TLA$^+$ Video Course – Lecture 9, Part 2

Leslie Lamport

THE ALTERNATING BIT PROTOCOL

THE PROTOCOL

This video should be viewed in conjunction with a Web page.
To find that page, search the Web for *TLA+ Video Course*.

In this part, we examine the Alternating Bit Protocol itself, and how it implements the liveness property of its high-level specification.

In the process, we learn about strong fairness and some more about using the TLC model checker.

THE SAFETY SPECIFICATION

What the Protocol Accomplishes

Remember what the AB protocol is supposed to accomplish.

What the Protocol Accomplishes

A | **B**

$AVar$: $\langle$“ ”, 1$\rangle$ | $BVar$: $\langle$“ ”, 1$\rangle$

A Sends:

B Receives:

Remember what the AB protocol is supposed to accomplish.

It starts with $AVar$ and $BVar$ having values like these, where the first component is an arbitrary data item.

What the Protocol Accomplishes

A | B
$AVar$: $\langle$"$Fred$", 0$\rangle$ | $BVar$: $\langle$" ", 1$\rangle$

A Sends: "$Fred$"

B Receives:

Remember what the AB protocol is supposed to accomplish.

It starts with $AVar$ and $BVar$ having values like these, where the first component is an arbitrary data item.

A sends a data item by setting the first element of $AVar$ to that item and complementing the one-bit second element.

What the Protocol Accomplishes

A	B
$AVar$: $\langle$"$Fred$", 0$\rangle$	$BVar$: $\langle$"$Fred$", 0$\rangle$

A Sends: "$Fred$"

B Receives: "$Fred$"

B receives that item.

What the Protocol Accomplishes

A		B	
$AVar$:	$\langle$"$Mary$", 1$\rangle$	$BVar$:	$\langle$"$Fred$", 0$\rangle$

A Sends: "$Fred$", "$Mary$"

B Receives: "$Fred$"

B receives that item.

A sends the next data item.

What the Protocol Accomplishes

A | **B**

$AVar$: $\langle$"$Mary$", 1$\rangle$ $BVar$: $\langle$"$Mary$", 1$\rangle$

A Sends: "$Fred$", "$Mary$"

B Receives: "$Fred$", "$Mary$"

B receives that item.

A sends the next data item.

And so on.

What the Protocol Accomplishes

A	B
$AVar$: $\langle$"$Mary$", 0$\rangle$	$BVar$: $\langle$"$Mary$", 1$\rangle$

A Sends: "$Fred$", "$Mary$", "$Mary$"

B Receives: "$Fred$", "$Mary$"

B receives that item.

A sends the next data item.

And so on.

What the Protocol Accomplishes

A	B
$AVar$: $\langle$"$Mary$", 0$\rangle$	$BVar$: $\langle$"$Mary$", 0$\rangle$

A Sends: "$Fred$", "$Mary$", "$Mary$"

B Receives: "$Fred$", "$Mary$", "$Mary$"

B receives that item.

A sends the next data item.

And so on.

What the Protocol Accomplishes

A	B
$AVar$: $\langle$"$Mary$", 0$\rangle$	$BVar$: $\langle$"$Mary$", 0$\rangle$

A Sends: "$Fred$", "$Mary$", "$Mary$", ...

B Receives: "$Fred$", "$Mary$", "$Mary$", ...

B receives that item.

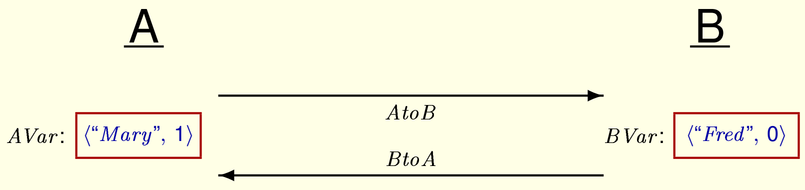
A sends the next data item.

And so on.

How the Protocol Works

Here’s how the protocol works.

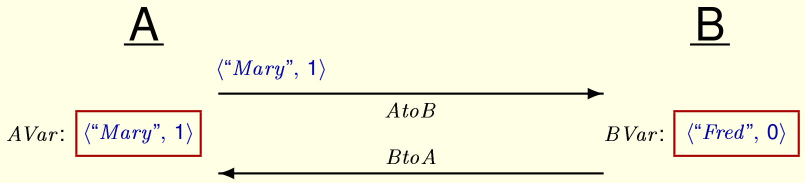
How the Protocol Works



Here's how the protocol works.

A and B communicate over two channels, one from A to B and one from B to A. The channels can lose messages.

How the Protocol Works

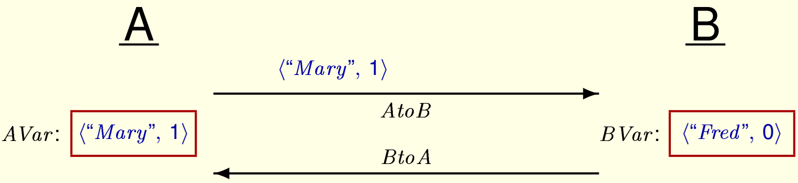


Here’s how the protocol works.

A and B communicate over two channels, one from A to B and one from B to A. The channels can lose messages.

A sends its current value to B.

How the Protocol Works

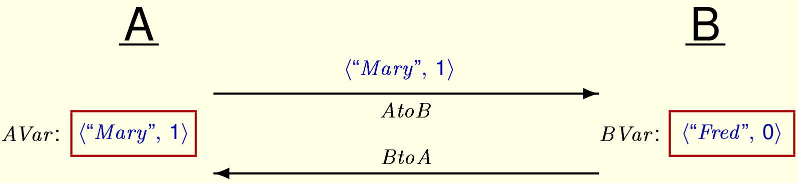


Here's how the protocol works.

A and B communicate over two channels, one from A to B and one from B to A. The channels can lose messages.

A sends its current value to B.

How the Protocol Works

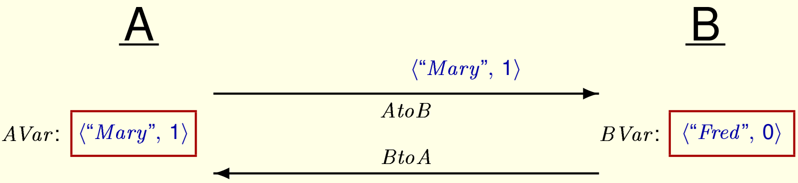


Here's how the protocol works.

A and B communicate over two channels, one from A to B and one from B to A. The channels can lose messages.

A sends its current value to B.

How the Protocol Works

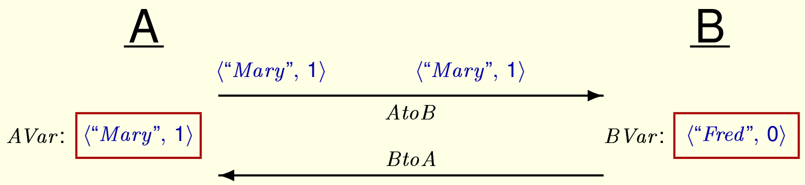


Here's how the protocol works.

A and B communicate over two channels, one from A to B and one from B to A. The channels can lose messages.

A sends its current value to B.

How the Protocol Works



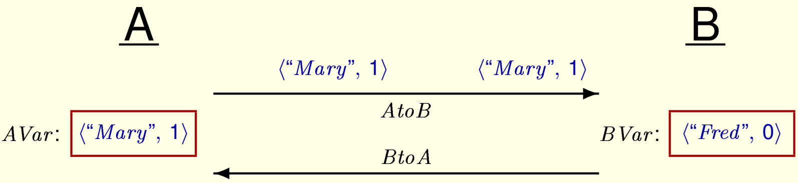
Here's how the protocol works.

A and B communicate over two channels, one from A to B and one from B to A. The channels can lose messages.

A sends its current value to B.
Since messages can be lost, A keeps sending its value

[slide 19]

How the Protocol Works

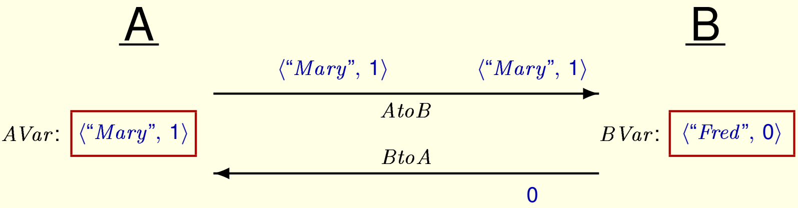


Here's how the protocol works.

A and B communicate over two channels, one from A to B and one from B to A. The channels can lose messages.

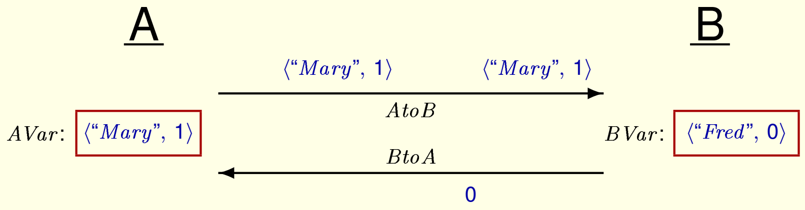
A sends its current value to B.
Since messages can be lost, A keeps sending its value

How the Protocol Works



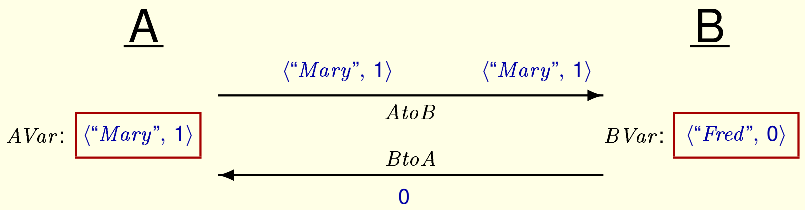
Meanwhile, B acknowledges the last value it received by sending its bit.

How the Protocol Works



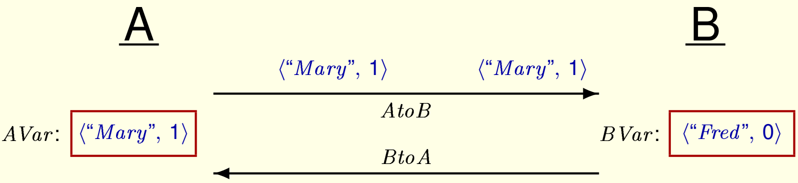
Meanwhile, B acknowledges the last value it received by sending its bit.

How the Protocol Works



Meanwhile, B acknowledges the last value it received by sending its bit.

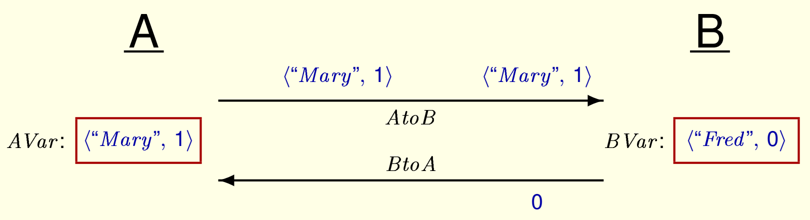
How the Protocol Works



Meanwhile, B acknowledges the last value it received by sending its bit.

And because the message might get lost,

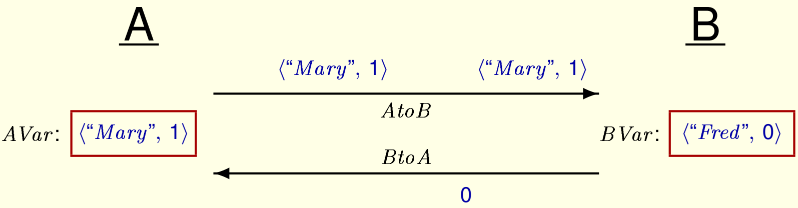
How the Protocol Works



Meanwhile, B acknowledges the last value it received by sending its bit.

And because the message might get lost,
B keeps sending it.

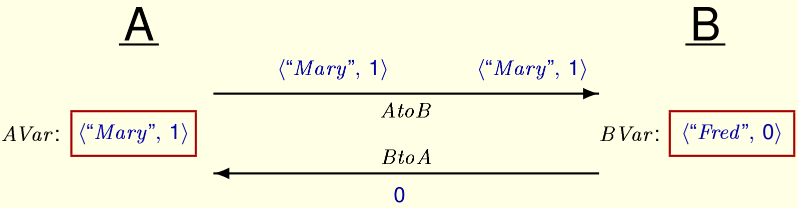
How the Protocol Works



Meanwhile, B acknowledges the last value it received by sending its bit.

And because the message might get lost,
B keeps sending it.

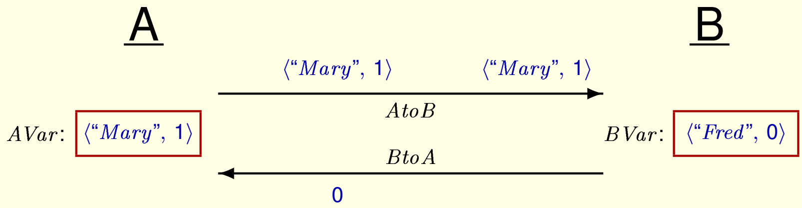
How the Protocol Works



Meanwhile, B acknowledges the last value it received by sending its bit.

And because the message might get lost,
B keeps sending it.

