TLA$^+$ Video Course – Lecture 8, Part 2

Leslie Lamport

IMPLEMENTATION

HOW IT WORKS

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 2
Implementation: How it Works

When you were a child, it must have been weird to learn that the earth was round. If you were raised in Asia, it probably seemed ridiculous that Americans were hanging upside down by their feet and didn't fall off into the sky. But you got used to it.

You probably found the idea of specifying systems with math strange enough. You will now learn things about TLA+ that even sophisticated computer scientists find weird. But they're pretty simple things, and you'll get used to them. Eventually, you'll realize that without them, TLA+ would be as weird as a flat earth.

THE THEOREM

Transaction Commit

The specification in module $TCommit$ has:

- declared variable $rmState$
- initial formula $TCInit$
- next-state formula $TCNext$

Remember the transaction commit spec.

It was in module $TCommit$ and had a single declared variable $rmState$, an initial formula $TCInit$, and a next-state formula $TCNext$.

Transaction Commit

The specification in module $TCommit$ has:

– declared variable $rmState$

– initial formula $TCInit$

– next-state formula $TCNext$

Its specification $TCSpec$ is therefore:

$$TCSpec \triangleq TCInit \land \Box[TCNext]_{\langle rmState \rangle}$$

Remember the transaction commit spec.

It was in module $TCommit$ and had a single declared variable $rmState$, an initial formula $TCInit$, and a next-state formula $TCNext$.

Its specification is therefore the temporal formula $TCSpec$ defined like this.

Transaction Commit

The specification in module $TCommit$ has:

– declared variable $rmState$

– initial formula $TCInit$

– next-state formula $TCNext$

Its specification $TCSpec$ is therefore:

$$TCSpec \triangleq TCInit \land \Box[TCNext]_{\langle rmState \rangle}$$

with a single variable can omit the $\langle\ \rangle$

Because it has only a single variable, we can omit the angle brackets

Transaction Commit

The specification in module $TCommit$ has:

– declared variable $rmState$

– initial formula $TCInit$

– next-state formula $TCNext$

Its specification $TCSpec$ is therefore:

$$TCSpec \triangleq TCInit \land \Box[TCNext]_{rmState}$$

with a single variable can omit the $\langle\ \rangle$

Because it has only a single variable, we can omit the angle brackets and write the subscript simply like this.

Transaction Commit

The specification in module $TCommit$ has:

– declared variable $rmState$

– initial formula $TCInit$

– next-state formula $TCNext$

Its specification $TCSpec$ is therefore:

$$TCSpec \triangleq TCInit \land \Box[TCNext]_{rmState}$$

with a single variable can omit the $\langle\ \rangle$

Because it has only a single variable, we can omit the angle brackets and write the subscript simply like this.

Module $TwoPhase$ contains:

INSTANCE $TCommit$

Module $TwoPhase$ contains this INSTANCE statement

Module $TwoPhase$ contains:

INSTANCE $TCommit$

Imports the definition of $TCSpec$.

Module $TwoPhase$ contains this INSTANCE statement

which imports the definition of $TCSpec$ as well as all other definitions from module $TCommit$.

Module $TwoPhase$ contains:

INSTANCE $TCommit$

THEOREM $TPSpec \Rightarrow TCSpec$

Module $TwoPhase$ contains this INSTANCE statement

which imports the definition of $TCSpec$ as well as all other definitions from module $TCommit$.

Module $TwoPhase$ also contains this theorem

Module $TwoPhase$ contains:

INSTANCE $TCommit$

THEOREM $TPSpec \Rightarrow TCSpec$

Asserts that for every behavior:
if it satisfies $TPSpec$
then it satisfies $TCSpec$.

Module $TwoPhase$ contains this INSTANCE statement

which imports the definition of $TCSpec$ as well as all other definitions from module $TCommit$.

Module $TwoPhase$ also contains this theorem

which asserts that for every behavior: if the behavior satisfies $TPSpec$ then it satisfies $TCSpec$.

Module $TwoPhase$ contains:

INSTANCE $TCommit$

THEOREM $TPSpec \Rightarrow TCSpec$

Every behavior satisfying $TPSpec$ satisfies $TCspec$.

In other words, every behavior that satisfies $TPSpec$ satisfies $TCSpec$.

Module $TwoPhase$ contains:

INSTANCE $TCommit$

THEOREM $TPSpec \Rightarrow TCSpec$

Every behavior satisfying $TPSpec$ satisfies $TCspec$.

$TPSpec$ implements $TCSpec$.

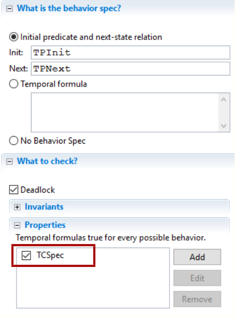
In other words, every behavior that satisfies $TPSpec$ satisfies $TCSpec$.

This is what it means for $TPSpec$ to implement $TCSpec$.

THEOREM $TPSpec \Rightarrow TCSpec$

THEOREM $TPSpec \Rightarrow TCSpec$

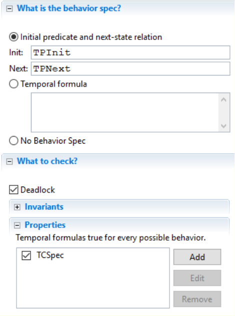
Let TLC check this theorem by adding $TCSpec$ as a property to check in a model you constructed for module $TwoPhase$.



Let TLC check this theorem by adding $TCSpec$ as a property to check in a model you constructed for module $TwoPhase$.

THEOREM $TPSpec \Rightarrow TCSpec$

Let TLC check this theorem by adding $TCSpec$ as a property to check in a model you constructed for module $TwoPhase$.



TLC should find no error.

Let TLC check this theorem by adding $TCSpec$ as a property to check in a model you constructed for module $TwoPhase$.

It should find no error.

THEOREM $TPSpec \Rightarrow TCSpec$

THEOREM $TPSpec \Rightarrow TCSpec$

How can this theorem make sense?

