TLA^{+} Video Course – Lecture 5

Leslie Lamport

TRANSACTION 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 5
Transaction Commit

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 priviledge 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.

An Old-Fashioned Wedding

Anne

Here's how an old-fashioned wedding goes.

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

An Old-Fashioned Wedding

Henry

Anne

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

Minister

Henry

Anne

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

Henry, wilt thou have this woman to thy wedded wife?

Minister

Henry

Anne

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

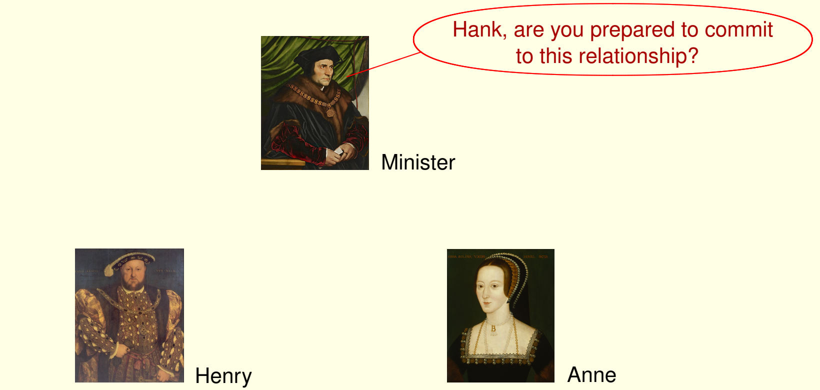
Minister

Henry

Anne

Let’s make it more modern. A 21st century minister might say:

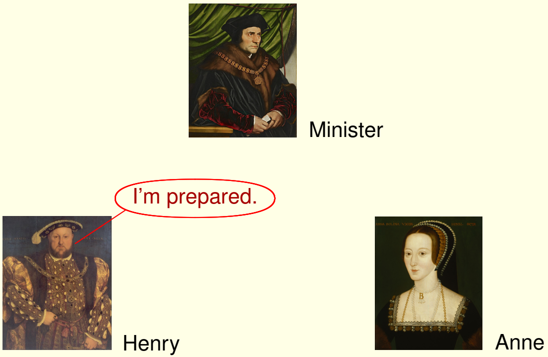
A More Modern Wedding



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:

A More Modern Wedding



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:

A More Modern Wedding

Anne, are you prepared to commit to this relationship?

Minister

Henry

Anne

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:

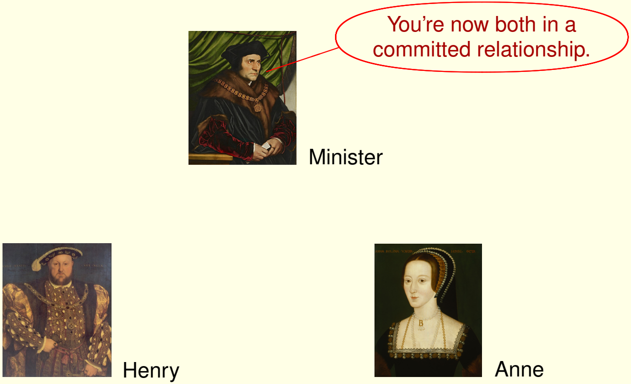
Anne, are you prepared to commit to this relationship? And Anne replies:

A More Modern Wedding

I’m prepared.

The minister then says:

A More Modern Wedding



I'm prepared.

The minister then says:

You're now both in a committed relationship.

What Can Go Wrong

Minister

Henry

Anne

What can go wrong in a wedding?

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

What Can Go Wrong

Anne, are you prepared to commit to this relationship?

Minister

Henry

Anne

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

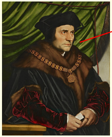
What can go wrong in a wedding?

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

This wedding is aborted.



Minister

Henry

Anne

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

Minister

Henry

Anne

Here’s another way the wedding can go amiss.

What Can Go Wrong

Here’s another way the wedding can go amiss.

Both the groom

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

Minister

Stop! Anne will regret it.

Henry

Anne

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.

What Can Go Wrong

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

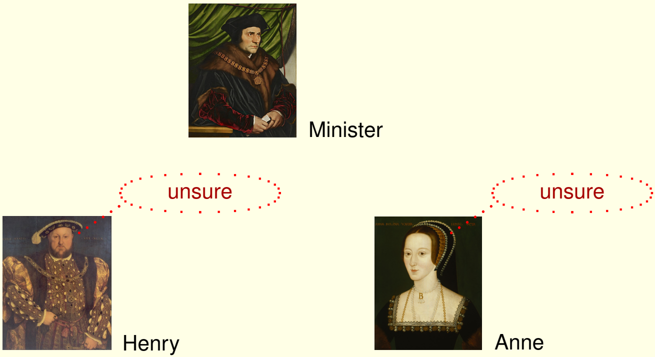
Minister

Henry

Anne

What does a wedding accomplish?

What a Wedding Accomplishes



What does a wedding accomplish?

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

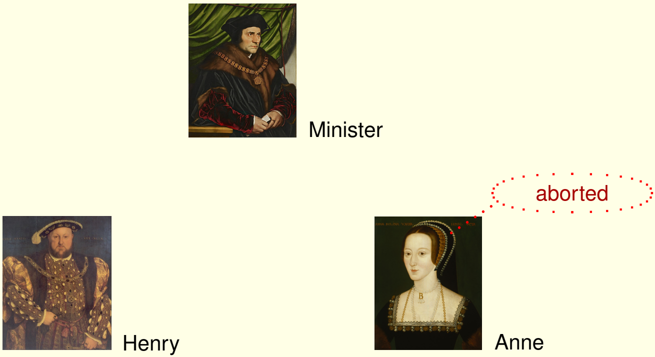
What a Wedding Accomplishes

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 a Wedding Accomplishes

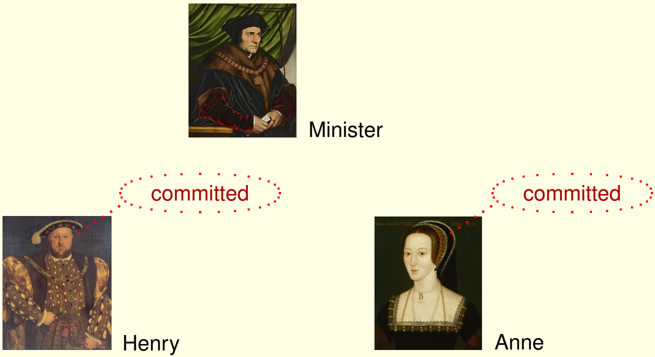


What does a wedding accomplish?

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

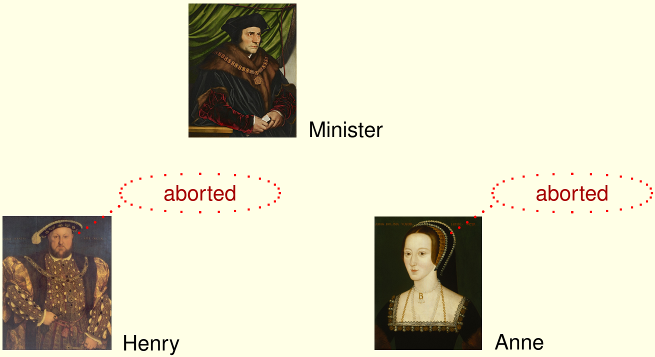
It allows them each to decide if they're prepared to commit to the relationship or if they want the wedding aborted.

What a Wedding Accomplishes



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

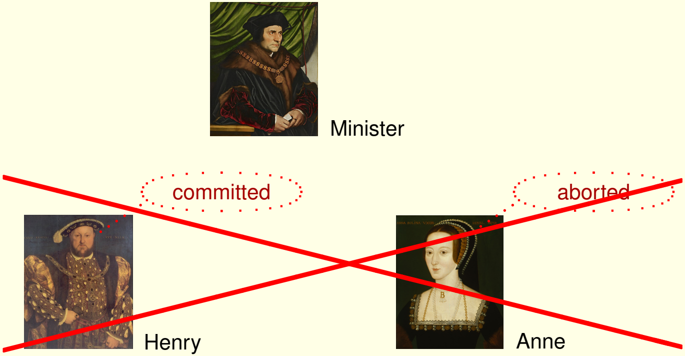
What a Wedding Accomplishes



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

Minister

Henry

Anne

What function does the minister perform?

The Minister

He implements the wedding.

Minister

Henry

Anne

What function does the minister perform?

His job is to implement the wedding.

The Minister

He implements the wedding.

He's part of how it works,

Minister

Henry

Anne

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.

Minister

Henry

Anne

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

Henry

Anne

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

Specifying a Wedding

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

Henry

Anne

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

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

Specifying a Wedding

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

 Henry

 Anne

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,

Henry

Anne

The specification mentions only the bride and groom,

Specifying a Wedding

The specification mentions only the bride and groom,

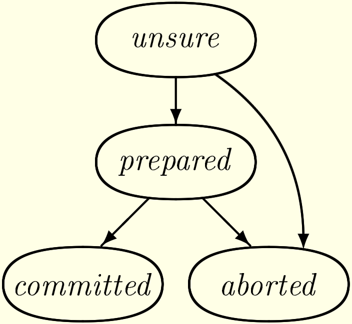
not the minister.

Henry

Anne

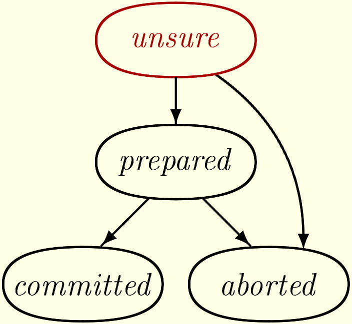
The specification mentions only the bride and groom, not the minister, who's part of how, not $what$.

The state/transition diagram of each participant:



Here's the state/transition diagram of each of the two participants: the bride and the groom.

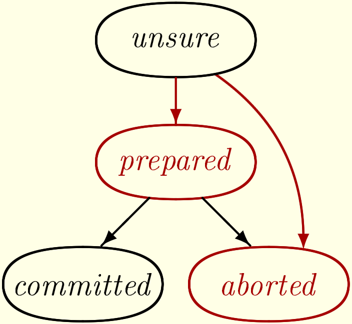
The state/transition diagram of each participant:



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.

The state/transition diagram of each participant:

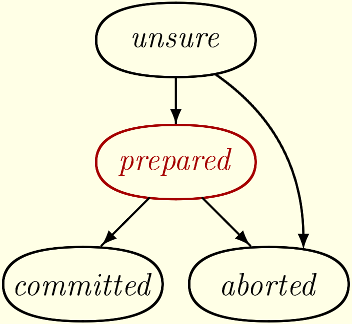


Here's the state/transition diagram of each of the two participants: the bride and the groom.

