IMPLEMENTATION WITH REFINEMENT

REFINEMENT MAPPINGS

This video should be viewed in conjunction with a Web page.
To find that page, search the Web for *TLA+ Video Course*.
Having finished the preliminaries, we head to our main goal: understanding what it means in general for one specification to implement another, and how we can check that it does. We will take a rather long path, and it may not always be clear where it’s leading. But just follow it step by step. The destination is worth the effort.
AB2 IMPLEMENTS AB
The $AB2$ protocol doesn’t just implement the $ABSpec$ spec.

The $AB2$ protocol doesn’t just implement the high level spec of module $ABSpec$. 

[slide 4]
The $AB2$ protocol doesn’t just implement the $ABSpec$ spec.

It implements the $AB$ protocol, where a $LoseMsg$ step of $AB$ is implemented by a $CorruptMsg$ step of $AB2$.

The $AB2$ protocol doesn’t just implement the high level spec of module $ABSpec$.

It actually implements the $AB$ protocol, where an $AB$ protocol step that loses a message is implemented by the $AB2$ protocol step that corrupts the message.
The $AB2$ protocol doesn’t just implement the $ABSpec$ spec.

It implements the $AB$ protocol, where a $LoseMsg$ step of $AB$ is implemented by a $CorruptMsg$ step of $AB2$.

Programmers find this confusing.

The $AB2$ protocol doesn’t just implement the high level spec of module $ABSpec$.

It actually implements the $AB$ protocol, where an $AB$ protocol step that loses a message is implemented by the $AB2$ protocol step that corrupts the message.

Most programmers will find this confusing.

[slide 6]
The $AB2$ protocol doesn’t just implement the $ABSpec$ spec. It implements the $AB$ protocol, where a $LoseMsg$ step of $AB$ is implemented by a $CorruptMsg$ step of $AB2$.

Programmers find this confusing. They don’t think losing a message is a step of the $AB$ protocol, but rather a step of the environment.

They don’t think of losing a message as a step of the $AB$ protocol, but rather as a step taken by the environment in which the protocol is executed.
Our specifications say nothing about who performs what steps.
Our specifications say nothing about who performs what steps.

We think that in the $AB$ spec:

Sender $A$ performs $ASnd$ and $ARcv$ steps.

Receiver $B$ performs $BSnd$ and $BRcv$ steps.

And the communication infrastructure performs $LoseMsg$ steps.

[slide 9]
Our specifications say nothing about who performs what steps.

We think that in the $AB$ spec:

- Sender $A$ performs $ASnd$ and $ARcv$ steps.
- Receiver $B$ performs $BSnd$ and $BRcv$ steps.

[slide 10]
Our specifications say nothing about who performs what steps.

We think that in the $AB$ spec:

- **Sender $A$** performs $ASnd$ and $ARcv$ steps.
- **Receiver $B$** performs $BSnd$ and $BRcv$ steps.
- The communication infrastructure performs $LoseMsg$ steps.

[slide 11]
Our specifications say nothing about who performs what steps.

We think that in the $AB$ spec:

- Sender $A$ performs $ASnd$ and $ARcv$ steps.
- Receiver $B$ performs $BSnd$ and $BRcv$ steps.
- The communication infrastructure performs $LoseMsg$ steps.

That’s just an interpretation we put on the spec.

But that’s just an interpretation that we put on the spec, suggested by the way we write the next-state action as the disjunction of subactions.
Our specifications say nothing about who performs what steps.

We think that in the $AB$ spec:
- Sender $A$ performs $ASnd$ and $ARcv$ steps.
- Receiver $B$ performs $BSnd$ and $BRcv$ steps.
- The communication infrastructure performs $LoseMsg$ steps.

That's just an interpretation we put on the spec.

It would be easy to make the spec suggest a different interpretation.

But that's just an interpretation that we put on the spec, suggested by the way we write the next-state action as the disjunction of subactions.

It would be easy to make the spec suggest a different interpretation—for example by decomposing the next-state action to suggest that $A$ and $B$ both send messages and cause the messages to be lost.
The $AB2$ protocol implements the $AB$ protocol, where $CorruptMsg$ steps implement $LoseMsg$ steps.

The $AB2$ protocol implements the $AB$ protocol, where $CorruptMsg$ steps implement $LoseMsg$ steps.
The $AB2$ protocol implements the $AB$ protocol, where $CorruptMsg$ steps implement $LoseMsg$ steps.

The goal: convince ourselves that this is true.

The $AB2$ protocol implements the $AB$ protocol, where $CorruptMsg$ steps implement $LoseMsg$ steps.

Our goal now is to convince ourselves that this is true.
The \(AB2\) protocol implements the \(AB\) protocol, where \(CorruptMsg\) steps implement \(LoseMsg\) steps.

The goal: convince ourselves that this is true.

This requires answering two questions:

The \(AB2\) protocol implements the \(AB\) protocol, where \(CorruptMsg\) steps implement \(LoseMsg\) steps.

Our goal goal now is to convince ourselves that this is true.

Reaching it requires answering two questions:
The $AB2$ protocol implements the $AB$ protocol, where $CorruptMsg$ steps implement $LoseMsg$ steps.

The goal: convince ourselves that this is true.

This requires answering two questions:

1. What does it mean?

The first is: What does it mean?
The $AB2$ protocol implements the $AB$ protocol, where $CorruptMsg$ steps implement $LoseMsg$ steps.

The goal: convince ourselves that this is true.

This requires answering two questions:

1. What does it mean?
2. How do we check it?

The first is: What does it mean?

And the second is: How do we check it?
The $AB^2$ protocol implements the $AB$ protocol, where $CorruptMsg$ steps implement $LoseMsg$ steps.

What does it mean?

First, exactly what does this mean?
The $AB^2$ protocol implements the $AB$ protocol, where $CorruptMsg$ steps implement $LoseMsg$ steps.

What does it mean?

For every behavior of $AB^2$ we can obtain a behavior of $AB$ by changing the state as follows:

First, exactly what does this mean?

It means that for every behavior of the $AB^2$ protocol we can obtain a behavior of the $AB$ protocol by changing the state as shown in the following example:

[slide 20]
The $AB2$ protocol implements the $AB$ protocol, where $CorruptMsg$ steps implement $LoseMsg$ steps.

What does it mean?

For every behavior of $AB2$ we can obtain a behavior of $AB$ by changing the state as follows:

<table>
<thead>
<tr>
<th>State of $AB2$</th>
<th>$AVar$</th>
<th>$BVar$</th>
<th>$AtoB2$</th>
<th>$BtoA2$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>⟨“Tom”, 1⟩</td>
<td>⟨“Ann”, 0⟩</td>
<td>⟨Bad, ⟨“Tom”, 1⟩⟩</td>
<td>⟨0, Bad, 0, Bad⟩</td>
</tr>
</tbody>
</table>

For this state in a behavior of $AB2$
The $AB2$ protocol implements the $AB$ protocol, where $\text{CorruptMsg}$ steps implement $\text{LoseMsg}$ steps.

What does it mean?

For every behavior of $AB2$ we can obtain a behavior of $AB$ by changing the state as follows:

<table>
<thead>
<tr>
<th>State of $AB2$</th>
<th>State of $AB$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$AVar = \langle \text{&quot;Tom&quot;}, 1 \rangle$</td>
<td>$AVar = \langle \text{&quot;Tom&quot;}, 1 \rangle$</td>
</tr>
<tr>
<td>$BVar = \langle \text{&quot;Ann&quot;}, 0 \rangle$</td>
<td>$BVar = \langle \text{&quot;Ann&quot;}, 0 \rangle$</td>
</tr>
<tr>
<td>$AtoB2 = \langle \text{Bad}, \langle \text{&quot;Tom&quot;}, 1 \rangle \rangle$</td>
<td>$AtoB = \langle \text{Bad}, \langle \text{&quot;Tom&quot;}, 1 \rangle \rangle$</td>
</tr>
<tr>
<td>$BtoA2 = \langle 0, \text{Bad}, 0, \text{Bad} \rangle$</td>
<td>$BtoA = \langle 0, \text{Bad}, 0, \text{Bad} \rangle$</td>
</tr>
</tbody>
</table>

For this state in a behavior of $AB2$ here’s how we get the corresponding state in a behavior of $AB$. 

[slide 22]
The $AB2$ protocol implements the $AB$ protocol, where $CorruptMsg$ steps implement $LoseMsg$ steps.

What does it mean?

For every behavior of $AB2$ we can obtain a behavior of $AB$ by changing the state as follows:

<table>
<thead>
<tr>
<th>State of $AB2$</th>
<th>State of $AB$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$AVar = \langle &quot;Tom&quot;, 1 \rangle$</td>
<td>$AVar = \langle &quot;Tom&quot;, 1 \rangle$</td>
</tr>
<tr>
<td>$BVar = \langle &quot;Ann&quot;, 0 \rangle$</td>
<td>$BVar = \langle &quot;Ann&quot;, 0 \rangle$</td>
</tr>
<tr>
<td>$AtoB2 = \langle Bad, \langle &quot;Tom&quot;, 1 \rangle \rangle$</td>
<td>$AtoB = \langle \rangle$</td>
</tr>
<tr>
<td>$BtoA2 = \langle 0, Bad, 0, Bad \rangle$</td>
<td>$BtoA = \langle \rangle$</td>
</tr>
</tbody>
</table>

For this state in a behavior of $AB2$ here’s how we get the corresponding state in a behavior of $AB$.

The values of $AVar$ and $BVar$ are the same.
The $AB2$ protocol implements the $AB$ protocol, where $CorruptMsg$ steps implement $LoseMsg$ steps.

What does it mean?

For every behavior of $AB2$ we can obtain a behavior of $AB$ by changing the state as follows:

<table>
<thead>
<tr>
<th>State of $AB2$</th>
<th>State of $AB$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$AVar = \langle &quot;Tom&quot;, 1 \rangle$</td>
<td>$AVar = \langle &quot;Tom&quot;, 1 \rangle$</td>
</tr>
<tr>
<td>$BVar = \langle &quot;Ann&quot;, 0 \rangle$</td>
<td>$BVar = \langle &quot;Ann&quot;, 0 \rangle$</td>
</tr>
<tr>
<td>$AtoB2 = \langle Bad, \langle &quot;Tom&quot;, 1 \rangle \rangle$</td>
<td>$AtoB = \langle \rangle$</td>
</tr>
<tr>
<td>$BtoA2 = \langle 0, Bad, 0, Bad \rangle$</td>
<td>$BtoA = \langle \rangle$</td>
</tr>
</tbody>
</table>

For this state in a behavior of $AB2$ here’s how we get the corresponding state in a behavior of $AB$.

The values of $AVar$ and $BVar$ are the same.

We obtain the sequence of messages $AtoB$ from the sequence of messages $AtoB2$.
The AB2 protocol implements the AB protocol, where CorruptMsg steps implement LoseMsg steps.

What does it mean?

For every behavior of AB2 we can obtain a behavior of AB by changing the state as follows:

<table>
<thead>
<tr>
<th>State of AB2</th>
<th>State of AB</th>
</tr>
</thead>
<tbody>
<tr>
<td>( AVar = \langle &quot;Tom&quot;, 1 \rangle )</td>
<td>( AVar = \langle &quot;Tom&quot;, 1 \rangle )</td>
</tr>
<tr>
<td>( BVar = \langle &quot;Ann&quot;, 0 \rangle )</td>
<td>( BVar = \langle &quot;Ann&quot;, 0 \rangle )</td>
</tr>
<tr>
<td>( AtoB2 = \langle Bad, \langle &quot;Tom&quot;, 1 \rangle \rangle )</td>
<td>( AtoB = \langle \langle &quot;Tom&quot;, 1 \rangle \rangle )</td>
</tr>
<tr>
<td>( BtoA2 = \langle 0, Bad, 0, Bad \rangle )</td>
<td>( BtoA = )</td>
</tr>
</tbody>
</table>

For this state in a behavior of AB2 here’s how we get the corresponding state in a behavior of AB.

The values of \( AVar \) and \( BVar \) are the same.

We obtain the sequence of messages \( AtoB \) from the sequence of messages \( AtoB2 \) by deleting the \( Bad \) messages.
The $AB2$ protocol implements the $AB$ protocol, where $CorruptMsg$ steps implement $LoseMsg$ steps.

What does it mean?