How can this theorem make sense?

THEOREM $TPSpec \Rightarrow TCSpec$

An assertion about behaviors whose states assign values to $rmState$, $tmState$, $tmPrepared$, and $msgs$.

How can this theorem make sense?

$TPSpec$, which is defined in module $TwoPhase$, is an assertion about behaviors whose states assign values to the four variables $rmState$, $tmState$, $tmPrepared$, and m-s-g-s.

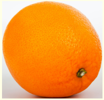
THEOREM $TPSpec \Rightarrow TCSpec$

An assertion about behaviors whose states assign values to $rmState$, $tmState$, $tmPrepared$, and $msgs$.

An assertion about behaviors whose states assign values to $rmState$.

$TCSpec$, which is defined in module $TCommit$, is an assertion about behaviors whose states assign a value to the single variable $rmState$.

THEOREM $TPSpec \Rightarrow TCSpec$

 $\Rightarrow$ 

$TCSpec$, which is defined in module $TCommit$, is an assertion about behaviors whose states assign a value to the single variable $rmState$.

Isn't this formula relating apples and oranges?

A state is an assignment of values to variables.

I've said that a state is an assignment of values to variables.

A state is an assignment of values to variables.

What variables?

I've said that a state is an assignment of values to variables.

But what variables.

A state is an assignment of values to variables.

What variables?

The variables declared in a module.

I've said that a state is an assignment of values to variables.

But what variables.

Everything I've said so far has led you to believe that a state assigns values to the variables declared in the current module.

A state is an assignment of values to variables.

What variables?

~~The variables declared in a module.~~

I've said that a state is an assignment of values to variables.

But what variables.

Everything I've said so far has led you to believe that a state assigns values to the variables declared in the current module.

But I've been fooling you because I wanted to delay hitting you with this bit of weirdness:

[slide 26]

A state is an assignment of values to variables.

What variables?

~~The variables declared in a module.~~

All possible variables.

upside down

A state actually assigns values to all possible variables.

A state is an assignment of values to variables.

What variables?

~~The variables declared in a module.~~

All possible variables. (There are infinitely many.)

upside down

A state actually assigns values to all possible variables.

That's right, to each of the infinite number of variables that you could (in principle) declare in a module.

Weird, huh?

Consider this state:

Consider this state.

Consider this state:

$$Mozart \ = \ \langle -37, \{14\} \rangle$$

Consider this state.

Consider this state:

$$\begin{aligned} Mozart &= \langle -37, \{14\} \rangle \\ rmState &= [r \in \{\text{“r1”}, \text{“r2”}, \text{“r3”}\} \mapsto \text{“working”}] \end{aligned}$$

Consider this state. I’m just showing

Consider this state:

$$\begin{aligned} Mozart &= \langle -37, \{14\} \rangle \\ rmState &= [r \in \{\text{“r1”}, \text{“r2”}, \text{“r3”}\} \mapsto \text{“working”}] \\ tmState &= \text{“ouch”} \end{aligned}$$

Consider this state. I'm just showing the values it

Consider this state:

$$Mozart = \langle -37, \{14\} \rangle$$
$$rmState = [r \in \{\text{"r1"}, \text{"r2"}, \text{"r3"}\} \mapsto \text{"working"}]$$
$$tmState = \text{"ouch"}$$
$$numberOfCustomersInTimbuktuStarbucks = 42$$

Consider this state. I'm just showing the values it assigns to a few

Consider this state:

$$
\begin{aligned}
Mozart &= \langle -37, \{14\} \rangle \\
rmState &= [r \in \{\text{“r1”}, \text{“r2”}, \text{“r3”}\} \mapsto \text{“working”}] \\
tmState &= \text{“ouch”} \\
numberOfCustomersInTimbuktuStarbucks &= 42 \\
msgs &= \{314\}
\end{aligned}
$$

Consider this state. I'm just showing the values it assigns to a few of the

Consider this state:

$$
\begin{aligned}
Mozart &= \langle -37, \{14\} \rangle \\
rmState &= [r \in \{\text{“r1”}, \text{“r2”}, \text{“r3”}\} \mapsto \text{“working”}] \\
tmState &= \text{“ouch”} \\
numberOfCustomersInTimbuktuStarbucks &= 42 \\
msgs &= \{314\} \\
&\vdots
\end{aligned}
$$

Consider this state. I’m just showing the values it assigns to a few of the infinite number of variables.

Consider this state:

$$Mozart = \langle -37, \{14\} \rangle$$

$$rmState = [r \in \{\text{"r1"}, \text{"r2"}, \text{"r3"}\} \mapsto \text{"working"}]$$

$$tmState = \text{"ouch"}$$

$$numberOfCustomersInTimbuktuStarbucks = 42$$

$$msgs = \{314\}$$

$$\vdots$$

$TCInit$ is true on it iff RM equals $\{\text{"r1"}, \text{"r2"}, \text{"r3"}\}$.

$TCInit$ is true on this state if and only if RM equals the set of three strings $r1$, $r2$, and $r3$.

Consider this state:

$$Mozart = \langle -37, \{14\} \rangle$$

$$rmState = [r \in \{\text{"r1"}, \text{"r2"}, \text{"r3"}\} \mapsto \text{"working"}]$$

$$tmState = \text{"ouch"}$$

$$numberOfCustomersInTimbuktuStarbucks = 42$$

$$msgs = \{314\}$$

$$\vdots$$

$TCInit$ is true on it iff RM equals $\{\text{"r1"}, \text{"r2"}, \text{"r3"}\}$.

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

$TCInit$ is true on this state if and only if RM equals the set of three strings $r1$, $r2$, and $r3$.

That's because this is the definition of $TCInit$.

Consider this state:

$$Mozart = \langle -37, \{14\} \rangle$$

$$rmState = [r \in \{\text{"r1"}, \text{"r2"}, \text{"r3"}\} \mapsto \text{"working"}]$$

$$tmState = \text{"ouch"}$$

$$numberOfCustomersInTimbuktuStarbucks = 42$$

$$msgs = \{314\}$$

$$\vdots$$

$TCInit$ is true on it iff RM equals {“r1”, “r2”, “r3”}.

