TLA$^+$ Video Course – Lecture 6

Leslie Lamport

TWO-PHASE 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 6
Transaction Commit

This lecture is about the two-phase commit protocol, a very simple, popular algorithm for implementing transaction commit.

Following in the footsteps of Jim Gray, I introduce the protocol by examining a wedding and the role of the minister.

But first, I'll describe the TLA+ notation for an important data type: records.

[slide 2]

RECORDS

We start with the TLA+ notation for records.

The definition

$$r \triangleq [prof \mapsto \text{“}Fred\text{”}, num \mapsto 42]$$

defines r to be a record with two fields

This definition of r defines it to be a record with two fields named

The definition

$$r \triangleq [prof \mapsto \text{“}Fred\text{”}, num \mapsto 42]$$

defines r to be a record with two fields

$prof$

This definition of r defines it to be a record with two fields named $prof$ and num.

The definition

$$r \triangleq [prof \mapsto \text{“}Fred\text{”}, num \mapsto 42]$$

defines r to be a record with two fields $prof$ and num.

This definition of r defines it to be a record with two fields named $prof$ and num.

The definition

$$r \triangleq [prof \mapsto \text{“}Fred\text{”}, num \mapsto 42]$$

defines r to be a record with two fields $prof$ and num.

The values of its two fields are

This definition of r defines it to be a record with two fields named $prof$ and num.

The values of the two fields can be written as

The definition

$$r \triangleq [prof \mapsto \text{“}Fred\text{”},\ num \mapsto 42]$$

defines r to be a record with two fields $prof$ and num.

The values of its two fields are

$$r.prof = \text{“}Fred\text{”}$$

This definition of r defines it to be a record with two fields named $prof$ and num.

The values of the two fields can be written as r dot $prof$, which equals the string “$Fred$”

The definition

$$r \triangleq [\,prof \mapsto \text{“}Fred\text{”},\ num \mapsto 42\,]$$

defines r to be a record with two fields $prof$ and num.

The values of its two fields are

$$r.prof = \text{“}Fred\text{”} \quad \text{and} \quad r.num = 42$$

This definition of r defines it to be a record with two fields named $prof$ and num.

The values of the two fields can be written as r dot $prof$, which equals the string “$Fred$” and r.num, which equals 42 .

The definition

$$r \triangleq [\,prof \mapsto \text{“}Fred\text{”},\ num \mapsto 42\,]$$

defines r to be a record with two fields $prof$ and num .

A record corresponds to a struct in C,

This definition of r defines it to be a record with two fields named $prof$ and num.

The values of the two fields can be written as r dot $prof$, which equals the string “$Fred$” and r.num, which equals 42 .

A record corresponds roughly to a Struct in C,

The definition

$$r \triangleq [\, prof \mapsto \text{"}Fred\text{"},\ num \mapsto 42 \,]$$

defines r to be a record with two fields $prof$ and num.

A record corresponds to a struct in C, except

$$[\, prof \mapsto \text{"}Fred\text{"},\ num \mapsto 42 \,] \;=\; [\, num \mapsto 42,\ prof \mapsto \text{"}Fred\text{"} \,]$$

This definition of r defines it to be a record with two fields named $prof$ and num.

The values of the two fields can be written as r dot $prof$, which equals the string "$Fred$" and r.num, which equals 42 .

A record corresponds roughly to a Struct in C, except that changing the orders of the fields makes no difference.

[slide 11]

$$[\, prof : \{\text{"}Fred\text{"}, \text{"}Ted\text{"}, \text{"}Ned\text{"}\},\ num : 0 \,..\, 99 \,]$$

This is the TLA+ notation for

$[\,prof : \{\text{"}Fred\text{"}, \text{"}Ted\text{"}, \text{"}Ned\text{"}\},\ num : 0\,..\,99\,]$

is the set of all records

$[\,prof \mapsto \ldots,\ num \mapsto \ldots\,]$

with

This is the TLA+ notation for the set of all records of this form with

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

is the set of all records

$[\, prof \mapsto \boxed{\ldots},\ num \mapsto \ldots\,]$

with $prof$ field

This is the TLA+ notation for the set of all records of this form with

the value of its $prof$ field

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

is the set of all records

$[\, prof \mapsto \ldots,\ num \mapsto \ldots \,]$

with $prof$ field in $\{\text{“}Fred\text{”}, \text{“}Ted\text{”}, \text{“}Ned\text{”}\}$

This is the TLA+ notation for the set of all records of this form with the value of its $prof$ field an element of this set

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

is the set of all records

$[prof \mapsto \ldots, \; num \mapsto \ldots]$

with $prof$ field in $\{“Fred”, “Ted”, “Ned”\}$

num field

This is the TLA+ notation for the set of all records of this form with

the value of its $prof$ field an element of this set

and the value of its num field

$[prof : \{\text{"}Fred\text{"}, \text{"}Ted\text{"}, \text{"}Ned\text{"}\}, \; num : 0\,..\,99]$

is the set of all records

$[prof \mapsto \ldots, \; num \mapsto \ldots]$

with $prof$ field in $\{\text{"}Fred\text{"}, \text{"}Ted\text{"}, \text{"}Ned\text{"}\}$

num field in $0\,..\,99$

This is the TLA+ notation for the set of all records of this form with

the value of its $prof$ field an element of this set

and the value of its num field an element of this set

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

is the set of all records

$[\mathit{prof} \mapsto \ldots,\ \mathit{num} \mapsto \ldots]$

with prof field in $\{\text{“}\mathit{Fred}\text{”}, \text{“}\mathit{Ted}\text{”}, \text{“}\mathit{Ned}\text{”}\}$

num field in $0\,..\,99$

So $[\mathit{prof} \mapsto \text{“}\mathit{Ned}\text{”}, \mathit{num} \mapsto 24]$

This is the TLA+ notation for the set of all records of this form with

the value of its prof field an element of this set

and the value of its num field an element of this set

So this record

$[\,prof : \{\text{"}Fred\text{"}, \text{"}Ted\text{"}, \text{"}Ned\text{"}\},\ num : 0\,..\,99\,]$

is the set of all records

$[\,prof \mapsto \ldots,\ num \mapsto \ldots\,]$

with $prof$ field in $\{\text{"}Fred\text{"}, \text{"}Ted\text{"}, \text{"}Ned\text{"}\}$

num field in $0\,..\,99$

So $[\,prof \mapsto \text{"}Ned\text{"},\ num \mapsto 24\,]$ is in this set.

This is the TLA+ notation for the set of all records of this form with

the value of its $prof$ field an element of this set

and the value of its num field an element of this set

So this record is in this set.

$[\, prof \mapsto \text{“}Fred\text{”},\ num \mapsto 42 \,]$

is a function

This record is actually a function,

$[\, prof \mapsto \text{“}Fred\text{”},\ num \mapsto 42 \,]$

is a function f

This record is actually a function, let's call it f,

$[\,prof \mapsto \text{“}Fred\text{”},\; num \mapsto 42\,]$

is a function f with domain $\{\text{“}prof\text{”}, \text{“}num\text{”}\}$

This record is actually a function, let's call it f, whose domain is the set containing the two strings $prof$ and num.

$[\, prof \mapsto \text{“}Fred\text{”},\ num \mapsto 42 \,]$

is a function f with domain $\{\text{“}prof\text{”}, \text{“}num\text{”}\}$

such that $f[\text{“}prof\text{”}] = \text{“}Fred\text{”}$

This record is actually a function, let's call it f, whose domain is the set containing the two strings $prof$ and num. such that f of the string $prof$ equals the string “$Fred$”

$[\,prof \mapsto \text{“}Fred\text{”},\; num \mapsto 42\,]$

is a function f with domain $\{\text{“}prof\text{”}, \text{“}num\text{”}\}$

such that $f[\text{“}prof\text{”}] = \text{“}Fred\text{”}$

$f[\text{“}num\text{”}] = 42$

This record is actually a function, let's call it f, whose domain is the set containing the two strings $prof$ and num. such that f of the string $prof$ equals the string "$Fred$" and f of the string num equals the number 42.

$[\, prof \mapsto \text{"}Fred\text{"},\ num \mapsto 42 \,]$

is a function f with domain $\{\text{"}prof\text{"}, \text{"}num\text{"}\}$

such that $f[\text{"}prof\text{"}] = \text{"}Fred\text{"}$

$f[\text{"}num\text{"}] = 42$

$f.prof$ is an abbreviation for $f[\text{"}prof\text{"}]$

This record is actually a function, let's call it f, whose domain is the set containing the two strings $prof$ and num. such that f of the string $prof$ equals the string "$Fred$" and f of the string num equals the number 42.

f dot $prof$ is just an abbreviation for f of the string $prof$.

$[\,prof \mapsto \text{“}Fred\text{”},\ num \mapsto 42\,]$

is a function f with domain $\{\text{“}prof\text{”}, \text{“}num\text{”}\}$

such that $f[\text{“}prof\text{”}] = \text{“}Fred\text{”}$

$f[\text{“}num\text{”}] = 42$

$$[f \text{ EXCEPT } ![\text{“}prof\text{”}] = \text{“}Red\text{”}\,]$$

This EXCEPT expression equals the record that’s the same as f except its $prof$ field equals the string Red.

$[\, prof \mapsto \text{“}Fred\text{”},\ num \mapsto 42\,]$

is a function f with domain $\{\text{“}prof\text{”}, \text{“}num\text{”}\}$

such that $f[\text{“}prof\text{”}] = \text{“}Fred\text{”}$

$f[\text{“}num\text{”}] = 42$

$[f \text{ EXCEPT } ![\text{“}prof\text{”}] = \text{“}Red\text{”}\,]$

can be abbreviated as

$[f \text{ EXCEPT } !.prof = \text{“}Red\text{”}\,]$

This EXCEPT expression equals the record that's the same as f except its $prof$ field equals the string Red.

We can abbreviate the EXCEPT by writing

$[\,prof \mapsto \text{"}Fred\text{"},\ num \mapsto 42\,]$

is a function f with domain $\{\text{"}prof\text{"}, \text{"}num\text{"}\}$

such that $f[\text{"}prof\text{"}] = \text{"}Fred\text{"}$

$f[\text{"}num\text{"}] = 42$

$$[f \text{ EXCEPT } ![\text{"}prof\text{"}] = \text{"}Red\text{"}]$$

can be abbreviated as

$$[f \text{ EXCEPT } !.prof = \text{"}Red\text{"}]$$

This EXCEPT expression equals the record that's the same as f except its $prof$ field equals the string Red.

We can abbreviate the EXCEPT by writing bang dot $prof$ instead of

$[\, prof \mapsto \text{“}Fred\text{”},\ num \mapsto 42 \,]$

is a function f with domain $\{\text{“}prof\text{”}, \text{“}num\text{”}\}$

such that $f[\text{“}prof\text{”}] = \text{“}Fred\text{”}$

$f[\text{“}num\text{”}] = 42$

$[f \text{ EXCEPT } ![\text{“}prof\text{”}] = \text{“}Red\text{”}]$

can be abbreviated as

$[f \text{ EXCEPT } !.prof = \text{“}Red\text{”}]$

This EXCEPT expression equals the record that's the same as f except its $prof$ field equals the string Red.

We can abbreviate the EXCEPT by writing bang dot $prof$ instead of bang of the string $prof$.

WEDDINGS

We now get to the two-phase commit protocol. As in the previous lecture, we begin with weddings.

What Transaction Commit Describes

Henry

Anne

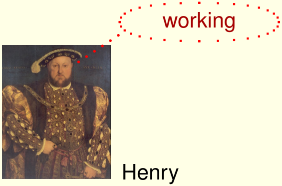
Transaction commit describes the states of the bride and groom.

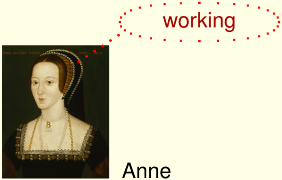
What Transaction Commit Describes

Transaction commit describes the states of the bride and groom.

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

What Transaction Commit Describes





Transaction commit describes the states of the bride and groom.

