TLA$^+$ Video Course – Lecture 8, Part 1

Leslie Lamport

IMPLEMENTATION

PRELIMINARIES

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 8, Part 1
Implementation: Preliminaries

This lecture explains what it means for the two-phase commit protocol to implement the specification of transaction commit. It's divided into two parts. Part One reviews and categorizes the kinds of TLA+ expressions you've already seen, and introduces a new kind: temporal formulas. A specification can be written as a single temporal formula. We begin with an explanation of logical implication.

[slide 2]

IMPLICATION

$$P \Rightarrow Q$$

This formula asserts that

$P \Rightarrow Q$

If P is true then Q is true.

This formula asserts that

If formula P is true then formula Q is true.

$P \boxed{\Rightarrow} Q$

This formula asserts that

If formula P is true then formula Q is true.

This symbol is read *implies* and is typed

$$P \;\Rightarrow\; Q$$

=>

This formula asserts that

If formula P is true then formula Q is true.

This symbol is read *implies* and is typed equals greater than.

[slide 7]

$P \Rightarrow Q$ equals

The formula P implies Q equals

$P \Rightarrow Q$ equals

IF P

The formula P implies Q equals

If P is true

$P \Rightarrow Q$ equals

IF P THEN Q

The formula P implies Q equals
If P is true then Q is true.

$P \Rightarrow Q$ equals

IF P THEN Q

ELSE we know nothing

The formula P implies Q equals
If P is true then Q is true.
Else, we know nothing.

The way we assert mathematically that we know nothing is

$P \Rightarrow Q$ equals

IF P THEN Q
ELSE TRUE

The formula P implies Q equals
If P is true then Q is true.
Else, we know nothing.

The way we assert mathematically that we know nothing is with the formula TRUE. Since saying that TRUE is true says nothing.

[slide 12]

$P \Rightarrow Q$ equals

A useful property of *implies* is that P implies Q equals

$P \Rightarrow Q$ **equals** $\neg Q \Rightarrow \neg P$

A useful property of *implies* is that P implies Q equals not Q implies not P.

$P \Rightarrow Q$ equals $\neg Q \Rightarrow \neg P$

because

IF P THEN Q
ELSE TRUE

A useful property of *implies* is that P implies Q equals not Q implies not P.

That's true because the definition of P implies Q

$P \Rightarrow Q$ equals $\neg Q \Rightarrow \neg P$

because

IF P THEN Q ELSE TRUE equals IF $\neg Q$ THEN $\neg P$ ELSE TRUE

A useful property of *implies* is that P implies Q equals not Q implies not P.

That's true because the definition of P implies Q

equals the definition of not Q implies not P.

$P \Rightarrow Q$ equals $\neg Q \Rightarrow \neg P$

because

IF	P	THEN	Q	equals	IF	$\neg Q$	THEN	$\neg P$
		ELSE	TRUE				ELSE	TRUE

We can check this by substituting all combinations of Boolean values for P and Q.

A useful property of *implies* is that P implies Q equals not Q implies not P.

That's true because the definition of P implies Q

equals the definition of not Q implies not P.

We can check this by substituting all four possible combinations of Boolean values for P and Q.

$P \Rightarrow Q$ equals $\neg Q \Rightarrow \neg P$

because

IF P THEN Q ELSE TRUE equals IF $\neg Q$ THEN $\neg P$ ELSE TRUE

For example: $P \leftarrow$ TRUE and $Q \leftarrow$ FALSE

For example, let's substitute TRUE for P and FALSE for Q.

$P \Rightarrow Q$ equals $\neg Q \Rightarrow \neg P$

because

IF TRUE THEN FALSE ELSE TRUE equals IF $\neg$FALSE THEN $\neg$TRUE ELSE TRUE

For example: $P \leftarrow$ TRUE and $Q \leftarrow$ FALSE

For example, let's substitute TRUE for P and FALSE for Q. like this.

$P \Rightarrow Q$ equals $\neg Q \Rightarrow \neg P$

because

$\boxed{\begin{array}{lll} \text{IF TRUE THEN} & \text{FALSE} \\ \quad\text{ELSE} & \text{TRUE} \end{array}}$ equals $\begin{array}{lll} \text{IF } \neg\text{FALSE THEN} & \neg\text{TRUE} \\ \quad\text{ELSE} & \text{TRUE} \end{array}$

For example, let's substitute TRUE for P and FALSE for Q. like this.

Evaluating this IF / THEN / ELSE expression yields

$P \Rightarrow Q$ equals $\neg Q \Rightarrow \neg P$

because

FALSE equals IF $\neg$FALSE THEN $\neg$TRUE
ELSE TRUE

For example, let's substitute TRUE for P and FALSE for Q. like this.

Evaluating this IF / THEN / ELSE expression yields the value FALSE.

$P \Rightarrow Q$ equals $\neg Q \Rightarrow \neg P$

because

FALSE equals IF $\boxed{\neg \text{FALSE}}$ THEN $\neg$TRUE
ELSE TRUE

For example, let's substitute TRUE for P and FALSE for Q. like this.

Evaluating this IF / THEN / ELSE expression yields the value FALSE.

Not FALSE

$P \Rightarrow Q$ equals $\neg Q \Rightarrow \neg P$

because

FALSE equals IF TRUE THEN ¬TRUE
ELSE TRUE

For example, let's substitute TRUE for P and FALSE for Q. like this.

Evaluating this IF / THEN / ELSE expression yields the value FALSE.

Not FALSE equals TRUE. So this IF / THEN / ELSE equals

$P \Rightarrow Q$ equals $\neg Q \Rightarrow \neg P$

because

FALSE equals $\neg$TRUE

For example, let's substitute TRUE for P and FALSE for Q. like this.

Evaluating this IF / THEN / ELSE expression yields the value FALSE.

Not FALSE equals TRUE. So this IF / THEN / ELSE equals not TRUE, which equals

$P \Rightarrow Q$ equals $\neg Q \Rightarrow \neg P$

because

FALSE equals FALSE

For example, let's substitute TRUE for P and FALSE for Q. like this.

Evaluating this IF / THEN / ELSE expression yields the value FALSE.

Not FALSE equals TRUE. So this IF / THEN / ELSE equals not TRUE, which equals FALSE, so the two formulas are equal for this substitution of Boolean values for P and Q.

$P \Rightarrow Q$ equals $\neg Q \Rightarrow \neg P$

because

IF P THEN Q ELSE TRUE equals IF $\neg Q$ THEN $\neg P$ ELSE TRUE

$P \Rightarrow Q$ equals $\neg Q \Rightarrow \neg P$

because

IF	P	THEN	Q	equals	IF	$\neg Q$	THEN	$\neg P$
		ELSE	TRUE				ELSE	TRUE

You can check the other values of P and Q.

You can check the other three possible substitutions of Boolean values for P and Q yourself.

$P \Rightarrow Q$ equals $\neg Q \Rightarrow \neg P$

Let's take a closer look at this equality.

$P \Rightarrow Q$ equals $\neg Q \Rightarrow \neg P$

Let's substitute: $P \leftarrow$ it's raining

$Q \leftarrow$ the ground is wet

Let's take a closer look at this equality.

Suppose we substitute "it's raining" for P and "the ground is wet" for Q.

$P \Rightarrow Q$ equals $\neg Q \Rightarrow \neg P$

If it's raining then the ground is wet.

Let's take a closer look at this equality.

Suppose we substitute "it's raining" for P and "the ground is wet" for Q.

The equality of these two formulas means that "If it's raining then the ground is wet."

[slide 30]

$P \Rightarrow Q$ equals $\neg Q \Rightarrow \neg P$

If it's raining then the ground is wet.

has the same meaning as

If the ground is not wet then it's not raining.