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

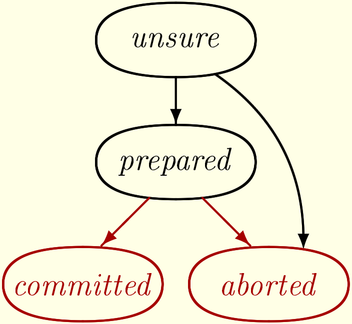
From that state, they can go into either the prepared or the aborted state.

The state/transition diagram of each participant:



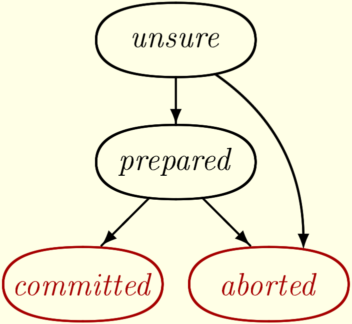
From the prepared state,

The state/transition diagram of each participant:



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

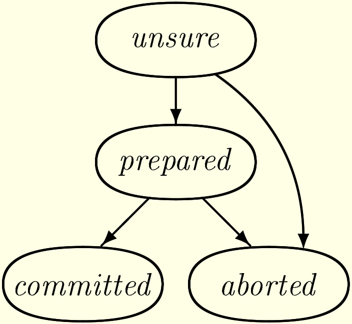
The state/transition diagram of each participant:



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

They remain forever in either of those two states.

The state/transition diagram of each participant:



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

They remain forever in either of those two states.

A Really Modern Wedding

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.

A Really Modern Wedding

 Henry

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

 Henry  Catherine A

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

 Henry  Catherine A  Anne B

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

 Henry

 Catherine A

 Anne B

 Jane

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


Henry


Catherine A


Anne B


Jane


Anne C

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


Henry


Catherine A


Anne B


Jane


Anne C


Catherine H

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


Henry


Catherine A


Anne B


Jane


Anne C


Catherine H

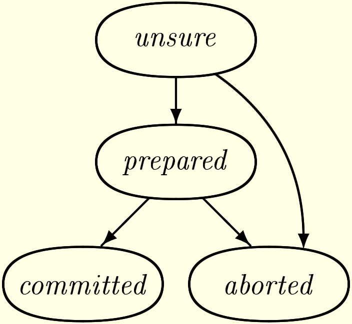

Catherine P

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.

TRANSACTION COMMIT

In a transaction commit . . .

A Wedding


Henry


Catherine A


Anne B


Jane

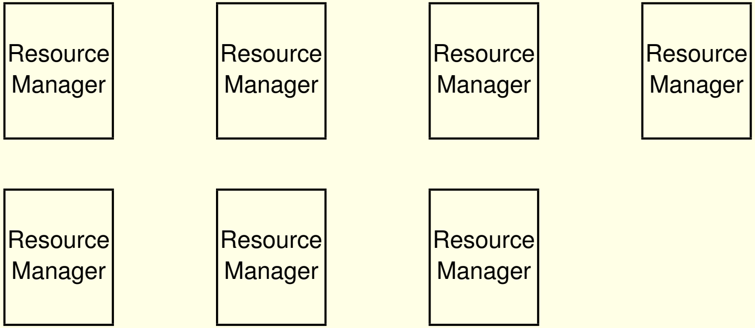

Anne C


Catherine H


Catherine P

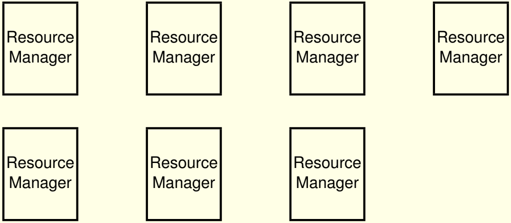
a wedding

A Database Transaction



a wedding is replaced by a database transaction.

A Database Transaction

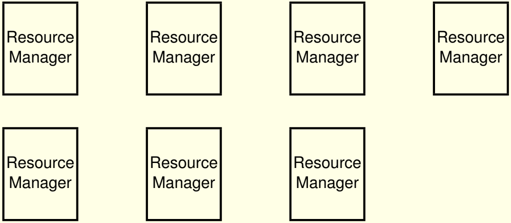


Is performed by the RMs.

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.

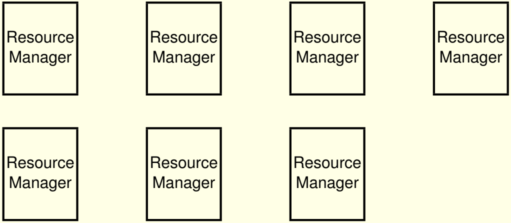
Can either commit or abort.

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.

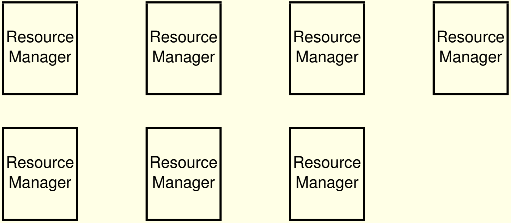
A Database Transaction



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



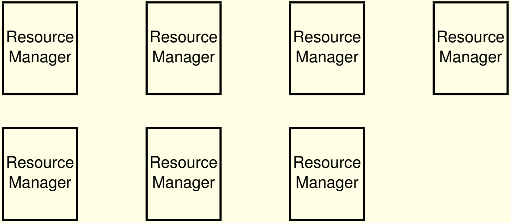
Can commit only if all RMs are prepared to commit.

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.

Transaction Commit $\longleftrightarrow$ Wedding

The execution of a transaction commit is just like a really modern wedding, which can have many participants.

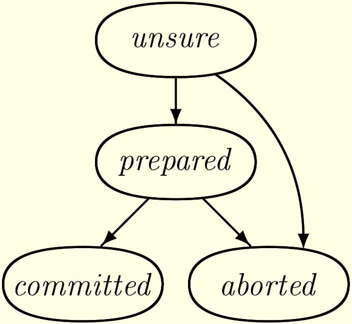
Transaction Commit $\longleftrightarrow$ Wedding

RM $\longleftrightarrow$ Participant

The execution of a transaction commit is just like a really modern wedding, which can have many participants.

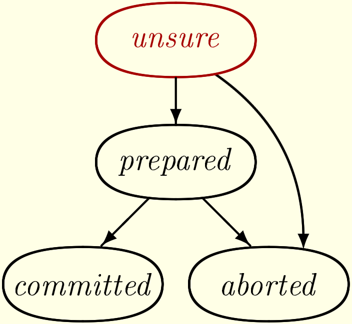
A resource manager corresponds to a participant in the wedding.

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.

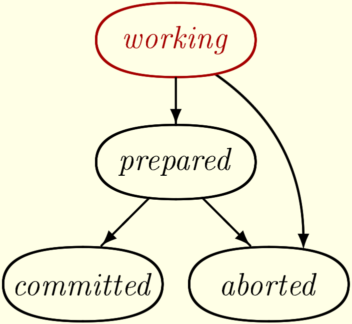
The state / transition diagram of each RM:



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

Except that the *unsure* state

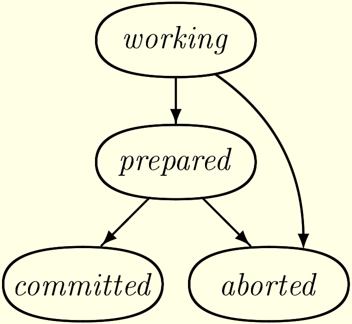
The state / transition diagram of each RM:



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 RM:



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 TLA^{+} SPEC

We now see how transaction commit can be specified in TLA+.

You will first:

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 $TCommit$.
- 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.

MODULE $TCommit$

The spec is in module $TCommit$.

MODULE $TCommit$

CONSTANT RM

The spec is in module $TCommit$.

It begins by declaring RM to be a constant

MODULE *TCommit*

$\boxed{\text{CONSTANT}}$ *RM*

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.

MODULE *TCommit*

CONSTANT $\boxed{RM}$ The set of all RMs.

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.

MODULE $TCommit$

CONSTANT RM The set of all RMs.

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?

MODULE $TCommit$

CONSTANT RM The set of all RMs.

Why is RM a set?

In TLA+, every value is a set.

In TLA+, every value is a set.

MODULE *TCommit*

CONSTANT RM The set of all RMs.

Why is RM a set?

In TLA+, every value is a set.

Even 42 and "abc" are sets.

In TLA+, every value is a set.

Even values like 42 and the string abc are sets.

MODULE *TCommit*

CONSTANT *RM*   The set of all RMs.

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.

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.

MODULE *TCommit*

CONSTANT *RM* The set of all RMs.

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*.*

[slide 90]

MODULE $TCommit$

CONSTANT RM

MODULE *TCommit*

CONSTANT *RM*

VARIABLE *rmState*

Next comes the declaration of the spec's single variable $rmState$.

MODULE $TCommit$

CONSTANT RM

VARIABLE $rmState$

$TCTypeOK \triangleq$

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.

MODULE *TCommit*

CONSTANT RM

VARIABLE $rmState$

$\boxed{TC}TypeOK \triangleq$

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.

[slide 94]

MODULE $TCommit$

CONSTANT RM

VARIABLE $rmState$

$TCTypeOK \triangleq$

$\quad rmState \in \boxed{\ldots}$

An array indexed by RMs.

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

MODULE $TCommit$

CONSTANT RM

VARIABLE $rmState$

$TCTypeOK \triangleq$
  $rmState \in \boxed{\ldots}$

An array indexed by RMs.

$rmState[r]$ is the state of RM r.

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

where $rmState[r]$ describes the state of resource manager r.

MODULE $TCommit$

CONSTANT RM

VARIABLE $rmState$

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

This is the TLA+ notation

MODULE *TCommit*

CONSTANT RM

VARIABLE $rmState$

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

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

MODULE *TCommit*

CONSTANT *RM*

VARIABLE *rmState*

$TCTypeOK \triangleq$

$rmState \in [RM \rightarrow \boxed{\cdots}\ ]$

The set of all arrays indexed by elements of RM

with values in $\cdots$.

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.

MODULE $TCommit$

CONSTANT RM

VARIABLE $rmState$

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

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

[slide 100]

MODULE $TCommit$

CONSTANT RM

VARIABLE $rmState$

$TCTypeOK \triangleq$
$\quad rmState \in [RM \rightarrow \boxed{\{\text{“working”}, \text{“prepared”}, \text{“committed”}, \text{“aborted”}\}}]$

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

[slide 101]

MODULE *TCommit*

CONSTANT RM

VARIABLE $rmState$

$TCTypeOK \triangleq$

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

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,*

MODULE *TCommit*

CONSTANT RM

VARIABLE $rmState$

$TCTypeOK \triangleq$

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

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,*

MODULE $TCommit$

CONSTANT RM

VARIABLE $rmState$

$TCTypeOK \triangleq$
$rmState \in [RM \rightarrow \{\text{“working”}, \text{“prepared”}, \boxed{\text{“committed”}}, \text{“aborted”}\}]$

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,*

$$\text{MODULE } TCommit$$

CONSTANT RM

VARIABLE $rmState$

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

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 105]

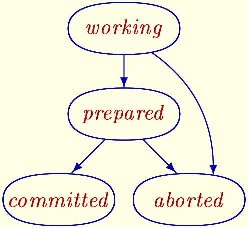
MODULE *TCommit*

CONSTANT RM

VARIABLE $rmState$

$TCTypeOK \triangleq$

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



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.* which represent the four possible states of a resource manager.