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

Except that Transaction Commit calls that state *working*. In a successful wedding, both reach the prepared state

What Transaction Commit Describes

prepared

Henry

working

Anne

They then each reach

What Transaction Commit Describes

They then each reach

What Transaction Commit Describes

They then each reach

the committed state.

What Transaction Commit Describes

They then each reach

the committed state.

TwoPhase Commit Adds the Minister

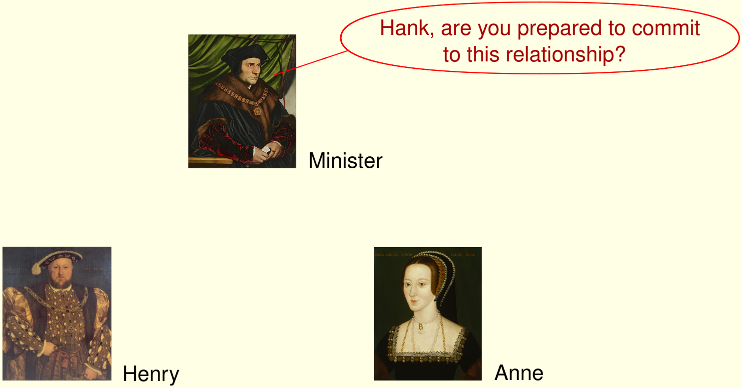
Minister

Henry

Anne

Two-phase commit adds the minister to help implement those state changes. He does that by communicating with the bride and groom.

TwoPhase Commit Adds the Minister



Two-phase commit adds the minister to help implement those state changes. He does that by communicating with the bride and groom.

TwoPhase Commit Adds the Minister

Two-phase commit adds the minister to help implement those state changes. He does that by communicating with the bride and groom.

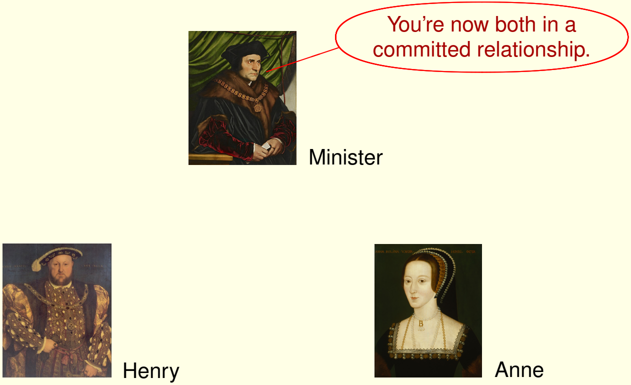
TwoPhase Commit Adds the Minister

Two-phase commit adds the minister to help implement those state changes. He does that by communicating with the bride and groom.

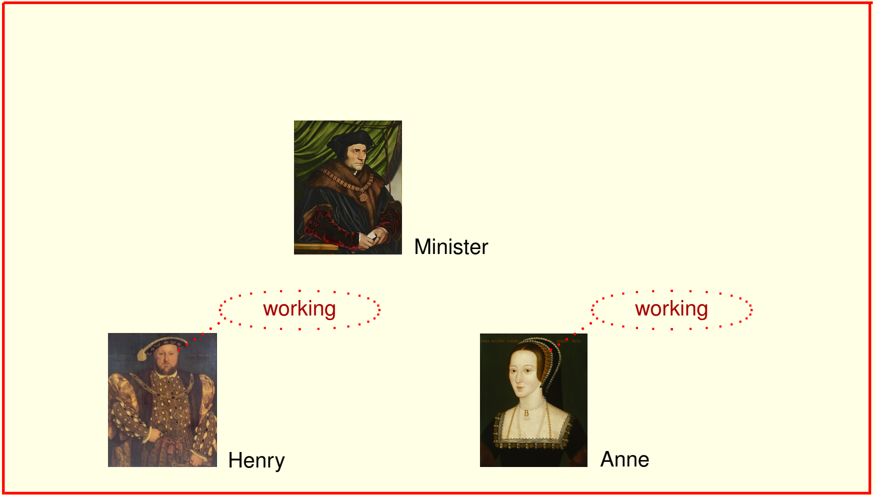
TwoPhase Commit Adds the Minister

Two-phase commit adds the minister to help implement those state changes. He does that by communicating with the bride and groom.

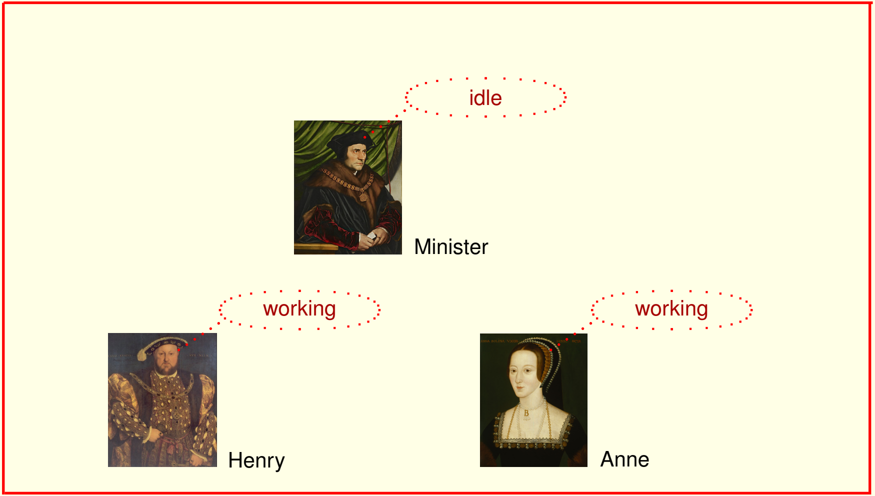
TwoPhase Commit Adds the Minister



Two-phase commit adds the minister to help implement those state changes. He does that by communicating with the bride and groom.



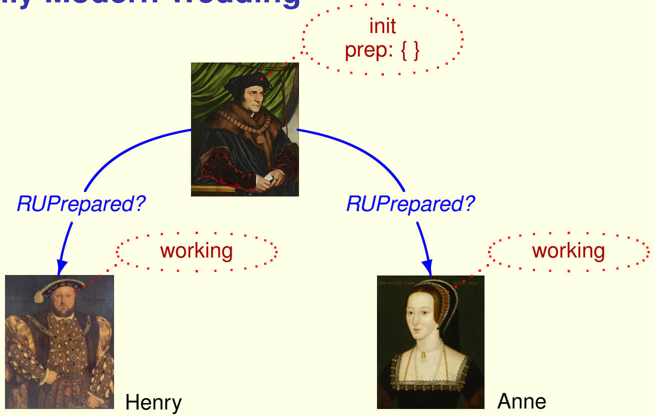
In addition to the states of the bride and groom,



In addition to the states of the bride and groom, there's the minister's state, which initially is $idle$.

In a really modern wedding, the parties communicate by texting.

A Really Modern Wedding

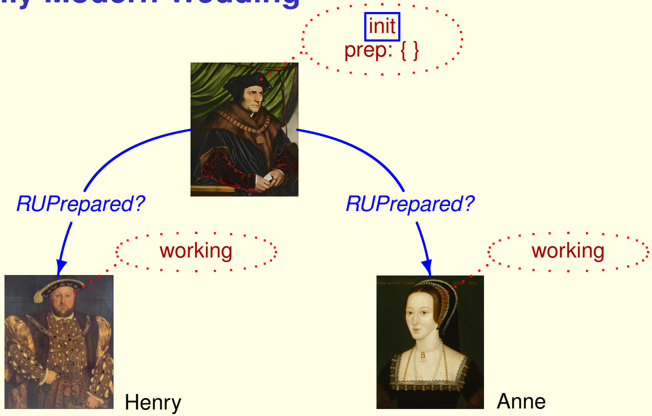


In addition to the states of the bride and groom, there's the minister's state, which initially is $idle$.

In a really modern wedding, the parties communicate by texting.

In addition to sending the "are you prepared" text, the minister's state changes to

A Really Modern Wedding

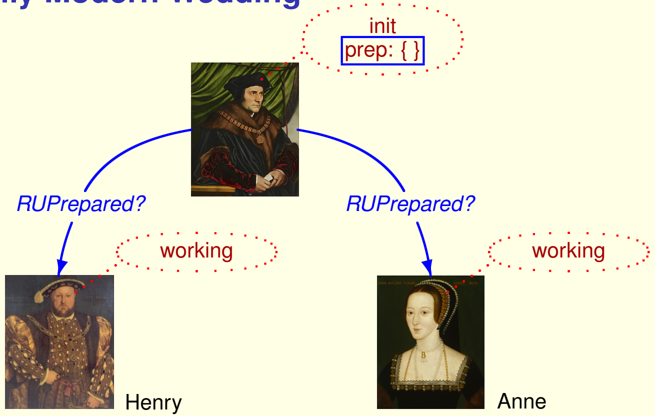


In addition to the states of the bride and groom, there's the minister's state, which initially is $idle$.

In a really modern wedding, the parties communicate by texting.

In addition to sending the "are you prepared" text, the minister's state changes to an $init$ state

A Really Modern Wedding

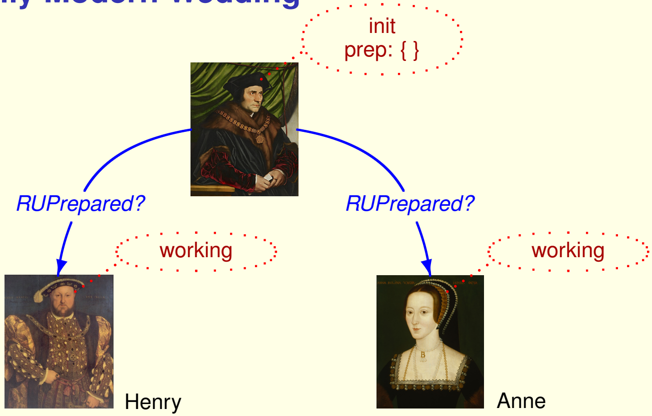


In addition to the states of the bride and groom, there's the minister's state, which initially is *idle*.

In a really modern wedding, the parties communicate by texting.

In addition to sending the "are you prepared" text, the minister's state changes to an *init* state in which the set of participants who he knows are prepared is empty.

A Really Modern Wedding

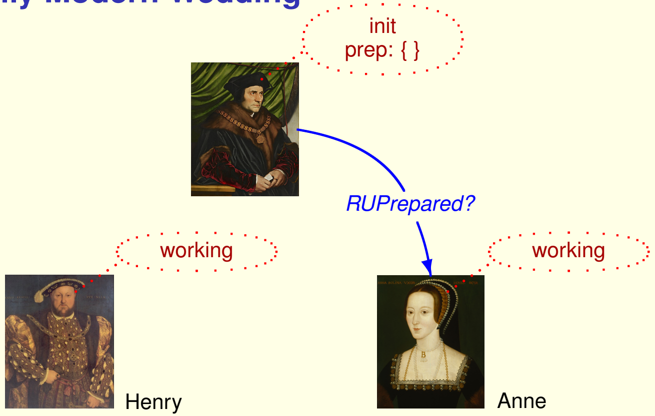


In addition to the states of the bride and groom, there's the minister's state, which initially is $idle$.

In a really modern wedding, the parties communicate by texting.