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

$TCInit$ is true on this state if and only if RM equals the set of three strings r1, r2, and r3.

That’s because this is the definition of $TCInit$.

Consider this state:

$$Mozart = \langle -37, \{14\} \rangle$$

$$rmState = \boxed{[r \in \{\text{“r1”}, \text{“r2”}, \text{“r3”}\} \mapsto \text{“working”}]}$$

$$tmState = \text{“ouch”}$$

$$numberOfCustomersInTimbuktuStarbucks = 42$$

$$msgs = \{314\}$$

$$\vdots$$

$TCInit$ is true on it iff RM equals {“r1”, “r2”, “r3”}.

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

$TCInit$ is true on this state if and only if RM equals the set of three strings $r1$, $r2$, and $r3$.

That’s because this is the definition of $TCInit$.

And this is the value of the variable $rmState$ in the state.

$TCSpec$ contains only variable $rmState$.

The only variable formula $TCSpec$ contains is $rmState$.

$TCSpec$ contains only variable $rmState$.

So, we can tell if a behavior satisfies $TCSpec$ by looking at the value of $rmState$ in each state.

The only variable formula $TCSpec$ contains is $rmState$.

So we can tell whether or not a behavior satisfies $TCSpec$ by looking only at the value assigned to $rmState$ by each of the behavior's states.

$TCSpec$ contains only variable $rmState$.

So, we can tell if a behavior satisfies $TCSpec$ by looking at the value of $rmState$ in each state.

All other variables can have any values.

The only variable formula $TCSpec$ contains is $rmState$.

So we can tell whether or not a behavior satisfies $TCSpec$ by looking only at the value assigned to $rmState$ by each of the behavior's states.

All the other variables can have any values in any of its states.

$TCSpec$ contains only variable $rmState$.

So, we can tell if a behavior satisfies $TCSpec$ by looking at the value of $rmState$ in each state.

All other variables can have any values.

$TCSpec$ allows $tmPrepared$ to equal

For example, in a behavior satisfying formula $TCSpec$, variable $tmPrepared$ could equal

$TCSpec$ contains only variable $rmState$.

So, we can tell if a behavior satisfies $TCSpec$ by looking at the value of $rmState$ in each state.

All other variables can have any values.

$TCSpec$ allows $tmPrepared$ to equal

in the 1^{st} state: $\{$“$orange$”, “$delicious$”, “$macintosh$”$\}$

For example, in a behavior satisfying formula $TCSpec$, variable $tmPrepared$ could equal this value in the first state

$TCSpec$ contains only variable $rmState$.

So, we can tell if a behavior satisfies $TCSpec$ by looking at the value of $rmState$ in each state.

All other variables can have any values.

$TCSpec$ allows $tmPrepared$ to equal

in the 1st state: $\{\text{"}orange\text{"}, \text{"}delicious\text{"}, \text{"}macintosh\text{"}\}$

in the 2nd state: $2^{48976553}$

For example, in a behavior satisfying formula $TCSpec$,
variable $tmPrepared$ could equal
this value in the first state
this value in the second state

$TCSpec$ contains only variable $rmState$.

So, we can tell if a behavior satisfies $TCSpec$ by looking at the value of $rmState$ in each state.

All other variables can have any values.

$TCSpec$ allows $tmPrepared$ to equal

in the 1st state: $\{$“$orange$”, “$delicious$”, “$macintosh$”$\}$

in the 2nd state: $2^{48976553}$

in the 3rd state: $[a \mapsto 22, b \mapsto \{13, \{13\}, \{\{13\}\}\}]$

For example, in a behavior satisfying formula $TCSpec$,
variable $tmPrepared$ could equal
this value in the first state
this value in the second state
this value in the third state

$TCSpec$ contains only variable $rmState$.

So, we can tell if a behavior satisfies $TCSpec$ by looking at the value of $rmState$ in each state.

All other variables can have any values.

$TCSpec$ allows $tmPrepared$ to equal

in the 1st state: $\{\text{“}orange\text{”}, \text{“}delicious\text{”}, \text{“}macintosh\text{”}\}$

in the 2nd state: $2^{48976553}$

in the 3rd state: $[a \mapsto 22, b \mapsto \{13, \{13\}, \{\{13\}\}\}]$

⋮

For example, in a behavior satisfying formula $TCSpec$, variable $tmPrepared$ could equal this value in the first state this value in the second state this value in the third state and so on.

This seems weird to most people because they think of specifications as programs.

This seems weird to most people because they think of specifications as programs.

This seems weird to most people because they think of specifications as programs.

They're not programs; they're mathematical formulas.

This seems weird to most people because they think of specifications as programs.

Specifications are not programs; they're mathematical formulas.

This seems weird to most people because they think of specifications as programs.

They're not programs; they're mathematical formulas.

In math, when you write:

$$\begin{aligned} x + y &= 7 \\ 2 * x - y &= 2 \end{aligned}$$

it doesn't mean that there's no variable z or w.

In math, when you write equations like this about the variables x and y, it doesn't mean that there's no variable z or w.

This seems weird to most people because they think of specifications as programs.

They're not programs; they're mathematical formulas.

In math, when you write:

$$\begin{aligned} x + y &= 7 \\ 2 * x - y &= 2 \end{aligned}$$

it doesn't mean that there's no variable z or w.

The equations say nothing about other variables.

In math, when you write equations like this about the variables x and y, it doesn't mean that there's no variable z or w.

The equations just say nothing about those other variables.

It's useful to think about specifications as follows.

A specification does not describe the correct behavior of a system.

A specification does not describe the correct behavior of a system.

A specification does not describe the correct behavior of a system.

It describes a universe in which the system and its environment are behaving correctly.

A specification does not describe the correct behavior of a system.