MODULE *TCommit*

CONSTANT RM

VARIABLE $rmState$

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

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.* which represent the four possible states of a resource manager.

MODULE $TCommit$

CONSTANT RM

VARIABLE $rmState$

$TCTypeOK \triangleq$

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

->

The right arrow is typed *dash greater than* in ASCII.

MODULE $TCommit$

CONSTANT RM

VARIABLE $rmState$

$$TCTypeOK \triangleq$$
$$\quad rmState \in [RM \rightarrow \{\text{“working”}, \text{“prepared”}, \text{“committed”}, \text{“aborted”}\}]$$

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

The initial predicate $TCInit$ asserts that $rmState$ equals

$$\text{MODULE } TCommit$$

CONSTANT RM

VARIABLE $rmState$

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

$$TCInit \triangleq rmState = \boxed{[r \in RM \mapsto \text{"working"}]}$$

The initial predicate $TCInit$ asserts that $rmState$ equals

this expression, which is TLA+ notation for

MODULE $TCommit$

CONSTANT RM

VARIABLE $rmState$

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

$TCInit \triangleq rmState = \boxed{[r \in RM \mapsto \text{"working"}]}$

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 $TCInit$ asserts that $rmState$ 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.

The TLA^{+} syntax for an array expression:

$$[\, variable \;\in\; set \;\mapsto\; expression \,]$$

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

The TLA$^+$ syntax for an array expression:

$$[\,variable \;\in\; set \;\mapsto\; expression\,]$$

|->

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

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

The TLA^{+} syntax for an array expression:

$$[\,variable \;\in\; set \;\mapsto\; expression\,]$$

$$[\qquad\qquad\qquad]$$

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

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

For example, inside square brackets

The TLA$^+$ syntax for an array expression:

$$[\, variable \;\in\; set \;\mapsto\; expression \,]$$

$$[\, i \qquad\qquad\qquad]$$

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

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

For example, inside square brackets
We put the variable i

The TLA⁺ syntax for an array expression:

$$[\,variable \in set \mapsto expression\,]$$

$$[\,i \in \qquad\qquad]$$

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

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

For example, inside square brackets
We put the variable i element of

[slide 116]

The TLA$^+$ syntax for an array expression:

$$[\,variable \;\in\; set \;\mapsto\; expression\,]$$

$$[\,i \;\in\; 1\,..\,42 \qquad\qquad]$$

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

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

For example, inside square brackets
We put the variable i element of The set of integers from one through 42

The TLA$^+$ syntax for an array expression:

$$[\,\mathit{variable} \;\in\; \mathit{set} \;\mapsto\; \mathit{expression}\,]$$

$$[\,i \in 1\,..\,42 \mapsto \quad]$$

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

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

For example, inside square brackets
We put the variable i element of The set of integers from one through 42
maps to symbol

[slide 118]

The TLA$^+$ syntax for an array expression:

$$[\,variable \;\in\; set \;\mapsto\; expression\,]$$

$$[\,i \in 1\,..\,42 \mapsto i^2\,]$$

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

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

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

The TLA+ syntax for an array expression:

$$[\, i \;\in\; 1\,..\,42 \;\mapsto\; i^2 \,]$$

So this definition

The TLA$^+$ syntax for an array expression:

$$sqr \triangleq [i \in 1\,..\,42 \mapsto i^2]$$

So this definition

The TLA$^+$ syntax for an array expression:

$$sqr \triangleq [i \in 1 \,..\, 42 \mapsto i^2]$$

Defines sqr to be an array with index set $1 \,..\, 42$

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:

$$sqr \triangleq [i \in 1 \,..\, 42 \mapsto i^2]$$

Defines sqr to be an array with index set $1 \,..\, 42$
such that $sqr[i] = i^2$ 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.

Terminology

Let’s look at some different terminology

Terminology

Programming **Math**

Let's look at some different terminology used in programming and math for the same things.

Terminology

Programming	Math
array	

Let's look at some different terminology used in programming and math for the same things.

What programmers call an array

[slide 126]

Terminology

Programming	**Math**
array	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.

Terminology

Programming	Math
array	function
index set	

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

Terminology

Programming	**Math**
array	function
index set	domain

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.

Terminology

Programming	**Math**
array	function
index set	domain
$f[e]$	

Programmers use square brackets for array application.

Terminology

Programming	**Math**
array	function
index set	domain
$f[e]$	$f(e)$

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

Terminology

Programming	**Math**
array	function
index set	domain
$f[e]$	$f(e)$

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 terminlogy for

Terminology

Programming	**Math**
~~array~~	function
index set	domain
$f[e]$	$f(e)$

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 terminlogy for functions

Terminology

Programming	**Math**
~~array~~	function
~~index set~~	domain
$f[e]$	$f(e)$

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 terminlogy for functions and their domains.

Terminology

Programming	**Math**
~~array~~	function
~~index set~~	domain
$f[e]$	$\cancel{f(e)}$

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 terminlogy for functions and their domains.

However, TLA+ uses square brackets for function application

Terminology

Programming	**Math**
~~array~~	function
~~index set~~	domain
$f[e]$	$\cancel{f(e)}$
	Has another use.

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 terminlogy 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 $0..n$.

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 $0 \,..\, n$.

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 $0 \,..\, n$.

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.

MODULE $TCommit$

CONSTANT RM

VARIABLE $rmState$

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

$$TCInit \triangleq rmState = [r \in RM \mapsto \text{"working"}]$$

Let's return to the spec

MODULE $TCommit$

CONSTANT RM

VARIABLE $rmState$

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

$$TCInit \triangleq rmState = [r \in RM \mapsto \text{"working"}]$$

$$\vdots$$

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

Let's return to the spec and jump down to the definition of the next-state formula $TCNext$

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

Let's return to the spec and jump down to the definition of the next-state formula $TCNext$

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

Let's return to the spec and jump down to the definition of the next-state formula $TCNext$

$$\exists\, r \in RM : Prepare(r) \lor Decide(r)$$

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

[slide 144]

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

There exists

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

there exists

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

`\E`

There exists

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

there exists

Typed backslash E in ASCII.

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

There exists r in RM

There exists some r in the set RM

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

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

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

$$\exists\, r \in RM : Prepare(r) \lor Decide(r)$$

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

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 $r4$.

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

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

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 $r1$, $r2$, $r3$, and $r4$.

Then this formula equals

$$\exists\, r \in RM : Prepare(r) \lor Decide(r)$$

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

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

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

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

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

then this formula equals

$$\begin{array}{ll} \lor & Prepare(\text{“}r1\text{”}) \lor Decide(\text{“}r1\text{”}) \\ \lor & \\ \lor & \\ \lor & \end{array}$$

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

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

If $RM = \{$"r1", $\boxed{\text{"}r\text{2"}}$, "$r$3", "$r$4"$\}$

then this formula equals

$$\begin{aligned}
&\lor\ Prepare(\text{"}r1\text{"}) \lor Decide(\text{"}r1\text{"})\\
&\lor\ Prepare(\text{"}r2\text{"}) \lor Decide(\text{"}r2\text{"})\\
&\lor\\
&\lor
\end{aligned}$$

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

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

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

then this formula equals

$$\begin{aligned}
&\lor\ \ Prepare(\text{"}r1\text{"}) \lor Decide(\text{"}r1\text{"})\\
&\lor\ \ Prepare(\text{"}r2\text{"}) \lor Decide(\text{"}r2\text{"})\\
&\lor\ \ Prepare(\text{"}r3\text{"}) \lor Decide(\text{"}r3\text{"})\\
&\lor
\end{aligned}$$

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

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

If $RM \;=\; \{$"r1", "r2", "r3", $\boxed{\text{"}r\text{4"}}\}$

then this formula equals

$$\begin{aligned}
&\lor\; Prepare(\text{"}r\text{1"}) \lor Decide(\text{"}r\text{1"})\\
&\lor\; Prepare(\text{"}r\text{2"}) \lor Decide(\text{"}r\text{2"})\\
&\lor\; Prepare(\text{"}r\text{3"}) \lor Decide(\text{"}r\text{3"})\\
&\lor\; Prepare(\text{"}r\text{4"}) \lor Decide(\text{"}r\text{4"})
\end{aligned}$$

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

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

$\exists$ declares r local to formula.

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

$$\exists \boxed{xyz} \in RM : Prepare(\boxed{xyz}) \vee Decide(\boxed{xyz})$$

$\exists$ declares r local to formula.

$r \leftarrow xyz$ 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.

$$\exists \boxed{xyz} \in RM : Prepare(\boxed{xyz}) \vee Decide(\boxed{xyz})$$

$\exists$ declares r local to formula.

$r \leftarrow xyz$ doesn't change meaning if xyz not declared or defined.

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.

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.

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

Let's now return to the spec and move back up

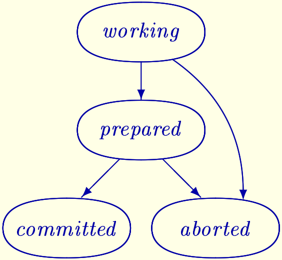
$$TCNext \triangleq \exists\, r \in RM : \boxed{Prepare}(r) \lor \boxed{Decide}(r)$$

Let's now return to the spec and move back up to the definitions of $Prepare$ and $Decide$,

$Prepare(r) \triangleq$

Let's now return to the spec and move back up to the definitions of $Prepare$ and $Decide$, starting with $Prepare$

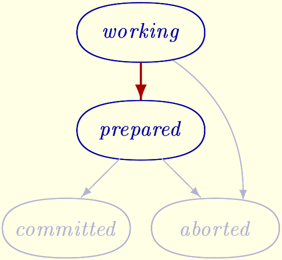
$Prepare(r) \triangleq$



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.

$Prepare(r) \triangleq$

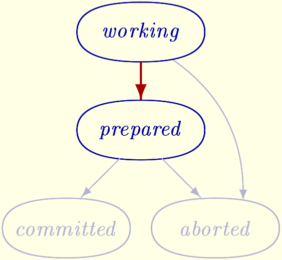


Let's now return to the spec and move back up to the definitions of $Prepare$ and $Decide$, starting with $Prepare$

Recall the state / transition graph of a resource manager.

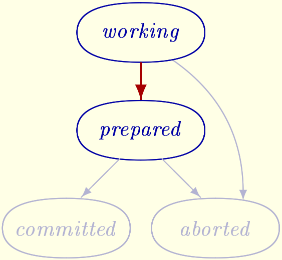
$Prepare$ of r describes the *working* to *prepared* step of resource manager r.