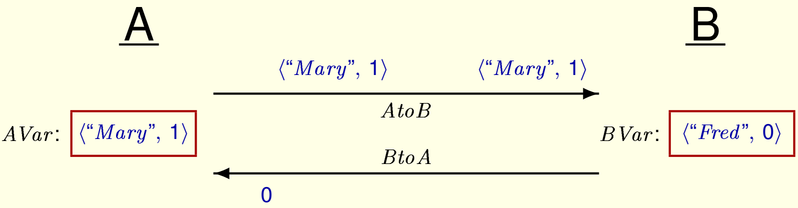
How the Protocol Works



Meanwhile, B acknowledges the last value it received by sending its bit.

And because the message might get lost,
B keeps sending it.

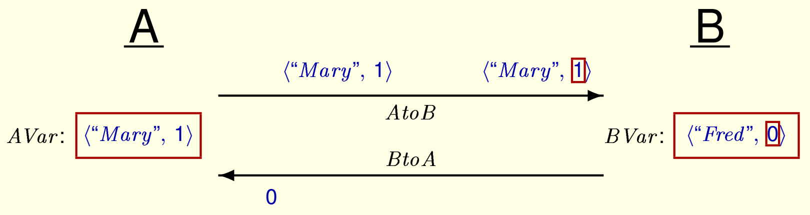
How the Protocol Works



Meanwhile, B acknowledges the last value it received by sending its bit.

And because the message might get lost,
B keeps sending it.

How the Protocol Works

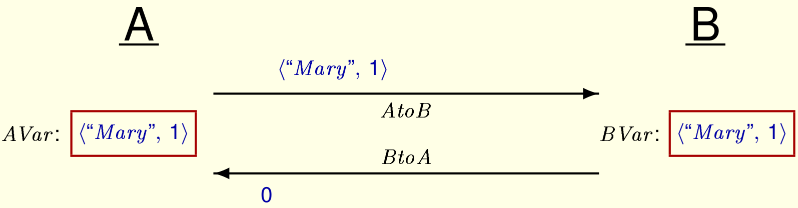


Meanwhile, B acknowledges the last value it received by sending its bit.

And because the message might get lost,
B keeps sending it.

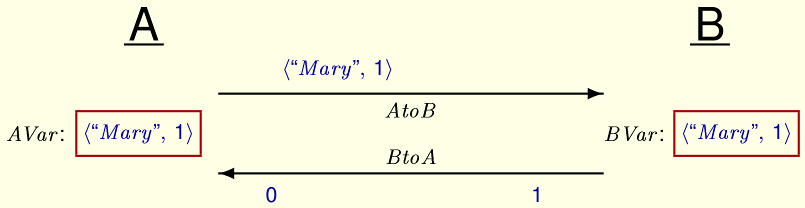
When B receives the next message on the channel A to B, it knows that this is a new value because the message's bit is different from its bit.

How the Protocol Works



So it changes $BVar$.

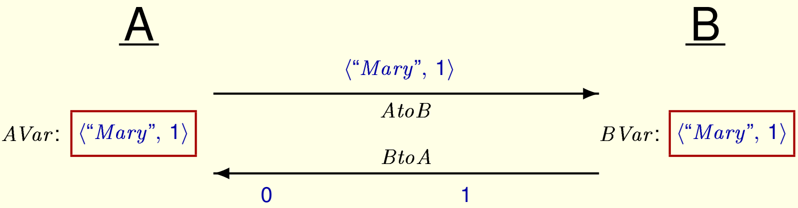
How the Protocol Works



So it changes $BVar$.

It then starts sending its new bit.

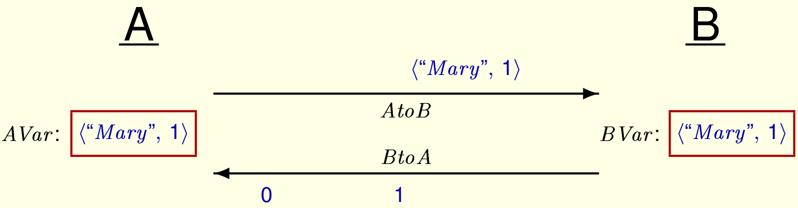
How the Protocol Works



So it changes $BVar$.

It then starts sending its new bit.

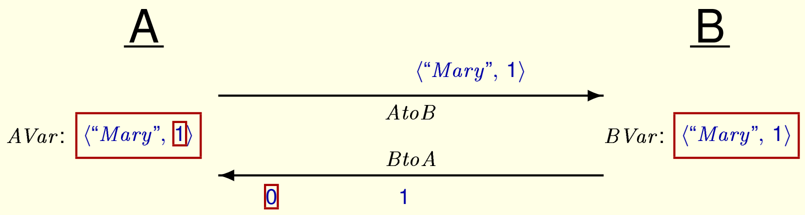
How the Protocol Works



So it changes $BVar$.

It then starts sending its new bit.

How the Protocol Works

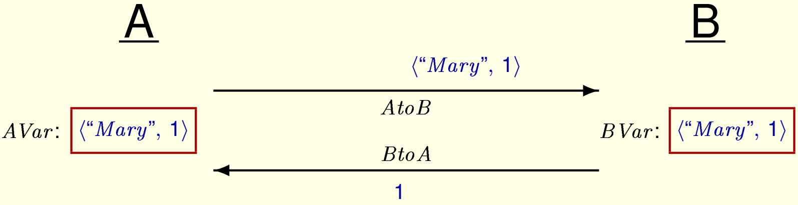


So it changes $BVar$.

It then starts sending its new bit.

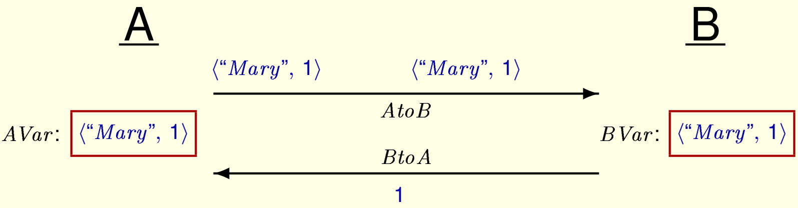
When A receives the next message on the channel B to A, it knows that this is an acknowledgement of its previous value because the message's bit is different from its bit.

How the Protocol Works



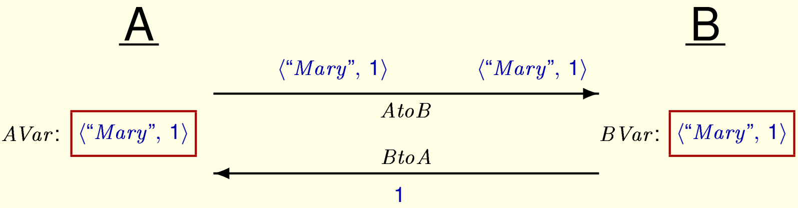
So A ignores the message

How the Protocol Works



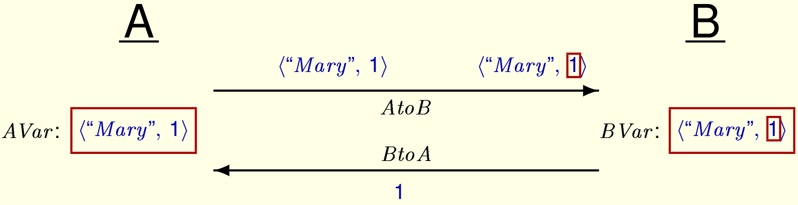
So A ignores the message and keeps sending its current value.

How the Protocol Works



So A ignores the message and keeps sending its current value.

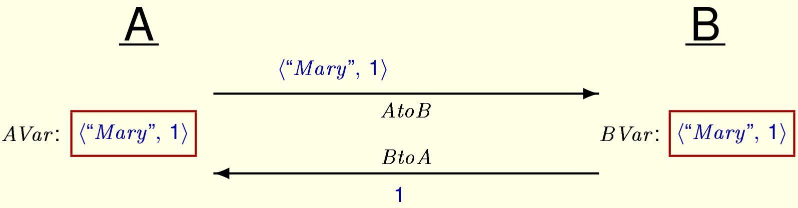
How the Protocol Works



So A ignores the message and keeps sending its current value.

Similarly, when B receives its next message on channel A to B, it knows this is a value it has already received because the message's bit is the same as its bit.

How the Protocol Works

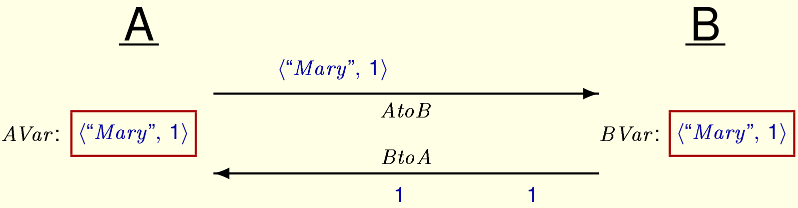


So A ignores the message and keeps sending its current value.

Similarly, when B receives its next message on channel A to B, it knows this is a value it has already received because the message's bit is the same as its bit.

So B ignores the message.

How the Protocol Works

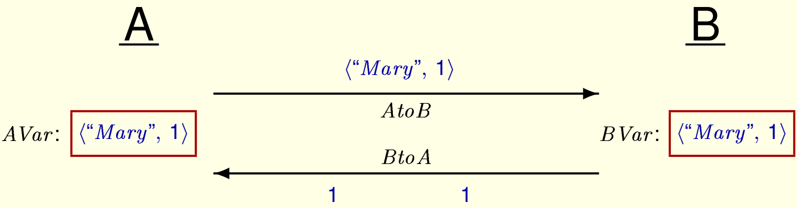


So A ignores the message and keeps sending its current value.

Similarly, when B receives its next message on channel A to B, it knows this is a value it has already received because the message's bit is the same as its bit.

So B ignores the message. and keeps sending its bit.

How the Protocol Works

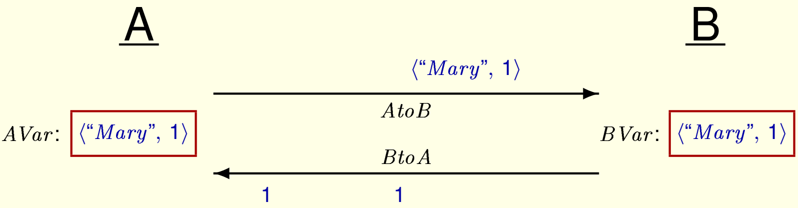


So A ignores the message and keeps sending its current value.

Similarly, when B receives its next message on channel A to B, it knows this is a value it has already received because the message's bit is the same as its bit.

So B ignores the message. and keeps sending its bit.

How the Protocol Works

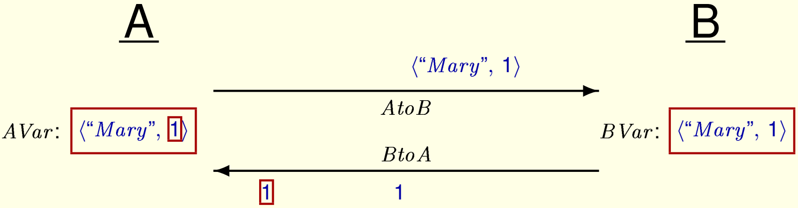


So A ignores the message and keeps sending its current value.

Similarly, when B receives its next message on channel A to B, it knows this is a value it has already received because the message's bit is the same as its bit.

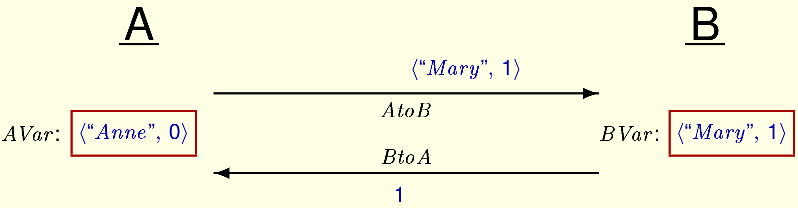
So B ignores the message. and keeps sending its bit.

How the Protocol Works



When A receives the next message on the channel B to A, it knows that this is an acknowledgement of its current value because the message's bit is the same as its bit.

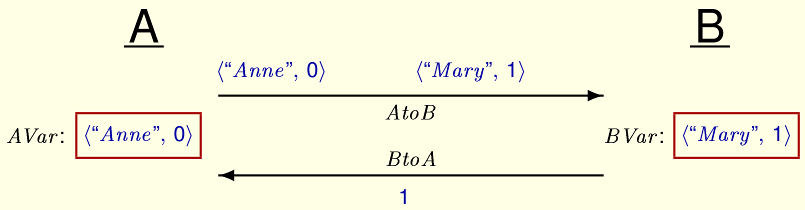
How the Protocol Works



When A receives the next message on the channel B to A, it knows that this is an acknowledgement of its current value because the message's bit is the same as its bit.

So A chooses a new data item and flips its bit.

How the Protocol Works

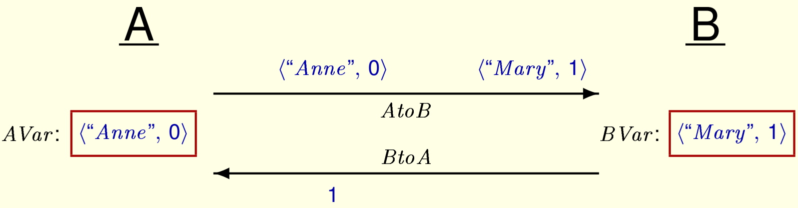


When A receives the next message on the channel B to A, it knows that this is an acknowledgement of its current value because the message's bit is the same as its bit.

So A chooses a new data item and flips its bit.

And so on.