Means the same thing as “If the ground is not wet then it's not raining.”

But does it?

$P \Rightarrow Q$ equals $\neg Q \Rightarrow \neg P$

If it's raining then the ground is wet.

has the same meaning as

If the ground is not wet then it's not raining.

Means the same thing as "If the ground is not wet then it's not raining."

But does it?

This sounds right.

$P \Rightarrow Q$ equals $\neg Q \Rightarrow \neg P$

If it's raining then the ground is wet.

has the same meaning as

If the ground is not wet then it's not raining.

Means the same thing as "If the ground is not wet then it's not raining."

But does it?

This sounds right. But this doesn't. That's because

$P \Rightarrow Q$ equals $\neg Q \Rightarrow \neg P$

If it's raining then the ground is wet.

has the same meaning as

If the ground is not wet then it's not raining.

In speech, implication asserts causality.

Means the same thing as "If the ground is not wet then it's not raining."

But does it?

This sounds right. But this doesn't. That's because in ordinary speech, implication asserts causality.

$P \Rightarrow Q$ equals $\neg Q \Rightarrow \neg P$

If it's raining then the ground is wet.

has the same meaning as

If the ground is not wet then it's not raining.

In speech, implication asserts causality.

Raining causes the ground to be wet.

$P \Rightarrow Q$ equals $\neg Q \Rightarrow \neg P$

If it's raining then the ground is wet.

has the same meaning as

If the ground is not wet then it's not raining.

In speech, implication asserts causality.

Raining causes the ground to be wet.

But, the ground not being wet doesn't cause it not to be raining.

So in ordinary speech, these two sentences don't have the same meaning.

$P \Rightarrow Q$ equals $\neg Q \Rightarrow \neg P$

If it's raining then the ground is wet.

has the same meaning as

If the ground is not wet then it's not raining.

In speech, implication asserts causality.

In math, implication asserts only correlation.

But in math and hence in TLA+, implication asserts only correlation, not causality.

In math, these two sentences and these two formulas have the same meaning. And TLA+ is math.

ORDINARY EXPRESSIONS

A *module-closed* expression is a TLA+
expression that

Let's define a module-closed expression of a module to be a TLA+
expression that

A *module-closed* expression is a TLA$^+$
expression that

(after expanding definitions)

Let's define a module-closed expression of a module to be a TLA$^+$
expression that (after expanding all definitions)

A *module-closed* expression is a TLA+ expression that contains only:

Let's define a module-closed expression of a module to be a TLA+ expression that (after expanding all definitions) contains only:

A *module-closed* expression is a TLA^{+} expression that contains only:

– built-in TLA^{+} operators and constructs,

Let's define a module-closed expression of a module to be a TLA^{+} expression that (after expanding all definitions) contains only:

built-in TLA^{+} operators and constructs.

A *module-closed* expression is a TLA^+ expression that contains only:

– built-in TLA^+ operators and constructs,

– numbers and strings

numbers and strings

A *module-closed* expression is a TLA+ expression that contains only:

- built-in TLA+ operators and constructs,
- numbers and strings, like 42 and "abc"

numbers and strings like 42 and the string abc.

A *module-closed* expression is a TLA$^+$ expression that contains only:

- built-in TLA$^+$ operators and constructs,
- numbers and strings,
- declared constants and variables,

numbers and strings

Identifiers declared in the module's CONSTANT and VARIABLE statements.

A *module-closed* expression is a TLA$^+$ expression that contains only:

- built-in TLA$^+$ operators and constructs,
- numbers and strings,
- declared constants and variables,
- identifiers declared locally within it.

numbers and strings

Identifiers declared in the module's CONSTANT and VARIABLE statements.

And identifiers declared locally within the expression.

A *module-closed* expression is a TLA^+ expression that contains only:

– identifiers declared locally within it.

Locally declared identifiers

A *module-closed* expression is a TLA$^+$ expression that contains only:

– identifiers declared locally within it.

Including ones introduced by:

Locally declared identifiers include identifiers introduced by these constructs occurring in the expression:

A *module-closed* expression is a TLA$^+$ expression that contains only:

– identifiers declared locally within it.

Including ones introduced by:

$\forall \boxed{v} \in S : \ \ldots$ and $\exists \boxed{v} \in S : \ \ldots$

Locally declared identifiers include identifiers introduced by these constructs occurring in the expression:

Forall and exists.

A *module-closed* expression is a TLA$^+$ expression that contains only:

– identifiers declared locally within it.

Including ones introduced by:

$\forall \boxed{v} \in S : \ldots$ and $\exists \boxed{v} \in S : \ldots$

$[\boxed{v} \in S \mapsto \ldots]$

Locally declared identifiers include identifiers introduced by these constructs occurring in the expression:

Forall and exists.

This function constructor.

A *module-closed* expression is a TLA+ expression that contains only:

- identifiers declared locally within it.

  Including ones introduced by:

  $\forall \boxed{v} \in S : \ \ldots$ and $\exists \boxed{v} \in S : \ \ldots$

  $[\boxed{v} \in S \mapsto \ \ldots]$

  $\{\boxed{v} \in S : \ \ldots\}$ and $\{\ldots : \boxed{v} \in S\}$

Locally declared identifiers include identifiers introduced by these constructs occurring in the expression:

Forall and exists.

This function constructor.

And these set constructors.

This expression is module-complete

$$\exists\, v \in Nat : x' = x + v$$

For example, this expression is module-complete

This expression is module-complete

$$\exists\, v \in Nat : x' = x + v$$

if x is a declared variable.

For example, this expression is module-complete if x is a declared variable.

This expression is module-complete

$$\exists v \in Nat : \boxed{x' = x + v}$$

This subexpression is not module-complete

For example, this expression is module-complete if x is a declared variable.

But this subexpression is not module-complete

This expression is module-complete

$$\exists v \in Nat : \boxed{x' = x + \boxed{v}}$$

This subexpression is not module-complete because v is locally declared outside it.

For example, this expression is module-complete if x is a declared variable.

But this subexpression is not module-complete because v is locally declared outside the subexpression.

A module-closed *formula* is a Boolean-valued module-closed expression.

A module-closed formula is a Boolean-valued module-closed expression.

A module-closed *formula* is a Boolean-valued module-closed expression.

(One whose value is either TRUE or FALSE.)

A module-closed formula is a Boolean-valued module-closed expression.

That is, one whose value is either TRUE or FALSE.

For example,

A module-closed *formula* is a Boolean-valued module-closed expression.

$$(x \in 1 \,..\, 42) \;\wedge\; (y' = x + 1)$$

A module-closed formula is a Boolean-valued module-closed expression.

That is, one whose value is either TRUE or FALSE.

For example, this expression – assuming x and y are declared variables.

A module-closed *formula* is a Boolean-valued module-closed expression.

$$(x \in 1\,..\,42) \;\wedge\; (y' = x + 1)$$

Be aware that quite a few people use the word *formula* to mean any mathematical expression. But I'll use it to mean a Boolean-valued expression.

For this lecture:

Just for this lecture:

For this lecture:

– expression means module-closed expression

Just for this lecture:

expression will mean *module-closed expression*

For this lecture:

– expression means module-closed expression

– formula means module-closed formula

Just for this lecture:

expression will mean *module-closed expression*

and *formula* will mean *module-closed formula*.

Constant Expressions

Constant Expressions.

Constant Expressions

A constant expression is a (module-complete) expression that

Constant Expressions.

A constant expression is a (module-complete) expression that

Constant Expressions

A constant expression is a (module-complete) expression that
(after expanding all definitions)

Constant Expressions.