For every behavior of $AB2$ we can obtain a behavior of $AB$ by changing the state as follows:

<table>
<thead>
<tr>
<th>State of $AB2$</th>
<th>State of $AB$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$AVar = \langle &quot;Tom&quot;, 1 \rangle$</td>
<td>$AVar = \langle &quot;Tom&quot;, 1 \rangle$</td>
</tr>
<tr>
<td>$BVar = \langle &quot;Ann&quot;, 0 \rangle$</td>
<td>$BVar = \langle &quot;Ann&quot;, 0 \rangle$</td>
</tr>
<tr>
<td>$AtoB2 = \langle Bad, \langle &quot;Tom&quot;, 1 \rangle \rangle$</td>
<td>$AtoB = \langle \langle &quot;Tom&quot;, 1 \rangle \rangle$</td>
</tr>
<tr>
<td>$BtoA2 = \langle 0, Bad, 0, Bad \rangle$</td>
<td>$BtoA = \langle \rangle$</td>
</tr>
</tbody>
</table>

And we do the same thing to obtain the sequence of messages $BtoA$ from the sequence of messages $BtoA2$. 

[slide 26]
The $AB2$ protocol implements the $AB$ protocol, where $CorruptMsg$ steps implement $LoseMsg$ steps.

What does it mean?

For every behavior of $AB2$ we can obtain a behavior of $AB$ by changing the state as follows:

<table>
<thead>
<tr>
<th>State of $AB2$</th>
<th>State of $AB$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$AVar$</td>
<td>$AVar$</td>
</tr>
<tr>
<td>$BVar$</td>
<td>$BVar$</td>
</tr>
<tr>
<td>$AtoB2$</td>
<td>$AtoB$</td>
</tr>
<tr>
<td>$BtoA2$</td>
<td>$BtoA$</td>
</tr>
</tbody>
</table>

And we do the same thing to obtain the sequence of messages $BtoA$ from the sequence of messages $BtoA2$. 

[slide 27]
The $AB^2$ protocol implements the $AB$ protocol, where $\text{CorruptMsg}$ steps implement $\text{LoseMsg}$ steps.

What does it mean?

For every behavior of $AB^2$ we can obtain a behavior of $AB$ by changing the state as follows:

<table>
<thead>
<tr>
<th>State of $AB^2$</th>
<th>State of $AB$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$AVar = \langle &quot;Tom&quot;, 1 \rangle$</td>
<td>$AVar = \langle &quot;Tom&quot;, 1 \rangle$</td>
</tr>
<tr>
<td>$BVar = \langle &quot;Ann&quot;, 0 \rangle$</td>
<td>$BVar = \langle &quot;Ann&quot;, 0 \rangle$</td>
</tr>
<tr>
<td>$Atob^2 = \langle \text{Bad}, \langle &quot;Tom&quot;, 1 \rangle \rangle$</td>
<td>$Atob = \langle \langle &quot;Tom&quot;, 1 \rangle \rangle$</td>
</tr>
<tr>
<td>$BtoA^2 = \langle 0, \text{Bad}, 0, \text{Bad} \rangle$</td>
<td>$BtoA = \langle 0, 0 \rangle$</td>
</tr>
</tbody>
</table>

And we do the same thing to obtain the sequence of messages $BtoA$ from the sequence of messages $BtoA^2$. 

[slide 28]
"$AB2$ implements $AB$" means that this transformation of states of the $AB2$ protocol to states of the $AB$ protocol.
State of $AB2$

$AVar = \langle "Tom", 1 \rangle$

$BVar = \langle "Ann", 0 \rangle$

$AtoB2 = \langle Bad, \langle "Tom", 1 \rangle \rangle$

$BtoA2 = \langle 0, Bad, 0, Bad \rangle$

State of $AB$

$AVar = \langle "Tom", 1 \rangle$

$BVar = \langle "Ann", 0 \rangle$

$AtoB = \langle \langle "Tom", 1 \rangle \rangle$

$BtoA = \langle 0, 0 \rangle$

Behavior of $AB2$  $\rightarrow$  Behavior of $AB$

"$AB2$ implements $AB$" means that this transformation of states of the $AB2$ protocol to states of the $AB$ protocol transforms a behavior of the $AB2$ protocol to a behavior of the $AB$ protocol.
To show this implementation, we first transform states of the AB2 protocol to produce behaviors satisfying a new specification SpecH.
To show this implementation, we first transform states of the $AB2$ protocol to produce behaviors satisfying a new specification $SpecH$. We obtain a state of $SpecH$ by
To show this implementation, we first transform states of the $AB2$ protocol to produce behaviors satisfying a new specification $SpecH$. We obtain a state of $SpecH$ by starting with a state of $AB2$.
To show this implementation, we first transform states of the \textit{AB2} protocol to produce behaviors satisfying a new specification \textit{SpecH}. We obtain a state of \textit{SpecH} by starting with a state of \textit{AB2} and then adding the values of the variables \textit{AtoB} and \textit{BtoA} from the state of \textit{AB}.

[slide 34]
To show this implementation, we first transform states of the \( AB2 \) protocol to produce behaviors satisfying a new specification \( SpecH \). We obtain a state of \( SpecH \) by starting with a state of \( AB2 \) and then adding the values of the variables \( AtoB \) and \( BtoA \) from the state of \( AB \).
The $AB_2$ protocol implements the $AB$ protocol if and only if every behavior allowed by formula $SpecH$ is a behavior of the $AB$ protocol.
The $AB2$ protocol implements the $AB$ protocol if and only if every behavior allowed by formula $SpecH$ is a behavior of the $AB$ protocol.

The $AB2$ protocol implements the $AB$ protocol if and only if every behavior allowed by formula $SpecH$ is a behavior of the $AB$ protocol.
The **AB2 protocol** implements the **AB** protocol if and only if every behavior allowed by formula \( SpecH \) is a behavior of the **AB** protocol.

This condition is expressed by the theorem that formula \( SpecH \) implies formula \( Spec \) of module **AB**.

[slide 38]
THEOREM \( SpecH \Rightarrow \) formula \( Spec \) of module \( AB \)

This answers our first question: What does it mean for \( AB2 \) to implement \( AB \)?
THEOREM \( SpecH \Rightarrow \) formula \( Spec \) of module \( AB \)

This answers the first question:

What does it mean for \( AB2 \) to implement \( AB \)?

This answers our first question: What does it mean for \( AB2 \) to implement \( AB \)?
THEOREM $SpecH \Rightarrow$ formula $Spec$ of module $AB$

This answers the first question:
What does it mean for $AB2$ to implement $AB$?

We now answer the second question:
How do we check it?

This answers our first question: What does it mean for $AB2$ to implement $AB$?

We now answer the second question: How do we check it?
THEOREM \[ SpecH \Rightarrow \text{formula} \ Spec \text{ of module} \ AB \]

This answers the first question:
What does it mean for \( AB2 \) to implement \( AB \) ?

We now answer the second question:
How do we check it?

To do this, we first write \( SpecH \) in TLA\(^+\).

This answers our first question: What does it mean for \( AB2 \) to implement \( AB \) ?

We now answer the second question: How do we check it?

To do this, we first actually write the formula \( SpecH \) in TLA\(^+\).
SPECIFYING SpecH
A behavior should satisfy $SpecH$ iff:

1. The values of $A\text{Var}$, $B\text{Var}$, $A\text{toB}$, $B\text{toA}$ satisfy the $AB_2$ spec.
2. In every state:
   - $A\text{toB} = A\text{toB}_2$ without corrupted messages.
   - $B\text{toA} = B\text{toA}_2$ without corrupted messages.

$SpecH$ should equal $\land (\text{FormulaSpec of module } AB_2) \land \land A\text{toB} = A\text{toB}_2$ without corrupted messages. 
$\land B\text{toA} = B\text{toA}_2$ without corrupted messages.

A behavior should satisfy $SpecH$ if and only if the following conditions hold.
A behavior should satisfy $SpecH$ iff:

- The values of $AVar$, $BVar$, $AtoB2$, $BtoA2$ satisfy the $AB2$ spec.

A behavior should satisfy $SpecH$ if and only if the following conditions hold. First, the values of the four variables of the $AB2$ spec should satisfy that spec.
A behavior should satisfy $SpecH$ iff:
- The values of $AVar$, $BVar$, $AtoB2$, $BtoA2$ satisfy the $AB2$ spec.
- In every state:

$SpecH$ should equal $\land$ Formula $Spec$ of module $AB2$ $\land$ $\land$ $AtoB$ without corrupted messages.
$\land$ $BtoA$ without corrupted messages.

A behavior should satisfy $SpecH$ if and only if the following conditions hold. First, the values of the four variables of the $AB2$ spec should satisfy that spec. Second, in every state of the behavior,
A behavior should satisfy $SpecH$ iff:

- The values of $AVar$, $BVar$, $AtoB2$, $BtoA2$ satisfy the $AB2$ spec.
- In every state:
  - $AtoB = AtoB2$ without corrupted messages.

A behavior should satisfy $SpecH$ if and only if the following conditions hold.

First, the values of the four variables of the $AB2$ spec should satisfy that spec.

Second, in every state of the behavior, $AtoB$ should equal the sequence obtained from $AtoB2$ by removing corrupted messages.
A behavior should satisfy $SpecH$ iff:

- The values of $AVar$, $BVar$, $AtoB2$, $BtoA2$ satisfy the $AB2$ spec.
- In every state:
  - $AtoB = AtoB2$ without corrupted messages.
  - $BtoA = BtoA2$ without corrupted messages.

And $BtoA$ should equal the sequence obtained from $BtoA2$ by removing corrupted messages.
A behavior should satisfy $SpecH$ iff:

– The values of $AVar$, $BVar$, $AtoB2$, $BtoA2$ satisfy the $AB2$ spec.
– In every state:
  – $AtoB = AtoB2$ without corrupted messages.
  – $BtoA = BtoA2$ without corrupted messages.

$SpecH$ should equal

\[ \land \]
\[ \land \]

And $BtoA$ should equal the sequence obtained from $BtoA2$ by removing corrupted messages.

So, $SpecH$ should be the conjunction of two formulas.
A behavior should satisfy $SpecH$ iff:

- The values of $AVar$, $BVar$, $AtoB2$, $BtoA2$ satisfy the $AB2$ spec.
- In every state:
  - $AtoB = AtoB2$ without corrupted messages.
  - $BtoA = BtoA2$ without corrupted messages.

$SpecH$ should equal

\[
\land \\
\land
\]

And $BtoA$ should equal the sequence obtained from $BtoA2$ by removing corrupted messages.

So, $SpecH$ should be the conjunction of two formulas.

The first formula, which expresses this condition,
A behavior should satisfy $SpecH$ iff:

- The values of $AVar$, $BVar$, $AtoB2$, $BtoA2$ satisfy the $AB2$ spec.
- In every state:
  - $AtoB = AtoB2$ without corrupted messages.
  - $BtoA = BtoA2$ without corrupted messages.

$SpecH$ should equal

\[ \land \text{ Formula } Spec \text{ of module } AB2 \]

And $BtoA$ should equal the sequence obtained from $BtoA2$ by removing corrupted messages.

So, $SpecH$ should be the conjunction of two formulas.

The first formula, which expresses this condition, is just formula $Spec$ of module $AB2$. 

[slide 51]
A behavior should satisfy $SpecH$ iff:

- The values of $AVar$, $BVar$, $AtoB2$, $BtoA2$ satisfy the $AB2$ spec.
- In every state:
  - $AtoB = AtoB2$ without corrupted messages.
  - $BtoA = BtoA2$ without corrupted messages.

$SpecH$ should equal

\[ \wedge \text{Formula } Spec \text{ of module } AB2 \]

And $BtoA$ should equal the sequence obtained from $BtoA2$ by removing corrupted messages.

So, $SpecH$ should be the conjunction of two formulas.

The first formula, which expresses this condition, is just formula $Spec$ of module $AB2$.

The second formula asserts that something is true in every state,
A behavior should satisfy $SpecH$ iff:

- The values of $AVar$, $BVar$, $AtoB2$, $BtoA2$ satisfy the $AB2$ spec.
- In every state:
  - $AtoB = AtoB2$ without corrupted messages.
  - $BtoA = BtoA2$ without corrupted messages.