How the Protocol Works

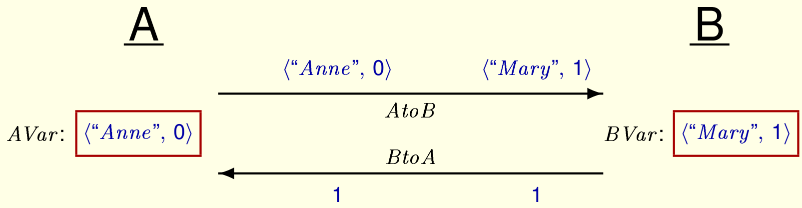


When A receives the next message on the channel B to A, it knows that this is an acknowledgement of its current value because the message's bit is the same as its bit.

So A chooses a new data item and flips its bit.

And so on.

How the Protocol Works

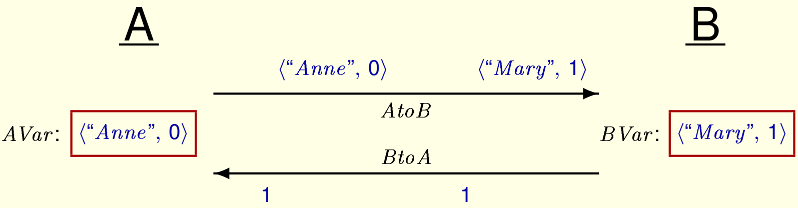


When A receives the next message on the channel B to A, it knows that this is an acknowledgement of its current value because the message's bit is the same as its bit.

So A chooses a new data item and flips its bit.

And so on.

How the Protocol Works



When A receives the next message on the channel B to A, it knows that this is an acknowledgement of its current value because the message's bit is the same as its bit.

So A chooses a new data item and flips its bit.

And so on.

The TLA+ Specification

We now look at the safety part of the TLA+ specification.

The TLA+ Specification

Download module AB and open it in the Toolbox.

We now look at the safety part of the TLA+ specification.

It's in module AB. Download that spec now and open it in the Toolbox.

The TLA+ Specification

Download module AB and open it in the Toolbox.

Nothing new except the use of operations on sequences.

We now look at the safety part of the TLA+ specification.

It's in module AB . Download that spec now and open it in the Toolbox.

There's nothing new in the safety spec except that it uses the operations on sequences we examined in part one of this lecture.

EXTENDS *Integers*, *Sequences*

As usual, the module begins with an EXTENDS statement that imports the Integers module

EXTENDS *Integers*, *Sequences*

Imports operators on sequences.

As usual, the module begins with an EXTENDS statement that imports the Integers module

and the Sequences module that defines the operators on sequences.

EXTENDS *Integers*, *Sequences*

CONSTANT *Data*

As usual, the module begins with an EXTENDS statement that imports the Integers module
and the Sequences module that defines the operators on sequences.

The constant *Data*

EXTENDS *Integers*, *Sequences*

CONSTANT *Data*   Same as in $ABSpec$.

As usual, the module begins with an EXTENDS statement that imports the Integers module
and the Sequences module that defines the operators on sequences.

The constant $Data$ is the same set of data items as in module $ABSpec$.

EXTENDS *Integers*, *Sequences*

CONSTANT *Data*

$Remove(i, seq) \triangleq$

As usual, the module begins with an EXTENDS statement that imports the Integers module
and the Sequences module that defines the operators on sequences.

The constant $Data$ is the same set of data items as in module $ABSpec$.

Remove of i, seek was defined in part 1 to equal

[slide 57]

EXTENDS *Integers*, *Sequences*

CONSTANT *Data*

$Remove(i, seq) \triangleq$ Sequence seq with its i^{th} element removed.

As usual, the module begins with an EXTENDS statement that imports the Integers module
and the Sequences module that defines the operators on sequences.

The constant $Data$ is the same set of data items as in module $ABSpec$.

Remove of i, seek was defined in part 1 to equal
sequence seq with its i^{th} element removed.

EXTENDS *Integers*, *Sequences*

CONSTANT *Data*

$$Remove(i, seq) \triangleq [j \in 1 \,..\, (Len(seq) - 1) \mapsto \text{IF } j < i \text{ THEN } seq[j] \text{ ELSE } seq[j+1]]$$

And this is the definition we saw before.

VARIABLES $AVar$, $BVar$

$AVar$ and $BVar$ are the same variables as in $ABSpec$,

VARIABLES $AVar$, $BVar$, $AtoB$, $BtoA$

$AVar$ and $BVar$ are the same variables as in $ABSpec$, while A *to* B and B *to* A are additional variables that represent the message channels.

VARIABLES $AVar$, $BVar$, $AtoB$, $BtoA$

$$vars \triangleq \langle AVar, BVar, AtoB, BtoA \rangle$$

$AVar$ and $BVar$ are the same variables as in $ABSpec$, while A *to* B and B *to* A are additional variables that represent the message channels.

As usual, we define $vars$ to be the tuple of all variables.

VARIABLES $AVar,\ BVar,\ AtoB,\ BtoA$

$$vars \triangleq \langle AVar,\ BVar,\ AtoB,\ BtoA \rangle$$

$$TypeOK \triangleq$$

$AVar$ and $BVar$ are the same variables as in $ABSpec$, while A *to* B and B *to* A are additional variables that represent the message channels.

As usual, we define $vars$ to be the tuple of all variables.

Next is the type-correctness invariant.

VARIABLES $AVar, BVar, AtoB, BtoA$

$$vars \triangleq \langle AVar, BVar, AtoB, BtoA \rangle$$

$$TypeOK \triangleq \land AVar \in Data \times \{0, 1\}$$
$$\land BVar \in Data \times \{0, 1\}$$

Same as in $ABSpec$.

$AVar$ and $BVar$ are the same variables as in $ABSpec$, while A *to* B and B *to* A are additional variables that represent the message channels.

As usual, we define $vars$ to be the tuple of all variables.

Next is the type-correctness invariant.

The possible values of $AVar$ and $BVar$ are the same as in $ABSpec$.

VARIABLES $AVar,\ BVar,\ AtoB,\ BtoA$

$$vars \triangleq \langle AVar,\ BVar,\ AtoB,\ BtoA \rangle$$

$$\begin{aligned} TypeOK \triangleq\ & \land AVar \in Data \times \{0, 1\} \\ & \land BVar \in Data \times \{0, 1\} \\ & \land AtoB \in Seq(Data \times \{0, 1\}) \end{aligned}$$

$AtoB$ is an element of

VARIABLES $AVar$, $BVar$, $AtoB$, $BtoA$

$$vars \triangleq \langle AVar, BVar, AtoB, BtoA \rangle$$

$$\begin{aligned} TypeOK \triangleq\ & \land AVar \in Data \times \{0, 1\} \\ & \land BVar \in Data \times \{0, 1\} \\ & \land AtoB \in \boxed{Seq}(Data \times \{0, 1\}) \end{aligned}$$

The set of sequences of

$AtoB$ is an element of the set of all sequences of

VARIABLES $AVar$, $BVar$, $AtoB$, $BtoA$

$$vars \triangleq \langle AVar, BVar, AtoB, BtoA \rangle$$

$$\begin{aligned} TypeOK \triangleq\ & \land AVar \in Data \times \{0, 1\} \\ & \land BVar \in Data \times \{0, 1\} \\ & \land AtoB \in Seq(\boxed{Data \times \{0, 1\}}) \end{aligned}$$

The set of sequences of values A can send.

$AtoB$ is an element of the set of all sequences of values that A can send.

VARIABLES $AVar$, $BVar$, $AtoB$, $BtoA$

$$vars \triangleq \langle AVar, BVar, AtoB, BtoA \rangle$$

$$TypeOK \triangleq \begin{array}{l} \land AVar \in Data \times \{0, 1\} \\ \land BVar \in Data \times \{0, 1\} \\ \land AtoB \in Seq(Data \times \{0, 1\}) \end{array}$$

A sends a message by appending it to the end of $AtoB$.

$AtoB$ is an element of the set of all sequences of values that A can send.

A sends a message by appending it to the end of $AtoB$.

VARIABLES $AVar, BVar, AtoB, BtoA$

$vars \triangleq \langle AVar, BVar, AtoB, BtoA \rangle$

$$TypeOK \triangleq \begin{array}{l} \land AVar \in Data \times \{0, 1\} \\ \land BVar \in Data \times \{0, 1\} \\ \land AtoB \in Seq(Data \times \{0, 1\}) \end{array}$$

A sends a message by appending it to the end of $AtoB$.

B receives the message at the head of $AtoB$.

$AtoB$ is an element of the set of all sequences of values that A can send.

A sends a message by appending it to the end of $AtoB$.

B receives the message at the head of $AtoB$.

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

$$vars \triangleq \langle AVar,\ BVar,\ AtoB,\ BtoA \rangle$$

$$\begin{aligned} TypeOK \triangleq\ & \land AVar \in Data \times \{0, 1\} \\ & \land BVar \in Data \times \{0, 1\} \\ & \land AtoB \in Seq(Data \times \{0, 1\}) \\ & \land BtoA \in Seq(\{0, 1\}) \end{aligned}$$

The set of sequences of bits

And similarly, the value of $BtoA$ is always a sequence of bits.

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

$$vars \triangleq \langle AVar,\ BVar,\ AtoB,\ BtoA \rangle$$

$$\begin{aligned} TypeOK \triangleq\ & \land AVar \in Data \times \{0, 1\} \\ & \land BVar \in Data \times \{0, 1\} \\ & \land AtoB \in Seq(Data \times \{0, 1\}) \\ & \land BtoA \in Seq(\{0, 1\}) \end{aligned}$$

$$\begin{aligned} Init \triangleq\ & \land AVar \in Data \times \{1\} \\ & \land BVar = AVar \end{aligned}$$

Same as in $ABSpec$

And similarly, the value of $BtoA$ is always a sequence of bits.

$AVar$ and $BVar$ have the same initial values as in $ABSpec$.

$\textsc{variables}\ AVar,\ BVar,\ AtoB,\ BtoA$

$$vars \triangleq \langle AVar,\ BVar,\ AtoB,\ BtoA \rangle$$

$$\begin{aligned} TypeOK \triangleq\ & \land AVar \in Data \times \{0, 1\} \\ & \land BVar \in Data \times \{0, 1\} \\ & \land AtoB \in Seq(Data \times \{0, 1\}) \\ & \land BtoA \in Seq(\{0, 1\}) \end{aligned}$$

$$\begin{aligned} Init \triangleq\ & \land AVar \in Data \times \{1\} \\ & \land BVar = AVar \\ & \land AtoB = \langle\rangle \\ & \land BtoA = \langle\rangle \end{aligned}$$

Channels are empty.

And similarly, the value of $BtoA$ is always a sequence of bits.

$AVar$ and $BVar$ have the same initial values as in $ABSpec$.

And the channels initially equal the empty sequence.

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

$$vars \triangleq \langle AVar,\ BVar,\ AtoB,\ BtoA \rangle$$

$$\begin{aligned} TypeOK \triangleq\ & \land AVar \in Data \times \{0, 1\} \\ & \land BVar \in Data \times \{0, 1\} \\ & \land AtoB \in Seq(Data \times \{0, 1\}) \\ & \land BtoA \in Seq(\{0, 1\}) \end{aligned}$$

$$\begin{aligned} Init \triangleq\ & \land AVar \in Data \times \{1\} \\ & \land BVar = AVar \\ & \land AtoB = \langle\rangle \\ & \land BtoA = \langle\rangle \end{aligned}$$

And similarly, the value of $BtoA$ is always a sequence of bits.

$AVar$ and $BVar$ have the same initial values as in $ABSpec$.

And the channels initially equal the empty sequence.

The subactions of $Next$

The next-state action is the disjunction of five subactions whose definitions come next.

The subactions of $Next$

$$ASnd \triangleq$$

The next-state action is the disjunction of five subactions whose definitions come next.

A-send is defined to be

The subactions of $Next$

$ASnd \triangleq$ A sends a message.

The next-state action is the disjunction of five subactions whose definitions come next.

A-send is defined to be the action of A sending a message.

The subactions of $Next$

$ASnd \triangleq$ A sends a message.

$ARcv \triangleq$

The next-state action is the disjunction of five subactions whose definitions come next.

A-send is defined to be the action of A sending a message.

A-receive is defined to be

The subactions of $Next$

$ASnd \triangleq$ A sends a message.

$ARcv \triangleq$ A receives a message.

The next-state action is the disjunction of five subactions whose definitions come next.

A-send is defined to be the action of A sending a message.

A-receive is defined to be the action of A receiving a message.

The subactions of $Next$

$ASnd \triangleq$ A sends a message.

$ARcv \triangleq$ A receives a message.

$BSnd \triangleq$

Similarly for *B-send*

The subactions of $Next$

$ASnd \triangleq$ A sends a message.

$ARcv \triangleq$ A receives a message.

$BSnd \triangleq$ B sends a message.

Similarly for *B-send*

The subactions of $Next$

$ASnd \triangleq$ A sends a message.

$ARcv \triangleq$ A receives a message.

$BSnd \triangleq$ B sends a message.

$BRcv \triangleq$

Similarly for *B-send* and *B-receive*.

The subactions of $Next$

$ASnd \triangleq$ A sends a message.

$ARcv \triangleq$ A receives a message.

$BSnd \triangleq$ B sends a message.

$BRcv \triangleq$ B receives a message.

Similarly for *B-send* and *B-receive*.

The subactions of $Next$

$ASnd \triangleq$ A sends a message.

$ARcv \triangleq$ A receives a message.

$BSnd \triangleq$ B sends a message.

$BRcv \triangleq$ B receives a message.

$LoseMsg \triangleq$

Similarly for *B-send* and *B-receive*.