A constant expression is a (module-complete) expression that

(after expanding all definitions)

Constant Expressions

A constant expression is a (module-complete) expression that

- Has no declared variables.

Constant Expressions.

A constant expression is a (module-complete) expression that

(after expanding all definitions)

Has no declared variables.

Constant Expressions

A constant expression is a (module-complete) expression that

- Has no declared variables.
- Has no non-constant operators.

And has no non-constant operators.

Constant Expressions

A constant expression is a (module-complete) expression that

- Has no declared variables.
- Has no non-constant operators.

  The only ones you've seen so far are $'$ (prime) and UNCHANGED.

And has no non-constant operators.

The only non-constant operators that you've seen so far are prime and UNCHANGED.

The value of a constant expression

The value of a constant expression

The value of a constant expression

$$foo \cup \{ n \in 1 \,..\, 22 \ : n^2 > m \}$$

The value of a constant expression like this one

The value of a constant expression

$$\boxed{foo} \cup \{ n \in 1 \,..\, 22 \; : n^2 > \boxed{m} \}$$

depends only on the values of the declared constants it contains.

The value of a constant expression like this one

depends only on the values of the declared constants it contains.
In this example, those are the constants foo and m.

The value of a constant expression

$$foo \cup \{\boxed{n} \in 1 \,..\, 22 : \boxed{n}^2 > m\}$$

depends only on the values of the declared constants it contains.

The value of a constant expression like this one

depends only on the values of the declared constants it contains.
In this example, those are the constants foo and m.

The constant n is locally defined within the expression.

An assumption

An assumption

An assumption

ASSUME $\boxed{\ldots}$

An assumption

which is asserted by an ASSUME statement

An assumption

ASSUME $\boxed{\ldots}$

must be a constant formula.

An assumption

which is asserted by an ASSUME statement

must be a constant formula.
Remember that a constant formula is a Boolean-valued constant expression.

[slide 75]

State Expressions

State expressions.

State Expressions

A state expression can contain anything a constant expression can

State expressions.

A state expression is an expression that can contain anything a constant expression can contain

State Expressions

A state expression can contain anything a constant expression can as well as declared variables.

State expressions.

A state expression is an expression that can contain anything a constant expression can contain as well as variables declared in a VARIABLES statement.

State Expressions

A state expression can contain anything a constant expression can as well as declared variables.

$$x + y[foo]$$

State expressions.

A state expression is an expression that can contain anything a constant expression can contain as well as variables declared in a VARIABLES statement.

For example, this is a state expression,

State Expressions

A state expression can contain anything a constant expression can as well as declared variables.

$$x + y[foo] \quad \text{if} \quad \begin{array}{l} \text{CONSTANT } foo \\ \text{VARIABLES } x,\ y \end{array}$$

State expressions.

A state expression is an expression that can contain anything a constant expression can contain as well as variables declared in a VARIABLES statement.

For example, this is a state expression, if foo is a declared constant and x and y are declared variables.

State Expressions

A state expression can contain anything a constant expression can as well as declared variables.

$$x + y[foo]$$

State expressions.

A state expression is an expression that can contain anything a constant expression can contain as well as variables declared in a VARIABLES statement.

For example, this is a state expression, if foo is a declared constant and x and y are declared variables.

State Expressions

A state expression can contain anything a constant expression can as well as declared variables.

$$x + y[foo]$$

The value of a state expression depends on:

The value of a state expression depends on:

State Expressions

A state expression can contain anything a constant expression can as well as declared variables.

$$x + y[foo]$$

The value of a state expression depends on:

– The values of declared constants.

The value of a state expression depends on:
The values of declared constants.

State Expressions

A state expression can contain anything a constant expression can as well as declared variables.

$$x + y[foo]$$

The value of a state expression depends on:

– The values of declared constants.

– The values of declared variables.

The value of a state expression depends on:
The values of declared constants.
and the values of declared variables.

State Expressions

A state expression can contain anything a constant expression can as well as declared variables.

$$x + y[foo]$$

The value of a state expression depends on:

– The values of declared constants.

– The values of declared variables.

I will ignore dependence on the values of declared constants.

The value of a state expression depends on:
The values of declared constants.
and the values of declared variables.
I will ignore all dependencies on the values of declared constants

State Expressions

A state expression can contain anything a constant expression can as well as declared variables.

$$x + y[foo]$$

The value of a state expression depends on:

– The values of declared variables.

I will ignore dependence on the values of declared constants.

The value of a state expression depends on:
The values of declared constants.
and the values of declared variables.
I will ignore all dependencies on the values of declared constants
and assume that the values of all declared constants are fixed throughout the discussion. And I'll avoid declared constants in the examples I use.

A state expression has a value on a state.

A state expression has a value on a state.

A state expression has a value on a state.

Remember that a state assigns values to variables.

A state expression has a value on a state.

Remember that a state assigns values to variables.

A state expression has a value on a state.

Remember that a state assigns values to variables.

If state s assigns $v \leftarrow Nat$ and $w \leftarrow -42$

A state expression has a value on a state.

Remember that a state assigns values to variables.

If state s assigns the set Nat of natural numbers to variable v and the number -42 to variable w,

A state expression has a value on a state.

Remember that a state assigns values to variables.

If state s assigns $v \leftarrow Nat$ and $w \leftarrow -42$, then

$v \cup \{w\}$

A state expression has a value on a state.

Remember that a state assigns values to variables.

If state s assigns the set Nat of natural numbers to variable v and the number -42 to variable w,

then this state expression

A state expression has a value on a state.

Remember that a state assigns values to variables.

If state s assigns $v \leftarrow Nat$ and $w \leftarrow -42$, then

$v \cup \{w\}$ has the value $Nat \cup \{-42\}$

A state expression has a value on a state.

Remember that a state assigns values to variables.

If state s assigns the set Nat of natural numbers to variable v and the number -42 to variable w,

then this state expression has this value

A state expression has a value on a state.

Remember that a state assigns values to variables.

If state s assigns $v \leftarrow Nat$ and $w \leftarrow -42$, then

$v \cup \{w\}$ has the value $Nat \cup \{-42\}$

on state s.

A state expression has a value on a state.

Remember that a state assigns values to variables.

If state s assigns the set Nat of natural numbers to variable v and the number -42 to variable w,

then this state expression has this value on state s.

A constant expression is a state expression that has the same value on all states.

A constant expression is a state expression that has the same value on all states.

A constant expression is a state expression that has the same value on all states.

The constant expression $2+2$ has the value 4 on every state.

A constant expression is a state expression that has the same value on all states.

The constant expression $2+2$ has the value 4 on every state.

Action Expressions

Action Expressions.

Action Expressions

An action expression can contain anything a state expression can

Action Expressions.

An action expression can contain anything a state expression can

Action Expressions

An action expression can contain anything a state expression can as well as $'$ (prime) and UNCHANGED.

Action Expressions.

An action expression can contain anything a state expression can as well as prime and UNCHANGED.

Action Expressions

An action expression can contain anything a state expression can as well as ′ (prime) and UNCHANGED.

A state expression has a value on a step (pair of states).

Action Expressions.

An action expression can contain anything a state expression can as well as prime and UNCHANGED. A state expression has a value on a step (remember that a step is a pair of states).

Action Expressions

An action expression can contain anything a state expression can as well as $'$ (prime) and UNCHANGED.

A state expression has a value on a step (pair of states).

If state s assigns $p \leftarrow 42$

Action Expressions.

An action expression can contain anything a state expression can as well as prime and UNCHANGED. A state expression has a value on a step (remember that a step is a pair of states).

If state s assigns the value 42 to variable p

Action Expressions

An action expression can contain anything a state expression can as well as $'$ (prime) and UNCHANGED.