Rather, it describes a history of the universe in which the system and its environment are behaving correctly.

A specification does not describe the correct behavior of a system.

It describes a universe in which the system and its environment are behaving correctly.

A specification does not describe the correct behavior of a system.

Rather, it describes a history of the universe in which the system and its environment are behaving correctly.

The spec describes not only the system, but other parts of the universe that the system depends on.

A specification does not describe the correct behavior of a system.

It describes a universe in which the system and its environment are behaving correctly.

For example, $msgs$ might describe an external communication protocol used by two-phase commit.

For example, the variable m-s-g-s might describe an external communication mechanism such as TCP used by the two-phase commit protocol.

A specification does not describe the correct behavior of a system.

It describes a universe in which the system and its environment are behaving correctly.

For example, $msgs$ might describe an external communication protocol used by two-phase commit.

The spec says nothing about irrelevant parts of the universe.

For example, the variable m-s-g-s might describe an external communication mechanism such as TCP used by the two-phase commit protocol.

The spec says nothing about parts of the universe that are not relevant to its abstraction of the system.

STUTTERING

THEOREM $TPSpec \Rightarrow TCSpec$

This theorem makes sense because

Now we see that this theorem makes sense because

THEOREM $\boxed{TPSpec} \Rightarrow \boxed{TCSpec}$

This theorem makes sense because both formulas are assertions about the same kind of behavior.

Now we see that this theorem makes sense because formulas $TPSpec$ and $TCSpec$ are both assertions about the same kind of behavior – one whose states assign values to all variables.

THEOREM $TPSpec \Rightarrow TCSpec$

This theorem makes sense because both formulas are assertions about the same kind of behavior.

It asserts that every behavior satisfying $TPSpec$ satisfies $TCSpec$.

Now we see that this theorem makes sense because formulas $TPSpec$ and $TCSpec$ are both assertions about the same kind of behavior – one whose states assign values to all variables.

The theorem asserts that every behavior satisfying $TPSpec$ also satisfies $TCSpec$.

THEOREM $TPSpec \Rightarrow TCSpec$

This theorem makes sense because both formulas are assertions about the same kind of behavior.

It asserts that every behavior satisfying $TPSpec$ satisfies $TCSpec$.

But how can it be true?

Now we see that this theorem makes sense because formulas $TPSpec$ and $TCSpec$ are both assertions about the same kind of behavior – one whose states assign values to all variables.

The theorem asserts that every behavior satisfying $TPSpec$ also satisfies $TCSpec$.

But how can this statement possibly be true?

THEOREM $TPSpec \Rightarrow TCSpec$

Formula $TPSpec$

THEOREM $\boxed{TPSpec} \Rightarrow TCSpec$

$$TPSpec \triangleq TPInit \land \Box[\,TPNext\,]_{\langle \ldots \rangle}$$

Formula $TPSpec$ is defined like this

THEOREM $TPSpec \Rightarrow TCSpec$

$$TPSpec \triangleq TPInit \land \Box[TPNext]_{\langle \ldots \rangle}$$

$TPNext$ allows $TMAbort$ steps.

Formula $TPSpec$ is defined like this where $TPNext$ allows $TMAbort$ steps

THEOREM $TPSpec \Rightarrow TCSpec$

$$TPSpec \triangleq TPInit \land \Box[TPNext]_{\langle \ldots \rangle}$$

$TPNext$ allows $TMAbort$ steps.

$$\begin{aligned} TMAbort \triangleq \\ &\land tmState = \text{"init"} \\ &\land tmState' = \text{"done"} \\ &\land msgs' = msgs \cup \{[type \mapsto \text{"Abort"}]\} \\ &\land \text{UNCHANGED } \langle rmState, tmPrepared \rangle \end{aligned}$$

Formula $TPSpec$ is defined like this where $TPNext$ allows $TMAbort$ steps and $TMAbort$ is defined like this

THEOREM $TPSpec \Rightarrow TCSpec$

$$TPSpec \triangleq TPInit \land \Box[TPNext]_{\langle \ldots \rangle}$$

$TPNext$ allows $TMAbort$ steps.

$$\begin{aligned} TMAbort \triangleq\ & \land tmState = \text{"init"} \\ & \land tmState' = \text{"done"} \\ & \land msgs' = msgs \cup \{[type \mapsto \text{"Abort"}]\} \\ & \land \text{UNCHANGED } \langle rmState, tmPrepared \rangle \end{aligned}$$

Formula $TPSpec$ is defined like this where $TPNext$ allows $TMAbort$ steps and $TMAbort$ is defined like this so its UNCHANGED conjunct allows only steps

THEOREM $TPSpec \Rightarrow TCSpec$

$$TPSpec \triangleq TPInit \land \Box[TPNext]_{\langle \ldots \rangle}$$

$TPNext$ allows $TMAbort$ steps, which leave $rmState$ unchanged.

$$
\begin{aligned}
TMAbort \triangleq\ & \\
& \land\ tmState = \text{“init”} \\
& \land\ tmState' = \text{“done”} \\
& \land\ msgs' = msgs \cup \{[type \mapsto \text{“Abort”}]\} \\
& \land\ \text{UNCHANGED}\ \langle rmState, tmPrepared \rangle
\end{aligned}
$$

Formula $TPSpec$ is defined like this where $TPNext$ allows $TMAbort$ steps and $TMAbort$ is defined like this so its UNCHANGED conjunct allows only steps that leave $rmState$ unchanged.

THEOREM $TPSpec \Rightarrow \boxed{TCSpec}$

$TPSpec \triangleq TPInit \land \Box[TPNext]_{\langle \ldots \rangle}$

$TPNext$ allows $TMAbort$ steps, which leave $rmState$ unchanged.

$TCSpec \triangleq TCInit \land \Box[TCNext]_{rmState}$

$TCSpec$ is defined like this

THEOREM $TPSpec \Rightarrow TCSpec$

$$TPSpec \triangleq TPInit \wedge \Box[TPNext]_{\langle \ldots \rangle}$$

$TPNext$ allows $TMAbort$ steps, which leave $rmState$ unchanged.

$$TCSpec \triangleq TCInit \wedge \Box[\boxed{TCNext}]_{rmState}$$

All $TCNext$ steps change $rmState$.