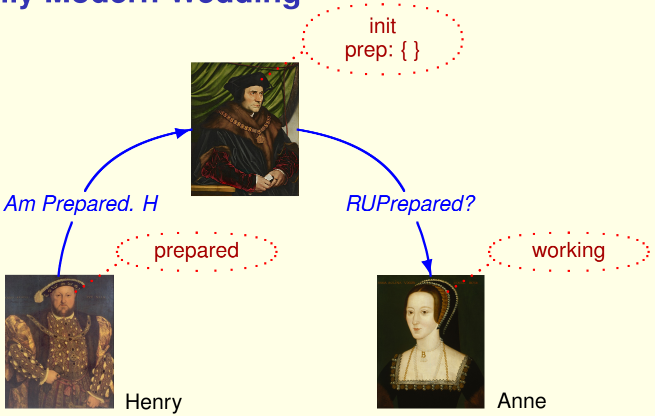
In addition to sending the "are you prepared" text, the minister's state changes to an $init$ state in which the set of participants who he knows are prepared is empty. Suppose Henry reads his text first

A Really Modern Wedding



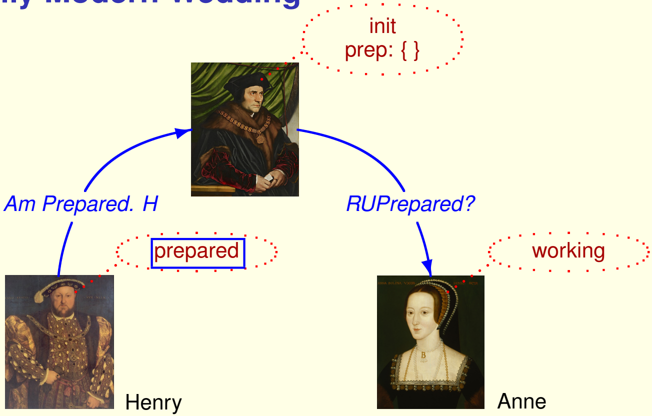
and replies with a text

A Really Modern Wedding



and replies with a text saying he’s prepared,

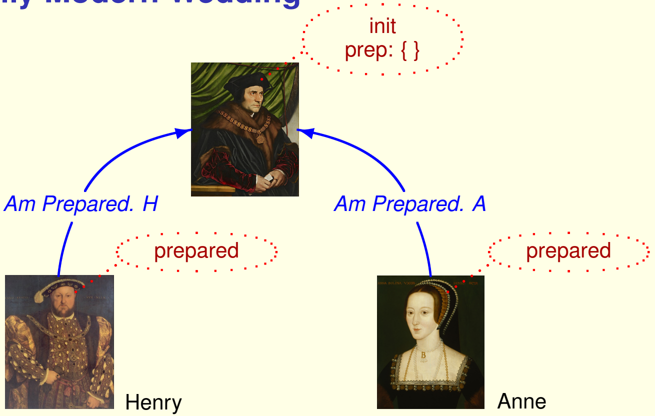
A Really Modern Wedding



and replies with a text saying he's prepared, changing his state to $prepared$.

And suppose Anne then

A Really Modern Wedding

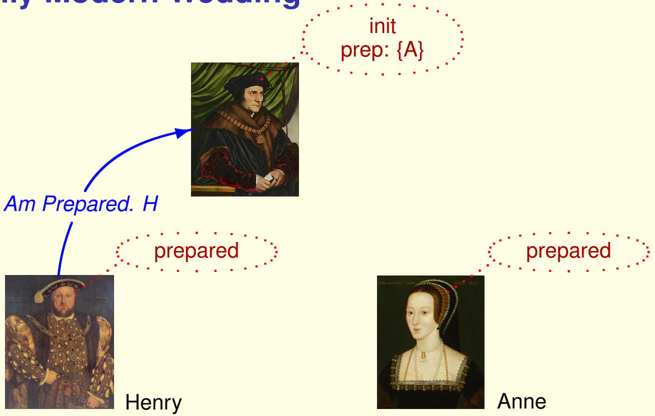


and replies with a text saying he's prepared, changing his state to *prepared*.

And suppose Anne then does the same.

The minister might then receive Anne's text

A Really Modern Wedding



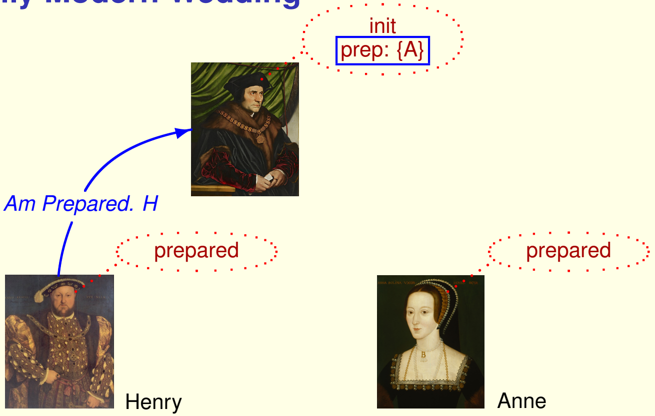
and replies with a text saying he's prepared, changing his state to *prepared*.

And suppose Anne then does the same.

The minister might then receive Anne's text

updating his state

A Really Modern Wedding



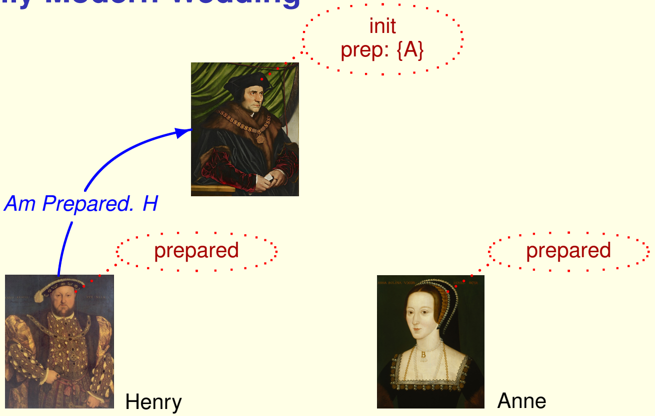
and replies with a text saying he's prepared, changing his state to *prepared*.

And suppose Anne then does the same.

The minister might then receive Anne's text

updating his state because he knows Anne is prepared.

A Really Modern Wedding



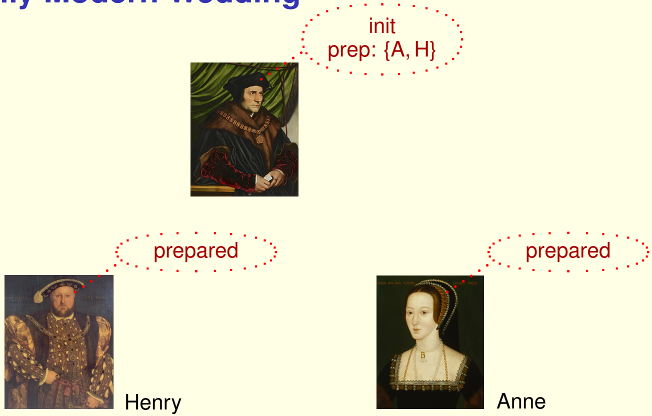
and replies with a text saying he's prepared, changing his state to $prepared$.

And suppose Anne then does the same.

The minister might then receive Anne's text

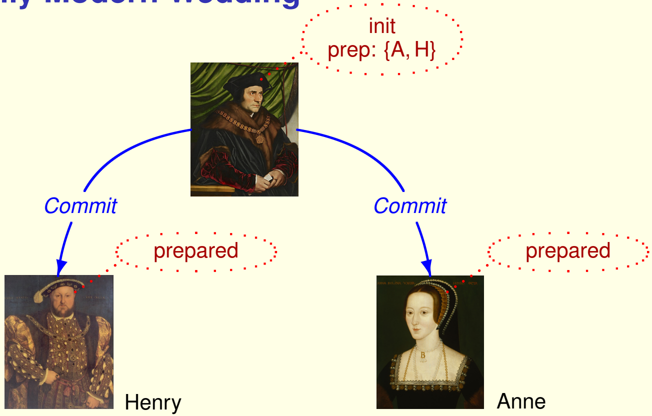
updating his state because he knows Anne is prepared. He similarly receives Henry's text

A Really Modern Wedding



and updates his state.
He can then send a text telling them to commit.

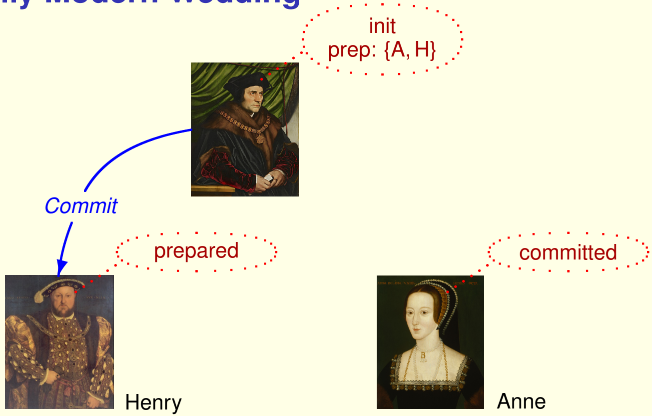
A Really Modern Wedding



and updates his state.
He can then send a text telling them to commit.

Anne might receive his text first,

A Really Modern Wedding

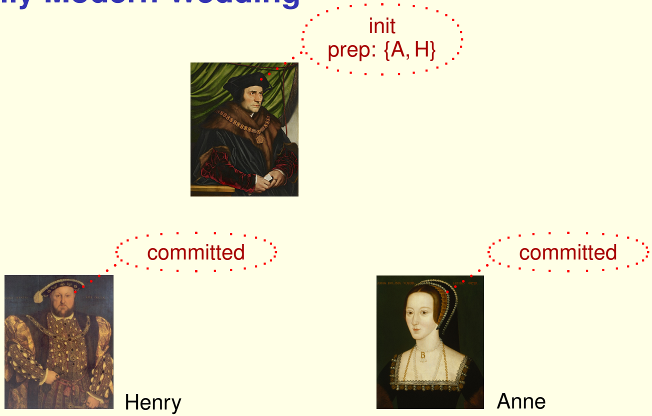


and updates his state.
He can then send a text telling them to commit.

Anne might receive his text first, causing her to become committed.

Henry might then receive his text,

A Really Modern Wedding

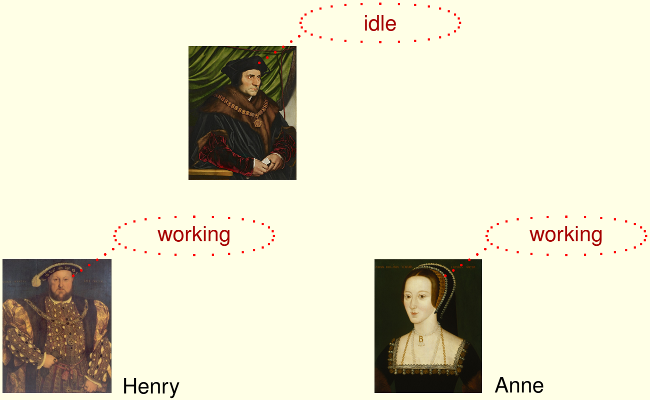


and updates his state.
He can then send a text telling them to commit.

Anne might receive his text first, causing her to become committed.

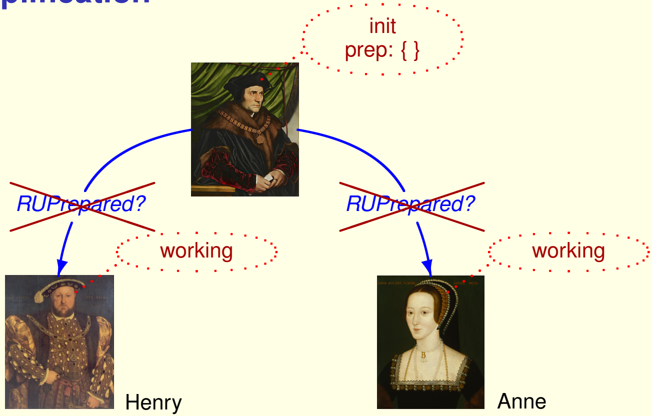
Henry might then receive his text, also becoming committed.

A Simplification



Let's simplify the algorithm a bit.