$SpecH$ should equal

\[ \wedge \text{Formula } Spec \text{ of module } AB2 \]
\[ \wedge \square \]

which is expressed by the temporal operator $always$. 
A behavior should satisfy $SpecH$ iff:

- The values of $AVar$, $BVar$, $AtoB2$, $BtoA2$ satisfy the $AB2$ spec.
- In every state:
  - $AtoB = AtoB2$ without corrupted messages.
  - $BtoA = BtoA2$ without corrupted messages.

$SpecH$ should equal

\[ \^ \text{Formula } Spec \text{ of module } AB2 \^ \square \]

which is expressed by the temporal operator $always$.

The condition satisfied by every state
A behavior should satisfy $SpecH$ iff:

– The values of $AVar$, $BVar$, $AtoB2$, $BtoA2$ satisfy the $AB2$ spec.
– In every state:
  – $AtoB = AtoB2$ without corrupted messages.
  – $BtoA = BtoA2$ without corrupted messages.

$SpecH$ should equal

$\wedge$ Formula $Spec$ of module $AB2$

$\wedge \Box \wedge AtoB = AtoB2$ without corrupted messages.

$\wedge BtoA = BtoA2$ without corrupted messages.

which is expressed by the temporal operator $always$.

The condition satisfied by every state

is the conjunction of these two conditions.
A behavior should satisfy \( SpecH \) iff:

- The values of \( AVar \), \( BVar \), \( AtoB2 \), \( BtoA2 \) satisfy the \( AB2 \) spec.
- In every state:
  - \( AtoB = AtoB2 \) without corrupted messages.
  - \( BtoA = BtoA2 \) without corrupted messages.

\( SpecH \) should equal

\[
\land \text{Formula } Spec \text{ of module } AB2
\land \Box \land AtoB = AtoB2 \text{ without corrupted messages.}
\land BtoA = BtoA2 \text{ without corrupted messages.}
\]

which is expressed by the temporal operator \( \text{always} \).

The condition satisfied by every state is the conjunction of these two conditions.

So here’s what \( SpecH \) should equal.

[slide 56]
So the definition of $SpecH$ should look like this.

$$SpecH \triangleq \wedge \text{ Formula } Spec \text{ of module } AB2$$

$$\wedge \Box \wedge Atob = Atob2 \text{ without corrupted messages.}$$

$$\wedge BtoA = BtoA2 \text{ without corrupted messages.}$$
\[ \text{Spec}H \triangleq \land \text{Formula Spec of module AB2} \]
\[ \land \Box \land \text{AtoB} = \text{AtoB}^2 \text{ without corrupted messages.} \]
\[ \land \text{BtoA} = \text{BtoA}^2 \text{ without corrupted messages.} \]

\text{Spec}H \text{ is defined in module AB}2H .

So the definition of SpecH should look like this.

We define SpecH in another module called AB2H .
\[
SpecH \triangleq \land \text{ Formula } Spec \text{ of module } AB2 \\
\land \Box \land AtoB = AtoB2 \text{ without corrupted messages.} \\
\land BtoA = BtoA2 \text{ without corrupted messages.}
\]

\textcolor{red}{\text{SpecH} \text{ is defined in module } AB2H.}

Stop the video and download that module now.

So the definition of \textcolor{red}{SpecH} should look like this.

\textcolor{red}{We define SpecH in another module called } AB2H.

\textcolor{red}{Stop the video and download that module now.}
\[ SpecH \triangleq \bigwedge \text{Formula } Spec \text{ of module } AB2 \]
\[ \bigwedge \Box \bigwedge AtoB = AtoB2 \text{ without corrupted messages.} \]
\[ \bigwedge BtoA = BtoA2 \text{ without corrupted messages.} \]

We start by writing this conjunct.
\[ SpecH \triangleq \land \text{Formula Spec of module } AB2 \]
\[ \land \Box \land AtoB = AtoB2 \text{ without corrupted messages.} \]
\[ \land BtoA = BtoA2 \text{ without corrupted messages.} \]

To permit \( AB2H \) to import Spec from \( AB2 \)

To permit module \( AB2H \) to import formula \( Spec \) from module \( AB2 \)

[slide 61]
\[ SpecH \triangleq \bigwedge \text{Formula } Spec \text{ of module } AB2 \]
\[ \bigwedge \Box \bigwedge \text{AtoB} = AtoB2 \text{ without corrupted messages.} \]
\[ \bigwedge \text{BtoA} = BtoA2 \text{ without corrupted messages.} \]

To permit \( AB2H \) to import \( Spec \) from \( AB2 \), it extends the same modules and declares the same constants and variables as \( AB2 \).

\( AB2H \) begins by extending the same modules and declaring the same constants and variables as module \( AB2 \).
\[ SpecH \triangleq \land \text{Formula } Spec \text{ of module } AB2 \]
\[ \land \Box \land AtoB = AtoB2 \text{ without corrupted messages}. \]
\[ \land BtoA = BtoA2 \text{ without corrupted messages}. \]

To permit \( AB2H \) to import \( Spec \) from \( AB2 \), it extends the same modules and declares the same constants and variables as \( AB2 \).

**EXTENDS** \( Integers, Sequences \)

**CONSTANTS** \( Data, Bad \)

**ASSUME** \( Bad \notin (Data \times \{0,1\}) \cup \{0,1\} \)

**VARIABLES** \( AVar, BVar, AtoB, BtoA \)

To permit module \( AB2H \) to import formula \( Spec \) from module \( AB2 \)

\( AB2H \) begins by extending the same modules and declaring the same constants and variables as module \( AB2 \).
\[ SpecH \triangleq \ \land \ \text{Formula Spec of module AB2} \]
\[ \land \ \Box \ \land \ AtoB = AtoB2 \text{ without corrupted messages.} \]
\[ \land \ BtoA = BtoA2 \text{ without corrupted messages.} \]

\[ AB2 \triangleq \text{INSTANCE AB2} \]

The module next imports the definitions from module AB2 with this \textit{instance} statement.

This imports
\( SpecH \triangleq \land \ \text{Formula} \ Spec \ \text{of module} \ AB2 \\
\land \Box \land AtoB = AtoB2 \ \text{without corrupted messages.} \\
\land BtoA = BtoA2 \ \text{without corrupted messages.} \\

AB2 \triangleq \text{INSTANCE} \ AB2

The module next imports the definitions from module \( AB2 \) with this \textit{instance} statement.

This imports \textit{formula} \ Spec \ of module \ AB2 \ as
\[ SpecH \overset{\triangleq}{=} \land AB2!Spec \]
\[ \land \Box \land AtoB = AtoB2 \text{ without corrupted messages.} \]
\[ \land BtoA = BtoA2 \text{ without corrupted messages.} \]

\[ AB2 \overset{\triangleq}{=} \text{INSTANCE } AB2 \]

The module next imports the definitions from module \( AB2 \) with this \textit{instance} statement.

This imports formula \( Spec \) of module \( AB2 \) as \( AB2 \text{ bang } Spec \).
\[ \text{SpecH} \triangleq \land AB^2!\text{Spec} \]
\[ \land \square \land AtoB = AtoB^2 \text{ without corrupted messages}. \]
\[ \land BtoA = BtoA^2 \text{ without corrupted messages}. \]

\[ AB^2 \triangleq \text{INSTANCE AB}^2 \]

The module next imports the definitions from module \( AB^2 \) with this \textit{instance} statement.

This imports formula \textit{Spec} of module \( AB^2 \) as \( AB^2 \text{ bang Spec} \).

To write this part of the definition of \textit{SpecH},

[slide 67]
\[ SpecH \triangleq \land AB2!Spec \]
\[ \land \Box \land AtoB = AtoB^2 \text{ without corrupted messages.} \]
\[ \land BtoA = BtoA^2 \text{ without corrupted messages.} \]

\[ AB2 \triangleq \text{INSTANCE } AB2 \]

**VARIABLES** \( AtoB, \ BtoA \)

The module next imports the definitions from module \( AB2 \) with this *instance* statement.

This imports formula \( Spec \) of module \( AB2 \) as \( AB2 \bang Spec \).

To write this part of the definition of \( SpecH \),

the module has to declare the variables \( AtoB \) and \( BtoA \).
\[ SpecH \overset{\Delta}{=} \land AB2!Spec \]
\[ \land \Box \land AtoB = AtoB2 \text{ without corrupted messages.} \]
\[ \land BtoA = BtoA2 \text{ without corrupted messages.} \]

\[ AB2 \overset{\Delta}{=} \text{INSTANCE } AB2 \]

\textbf{VARIABLES} \( AtoB, BtoA \)

To write this part of the definition,
SpecH \overset{\Delta}{=} \land AB2!Spec
\land \square \land AtoB = AtoB2 \text{ without corrupted messages.}
\land BtoA = BtoA2 \text{ without corrupted messages.}

AB2 \overset{\Delta}{=} \text{INSTANCE } AB2

\text{VARIABLES } AtoB, BtoA

Defines \textit{RemoveBad(seq)} to be the sequence obtained by removing \textit{Bad} elements from a sequence \textit{seq}.

To write this part of the definition, the module next defines the operator \textit{RemoveBad} so that \textit{RemoveBad of seq} is the sequence obtained by removing elements equal to \textit{Bad} from a sequence \textit{seq}.
The definition of course is almost identical to the recursive definition of $RemoveX$ in part 1 of this lecture. It begins with a RECURSIVE declaration.
\[ SpecH \triangleq \land AB2!Spec \]
\[ \land \Box \land AtoB = AtoB2 \text{ without corrupted messages.} \]
\[ \land BtoA = BtoA2 \text{ without corrupted messages.} \]

RECURSIVE \( \text{RemoveBad}(\_ ) \)

\( \text{RemoveBad}(seq) \triangleq \)

The definition of course is almost identical to the recursive definition of \( \text{RemoveX} \) in part 1 of this lecture. It begins with a RECURSIVE declaration.

It then defines \( \text{RemoveBad} \) of \( seq \) to be
\[ SpecH \triangleq \land A B 2!Spec \land \Box \land AtoB = AtoB 2 \text{ without corrupted messages.} \land BtoA = BtoA 2 \text{ without corrupted messages.} \]

**RECURSIVE** \( \text{RemoveBad}(\_ ) \)

\( \text{RemoveBad}(seq) \triangleq \)

\[
\text{IF } seq = \langle \rangle \\
\text{THEN } \langle \rangle \\
\text{ELSE}
\]

The definition of course is almost identical to the recursive definition of \( \text{RemoveX} \) in part 1 of this lecture. It begins with a **RECURSIVE** declaration.

It then defines \( \text{RemoveBad} \) of \( seq \) to be If \( seq \) is the empty sequence, then the empty sequence.

[slide 73]
\[ SpecH \triangleq \wedge AB2!Spec \]
\[ \wedge \Box \wedge AtoB = AtoB2 \text{ without corrupted messages.} \]
\[ \wedge BtoA = BtoA2 \text{ without corrupted messages.} \]

**RECURSIVE** \( RemoveBad(\_\_\_) \)

\[ RemoveBad(seq) \triangleq \]
\[ \text{IF } seq = \langle \rangle \]
\[ \text{THEN } \langle \rangle \]
\[ \text{ELSE IF } Head(seq) = Bad \]
\[ \text{THEN} \]
\[ \text{ELSE} \]

The definition of course is almost identical to the recursive definition of \( RemoveX \) in part 1 of this lecture. It begins with a **RECURSIVE** declaration.

It then defines \( RemoveBad \) of \( seq \) to be If \( seq \) is the empty sequence, then the empty sequence.

else if the head of \( seq \) equals \( Bad \),
\[ SpecH \triangleq \land AB2!Spec \]
\[ \land \Box \land AtoB = AtoB2 \text{ without corrupted messages.} \]
\[ \land BtoA = BtoA2 \text{ without corrupted messages.} \]

The definition of course is almost identical to the recursive definition of \textit{RemoveX} in part 1 of this lecture. It begins with a \texttt{RECURSIVE} declaration.

It then defines \textit{RemoveBad} of \textit{seq} to be If \textit{seq} is the empty sequence, then the empty sequence.

else if the head of \textit{seq} equals \textit{Bad}, then \textit{RemoveBad} of the tail of \textit{seq}.