$TCSpec$ is defined like this where all $TCNext$ steps change the value of $rmState$.

THEOREM $TPSpec \Rightarrow TCSpec$

$$TPSpec \triangleq TPInit \land \Box[\, TPNext \,]_{\langle \ldots \rangle}$$

$TPNext$ allows $TMAbort$ steps, which leave $rmState$ unchanged.

$$TCSpec \triangleq TCInit \land \Box[\, TCNext \,]_{rmState}$$

All $TCNext$ steps change $rmState$.

$TCSpec$ is defined like this where all $TCNext$ steps change the value of $rmState$.

A $TMAbort$ step therefore can't be a $TCNext$ step.

THEOREM $TPSpec \Rightarrow TCSpec$

$TPSpec \triangleq TPInit \wedge \Box[TPNext]_{\langle \ldots \rangle}$

$TPNext$ allows $TMAbort$ steps, which leave $rmState$ unchanged.

$TCSpec \triangleq TCInit \wedge \Box[TCNext]_{rmState}$

All $TCNext$ steps change $rmState$.

How can a behavior satisfying $TPSpec$ also satisfy $TCSpec$ if it has a $TMAbort$ step?

$TCSpec$ is defined like this where all $TCNext$ steps change the value of $rmState$.

A $TMAbort$ step therefore can't be a $TCNext$ step.

So how can a behavior satisfying $TPSpec$ also satisfy $TCSpec$ if it has a $TMAbort$ step?

THEOREM $TPSpec \Rightarrow TCSpec$

$$TPSpec \triangleq TPInit \land \Box[\, TPNext \,]_{\langle \ldots \rangle}$$

$TPNext$ allows $TMAbort$ steps, which leave $rmState$ unchanged.

$$TCSpec \triangleq TCInit \land \Box[\, TCNext \,]_{rmState}$$

All $TCNext$ steps change $rmState$.

How can a behavior satisfying $TPSpec$ also satisfy $TCSpec$ if it has a $TMAbort$ step?

How can the theorem be true?

$TCSpec$ is defined like this where all $TCNext$ steps change the value of $rmState$.

A $TMAbort$ step therefore can't be a $TCNext$ step.

So how can a behavior satisfying $TPSpec$ also satisfy $TCSpec$ if it has a $TMAbort$ step? And how can this theorem be true?

$$TCSpec \triangleq TCInit \land \Box[TCNext]_{rmState}$$

The answer to this question lies

$$TCSpec \triangleq TCInit \land \Box[TCNext]_{rmState}$$

The answer to this question lies in the meaning of this part of the formula that we've been ignoring.

$TCSpec \triangleq TCInit \land \Box [TCNext]_{rmState}$

$\Box [TCNext]_{rmState}$ is true on a behavior iff

The answer to this question lies in the meaning of this part of the formula that we've been ignoring.

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

$TCSpec \triangleq TCInit \wedge \Box [TCNext]_{rmState}$

$\Box [TCNext]_{rmState}$ is true on a behavior iff

$[TCNext]_{rmState}$ is true on every step of the behavior.

The answer to this question lies in the meaning of this part of the formula that we've been ignoring.

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

$$TCSpec \triangleq TCInit \land \Box [\, TCNext \,]_{rmState}$$

$\Box [\, TCNext \,]_{rmState}$ is true on a behavior iff

$[\, TCNext \,]_{rmState}$ is true on every step of the behavior.

$$[\, TCNext \,]_{rmState} \triangleq TCNext \lor (\text{UNCHANGED } rmState)$$

This formula is an abbreviation for the action $TCNext$ disjunction UNCHANGED $rmState$. .

$TCSpec \triangleq TCInit \land \Box [TCNext]_{rmState}$

$\Box [TCNext]_{rmState}$ is true on a behavior iff

$TCNext \lor (\text{UNCHANGED } rmState)$ is true on every step.

$$[TCNext]_{rmState} \triangleq TCNext \lor (\text{UNCHANGED } rmState)$$

This formula is an abbreviation for the action $TCNext$ disjunction UNCHANGED $rmState$. .

So the always formula asserts that $TCNext$ or UNCHANGED $rmState$ is true on every step.

$TCSpec \triangleq TCInit \land \Box [TCNext]_{rmState}$

$\Box [TCNext]_{rmState}$ is true on a behavior iff every step satisfies $TCNext$ or leaves $rmState$ unchanged.

$$[TCNext]_{rmState} \triangleq TCNext \lor (\text{UNCHANGED } rmState)$$

This formula is an abbreviation for the action $TCNext$ disjunction UNCHANGED $rmState$. .

So the always formula asserts that $TCNext$ or UNCHANGED $rmState$ is true on every step.

which is the same as the assertion that every step satisfies $TCNext$ or leaves $rmState$ unchanged.

$TCSpec \triangleq TCInit \land \Box [TCNext]_{rmState}$

$\Box [TCNext]_{rmState}$ is true on a behavior iff every step satisfies $TCNext$ or leaves $rmState$ unchanged.

If steps leaving $rmState$ unchanged were not allowed by $TCSpec$.

$TCSpec \triangleq TCInit \land \Box [TCNext]_{rmState}$

$\Box [TCNext]_{rmState}$ is true on a behavior iff every step satisfies $TCNext$ or leaves $rmState$ unchanged.

THEOREM $TPSpec \Rightarrow TCSpec$

would not be true otherwise.

If steps leaving $rmState$ unchanged were not allowed by $TCSpec$, then the theorem would not be true.

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

Similarly, for the two-phase commit spec

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

True on a behavior iff every step satisfies $TPNext$ or leaves $rmState$, $tmState$, $tmPrepared$, and $msgs$ unchanged.