A state expression has a value on a step (pair of states).

If state s assigns $p \leftarrow 42$ and
state t assigns $q \leftarrow 24$

Action Expressions.

An action expression can contain anything a state expression can as well as prime and UNCHANGED. A state expression has a value on a step (remember that a step is a pair of states).

If state s assigns the value 42 to variable p
and state t assigns the value 24 to variable q

Action Expressions

An action expression can contain anything a state expression can as well as $'$ (prime) and UNCHANGED.

A state expression has a value on a step (pair of states).

If state s assigns $p \leftarrow 42$ and
state t assigns $q \leftarrow 24$, then

$$p - q'$$

then the action expression $p - q'$

Action Expressions

An action expression can contain anything a state expression can as well as $'$ (prime) and UNCHANGED.

A state expression has a value on a step (pair of states).

If state s assigns $p \leftarrow 42$ and
state t assigns $q \leftarrow 24$, then

$p - q'$ has the value $42 - 24$

then the action expression $p - q'$
has the value $42 - 24$, (which equals 18)

Action Expressions

An action expression can contain anything a state expression can as well as $'$ (prime) and UNCHANGED.

A state expression has a value on a step (pair of states).

If state s assigns $p \leftarrow 42$ and
state t assigns $q \leftarrow 24$, then

$p - q'$ has the value $42 - 24$

on the step $s \rightarrow t$.

then the action expression $p - q'$
has the value $42 - 24$, (which equals 18)
on the step s t.

A state expression is an action expression whose value on the step $s \rightarrow t$ depends only on state s .

A state expression is an action expression whose value on the step $s \quad t$ depends only on the first state s.

A state expression is an action expression whose value on the step $s \rightarrow t$ depends only on state s .

An action formula is called an action.

A state expression is an action expression whose value on the step s t depends only on the first state s.

An action *formula* is called simply an action.

Priming a State Expression

So far we've only primed variables. We can actually prime any state expression.

Priming a State Expression

For any state expression e the value of the action expression e' on $s \rightarrow t$ is the value of e on state t.

So far we've only primed variables. We can actually prime any state expression.

For any state expression e, the value of the action expression e prime on the step $s \quad t$ is the value of e on state t.

Priming a State Expression

For any state expression e the value of the action expression e' on $s \rightarrow t$ is the value of e on state t.

UNCHANGED e equals $e' = e$

So far we've only primed variables. We can actually prime any state expression.

For any state expression e, the value of the action expression e prime on the step s t is the value of e on state t.

UNCHANGED of an expression e is defined to equal the formula $e' = e$.

Priming a State Expression

For any state expression e the value of the action expression e' on $s \rightarrow t$ is the value of e on state t.

UNCHANGED e equals $e' = e$

UNCHANGED $\langle x, y, z \rangle$

So far we've only primed variables. We can actually prime any state expression.

For any state expression e, the value of the action expression e prime on the step s t is the value of e on state t.

UNCHANGED of an expression e is defined to equal the formula $e' = e$.

Therefore, UNCHANGED of a triple x, y, z

Priming a State Expression

For any state expression e the value of the action expression e' on $s \rightarrow t$ is the value of e on state t.

UNCHANGED e equals $e' = e$

UNCHANGED $\langle x, y, z \rangle$ equals $\langle x, y, z \rangle' = \langle x, y, z \rangle$

by definition of UNCHANGED is equivalent to *the triple primed equals the triple*.

Priming a State Expression

For any state expression e the value of the action expression e' on $s \rightarrow t$ is the value of e on state t.

UNCHANGED e equals $e' = e$

UNCHANGED $\langle x, y, z \rangle$ equals $\boxed{\langle x, y, z \rangle'} = \langle x, y, z \rangle$

$\langle x', y', z' \rangle$

by definition of UNCHANGED is equivalent to *the triple primed equals the triple*.

The value of a triple in the next state is the triple of the values of its components in the next state,

Priming a State Expression

For any state expression e the value of the action expression e' on $s \to t$ is the value of e on state t.

UNCHANGED e equals $e' = e$

UNCHANGED $\langle x, y, z \rangle$ equals $\langle x, y, z \rangle' = \langle x, y, z \rangle$

equals $\langle x', y', z' \rangle = \langle x, y, z \rangle$

by definition of UNCHANGED is equivalent to *the triple primed equals the triple*.

The value of a triple in the next state is the triple of the values of its components in the next state, so we have this equality of formulas.

Priming a State Expression

For any state expression e the value of the action expression e' on $s \rightarrow t$ is the value of e on state t.

UNCHANGED e equals $e' = e$

UNCHANGED $\langle x, y, z \rangle$ equals $\langle x, y, z \rangle' = \langle x, y, z \rangle$

equals $\langle x', y', z' \rangle = \langle x, y, z \rangle$

equals $(x' = x) \wedge (y' = y) \wedge (z' = z)$

by definition of UNCHANGED is equivalent to *the triple primed equals the triple*.

The value of a triple in the next state is the triple of the values of its components in the next state, so we have this equality of formulas. Which in turn gives us this formula, since two triples are equal if and only if their corresponding components are equal.

Priming a State Expression

For any state expression e the value of the action expression e' on $s \rightarrow t$ is the value of e on state t.

UNCHANGED e equals $e' = e$

UNCHANGED $\langle x, y, z \rangle$ equals $\langle x, y, z \rangle' = \langle x, y, z \rangle$

equals $\langle x', y', z' \rangle = \langle x, y, z \rangle$

equals $(x' = x) \wedge (y' = y) \wedge (z' = z)$

by definition of UNCHANGED is equivalent to *the triple primed equals the triple*.

The value of a triple in the next state is the triple of the values of its components in the next state, so we have this equality of formulas. Which in turn gives us this formula, since two triples are equal if and only if their corresponding components are equal.

TEMPORAL FORMULAS

Temporal Formulas

A temporal formula

A temporal formula is something we haven't seen before.

A temporal formula has a Boolean value on a sequence $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \cdots$ of states.

A temporal formula is something we haven't seen before.

It has a Boolean value on a sequence of states.

A temporal formula has a Boolean value on a sequence
$s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \cdots$ of states.

TLA$^+$ has only Boolean-valued temporal expressions.

A temporal formula is something we haven't seen before.

It has a Boolean value on a sequence of states.

TLA$^+$ has only Boolean-valued temporal expressions – that is, temporal formulas.

A temporal formula has a Boolean value on a sequence $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \cdots$ of states.

A temporal formula is something we haven't seen before.

It has a Boolean value on a sequence of states.

TLA$^+$ has only Boolean-valued temporal expressions – that is, temporal formulas.

A temporal formula has a Boolean value on a sequence $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \cdots$ of states.

A sequence of states

A temporal formula has a Boolean value on a behavior $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \cdots$.

A sequence of states is just what we've been calling a *behavior*.

A temporal formula has a Boolean value on a behavior $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \cdots$.

We will now write a specification as a temporal formula

A sequence of states is just what we've been calling a *behavior*.

We will now write a specification as a temporal formula

A temporal formula has a Boolean value on a behavior $s_1 \to s_2 \to s_3 \to \cdots$.

We will now write a specification as a temporal formula –
a formula whose value is TRUE on the behaviors allowed by the spec.

A sequence of states is just what we've been calling a *behavior*.

We will now write a specification as a temporal formula – a formula whose value is TRUE on just those behaviors that are allowed by the spec.

As an example,

A temporal formula has a Boolean value on a behavior
$s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \cdots$.

We will now write a specification as a temporal formula –
a formula whose value is TRUE on the behaviors allowed by the spec.

We now define $TPSpec$ to be the specification
of the two-phase commit protocol.