[slide 75]
\( SpecH \triangleq A B2!Spec \)
\[ \triangleq \Box A \land AtoB = AtoB2 \text{ without corrupted messages.} \]
\[ \land BtoA = BtoA2 \text{ without corrupted messages.} \]

**RECURSIVE** \( RemoveBad(\_ \_ \)\)

\( RemoveBad(seq) \triangleq \)

IF \( seq = \langle \rangle \)

THEN \( \langle \rangle \)

ELSE IF \( Head(seq) = Bad \)

THEN \( RemoveBad(Tail(seq)) \)

ELSE \( \langle Head(seq) \rangle \circ RemoveBad(Tail(seq)) \)

Else, the sequence obtained by prepending the head of \( seq \) to the front of \( RemoveBad \) of the tail of \( seq \).
\[ SpecH \triangleq \land AB2!Spec \]
\[ \land \Box \land AtoB = AtoB2 \text{ without corrupted messages.} \]
\[ \land BtoA = BtoA2 \text{ without corrupted messages.} \]

RECURSIVE \( RemoveBad(_) \)

\( RemoveBad(seq) \triangleq \)
\[
\text{IF } seq = \langle \rangle \\
\text{THEN } \langle \rangle \\
\text{ELSE IF } \text{Head}(seq) = \text{Bad} \\
\text{THEN } RemoveBad(Tail(seq)) \\
\text{ELSE } \langle \text{Head}(seq) \rangle \circ RemoveBad(Tail(seq))
\]

Else, the sequence obtained by prepending the head of \( seq \) to the front of \( RemoveBad \) of the tail of \( seq \).

We can use it to replace these pseudo-expressions
\[ SpecH \triangleq \land AB2!Spec \]
\[ \land \square \land AtoB = RemoveBad(AtoB2) \]
\[ \land BtoA = RemoveBad(BtoA2) \]

**RECURSIVE** \( RemoveBad(\_\_\_\_\_\_\_) \)

\[ RemoveBad(seq) \triangleq \]

IF \( seq = \langle \rangle \)
THEN \( \langle \rangle \)
ELSE IF \( Head(seq) = Bad \)
THEN \( RemoveBad(Tail(seq)) \)
ELSE \( \langle Head(seq) \rangle \circ RemoveBad(Tail(seq)) \)

Else, the sequence obtained by prepending the head of \( seq \) to the front of \( RemoveBad \) of the tail of \( seq \).

We can use it to replace these pseudo-expressions with real expressions.
\[
\begin{align*}
SpecH & \triangleq \bigwedge AB2!Spec \\
& \land \Box \land AtoB = \text{RemoveBad}(AtoB2) \\
& \land BtoA = \text{RemoveBad}(BtoA2)
\end{align*}
\]

Else, the sequence obtained by prepending the head of \(seq\) to the front of \(\text{RemoveBad}\) of the tail of \(seq\).

We can use it to replace these pseudo-expressions with real expressions.

This completes the definition of \(SpecH\), which comes next in the module.

[slide 79]
\[
\text{SpecH} \triangleq \land \ AB2!Spec \\
\land \Box \land \ AtoB = RemoveBad(AtoB2) \\
\land \ BtoA = RemoveBad(BtoA2)
\]

AtoB and BtoA are imaginary variables
\[ SpecH \triangleq \land \ AB2!Spec \]
\[ \land \Box \land AtoB = RemoveBad(AtoB2) \]
\[ \land BtoA = RemoveBad(BtoA2) \]

\( AtoB \) and \( BtoA \) are imaginary variables added to \( AB2!Spec \) to show that it implements the \( AB \) protocol spec.
\[ SpecH \triangleq \land AB2!Spec \]
\[ \land \Box \land AtoB = \text{RemoveBad}(AtoB2) \]
\[ \land BtoA = \text{RemoveBad}(BtoA2) \]

\(AtoB\) and \(BtoA\) are imaginary variables added to \(AB2!Spec\) to show that it implements the \(AB\) protocol spec.

They are not meant to be implemented by the \(AB2\) protocol.

\[ \text{[slide 82]} \]
\[ \text{SpecH } \triangleq \land AB2!Spec \]
\[ \land \Box \land AtoB = \text{RemoveBad}(AtoB2) \]
\[ \land BtoA = \text{RemoveBad}(BtoA2) \]

\( AtoB \) and \( BtoA \) are imaginary variables added to \( AB2!Spec \) to show that it implements the \( AB \) protocol spec. They are not meant to be implemented by the \( AB2 \) protocol.

If we ignore the values of \( AtoB \) and \( BtoA \), then \( \text{SpecH} \) and \( AB2!Spec \) allow the same behaviors.

\( AtoB \) and \( BtoA \) are imaginary variables added to \( AB2!Spec \) to show that it implements the \( AB \) protocol spec. They are not meant to be implemented by the \( AB2 \) protocol.

If we ignore the values of \( AtoB \) and \( BtoA \), then \( \text{SpecH} \) and \( AB2!Spec \) allow the same behaviors.
CHECKING IMPLEMENTATION
Our goal is to check:

\[
\text{THEOREM } SpecH \Rightarrow \text{ formula } Spec \text{ of module } AB
\]

Remember that our goal is to check this theorem
Our goal is to check:

\[ \text{THEOREM } SpecH \Rightarrow \text{ formula } Spec \text{ of module } AB \]

which asserts that \( AB2 \) implements \( AB \).

Remember that our goal is to check this theorem which asserts that the \( AB2 \) protocol implements the \( AB \) protocol.

But first we have to write it in TLA\(^+\).
Our goal is to check:

\[ \text{THEOREM } SpecH \Rightarrow \text{ formula } Spec \text{ of module } AB \]

This is defined in \( AB2H \).

Remember that our goal is to check this theorem which asserts that the \( AB2 \) protocol implements the \( AB \) protocol.

But first we have to write it in TLA⁺.

We just defined \( SpecH \) in module \( AB2H \).
Our goal is to check:

\textbf{THEOREM} \quad SpecH \Rightarrow \textbf{formula} \ Spec \ of \ module \ AB

Have to write in $AB2H$.

Remember that our goal is to check this theorem which asserts that the $AB2$ protocol implements the $AB$ protocol.

But first we have to write it in TLA$^+$. 

We just defined $SpecH$ in module $AB2H$.

We now have to write formula $Spec$ of module $AB$ in module $AB2H$. But that's easy.
Our goal is to check:

\[
\text{THEOREM } SpecH \Rightarrow \text{formula } Spec \text{ of module } AB
\]

\[
AB \overset{\triangle}{=} \text{INSTANCE } AB
\]

We just add this INSTANCE statement to module \( AB2H \).
Our goal is to check:

\[
\text{THEOREM } SpecH \Rightarrow \text{formula } Spec \text{ of module } AB
\]

\[
AB \overset{\Delta}{=} \text{INSTANCE } AB
\]

\[
\text{THEOREM } SpecH \Rightarrow AB!Spec
\]

We just add this \text{INSTANCE} statement to module \text{AB2H}.

which defines \text{AB} \bang \text{Spec} to be this formula.
Our goal is to check:

$$\text{THEOREM } SpecH \Rightarrow \text{ formula } Spec \text{ of module } AB$$

$$AB \triangleq \text{ INSTANCE } AB$$

$$\text{THEOREM } SpecH \Rightarrow AB!Spec$$

We just add this \text{INSTANCE} statement to module $AB2H$.

which defines $AB$ bang $Spec$ to be this formula.
THEOREM \( SpecH \Rightarrow AB!Spec \)

TLC can’t check this theorem
THEOREM \( \text{SpecH} \Rightarrow AB!\text{Spec} \)

TLC can’t check this theorem
THEOREM \( SpecH \implies AB!Spec \)

TLC can’t check this theorem

TLC can’t check this theorem
\[ SpecH \triangleq \land AB2!Spec \]
\[ \land \Box \land AtoB = RemoveBad(AtoB2) \]
\[ \land BtoA = RemoveBad(BtoA2) \]

THEOREM \[ SpecH \implies AB!Spec \]

TLC can’t check this theorem because \( SpecH \)
\[ SpecH \triangleq \land AB2!Spec \land \Box \land AtoB = RemoveBad(AtoB2) \land BtoA = RemoveBad(BtoA2) \]

**THEOREM** \( SpecH \Rightarrow AB!Spec \)

TLC can’t check this theorem because \( SpecH \) doesn’t have the standard form for a TLA\(^+\) safety spec:

\[ InitH \land \Box [NextH]_{varsH} \]

TLC can’t check this theorem because \( SpecH \) doesn’t have the standard form for a TLA\(^+\) safety spec — which has an initial-state formula and a next-state action.

[slide 96]
We could solve this problem by rewriting $SpecH$. 

You can read module $AB2H$ to see how it's done.

Here, I'll take a longer approach that leads to greater insight into implementation.
We could solve this problem by rewriting $SpecH$.

This is done in module $AB2H$ by defining a specification $SpecHH$ that TLC can handle.
We could solve this problem by rewriting $SpecH$.

This is done in module $AB2H$ by defining a specification $SpecHH$ that TLC can handle and is equivalent to $SpecH$.

[slide 99]
We could solve this problem be rewriting $SpecH$.

This is done in module $AB2H$ by defining a specification $SpecHH$ that TLC can handle and is equivalent to $SpecH$.

You can read module $AB2H$ to see how it’s done.
We could solve this problem by rewriting $SpecH$. 

This is done in module $AB2H$ by defining a specification $SpecHH$ that TLC can handle and is equivalent to $SpecH$.

You can read module $AB2H$ to see how it’s done.

I’ll take a longer approach
We could solve this problem by rewriting $SpecH$.

This is done in module $AB2H$ by defining a specification $SpecHH$ that TLC can handle and is equivalent to $SpecH$.

You can read module $AB2H$ to see how it’s done.

I’ll take a longer approach that leads to greater insight into implementation.
SIMPLIFYING REFINEMENT
\[ SpecH \overset{\Delta}{=} \land \land AB2!Spec \]
\[ \land \Box \land AtoB = RemoveBad(AtoB2) \]
\[ \land BtoA = RemoveBad(BtoA2) \]

\[ AB \overset{\Delta}{=} \text{INSTANCE } AB \]

THEOREM \[ SpecH \Rightarrow AB!Spec \]

This is where we are in Module \( AB2H \).
\[ SpecH \triangleq \land AB2!Spec \]
\[ \land \Box \land AtoB = RemoveBad(AtoB2) \]
\[ \land BtoA = RemoveBad(BtoA2) \]

\[ AB \triangleq \text{INSTANCE} \ AB \]

**THEOREM**  \[ SpecH \Rightarrow AB!Spec \]

By the Temporal Substitution Rule

This is where we are in Module \( AB2H \).

By the Temporal Substitution Rule
$SpecH \triangleq □(∧ AtoB = RemoveBad(AtoB2) ∧ BtoA = RemoveBad(BtoA2))$

$AB \triangleq INSTANCE AB$

THEOREM $SpecH \Rightarrow AB!Spec$

By the Temporal Substitution Rule, this formula implies

This is where we are in Module $AB2H$.

By the Temporal Substitution Rule this always formula implies
By the Temporal Substitution Rule, this formula implies

\[ AB!Spec = (AB!Spec \text{ with } AtoB \leftarrow RemoveBad(AtoB2), \]
\[ BtoA \leftarrow RemoveBad(BtoA2) \]

This is where we are in Module \( AB2H \).

By the Temporal Substitution Rule this \textit{always} formula implies that \( AB \) bang \textit{Spec} equals \( AB \) bang \textit{Spec} with this substitution.
\textit{SpecH} \overset{\triangle}{=} \land AB2!\text{Spec} \\
\land \square \land AtoB = \text{RemoveBad}(AtoB2) \\
\land BtoA = \text{RemoveBad}(BtoA2)

\hspace{1cm} AB \overset{\triangle}{=} \text{INSTANCE } AB

THEOREM \hspace{0.5cm} \text{SpecH} \Rightarrow AB!\text{Spec}

\text{SpecH} \text{ implies } \hspace{0.5cm} AB!\text{Spec} = (AB!\text{Spec WITH AtoB} \leftarrow \text{RemoveBad}(AtoB2), \\
\hspace{2.5cm} BtoA \leftarrow \text{RemoveBad}(BtoA2))

And since \text{SpecH} implies the \textit{always} formula, it also implies this equality.
\[ SpecH \triangleq \land AB2!Spec \]
\[ \land \Box \land AtoB = RemoveBad(AtoB2) \]
\[ \land BtoA = RemoveBad(BtoA2) \]

\[ AB \triangleq \text{INSTANCE } AB \]

**THEOREM** \( SpecH \implies AB!Spec \)

\[ SpecH \text{ implies } \]
\[ AB!Spec = (AB!Spec \text{ with } AtoB \leftarrow RemoveBad(AtoB2), \]
\[ BtoA \leftarrow RemoveBad(BtoA2) ) \]

And since \( SpecH \) implies the *always* formula, it also implies this equality.