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



This step can be taken only when the current state of resource manager r is $working$, so $rmState$ of r equal to the string *working* must be true.

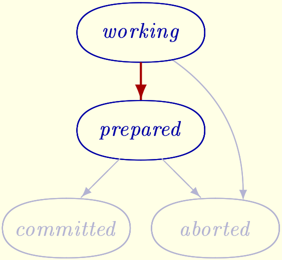
$$Prepare(r) \triangleq \land\ rmState[r] = \text{"working"}$$
$$\land$$



This step can be taken only when the current state of resource manager r is $working$, so $rmState$ of r equal to the string *working* must be true.

The step must change the value of $rmState$ of r to the string $prepared$.

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



This step can be taken only when the current state of resource manager r is $working$, so $rmState$ of r equal to the string *working* must be true.

The step must change the value of $rmState$ of r to the string $prepared$.

Most people think that condition is expressed like this.

$$
\begin{aligned}
Prepare(r) \;\triangleq\;\; & \land\, rmState[r] = \text{“working”} \\
& \land\, \boxed{rmState'[r] = \text{“prepared”}}
\end{aligned}
$$

What's wrong with this?

This step can be taken only when the current state of resource manager r is $working$, so $rmState$ of r equal to the string *working* must be true.

The step must change the value of $rmState$ of r to the string $prepared$.

Most people think that condition is expressed like this.

Stop the video and figure out why this is wrong.

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

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

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

What does this formula say?

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

What does this formula actually say?

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

What does this formula 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?

It says that the value of $rmState$ of r in the new state is the string “$prepared$”.

$rmState'[r] = \text{"}prepared\text{"}$

What does this formula say?

The value of $rmState[r]$ in the new state is "$prepared$".

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?

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

What does this formula say?

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

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

Nothing!

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

Absolutely nothing!

$\cancel{rmState'[r] = \text{"}prepared\text{"}}$

The spec can't just say what the new value of $rmState$ of r is.

$\cancel{rmState'[r] = \text{"}prepared\text{"}}$

$rmState' =$

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.

$\cancel{rmState'[r] = \text{"}prepared\text{"}}$

$$rmState' = [\, s \in RM \mapsto \;\; \cdots \;\;]$$

It looks like this, where we have to replace the dot dot dot

$\cancel{rmState'[r] = \text{"}prepared\text{"}}$

$$rmState' = [\, s \in RM \mapsto \quad \cdots \quad]$$

$\uparrow$

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.

$\cancel{rmState'[r] = \text{"}prepared\text{"}}$

$$rmState' = [\, s \in RM \mapsto \text{IF } s = r \text{ THEN } \text{"}prepared\text{"} \\ \text{ELSE } \quad]$$

If s is resource manager r, then the value of $rmState$ of s in the new state should be the string $prepared$

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

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

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.

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

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

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.

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

$$[\, s \in RM \;\mapsto\; \text{IF } s = r \text{ THEN } \text{"}prepared\text{"} \\ \qquad\qquad\qquad\qquad\qquad \text{ELSE } rmState[s]\,]$$

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.

This is correct, but it's too long-winded.
We need a shorter way to write this expression.

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

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

TLA+ provides this EXCEPT construct.

Everyone hates it.

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

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

TLA+ provides this EXCEPT construct.

Everyone hates it.

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

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

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

meaningless syntax

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.

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

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

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.

$$
\begin{aligned}
Prepare(r) \triangleq\ & \land\ rmState[r] = \text{“working”} \\
& \land\ rmState' = [rmState\ \textsc{except}\ ![r] = \text{“prepared”}\,]
\end{aligned}
$$

So, here's the complete definition of $Prepare$.

$$Decide(r) \triangleq$$

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

$$Decide(r) \triangleq \lor$$

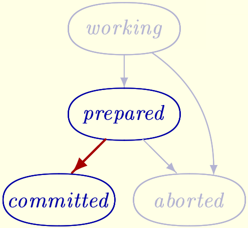
$$\lor$$

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.

$Decide(r) \triangleq \lor$ Describes a $prepared \rightarrow committed$ step.

$\lor$



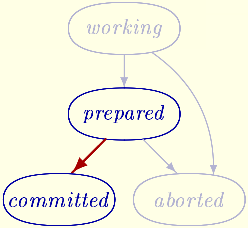
Now for the definition of $Decide$. It describes possible steps in which resource manager r reaches a $committed$ or $aborted$ state.

It's the disjunction of two formulas.

The first describes a step in which resource manager r goes from the *prepared* state to the *committed* state.

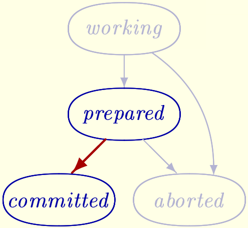
$$Decide(r) \triangleq \lor \land rmState[r] = \text{"prepared"}$$

$$\lor$$



Such a step can occur only if r is in the $prepared$ state.

$$Decide(r) \triangleq \lor \land rmState[r] = \text{"prepared"}$$
$$\qquad\qquad \land \text{ every RM is } prepared \text{ or } committed$$
$$\lor$$

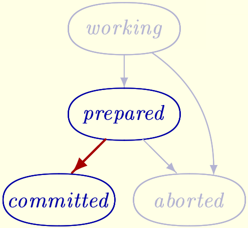


Such a step can occur only if r is in the *prepared* state.

r can commit only if every resource manager is in the *prepared* or *committed* state.

$$Decide(r) \triangleq \lor \land rmState[r] = \text{"prepared"}$$
$$\land\ canCommit$$

$$\lor$$

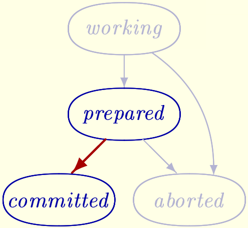


Such a step can occur only if r is in the $prepared$ state.

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

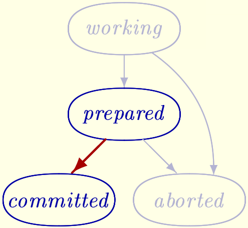
This condition is written in a formula named $canCommit$, whose definition we'll look at later.

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



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

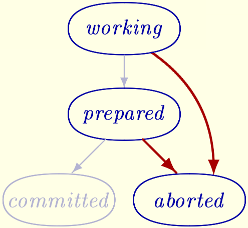
$$
\begin{array}{lll}
Decide(r) & \triangleq & \lor \land rmState[r] = \text{"prepared"} \\
& & \quad \land canCommit \\
& & \quad \land rmState' = [rmState \ \boxed{\text{EXCEPT}} \ ![r] = \text{"committed"}] \\
& & \lor
\end{array}
$$



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 EXCEPT .

$$
\begin{aligned}
Decide(r) \triangleq\ & \lor \land rmState[r] = \text{“prepared”} \\
& \quad \land canCommit \\
& \quad \land rmState' = [rmState \text{ EXCEPT } ![r] = \text{“committed”}] \\
& \lor \ \text{Describes steps that abort.}
\end{aligned}
$$

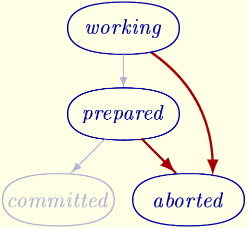


And in the new state, r is $committed$ and the state of every other resource manager remains the same.

This is expressed with our friend EXCEPT .

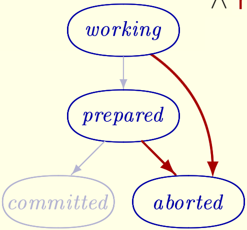
The second disjunction describes possible transitions to the $aborted$ state.

$$
\begin{aligned}
Decide(r) \;\triangleq\;\; & \lor \land rmState[r] = \text{"prepared"} \\
& \quad \land canCommit \\
& \quad \land rmState' = [rmState \text{ EXCEPT } ![r] = \text{"committed"}] \\
& \lor \land rmState[r] \in \{\text{"working"}, \text{"prepared"}\}
\end{aligned}
$$



r can abort from the $working$ or $prepared$ state, so $rmState$ of r must be an element of the set consisting of the two strings $working$ and $prepared$.

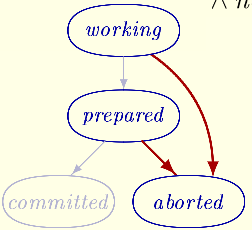
$$
\begin{aligned}
Decide(r) \;\triangleq\; & \lor \land\; rmState[r] = \text{“prepared”} \\
& \land\; canCommit \\
& \land\; rmState' = [rmState \text{ EXCEPT } ![r] = \text{“committed”}] \\
& \lor \land\; rmState[r] \in \{\text{“working”}, \text{“prepared”}\} \\
& \land\; \text{no RM is } committed
\end{aligned}
$$



r can abort from the *working* or *prepared* state, so *rmState* of r must be an element of the set consisting of the two strings *working* and *prepared*.

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

$$Decide(r) \triangleq \begin{array}{l} \lor \land rmState[r] = \text{“prepared”} \\ \quad \land canCommit \\ \quad \land rmState' = [rmState \text{ EXCEPT } ![r] = \text{“committed”}] \\ \lor \land rmState[r] \in \{\text{“working”}, \text{“prepared”}\} \\ \quad \land notCommitted \end{array}$$

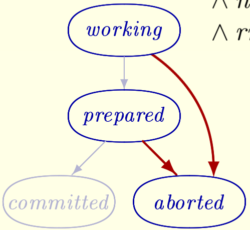


r can abort from the $working$ or $prepared$ state, so $rmState$ of r must be an element of the set consisting of the two strings $working$ and $prepared$.

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

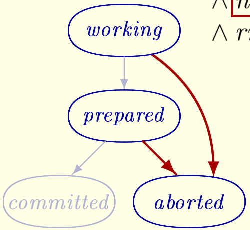
This condition is written as formula $notCommitted$, whose definition we'll look at later.

$$
\begin{aligned}
Decide(r) \triangleq\ & \lor \land rmState[r] = \text{"prepared"} \\
& \quad \land canCommit \\
& \quad \land rmState' = [rmState \ \text{EXCEPT} \ ![r] = \text{"committed"}] \\
& \lor \land rmState[r] \in \{\text{"working"}, \text{"prepared"}\} \\
& \quad \land notCommitted \\
& \quad \land rmState' = [rmState \ \text{EXCEPT} \ ![r] = \text{"aborted"}]
\end{aligned}
$$



And the state of r changes to $aborted$, while the state of all other resource managers remain the same.

$$
\begin{array}{lcl}
Decide(r) & \triangleq & \lor \land rmState[r] = \text{“prepared”} \\
 & & \land \boxed{canCommit} \\
 & & \land rmState' = [rmState \text{ EXCEPT } ![r] = \text{“committed”}] \\
 & & \lor \land rmState[r] \in \{\text{“working”}, \text{“prepared”}\} \\
 & & \land \boxed{notCommitted} \\
 & & \land rmState' = [rmState \text{ EXCEPT } ![r] = \text{“aborted”}]
\end{array}
$$



And the state of r changes to $aborted$, while the state of all other resource managers remain the same.