A sequence of states is just what we've been calling a *behavior*.

We will now write a specification as a temporal formula – a formula whose value is TRUE on just those behaviors that are allowed by the spec.

As an example, we now define the temporal formula $TPSpec$ to be the specification of the two-phase commit protocol.

The two-phase commit spec has

initial formula $TPInit$

next-state formula $TPNext$

Recall that the two-phase commit spec has initial formula $TPInit$ and next-state formula $TPNext$.

The two-phase commit spec has

initial formula $TPInit$

next-state formula $TPNext$

$TPSpec$ should be true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ iff

Recall that the two-phase commit spec has initial formula $TPInit$ and next-state formula $TPNext$.

The temporal formula $TPSpec$ should be true on a behavior if and only if:

The two-phase commit spec has

initial formula $TPInit$

next-state formula $TPNext$

$TPSpec$ should be true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ $\boxed{\text{iff}}$

if and only if

Recall that the two-phase commit spec has initial formula $TPInit$ and next-state formula $TPNext$.

The temporal formula $TPSpec$ should be true on a behavior if and only if:

This is an abbreviation for if and only if.

The two-phase commit spec has

initial formula $TPInit$

next-state formula $TPNext$

$TPSpec$ should be true on $\boxed{s_1} \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ iff

$TPInit$ is true on s_1

Recall that the two-phase commit spec has initial formula $TPInit$ and next-state formula $TPNext$.

The temporal formula $TPSpec$ should be true on a behavior if and only if:

This is an abbreviation for if and only if.

$TPSpec$ should be true on the behavior if and only if $TPInit$ is true on the behavior's first state.

The two-phase commit spec has

initial formula $TPInit$

next-state formula $TPNext$

$TPSpec$ should be true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ iff

$TPInit$ is true on s_1

$TPNext$ is true on $s_i \rightarrow s_{i+1}$ for all i

And $TPNext$ is true on all steps

The two-phase commit spec has

initial formula $TPInit$

next-state formula $TPNext$

$TPSpec$ should be true on $\boxed{s_1 \rightarrow s_2} \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ iff

$TPInit$ is true on s_1

$TPNext$ is true on $s_i \rightarrow s_{i+1}$ for all i

And $TPNext$ is true on all steps

The two-phase commit spec has

initial formula $TPInit$

next-state formula $TPNext$

$TPSpec$ should be true on $s_1 \rightarrow \boxed{s_2 \rightarrow s_3} \rightarrow s_4 \rightarrow \cdots$ iff

$TPInit$ is true on s_1

$TPNext$ is true on $s_i \rightarrow s_{i+1}$ for all i

And $TPNext$ is true on all steps

The two-phase commit spec has

initial formula $TPInit$

next-state formula $TPNext$

$TPSpec$ should be true on $s_1 \rightarrow s_2 \rightarrow \boxed{s_3 \rightarrow s_4} \rightarrow \cdots$ iff

$TPInit$ is true on s_1

$TPNext$ is true on $s_i \rightarrow s_{i+1}$ for all i

And $TPNext$ is true on all steps

The two-phase commit spec has

initial formula $TPInit$

next-state formula $TPNext$

$TPSpec$ should be true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ iff

$TPInit$ is true on s_1

$TPNext$ is true on $s_i \rightarrow s_{i+1}$ for all i

And $TPNext$ is true on all steps of the behavior.

$TPSpec$ should be true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ iff

$TPInit$ is true on s_1

$TPNext$ is true on $s_i \rightarrow s_{i+1}$ for all i

Let's consider the first condition.

When the state formula $TPInit$ is considered to be an action…

$TPSpec$ should be true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ iff

$TPInit$ is true on s_1

$TPNext$ is true on $s_i \rightarrow s_{i+1}$ for all i

The value of $TPInit$ on $s_1 \rightarrow s_2$
equals value on s_1 .

Let's consider the first condition.

When the state formula $TPInit$ is considered to be an action... its value on a step equals its value on the first state.

Similarly, when we consider it to be a temporal formula...

$TPSpec$ should be true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ iff

$TPInit$ is true on s_1

$TPNext$ is true on $s_i \rightarrow s_{i+1}$ for all i

The value of $TPInit$ on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$

equals value on s_1 .

Let's consider the first condition.

When the state formula $TPInit$ is considered to be an action... its value on a step equals its value on the first state.

Similarly, when we consider it to be a temporal formula... the same is true for its value on a *behavior*.

$TPSpec$ should be true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ iff

$TPInit$ is true on s_1

$TPNext$ is true on $s_i \rightarrow s_{i+1}$ for all i

$TPInit$ is true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ iff
it is true on s_1 .

Which means $TPInit$ is true on the behavior if and only if it's true on the behavior's first state.

$TPSpec$ should be true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ iff

$TPInit$ is true on s_1

$TPNext$ is true on $s_i \rightarrow s_{i+1}$ for all i

$TPInit$ is true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ iff it is true on s_1 .

Which means $TPInit$ is true on the behavior if and only if it's true on the behavior's first state.

So this first condition can be written

$TPSpec$ should be true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ iff

$TPInit$ is true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$

$TPNext$ is true on $s_i \rightarrow s_{i+1}$ for all i

$TPInit$ is true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ iff
it is true on s_1 .

Which means $TPInit$ is true on the behavior if and only if it's true on the behavior's first state.

So this first condition can be written like this.

$TPSpec$ should be true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ iff

$TPInit$ is true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$

$TPNext$ is true on $s_i \rightarrow s_{i+1}$ for all i

$TPInit$ is true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ iff
it is true on s_1 .

$TPSpec$ should be true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ iff

$TPInit$ is true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$

$TPNext$ is true on $s_i \rightarrow s_{i+1}$ for all i

$TPInit$ is true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ iff it is true on s_1.

A state formula like $TPInit$ is true on a behavior if and only if it's true on the first state of the behavior.

Similarly

$TPSpec$ should be true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ iff

$TPInit$ is true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$

$TPNext$ is true on $s_i \rightarrow s_{i+1}$ for all i

$TPNext$ is true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ iff it is true on $s_1 \rightarrow s_2$.

A state formula like $TPInit$ is true on a behavior if and only if it's true on the first state of the behavior.

Similarly an action like $TPNext$ is true on a behavior if and only if it's true on the first step of the behavior.

$TPSpec$ should be true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ iff

$TPInit$ is true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$

$TPNext$ is true on $s_i \rightarrow s_{i+1}$ for all i

$TPNext$ is true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ iff it is true on $s_1 \rightarrow s_2$.

$$\Box\, TPNext$$

A state formula like $TPInit$ is true on a behavior if and only if it's true on the first state of the behavior.

Similarly an action like $TPNext$ is true on a behavior if and only if it's true on the first step of the behavior.

If we apply this temporal operator to the action $TPNext$

$TPSpec$ should be true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ iff

$TPInit$ is true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$

$TPNext$ is true on $s_i \rightarrow s_{i+1}$ for all i

$TPNext$ is true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ iff it is true on $s_1 \rightarrow s_2$.

$\Box\, TPNext$

[ ] in ASCII

A state formula like $TPInit$ is true on a behavior if and only if it's true on the first state of the behavior.

Similarly an action like $TPNext$ is true on a behavior if and only if it's true on the first step of the behavior.

If we apply this temporal operator to the action $TPNext$

This operator is typed *left bracket right bracket*

$TPSpec$ should be true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ iff

$TPInit$ is true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$

$TPNext$ is true on $s_i \rightarrow s_{i+1}$ for all i

$TPNext$ is true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ iff it is true on $s_1 \rightarrow s_2$.