Therefore, this theorem

[slide 109]
And since $SpecH$ implies the always formula, it also implies this equality.

Therefore, this theorem is equivalent to the theorem we get by replacing $AB$ bang $Spec$ by $AB$ bang $Spec$ with the substitutions.

[slide 110]
\[ SpecH \triangleq \land AB!Spec \land \Box \land AtoB = RemoveBad(AtoB2) \land BtoA = RemoveBad(BtoA2) \]

\[ AB \triangleq \text{INSTANCE } AB \]

**THEOREM** \( SpecH \Rightarrow AB!Spec \)

**THEOREM** \( SpecH \Rightarrow (AB!Spec \text{ WITH } AtoB \leftrightarrow RemoveBad(AtoB2), BtoA \leftrightarrow RemoveBad(BtoA2) ) \)

And since \( SpecH \) implies the \textit{always} formula, it also implies this equality.

Therefore, this theorem is equivalent to the theorem we get by replacing \( AB \) bang \( Spec \) by \( AB \) bang \( Spec \) with the substitutions.

[slide 111]
\[ SpecH \triangleq \wedge AB^2!Spec \]
\[ \wedge □ \wedge AtoB = RemoveBad(AtoB^2) \]
\[ \wedge BtoA = RemoveBad(BtoA^2) \]

\[ AB \triangleq INSTANCE AB \]

**THEOREM** \(SpecH \Rightarrow AB!Spec\)

**THEOREM** \(SpecH \Rightarrow (AB!Spec \text{ WITH } AtoB \leftarrow RemoveBad(AtoB^2), \]
\[\quad \quad BtoA \leftarrow RemoveBad(BtoA^2) ) \]

Does not contain \(AtoB\) or \(BtoA\).

Since this formula is obtained by substituting for \(AtoB\) and \(BtoA\), it does not contain those two variables.
\[ SpecH \triangleq \land AB2!Spec \land \Box \land AtoB = RemoveBad(AtoB2) \land BtoA = RemoveBad(BtoA2) \]

\[ AB \triangleq \text{INSTANCE } AB \]

**THEOREM** \[ SpecH \Rightarrow AB!Spec \]

**THEOREM** \[ SpecH \Rightarrow (AB!Spec \text{ WITH } AtoB \leftarrow \text{RemoveBad}(AtoB2), BtoA \leftarrow \text{RemoveBad}(BtoA2)) \]

Does not contain \( AtoB \) or \( BtoA \).

Since this formula is obtained by substituting for \( AtoB \) and \( BtoA \), it does not contain those two variables.

Hence, in the theorem,
\[ SpecH \triangleq \land AB2!Spec \]
\[ \land \Box \land AtoB = RemoveBad(AtoB2) \]
\[ \land BtoA = RemoveBad(BtoA2) \]

\[ AB \triangleq INSTANCE\ AB \]

**THEOREM** \[ SpecH \Rightarrow AB!Spec \]

**THEOREM** \[ SpecH \Rightarrow (AB!Spec \text{ WITH } AtoB \leftarrow RemoveBad(AtoB2), BtoA \leftarrow RemoveBad(BtoA2)) \]

Does not contain \( AtoB \) or \( BtoA \).

Since this formula is obtained by substituting for \( AtoB \) and \( BtoA \), it does not contain those two variables.

Hence, in the theorem, this *always* conjunct of \( SpecH \) is irrelevant.
\[
\text{SpecH} \triangleq \wedge \text{AB2!Spec} \\
\wedge \Box \wedge \text{AtoB} = \text{RemoveBad}(\text{AtoB2}) \\
\wedge \text{BtoA} = \text{RemoveBad}(\text{BtoA2})
\]

\[AB \triangleq \text{INSTANCE AB}\]

THEOREM \text{SpecH} \Rightarrow \text{AB!Spec}

\begin{align*}
\text{THEOREM } & \text{SpecH} \Rightarrow (\text{AB!Spec} \text{ WITH } A\text{toB} \leftarrow \text{RemoveBad}(A\text{toB2}), \\
& \text{BtoA} \leftarrow \text{RemoveBad}(B\text{toA2}) )
\end{align*}

Since this formula is obtained by substituting for \(A\text{toB}\) and \(B\text{toA}\), it does not contain those two variables.

Hence, in the theorem, this \textit{always} conjunct of \text{SpecH} is irrelevant.

So we can replace \text{SpecH} in the theorem

[slide 115]
\[ SpecH \triangleq \land \Box^2 \land AtoB = RemoveBad(AtoB2) \land BtoA = RemoveBad(BtoA2) \]

\[ AB \triangleq INSTANCE AB \]

THEOREM \[ SpecH \Rightarrow AB!Spec \]

THEOREM \[ SpecH \Rightarrow (AB!Spec \text{ with } AtoB \leftarrow RemoveBad(AtoB2), BtoA \leftarrow RemoveBad(BtoA2) ) \]

Since this formula is obtained by substituting for \( AtoB \) and \( BtoA \), it does not contain those two variables.

Hence, in the theorem, this \textit{always} conjunct of \( SpecH \) is irrelevant.

So we can replace \( SpecH \) in the theorem \textit{by just this conjunct}
\[ SpecH \overset{\Delta}{=} \bigwedge AB2!Spec \]
\[ \bigwedge \Box \bigwedge AtoB = RemoveBad(AtoB2) \]
\[ \bigwedge BtoA = RemoveBad(BtoA2) \]

\[ AB \overset{\Delta}{=} \text{INSTANCE } AB \]

\textbf{THEOREM} \  SpecH \ \Rightarrow \ AB!Spec 

\textbf{THEOREM} \ SpecH \ \Rightarrow \ (AB!Spec \ \text{WITH} \ AtoB \leftarrow RemoveBad(AtoB2), \ BtoA \leftarrow RemoveBad(BtoA2)) 

\textbf{THEOREM} \ AB2!Spec \ \Rightarrow \ (AB!Spec \ \text{WITH} \ AtoB \leftarrow RemoveBad(AtoB2), \ BtoA \leftarrow RemoveBad(BtoA2)) 

Since this formula is obtained by substituting for \( AtoB \) and \( BtoA \), it does not contain those two variables.

Hence, in the theorem, this \textit{always} conjunct of \( SpecH \) is irrelevant.

So we can replace \( SpecH \) in the theorem by just this conjunct to get this equivalent theorem.
\[ SpecH \triangleq \land AB2!Spec \land \Box \land AtoB = RemoveBad(AtoB2) \land BtoA = RemoveBad(BtoA2) \]

\[ AB \triangleq \text{INSTANCE } AB \]

THEOREM \[ SpecH \Rightarrow AB!Spec \]

THEOREM \[ SpecH \Rightarrow (AB!Spec \text{ WITH } AtoB \leftarrow RemoveBad(AtoB2), BtoA \leftarrow RemoveBad(BtoA2)) \]

THEOREM \[ AB2!Spec \Rightarrow (AB!Spec \text{ WITH } AtoB \leftarrow RemoveBad(AtoB2), BtoA \leftarrow RemoveBad(BtoA2)) \]

Since this formula is obtained by substituting for \( AtoB \) and \( BtoA \), it does not contain those two variables.

Hence, in the theorem, this \textit{always} conjunct of \( SpecH \) is irrelevant.

So we can replace \( SpecH \) in the theorem by just this conjunct to get this equivalent theorem.

[slide 118]
\[ SpecH \triangleq \land AB2!Spec \land \Box \land AtoB = RemoveBad(AtoB2) \land BtoA = RemoveBad(BtoA2) \]

\[ AB \triangleq \text{INSTANCE } AB \]

THEOREM  \[ SpecH \Rightarrow AB!Spec \]

THEOREM  \[ SpecH \Rightarrow (AB!Spec \text{ WITH } AtoB \leftarrow RemoveBad(AtoB2), BtoA \leftarrow RemoveBad(BtoA2)) \]

THEOREM  \[ AB2!Spec \Rightarrow (AB!Spec \text{ WITH } AtoB \leftarrow RemoveBad(AtoB2), BtoA \leftarrow RemoveBad(BtoA2)) \]

So we can replace the theorem we want to prove
\[
Spec \underline{H} \overset{\Delta}{=} \land AB^2!Spec \\
\land \Box \land AtoB = RemoveBad(AtoB^2) \\
\land BtoA = RemoveBad(BtoA^2)
\]

\[
AB \overset{\Delta}{=} \text{INSTANCE AB}
\]

\[
\text{THEOREM } AB^2!Spec \Rightarrow (AB!Spec \text{ WITH } AtoB \leftarrow RemoveBad(AtoB^2), \\
BtoA \leftarrow RemoveBad(BtoA^2))
\]

So we can replace the theorem we want to prove by this one.

[slide 120]
\[ SpecH \triangleq \land AB2!Spec \]
\[ \land \Box \land AtoB = \text{RemoveBad}(AtoB2) \]
\[ \land BtoA = \text{RemoveBad}(BtoA2) \]

\[ AB \triangleq \text{INSTANCE AB} \]

**THEOREM** \( AB2!Spec \Rightarrow (AB!Spec \text{ WITH } AtoB \leftarrow \text{RemoveBad}(AtoB2), BtoA \leftarrow \text{RemoveBad}(BtoA2) ) \)

And we can just check this theorem.
\[
\text{SpecH} \triangleq \Box \Box \text{AtoB} \iff \text{RemoveBad}(\text{AtoB}_2) \\
\Box \Box \text{BtoA} = \text{RemoveBad}(\text{BtoA}_2)
\]

\[
\text{AB} \triangleq \text{INSTANCE AB}
\]

THEOREM \(\text{AB}_2!\text{Spec} \Rightarrow (\text{AB}!\text{Spec} \text{ WITH } \text{AtoB} \leftarrow \text{RemoveBad}(\text{AtoB}_2), \text{BtoA} \leftarrow \text{RemoveBad}(\text{BtoA}_2))\)

And we can just check this theorem.

First, notice that \(\text{SpecH}\) doesn’t appear in the theorem any more, so we don’t need to define it.
And we can just check this theorem.

First, notice that $SpecH$ doesn’t appear in the theorem any more, so we don’t need to define it.
These statements are in module $AB2H$.

$AB \triangleq \text{INSTANCE } AB$

\text{THEOREM } AB2!Spec \Rightarrow (AB!Spec \text{ WITH } AtoB \leftarrow \text{RemoveBad}(AtoB2),
\hspace{1cm} BtoA \leftarrow \text{RemoveBad}(BtoA2))$

And we can just check this theorem.

First, notice that $SpecH$ doesn’t appear in the theorem any more, so we don’t need to define it.

The instance statement and the theorem are in module $AB2H$. 

[slide 124]
These statements are in module $AB2H$.

Let’s move them to module $AB2$.

$$AB \triangleq \text{INSTANCE } AB$$

**THEOREM** $AB2!Spec \Rightarrow (AB!Spec \text{ with } AtoB \leftarrow \text{RemoveBad}(AtoB2), BtoA \leftarrow \text{RemoveBad}(BtoA2))$

And we can just check this theorem.

First, notice that $SpecH$ doesn’t appear in the theorem any more, so we don’t need to define it.

The instance statement and the theorem are in module $AB2H$.

Let’s move them to module $AB2$.

[slide 125]
These statements are in module $AB2H$.

Let’s move them to module $AB$.  

$$AB \triangleq \text{INSTANCE } AB$$

THEOREM $\boxed{AB!Spec} \Rightarrow (\text{AB}!\text{Spec WITH} \ AtoB \leftarrow \text{RemoveBad}(\text{AtoB2}), \ BtoA \leftarrow \text{RemoveBad}(\text{BtoA2}) )$

When we do that, the formula called $AB!Spec$ in module $AB2H$
These statements are in module $AB2H$.

Let’s move them to module $AB2$.

$$AB \triangleq \text{INSTANCE } AB$$

**THEOREM** $Spec \Rightarrow (AB!Spec \text{ WITH } AtoB \leftarrow \text{RemoveBad}(AtoB2) , BtoA \leftarrow \text{RemoveBad}(BtoA2))$

When we do that, the formula called $AB2!Spec$ in module $AB2H$ is simply called $Spec$. [slide 127]
These statements are in module $AB2H$.