And *Lose-Message* is the action

The subactions of $Next$

$ASnd \triangleq$ A sends a message.

$ARcv \triangleq$ A receives a message.

$BSnd \triangleq$ B sends a message.

$BRcv \triangleq$ B receives a message.

$LoseMsg \triangleq$ A message is lost.

Similarly for *B-send* and *B-receive*.

And *Lose-Message* is the action that describes losing a message.

$ASnd \triangleq$

The definition of *A-send* is simple.

$$ASnd \triangleq \land AtoB' = Append(AtoB, AVar)$$

The definition of *A-send* is simple.

It appends the value of $AVar$ to the end of the sequence A-to-B

$$
\begin{aligned}
ASnd \triangleq\ & \land\ AtoB' = Append(AtoB,\ AVar) \\
& \land\ \text{UNCHANGED}\ \langle AVar,\ BtoA,\ BVar \rangle
\end{aligned}
$$

The definition of *A-send* is simple.

It appends the value of $AVar$ to the end of the sequence A-to-B

And leaves all the other variables unchanged.

The action is always enabled.

$$ASnd \triangleq \land AtoB' = Append(AtoB,\ AVar)$$
$$\qquad \land \text{UNCHANGED } \langle AVar,\ BtoA,\ BVar \rangle$$

$$ARcv \triangleq$$

The definition of *A-send* is simple.

It appends the value of $AVar$ to the end of the sequence A-to-B

And leaves all the other variables unchanged.

The action is always enabled.

The action of A receiving a message from B

[slide 88]

$$ASnd \triangleq \land AtoB' = Append(AtoB, AVar)$$
$$\qquad \land \text{UNCHANGED } \langle AVar, BtoA, BVar \rangle$$

$$ARcv \triangleq \land BtoA \neq \langle\rangle$$

is enabled only when the sequence B-to-A of messages from B is not empty.

$$ASnd \triangleq \land AtoB' = Append(AtoB, AVar)$$
$$\land \text{UNCHANGED } \langle AVar, BtoA, BVar \rangle$$

$$ARcv \triangleq \land BtoA \neq \langle\rangle$$
$$\land \text{IF } Head(BtoA) = AVar[2]$$
$$\text{THEN}$$
$$\text{ELSE}$$

is enabled only when the sequence B-to-A of messages from B is not empty.

If the bit at the head of B-to-A equals $AVar$'s bit, so B is acknowledging $AVar$'s current value,

$$
\begin{aligned}
ASnd \triangleq\ & \land\ AtoB' = Append(AtoB,\ AVar) \\
& \land\ \text{UNCHANGED}\ \langle AVar,\ BtoA,\ BVar \rangle
\end{aligned}
$$

$$
\begin{aligned}
ARcv \triangleq\ & \land\ BtoA \neq \langle\rangle \\
& \land\ \text{IF}\ Head(BtoA) = AVar[2] \\
& \qquad \text{THEN}\ \exists\, d \in Data : \\
& \qquad\qquad \boxed{AVar'} = \langle d,\ 1 - AVar[2] \rangle \\
& \qquad \text{ELSE}
\end{aligned}
$$

is enabled only when the sequence B-to-A of messages from B is not empty.

If the bit at the head of B-to-A equals $AVar$'s bit, so B is acknowledging $AVar$'s current value,
then the new value of $AVar$ is set just like in the A action of $ABSpec$: to a pair

$$ASnd \triangleq \land AtoB' = Append(AtoB, AVar)$$
$$\land \text{UNCHANGED } \langle AVar, BtoA, BVar \rangle$$

$$ARcv \triangleq \land BtoA \neq \langle \rangle$$
$$\land \text{IF } Head(BtoA) = AVar[2]$$
$$\text{THEN } \exists d \in Data :$$
$$AVar' = \langle d, 1 - AVar[2] \rangle$$
$$\text{ELSE}$$

is enabled only when the sequence B-to-A of messages from B is not empty.

If the bit at the head of B-to-A equals $AVar$'s bit, so B is acknowledging $AVar$'s current value,
then the new value of $AVar$ is set just like in the A action of $ABSpec$: to a pair
whose first element is a non-deterministically chosen element of $Data$,

$$
\begin{aligned}
ASnd \triangleq\ & \land\ AtoB' = Append(AtoB,\ AVar) \\
& \land\ \text{UNCHANGED}\ \langle AVar,\ BtoA,\ BVar \rangle
\end{aligned}
$$

$$
\begin{aligned}
ARcv \triangleq\ & \land\ BtoA \neq \langle\rangle \\
& \land\ \text{IF}\ Head(BtoA) = AVar[2] \\
& \qquad \text{THEN}\ \exists\, d \in Data : \\
& \qquad\qquad AVar' = \langle d,\ \boxed{1 - AVar[2]} \rangle \\
& \qquad \text{ELSE}
\end{aligned}
$$

and whose second element is the complement of the current value of $AVar$'s bit.

$$
\begin{aligned}
ASnd \;\triangleq\; & \land\ AtoB' = Append(AtoB,\ AVar) \\
& \land\ \text{UNCHANGED}\ \langle AVar,\ BtoA,\ BVar \rangle
\end{aligned}
$$

$$
\begin{aligned}
ARcv \;\triangleq\; & \land\ BtoA \neq \langle\rangle \\
& \land\ \text{IF}\ Head(BtoA) = AVar[2] \\
& \qquad \text{THEN}\ \exists\, d \in Data : \\
& \qquad\qquad AVar' = \langle d,\ 1 - AVar[2] \rangle \\
& \qquad \text{ELSE}\ \ AVar' = AVar
\end{aligned}
$$

and whose second element is the complement of the current value of $AVar$'s bit.

Otherwise, $AVar$ is unchanged.

$$
\begin{aligned}
ASnd \triangleq\ & \land\ AtoB' = Append(AtoB,\ AVar) \\
& \land\ \text{UNCHANGED } \langle AVar,\ BtoA,\ BVar \rangle
\end{aligned}
$$

$$
\begin{aligned}
ARcv \triangleq\ & \land\ BtoA \neq \langle\rangle \\
& \land\ \text{IF } Head(BtoA) = AVar[2] \\
& \qquad \text{THEN } \exists\, d \in Data : \\
& \qquad\qquad AVar' = \langle d,\ 1 - AVar[2] \rangle \\
& \qquad \text{ELSE } AVar' = AVar \\
& \land\ BtoA' = Tail(BtoA)
\end{aligned}
$$

and whose second element is the complement of the current value of $AVar$'s bit.

Otherwise, $AVar$ is unchanged.

And the message A is receiving, which is at the head of the sequence B-to-A, is removed from B-to-A.

$$
\begin{aligned}
BSnd \;\triangleq\;\; & \land\; BtoA' = Append(BtoA,\; BVar[2]) \\
& \land\; \text{UNCHANGED}\; \langle AVar,\; BVar,\; AtoB \rangle
\end{aligned}
$$

$$
\begin{aligned}
BRcv \;\triangleq\;\; & \land\; AtoB \neq \langle\rangle \\
& \land\; \text{IF}\;\; Head(AtoB)[2] \neq BVar[2] \\
& \qquad \text{THEN}\;\; BVar' = Head(AtoB) \\
& \qquad \text{ELSE}\;\;\; BVar' = BVar \\
& \land\; AtoB' = Tail(AtoB) \\
& \land\; \text{UNCHANGED}\; \langle AVar,\; BtoA \rangle
\end{aligned}
$$

The definitions of $BSnd$ and $BRcv$ are similar; you can read them yourself.

$$LoseMsg \triangleq$$

Next comes the definition of *Lose Message*.

$LoseMsg \triangleq \land \lor$ Remove a message from $AtoB$.

$\lor$ Remove a message from $BtoA$.

$\land$ UNCHANGED $\langle AVar, BVar \rangle$

Next comes the definition of *Lose Message*.

It removes a message from $AtoB$ or $BtoA$ and leaves $AVar$ and $BVar$ unchanged.

$$LoseMsg \triangleq \land \lor \land \exists i \in 1 \,..\, Len(AtoB) :$$

$$\lor \text{ Remove a message from } BtoA.$$

$$\land \text{UNCHANGED } \langle AVar, BVar \rangle$$

Next comes the definition of *Lose Message*.

It removes a message from $AtoB$ or $BtoA$ and leaves $AVar$ and $BVar$ unchanged.

The formula that describes removing a message from $AtoB$ asserts that for some i between 1 and the length of the sequence $AtoB$

$$
\begin{aligned}
LoseMsg \triangleq\ & \land \lor \land \exists i \in 1 \,..\, Len(AtoB) : \\
& \qquad\qquad AtoB' = Remove(i,\ AtoB) \\
& \quad\ \lor \text{ Remove a message from } BtoA. \\
& \land \text{UNCHANGED } \langle AVar,\ BVar \rangle
\end{aligned}
$$

the new value of $AtoB$ is the sequence obtained by removing the i^{th} element from the current value of $AtoB$.

$$LoseMsg \triangleq \land \lor \land \exists i \in 1 \,..\, Len(AtoB) : \\ AtoB' = Remove(i, AtoB) \\ \land BtoA' = BtoA$$

$\lor$ Remove a message from $BtoA$.

$$\land \text{UNCHANGED } \langle AVar, BVar \rangle$$

the new value of $AtoB$ is the sequence obtained by removing the i^{th} element from the current value of $AtoB$.

And $BtoA$ is unchanged.

$$
\begin{array}{l}
LoseMsg \triangleq \land \lor \land \exists i \in 1 \,..\, Len(AtoB) : \\
\qquad\qquad\qquad\qquad AtoB' = Remove(i,\ AtoB) \\
\qquad\qquad\qquad\quad \land\ BtoA' = BtoA \\
\qquad\qquad\quad \lor\ \text{Remove a message from } BtoA\,. \\
\\
\qquad\qquad \land\ \text{UNCHANGED}\ \langle AVar,\ BVar \rangle
\end{array}
$$

the new value of $AtoB$ is the sequence obtained by removing the i^{th} element from the current value of $AtoB$.

And $BtoA$ is unchanged.

The formula that describes removing a message from $BtoA$

$$
\begin{aligned}
LoseMsg \triangleq\ & \land \lor \land \exists i \in 1 \,..\, Len(AtoB) : \\
& \qquad\qquad AtoB' = Remove(i, AtoB) \\
& \qquad \land BtoA' = BtoA \\
& \quad \lor \land \exists i \in 1 \,..\, Len(BtoA) : \\
& \qquad\qquad BtoA' = Remove(i, BtoA) \\
& \qquad \land AtoB' = AtoB \\
& \land \text{UNCHANGED } \langle AVar, BVar \rangle
\end{aligned}
$$

the new value of $AtoB$ is the sequence obtained by removing the i^{th} element from the current value of $AtoB$.

And $BtoA$ is unchanged.

The formula that describes removing a message from $BtoA$ is similar.

$$Next \triangleq ASnd \lor ARcv \lor BSnd \lor BRcv \lor LoseMsg$$

Then comes the definition of $Next$

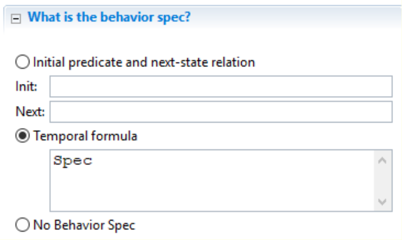
$$Next \triangleq ASnd \lor ARcv \lor BSnd \lor BRcv \lor LoseMsg$$

$$Spec \triangleq Init \land \Box[Next]_{vars}$$

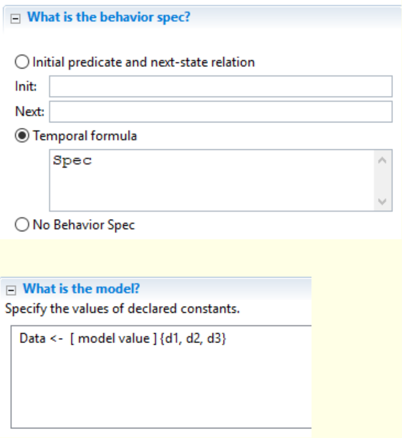
Then comes the definition of $Next$

and the standard safety specification.

CHECKING SAFETY

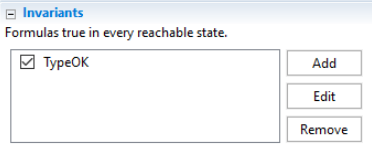


Create a new model with the default specification $Spec$,

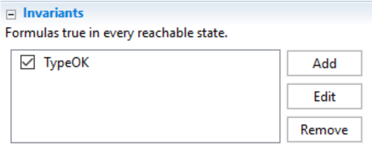


Create a new model with the default specification $Spec$,

letting $Data$ be a small set of model values.



Have TLC check that $TypeOK$ is an invariant.



But don’t run TLC yet.

Have TLC check that $TypeOK$ is an invariant.

But don’t run it yet.

A and B can keep sending messages faster than they get lost or received.

A and B can keep sending messages faster than they get lost or received.

A and B can keep sending messages faster than they get lost or received.

There is no limit to how long the sequences $AtoB$ and $BtoA$ can be.

A and B can keep sending messages faster than they get lost or received.

So there's no limit to how long the sequences $AtoB$ and $BtoA$ can be.

A and B can keep sending messages faster than they get lost or received.

There is no limit to how long the sequences $AtoB$ and $BtoA$ can be.

There are infinitely many reachable states

A and B can keep sending messages faster than they get lost or received.

So there's no limit to how long the sequences $AtoB$ and $BtoA$ can be.