We now look at the definitions of $canCommit$ and $notCommitted$, but first a digression.

$$\exists\, r \in RM : Prepare(r) \lor Decide(r)$$

Remember that this formula asserts:

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

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 $RM = \{\text{“}r\text{1”}, \text{“}r\text{2”}, \text{“}r\text{3”}, \text{“}r\text{4”}\}$

then the formula equals

$$\begin{array}{l} \lor \; Prepare(\text{“}r\text{1”}) \lor Decide(\text{“}r\text{1”}) \\ \lor \; Prepare(\text{“}r\text{2”}) \lor Decide(\text{“}r\text{2”}) \\ \lor \; Prepare(\text{“}r\text{3”}) \lor Decide(\text{“}r\text{3”}) \\ \lor \; Prepare(\text{“}r\text{4”}) \lor Decide(\text{“}r\text{4”}) \end{array}$$

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.

$$\exists\, r \in RM : Prepare(r) \lor Decide(r)$$

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

$$\boxed{\forall} r \in RM : Prepare(r) \lor Decide(r)$$

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 this *forall* symbol.

[slide 206]

$$\boxed{\forall} r \in RM : Prepare(r) \lor Decide(r)$$

`\A`

Typed $backslash$ *A* in ASCII.

$$\forall\, r \in RM : Prepare(r) \lor Decide(r)$$

Typed $backslash$ *A* in ASCII.

This formula asserts that:

$$\forall r \in RM : \boxed{Prepare(r) \lor Decide(r)}$$

For all r in RM, this subformula is true.

Typed $backslash$ *A* in ASCII.

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

$$\forall r \in RM : Prepare(r) \lor Decide(r)$$

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

then the formula equals

$$\begin{array}{|l|l}\hline \land & Prepare(\text{"}r1\text{"}) \lor Decide(\text{"}r1\text{"}) \\ \land & Prepare(\text{"}r2\text{"}) \lor Decide(\text{"}r2\text{"}) \\ \land & Prepare(\text{"}r3\text{"}) \lor Decide(\text{"}r3\text{"}) \\ \land & Prepare(\text{"}r4\text{"}) \lor Decide(\text{"}r4\text{"}) \\ \hline \end{array}$$

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 *forall* formula equals this *conjunction* of four formulas.

Now to the definitions of $canCommit$ and $notCommitted$.

$canCommit \triangleq$

Remember that $canCommit$ should assert that

$canCommit \triangleq$ every RM is $prepared$ or $committed$

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

$$canCommit \triangleq \forall r \in RM : rmState[r] \in \{\text{"prepared"}, \text{"committed"}\}$$

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

This formula asserts that for every resource manager r, the value of $rmState$ of r is either the string $prepared$ or the string $committed$.

$notCommitted \triangleq$ no RM is $committed$

Remember that $notCommitted$ should assert that no resource manager is committed.

$$notCommitted \triangleq \forall r \in RM : rmState[r] \neq \text{"committed"}$$

Remember that $notCommitted$ 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 $committed$.

$$
\begin{aligned}
Decide(r) \triangleq\ & \lor \land rmState[r] = \text{“prepared”} \\
& \quad \land canCommit \\
& \quad \land rmState' = [rmState \text{ EXCEPT } ![r] = \text{“committed”}] \\
& \lor \land rmState[r] \in \{\text{“working”}, \text{“prepared”}\} \\
& \quad \land notCommitted \\
& \quad \land rmState' = [rmState \text{ EXCEPT } ![r] = \text{“aborted”}]
\end{aligned}
$$

Let’s take another look at the definition of $Decide$.

$$
\begin{array}{lll}
Decide(r) & \triangleq & \lor \land rmState[r] = \text{"prepared"} \\
& & \quad \land \boxed{canCommit} \\
& & \quad \land rmState' = [rmState \text{ EXCEPT } ![r] = \text{"committed"}] \\
& & \lor \land rmState[r] \in \{\text{"working"}, \text{"prepared"}\} \\
& & \quad \land notCommitted \\
& & \quad \land rmState' = [rmState \text{ EXCEPT } ![r] = \text{"aborted"}]
\end{array}
$$

Let's take another look at the definition of $Decide$.

Replacing $canCommit$ by its definition doesn't change the meaning of $Decide$ of r.

$$
\begin{aligned}
Decide(r) \;\triangleq\; & \lor \land\ rmState[r] = \text{“prepared”} \\
& \quad \land\ \boxed{canCommit} \\
& \quad \land\ rmState' = [rmState \text{ EXCEPT } ![r] = \text{“committed”}] \\
& \lor \land\ rmState[r] \in \{\text{“working”}, \text{“prepared”}\} \\
& \quad \land\ notCommitted \\
& \quad \land\ rmState' = [rmState \text{ EXCEPT } ![r] = \text{“aborted”}]
\end{aligned}
$$

$$
canCommit \;\triangleq\; \forall\, r \in RM : rmState[r] \in \{\text{“prepared”}, \text{“committed”}\}
$$

Let’s take another look at the definition of $Decide$.

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

Here’s the definition of $canCommit$ again.

$$
\begin{aligned}
Decide(r) \triangleq\ & \lor\ \land\ rmState[r] = \text{"prepared"} \\
& \ \land\ \boxed{\forall\, s \in RM : rmState[s] \in \{\text{"prepared"}, \text{"committed"}\}} \\
& \ \land\ rmState' = [rmState\ \text{EXCEPT}\ ![r] = \text{"committed"}] \\
& \lor\ \land\ rmState[r] \in \{\text{"working"}, \text{"prepared"}\} \\
& \ \land\ notCommitted \\
& \ \land\ rmState' = [rmState\ \text{EXCEPT}\ ![r] = \text{"aborted"}]
\end{aligned}
$$

$$
canCommit \triangleq \forall\, r \in RM : rmState[r] \in \{\text{"prepared"}, \text{"committed"}\}
$$

Let's take another look at the definition of $Decide$.

Replacing $canCommit$ by *its* definition doesn't change the meaning of $Decide$ of r.

Here's the definition of $canCommit$ again.

Replacing $canCommit$ by its definition yields this formula.

$$
\begin{aligned}
Decide(r) \triangleq\ & \lor \land rmState[r] = \text{"prepared"} \\
& \quad \land \boxed{\forall s \in RM : rmState[s] \in \{\text{"prepared"}, \text{"committed"}\}} \\
& \quad \land rmState' = [rmState \text{ EXCEPT } ![r] = \text{"committed"}] \\
& \lor \land rmState[r] \in \{\text{"working"}, \text{"prepared"}\} \\
& \quad \land notCommitted \\
& \quad \land rmState' = [rmState \text{ EXCEPT } ![r] = \text{"aborted"}]
\end{aligned}
$$

$$
canCommit \triangleq \forall \boxed{r} \in RM : rmState[r] \in \{\text{"prepared"}, \text{"committed"}\}
$$

We have to change the bound variable r used in the definition of $canCommit$.

$$
\begin{aligned}
Decide(r) \;\triangleq\; & \lor \land\ rmState[r] = \text{“prepared”} \\
& \quad \land\ \forall \boxed{s} \in RM : rmState[\boxed{s}] \in \{\text{“prepared”}, \text{“committed”}\} \\
& \quad \land\ rmState' = [rmState\ \text{EXCEPT}\ ![r] = \text{“committed”}] \\
& \lor \land\ rmState[r] \in \{\text{“working”}, \text{“prepared”}\} \\
& \quad \land\ notCommitted \\
& \quad \land\ rmState' = [rmState\ \text{EXCEPT}\ ![r] = \text{“aborted”}]
\end{aligned}
$$

$$
canCommit \;\triangleq\; \forall \boxed{r} \in RM : rmState[r] \in \{\text{“prepared”}, \text{“committed”}\}
$$

We have to change the bound variable r used in the definition of $canCommit$.
to some other variable like s

$$
\begin{aligned}
Decide(\boxed{r}) \triangleq\ & \lor \land rmState[r] = \text{"prepared"} \\
& \quad \land \forall \boxed{s} \in RM : rmState[\boxed{s}] \in \{\text{"prepared"}, \text{"committed"}\} \\
& \quad \land rmState' = [rmState \text{ EXCEPT } ![r] = \text{"committed"}] \\
& \lor \land rmState[r] \in \{\text{"working"}, \text{"prepared"}\} \\
& \quad \land notCommitted \\
& \quad \land rmState' = [rmState \text{ EXCEPT } ![r] = \text{"aborted"}]
\end{aligned}
$$

$$
canCommit \triangleq \forall \boxed{r} \in RM : rmState[r] \in \{\text{"prepared"}, \text{"committed"}\}
$$

We have to change the bound variable r used in the definition of $canCommit$.
to some other variable like s to avoid a name conflict with this r

$$
\begin{aligned}
Decide(r) \;\triangleq\; & \lor \land\, rmState[r] = \text{“prepared”} \\
& \quad \land\, \forall\, s \in RM : rmState[s] \in \{\text{“prepared”}, \text{“committed”}\} \\
& \quad \land\, rmState' = [rmState \text{ EXCEPT } ![r] = \text{“committed”}] \\
& \lor \land\, rmState[r] \in \{\text{“working”}, \text{“prepared”}\} \\
& \quad \land\, \boxed{notCommitted} \\
& \quad \land\, rmState' = [rmState \text{ EXCEPT } ![r] = \text{“aborted”}]
\end{aligned}
$$

We have to change the bound variable r used in the definition of $canCommit$. to some other variable like s to avoid a name conflict with this r

Similarly, we can replace $notCommitted$

$$
\begin{array}{lll}
Decide(r) & \triangleq & \lor \land rmState[r] = \text{“prepared”} \\
 & & \land \forall s \in RM : rmState[s] \in \{\text{“prepared”}, \text{“committed”}\} \\
 & & \land rmState' = [rmState \text{ EXCEPT } ![r] = \text{“committed”}] \\
 & & \lor \land rmState[r] \in \{\text{“working”}, \text{“prepared”}\} \\
 & & \land \forall s \in RM : rmState[s] \neq \text{“committed”} \\
 & & \land rmState' = [rmState \text{ EXCEPT } ![r] = \text{“aborted”}]
\end{array}
$$

We have to change the bound variable r used in the definition of $canCommit$. to some other variable like s to avoid a name conflict with this r

Similarly, we can replace $notCommitted$

with its definition.

$$
\begin{aligned}
Decide(r) \;\triangleq\; & \lor\; \land\; rmState[r] = \text{“prepared”} \\
& \;\; \land\; \forall\, s \in RM : rmState[s] \in \{\text{“prepared”}, \text{“committed”}\} \\
& \;\; \land\; rmState' = [rmState \text{ EXCEPT } ![r] = \text{“committed”}] \\
& \lor\; \land\; rmState[r] \in \{\text{“working”}, \text{“prepared”}\} \\
& \;\; \land\; \forall\, s \in RM : rmState[s] \neq \text{“committed”} \\
& \;\; \land\; rmState' = [rmState \text{ EXCEPT } ![r] = \text{“aborted”}]
\end{aligned}
$$

Definitions provide a simple and powerful way of hierarchically decomposing formulas to make them easier to read.