Let’s move them to module $AB2$.

$$AB \overset{\Delta}{=} \text{INSTANCE } AB$$

THEOREM $Spec \Rightarrow (AB!Spec \text{ WITH } AtoB \leftarrow \text{RemoveBad}(AtoB2),$

$$BtoA \leftarrow \text{RemoveBad}(BtoA2) )$$

When we do that, the formula called $AB2!Spec$ in module $AB2H$ is simply called $Spec$. 

[slide 128]
These statements are in module $AB2H$. Let’s move them to module $AB2$.

\[ AB \triangleq \text{INSTANCE } AB \]

THEOREM $Spec \Rightarrow (AB!Spec \text{ WITH } AtoB \leftarrow \text{RemoveBad}(AtoB2), BtoA \leftarrow \text{RemoveBad}(BtoA2))$

TLC can handle this specification.

When we do that, the formula called $AB2!Spec$ in module $AB2H$ is simply called $Spec$.

Inside module $AB2$, $Spec$ is an ordinary specification that TLC can handle.
These statements are in module $AB2H$. Let’s move them to module $AB2$.

$$AB \overset{\triangle}{=} \text{INSTANCE } AB$$

THEOREM $Spec \Rightarrow (AB!Spec \text{ WITH } AtoB \leftarrow \text{RemoveBad}(AtoB2), BtoA \leftarrow \text{RemoveBad}(BtoA2))$

But this isn’t TLA$^+$. When we do that, the formula called $AB2!Spec$ in module $AB2H$ is simply called $Spec$.

Inside module $AB2$, $Spec$ is an ordinary specification that TLC can handle. But this WITH formula is just a notation that I’m using here. It’s not legal TLA$^+$. To see how to write it in TLA$^+$,
These statements are in module $AB2H$.
Let’s move them to module $AB2$.

$AB \triangleq \text{INSTANCE } AB$

$\text{THEOREM } Spec \Rightarrow (AB!Spec \text{ WITH } AtoB \leftrightarrow \text{RemoveBad}(AtoB2),
BtoA \leftrightarrow \text{RemoveBad}(BtoA2))$

When we do that, the formula called $AB2!Spec$ in module $AB2H$ is simply called $Spec$.

Inside module $AB2$, $Spec$ is an ordinary specification that TLC can handle.

But this WITH formula is just a notation that I’m using here. It’s not legal TLA$^+$. To see how to write it in TLA$^+$, we need to examine the INSTANCE statement.
After expanding all definitions,
\[ AB \triangleq \text{INSTANCE } AB \]

After expanding all definitions,

After expanding all definitions,
\( AB \overset{\Delta}{=} \text{INSTANCE} \ AB \)

After expanding all definitions, formula \( Spec \) of \( AB \) contains only TLA\(^+\) operators.
\[ AB \triangleq \text{INSTANCE } AB \]

After expanding all definitions, formula \( Spec \) of \( AB \) contains only TLA\(^+\) operators and the declared symbols of \( AB \):

After expanding all definitions, formula \( Spec \) of module \( AB \) contains only TLA\(^+\) operators and the declared symbols of the module, which are:
\[ AB \triangleq \text{INSTANCE } AB \]

After expanding all definitions, formula \( Spec \) of \( AB \) contains only TLA\(^+\) operators and the declared symbols of \( AB : \)

\[ Data, \]

After expanding all definitions, formula \( Spec \) of module \( AB \) contains only TLA\(^+\) operators and the declared symbols of the module, which are: The constant \( Data \)
\[ AB \triangleq \text{INSTANCE } AB \]

After expanding all definitions, formula \( Spec \) of \( AB \) contains only TLA\(^+\) operators and the declared symbols of \( AB \):

\[ Data, AVar, BVar, AtoB, BtoA \]

After expanding all definitions, formula \( Spec \) of module \( AB \) contains only TLA\(^+\) operators and the declared symbols of the module, which are: The constant \( Data \) and the module's four variables.
After expanding all definitions, formula $Spec$ of $AB$ contains only TLA$^+$ operators and the declared symbols of $AB$:

$$Data, A\text{Var}, B\text{Var}, A\text{toB}, B\text{toA}$$

To import a definition from $AB$ into an arbitrary module $M$, 

...
After expanding all definitions, formula $Spec$ of $AB$ contains only TLA$^+$ operators and the declared symbols of $AB$:

$\text{Data}, \text{AVar}, \text{BVar}, \text{AtoB}, \text{BtoA}$

To import a definition from $AB$ into an arbitrary module $M$, we must substitute expressions of module $M$ for those symbols.
After expanding all definitions, formula $Spec$ of $AB$ contains only TLA$^+$ operators and the declared symbols of $AB$:

$$Data, AVar, BVar, AtoB, BtoA$$

To import a definition from $AB$ into an arbitrary module $M$, we must substitute expressions of $M$ for those symbols.

This is done by a $WITH$ clause.
This is done by a WITH clause having this syntax,
\[ AB \overset{\Delta}{=} \text{INSTANCE } AB \text{ WITH } \begin{align*}
\text{Data} & \leftarrow \ldots, \\
\text{AVar} & \leftarrow \ldots, \\
\text{BVar} & \leftarrow \ldots, \\
\text{AtoB} & \leftarrow \ldots, \\
\text{BtoA} & \leftarrow \ldots
\end{align*} \]

The declared constants and variables of \( AB \).

This is done by a \textit{WITH} clause having this syntax, where these are the declared constants and variables of the instantiated module \( AB \).
The declared constants and variables of \( AB \).

The expressions of module \( M \) to be substituted for them.

This is done by a \texttt{WITH} clause having this syntax, where these are the declared constants and variables of the instantiated module \( AB \).

And these are the expressions of the current module \( M \) to be substituted for them.
This is done by a `WITH` clause having this syntax, where these are the declared constants and variables of the instantiated module $AB$.

And these are the expressions of the current module $M$ to be substituted for them.
When we instantiated module $AB$ in module $AB2H$, 

\[
AB \triangleq \text{INSTANCE } AB \text{ WITH } Data \gets \ldots, \\
AVar \gets \ldots, \ BVar \gets \ldots, \\
AtoB \gets \ldots, \ BtoA \gets \ldots 
\]
When we instantiated module $AB$ in module $AB2H$, for each of these declared symbols of module $AB$
When we instantiated module $AB$ in module $AB2H$, for each of these declared symbols of module $AB$ we substituted the symbols of the same name from module $AB2H$. 
When we instantiated module \( AB \) in module \( AB2H \), for each of these declared symbols of module \( AB \) we substituted the symbols of the same name from module \( AB2H \).

Substituting a symbol of the same name for a symbol of the instantiated module is the default.
When we instantiated module $AB$ in module $AB2H$, for each of these declared symbols of module $AB$ we substituted the symbols of the same name from module $AB2H$.

Substituting a symbol of the same name for a symbol of the instantiated module is the default if we omit a substitution for that symbol from the WITH clause.
This is the default if we omit a substitution from the WITH clause.

So we could eliminate the WITH clause in module $AB2H$.

So we could eliminate the entire WITH clause from the INSTANCE statement in module $AB2H$.
In module $AB2$
In module $AB2$

\[ AB \overset{\Delta}{=} \text{INSTANCE } AB \text{ WITH } Data \leftarrow Data, \\
AVar \leftarrow AVar, BVar \leftarrow BVar, \\
AtoB \leftarrow AtoB, BtoA \leftarrow BtoA \]
In module $AB2$ we want the default substitutions for $Data$, $AVar$, and $BVar$,
In module $AB2$ we want the default substitutions for $Data$, $AVar$, and $BVar$, so we can omit them from the `WITH` clause.
In module $AB2$ we want the default substitutions for $Data$, $AVar$, and $BVar$, so we can omit them from the $WITH$ clause.
The symbols $AtoB$ and $BtoA$ are not declared in module $AB2$, so we need to substitute for them some expressions we can write in $AB2$.

If you remember how we got to this point, you should be able to guess that we’re going to substitute
The symbols $AtoB$ and $BtoA$ are not declared in module $AB2$, so we need to substitute for them some expressions we can write in $AB2$.

If you remember how we got to this point, you should be able to guess that we’re going to substitute these expressions for them – after adding the definition of $RemoveBad$ to module $AB2$. 

[slide 157]
Recall that we were trying to check this theorem
Recall that we were trying to check this theorem and we faced the problem that this isn’t a TLA\(^+\) formula.
\[ AB \triangleq \text{INSTANCE } AB \text{ WITH } AtoB \leftarrow \text{RemoveBad}(AtoB2), \]
\[ BtoA \leftarrow \text{RemoveBad}(BtoA2) \]

**THEOREM** \( Spec \Rightarrow (AB!Spec \text{ WITH } AtoB \leftarrow \text{RemoveBad}(AtoB2), \]
\[ BtoA \leftarrow \text{RemoveBad}(BtoA2) ) \]

This non-TLA\(^+\) formula

Recall that we were trying to check this theorem and we faced the problem that this isn’t a TLA\(^+\) formula.

But this non-TLA\(^+\) formula
\begin{align*}
\text{THEOREM } Spec & \Rightarrow AB!Spec \\
& \quad \quad \quad \quad \quad \text{WITH } AtoB \leftarrow \text{RemoveBad}(AtoB2), \\
& \quad \quad \quad \quad \quad \quad \quad \quad BtoA \leftarrow \text{RemoveBad}(BtoA2)
\end{align*}

This non-TLA\(^+\) formula can now be written as \( AB!Spec \).

Recall that we were trying to check this theorem and we faced the problem that this isn’t a TLA\(^+\) formula.

But this non-TLA\(^+\) formula can now be written simply as \( AB \text{ bang } Spec \).
MODULE AB2

\[ AB \eq INSTANCE AB \text{ WITH } AtoB \leftarrow \text{RemoveBad}(AtoB2), \]
\[ BtoA \leftarrow \text{RemoveBad}(BtoA2) \]

THEOREM Spec \Rightarrow AB!Spec \text{ WITH } AtoB \leftarrow \text{RemoveBad}(AtoB2), \]
\[ BtoA \leftarrow \text{RemoveBad}(BtoA2) \]

This non-TLA\(^+\) formula can now be written as \(AB!Spec\). The substitution is done by the INSTANCE statement.

Recall that we were trying to check this theorem and we faced the problem that this isn’t a TLA\(^+\) formula.

But this non-TLA\(^+\) formula can now be written simply as \(AB\) bang \(Spec\).

Because the substitutions we wanted the formula to express are performed by the INSTANCE statement.

[slide 162]
Recall that we were trying to check this theorem and we faced the problem that this isn’t a TLA+ formula.

But this non-TLA+ formula can now be written simply as $AB\ bang Spec$

Because the substitutions we wanted the formula to express are performed by the INSTANCE statement.
MODULE $AB2$

$$AB \triangleq \text{INSTANCE} \ AB \ \text{WITH} \ AtoB \leftarrow \text{RemoveBad}(AtoB2),$$

$$BtoA \leftarrow \text{RemoveBad}(BtoA2)$$

THEOREM $Spec \Rightarrow AB!Spec$

TLC can check this theorem.

And TLC can now check this theorem.

Whew! We’ve finally reached our goal. But it took us so long, you may have forgotten why we wanted to get here. So, let’s review what we’ve accomplished.
WHAT WE DID AND WHY
The $AB2$ protocol implements the $AB$ protocol, where $\text{RemoveBad}(AtoB2)$ implements $AtoB$ and $\text{RemoveBad}(BtoA2)$ implements $BtoA$.

We saw that the $AB2$ protocol implements the $AB$ protocol, where $\text{RemoveBad}$ of $AtoB2$ implements variable $AtoB$ of $AB$, and $\text{RemoveBad}$ of $BtoA2$ implements variable $BtoA$ of $AB$. 

[slide 166]
The $AB2$ protocol implements the $AB$ protocol, where
$\text{RemoveBad}(AtoB2)$ implements $AtoB$ and
$\text{RemoveBad}(BtoA2)$ implements $BtoA$.

This means

We then saw that this means that

[slide 167]
The $AB2$ protocol implements the $AB$ protocol, where $RemoveBad(AtoB2)$ implements $AtoB$ and $RemoveBad(BtoA2)$ implements $BtoA$.

means

\[
\text{THEOREM } \left( \begin{array}{c}
\wedge \text{Spec of } AB2 \\
\wedge \Box \wedge AtoB = RemoveBad(AtoB2) \\
\wedge BtoA = RemoveBad(BtoA2) \\
\end{array} \right) \Rightarrow \text{Spec of } AB
\]

We saw that the $AB2$ protocol implements the $AB$ protocol, where $RemoveBad$ of $AtoB2$ implements variable $AtoB$ of $AB$, and $RemoveBad$ of $BtoA2$ implements variable $BtoA$ of $AB$.