Similarly, for the two-phase commit spec

This *always* formula is true on a behavior if and only if every step of the behavior satisfies the next-state formula $TPNext$ or else leaves all the specification variables unchanged.

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

True on a behavior iff every step satisfies $TPNext$ or leaves $rmState$, $tmState$, $tmPrepared$, and $msgs$ unchanged.

stuttering steps

Similarly, for the two-phase commit spec
This *always* formula is true on a behavior if and only if every step of the behavior satisfies the next-state formula $TPNext$ or else leaves all the specification variables unchanged.

Steps that leave all the spec's variables unchanged are called *stuttering steps*.

Stuttering Steps

upside down

Most people find stuttering steps weird.

Stuttering Steps

All TLA$^+$ specs allow stuttering steps.

upside down

Most people find stuttering steps weird.

Every TLA+ spec allows them.

Stuttering Steps

All TLA^+ specs allow stuttering steps.

If they didn't, $TPSpec$ would allow the value of

$$numberOfCustomersInTimbuktuStarbucks$$

to change only when the protocol took a step.

upside down

Most people find stuttering steps weird.

Every TLA+ spec allows them.

If they didn't, the two-phase commit spec would allow the value of every variable in the universe to change only when the two-phase commit protocol took a step.

And that would be *really* weird.

Stuttering Steps

All TLA+ specs allow stuttering steps.

If they didn't, $TPSpec$ would allow the value of

$numberOfCustomersInTimbuktuStarbucks$

to change only when the protocol took a step.

The most important reason:

THEOREM $TPSpec \Rightarrow TCSpec$

But the most important reason to allow stuttering steps is embodied in this theorem:

Stuttering Steps

All TLA⁺ specs allow stuttering steps.

If they didn't, $TPSpec$ would allow the value of

$numberOfCustomersInTimbuktuStarbucks$

to change only when the protocol took a step.

The most important reason:

THEOREM $TPSpec \Rightarrow TCSpec$

Implementation is implication.

But the most important reason to allow stuttering steps is embodied in this theorem:
Implementation becomes simple logical implication.

THEOREM $TPSpec \Rightarrow TCSpec$

Mathematical simplicity is not an end in itself.

Mathematical simplicity is not an end in itself.

THEOREM $TPSpec \Rightarrow TCSpec$

Mathematical simplicity is not an end in itself.

It's a sign that we're doing things right.

Mathematical simplicity is not an end in itself.

But it *is* a sign that we're doing things right.

TERMINATION AND STOPPING

Specification $SimpleProgram$ of Lectures 1 and 2

Remember our first example: Specification $SimpleProgram$ of Lectures 1 and 2.

Specification $SimpleProgram$ of Lectures 1 and 2

– declared variables pc and i

– initial formula $Init$

– next-state formula $Next$

Remember our first example: Specification $SimpleProgram$ of Lectures 1 and 2.

It had two variables pc and i, initial formula $Init$, and next-state formula $Next$.

$$Init \land \Box [Next]_{\langle pc, i \rangle}$$

Remember our first example: Specification $SimpleProgram$ of Lectures 1 and 2.

It had two variables pc and i, initial formula $Init$, and next-state formula $Next$.

Here's how we now write its specification as a temporal formula.

$$Init \land \Box [Next]_{\langle pc, i \rangle}$$

$$\begin{bmatrix} pc : \text{“}start\text{”} \\ i \;\; : 0 \end{bmatrix} \rightarrow \begin{bmatrix} pc : \text{“}middle\text{”} \\ i \;\; : 43 \end{bmatrix} \rightarrow \begin{bmatrix} pc : \text{“}done\text{”} \\ i \;\; : 44 \end{bmatrix}$$

Here’s how we originally would have written a behavior satisfying this spec.

$$Init \wedge \Box [Next]_{\langle pc, i \rangle}$$

$$\begin{bmatrix} pc : \text{“}start\text{”} \\ i \;\; : \mathsf{0} \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} pc : \text{“}middle\text{”} \\ i \;\; : \mathsf{43} \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} pc : \text{“}done\text{”} \\ i \;\; : \mathsf{44} \\ \vdots \end{bmatrix}$$

In this lecture, we saw that the states of the behavior actually assign variables to infinitely many other variables.

$$Init \land \Box [Next]_{\langle pc, i \rangle}$$

$$\begin{bmatrix} pc : \text{“}start\text{”} \\ i \ \ : \ 0 \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} pc : \text{“}middle\text{”} \\ i \ \ : \ 43 \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} pc : \text{“}middle\text{”} \\ i \ \ : \ 43 \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} pc : \text{“}done\text{”} \\ i \ \ : \ 44 \\ \vdots \end{bmatrix}$$

In this lecture, we saw that the states of the behavior actually assign variables to infinitely many other variables.

Then we saw that the spec allows stuttering steps.

$$Init \land \Box [Next]_{\langle pc, i \rangle}$$

$$\begin{bmatrix} pc : \text{“}start\text{”} \\ i \ \ : 0 \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} pc : \text{“}middle\text{”} \\ i \ \ : 43 \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} pc : \text{“}middle\text{”} \\ i \ \ : 43 \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} pc : \text{“}done\text{”} \\ i \ \ : 44 \\ \vdots \end{bmatrix}$$

In this lecture, we saw that the states of the behavior actually assign variables to infinitely many other variables.

Then we saw that the spec allows stuttering steps.

$$Init \land \Box[Next]_{\langle pc,\, i \rangle}$$

$$\begin{bmatrix} pc : \text{"}start\text{"} \\ i \ \ : 0 \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} pc : \text{"}middle\text{"} \\ i \ \ : 43 \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} pc : \text{"}middle\text{"} \\ i \ \ : 43 \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} pc : \text{"}done\text{"} \\ i \ \ : 44 \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} pc : \text{"}done\text{"} \\ i \ \ : 44 \\ \vdots \end{bmatrix}$$

In this lecture, we saw that the states of the behavior actually assign variables to infinitely many other variables.

Then we saw that the spec allows stuttering steps.