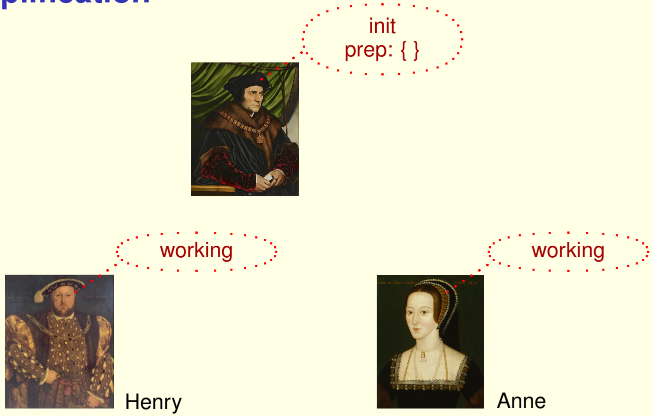
A Simplification



Let's simplify the algorithm a bit.

We eliminate the Minister's first text.

A Simplification



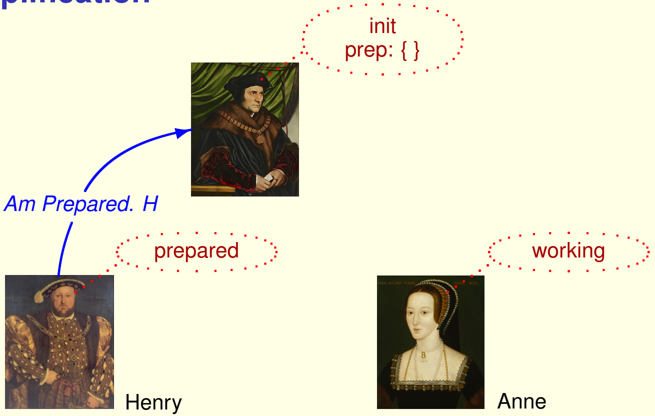
Let’s simplify the algorithm a bit.

We eliminate the Minister’s first text.

Instead we start in this state.

Henry and Anne can send their “I’m prepared” text without hearing from the minister.

A Simplification



For example, Henry might send his “I’m prepared” text first, changing his state to $prepared$.

$RUPrepared?$ message not required by $TCommit$.

The $RUPrepared?$ message is not needed to implement the $TCommit$ spec.

$RUPrepared?$ message not required by $TCommit$.

Simplicity, simplicity, simplicity!

The $RUPrepared?$ message is not needed to implement the $TCommit$ spec.

We want the simplest spec that can catch the errors we're looking for—namely, ones that would cause two-phase commit not to satisfy the $TCommit$ spec.

THE TLA$^+$ SPEC

OK, let's stop looking at pictures and start reading the TLA+ specification.

Stop the video:

– In the Toolbox, create a new module named $TwoPhase$ in the same folder as $TCommit$.

– Copy the body of the spec from the web page and paste it into the module.

First, stop the video and, in the Toolbox, create a new module named $TwoPhase$ in the same folder as module $TCommit$.

Copy the body of the spec from the web page and paste it into the module.

Do it now.

CONSTANT RM

The spec begins by declaring the set RM of resource managers, just like in $TCommit$.

CONSTANT RM

VARIABLES $rmState$

The spec begins by declaring the set RM of resource managers, just like in $TCommit$.

Variable $rmState$ decribes the state of the resource managers, again like in $TCommit$.

CONSTANT RM

VARIABLES $rmState$, $tmState$, $tmPrepared$

The spec begins by declaring the set RM of resource managers, just like in $TCommit$.

Variable $rmState$ decribes the state of the resource managers, again like in $TCommit$.

Variables $tmState$ and $tmPrepared$ describe the state of the minister, who we now call the Transaction Manager.

CONSTANT RM

VARIABLES $rmState$, $tmState$, $tmPrepared$

$tmState$ is this part of the transaction manager's state.

CONSTANT RM

VARIABLES $rmState$, $tmState$, $tmPrepared$

$tmState$ is this part of the transaction manager's state.

And $tmPrepared$ is this part, the set of resource managers he knows are prepared.

CONSTANT RM

VARIABLES $rmState$, $tmState$, $tmPrepared$, $msgs$

$tmState$ is this part of the transaction manager's state.

And $tmPrepared$ is this part, the set of resource managers he knows are prepared.

And m-s-g-s describes the messages that are in transit.

CONSTANT RM

VARIABLES $rmState$, $tmState$, $tmPrepared$, $msgs$

$Messages \triangleq \ldots$

$tmState$ is this part of the transaction manager's state.

And $tmPrepared$ is this part, the set of resource managers he knows are prepared.

And m-s-g-s describes the messages that are in transit.

Next comes a definition that we'll skip over for now.

$TPTypeOK \triangleq$

We then have the type invariant. In this spec, conventional names like $TypeOK$ are prefaced with TP.

$$TPTypeOK \triangleq$$
$$\land\ rmState \in [RM \to \{\text{"working"}, \text{"prepared"}, \text{"committed"}, \text{"aborted"}\}]$$

We then have the type invariant. In this spec, conventional names like $TypeOK$ are prefaced with TP.

As in $TCommit$, the value of variable $rmState$ should be a function from resource managers to this set of four strings.

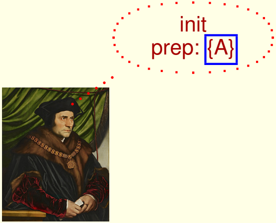
$$
\begin{aligned}
&TPTypeOK \triangleq \\
&\quad \land\ rmState \in [RM \rightarrow \{\text{"working"}, \text{"prepared"}, \text{"committed"}, \text{"aborted"}\}] \\
&\quad \land\ tmState \in \{\text{"init"}, \text{"done"}\}
\end{aligned}
$$

We then have the type invariant. In this spec, conventional names like $TypeOK$ are prefaced with TP.

As in $TCommit$, the value of variable $rmState$ should be a function from resource managers to this set of four strings.

The value of $tmState$ is either $init$ or $done$.

$$
\begin{aligned}
&TPTypeOK \triangleq \\
&\quad \land\ rmState \in [RM \rightarrow \{\text{"working"}, \text{"prepared"}, \text{"committed"}, \text{"aborted"}\}] \\
&\quad \land\ tmState \in \{\text{"init"}, \text{"done"}\} \\
&\quad \land\ tmPrepared \subseteq RM
\end{aligned}
$$



This asserts that $tmPrepared$ is a subset of the set RM of resource managers

$$
\begin{aligned}
&\mathit{TPTypeOK} \triangleq \\
&\quad \land\ \mathit{rmState} \in [\mathit{RM} \rightarrow \{\text{"working"}, \text{"prepared"}, \text{"committed"}, \text{"aborted"}\}] \\
&\quad \land\ \mathit{tmState} \in \{\text{"init"}, \text{"done"}\} \\
&\quad \land\ \mathit{tmPrepared} \subseteq \mathit{RM}
\end{aligned}
$$

`\subseteq`

This asserts that $tmPrepared$ is a subset of the set RM of resource managers

This symbol, typed backslash subset-e-q, is read "is a subset of". The third conjunct means that every element of the set $tmPrepared$ is an element of the set RM.

$$
\begin{aligned}
&TPTypeOK \triangleq \\
&\quad \land rmState \in [RM \rightarrow \{\text{"working"}, \text{"prepared"}, \text{"committed"}, \text{"aborted"}\}] \\
&\quad \land tmState \in \{\text{"init"}, \text{"done"}\} \\
&\quad \land tmPrepared \subseteq RM \\
&\quad \land msgs \subseteq Messages
\end{aligned}
$$

Similarly TPTypeOK also asserts that the value of m-s-g-s is a subset of the set $Messages$.

Sending Messages

The spec must describe sending messages.

A spec of two-phase commit has to describe the sending of messages.

The spec need not describe the actual mechanism by which messages are sent.

[slide 82]

Sending Messages

The spec must describe sending messages.

It should specify only what's required of message passing.

It should describe only what the algorithm requires of message passing.

Since two-phase commit requires no assumptions about the order in which different messages are received, the simplest natural representation

Sending Messages

The spec must describe sending messages.

It should specify only what's required of message passing.

A simple method:

Let $msgs$ be the set of messages currently in transit.

It should describe only what the algorithm requires of message passing.

Since two-phase commit requires no assumptions about the order in which different messages are received, the simplest natural representation is to let m-s-g-s be a single set containing all messages in transit. Receiving a message removes it from the set m-s-g-s.

A Simpler Method

There's a simpler method that's not obvious to most people.

A Simpler Method

Let $msgs$ be the set of all messages ever sent.

There's a simpler method that's not obvious to most people.

It's to let m-s-g-s be the set of all messages that have ever been sent. So the action of receiving a message doesn't remove the message from the set. One advantage is that

A Simpler Method

Let $msgs$ be the set of all messages ever sent.

A single message can be received by multiple processes.

There's a simpler method that's not obvious to most people.

It's to let m-s-g-s be the set of all messages that have ever been sent. So the action of receiving a message doesn't remove the message from the set. One advantage is that

A single message in m-s-g-s can be received by several processes. It also means that

A Simpler Method

Let $msgs$ be the set of all messages ever sent.

A single message can be received by multiple processes.

A process can receive the same message multiple times.

A process can received the same message multiple times.

This can happen with real message passing, and it's useful to know that

A Simpler Method

Let $msgs$ be the set of all messages ever sent.

A single message can be received by
multiple processes.

A process can receive the same message multiple times.

Two-phase commit still works.

A process can received the same message multiple times.

This can happen with real message passing, and it's useful to know that

The two-phase commit protocol still works even if it does happen.

Let's return now to the spec.

$$
\begin{aligned}
&TPTypeOK \triangleq \\
&\quad \land\ rmState \in [RM \rightarrow \{\text{"working"}, \text{"prepared"}, \text{"committed"}, \text{"aborted"}\}] \\
&\quad \land\ tmState \in \{\text{"init"}, \text{"done"}\} \\
&\quad \land\ tmPrepared \subseteq RM \\
&\quad \land\ \boxed{msgs \subseteq Messages}
\end{aligned}
$$

Remember the type assertion for m-s-g-s: that it's a subset of the set Messages.

$$Messages \triangleq [type : \{\text{"Prepared"}\}, rm : RM] \cup [type : \{\text{"Commit"}, \text{"Abort"}\}]$$

Here is the definition of the set $Messages$.

$Messages \triangleq [type : \{\text{"Prepared"}\}, rm : RM] \cup [type : \{\text{"Commit"}, \text{"Abort"}\}]$

Here is the definition of the set $Messages$.

This is the set union operator, where

$Messages \triangleq [type : \{\text{"Prepared"}\}, rm : RM] \cup [type : \{\text{"Commit"}, \text{"Abort"}\}]$

$$S \cup T$$

Here is the definition of the set $Messages$.

This is the set union operator, where S union T is the set of all elements in S or T or both.

$Messages \triangleq [type : \{\text{"Prepared"}\}, rm : RM] \cup [type : \{\text{"Commit"}, \text{"Abort"}\}]$

\union

\cup

Here is the definition of the set $Messages$.

This is the set union operator, where S union T is the set of all elements in S or T or both.

Union is typed either backslash $union$ or backslash cup.

$$Messages \triangleq [type : \{\text{"Prepared"}\}, rm : RM] \cup [type : \{\text{"Commit"}, \text{"Abort"}\}]$$

So Messages is the union of two sets, the first

$$Messages \triangleq [type : \{\text{"Prepared"}\}, rm : RM] \cup [type : \{\text{"Commit"}, \text{"Abort"}\}]$$

So Messages is the union of two sets, the first is the set of records whose $type$ field is an element of the set containing the single element $Prepared$, and whose rm field is an element of the set RM of resource managers.

$$Messages \triangleq [type : \{\text{"Prepared"}\}, rm : RM] \cup [type : \{\text{"Commit"}, \text{"Abort"}\}]$$

$$[\, type \mapsto \text{"}Prepared\text{"}, rm \mapsto r \,]$$

So Messages is the union of two sets, the first is the set of records whose $type$ field is an element of the set containing the single element $Prepared$, and whose rm field is an element of the set RM of resource managers.

A record with $type$ field equal to the string $Prepared$ and rm field equal to the resource manager r represents

$$Messages \triangleq [type : \{\text{“Prepared”}\},\ rm : RM] \ \cup\ [type : \{\text{“Commit”},\ \text{“Abort”}\}]$$

$$[\, type \mapsto \text{“}Prepared\text{”},\ rm \mapsto r \,]$$

Represents a $Prepared$ message sent by r to the TM.