The specification allows infinitely many reachable states, and since TLC tries to compute all reachable states,

A and B can keep sending messages faster than they get lost or received.

There is no limit to how long the sequences $AtoB$ and $BtoA$ can be.

There are infinitely many reachable states, so TLC will run forever.

A and B can keep sending messages faster than they get lost or received.

So there's no limit to how long the sequences $AtoB$ and $BtoA$ can be.

The specification allows infinitely many reachable states, and since TLC tries to compute all reachable states, it will run forever.

A and B can keep sending messages faster than they get lost or received.

There is no limit to how long the sequences $AtoB$ and $BtoA$ can be.

There are infinitely many reachable states, so TLC will run forever.

We could change the spec to limit the lengths of $AtoB$ and $BtoA$

We could change the spec to limit the lengths of $AtoB$ and $BtoA$,

A and B can keep sending messages faster than they get lost or received.

There is no limit to how long the sequences $AtoB$ and $BtoA$ can be.

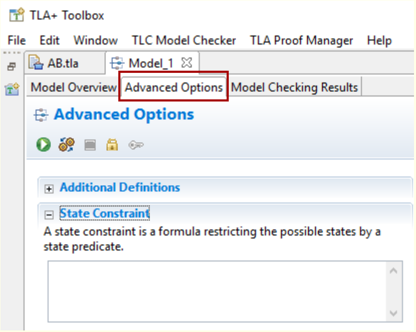
There are infinitely many reachable states, so TLC will run forever.

We could change the spec to limit the lengths of $AtoB$ and $BtoA$, but we shouldn't have to change the specification to model check it.

We could change the spec to limit the lengths of $AtoB$ and $BtoA$, but we shouldn't have to change the spec to model check it.

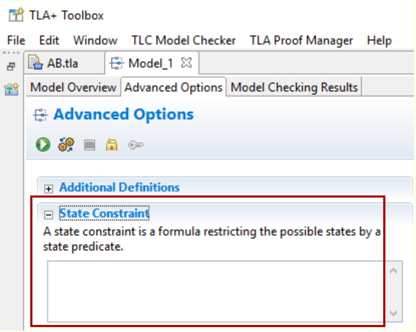
We can tell TLC to examine only states where $AtoB$ and $BtoA$ are not too long.

Here's how we can tell TLC to examine only states in which $AtoB$ and $BtoA$ aren't too long.



Here's how we can tell TLC to examine only states in which $AtoB$ and $BtoA$ aren't too long.

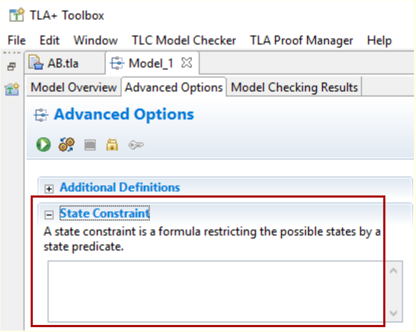
On the model's advanced options page,



Here's how we can tell TLC to examine only states in which $AtoB$ and $BtoA$ aren't too long.

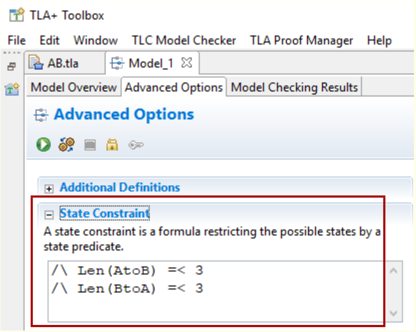
On the model's advanced options page, go to the *state constraint* section.

[slide 119]



Tell TLC to examine only states with $Len(AtoB)$ and $Len(BtoA)$ at most 3.

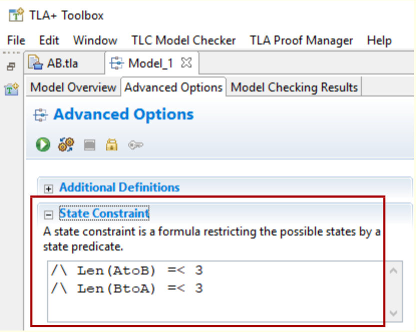
For example, you can tell TLC to examine only states in which the lengths of $AtoB$ and $BtoA$ are at most 3,



Tell TLC to examine only states with $Len(AtoB)$ and $Len(BtoA)$ at most 3.

For example, you can tell TLC to examine only states in which the lengths of $AtoB$ and $BtoA$ are at most 3,

by entering this state formula.



Tell TLC to examine only states with $Len(AtoB)$ and $Len(BtoA)$ at most 3.

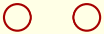
For example, you can tell TLC to examine only states in which the lengths of $AtoB$ and $BtoA$ are at most 3,
by entering this state formula.

To understand exactly what this does

How TLC Computes Reachable States

you need to understand how TLC computes reachable states when it has no state constraint.

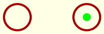
How TLC Computes Reachable States



you need to understand how TLC computes reachable states when it has no state constraint.

Starting from the set of initial states.

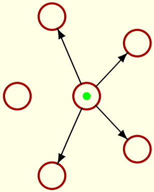
How TLC Computes Reachable States



you need to understand how TLC computes reachable states when it has no state constraint.

Starting from the set of initial states. It chooses one.

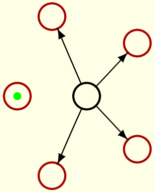
How TLC Computes Reachable States



you need to understand how TLC computes reachable states when it has no state constraint.

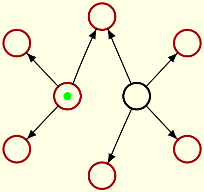
Starting from the set of initial states. It chooses one. and computes all possible next states from that state.

How TLC Computes Reachable States



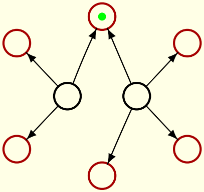
It then chooses another state to explore.

How TLC Computes Reachable States



It then chooses another state to explore. and finds all possible next states from it.

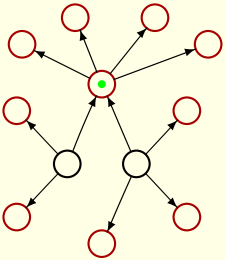
How TLC Computes Reachable States



It then chooses another state to explore. and finds all possible next states from it.

It then chooses another unexplored state

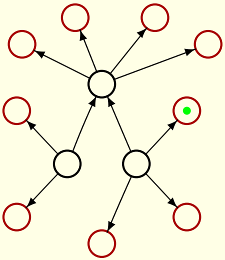
How TLC Computes Reachable States



It then chooses another state to explore. and finds all possible next states from it.

It then chooses another unexplored state and finds its next states.

How TLC Computes Reachable States

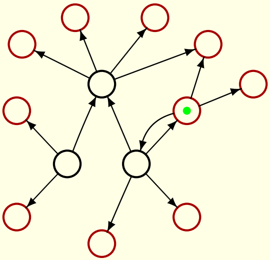


It then chooses another state to explore. and finds all possible next states from it.

It then chooses another unexplored state and finds its next states.

And it keeps on doing this.

How TLC Computes Reachable States

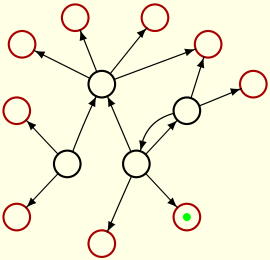


It then chooses another state to explore. and finds all possible next states from it.

It then chooses another unexplored state and finds its next states.

And it keeps on doing this.

How TLC Computes Reachable States

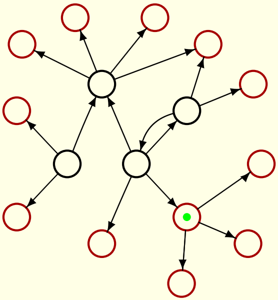


It then chooses another state to explore. and finds all possible next states from it.

It then chooses another unexplored state and finds its next states.

And it keeps on doing this.

How TLC Computes Reachable States

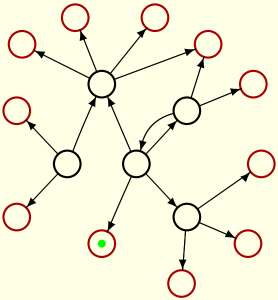


It then chooses another state to explore. and finds all possible next states from it.

It then chooses another unexplored state and finds its next states.

And it keeps on doing this.

How TLC Computes Reachable States

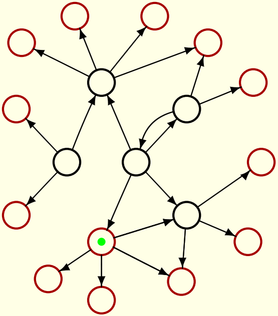


It then chooses another state to explore. and finds all possible next states from it.

It then chooses another unexplored state and finds its next states.

And it keeps on doing this.

How TLC Computes Reachable States

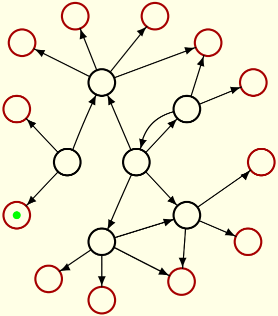


It then chooses another state to explore. and finds all possible next states from it.

It then chooses another unexplored state and finds its next states.

And it keeps on doing this.

How TLC Computes Reachable States



It then chooses another state to explore. and finds all possible next states from it.

It then chooses another unexplored state and finds its next states.

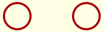
And it keeps on doing this.

And so on, until it has explored all reachable states.

How TLC Uses a Constraint

Now here's how TLC computes reachable states when it *has* a state constraint.

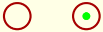
How TLC Uses a Constraint



Now here's how TLC computes reachable states when it *has* a state constraint.

Starting from the set of initial states.

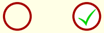
How TLC Uses a Constraint



Now here's how TLC computes reachable states when it *has* a state constraint.

Starting from the set of initial states. It chooses one and then checks if the state satisfies the constraint.

How TLC Uses a Constraint

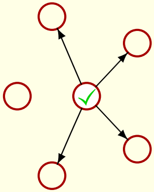


Now here's how TLC computes reachable states when it *has* a state constraint.

Starting from the set of initial states. It chooses one and then checks if the state satisfies the constraint.

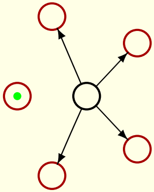
Let's suppose it does.

How TLC Uses a Constraint



As before, TLC then computes all possible next states from that state

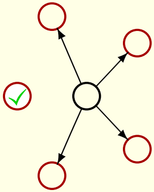
How TLC Uses a Constraint



As before, TLC then computes all possible next states from that state

and chooses another state to explore. It checks if *that* state satisfies the constraint

How TLC Uses a Constraint

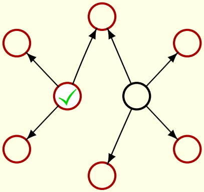


As before, TLC then computes all possible next states from that state

and chooses another state to explore. It checks if *that* state satisfies the constraint. Again, let's suppose it does.

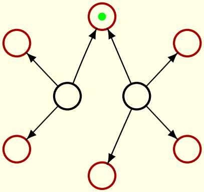
[slide 144]

How TLC Uses a Constraint



TLC then finds all possible next states from it.

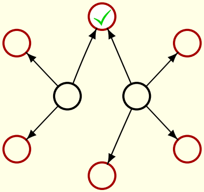
How TLC Uses a Constraint



TLC then finds all possible next states from it.

It keeps going like this

How TLC Uses a Constraint

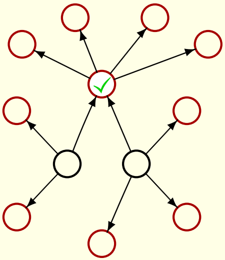


TLC then finds all possible next states from it.

It keeps going like this

As long as it finds states that satisfy the constraint.

How TLC Uses a Constraint

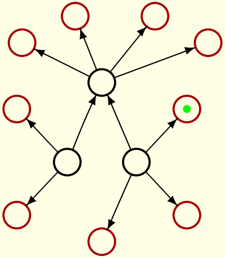


TLC then finds all possible next states from it.

It keeps going like this

As long as it finds states that satisfy the constraint.

How TLC Uses a Constraint

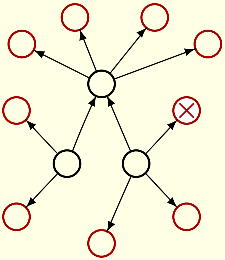


TLC then finds all possible next states from it.

It keeps going like this

As long as it finds states that satisfy the constraint.

How TLC Uses a Constraint



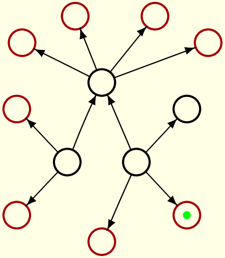
TLC then finds all possible next states from it.

It keeps going like this

As long as it finds states that satisfy the constraint.

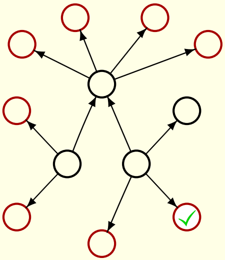
Suppose it now finds a state that doesn't satisfy the constraint.

How TLC Uses a Constraint



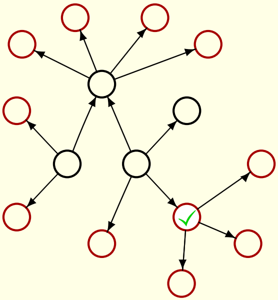
It doesn't explore further from that state and instead just goes on to the next unexplored state,

How TLC Uses a Constraint



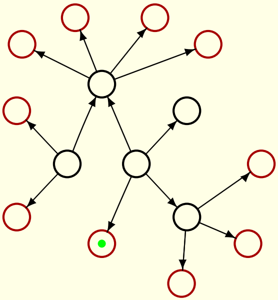
It doesn't explore further from that state and instead just goes on to the next unexplored state, exploring that state if it satisfies the constraint.