It also allows stuttering steps at the end.

$$Init \land \Box [Next]_{\langle pc, i \rangle}$$

$$\begin{bmatrix} pc : \text{“}start\text{”} \\ i \;\; : 0 \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} pc : \text{“}middle\text{”} \\ i \;\; : 43 \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} pc : \text{“}middle\text{”} \\ i \;\; : 43 \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} pc : \text{“}done\text{”} \\ i \;\; : 44 \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} pc : \text{“}done\text{”} \\ i \;\; : 44 \\ \vdots \end{bmatrix}$$

$$\rightarrow \begin{bmatrix} pc : \text{“}done\text{”} \\ i \;\; : 44 \\ \vdots \end{bmatrix}$$

In this lecture, we saw that the states of the behavior actually assign variables to infinitely many other variables.

Then we saw that the spec allows stuttering steps.

It also allows stuttering steps at the end.

$$Init \wedge \Box[Next]_{\langle pc,\, i \rangle}$$

$$\begin{bmatrix} pc : \text{“}start\text{”} \\ i \;\; : \mathsf{0} \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} pc : \text{“}middle\text{”} \\ i \;\; : \mathsf{43} \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} pc : \text{“}middle\text{”} \\ i \;\; : \mathsf{43} \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} pc : \text{“}done\text{”} \\ i \;\; : \mathsf{44} \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} pc : \text{“}done\text{”} \\ i \;\; : \mathsf{44} \\ \vdots \end{bmatrix}$$

$$\rightarrow \begin{bmatrix} pc : \text{“}done\text{”} \\ i \;\; : \mathsf{44} \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} pc : \text{“}done\text{”} \\ i \;\; : \mathsf{44} \\ \vdots \end{bmatrix}$$

In this lecture, we saw that the states of the behavior actually assign variables to infinitely many other variables.

Then we saw that the spec allows stuttering steps.

It also allows stuttering steps at the end.

$$Init \land \Box [Next]_{\langle pc,\, i \rangle}$$

$$\begin{bmatrix} pc : \text{"}start\text{"} \\ i \ \ : 0 \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} pc : \text{"}middle\text{"} \\ i \ \ : 43 \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} pc : \text{"}middle\text{"} \\ i \ \ : 43 \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} pc : \text{"}done\text{"} \\ i \ \ : 44 \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} pc : \text{"}done\text{"} \\ i \ \ : 44 \\ \vdots \end{bmatrix}$$

$$\rightarrow \begin{bmatrix} pc : \text{"}done\text{"} \\ i \ \ : 44 \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} pc : \text{"}done\text{"} \\ i \ \ : 44 \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} pc : \text{"}done\text{"} \\ i \ \ : 44 \\ \vdots \end{bmatrix}$$

In this lecture, we saw that the states of the behavior actually assign variables to infinitely many other variables.

Then we saw that the spec allows stuttering steps.

It also allows stuttering steps at the end.

$$Init \land \Box [Next]_{\langle pc, i \rangle}$$

$$\begin{bmatrix} pc : \text{“}start\text{”} \\ i \;\; : 0 \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} pc : \text{“}middle\text{”} \\ i \;\; : 43 \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} pc : \text{“}middle\text{”} \\ i \;\; : 43 \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} pc : \text{“}done\text{”} \\ i \;\; : 44 \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} pc : \text{“}done\text{”} \\ i \;\; : 44 \\ \vdots \end{bmatrix}$$

$$\rightarrow \begin{bmatrix} pc : \text{“}done\text{”} \\ i \;\; : 44 \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} pc : \text{“}done\text{”} \\ i \;\; : 44 \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} pc : \text{“}done\text{”} \\ i \;\; : 44 \\ \vdots \end{bmatrix} \rightarrow \ldots$$

In this lecture, we saw that the states of the behavior actually assign variables to infinitely many other variables.

Then we saw that the spec allows stuttering steps.

It also allows stuttering steps at the end.

In fact it allows an infinite number of stuttering steps at the end.

We represent a terminating execution by a behavior ending in an infinite sequence of stuttering steps.

We represent a terminating (or deadlocked) execution by a behavior ending in an infinite sequence of stuttering steps.

We represent a terminating execution by a behavior ending in an infinite sequence of stuttering steps.

The universe keeps going even if the system terminates.

We represent a terminating (or deadlocked) execution by a behavior ending in an infinite sequence of stuttering steps.

This is natural, because a behavior represents a history of the universe, and the universe keeps going even if the system we're specifying terminates.

We represent a terminating execution by a behavior ending in an infinite sequence of stuttering steps.

The universe keeps going even if the system terminates.

All behaviors are infinite sequences of states.

We represent a terminating (or deadlocked) execution by a behavior ending in an infinite sequence of stuttering steps.

This is natural, because a behavior represents a history of the universe, and the universe keeps going even if the system we're specifying terminates.

This means that all behaviors are infinite sequences of states, so we don't have to consider finite behaviors.

$$Init \wedge \Box [Next]_{\langle pc, i \rangle}$$

This specification is also satisfied by a behavior that

$$Init \wedge \Box [Next]_{\langle pc, i \rangle}$$

$$\begin{bmatrix} pc : \text{"}start\text{"} \\ i \;\; : 0 \\ \vdots \end{bmatrix}$$

This specification is also satisfied by a behavior that starts in a state satisfying $Init$,

$$Init \land \Box [Next]_{\langle pc, i \rangle}$$

$$\begin{bmatrix} pc : \text{"}start\text{"} \\ i \;\; : \mathsf{0} \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} pc : \text{"}middle\text{"} \\ i \;\; : \mathsf{43} \\ \vdots \end{bmatrix}$$

This specification is also satisfied by a behavior that starts in a state satisfying $Init$,

takes a step satisfying action $Next$,

$$Init \wedge \Box [Next]_{\langle pc, i \rangle}$$

$$\begin{bmatrix} pc : \text{“}start\text{”} \\ i \;\; : 0 \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} pc : \text{“}middle\text{”} \\ i \;\; : 43 \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} pc : \text{“}middle\text{”} \\ i \;\; : 43 \\ \vdots \end{bmatrix}$$