So Messages is the union of two sets, the first is the set of records whose $type$ field is an element of the set containing the single element $Prepared$, and whose rm field is an element of the set RM of resource managers.

A record with $type$ field equal to the string $Prepared$ and rm field equal to the resource manager r represents a $Prepared$ message sent by resource manager r to the Transaction Manager.

$$Messages \triangleq [type : \{\text{"Prepared"}\}, rm : RM] \cup \boxed{[type : \{\text{"Commit"}, \text{"Abort"}\}]}$$

Each record in that set represents either a $Commit$ or an $Abort$ message sent by the transaction manager to all the resource managers.

This set equals

$$Messages \triangleq [type : \{\text{"Prepared"}\}, rm : RM] \cup \boxed{[type : \{\text{"Commit"}, \text{"Abort"}\}]}$$

$$\{ [type \mapsto \text{"}Commit\text{"}], [type \mapsto \text{"}Abort\text{"}] \}$$

Each record in that set represents either a $Commit$ or an $Abort$ message sent by the transaction manager to all the resource managers.

This set equals the set containing two elements, each a record with only a $type$ field.

$Messages \triangleq [type : \{\text{"Prepared"}\}, rm : RM] \cup [type : \{\text{"Commit"}, \text{"Abort"}\}]$

$\{ [type \mapsto \text{"}Commit\text{"}], [type \mapsto \text{"}Abort\text{"}] \}$

Each record represents a message
sent by the TM to all RMs.

Each record in that set represents either a *Commit* or an *Abort* message sent by the transaction manager to all the resource managers.

This set equals the set containing two elements, each a record with only a *type* field.

These records represent a *commit* and an *abort* message sent by the transaction manager to all the resource managers.

$$
\begin{aligned}
&TPInit \triangleq \\
&\quad \land\ rmState = [r \in RM \mapsto \text{"working"}] \\
&\quad \land\ tmState = \text{"init"} \\
&\quad \land\ tmPrepared = \{\} \\
&\quad \land\ msgs = \{\}
\end{aligned}
$$

Here's the initial state formula.

$TPInit \triangleq$

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

$\land$ $tmState = \text{"init"}$

$\land$ $tmPrepared = \{\}$

$\land$ $msgs = \{\}$

Here's the initial state formula.

$rmState$ has the same initial value as in $TCommit$ – a function that assigns the string $working$ to every resource manager.

$TPInit \triangleq$

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

$\wedge\ tmState = \text{"init"}$

$\wedge\ tmPrepared = \{\}$

$\wedge\ msgs = \{\}$

Here's the initial state formula.

$rmState$ has the same initial value as in $TCommit$ – a function that assigns the string $working$ to every resource manager.

Here are the initial values of the variables describing the transaction manager's state.

$$
\begin{aligned}
&TPInit \triangleq \\
&\quad \land\ rmState = [r \in RM \mapsto \text{“working”}] \\
&\quad \land\ tmState = \text{“init”} \\
&\quad \land\ tmPrepared = \{\} \\
&\quad \land\ \boxed{msgs = \{\}}
\end{aligned}
$$

Here's the initial state formula.

$rmState$ has the same initial value as in $TCommit$ – a function that assigns the string $working$ to every resource manager.

Here are the initial values of the variables describing the transaction manager's state.

And initially, no messages have been sent.

[slide 105]

$$
\begin{aligned}
&TPInit \triangleq \\
&\quad \land\ rmState = [r \in RM \mapsto \text{“working”}] \\
&\quad \land\ tmState = \text{“init”} \\
&\quad \land\ tmPrepared = \{\} \\
&\quad \land\ msgs = \{\}
\end{aligned}
$$

Next come the definitions of subformulas of the next-state formula, starting with those subformulas that describe actions taken by the transaction manager.

$TMRcvPrepared(r) \triangleq$

Describes the receipt of a $Prepared$ message from RM r by TM.

This subformula describes the receipt of a $Prepared$ message from resource manager r by the transaction manager.

$TMRcvPrepared(r) \triangleq$
$\quad \land\ tmState =$ "init"

This subformula describes the receipt of a $Prepared$ message from resource manager r by the transaction manager.

The message can be received only when the transaction manager is in its $init$ state

$$
\begin{array}{l}
TMRcvPrepared(r) \triangleq \\
\quad \land\ tmState = \text{"init"} \\
\quad \land\ [type \mapsto \text{"Prepared"},\ rm \mapsto r] \in msgs
\end{array}
$$

This subformula describes the receipt of a *Prepared* message from resource manager r by the transaction manager.

The message can be received only when the transaction manager is in its *init* state

and there is a *Prepared* message from resource manager r in the set m-s-g-s of sent messages.

$$TMRcvPrepared(r) \triangleq$$
$$\land\ tmState = \text{"init"}$$
$$\land\ [type \mapsto \text{"Prepared"},\ rm \mapsto r] \in msgs$$
$$\land\ tmPrepared' =$$

It sets the new value of $tmPrepared$ to the union of its current value and the set containing the element r.

$TMRcvPrepared(r) \triangleq$

$\quad \land\ tmState = \text{"init"}$

$\quad \land\ [type \mapsto \text{"Prepared"},\ rm \mapsto r] \in msgs$

$\quad \land\ tmPrepared' = tmPrepared \cup$

It sets the new value of $tmPrepared$ to the union of its current value and the set containing the element r.

$$
\begin{aligned}
&TMRcvPrepared(r) \triangleq \\
&\quad \land\ tmState = \text{"init"} \\
&\quad \land\ [type \mapsto \text{"Prepared"},\ rm \mapsto r] \in msgs \\
&\quad \land\ tmPrepared' = tmPrepared \cup \{r\}
\end{aligned}
$$

It sets the new value of $tmPrepared$ to the union of its current value and the set containing the element r.

$TMRcvPrepared(r) \triangleq$

$\land\ tmState =$ "init"

$\land\ [type \mapsto$ "Prepared", $rm \mapsto r] \in msgs$

$\land\ tmPrepared' = tmPrepared \cup \{r\}$

Adds r to $tmPrepared$.

It sets the new value of $tmPrepared$ to the union of its current value and the set containing the element r.

In other words, it adds r to the set $tmPrepared$.

$$
\begin{array}{l}
TMRcvPrepared(r) \triangleq \\
\quad \land\ tmState = \text{“init”} \\
\quad \land\ [type \mapsto \text{“Prepared”},\ rm \mapsto r] \in msgs \\
\quad \land\ tmPrepared' = tmPrepared \cup \{r\} \\
\quad \land\ \text{UNCHANGED}\ \langle rmState,\ tmState,\ msgs \rangle
\end{array}
$$

It sets the new value of $tmPrepared$ to the union of its current value and the set containing the element r.

In other words, it adds r to the set $tmPrepared$.

And finally, there's an UNCHANGED formula.

[slide 114]

$TMRcvPrepared(r) \triangleq$

$\land\ tmState = \text{"init"}$

$\land\ [type \mapsto \text{"Prepared"},\ rm \mapsto r] \in msgs$

$\land\ tmPrepared' = tmPrepared \cup \{r\}$

$\land\ \text{UNCHANGED}\ \langle rmState,\ tmState,\ msgs\rangle$

a triple

This expression is an ordered triple.

$$
\begin{aligned}
&TMRcvPrepared(r) \triangleq \\
&\quad \land\ tmState = \text{“init”} \\
&\quad \land\ [type \mapsto \text{“Prepared”},\ rm \mapsto r] \in msgs \\
&\quad \land\ tmPrepared' = tmPrepared \cup \{r\} \\
&\quad \land\ \text{UNCHANGED}\ \langle rmState,\ tmState,\ msgs \rangle
\end{aligned}
$$

<< >>

This expression is an ordered triple.

The angle brackets are typed less-than-less-than and greater-that-greater-than.

$$TMRcvPrepared(r) \triangleq$$
$$\land\ tmState = \text{"init"}$$
$$\land\ [type \mapsto \text{"Prepared"},\ rm \mapsto r] \in msgs$$
$$\land\ tmPrepared' = tmPrepared \cup \{r\}$$
$$\land\ \text{UNCHANGED}\ \langle rmState,\ tmState,\ msgs \rangle$$

This expression is an ordered triple.

The angle brackets are typed less-than-less-than and greater-that-greater-than.

The entire UNCHANGED formula is equivalent to

$TMRcvPrepared(r) \triangleq$

$\land\ tmState = \text{"init"}$

$\land\ [type \mapsto \text{"Prepared"},\ rm \mapsto r] \in msgs$

$\land\ tmPrepared' = tmPrepared \cup \{r\}$

$\land\ \text{UNCHANGED}\ \langle rmState,\ tmState,\ msgs \rangle$

Equivalent to $\land\ rmState' = rmState$

$\land\ tmState' = tmState$

$\land\ msgs' = msgs$

This expression is an ordered triple.

The angle brackets are typed less-than-less-than and greater-that-greater-than.

The entire UNCHANGED formula is equivalent to this formula

$TMRcvPrepared(r) \triangleq$
$\land\ tmState = \text{“init”}$
$\land\ [type \mapsto \text{“Prepared”},\ rm \mapsto r] \in msgs$
$\land\ tmPrepared' = tmPrepared \cup \{r\}$
$\land\ \text{UNCHANGED}\ \langle rmState,\ tmState,\ msgs \rangle$

Equivalent to $\land\ rmState' = rmState$
$\land\ tmState' = tmState$
$\land\ msgs' = msgs$

Which asserts $rmState$, $tmState$, and $msgs$ are left unchanged.

This expression is an ordered triple.

The angle brackets are typed less-than-less-than and greater-that-greater-than.

The entire UNCHANGED formula is equivalent to this formula which asserts that the values of the variables $rmState$, $tmState$, and m-s-g-s are all left unchanged.

$$
\begin{array}{l}
TMRcvPrepared(r) \triangleq \\
\quad \land\ tmState = \text{“init”} \\
\quad \land\ [type \mapsto \text{“Prepared”},\ rm \mapsto r] \in msgs \\
\quad \land\ tmPrepared' = tmPrepared \cup \{r\} \\
\quad \land\ \text{UNCHANGED}\ \langle rmState,\ tmState,\ msgs \rangle
\end{array}
$$

These two conjunctions have no primes.

$TMRcvPrepared(r) \triangleq$

$\land\ tmState = \text{"init"}$

$\land\ [type \mapsto \text{"Prepared"},\ rm \mapsto r] \in msgs$

$\land\ tmPrepared' = tmPrepared \cup \{r\}$

$\land\ \text{UNCHANGED}\ \langle rmState,\ tmState,\ msgs \rangle$

These two conjunctions have no primes.

$TMRcvPrepared(r) \triangleq$

$\land\ tmState = \text{"init"}$

$\land\ [type \mapsto \text{"Prepared"},\ rm \mapsto r] \in msgs$

$\land\ tmPrepared' = tmPrepared \cup \{r\}$

$\land\ \text{UNCHANGED}\ \langle rmState,\ tmState,\ msgs \rangle$

Conditions on the first state of a step.

These two conjunctions have no primes.

They're conditions on the first state of a step.

$TMRcvPrepared(r) \triangleq$

$\land\ tmState = \text{"init"}$

$\land\ [type \mapsto \text{"Prepared"},\ rm \mapsto r] \in msgs$

$\land\ tmPrepared' = tmPrepared \cup \{r\}$

$\land\ \text{UNCHANGED}\ \langle rmState,\ tmState,\ msgs \rangle$

Conditions on the first state of a step.

Enabling conditions.

They're called enabling conditions of the formula.