How TLC Uses a Constraint



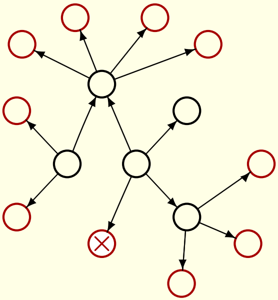
It doesn't explore further from that state and instead just goes on to the next unexplored state, exploring that state if it satisfies the constraint.

How TLC Uses a Constraint



It doesn’t explore further from that state and instead just goes on to the next unexplored state, exploring that state if it satisfies the constraint.

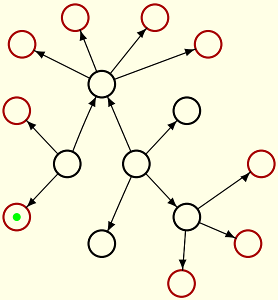
How TLC Uses a Constraint



It doesn't explore further from that state and instead just goes on to the next unexplored state, exploring that state if it satisfies the constraint.

And continuing like that, exploring only states that satisfy the constraint,

How TLC Uses a Constraint



It doesn't explore further from that state and instead just goes on to the next unexplored state, exploring that state if it satisfies the constraint.

And continuing like that, exploring only states that satisfy the constraint, until it finds no more states to explore.

You can now run TLC on your model.

You can now run TLC on your model.

The AB protocol should implement its high-level specification,

You can now run TLC on your model.

The alternating bit protocol should implement its high-level specification,

The AB protocol should implement its high-level specification, so formula $Spec$ of module AB should imply formula $Spec$ of module $ABSpec$.

You can now run TLC on your model.

The alternating bit protocol should implement its high-level specification, which means that formula $Spec$ of module AB should imply formula $Spec$ of module $ABSpec$.

The AB protocol should implement its high-level specification, so formula $Spec$ of module AB should imply formula $Spec$ of module $ABSpec$.

This should be a theorem of module AB,

You can now run TLC on your model.

The alternating bit protocol should implement its high-level specification, which means that formula $Spec$ of module AB should imply formula $Spec$ of module $ABSpec$.

This should be a theorem of module AB that TLC can check,

The AB protocol should implement its high-level specification, so formula $Spec$ of module AB should imply formula $Spec$ of module $ABSpec$.

This should be a theorem of module AB, but how can we write it?

You can now run TLC on your model.

The alternating bit protocol should implement its high-level specification, which means that formula $Spec$ of module AB should imply formula $Spec$ of module $ABSpec$.

This should be a theorem of module AB that TLC can check, but how can we write it?

The AB protocol should implement its high-level specification, so formula $Spec$ of module AB should imply formula $Spec$ of module $ABSpec$.

This should be a theorem of module AB, but how can we write it?

INSTANCE $ABSpec$

is illegal in module AB because it imports definitions of $Spec$, ... , which are already defined in AB.

The statement "INSTANCE $ABSpec$" is illegal in module AB because it imports definitions of identifiers like $Spec$, which are already defined in AB.

$ABS \triangleq$ INSTANCE $ABSpec$

Module AB contains the statement: A-B-S *is defined to equal* this instantiation.

$ABS \triangleq$ INSTANCE $ABSpec$

Imports definitions of $Spec, \ldots$ from $ABSpec$

Module AB contains the statement: A-B-S *is defined to equal* this instantiation.

This statement imports into module AB all the definitions, such as that of $Spec$, from module $ABSpec$

$ABS \triangleq$ INSTANCE $ABSpec$

Imports definitions of $Spec, \ldots$ from $ABSpec$
renamed as $ABS!Spec, \ldots$.

Module AB contains the statement: A-B-S *is defined to equal* this instantiation.

This statement imports into module AB all the definitions, such as that of $Spec$, from module $ABSpec$ except renaming them by prefacing their names with A-B-S-bang.

$ABS \triangleq$ INSTANCE $ABSpec$

Imports definitions of $Spec, \ldots$ from $ABSpec$
renamed as $ABS!Spec, \ldots$.

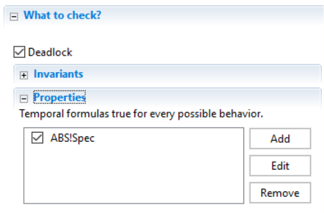
THEOREM $Spec \Rightarrow ABS!Spec$

This theorem states that the safety specification of the alternating bit protocol implements its high-level safety specification from module $ABSpec$.

$ABS \triangleq$ INSTANCE $ABSpec$

Imports definitions of $Spec, \ldots$ from $ABSpec$ renamed as $ABS!Spec, \ldots$.

THEOREM $Spec \Rightarrow ABS!Spec$



This theorem states that the safety specification of the alternating bit protocol implements its high-level safety specification from module $ABSpec$.

TLC will verify it by checking that specification $Spec$ satisfies the temporal property A-B-S bang spec .

LIVENESS

The complete AB protocol specification should be

The complete protocol specification should be

The complete AB protocol specification should be

$$FairSpec \triangleq Spec \land \text{fairness properties}$$

The complete protocol specification should be a formula we'll call $FairSpec$ that's the conjunction of the safety spec and one or more fairness properties.

$$FairSpec \triangleq Spec \land \text{fairness properties}$$

Should imply that messages keep getting sent and received.

The complete protocol specification should be a formula we'll call $FairSpec$ that's the conjunction of the safety spec and one or more fairness properties.

These fairness properties should imply that messages keep getting sent and received.

$FairSpec \triangleq Spec \land$ fairness properties

Should imply that messages
keep getting sent and received.

THEOREM $FairSpec \Rightarrow ABS!FairSpec$

Which means that this theorem should be true.

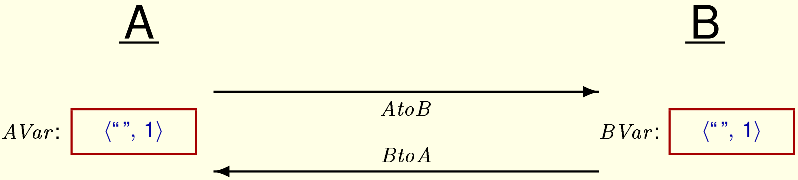
$$FairSpec \triangleq Spec \land \text{fairness properties}$$

Which means that this theorem should be true.

$$FairSpec \triangleq Spec \land \mathrm{WF}_{vars}(Next)$$

Weak fairness of the $Next$ action doesn't work.

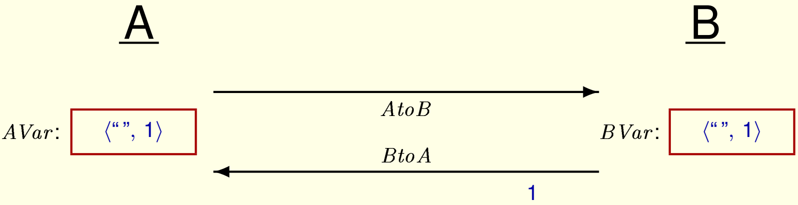
$$FairSpec \triangleq Spec \land \mathrm{WF}_{vars}(Next)$$



Weak fairness of the $Next$ action doesn't work.

For example, it allows a behavior in which

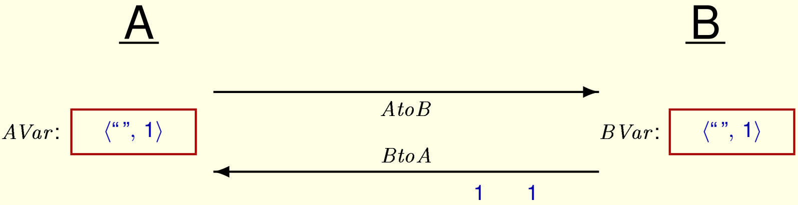
$$FairSpec \triangleq Spec \land \mathrm{WF}_{vars}(Next)$$



Weak fairness of the $Next$ action doesn't work.

For example, it allows a behavior in which B just keeps sending acknowledgments

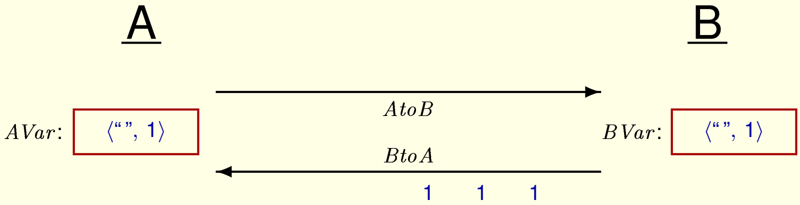
$$FairSpec \triangleq Spec \land \mathrm{WF}_{vars}(Next)$$



Weak fairness of the $Next$ action doesn't work.

For example, it allows a behavior in which B just keeps sending acknowledgments

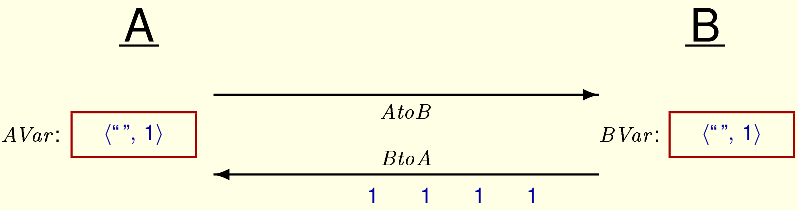
$$FairSpec \triangleq Spec \land \mathrm{WF}_{vars}(Next)$$



Weak fairness of the $Next$ action doesn't work.

For example, it allows a behavior in which B just keeps sending acknowledgments

$$FairSpec \triangleq Spec \land \mathrm{WF}_{vars}(Next)$$

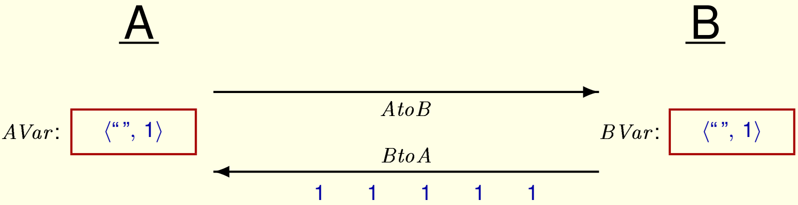


Weak fairness of the $Next$ action doesn’t work.

For example, it allows a behavior in which B just keeps sending acknowledgments

and nothing else ever happens.

$$FairSpec \triangleq Spec \land \mathrm{WF}_{vars}(Next)$$

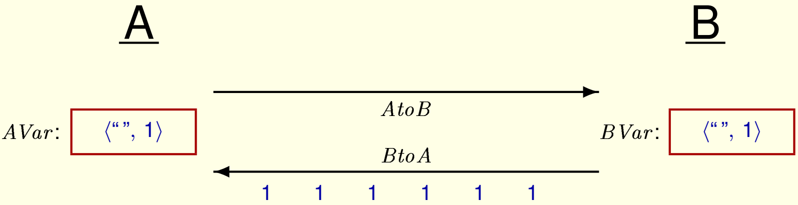


Weak fairness of the $Next$ action doesn't work.

For example, it allows a behavior in which B just keeps sending acknowledgments

and nothing else ever happens.

$$FairSpec \triangleq Spec \land \mathrm{WF}_{vars}(Next)$$



Weak fairness of the $Next$ action doesn't work.

For example, it allows a behavior in which B just keeps sending acknowledgments

and nothing else ever happens.

$FairSpec \triangleq Spec \land$ fairness properties

Weak fairness of the $Next$ action doesn't work.

For example, it allows a behavior in which B just keeps sending acknowledgments

and nothing else ever happens.

So we need a stronger fairness property.

$$FairSpec \triangleq Spec \land \text{fairness properties}$$

$$Next \triangleq ASnd \lor ARcv \lor BSnd \lor BRcv \lor LoseMsg$$

Remember the definition of the next-state action.

$FairSpec \triangleq Spec \land$ fairness properties

$Next \triangleq \boxed{ASnd} \lor \boxed{ARcv} \lor \boxed{BSnd} \lor \boxed{BRcv} \lor LoseMsg$

Remember the definition of the next-state action.

We need separate fairness requirements on these four subactions, to make sure that each of them keeps being executed.

$FairSpec \triangleq Spec \land$ fairness properties

$Next \triangleq ASnd \lor ARcv \lor BSnd \lor BRcv \lor \boxed{LoseMsg}$

Remember the definition of the next-state action.

We need separate fairness requirements on these four subactions, to make sure that each of them keeps being executed.

We don't want any fairness requirement on the Lose-Message action because we don't want to require that messages have to be lost.

$FairSpec \triangleq Spec \land$ fairness properties

$Next \triangleq \boxed{ASnd} \lor \boxed{ARcv} \lor \boxed{BSnd} \lor \boxed{BRcv} \lor LoseMsg$

Remember the definition of the next-state action.

We need separate fairness requirements on these four subactions, to make sure that each of them keeps being executed.

We don't want any fairness requirement on the Lose-Message action because we don't want to require that messages have to be lost.