We then saw that this means that this theorem is true,

[slide 168]
The $AB2$ protocol implements the $AB$ protocol, where $\text{RemoveBad}(AtoB2)$ implements $AtoB$ and $\text{RemoveBad}(BtoA2)$ implements $BtoA$.

means

$$\text{THEOREM} \left( \land \text{Spec of } AB2 \land \square \land AtoB = \text{RemoveBad}(AtoB2) \land BtoA = \text{RemoveBad}(BtoA2) \right) \Rightarrow \text{Spec of } AB$$

$SpecH$

We saw that the $AB2$ protocol implements the $AB$ protocol, where $\text{RemoveBad}$ of $AtoB2$ implements variable $AtoB$ of $AB$, and $\text{RemoveBad}$ of $BtoA2$ implements variable $BtoA$ of $AB$.

We then saw that this means that this theorem is true, where this is the formula we called $SpecH$.
The $AB2$ protocol implements the $AB$ protocol, where $\text{RemoveBad}(AtoB2)$ implements $AtoB$ and $\text{RemoveBad}(BtoA2)$ implements $BtoA$.

Th\[slide 170\]

We saw that the $AB2$ protocol implements the $AB$ protocol, where $\text{RemoveBad}$ of $AtoB2$ implements variable $AtoB$ of $AB$, and $\text{RemoveBad}$ of $BtoA2$ implements variable $BtoA$ of $AB$.

We then saw that this means that this theorem is true, where this is the formula we called $SpecH$. [slide 170]
The $AB2$ protocol implements the $AB$ protocol, where
\[ RemoveBad(AtoB2) \text{ implements } AtoB \] and
\[ RemoveBad(BtoA2) \text{ implements } BtoA \].

means

\[
\text{THEOREM } \left( \land \text{ Spec of } AB2 \\
\land \Box \land AtoB = RemoveBad(AtoB2) \\
\land BtoA = RemoveBad(BtoA2) \right) \Rightarrow \text{ Spec of } AB
\]

which in $AB2$ is equivalent to

And we then saw that in module $AB2$ we can write an equivalent assertion as
The $AB2$ protocol implements the $AB$ protocol, where
\[ \text{RemoveBad}(AtoB2) \text{ implements } AtoB \text{ and } \]
\[ \text{RemoveBad}(BtoA2) \text{ implements } BtoA \, . \]

means

\[
\text{THEOREM} \left( \land \text{Spec of } AB2 \\
\land \Box \land AtoB = \text{RemoveBad}(AtoB2) \\
\land BtoA = \text{RemoveBad}(BtoA2) \right) \Rightarrow \text{Spec of } AB
\]

which in $AB2$ is equivalent to

\[
AB \triangleq \text{INSTANCE } AB \text{ WITH } AtoB \leftarrow \text{RemoveBad}(AtoB2), \]
\[
BtoA \leftarrow \text{RemoveBad}(BtoA2)
\]

\[
\text{THEOREM } \text{Spec } \Rightarrow AB!\text{Spec}
\]

And we then saw that in module $AB2$ we can write an equivalent assertion as this INSTANCE statement and theorem.
The $AB2$ protocol implements the $AB$ protocol, where $\text{RemoveBad}(AtoB2)$ implements $AtoB$ and $\text{RemoveBad}(BtoA2)$ implements $BtoA$.

\[ \begin{align*} AB & \triangleq \text{INSTANCE } AB \text{ WITH } \\
& AtoB \leftarrow \text{RemoveBad}(AtoB2), \\
& BtoA \leftarrow \text{RemoveBad}(BtoA2) \\
\end{align*} \]

**THEOREM** $\text{Spec} \Rightarrow AB!\text{Spec}$

And we then saw that in module $AB2$ we can write an equivalent assertion as this \text{INSTANCE} statement and theorem.

These substitutions are called a refinement mapping.
The \textit{AB2} protocol implements the \textit{AB} protocol under the refinement mapping

\begin{align*}
AtoB & \leftarrow \text{RemoveBad}(AtoB2), \\
BtoA & \leftarrow \text{RemoveBad}(BtoA2)
\end{align*}

\begin{align*}
\text{refinement mapping} \\
\text{INSTANCE } AB & \triangleq \text{AB}2 \text{ with } AtoB \leftarrow \text{RemoveBad}(AtoB2), \\
& \quad BtoA \leftarrow \text{RemoveBad}(BtoA2) \\
\text{THEOREM } Spec & \Rightarrow AB!Spec
\end{align*}

And we then saw that in module \textit{AB2} we can write an equivalent assertion as this \texttt{INSTANCE} statement and theorem.

These substitutions are called a refinement mapping.

And we say that the \textit{AB2} protocol implements the \textit{AB} protocol under this refinement mapping.

[slide 174]
The $AB2$ protocol implements the $AB$ protocol under the refinement mapping

$$AtoB \leftarrow RemoveBad(AtoB2),$$
$$BtoA \leftarrow RemoveBad(BtoA2)$$

means

$$AB \triangleq INSTANCE \ AB \ WITH \ AtoB \leftarrow RemoveBad(AtoB2),$$
$$BtoA \leftarrow RemoveBad(BtoA2)$$

THEOREM $Spec \Rightarrow AB!Spec$

So that means
The AB2 protocol implements the AB protocol under the refinement mapping

\[
\begin{align*}
AtoB & \leftarrow \text{RemoveBad}(AtoB2), \\
BtoA & \leftarrow \text{RemoveBad}(BtoA2)
\end{align*}
\]

means

\[
\begin{align*}
AB & \triangleq \text{INSTANCE } AB \text{ WITH } AtoB \leftarrow \text{RemoveBad}(AtoB2), \\
& \quad BtoA \leftarrow \text{RemoveBad}(BtoA2)
\end{align*}
\]

THEOREM \( Spec \Rightarrow AB!Spec \)

So that means that this theorem is true.

And TLC can check the theorem by using a model with \( Spec \) as the behavior specification and \( AB \) bang \( Spec \) as the temporal property to be checked.
If $Spec_2$ does not contain all the variables of $Spec_1$, then $Spec_2$ can implement $Spec_1$ only under a refinement mapping that assigns expressions of $Spec_2$ to all the variables in $Spec_1$ that are not also in $Spec_2$. This is the usual case.

In general, if a specification $Spec_2$ does not contain all the variables of a specification $Spec_1$,
If $Spec_2$ does not contain all the variables of $Spec_1$, then $Spec_2$ can implement $Spec_1$ only under a refinement mapping.

In general, if a specification $Spec_2$ does not contain all the variables of a specification $Spec_1$, then $Spec_2$ can implement $Spec_1$ only under a refinement mapping.
If $Spec_2$ does not contain all the variables of $Spec_1$, then $Spec_2$ can implement $Spec_1$ only under a refinement mapping that assigns expressions of $Spec_2$ to all the variables in $Spec_1$ that are not also in $Spec_2$.

In general, if a specification $Spec_2$ does not contain all the variables of a specification $Spec_1$, then $Spec_2$ can implement $Spec_1$ only under a refinement mapping that assigns expressions of $Spec_2$ to all the variables in $Spec_1$ that are not also in $Spec_2$. 
If $Spec_2$ does not contain all the variables of $Spec_1$, then $Spec_2$ can implement $Spec_1$ only under a refinement mapping that assigns expressions of $Spec_2$ to all the variables in $Spec_1$ that are not also in $Spec_2$.

This is the usual case.
If $\text{Spec}_2$ does not contain all the variables of $\text{Spec}_1$, then $\text{Spec}_2$ can implement $\text{Spec}_1$ only under a refinement mapping that assigns expressions of $\text{Spec}_2$ to all the variables in $\text{Spec}_1$ that are not also in $\text{Spec}_2$.

This is the usual case.

Even if $\text{Spec}_1$ and $\text{Spec}_2$ have a variable $v$ in common,
If $Spec_2$ does not contain all the variables of $Spec_1$, then $Spec_2$ can implement $Spec_1$ only under a refinement mapping that assigns expressions of $Spec_2$ to all the variables in $Spec_1$ that are not also in $Spec_2$.

This is the usual case.

Even if $Spec_1$ and $Spec_2$ have a variable $v$ in common, the refinement mapping might substitute an expression of $Spec_2$ other than $v$ for the variable $v$ of $Spec_1$.
What does it mean for a program to implement a TLA+ specification $Spec$?

What does it mean for a program written in a programming language to implement a TLA+ specification $Spec$?
What does it mean for a program to implement a TLA\textsuperscript{+} specification $Spec$?

It means that we can, in principle, write a TLA\textsuperscript{+} specification $Spec_{Pgm}$ of the program,

It means that we can, in principle, write a TLA\textsuperscript{+} specification $Spec_{Pgm}$ of the program,
What does it mean for a program to implement a TLA⁺ specification $Spec$?

It means that we can, in principle, write a TLA⁺ specification $Spec_{Pgm}$ of the program, and $Spec_{Pgm}$ implements $Spec$ under a suitable refinement mapping.

It means that we can, in principle, write a TLA⁺ specification $Spec_{Pgm}$ of the program, and $Spec_{Pgm}$ implements $Spec$ under a suitable refinement mapping.
What does it mean for a program to implement a TLA+ specification $Spec$?

It means that we can, in principle, write a TLA+ specification $Spec_{Pgm}$ of the program, and $Spec_{Pgm}$ implements $Spec$ under a suitable refinement mapping.

We can’t do that in practice, because $Spec_{Pgm}$ would be much too long and complicated,
What does it mean for a program to implement a TLA\(^+\) specification \(Spec\) ?

It means that we can, in principle, write a TLA\(^+\) specification \(Spec_{Pgm}\) of the program, and \(Spec_{Pgm}\) implements \(Spec\) under a suitable refinement mapping.

We can’t do that in practice, but understanding refinement mappings can help prevent implementation errors.
Even if you can’t write the refinement mapping in TLA⁺, you should be able to explain informally how the spec’s variables are implemented by the program’s state. The informal refinement mapping explains what the program is doing. Writing it down, perhaps as comments in the code, can expose errors in the program.
Even if you can’t write the refinement mapping in TLA⁺, you should be able to explain informally how the spec’s variables are implemented by the program’s state.

[slide 189]
Even if you can’t write the refinement mapping in TLA⁺, you should be able to explain informally how the spec’s variables are implemented by the program’s state.

The informal refinement mapping explains what the program is doing.
Even if you can’t write the refinement mapping in TLA⁺, you should be able to explain informally how the spec’s variables are implemented by the program’s state.

The informal refinement mapping explains what the program is doing.

Writing it down can expose errors in the program.
IMAGINARY VARIABLES
We added imaginary variables $AtoBgood$ and $BtoAgood$ to the $AB2$ protocol spec to obtain $SpecP$.
We added imaginary variables $AtoBgood$ and $BtoAgood$ to the $AB2$ protocol spec to obtain $SpecP$.

We did that to write a desired liveness property.

We added the imaginary variables $AtoBgood$ and $BtoAgood$ to the $AB2$ protocol specification to obtain specification $SpecP$.

We did that in order to write a desired liveness property.
We added imaginary variables $A_{toBgood}$ and $B_{toAgood}$ to the $AB^2$ protocol spec to obtain $SpecP$.

We added imaginary variables $A_{toB}$ and $B_{toA}$ to the $AB^2$ protocol spec to obtain $SpecH$.

We added the imaginary variables $A_{toBgood}$ and $B_{toAgood}$ to the $AB^2$ protocol specification to obtain specification $SpecP$.

We did that in order to write a desired liveness property.

We added imaginary variables $A_{toB}$ and $B_{toA}$ to the $AB^2$ protocol specification to obtain specification $SpecH$.

[slide 195]
We added imaginary variables $AtoB_{good}$ and $BtoA_{good}$ to the $AB2$ protocol spec to obtain $SpecP$.

We added imaginary variables $AtoB$ and $BtoA$ to the $AB2$ protocol spec to obtain $SpecH$.

We did that to show $AB2$ implements $AB$.