Enabling conditions should almost always go at the beginning of an action formula – a formula that contains primed variables. That makes the formula easier to understand, and TLC often can't handle the action formula if you don't.

$$
\begin{aligned}
&TMRcvPrepared(r) \triangleq \\
&\quad \land\ tmState = \text{“init”} \\
&\quad \land\ [type \mapsto \text{“Prepared”},\ rm \mapsto r] \in msgs \\
&\quad \land\ tmPrepared' = tmPrepared \cup \{r\} \\
&\quad \land\ \text{UNCHANGED}\ \langle rmState,\ tmState,\ msgs \rangle
\end{aligned}
$$

The step doesn’t remove the message from m-s-g-s or change $tmState$

$TMRcvPrepared(r) \triangleq$

$\land\ tmState = \text{"init"}$

$\land\ [type \mapsto \text{"Prepared"},\ rm \mapsto r] \in msgs$

$\land\ tmPrepared' = tmPrepared \cup \{r\}$

$\land\ \text{UNCHANGED}\ \langle rmState,\ tmState,\ msgs \rangle$

The step doesn't remove the message from m-s-g-s or change $tmState$

so the formula is still enabled after the step.

$$TMRcvPrepared(r) \triangleq$$
$$\land\ tmState = \text{"init"}$$
$$\land\ [type \mapsto \text{"Prepared"},\ rm \mapsto r] \in msgs$$
$$\land\ tmPrepared' = tmPrepared \cup \{r\}$$
$$\land\ \text{UNCHANGED}\ \langle rmState,\ tmState,\ msgs \rangle$$

r in $tmPrepared$

The step doesn't remove the message from m-s-g-s or change $tmState$

so the formula is still enabled after the step.

But the step adds the element r to $tmPrepared$, so any subsequent step allowed by $TMRcvPrepared(r)$ occurs with r in $tmPrepared$,

$TMRcvPrepared(r) \triangleq$
$\land\ tmState = \text{"init"}$
$\land\ [type \mapsto \text{"Prepared"},\ rm \mapsto r] \in msgs$
$\land\ tmPrepared' = tmPrepared \cup \{r\}$
$\land\ \text{UNCHANGED}\ \langle rmState,\ tmState,\ msgs \rangle$

r in $tmPrepared$ implies $tmPrepared' = tmPrepared$

The step doesn't remove the message from m-s-g-s or change $tmState$

so the formula is still enabled after the step.

But the step adds the element r to $tmPrepared$, so any subsequent step allowed by $TMRcvPrepared(r)$ occurs with r in $tmPrepared$, which implies that $tmPrepared$ is unchanged.

$TMRcvPrepared(r) \triangleq$
$\land\ tmState = \text{"init"}$
$\land\ [type \mapsto \text{"Prepared"},\ rm \mapsto r] \in msgs$
$\land\ tmPrepared' = tmPrepared \cup \{r\}$
$\land\ \text{UNCHANGED}\ \langle rmState,\ tmState,\ msgs \rangle$

r in $tmPrepared$ implies $tmPrepared' = tmPrepared$

A set can't contain two copies of r.

Because a set either contains an element or it doesn't; it can't contain multiple copies of the same element.

$TMRcvPrepared(r) \triangleq$
$\quad \land\ tmState = \text{"init"}$
$\quad \land\ [type \mapsto \text{"Prepared"},\ rm \mapsto r] \in msgs$
$\quad \land\ tmPrepared' = tmPrepared \cup \{r\}$
$\quad \land\ \text{UNCHANGED}\ \langle rmState,\ tmState,\ msgs \rangle$

r in $tmPrepared$ implies $tmPrepared' = tmPrepared$

Because a set either contains an element or it doesn't; it can't contain multiple copies of the same element.

So if r is in $tmPrepared$, then the step leaves $tmPrepared$ unchanged.

$TMRcvPrepared(r) \triangleq$

$\land\ tmState = \text{"init"}$

$\land\ [type \mapsto \text{"Prepared"},\ rm \mapsto r] \in msgs$

$\land\ tmPrepared' = tmPrepared \cup \{r\}$

$\land\ \text{UNCHANGED}\ \langle rmState,\ tmState,\ msgs \rangle$

r in $tmPrepared$ implies $tmPrepared' = tmPrepared$

Because a set either contains an element or it doesn't; it can't contain multiple copies of the same element.

So if r is in $tmPrepared$, then the step leaves $tmPrepared$ unchanged.

The step also leaves all the other variables unchanged.

$TMRcvPrepared(r) \triangleq$
$\land\ tmState = \text{"init"}$
$\land\ [type \mapsto \text{"Prepared"},\ rm \mapsto r] \in msgs$
$\land\ tmPrepared' = tmPrepared \cup \{r\}$
$\land$ UNCHANGED $\langle rmState,\ tmState,\ msgs \rangle$

r in $tmPrepared$ implies all variables are unchanged.

Because a set either contains an element or it doesn't; it can't contain multiple copies of the same element.

So if r is in $tmPrepared$, then the step leaves $tmPrepared$ unchanged.

The step also leaves all the other variables unchanged.

So all subsequent $TMRcvPrepared(r)$ steps leave all the variables unchanged.

$$
\begin{array}{l}
TMRcvPrepared(r) \triangleq \\
\quad \land\ tmState = \text{"init"} \\
\quad \land\ [type \mapsto \text{"Prepared"},\ rm \mapsto r] \in msgs \\
\quad \land\ tmPrepared' = tmPrepared \cup \{r\} \\
\quad \land\ \text{UNCHANGED}\ \langle rmState,\ tmState,\ msgs \rangle
\end{array}
$$

r in $tmPrepared$ implies all variables are unchanged.

Steps leaving all variables unchanged make no difference.

We will see later why steps that leave all variables unchanged make no difference and are always allowed.

THE REST OF THE SPEC

You should now be able to understand the rest of the spec.

In fact, you should be able to write most of it yourself.

I will describe the remaining subformulas of $TPNext$.

I will now describe the steps allowed by each of the remaining subformulas of the next-state formula $TPNext$.

I will describe the remaining subformulas of $TPNext$.

After each description

I will now describe the steps allowed by each of the remaining subformulas of the next-state formula $TPNext$.

After each description,

I will describe the remaining subformulas of $TPNext$.

After each description

– Stop the video.

I will now describe the steps allowed by each of the remaining subformulas of the next-state formula $TPNext$.

After each description, stop the video,

I will describe the remaining subformulas of $TPNext$.

After each description

– Stop the video.

– Write the definition.

I will now describe the steps allowed by each of the remaining subformulas of the next-state formula $TPNext$.

After each description, stop the video, write down the definition,

I will describe the remaining subformulas of $TPNext$.

After each description

- Stop the video.
- Write the definition.
- Compare it with the one in the module.

I will now describe the steps allowed by each of the remaining subformulas of the next-state formula $TPNext$.

After each description, stop the video, write down the definition, and compare it with the definition in the module.

I will describe the remaining subformulas of $TPNext$.

After each description

- Stop the video.
- Write the definition.
- Compare it with the one in the module.

Save your definitions that differ.

If your definition is significantly different from the one in the module, save it.

Later you can let TLC check if it's correct.

We'll start with the other two subformulas that represent steps performed by the transaction manager.

$$TMCommit \triangleq$$

Formula $TMCommit$

$TMCommit \triangleq$

It allows steps where the TM sends *Commit* messages to the RMs and sets $tmState$ to "$done$".

Formula $TMCommit$

allows steps where the transaction manager sends *Commit* messages to the resource managers and sets $tmState$ to the string "$done$".

$TMCommit \triangleq$

It allows steps where the TM sends *Commit* messages to the RMs and sets $tmState$ to "$done$".

The messages are sent by adding $[type \mapsto \text{"}Commit\text{"}]$ to $msgs$.

Formula $TMCommit$

allows steps where the transaction manager sends *Commit* messages to the resource managers and sets $tmState$ to the string "$done$".

The sending of those messages is described by adding the record with $type$ field equal to the string *Commit* to the set $msgs$.

$TMCommit \triangleq$

It allows steps where the TM sends *Commit* messages to the RMs and sets $tmState$ to "$done$".

It is enabled if $tmState$ equals "$init$" and $tmPrepared$ equals RM.

The formula is enabled if and only if $tmState$ equals "$init$" and $tmPrepared$ equals the set of resource managers.

$TMCommit \triangleq$

It allows steps where the TM sends *Commit* messages to the RMs and sets $tmState$ to "$done$".

It is enabled if $tmState$ equals "$init$" and $tmPrepared$ equals RM.

Write the definition now.

The formula is enabled if and only if $tmState$ equals "$init$" and $tmPrepared$ equals the set of resource managers.

Stop the video and write your definition now.

$TMAbort \triangleq$

Formula $TMAbort$

$TMAbort \triangleq$

The TM sends *Abort* messages to the RMs and sets $tmState$ to "$done$".

Formula $TMAbort$

allows steps where the transaction manager sends *Abort* messages to the resource managers and sets $tmState$ to the string "$done$".

$TMAbort \triangleq$

The TM sends *Abort* messages to the RMs and sets $tmState$ to "$done$".

It is enabled if $tmState$ equals "$init$".

Formula $TMAbort$

allows steps where the transaction manager sends *Abort* messages to the resource managers and sets $tmState$ to the string "$done$".

The formula is enabled if and only if $tmState$ equals "$init$".

Next come the formulas describing steps performed by the resource managers.

$RMPrepare(r) \triangleq$

Formula $RMPrepare$ of r.

$RMPrepare(r) \triangleq$

RM r sets its state to "$prepared$" and sends a *Prepared* message to the TM.

Formula $RMPrepare$ of r.

Resource manager r sets its state to $prepared$ and sends a *Prepared* message to the transaction manager.

$RMPrepare(r) \triangleq$

RM r sets its state to "$prepared$" and sends a *Prepared* message to the TM.

It's enabled if $rmState[r]$ equals "$working$".

Formula $RMPrepare$ of r.

Resource manager r sets its state to $prepared$ and sends a *Prepared* message to the transaction manager.

It's enabled if and only if $rmState$ of r equals "$working$".

$RMChooseToAbort(r) \triangleq$

Formula $RMChooseToAbort$ of r.

$RMChooseToAbort(r) \triangleq$

When in its "$working$" state, RM r can go to the "$aborted$" state.

Formula $RMChooseToAbort$ of r.

When in its "$working$" state, resource manager r can go to the "$aborted$" state.

After r has aborted, no RM can ever commit; and the TM should eventually take a $TMAbort$ step.

After r has aborted, no resource manager can ever commit; and the transaction manager should eventually take a $TMAbort$ step.

After r has aborted, no RM can ever commit; and the TM should eventually take a $TMAbort$ step.

In practice, r would inform the TM that it has aborted so the TM knows it should abort the transaction.

After r has aborted, no resource manager can ever commit; and the transaction manager should eventually take a $TMAbort$ step.

In practice, r would inform the transaction manager that it has aborted so the transaction manager knows it should abort the transaction.

After r has aborted, no RM can ever commit; and the TM should eventually take a $TMAbort$ step.

In practice, r would inform the TM that it has aborted so the TM knows it should abort the transaction.

But that optimization isn't relevant for implementing $TCommit$.

After r has aborted, no resource manager can ever commit; and the transaction manager should eventually take a $TMAbort$ step.

In practice, r would inform the transaction manager that it has aborted so the transaction manager knows it should abort the transaction.

But that's an optimization and isn't relevant for implementing $TCommit$, so we omit it from the spec.

$RMRcvCommitMsg(r) \triangleq$

$RMRcvAbortMsg(r) \triangleq$

Formulas $RMRcvCommitMsg$ of r and $RMRcvAbortMsg$ of r.

$RMRcvCommitMsg(r) \triangleq$

$RMRcvAbortMsg(r) \triangleq$

RM r receives a "$commit$" or "$abort$" message and sets its state accordingly.

Formulas $RMRcvCommitMsg$ of r and $RMRcvAbortMsg$ of r.

Resource manager r receives a "$commit$" or "$abort$" message and sets its state accordingly.