So, let's try weak fairness of these actions.

$$FairSpec \triangleq Spec \land \mathrm{SF}_{vars}(ARcv) \land \mathrm{SF}_{vars}(BRcv) \land \mathrm{WF}_{vars}(ASnd) \land \mathrm{WF}_{vars}(BSnd)$$

Module AB contains this definition.

$$FairSpec \triangleq Spec \land \boxed{\mathrm{S}}\mathrm{F}_{vars}(ARcv) \land \boxed{\mathrm{S}}\mathrm{F}_{vars}(BRcv) \land$$
$$\mathrm{WF}_{vars}(ASnd) \land \mathrm{WF}_{vars}(BSnd)$$

Module AB contains this definition.

Change it by replacing these two ess-es by double-ewes.

$$FairSpec \triangleq Spec \land \mathrm{WF}_{vars}(ARcv) \land \mathrm{WF}_{vars}(BRcv) \land \mathrm{WF}_{vars}(ASnd) \land \mathrm{WF}_{vars}(BSnd)$$

Module AB contains this definition.

Change it by replacing these two ess-es by double-ewes.

This is a plausible specification, so

$$FairSpec \triangleq Spec \land \mathrm{WF}_{vars}(ARcv) \land \mathrm{WF}_{vars}(BRcv) \land \mathrm{WF}_{vars}(ASnd) \land \mathrm{WF}_{vars}(BSnd)$$

THEOREM $FairSpec \Rightarrow ABS!FairSpec$

Module AB contains this definition.

Change it by replacing these two ess-es by double-ewes.

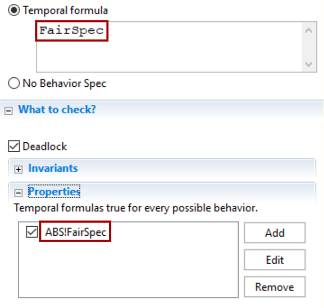
This is a plausible specification, so let's check if it satisfies this theorem.

Clone your model (removing any symmetry set).

Make a clone of the model you used before (removing any symmetry set).

Clone your model (removing any symmetry set).

Modify the specification and property to check.



Make a clone of the model you used before (removing any symmetry set).

In the clone, modify the specification and property to check by replacing $Spec$ with $FairSpec$.

Run TLC on the model.

Run TLC on the model.

Run TLC on the model.

It reports that the temporal property was violated

Run TLC on the model.

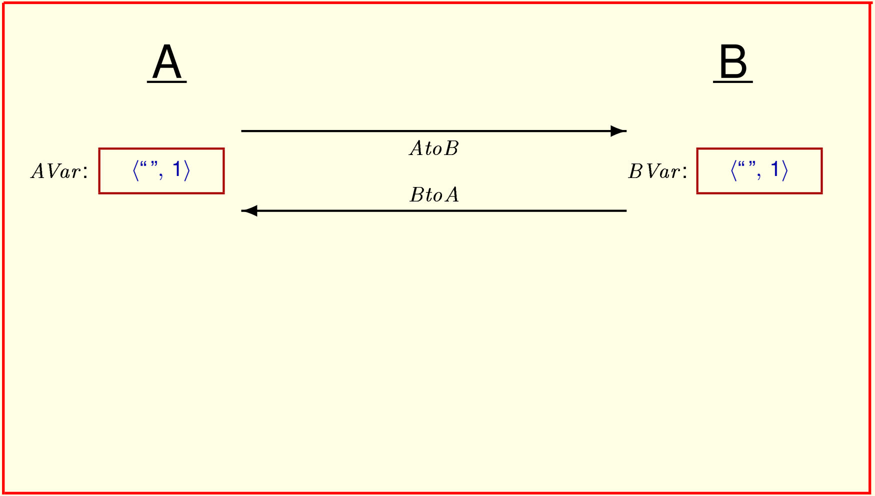
It reports that the temporal property was violated

Run TLC on the model.

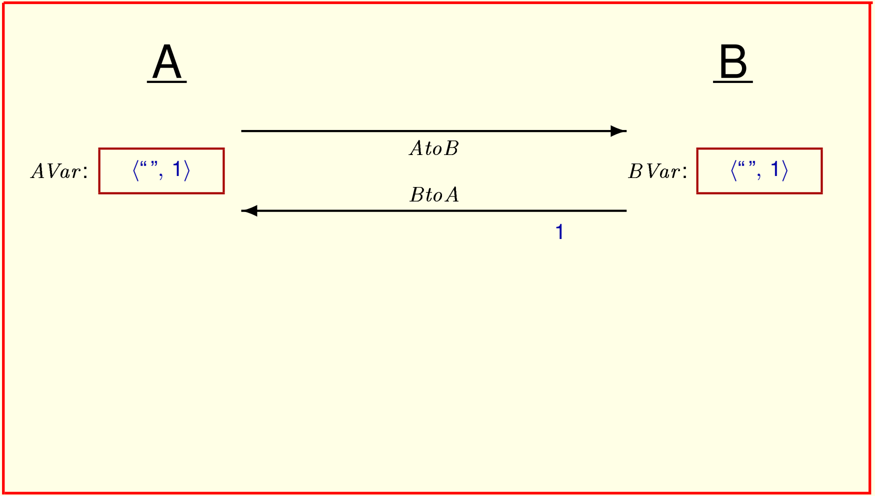
It reports that the temporal property was violated and produces a counterexample.

Run TLC on the model.

It reports that the temporal property was violated and produces a counterexample.

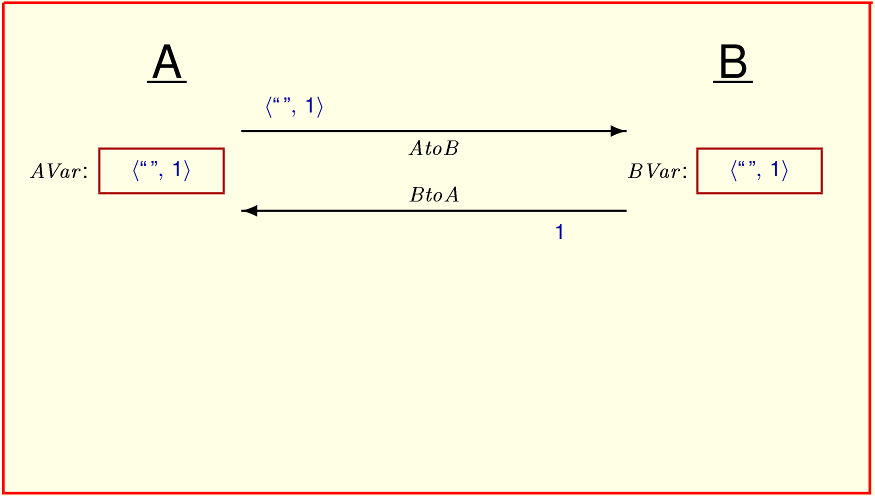


Here's the counterexample that TLC finds.



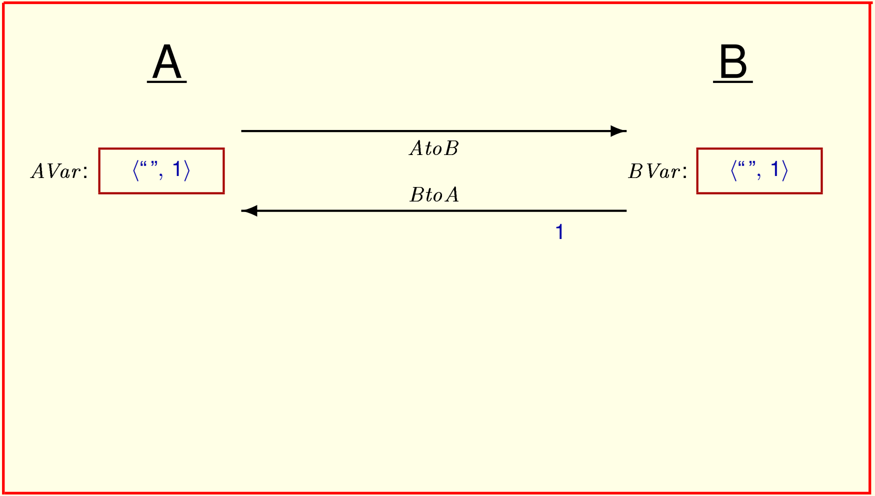
Here's the counterexample that TLC finds.

B sends an acknowledgment,



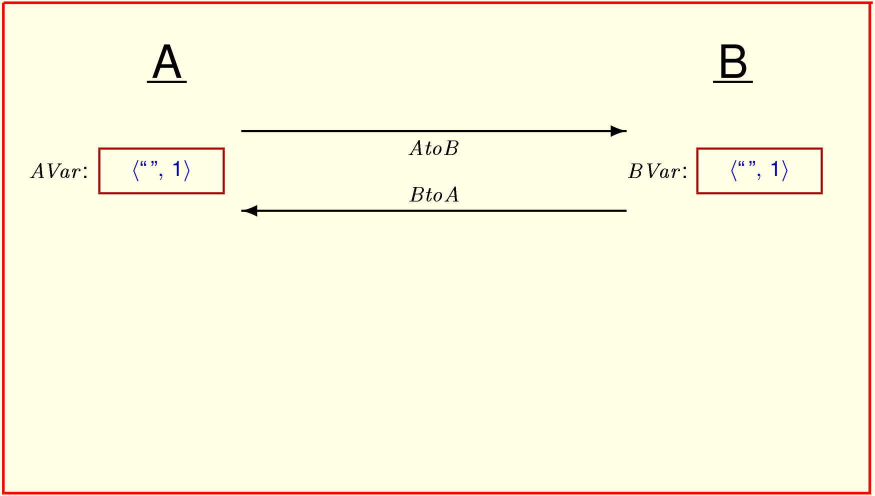
Here's the counterexample that TLC finds.

B sends an acknowledgment, A sends its value,



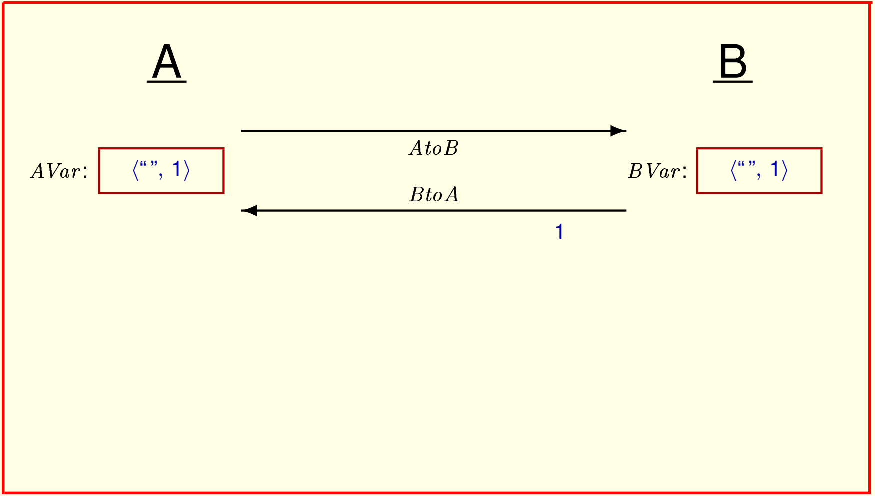
Here’s the counterexample that TLC finds.

B sends an acknowledgment, A sends its value, A’s message is lost,



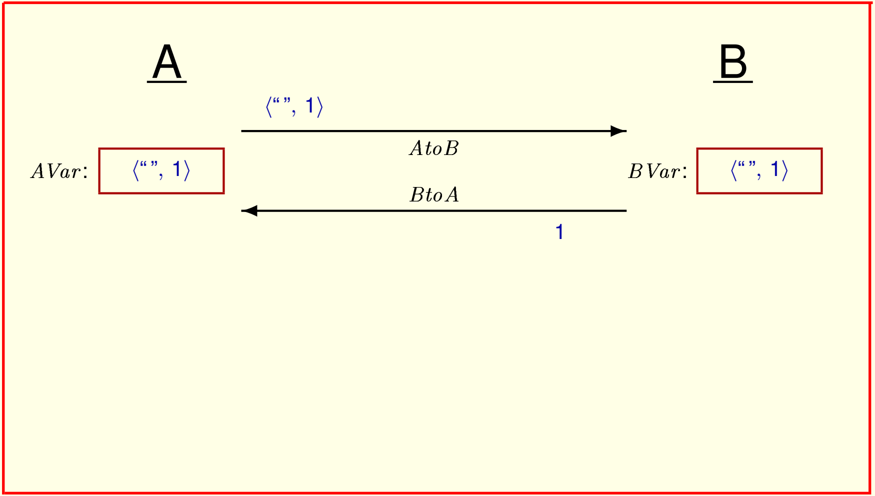
Here's the counterexample that TLC finds.

B sends an acknowledgment, A sends its value, A's message is lost, B's message is lost,



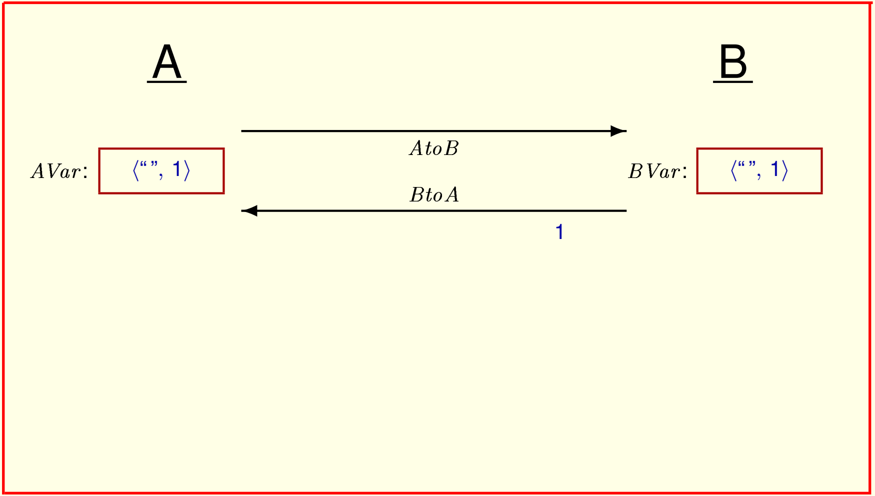
Here's the counterexample that TLC finds.