This specification is also satisfied by a behavior that starts in a state satisfying $Init$,

takes a step satisfying action $Next$,

takes a stuttering step,

$$Init \wedge \Box [Next]_{\langle pc, i \rangle}$$

$$\begin{bmatrix} pc : \text{“}start\text{”} \\ i \;\; : 0 \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} pc : \text{“}middle\text{”} \\ i \;\; : 43 \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} pc : \text{“}middle\text{”} \\ i \;\; : 43 \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} pc : \text{“}middle\text{”} \\ i \;\; : 43 \\ \vdots \end{bmatrix}$$

This specification is also satisfied by a behavior that starts in a state satisfying $Init$,
takes a step satisfying action $Next$,
takes a stuttering step,
takes another stuttering step,

$$Init \wedge \Box [Next]_{\langle pc, i \rangle}$$

$$\begin{bmatrix} pc : \text{“}start\text{”} \\ i \;\; : 0 \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} pc : \text{“}middle\text{”} \\ i \;\; : 43 \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} pc : \text{“}middle\text{”} \\ i \;\; : 43 \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} pc : \text{“}middle\text{”} \\ i \;\; : 43 \\ \vdots \end{bmatrix} \rightarrow \cdots$$

This specification is also satisfied by a behavior that starts in a state satisfying $Init$,
takes a step satisfying action $Next$,
takes a stuttering step,
takes another stuttering step,
and keeps on taking stuttering steps forever.

$$Init \land \Box [Next]_{\langle pc,\, i \rangle}$$

$$\begin{bmatrix} pc : \text{“}start\text{”} \\ i \;\;: 0 \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} pc : \text{“}middle\text{”} \\ i \;\;: 43 \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} pc : \text{“}middle\text{”} \\ i \;\;: 43 \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} pc : \text{“}middle\text{”} \\ i \;\;: 43 \\ \vdots \end{bmatrix} \rightarrow \cdots$$

These stuttering steps are allowed by the spec.

All these stuttering steps are allowed by the spec.

$$Init \land \Box [Next]_{\langle pc, i \rangle}$$

$$\begin{bmatrix} pc : \text{“}start\text{”} \\ i \ : 0 \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} pc : \text{“}middle\text{”} \\ i \ : 43 \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} pc : \text{“}middle\text{”} \\ i \ : 43 \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} pc : \text{“}middle\text{”} \\ i \ : 43 \\ \vdots \end{bmatrix} \rightarrow \ldots$$

This behavior represents an execution in which the program stops before reaching a terminating state.

All these stuttering steps are allowed by the spec.

This behavior represents an execution in which the program stops before reaching a terminating state.

$$Init \land \Box [Next]_{\langle pc,\, i \rangle}$$

Our specs allow a system to stop at any time.

upside down

All the specs we have written so far allow the system being specified to stop at any time by taking infinitely many stuttering steps.

$$Init \land \Box [Next]_{\langle pc,\, i \rangle}$$

Our specs allow a system to stop at any time.

They specify what the system may do.

upside down

All the specs we have written so far allow the system being specified to stop at any time by taking infinitely many stuttering steps.

Our specs specify what the system *may* do.

$$Init \land \Box [Next]_{\langle pc,\, i \rangle}$$

Our specs allow a system to stop at any time.

They specify what the system may do.

They don't specify what it must do.

upside down

All the specs we have written so far allow the system being specified to stop at any time by taking infinitely many stuttering steps.

Our specs specify what the system *may* do.
They don't specify what it *must* do; they allow it to do nothing.

$$Init \land \Box [Next]_{\langle pc,\, i \rangle}$$

Our specs allow a system to stop at any time.

They specify what the system may do.

They don't specify what it must do.

Exactly what *may* and *must* mean will be explained later.

Exactly what *may* and *must* mean will be explained later.

$$Init \wedge \Box [Next]_{\langle pc,\, i \rangle}$$

Our specs allow a system to stop at any time.

They specify what the system may do.
They don't specify what it must do.

Exactly what *may* and *must* mean will be explained later.

They are very different requirements and should be specified separately.

Exactly what *may* and *must* mean will be explained later.

But they are very different kinds of requirements and they should be specified separately.

$$Init \wedge \Box [Next]_{\langle pc,\, i \rangle}$$

We add *must* requirements

We add *must* requirements

$$Init \wedge \Box[Next]_{\langle pc,\, i \rangle} \;\wedge\; L$$

We add *must* requirements by conjoining a temporal formula to the specification.

We add *must* requirements by conjoining a temporal formula to the specification.

$$Init \wedge \Box [Next]_{\langle pc,\, i \rangle} \;\wedge\; L$$

We add *must* requirements by conjoining a temporal formula to the specification.

That is the subject of the next lecture.

We add *must* requirements by conjoining a temporal formula to the specification.

How that's done is the main subject of the next lecture.

$$Init \land \Box [Next]_{\langle pc,\, i \rangle} \land \boxed{L}$$

This is a tiny part of a spec.

The *must* formula is just a tiny part of a spec.

$$\boxed{Init \wedge \Box[Next]_{\langle pc,\, i\rangle}} \wedge \ L$$

This is a tiny part of a spec.

This is the larger and more important part.

The *must* formula is just a tiny part of a spec.

The *may* formula is much larger and usually more important.

$$Init \land \Box [Next]_{\langle pc, i \rangle}$$

This is a tiny part of a spec.

This is the larger and more important part.

You can write useful specs that say what the system *may* do.

The *must* formula is just a tiny part of a spec.

The *may* formula is much larger and usually more important.

With what you've learned so far, you can write specs that are quite useful even though they specify only what they system *may* do.

You are now ready to be fruitful and specify. At least to specify what a system *may* do. In the next lecture, you'll learn how to specify what it *must* do.

End of Lecture 8, Part 2

IMPLEMENTATION

HOW IT WORKS