$TPNext \triangleq$

The next-state formula

$$
\begin{array}{l}
TPNext \triangleq \\
\quad \lor\ TMCommit \lor TMAbort \\
\quad \lor\ \exists\, r \in RM : \\
\qquad\quad TMRcvPrepared(r) \lor RMPrepare(r) \lor RMChooseToAbort(r) \\
\qquad\qquad \lor\ RMRcvCommitMsg(r) \lor RMRcvAbortMsg(r)
\end{array}
$$

The next-state formula

is the disjunction of all seven subformulas

$$
\begin{array}{l}
TPNext \triangleq \\
\quad \lor\ TMCommit \lor TMAbort \\
\quad \lor\ \exists\, r \in RM : \\
\qquad\quad TMRcvPrepared(r) \lor RMPrepare(r) \lor RMChooseToAbort(r) \\
\qquad\qquad \lor\ RMRcvCommitMsg(r) \lor RMRcvAbortMsg(r)
\end{array}
$$

The next-state formula

is the disjunction of all seven subformulas

where the formulas with parameter r are existentially quantified over all r in the set of resource managers.

$$
\begin{aligned}
&TPNext \triangleq \\
&\quad \lor\ TMCommit \lor TMAbort \\
&\quad \lor\ \exists\, r \in RM : \\
&\qquad TMRcvPrepared(r) \lor RMPrepare(r) \lor RMChooseToAbort(r) \\
&\qquad\quad \lor\ RMRcvCommitMsg(r) \lor RMRcvAbortMsg(r)
\end{aligned}
$$

Existential quantification over the disjunction of these formulas

$$
\begin{aligned}
&TPNext \triangleq \\
&\quad \lor\ TMCommit \lor TMAbort \\
&\quad \lor\ \exists\, r \in RM : \\
&\qquad TMRcvPrepared(r) \lor RMPrepare(r) \lor RMChooseToAbort(r) \\
&\qquad\quad \lor RMRcvCommitMsg(r) \lor RMRcvAbortMsg(r)
\end{aligned}
$$

is equivalent to

$$
\begin{aligned}
&\lor\ \exists\, r \in RM : TMRcvPrepared(r) \\
&\lor\ \exists\, r \in RM : RMPrepare(r) \\
&\qquad\qquad \vdots \\
&\lor\ \exists\, r \in RM : RMRcvAbortMsg(r)
\end{aligned}
$$

Existential quantification over the disjunction of these formulas

is equivalent to the disjunction of existential quantification over each one.

$$
\begin{aligned}
&TPNext \triangleq \\
&\quad \lor\ TMCommit \lor TMAbort \\
&\quad \lor\ \exists\, r \in RM : \\
&\qquad\quad TMRcvPrepared(r) \lor RMPrepare(r) \lor RMChooseToAbort(r) \\
&\qquad\qquad \lor\ RMRcvCommitMsg(r) \lor RMRcvAbortMsg(r)
\end{aligned}
$$

is equivalent to

$$
\begin{aligned}
&\lor\ \exists\, r \in RM : TMRcvPrepared(r) \\
&\lor\ \exists\, r \in RM : RMPrepare(r) \\
&\qquad\qquad \vdots \\
&\lor\ \exists\, r \in RM : RMRcvAbortMsg(r)
\end{aligned}
$$

Existential quantification over the disjunction of these formulas

is equivalent to the disjunction of existential quantification over each one.

Stop the video and convince yourself that these two formulas are equivalent.

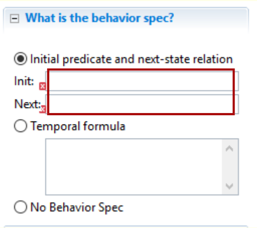
CHECKING THE SPEC

Let's now check the specification.

Create a New Model

In the Toolbox, create a new model.

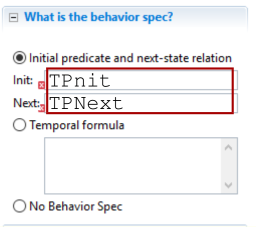
Create a New Model



In the Toolbox, create a new model.

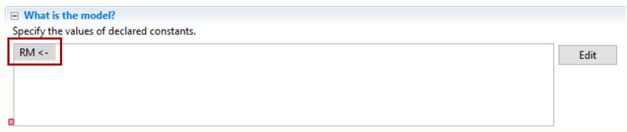
Because we're not using the default names,

Create a New Model

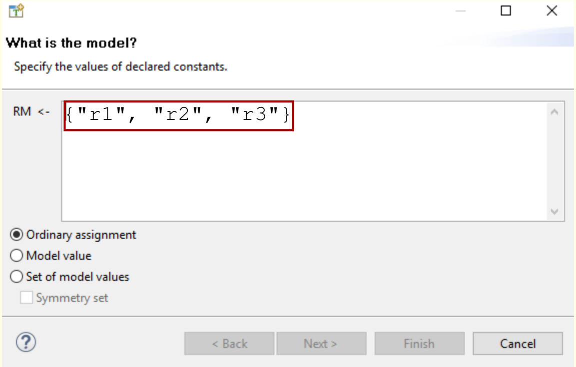


In the Toolbox, create a new model.

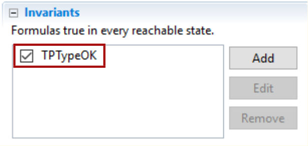
Because we're not using the default names, you'll have to enter the initial and next-state formulas.



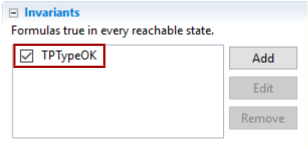
You'll also have to enter a value for the constant RM.



As we did for $TCommit$, let RM be the set of three strings $r1$, $r2$, and $r3$.



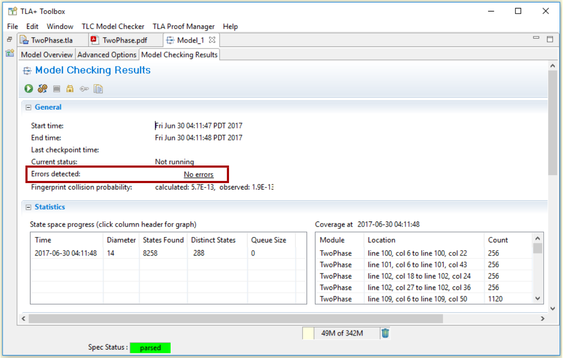
And add $TPTypeOK$ as an invariant to be checked.



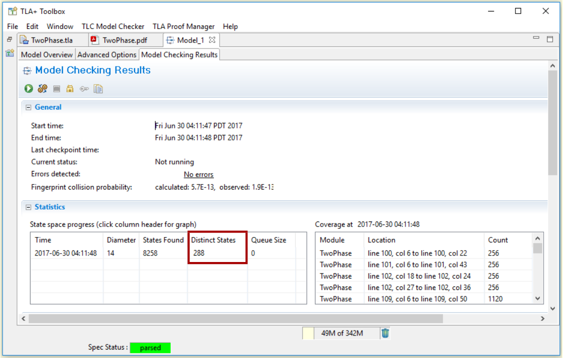
Run TLC.

And add $TPTypeOK$ as an invariant to be checked.

Run TLC on the model.



TLC should detect no errors.



TLC should detect no errors.

Remember the number of distinct states that TLC found.

Check Your Definitions

You can now check the definitions you wrote of those six subformulas of the next-state formula.

Check Your Definitions

To check a definition:

You can now check the definitions you wrote of those six subformulas of the next-state formula.

To check a definition that you're not sure of:

Check Your Definitions

To check a definition:

– Comment out the definition in the spec.

You can now check the definitions you wrote of those six subformulas of the next-state formula.

To check a definition that you're not sure of: Comment out the definition that's in the spec.

Check Your Definitions

To check a definition:

- Comment out the definition in the spec.
- Insert your definition.

You can now check the definitions you wrote of those six subformulas of the next-state formula.

To check a definition that you're not sure of: Comment out the definition that's in the spec. Insert your definition.

Check Your Definitions

To check a definition:

– Comment out the definition in the spec.

– Insert your definition.

– Run TLC.

You can now check the definitions you wrote of those six subformulas of the next-state formula.

To check a definition that you're not sure of: Comment out the definition that's in the spec. Insert your definition. And run TLC on the same model.

Check Your Definitions

To check a definition:

- Comment out the definition in the spec.
- Insert your definition.
- Run TLC.

TLC should find no error and again find 288 distinct states.

Your definition is probably correct if TLC finds no error and again finds 288 distinct states.

MODEL VALUES

Model Values

Symmetry Sets

Symmetry Sets

Symmetry Sets

All RMs are identical / interchangeable.

Symmetry Sets

In two-phase commit, every resource manager plays an identical role. The resource managers are interchangeable.

Symmetry Sets

All RMs are identical / interchangeable.

Suppose $RM = \{\text{“}r1\text{”}, \text{“}r2\text{”}, \text{“}r3\text{”}\}$.

Symmetry Sets

In two-phase commit, every resource manager plays an identical role. The resource managers are interchangeable.

For example, suppose the resource managers are named “r1”, “r2”, and “r3”.

Symmetry Sets

All RMs are identical / interchangeable.

Suppose $RM = \{$“$r1$”, “$r2$”, “$r3$”$\}$.

“$r1$” $\leftrightarrow$ “$r3$” in one possible state yields a possible state.

If we interchange “$r1$” and “$r3$” in a possible state of a behavior, we get another possible state of a behavior.

Symmetry Sets

All RMs are identical / interchangeable.

Suppose $RM = \{\text{“}r1\text{”}, \text{“}r2\text{”}, \text{“}r3\text{”}\}$.

“r1” $\leftrightarrow$ “r3” means

If we interchange “r1” and “r3” in a possible state of a behavior, we get another possible state of a behavior.

Interchanging “r1” and “r3” in a state means

Symmetry Sets

All RMs are identical / interchangeable.

Suppose $RM = \{$"r1", "r2", "r3"$\}$.

"r1" $\leftrightarrow$ "r3" means

- $rmState$["r1"] $\leftrightarrow$ $rmState$["r3"]

If we interchange "r1" and "r3" in a possible state of a behavior, we get another possible state of a behavior.

Interchanging "r1" and "r3" in a state means

interchanging the values of $rmState$["r1"] and $rmState$["r3"],

Symmetry Sets

All RMs are identical / interchangeable.

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

“r1” $\leftrightarrow$ “r3” means

- $rmState[\text{“}r\text{1”}] \leftrightarrow rmState[\text{“}r\text{3”}]$
- $[type \mapsto \text{“}Prepared\text{”}, rm \mapsto \text{“}r\text{1”}] \in msgs$

replacing this message in m-s-g-s

Symmetry Sets

All RMs are identical / interchangeable.

Suppose $RM = \{\text{“}r1\text{”}, \text{“}r2\text{”}, \text{“}r3\text{”}\}$.

$\text{“}r1\text{”} \leftrightarrow \text{“}r3\text{”}$ means

- $rmState[\text{“}r1\text{”}] \leftrightarrow rmState[\text{“}r3\text{”}]$
- $[type \mapsto \text{“}Prepared\text{”}, rm \mapsto \text{“}r1\text{”}] \in msgs$

  $\leftrightarrow$

  $[type \mapsto \text{“}Prepared\text{”}, rm \mapsto \text{“}r3\text{”}] \in msgs$

replacing this message in m-s-g-s

with this one, and vice-versa.

Symmetry Sets

All RMs are identical / interchangeable.

Suppose $RM = \{\text{"}r1\text{"}, \text{"}r2\text{"}, \text{"}r3\text{"}\}$.

"$r1$" $\leftrightarrow$ "$r3$" means

- $rmState[\text{"}r1\text{"}] \leftrightarrow rmState[\text{"}r3\text{"}]$
- $[type \mapsto \text{"}Prepared\text{"}, rm \mapsto \text{"}r1\text{"}] \in msgs$
  $$\leftrightarrow$$
  $[type \mapsto \text{"}Prepared\text{"}, rm \mapsto \text{"}r3\text{"}] \in msgs$

. . .

replacing this message in m-s-g-s

with this one, and vice-versa.

and so on.

“$r1$” $\leftrightarrow$ “$r3$” in all states of a behavior b allowed by $TwoPhase$

Moreover, if we interchange $r1$ and $r3$ in every state of a behavior b allowed by the TwoPhase spec,

"r1" $\leftrightarrow$ "r3" in all states of a behavior b allowed by $TwoPhase$
produces a behavior $b_{1 \leftrightarrow 3}$ allowed by $TwoPhase$.