We did that in order to show that the $AB2$ protocol’s safety spec implements the $AB$ protocol’s safety spec.
We added imaginary variables \( AtoBgood \) and \( BtoAgood \) to the \( AB2 \) protocol spec to obtain \( SpecP \).

We added imaginary variables \( AtoB \) and \( BtoA \) to the \( AB2 \) protocol spec to obtain \( SpecH \).

\( Spec2 \) obtained by adding imaginary variables to \( Spec1 \)

We did that in order to show that the \( AB2 \) protocol’s safety spec implements the \( AB \) protocol’s safety spec.

In general, a specification \( Spec2 \) is obtained by adding imaginary variables to a specification \( Spec1 \)
We added imaginary variables $A_{toB good}$ and $B_{toA good}$ to the $AB^2$ protocol spec to obtain $SpecP$.

We added imaginary variables $A_{toB}$ and $B_{toA}$ to the $AB^2$ protocol spec to obtain $SpecH$.

$Spec2$ obtained by adding imaginary variables to $Spec1$ means $Spec2$ and $Spec1$ allow the same behaviors.

We did that in order to show that the $AB^2$ protocol’s safety spec implements the $AB$ protocol’s safety spec.

In general, a specification $Spec2$ is obtained by adding imaginary variables to a specification $Spec1$ means that $Spec2$ and $Spec1$ allow the same behaviors.
We added imaginary variables \( Atobgood \) and \( Btogaood \) to the \( AB2 \) protocol spec to obtain \( SpecP \).

We added imaginary variables \( Atob \) and \( Btoga \) to the \( AB2 \) protocol spec to obtain \( SpecH \).

\( Spec2 \) obtained by adding imaginary variables to \( Spec1 \) means \( Spec2 \) and \( Spec1 \) allow the same behaviors if we ignore the values of the imaginary variables.

We did that in order to show that the \( AB2 \) protocol’s safety spec implements the \( AB \) protocol’s safety spec.

In general, a specification \( Spec2 \) is obtained by adding imaginary variables to a specification \( Spec1 \) means that \( Spec2 \) and \( Spec1 \) allow the same behaviors if we ignore the values of the imaginary variables.
We added imaginary variables $A_{to}B$ and $B_{to}A$ to the $AB2$ protocol spec to obtain $SpecH$.

We did that to show $AB2$ implements $AB$.

We added imaginary variables to show that the $AB2$ spec implements the $AB$ spec.
We added imaginary variables $A \to B$ and $B \to A$ to the $AB2$ protocol spec to obtain $SpecH$.

We did that to show $AB2$ implements $AB$.

This wasn’t necessary because we could use a refinement mapping instead.

We added imaginary variables to show that the $AB2$ spec implements the $AB$ spec.

This wasn’t necessary because we were able to use a refinement mapping instead.
We added imaginary variables $AtoB$ and $BtoA$ to the $AB2$ protocol spec to obtain $SpecH$.

We did that to show $AB2$ implements $AB$.

This wasn’t necessary because we could use a refinement mapping instead.

Sometimes we have to add imaginary variables to define a refinement mapping.

We added imaginary variables to show that the $AB2$ spec implements the $AB$ spec.

This wasn’t necessary because we were able to use a refinement mapping instead.

But sometimes we have to add imaginary variables in order to define a refinement mapping.
The $AB$ and $AB2$ protocols are essentially the same (ignoring liveness).

The $AB$ and $AB2$ protocols are essentially the same, if we ignore liveness.
The $AB$ and $AB2$ protocols are essentially the same (ignoring liveness).

**So** Spec of $AB$ should implement Spec of $AB2$ under a refinement mapping.
The $AB$ and $AB2$ protocols are essentially the same (ignoring liveness).

So $Spec$ of $AB$ should implement $Spec$ of $AB2$ under a refinement mapping.

Showing this requires adding to module $AB$.
The $AB$ and $AB2$ protocols are essentially the same (ignoring liveness).

So $Spec$ of $AB$ should implement $Spec$ of $AB2$ under a refinement mapping.

Showing this requires adding to module $AB$:

$$AB2 \triangleq \text{INSTANCE } AB2 \text{ WITH } AtoB2 \leftarrow \ldots, BtoA2 \leftarrow \ldots$$

Showing this requires adding to module $AB$:

An INSTANCE statement giving the refinement mapping

[slide 206]
The $AB$ and $AB2$ protocols are essentially the same (ignoring liveness).

So $Spec$ of $AB$ should implement $Spec$ of $AB2$ under a refinement mapping.

Showing this requires adding to module $AB$

$$AB2 \triangleq \text{INSTANCE } AB2 \text{ WITH } AtoB2 \leftarrow \ldots, BtoA2 \leftarrow \ldots$$

THEOREM $Spec \Rightarrow AB2!Spec$

Showing this requires adding to module $AB$:

An INSTANCE statement giving the refinement mapping and checking this theorem.

[slide 207]
The $AB$ and $AB2$ protocols are essentially the same (ignoring liveness).

So $Spec$ of $AB$ should implement $Spec$ of $AB2$ under a refinement mapping.

Showing this requires adding to module $AB$

$$AB2 \triangleq \text{INSTANCE } AB2 \text{ WITH } AtoB2 \leftarrow \ldots, \ BtoA2 \leftarrow \ldots$$

THEOREM $Spec \Rightarrow AB2!Spec$  \hspace{1cm} \text{expressions of module } AB

These expressions of the refinement mapping must be written in terms of the variables of module $AB$.  

[slide 208]
The $AB$ and $AB2$ protocols are essentially the same (ignoring liveness).

So $Spec$ of $AB$ should implement $Spec$ of $AB2$ under a refinement mapping.

Showing this requires adding to module $AB$

$$AB2 \triangleq \text{INSTANCE } AB2 \text{ WITH } AtoB2 \leftarrow \ldots, BtoA2 \leftarrow \ldots$$

THEOREM $Spec \Rightarrow AB2!Spec$

expressions of module $AB$

Impossible without adding imaginary variables to $Spec$ of $AB$ that remember where messages were lost from $AtoB$ and $BtoA$.

These expressions of the refinement mapping must be written in terms of the variables of module $AB$.

This is impossible without adding imaginary variables to specification $Spec$ of module $AB$ that remember where messages that were lost from the message sequences $AtoB$ and $BtoA$ used to be.
Imaginary Variables

Imaginary Variables

[slide 210]
Imaginary Variables

– Need not describe actual state of the system.

Imaginary Variables

Need not describe any actual state of the system.
Imaginary Variables

– Need not describe actual state of the system.

   If their values can be described in terms of the original variables, then they are unnecessary.
Imaginary Variables

– Need not describe actual state of the system.

If their values can be described in terms of the original variables, then they are unnecessary.

We didn’t need to add imaginary variables $AtoB$ and $BtoA$ to the $AB2$ protocol to show it implements the $AB$ protocol spec.

For example, we didn’t need to add imaginary variables $AtoB$ and $BtoA$ to the $AB2$ protocol spec in order to show that it implements the $AB$ protocol spec.
Imaginary Variables

– Need not describe actual state of the system.

If their values can be described in terms of the original variables, then they are unnecessary.

We didn’t need to add imaginary variables $AtoB$ and $BtoA$ to the $AB2$ protocol to show it implements the $AB$ protocol because we could specify their values with a refinement mapping.

For example, we didn’t need to add imaginary variables $AtoB$ and $BtoA$ to the $AB2$ protocol spec in order to show that it implements the $AB$ protocol spec because we could specify the values of those variables of the $AB$ spec with a refinement mapping.
Imaginary Variables

– Need not describe actual state of the system.

– Are not meant to be implemented.

For example, we didn’t need to add imaginary variables $AtoB$ and $BtoA$ to the $AB^2$ protocol spec in order to show that it implements the $AB$ protocol spec because we could specify the values of those variables of the $AB$ spec with a refinement mapping.

Imaginary variables are not meant to be implemented.
Imaginary Variables

– Need not describe actual state of the system.
– Are not meant to be implemented.
– May be needed to construct a refinement mapping.

And imaginary variables may be needed to construct a refinement mapping.
Imaginary Variables

– Need not describe actual state of the system.
– Are not meant to be implemented.
– May be needed to construct a refinement mapping.

And imaginary variables may be needed to construct a refinement mapping.

Imaginary variables
Auxiliary Variables

– Need not describe actual state of the system.
– Are not meant to be implemented.
– May be needed to construct a refinement mapping.

And imaginary variables may be needed to construct a refinement mapping. Imaginary variables are usually called auxiliary variables.
Auxiliary Variables

– Need not describe actual state of the system.
– Are not meant to be implemented.
– May be needed to construct a refinement mapping.

You can learn more about them by stopping the video and downloading the paper

*Auxiliary Variables in TLA*+
WHAT’S NEXT?
This is the last lecture of the TLA+ Video Course.

Writing good specs takes practice. Reading other people’s specs can help. I hope the TLA+ web pages will eventually contain many examples of realistic specs.
You’re now ready to write your own specs, including liveness conditions, and to show that one spec implements another.
This is the last lecture of the TLA+ Video Course.

You’re now ready to write your own specs, including liveness conditions, and to show that one spec implements another.

It may not be easy at first.
This is the last lecture of the TLA+ Video Course.

You’re now ready to write your own specs, including liveness conditions, and to show that one spec implements another.

It may not be easy at first.

Writing good specs takes practice.
This is the last lecture of the TLA+ Video Course.

You’re now ready to write your own specs, including liveness conditions, and to show that one spec implements another.

It may not be easy at first.

Writing good specs takes practice.
Reading other people’s specs can help.
This is the last lecture of the TLA+ Video Course.

You’re now ready to write your own specs, including liveness conditions, and to show that one spec implements another.

It may not be easy at first.

Writing good specs takes practice.
Reading other people’s specs can help.

I hope the TLA+ web pages will eventually contain many examples of realistic specs.
There is still plenty to learn about TLA\(^+\) and its tools:

- A few TLA\(^+\) features.
- Many Toolbox features.
- PlusCal

There is still plenty for you to learn about TLA\(^+\) and its tools:
There is still plenty to learn about TLA$^+$ and its tools:

- A few TLA$^+$ features.

There is still plenty for you to learn about TLA$^+$ and its tools:

There are a few TLA$^+$ features that you haven’t seen.
There is still plenty to learn about TLA$^+$ and its tools:

- A few TLA$^+$ features.

You can find out about them by browsing:

There is still plenty for you to learn about TLA$^+$ and its tools:

There are a few TLA$^+$ features that you haven’t seen.

You can find out about them by browsing the *Specifying Systems* book.
There is still plenty to learn about TLA\(^+\) and its tools:

- A few TLA\(^+\) features.
- Many Toolbox features.

There is still plenty for you to learn about TLA\(^+\) and its tools:

There are a few TLA\(^+\) features that you haven’t seen.

You can find out about them by browsing the *Specifying Systems* book.

There are many Toolbox features that I haven’t shown you.
There is still plenty to learn about TLA+ and its tools:

- A few TLA+ features.

- Many Toolbox features.
  You can find out about them by browsing the Toolbox’s help pages.

There is still plenty for you to learn about TLA+ and its tools:

There are a few TLA+ features that you haven’t seen.
You can find out about them by browsing the Specifying Systems book.

There are many Toolbox features that I haven’t shown you.
You can find them by browsing the Toolbox’s help pages.
There is still plenty to learn about TLA+ and its tools:

– A few TLA+ features.
– Many Toolbox features.
– PlusCal

And there’s the PlusCal algorithm language,
There is still plenty to learn about TLA\(^+\) and its tools:

- A few TLA\(^+\) features.
- Many Toolbox features.
- **PlusCal** A language for writing TLA\(^+\) specs that look more familiar to programmers.

And there’s the PlusCal algorithm language,

A language for writing TLA+ specs that look more familiar to programmers.
There is still plenty to learn about TLA+ and its tools:

- A few TLA+ features.
- Many Toolbox features.
- PlusCal A language for writing TLA+ specs that look more familiar to programmers.

See the TLA+ Web site for documentation.

And there’s the PlusCal algorithm language,
A language for writing TLA+ specs that look more familiar to programmers.

See the TLA+ Web site for documentation.
This is the end of the course. You’ve come a long way – perhaps further than you realize. As you go forward, remember to take the time to stop and think. I hope what you’ve learned here will help you do that.
End of Lecture 10, Part 2

IMPLEMENTATION

WITH REFINEMENT

REFINEMENT MAPPINGS