$\Box\, TPNext$

[ ] in ASCII Read *always*

A state formula like $TPInit$ is true on a behavior if and only if it's true on the first state of the behavior.

Similarly an action like $TPNext$ is true on a behavior if and only if it's true on the first step of the behavior.

If we apply this temporal operator to the action $TPNext$

This operator is typed *left bracket right bracket* and is read *always*.

$TPSpec$ should be true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ iff

$TPInit$ is true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$

$TPNext$ is true on $s_i \rightarrow s_{i+1}$ for all i

$TPNext$ is true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ iff it is true on $s_1 \rightarrow s_2$.

$$\Box\, TPNext$$

The temporal formula *always* $TPNext$

$TPSpec$ should be true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ iff

$TPInit$ is true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$

$TPNext$ is true on $s_i \rightarrow s_{i+1}$ for all i

$TPNext$ is true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ iff it is true on $s_1 \rightarrow s_2$.

$\Box\, TPNext$ is true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ iff

The temporal formula *always* $TPNext$ is true on a behavior if and only if

$TPSpec$ should be true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ iff

$TPInit$ is true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$

$TPNext$ is true on $s_i \rightarrow s_{i+1}$ for all i

$TPNext$ is true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ iff it is true on $s_1 \rightarrow s_2$.

$\Box\, TPNext$ is true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ iff

$TPNext$ is true on $s_i \rightarrow s_{i+1}$ for all i

The temporal formula *always* $TPNext$ is true on a behavior if and only if $TPNext$ is true on every step of the behavior.

$TPSpec$ should be true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ iff

$TPInit$ is true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$

$\boxed{TPNext \text{ is true on } s_i \rightarrow s_{i+1} \text{ for all } i}$

$TPNext$ is true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ iff it is true on $s_1 \rightarrow s_2$.

$\Box\, TPNext$ is true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ iff

$\boxed{TPNext \text{ is true on } s_i \rightarrow s_{i+1} \text{ for all } i}$

The temporal formula *always* $TPNext$ is true on a behavior if and only if $TPNext$ is true on every step of the behavior.

Which is exactly the second condition that $TPSpec$ should assert.

$TPSpec$ should be true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ iff

$TPInit$ is true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$

$\boxed{TPNext \text{ is true on } s_i \rightarrow s_{i+1} \text{ for all } i}$

$TPNext$ is true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ iff
it is true on $s_1 \rightarrow s_2$.

$\Box\, TPNext$ is true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ iff

$TPNext$ is true on $s_i \rightarrow s_{i+1}$ for all i

The temporal formula *always* $TPNext$ is true on a behavior if and only if $TPNext$ is true on every step of the behavior.

Which is exactly the second condition that $TPSpec$ should assert.

So we can restate that condition

$TPSpec$ should be true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ iff

$TPInit$ is true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$

$TPNext$ is true on $s_i \rightarrow s_{i+1}$ for all i

$TPNext$ is true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ iff it is true on $s_1 \rightarrow s_2$.

$\Box\, TPNext$ is true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ iff

$TPNext$ is true on $s_i \rightarrow s_{i+1}$ for all i

The temporal formula *always* $TPNext$ is true on a behavior if and only if $TPNext$ is true on every step of the behavior.

Which is exactly the second condition that $TPSpec$ should assert.

So we can restate that condition this way.

[slide 151]

$TPSpec$ should be true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ iff

$TPInit$ is true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$

$\boxed{\Box\, TPNext \text{ is true on } s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots}$

$TPNext$ is true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ iff
it is true on $s_1 \rightarrow s_2$.

$\Box\, TPNext$ is true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ iff

$TPNext$ is true on $s_i \rightarrow s_{i+1}$ for all i

The temporal formula *always* $TPNext$ is true on a behavior if and only if $TPNext$ is true on every step of the behavior.

Which is exactly the second condition that $TPSpec$ should assert.

So we can restate that condition this way.

$TPSpec$ should be true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ iff

$TPInit$ is true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$

$\Box TPNext$ is true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$

From this, we see that

$TPSpec$ should be true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ iff

$TPInit$ is true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$

$\Box\, TPNext$ is true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$

$TPSpec \;\triangleq$

From this, we see that $TPSpec$ should be defined to equal

$TPSpec$ should be true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ iff

$\boxed{TPInit \quad \text{is true on} \quad s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots}$

$\Box TPNext$ is true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$

$$TPSpec \triangleq TPInit$$

From this, we see that $TPSpec$ should be defined to equal

$TPInit$

$TPSpec$ should be true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ iff

$TPInit$ is true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$

$\boxed{\Box\, TPNext \text{ is true on } s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots}$

$$TPSpec \triangleq TPInit \land \Box\, TPNext$$

From this, we see that $TPSpec$ should be defined to equal

$TPInit$

conjoined with *always* $TPNext$.

$TPSpec$ should be true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ iff

$TPInit$ is true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$

$\Box TPNext$ is true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$

$$TPSpec \triangleq TPInit \land \Box TPNext$$

So this is our definition of the temporal formula $TPSpec$ that is the specification of the two-phase commit protocol.

Look how simple it is. Unfortunately, it's too simple.

If you look near the end of module $TwoPhase$, you'll find this definition.

$TPSpec$ should be true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ iff

$TPInit$ is true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$

$\Box\, TPNext$ is true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$

$$TPSpec \;\triangleq\; TPInit \;\land\; \Box\,[TPNext]_{\langle rmState,\, tmState,\, tmPrepared,\, msgs \rangle}$$

So this is our definition of the temporal formula $TPSpec$ that is the specification of the two-phase commit protocol.

Look how simple it is. Unfortunately, it's too simple.

If you look near the end of module $TwoPhase$, you'll find this definition.

Where this part

$TPSpec$ should be true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ iff

$TPInit$ is true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$

$\Box\, TPNext$ is true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$

$$TPSpec \;\triangleq\; TPInit \;\land\; \boxed{\Box\,[TPNext]_{\langle rmState,\, tmState,\, tmPrepared,\, msgs\rangle}}$$

So this is our definition of the temporal formula $TPSpec$ that is the specification of the two-phase commit protocol.

Look how simple it is. Unfortunately, it's too simple.

If you look near the end of module $TwoPhase$, you'll find this definition.

Where this part is typed like this.

$TPSpec$ should be true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ iff

$TPInit$ is true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$

$\Box\, TPNext$ is true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$

$$TPSpec \;\triangleq\; TPInit \;\wedge\; \boxed{\Box\,[TPNext]_{\langle rmState,\, tmState,\, tmPrepared,\, msgs\rangle}}$$

```
[][TPNext]_<<rmState, tmState, tmPrepared, msgs>>
```

So this is our definition of the temporal formula $TPSpec$ that is the specification of the two-phase commit protocol.

Look how simple it is. Unfortunately, it's too simple.

If you look near the end of module $TwoPhase$, you'll find this definition.

Where this part is typed like this.

$TPSpec$ should be true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ iff

$TPInit$ is true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$

$\Box TPNext$ is true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$

$$TPSpec \triangleq TPInit \land \Box [TPNext]_{\langle rmState, tmState, tmPrepared, msgs \rangle}$$

So this is our definition of the temporal formula $TPSpec$ that is the specification of the two-phase commit protocol.

Look how simple it is. Unfortunately, it's too simple.

If you look near the end of module $TwoPhase$, you'll find this definition.