Definitions provide a simple and powerful way of hierarchically decomposing formulas to make them easier to read.

$$
\begin{aligned}
Decide(r) \;\triangleq\; & \lor \land\; rmState[r] = \text{“prepared”} \\
& \quad\; \land\; \forall\, s \in RM : rmState[s] \in \{\text{“prepared”}, \text{“committed”}\} \\
& \quad\; \land\; rmState' = [rmState \;\text{EXCEPT}\; ![r] = \text{“committed”}] \\
& \lor \land\; rmState[r] \in \{\text{“working”}, \text{“prepared”}\} \\
& \quad\; \land\; \forall\, s \in RM : rmState[s] \neq \text{“committed”} \\
& \quad\; \land\; rmState' = [rmState \;\text{EXCEPT}\; ![r] = \text{“aborted”}]
\end{aligned}
$$

$Decide(r)$ depends on

Whether a $Decide$ of r step is possible and what it can do depends on

$$
\begin{aligned}
Decide(r) \;\triangleq\; &\lor \land\ rmState[r] = \text{“prepared”}\\
&\ \land\ \boxed{\forall\, s \in RM : rmState[s] \in \{\text{“prepared”}, \text{“committed”}\}}\\
&\ \land\ rmState' = [rmState \text{ EXCEPT } ![r] = \text{“committed”}]\\
&\lor \land\ rmState[r] \in \{\text{“working”}, \text{“prepared”}\}\\
&\ \land\ \boxed{\forall\, s \in RM : rmState[s] \neq \text{“committed”}}\\
&\ \land\ rmState' = [rmState \text{ EXCEPT } ![r] = \text{“aborted”}]
\end{aligned}
$$

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

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

$$
\begin{aligned}
Decide(r) \triangleq\ & \lor \land rmState[r] = \text{“prepared”} \\
& \quad \land \boxed{\forall s \in RM : rmState[s] \in \{\text{“prepared”}, \text{“committed”}\}} \\
& \quad \land rmState' = [rmState \text{ EXCEPT } ![r] = \text{“committed”}] \\
& \lor \land rmState[r] \in \{\text{“working”}, \text{“prepared”}\} \\
& \quad \land \boxed{\forall s \in RM : rmState[s] \neq \text{“committed”}} \\
& \quad \land rmState' = [rmState \text{ EXCEPT } ![r] = \text{“aborted”}]
\end{aligned}
$$

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

How can this be implemented?

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

How can this be implemented?

$$
\begin{array}{lll}
Decide(r) \triangleq & \lor & \land\ rmState[r] = \text{"prepared"} \\
 & & \land\ \boxed{\forall s \in RM : rmState[s] \in \{\text{"prepared"}, \text{"committed"}\}} \\
 & & \land\ rmState' = [rmState\ \text{EXCEPT}\ ![r] = \text{"committed"}] \\
 & \lor & \land\ rmState[r] \in \{\text{"working"}, \text{"prepared"}\} \\
 & & \land\ \boxed{\forall s \in RM : rmState[s] \neq \text{"committed"}} \\
 & & \land\ rmState' = [rmState\ \text{EXCEPT}\ ![r] = \text{"aborted"}]
\end{array}
$$

$Decide(r)$ 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?

Whether a $Decide$ of r 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?

$$
\begin{array}{lcll}
Decide(r) & \triangleq & \vee & \wedge\ rmState[r] = \text{“prepared”} \\
 & & & \wedge\ \forall s \in RM : rmState[s] \in \{\text{“prepared”}, \text{“committed”}\} \\
 & & & \wedge\ rmState' = [rmState \text{ EXCEPT } ![r] = \text{“committed”}] \\
 & & \vee & \wedge\ rmState[r] \in \{\text{“working”}, \text{“prepared”}\} \\
 & & & \wedge\ \forall s \in RM : rmState[s] \neq \text{“committed”} \\
 & & & \wedge\ rmState' = [rmState \text{ EXCEPT } ![r] = \text{“aborted”}]
\end{array}
$$

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

We don’t care.

Whether a $Decide$ of r 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?

We don’t care.

$$
\begin{aligned}
Decide(r) \;\triangleq\; & \lor \land\, rmState[r] = \text{"prepared"} \\
& \quad \land\, \forall\, s \in RM : rmState[s] \in \{\text{"prepared"}, \text{"committed"}\} \\
& \quad \land\, rmState' = [rmState \text{ EXCEPT } ![r] = \text{"committed"}] \\
& \lor \land\, rmState[r] \in \{\text{"working"}, \text{"prepared"}\} \\
& \quad \land\, \forall\, s \in RM : rmState[s] \neq \text{"committed"} \\
& \quad \land\, rmState' = [rmState \text{ EXCEPT } ![r] = \text{"aborted"}]
\end{aligned}
$$

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

We don’t care.

We’re writing a spec of **what** transaction commit should do,

We’re writing a spec of **what** transaction commit should accomplish,

$$
\begin{array}{lll}
Decide(r) & \triangleq & \lor \land rmState[r] = \text{“prepared”} \\
 & & \quad \land \forall s \in RM : rmState[s] \in \{\text{“prepared”}, \text{“committed”}\} \\
 & & \quad \land rmState' = [rmState \text{ EXCEPT } ![r] = \text{“committed”}] \\
 & & \lor \land rmState[r] \in \{\text{“working”}, \text{“prepared”}\} \\
 & & \quad \land \forall s \in RM : rmState[s] \neq \text{“committed”} \\
 & & \quad \land rmState' = [rmState \text{ EXCEPT } ![r] = \text{“aborted”}]
\end{array}
$$

$Decide(r)$ 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.

$$
\begin{array}{lll}
Decide(r) & \triangleq & \lor \land rmState[r] = \text{"prepared"} \\
& & \quad \land canCommit \\
& & \quad \land rmState' = [rmState \ \text{EXCEPT} \ ![r] = \text{"committed"}] \\
& & \lor \land rmState[r] \in \{\text{"working"}, \text{"prepared"}\} \\
& & \quad \land notCommitted \\
& & \quad \land rmState' = [rmState \ \text{EXCEPT} \ ![r] = \text{"aborted"}]
\end{array}
$$

Let's take one more look at the original definition of $Decide$.

$$Decide(r) \triangleq \lor \boxed{\begin{array}{l} \land\ rmState[r] = \text{"prepared"} \\ \land\ canCommit \\ \land\ rmState' = [rmState\ \text{EXCEPT}\ ![r] = \text{"committed"}] \end{array}} \\ \lor \boxed{\begin{array}{l} \land\ rmState[r] \in \{\text{"working"}, \text{"prepared"}\} \\ \land\ notCommitted \\ \land\ rmState' = [rmState\ \text{EXCEPT}\ ![r] = \text{"aborted"}] \end{array}}$$

Let's take one more look at the original definition of $Decide$.

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

$$DecideC(r) \triangleq \boxed{\begin{array}{l} \land\ rmState[r] = \text{“prepared”} \\ \land\ canCommit \\ \land\ rmState' = [rmState \text{ EXCEPT } ![r] = \text{“committed”}] \end{array}}$$

$$DecideA(r) \triangleq \boxed{\begin{array}{l} \land\ rmState[r] \in \{\text{“working”}, \text{“prepared”}\} \\ \land\ notCommitted \\ \land\ rmState' = [rmState \text{ EXCEPT } ![r] = \text{“aborted”}] \end{array}}$$

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

$$
\begin{aligned}
DecideC(r) \;&\triangleq\; \land\ rmState[r] = \text{“prepared”} \\
&\;\; \land\ canCommit \\
&\;\; \land\ rmState' = [rmState\ \text{EXCEPT}\ ![r] = \text{“committed”}] \\
DecideA(r) \;&\triangleq\; \land\ rmState[r] \in \{\text{“working”}, \text{“prepared”}\} \\
&\;\; \land\ notCommitted \\
&\;\; \land\ rmState' = [rmState\ \text{EXCEPT}\ ![r] = \text{“aborted”}]
\end{aligned}
$$

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

And in the definition of $TCNext$

$$
\begin{array}{lcl}
DecideC(r) & \triangleq & \land\ rmState[r] = \text{“prepared”} \\
 & & \land\ canCommit \\
 & & \land\ rmState' = [rmState\ \text{EXCEPT}\ ![r] = \text{“committed”}] \\
DecideA(r) & \triangleq & \land\ rmState[r] \in \{\text{“working”}, \text{“prepared”}\} \\
 & & \land\ notCommitted \\
 & & \land\ rmState' = [rmState\ \text{EXCEPT}\ ![r] = \text{“aborted”}]
\end{array}
$$

$$
TCNext \triangleq \exists\, r \in RM : Prepare(r) \lor \boxed{Decide(r)}
$$

And in the definition of $TCNext$ replace $Decide$ of r

$$
\begin{aligned}
DecideC(r) \;\triangleq\; & \land\; rmState[r] = \text{“prepared”} \\
& \land\; canCommit \\
& \land\; rmState' = [rmState \text{ EXCEPT } ![r] = \text{“committed”}] \\
DecideA(r) \;\triangleq\; & \land\; rmState[r] \in \{\text{“working”}, \text{“prepared”}\} \\
& \land\; notCommitted \\
& \land\; rmState' = [rmState \text{ EXCEPT } ![r] = \text{“aborted”}]
\end{aligned}
$$

$$
TCNext \;\triangleq\; \exists\, r \in RM : Prepare(r) \lor DecideC(r) \lor DecideA(r)
$$

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

$$
\begin{array}{lcl}
DecideC(r) & \triangleq & \land\ rmState[r] = \text{"prepared"} \\
 & & \land\ canCommit \\
 & & \land\ rmState' = [rmState\ \text{EXCEPT}\ ![r] = \text{"committed"}] \\
DecideA(r) & \triangleq & \land\ rmState[r] \in \{\text{"working"}, \text{"prepared"}\} \\
 & & \land\ notCommitted \\
 & & \land\ rmState' = [rmState\ \text{EXCEPT}\ ![r] = \text{"aborted"}]
\end{array}
$$

$$TCNext \triangleq \exists\, r \in RM : Prepare(r) \lor DecideC(r) \lor DecideA(r)$$

There are many ways to decompose a next-state formula into subformulas.

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.

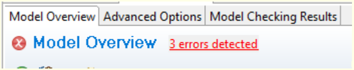
CHECKING THE SPEC

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

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

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

The Toolbox reports 3 errors.