Moreover, if we interchange r1 and r3 in every state of a behavior b allowed by the TwoPhase spec,

we get another behavior, let's call it b-1-3, that's also allowed by the spec.

“$r1$” $\leftrightarrow$ “$r3$” in all states of a behavior b allowed by $TwoPhase$
produces a behavior $b_{1\leftrightarrow 3}$ allowed by $TwoPhase$.

TLC does not have to check $b_{1\leftrightarrow 3}$ if it has checked b .

Moreover, if we interchange $r1$ and $r3$ in every state of a behavior b allowed by the TwoPhase spec,

we get another behavior, let’s call it b-1-3, that’s also allowed by the spec.

TLC doesn’t have to check that some property of two-phase commit holds in behavior b-1-3 if it has checked that it holds for behavior b.

“$r1$” $\leftrightarrow$ “$r3$” in all states of a behavior b allowed by $TwoPhase$
produces a behavior $b_{1\leftrightarrow 3}$ allowed by $TwoPhase$.

TLC does not have to check $b_{1\leftrightarrow 3}$ if it has checked b.

RM is a **symmetry set** of $TwoPhase$.

Because this observation holds for interchanging any two elements of RM, we say that RM is a *symmetry set* of the specification.

"$r1$" $\leftrightarrow$ "$r3$" in all states of a behavior b allowed by $TwoPhase$ produces a behavior $b_{1\leftrightarrow 3}$ allowed by $TwoPhase$.

TLC does not have to check $b_{1\leftrightarrow 3}$ if it has checked b.

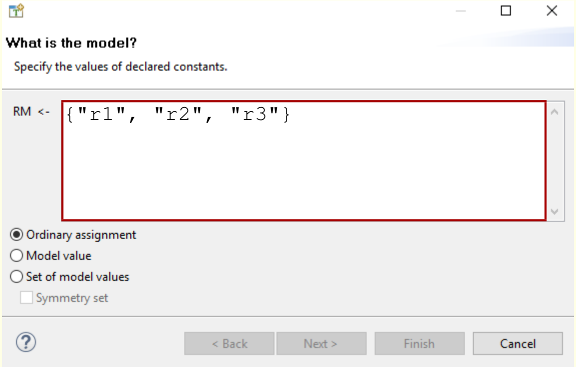
RM is a **symmetry set** of $TwoPhase$.

TLC will check fewer states if the model sets a symmetry set to a set of model values.

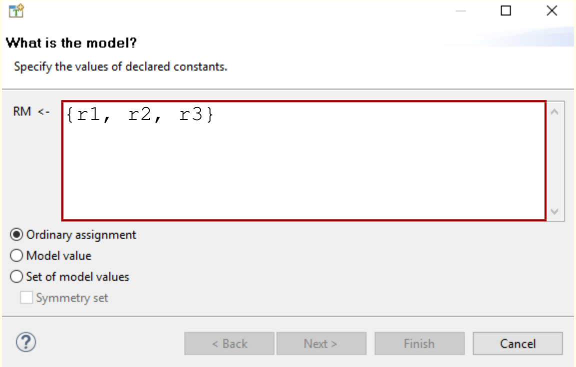
Because this observation holds for interchanging any two elements of RM, we say that RM is a *symmetry set* of the specification.

TLC will check fewer states if the model sets a symmetry set to a set consisting a special kind of values called model values.

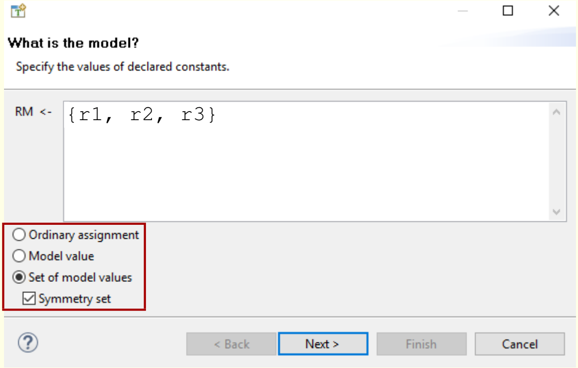
Let's do that now for our model.



Replace this set of strings

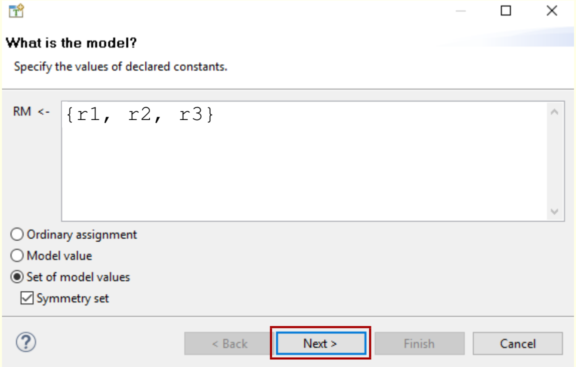


Replace this set of strings with this set of identifiers. We can use any identifiers that aren't defined in the spec.



Replace this set of strings with this set of identifiers. We can use any identifiers that aren't defined in the spec.

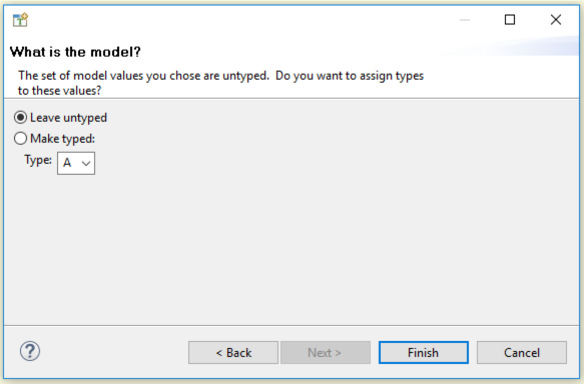
Select *Set of model values* and check *Symmetry set*.



Replace this set of strings with this set of identifiers. We can use any identifiers that aren't defined in the spec.

Select *Set of model values* and check *Symmetry set*.

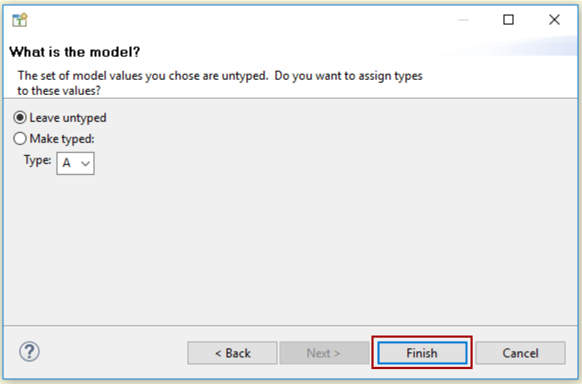
Then click *Next*



Replace this set of strings with this set of identifiers. We can use any identifiers that aren't defined in the spec.

Select *Set of model values* and check *Symmetry set*.

Then click *Next* and then



Replace this set of strings with this set of identifiers. We can use any identifiers that aren't defined in the spec.

Select *Set of model values* and check *Symmetry set*.

Then click *Next* and then click *Finish*.

Run the model.

Now run the model.

Run the model.

The model has the same 288 reachable states as before.

Now run the model.

Because there are still 3 resource managers, the model has the same 288 reachable states as before.

Run the model.

The model has the same 288 reachable states as before.

Statistics

State space progress (click column header for graph)

Time	Diameter	States Found	Distinct States	Queue Size
2017-06-30 10:47:08	11	318	80	0

Now run the model.

Because there are still 3 resource managers, the model has the same 288 reachable states as before.

But TLC only had to check 80 of them—fewer than one-third as many states .

TLC may miss errors if you claim a set is a symmetry set when it's not.

TLC may miss errors if you claim a set is a symmetry set when it's not.

TLC may miss errors if you claim a set is a symmetry set when it’s not.

For now, you can declare a set to be a symmetry set if its model values are not used elsewhere.

TLC may miss errors if you claim a set is a symmetry set when it’s not.

For now, you can safely declare a set to be a symmetry set if its model values are not used elsewhere.

TLC may miss errors if you claim a set is a symmetry set when it's not.

For now, you can declare a set to be a symmetry set if its model values are not used elsewhere.

The next lecture fully explains when a set of model values can be a symmetry set.

TLC may miss errors if you claim a set is a symmetry set when it's not.

For now, you can safely declare a set to be a symmetry set if its model values are not used elsewhere.

The next lecture fully explains when a set of model values can be a symmetry set.

CORRECTNESS OF TWO-PHASE COMMIT

Correctness of the two-phase commit protocol.

We've checked that $TypeOK$ is an invariant of the spec.

So far, we've only checked that $TypeOK$ is an invariant of the spec.

We've checked that $TypeOK$ is an invariant of the spec.

We should check that formula $TCConsistent$ of $TCommit$, which asserts that one RM can't commit and another abort, is also an invariant.

So far, we've only checked that $TypeOK$ is an invariant of the spec.

To check that two-phase commit actually is a transaction commit protocol, we should check that formula $TCConsistent$ of the $TCommit$ spec, which asserts that one resource manager can't commit if another aborts, is also an invariant of the $TwoPhase$ spec.

We've checked that $TypeOK$ is an invariant of the spec.

We should check that formula $TCConsistent$ of $TCommit$, which asserts that one RM can't commit and another abort, is also an invariant.

The statement

INSTANCE $TCommit$

imports the definitions from $TCommit$ into module $TwoPhase$.

The stuff at the end of module $TwoPhase$ that I haven't talked about includes this INSTANCE statement, which imports all the definitions from module $TCommit$, including the definition of $TCConsistent$, into the current module $TwoPhase$.

We've checked that $TypeOK$ is an invariant of the spec.

We should check that formula $TCConsistent$ of $TCommit$, which asserts that one RM can't commit and another abort, is also an invariant.

The statement

INSTANCE $TCommit$

imports the definitions from $TCommit$ into module $TwoPhase$.

Add the invariant $TCConsistent$ to your model and have TLC check it.

The stuff at the end of module $TwoPhase$ that I haven't talked about includes this INSTANCE statement, which imports all the definitions from module $TCommit$, including the definition of $TCConsistent$, into the current module $TwoPhase$.

So you can just add the invariant $TCConsistent$ to your model and have TLC check that it is indeed an invariant of the $TwoPhase$ spec.

Two-phase commit doesn't just maintain the invariance of $TCConsistent$

The two-phase commit protocol doesn't just maintain the same invariant $TCConsistent$ as transaction commit;

Two-phase commit doesn't just maintain the invariance of $TCConsistent$; it implements the specification of transaction commit.

The two-phase commit protocol doesn't just maintain the same invariant $TCConsistent$ as transaction commit;
it actually implements the transaction commit specification.

Two-phase commit doesn't just maintain the invariance of $TCConsistent$; it implements the specification of transaction commit.

What does that mean?

The two-phase commit protocol doesn't just maintain the same invariant $TCConsistent$ as transaction commit;
it actually implements the transaction commit specification.

But just what does that mean?

Two-phase commit doesn't just maintain the invariance of $TCConsistent$; it implements the specification of transaction commit.

What does that mean?

A later lecture will explain precisely what it means

The two-phase commit protocol doesn't just maintain the same invariant $TCConsistent$ as transaction commit;
it actually implements the transaction commit specification.

But just what does that mean?

In a later lecture, I'll explain precisely what it means for the $TwoPhase$ spec to implement the $TCommit$ spec

Two-phase commit doesn't just maintain the invariance of $TCConsistent$; it implements the specification of transaction commit.

What does that mean?

A later lecture will explain precisely what it means, and how to check that it does.

and I'll show how to check that it does.

The Two-Phase Commit specification is bigger than the Die Hard and Transaction Commit specs. It's still small and simple, but we're on the path towards specifying real systems. And you're well on the way to learning the TLA+ you'll need to start writing your own specs.

In the next lecture, you'll see a real spec of a real distributed algorithm.

End of Lecture 6

TWO-PHASE COMMIT