Where this part is typed like this. In general, The specification with

The specification with

The specification with initial formula $Init$,

The specification with initial formula $Init$,

The specification with initial formula $Init$,
next-state formula $Next$,

The specification with initial formula $Init$, next-state formula $Next$,

The specification with initial formula $Init$,
next-state formula $Next$,
declared variables $v_1, \ldots, v_n$

The specification with initial formula $Init$, next-state formula $Next$, and declared variables v-one through v-n

The specification with initial formula $Init$,
next-state formula $Next$,
declared variables $v_1, \ldots, v_n$
is expressed by the temporal formula

$$Init \;\wedge\; \Box[Next]_{\langle v_1, \ldots, v_n \rangle}$$

The specification with initial formula $Init$, next-state formula $Next$, and declared variables v-one through v-n is expressed by this temporal formula.

The specification with initial formula $Init$,
next-state formula $Next$,
declared variables $v_1, \ldots, v_n$
is expressed by the temporal formula

$$Init \ \wedge \ \Box[Next]_{\langle v_1, \ldots, v_n \rangle}$$

```
Init /\ [][Next]_<< v_1, ... , v_n >>
```

The specification with initial formula $Init$, next-state formula $Next$, and declared variables v-one through v-n is expressed by this temporal formula.

which is typed like this.

The specification with initial formula $Init$,
next-state formula $Next$,
declared variables $v_1, \ldots, v_n$
is expressed by the temporal formula

$$Init \;\wedge\; \Box [Next]_{\langle v_1, \ldots, v_n \rangle}$$

The specification with initial formula $Init$, next-state formula $Next$, and declared variables v-one through v-n is expressed by this temporal formula.

which is typed like this.

For now, you should ignore the red part and pretend the formula is this

The specification with initial formula $Init$,
next-state formula $Next$,
declared variables $v_1, \ldots, v_n$
is expressed by the temporal formula

$$Init \ \wedge \ \Box Next$$

a temporal formula that is true on behaviors

The specification with initial formula $Init$,
next-state formula $Next$,
declared variables $v_1, \ldots, v_n$
is expressed by the temporal formula

$$\boxed{Init} \land \Box Next$$

a temporal formula that is true on behaviors
for which $Init$ is true on the initial state

The specification with initial formula $Init$,
next-state formula $Next$,
declared variables $v_1, \ldots, v_n$
is expressed by the temporal formula

$$Init \;\land\; \boxed{\Box\, Next}$$

a temporal formula that is true on behaviors
for which $Init$ is true on the initial state
and $Next$ is true on every step.

The specification with initial formula $Init$,
next-state formula $Next$,
declared variables $v_1, \ldots, v_n$
is expressed by the temporal formula

$$Init \;\land\; \Box[Next]_{\langle v_1, \ldots, v_n \rangle}$$

a temporal formula that is true on behaviors for which $Init$ is true on the initial state and $Next$ is true on every step.

To help you do that, I'll color the other stuff gray.

The specification with initial formula $Init$,
next-state formula $Next$,
declared variables $v_1, \ldots, v_n$
is expressed by the temporal formula

$$Init \;\wedge\; \Box[Next]_{\langle v_1, \ldots, v_n \rangle}$$

a temporal formula that is true on behaviors for which $Init$ is true on the initial state and $Next$ is true on every step.

To help you do that, I'll color the other stuff gray.

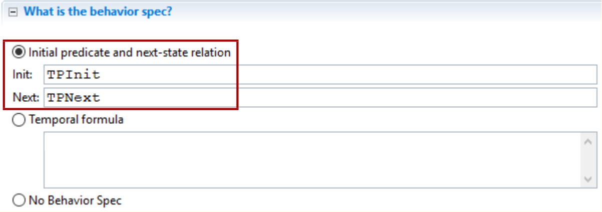
To tell TLC that the spec is:

$$TPInit \land \Box[TPNext]_{\langle rmState, tmState, tmPrepared, msgs \rangle}$$

To tell TLC that the spec for a model is this temporal formula

To tell TLC that the spec is:

$$TPInit \land \Box[TPNext]_{\langle rmState, tmState, tmPrepared, msgs \rangle}$$

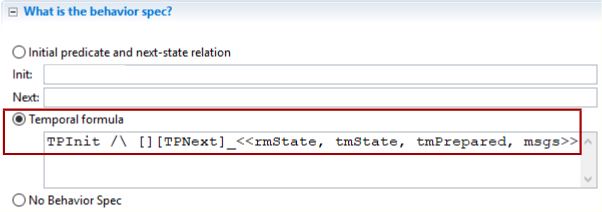


To tell TLC that the spec for a model is this temporal formula

We can give it the initial formula and next-state formula.

To tell TLC that the spec is:

$$TPInit \land \Box[TPNext]_{\langle rmState, tmState, tmPrepared, msgs \rangle}$$



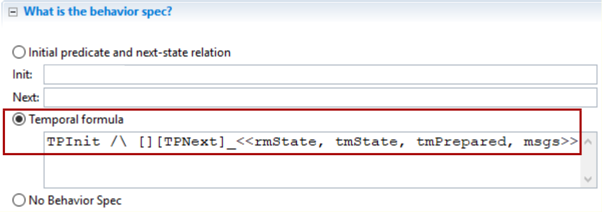
To tell TLC that the spec for a model is this temporal formula

We can give it the initial formula and next-state formula.

Or we can give it the temporal formula.

To tell TLC that the spec is:

$$TPSpec \triangleq TPInit \land \Box[TPNext]_{\langle rmState, tmState, tmPrepared, msgs \rangle}$$



To tell TLC that the spec for a model is this temporal formula

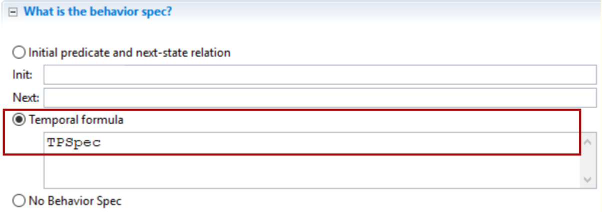
We can give it the initial formula and next-state formula.

Or we can give it the temporal formula.

If we've given this formula a name

To tell TLC that the spec is:

$$TPSpec \triangleq TPInit \land \Box[TPNext]_{\langle rmState, tmState, tmPrepared, msgs \rangle}$$



To tell TLC that the spec for a model is this temporal formula

We can give it the initial formula and next-state formula.

Or we can give it the temporal formula.

If we've given this formula a name

Then we can just give TLC that name.

Applying $\Box$ to a State Formula

Let's now see what it means to apply the *Always* operator to a state formula.

Applying $\Box$ to a State Formula

For the action $TPNext$:

$\Box\, TPNext$ is true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ iff
$TPNext$ is true on $s_i \rightarrow s_{i+1}$ for all i.

Let's now see what it means to apply the *Always* operator to a state formula.

For the action $TPNext$, *always* $TPNext$ is true on a behavior if and only if $TPNext$ is true on every step of the behavior.

Applying $\Box$ to a State Formula

For the action $TPNext$:

$\Box\, TPNext$ is true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ iff
$TPNext$ is true on $s_i \rightarrow s_{i+1}$ for all i.

The state formula $TPTypeOK$ is an action

Let's now see what it means to apply the *Always* operator to a state formula.

For the action $TPNext$, *always* $TPNext$ is true on a behavior if and only if $TPNext$ is true on every step of the behavior.

A state formula like $TPTypeOK$ is an action

Applying $\Box$ to a State Formula

For the action $TPNext$:

> $\Box\, TPNext$ is true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ iff
> $TPNext$ is true on $s_i \rightarrow s_{i+1}$ for all i .

The state formula $TPTypeOK$ is an action,
so

> $\Box\, TPTypeOK$ is true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ iff
> $TPTypeOK$ is true on $s_i \rightarrow s_{i+1}$ for all i .

So *always* $TPTypeOK$ is true on a behavior if and only if $TPTypeOK$ is true on every step of the behavior.