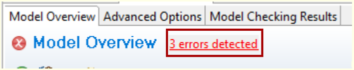


In the Toolbox, create a new model for the $TCommit$ spec.

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

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

The Toolbox reports 3 errors.



Click here

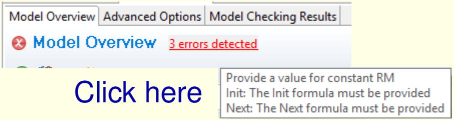
In the Toolbox, create a new model for the $TCommit$ spec.

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

Clicking here, raises

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

The Toolbox reports 3 errors.



Click here
for a list of errors.

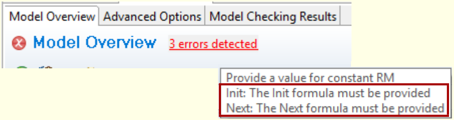
In the Toolbox, create a new model for the $TCommit$ spec.

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

Clicking here, raises this report.

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

The Toolbox reports 3 errors.

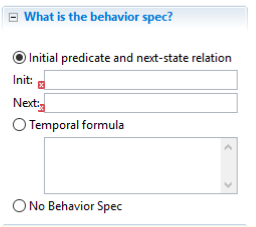


In the Toolbox, create a new model for the $TCommit$ spec.

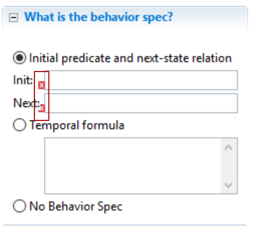
The Toolbox reports that it found three errors in the model.

Clicking here, raises this report.

These two errors occur . . .

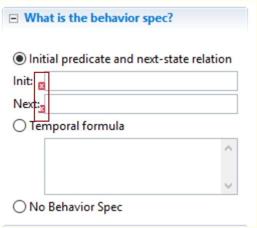

What is the behavior spec?
Initial predicate and next-state relation
Init:
Next:
Temporal formula
No Behavior Spec

here.



here.

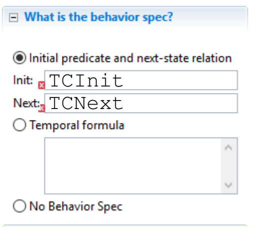
They’re indicated by these little red Xs.



here.

They're indicated by these little red Xs.

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

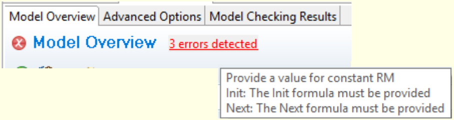


here.

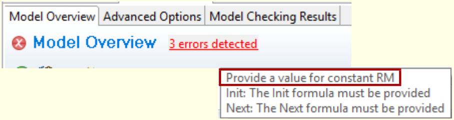
They're indicated by these little red Xs.

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 .

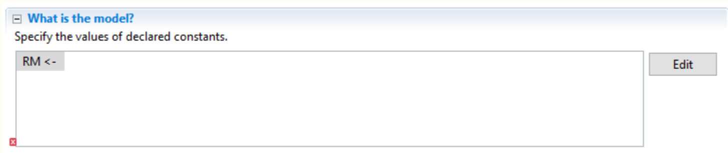
Enter them now.



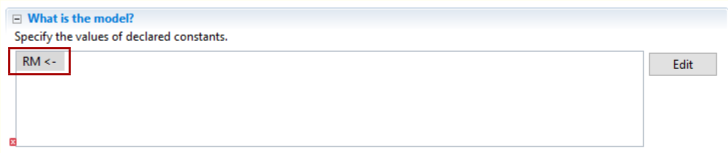
This error tells us that the model has to provide a value for the declared constant RM.



This error tells us that the model has to provide a value for the declared constant RM.



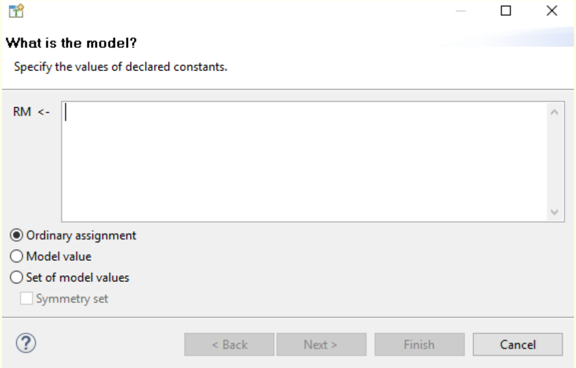
Go to the *What is the model?* area.



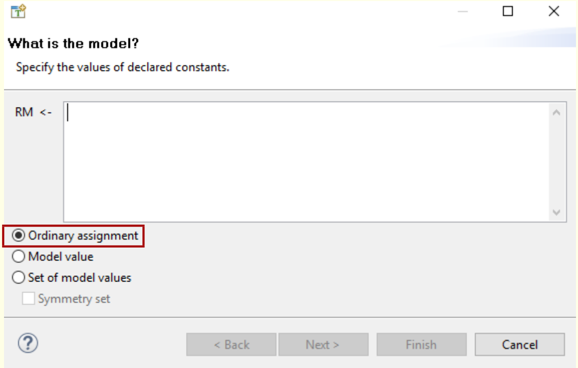
Go to the *What is the model?* area.

And double-click on RM.

[slide 253]

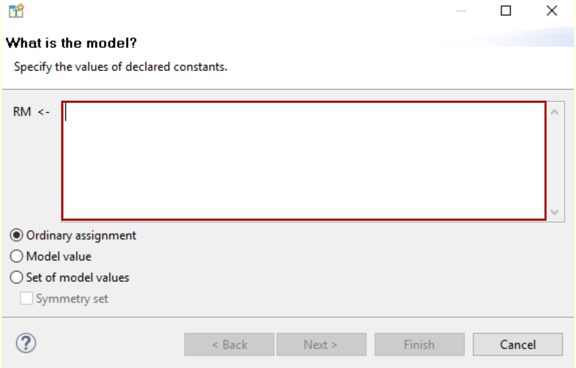


We now tell the Toolbox what value the model should assign to RM.



We now tell the Toolbox what value the model should assign to RM.

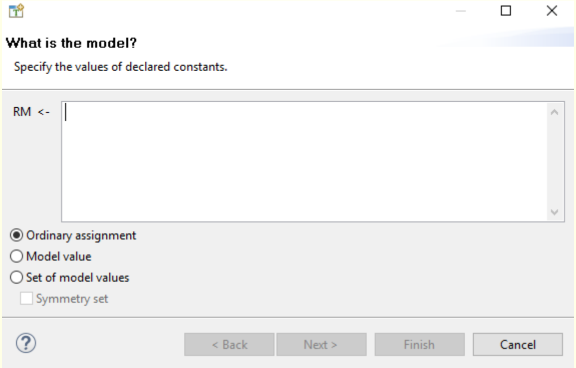
Make sure *Ordinary assignment* is selected.



We now tell the Toolbox what value the model should assign to RM.

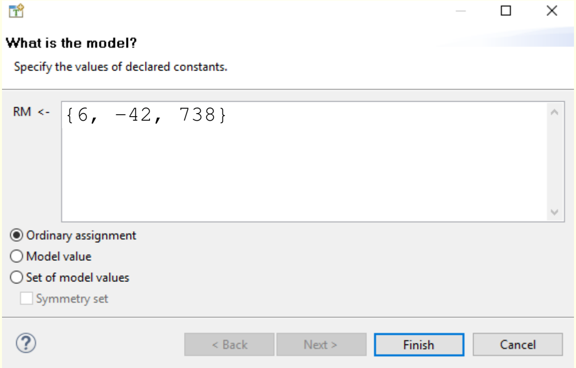
Make sure *Ordinary assignment* is selected.

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.

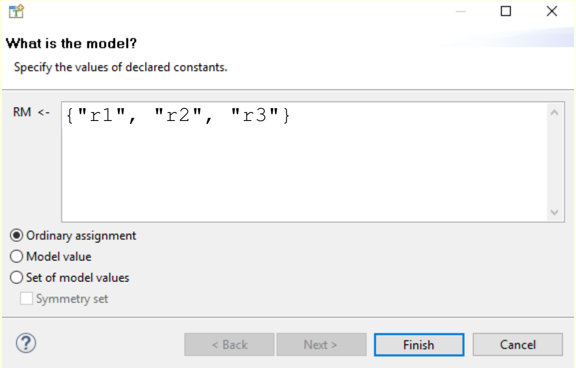


You should usually start with the smallest possible model, which in this case means letting the set RM have only a single element.

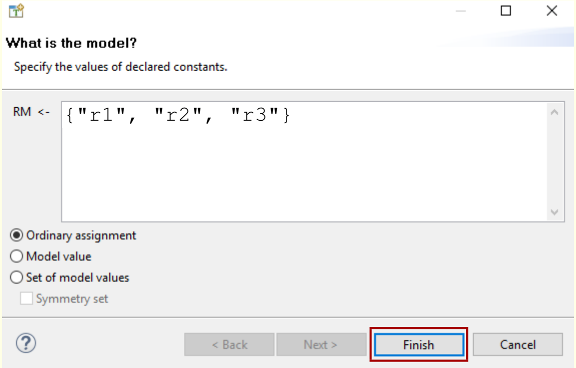
But this spec is so simple, let's make it a set of three elements. The actual elements don't matter.

We could let it be a set of 3 integers.

[slide 258]



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

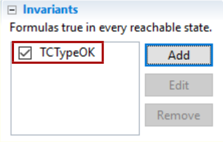
We first check that the spec is type correct

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.

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.

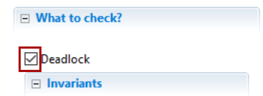
As in $SimpleProgram$, we have to tell TLC not to check for deadlock.

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

As we saw in the $SimpleProgram$ spec of the third video, this means we have to tell TLC not to check for deadlock.

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

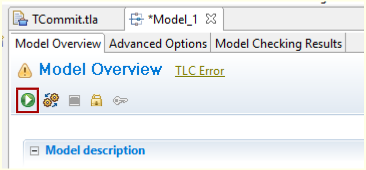
As in $SimpleProgram$, we have to tell TLC not to check for deadlock.



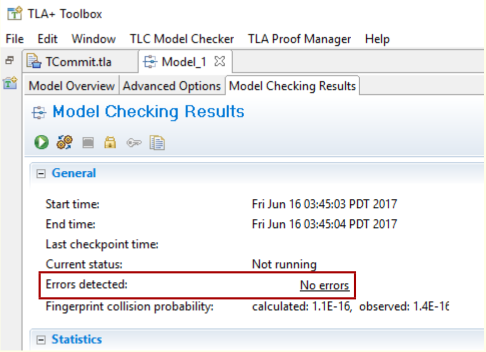
So, uncheck this box

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.

Be Suspicious of Success

Always be suspicious of success.

