed comments, see the Toolbox’s Help pages.

To do that, click help then Dynamic Help.

Then

[slide 316]
Typing Boxed Comments

To find out how to type boxed comments, see the Toolbox’s Help pages. To do that, click help then Dynamic Help. Then click Contents.
Typing Boxed Comments

Open the Toolbox User Guide
Typing Boxed Comments

Open the Toolbox User Guide and find the Editing Modules page.
Typing Boxed Comments

Open the Toolbox User Guide and find the Editing Modules page. On that page

[slide 320]
Typing Boxed Comments

Open the Toolbox User Guide and find the Editing Modules page.

On that page go to Editing Comments.
Typing Boxed Comments

This page has lots more useful information.

Open the Toolbox User Guide and find the Editing Modules page.
On that page go to Editing Comments.
The Editing Modules page also has lots of other useful information.
A separator line:

---

You can make a spec easier to read by adding horizontal separator lines like this.

[slide 323]
A separator line:

--------------------------------------

Pretty printed like:

|----------------------------------|

You can make a spec easier to read by adding horizontal separator lines like this.

The line is pretty printed like this.
You can make a spec easier to read by adding horizontal separator lines like this.

The line is pretty printed like this.

These lines are purely decorative. They go between statements.
The specification of Transaction Commit, like the Die Hard specification, is very simple. But it moved us a tiny bit closer to real computer systems. And you’ve now learned a lot of the TLA+ you need to specify those systems.

Next, we examine two-phase commit – an algorithm for implementing transaction commit.
End of Lecture 5

TRANSACTION COMMIT