Applying $\Box$ to a State Formula

For the action $TPNext$:

$\Box$ $TPNext$ is true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ iff
$TPNext$ is true on $s_i \rightarrow s_{i+1}$ for all i.

The state formula $TPTypeOK$ is an action whose value on $s_i \rightarrow s_{i+1}$ depends only on s_i

$\Box$ $TPTypeOK$ is true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ iff
$TPTypeOK$ is true on $s_i \rightarrow s_{i+1}$ for all i.

So *always* $TPTypeOK$ is true on a behavior if and only if $TPTypeOK$ is true on every step of the behavior.

But a state formula is an action whose value on a step depends only on the first state of the step.

Applying $\Box$ to a State Formula

For the action $TPNext$:

$\Box\, TPNext$ is true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ iff
$TPNext$ is true on $s_i \rightarrow s_{i+1}$ for all i.

The state formula $TPTypeOK$ is an action whose value on $s_i \rightarrow s_{i+1}$ depends only on s_i, so

$\Box\, TPTypeOK$ is true on $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \cdots$ iff
$TPTypeOK$ is true on $\boxed{s_i}$ for all i.

So *always* $TPTypeOK$ is true on a behavior if and only if $TPTypeOK$ is true on every step of the behavior.

But a state formula is an action whose value on a step depends only on the first state of the step.

So *always* $TPTypeOK$ is true on a behavior if and only if $TPTypeOK$ is true on every *state* of the behavior.

$\Box\, TPTypeOK$ is true on a behavior iff
$TPTypeOK$ is true on every state of the behavior.

$\Box TPTypeOK$ is true on a behavior iff
$TPTypeOK$ is true on every state of the behavior.

You can write $\Box TPTypeOK$.

You can write simply *always* $TPTypeOK$.

$\square$ $TPTypeOK$ is true on a behavior iff
$TPTypeOK$ is true on every state of the behavior.

You can write $\square TPTypeOK$.

You don't need the $[\ ]_{\langle rmState,\, tmState,\, tmPrepared,\, msgs \rangle}$
for $\square$ $state\ formula$.

You can write simply *always* $TPTypeOK$.

You don't need the square brackets and subscript when you apply *always* to a state formula.

THEOREMS

Theorems

For a temporal formula TF

THEOREM TF

asserts that TF is true on every possible behavior.

If TF is a temporal formula, the statement THEOREM TF asserts that TF is true on every possible behavior.

For a temporal formula TF

THEOREM TF

asserts that TF is true on every possible behavior.

Not just for behaviors satisfying some spec.

If TF is a temporal formula, the statement THEOREM TF asserts that TF is true on every possible behavior.

That's every possible behavior, not just every behavior satisfying some spec.

THEOREM $TPSpec \Rightarrow \Box TPTypeOK$

This theorem

THEOREM $TPSpec \Rightarrow \Box TPTypeOK$

Asserts that for every behavior:

This theorem asserts that for every behavior

THEOREM $TPSpec \Rightarrow \Box TPTypeOK$

Asserts that for every behavior:

if $TPSpec$ is true on the behavior

This theorem asserts that for every behavior if $TPSpec$ is true on the behavior

THEOREM $TPSpec \Rightarrow \Box TPTypeOK$

Asserts that for every behavior:

if $TPSpec$ is true on the behavior

then $\Box TPTypeOK$ is true on the behavior

This theorem asserts that for every behavior if $TPSpec$ is true on the behavior then *always* $TPTypeOK$ is true on that behavior.

THEOREM $TPSpec \Rightarrow \Box TPTypeOK$

Asserts that for every behavior:

if $TPSpec$ is true on the behavior

then $\Box TPTypeOK$ is true on the behavior

This theorem asserts that for every behavior if $TPSpec$ is true on the behavior then *always* $TPTypeOK$ is true on that behavior.

$TPSpec$ true on the behavior

THEOREM $TPSpec \Rightarrow \Box TPTypeOK$

Asserts that for every behavior:

if the behavior satisfies $TPSpec$

then $\Box TPTypeOK$ is true on the behavior

This theorem asserts that for every behavior if $TPSpec$ is true on the behavior then *always* $TPTypeOK$ is true on that behavior.

$TPSpec$ true on the behavior just means that the behavior satisfies $TPSpec$.

THEOREM $TPSpec \Rightarrow \Box TPTypeOK$

Asserts that for every behavior:

if the behavior satisfies $TPSpec$

then $\Box TPTypeOK$ is true on the behavior

This theorem asserts that for every behavior if $TPSpec$ is true on the behavior then *always* $TPTypeOK$ is true on that behavior.

$TPSpec$ true on the behavior just means that the behavior satisfies $TPSpec$.

Always $TPTypeOK$ is true on the behavior

THEOREM $TPSpec \Rightarrow \Box TPTypeOK$

Asserts that for every behavior:

if the behavior satisfies $TPSpec$

then $TPTypeOK$ is true on every state of the behavior

This theorem asserts that for every behavior if $TPSpec$ is true on the behavior then *always* $TPTypeOK$ is true on that behavior.

$TPSpec$ true on the behavior just means that the behavior satisfies $TPSpec$.

Always $TPTypeOK$ is true on the behavior means that $TPTypeOK$ is true on every state of the behavior.

THEOREM $TPSpec \Rightarrow \Box TPTypeOK$

Asserts that for every behavior:

if the behavior satisfies $TPSpec$

then $TPTypeOK$ is true on every state of the behavior

So this theorem

THEOREM $TPSpec \Rightarrow \Box TPTypeOK$

Asserts that for every behavior:

if the behavior satisfies $TPSpec$

then $TPTypeOK$ is true on every state of the behavior

Asserts that $TPTypeOK$ is an invariant of $TPSpec$.

So this theorem

asserts that $TPTypeOK$ is an invariant of the specification $TPSpec$.

THEOREM $TPSpec \Rightarrow \Box TPTypeOK$

Asserts that $TPTypeOK$ is an invariant of $TPSpec$.

THEOREM $TPSpec \Rightarrow \Box TPTypeOK$

Asserts that $TPTypeOK$ is an invariant of $TPSpec$.

TLC does not automatically check theorems.

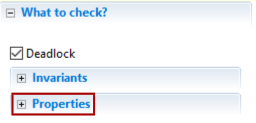
TLC does not automatically check theorems. (But you should put them in your specs to tell the reader what you expect to be true.)

THEOREM $TPSpec \Rightarrow \Box TPTypeOK$

Asserts that $TPTypeOK$ is an invariant of $TPSpec$.

TLC does not automatically check theorems.

To check this theorem, add `[]TPTypeOK` to



for a model with behavior spec `TPSpec`.

TLC does not automatically check theorems. (But you should put them in your specs to tell the reader what you expect to be true.)

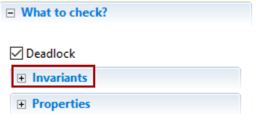
To check this theorem with TLC, add *always* $TPTypeOK$ to the *Properties* list of the *What to check* section of the *Model overview* page for a model having $TPSpec$ as its behavior specification.

THEOREM $TPSpec \Rightarrow \Box TPTypeOK$

Asserts that $TPTypeOK$ is an invariant of $TPSpec$.

TLC does not automatically check theorems.

To check this theorem, add `TPTypeOK` to



for a model with behavior spec `TPSpec`.

Or, since this is an invariance property, you can just check that $TPTypeOK$ (without the *always*) is an invariant of $TPSpec$.

We’re now ready to explain in Part Two what it means for the two-phase commit protocol to implement the specification of transaction commit, and how to use TLC to check that it does.

End of Lecture 8, Part 1

IMPLEMENTATION

PRELIMINARIES