Be Suspicious of Success

Statistics

State space progress (click column header for graph)

Time	Diameter	States Found	Distinct States	Queue Size
2017-06-16 04:04:21	7	94	34	0

Coverage at 2017-06-16 04:04:21

Module	Location	Count
TCommit	line 39, col 18 to line 39, col 62	27
TCommit	line 43, col 21 to line 43, col 66	12
TCommit	line 46, col 21 to line 46, col 64	54

Always be suspicious of success.

Check the statistics of the TLC run.

Be Suspicious of Success

Statistics

State space progress (click column header for graph)

Time	Diameter	States Found	Distinct States	Queue Size
2017-06-16 04:04:21	7	94	$\boxed{34}$	0

Coverage at 2017-06-16 04:04:21

Module	Location	Count
TCommit	line 39, col 18 to line 39, col 62	27
TCommit	line 43, col 21 to line 43, col 66	12
TCommit	line 46, col 21 to line 46, col 64	54

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?

Be Suspicious of Success

⊟ **Statistics**

State space progress (click column header for graph)

Time	Diameter	States Found	Distinct States	Queue Size
2017-06-16 04:04:21	7	94	34	0

Coverage at 2017-06-16 04:04:21

Module	Location	Count
TCommit	line 39, col 18 to line 39, col 62	27
TCommit	line 43, col 21 to line 43, col 66	12
TCommit	line 46, col 21 to line 46, col 64	54

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.

Be Suspicious of Success

Statistics

State space progress (click column header for graph)

Time	Diameter	States Found	Distinct States	Queue Size
2017-06-16 04:04:21	7	94	34	0

Coverage at 2017-06-16 04:04:21

Module	Location	Count
TCommit	line 39, col 18 to line 39, col 62	27
TCommit	line 43, col 21 to line 43, col 66	12
TCommit	line 46, col 21 to line 46, col 64	54

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

Be Suspicious of Success

Statistics

State space progress (click column header for graph)

Time	Diameter	States Found	Distinct States	Queue Size
2017-06-16 04:04:21	7	94	34	0

Coverage at 2017-06-16 04:04:21

Module	Location	Count
TCommit	line 39, col 18 to line 39, col 62	27
TCommit	line 43, col 21 to line 43, col 66	12
TCommit	line 46, col 21 to line 46, col 64	54

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 $TCommit$ is:

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

Expressed by formula $TCConsistent$.

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.

This condition is expressed by formula $TCConsistent$ that's defined in the module as follows.

$TCConsistent \triangleq$

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

$$TCConsistent \triangleq$$
$$\forall\, r1,\ r2 \in RM :$$

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

$$TCConsistent \triangleq$$
$$\forall\, r1,\ r2 \in RM\ :$$

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:

$$TCConsistent \triangleq$$
$$\forall r1, r2 \in RM : \neg$$

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:

It is not true that

$$TCConsistent \triangleq$$
$$\forall\, r1,\, r2 \in RM : \boxed{\neg}$$

written ! in C

This negation operator is written as exclamation point in C

$$TCConsistent \triangleq$$
$$\forall\, r1,\, r2 \in RM : \boxed{\neg}$$

written ~ in ASCII

This negation operator is written as exclamation point in C

In TLA+ its written as tilde.

$$TCConsistent \triangleq$$
$$\forall\, r1,\, r2 \in RM : \neg$$

This negation operator is written as exclamation point in C

In TLA+ its written as tilde.

So, $TCConsistent$ asserts that for all $r1$ and $r2$ in RM it's not true that

$$
\begin{aligned}
&TCConsistent \triangleq \\
&\quad \forall\, r1,\ r2 \in RM : \neg \land rmState[r1] = \text{“aborted”} \\
&\qquad\qquad\qquad\qquad\quad\ \ \land rmState[r2] = \text{“committed”}
\end{aligned}
$$

This negation operator is written as exclamation point in C

In TLA+ its written as tilde.

So, $TCConsistent$ asserts that for all $r1$ and $r2$ in RM it's not true that

$rmState$ of $r1$ equals aborted and $rmState$ of $r2$ equals $committed$.

$$
\begin{aligned}
&TCConsistent \triangleq \\
&\quad \forall\, r1,\, r2 \in RM : \neg \land rmState[r1] = \text{"aborted"} \\
&\qquad\qquad\qquad\qquad\qquad\;\; \land rmState[r2] = \text{"committed"}
\end{aligned}
$$

Add the invariant $TCConsistent$.

Add the invariant $TCConsistent$ to the model.

$$TCConsistent \triangleq \forall r1, r2 \in RM : \neg \land rmState[r1] = \text{"aborted"} \\ \land rmState[r2] = \text{"committed"}$$

Add the invariant $TCConsistent$.

Run TLC on the model.

Add the invariant $TCConsistent$ to the model.

And run TLC on the model.

$$TCConsistent \triangleq \forall r1, r2 \in RM : \neg \land rmState[r1] = \text{"aborted"} \\ \land rmState[r2] = \text{"committed"}$$

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 *forall* and *exists* extends as far as possible.

For example, this expression

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 *forall* and *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 *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

or similar brackets or braces

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

The expression

$$\begin{array}{l} \wedge \\ \wedge\ \forall\, x \in S : \ \ldots \\ \wedge \end{array}$$

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

$$\begin{array}{l} \wedge \\ \wedge (\forall x \in S : \ \ldots) \\ \wedge \end{array}$$

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

$$\begin{aligned} & \forall x \in S \ : \ \ldots \\ \wedge \ & \forall x \in T \ : \ \ldots \end{aligned}$$

This expression

For example, this expression

$$\begin{aligned} & \forall x \in S : (\ldots \\ \wedge\ & \forall x \in T : \ldots) \end{aligned}$$

This expression is parsed like this

For example, this expression is parsed as if these parentheses were added,

$$\begin{array}{l} \forall x \in S \, : \, (\ldots \\ \qquad\qquad \land \; \forall x \in T \, : \, \ldots) \end{array}$$

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.

$$\forall x \in S : (\ldots \\ \land \forall \boxed{x} \in T : \ldots)$$

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

The expression is illegal because x is declared here

So the expression is illegal because this x, which is declared in the inner forall

$$\forall \boxed{x} \in S : (\ldots$$
$$\land \ \forall x \in T : \ldots)$$

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 redeclare an identifier that's already declared.

COMMENTS

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 \* 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 \* An end of line comment.
```

TLA+ provides two kinds of comments.

An end of line comment begins with backslash asterisk.

TLA$^+$ has two kinds of comments.

```
x' = x + 1 \* An end of line comment.

x' = x + (* 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

TLA$^+$ has two kinds of comments.

```
x' = x + 1 \* An end of line comment.

x' = x + (* 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

```
x' = x + 1 (***************************)
           (* This is a boxed comment.  *)
           (* It looks very nice when   *)
           (* it's pretty-printed.      *)
           (***************************)
```

$x' = x + 1$ 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.       *)
           (****************************)
```

$x' = x + 1$ 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

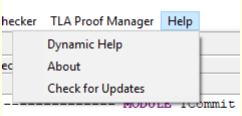
To find out how to type boxed comments,

Typing Boxed Comments

See the Toolbox's Help.

To find out how to type boxed comments, See the Toolbox's Help pages.

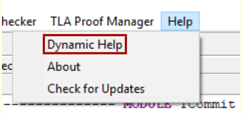
Typing Boxed Comments



To find out how to type boxed comments, See the Toolbox's Help pages.

To do that, click help

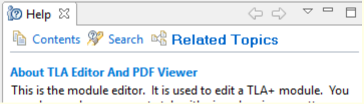
Typing Boxed Comments



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

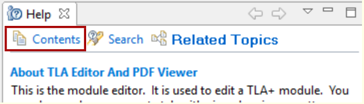


To find out how to type boxed comments, See the Toolbox's Help pages.

To do that, click help then Dynamic Help.

Then

Typing Boxed Comments

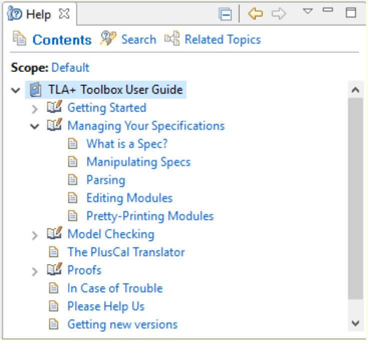


To find out how to type boxed comments, See the Toolbox's Help pages.

To do that, click help then Dynamic Help.

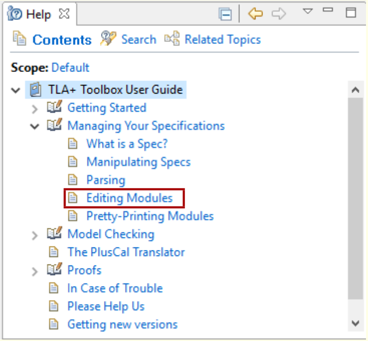
Then Click Contents.

Typing Boxed Comments



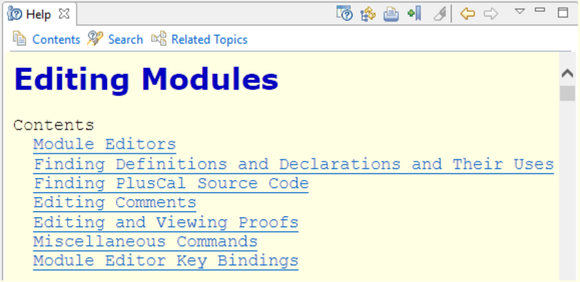
Open the Toolbox User Guide

Typing Boxed Comments



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

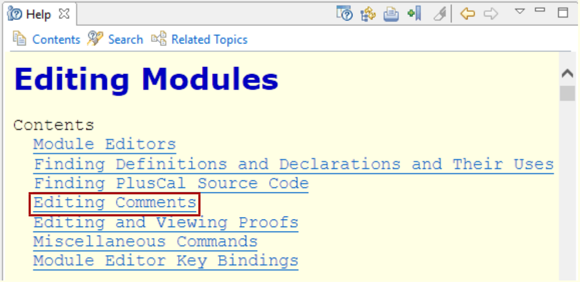
Typing Boxed Comments



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

On that page

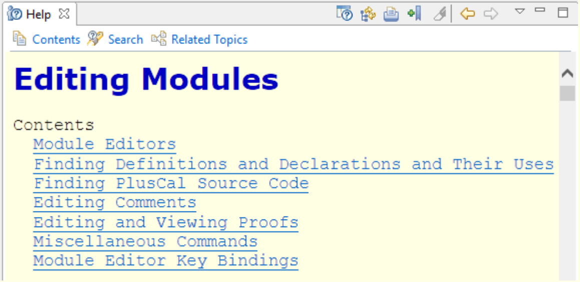
Typing Boxed Comments



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.

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.

A separator line:

\-

Pretty printed like:

Purely decorative.

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