B sends an acknowledgment, A sends its value, A's message is lost, B's message is lost, B sends a message,



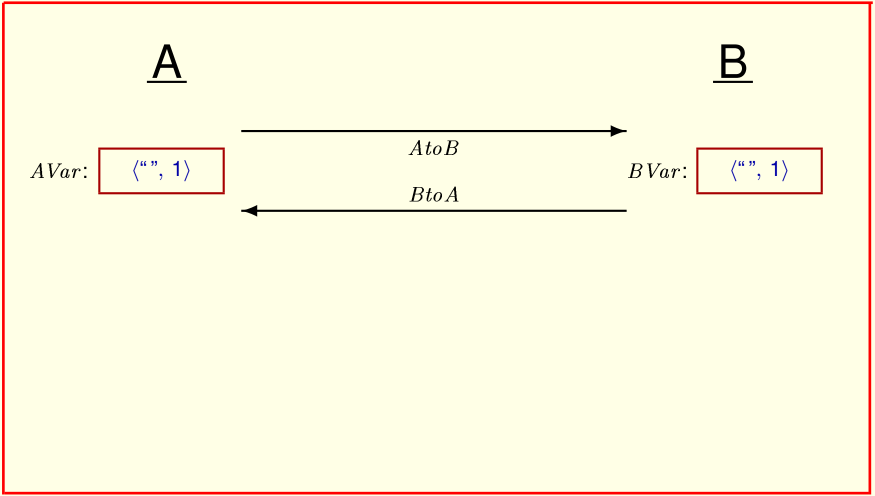
Here's the counterexample that TLC finds.

B sends an acknowledgment, A sends its value, A's message is lost, B's message is lost, B sends a message, A sends a message,



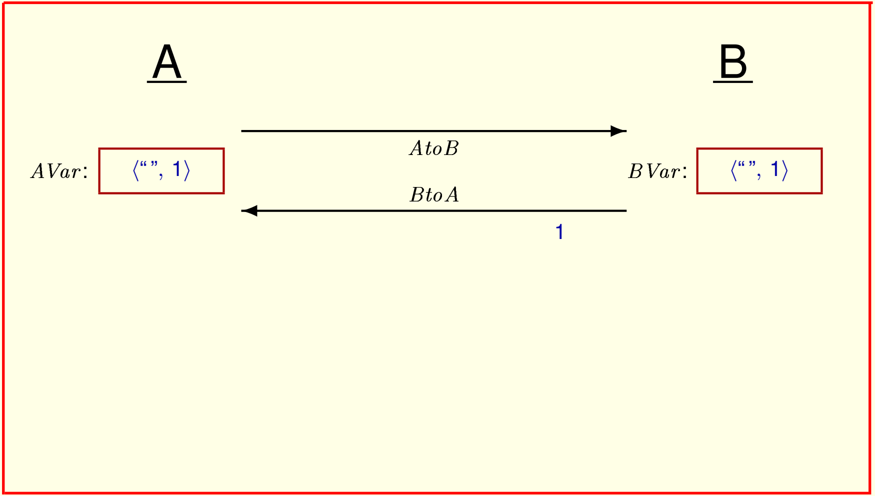
Here's the counterexample that TLC finds.

B sends an acknowledgment, A sends its value, A's message is lost, B's message is lost, B sends a message, A sends a message, A's message is lost,



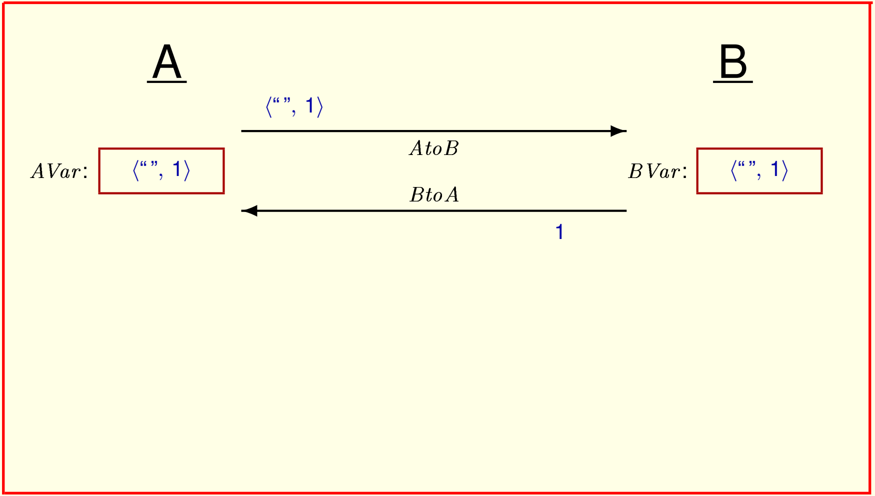
Here's the counterexample that TLC finds.

B sends an acknowledgment, A sends its value, A's message is lost, B's message is lost, B sends a message, A sends a message, A's message is lost, B's message is lost,



Here's the counterexample that TLC finds.

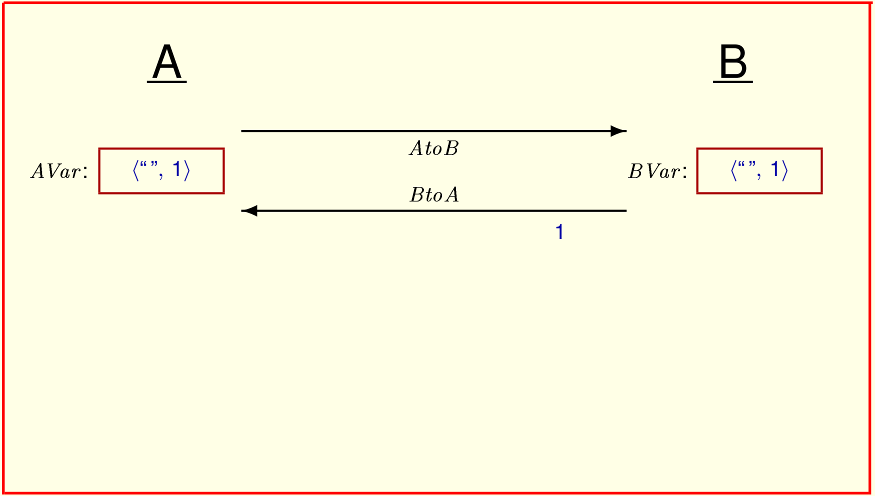
B sends an acknowledgment, A sends its value, A's message is lost, B's message is lost, B sends a message, A sends a message, A's message is lost, B's message is lost, B sends a message,



Here's the counterexample that TLC finds.

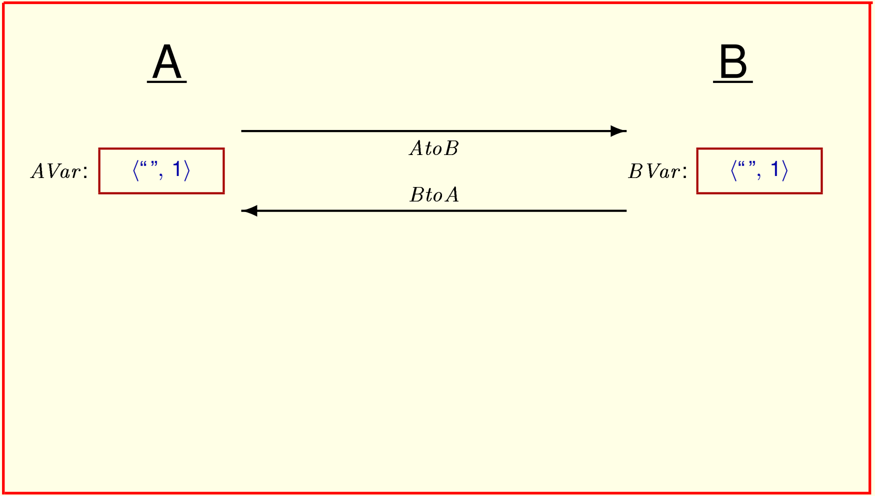
B sends an acknowledgment, A sends its value, A's message is lost, B's message is lost, B sends a message, A sends a message, A's message is lost, B's message is lost, B sends a message, A sends a message,

[slide 206]



Here's the counterexample that TLC finds.

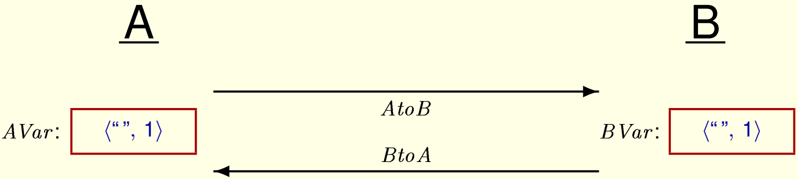
B sends an acknowledgment, A sends its value, A's message is lost, B's message is lost, B sends a message, A sends a message, A's message is lost, B's message is lost, B sends a message, A sends a message, A's message is lost,



Here's the counterexample that TLC finds.

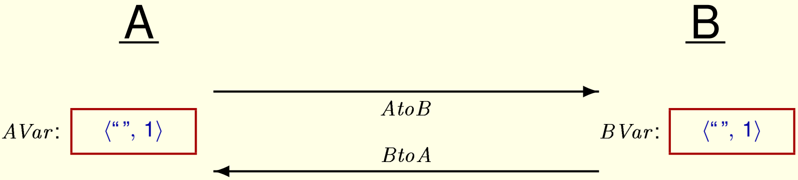
B sends an acknowledgment, A sends its value, A's message is lost, B's message is lost, B sends a message, A sends a message, A's message is lost, B's message is lost, B sends a message, A sends a message, A's message is lost, B's message is lost.

And this continues forever.



$\mathrm{WF}_{vars}(ASnd)$ and $\mathrm{WF}_{vars}(BSnd)$ are true because $ASnd$ and $BSnd$ steps keep occurring.

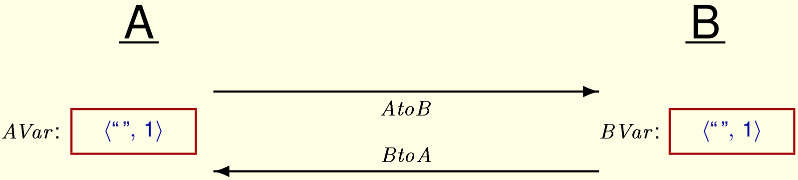
Weak fairness of A-send and B-send are true for this behavior because A-send and B-send steps keep occurring.



What about $\mathrm{WF}_{vars}(ARcv)$?

Weak fairness of A-send and B-send are true for this behavior because A-send and B-send steps keep occurring.

What about weak fairness of A-receive?



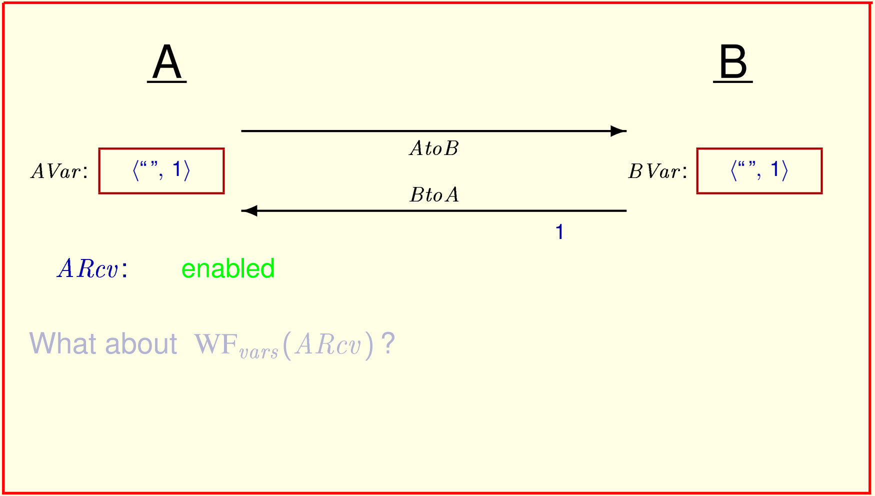
$ARcv$: not enabled

What about $\mathrm{WF}_{vars}(ARcv)$?

Weak fairness of A-send and B-send are true for this behavior because A-send and B-send steps keep occurring.

What about weak fairness of A-receive?

A-receive is not enabled in the initial state, since $BtoA$ contains no messages.

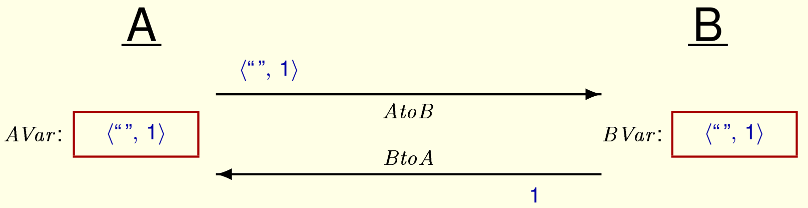


Weak fairness of A-send and B-send are true for this behavior because A-send and B-send steps keep occurring.

What about weak fairness of A-receive?

A-receive is not enabled in the initial state, since $BtoA$ contains no messages.

It becomes enabled when B sends a message.



$ARcv$: enabled

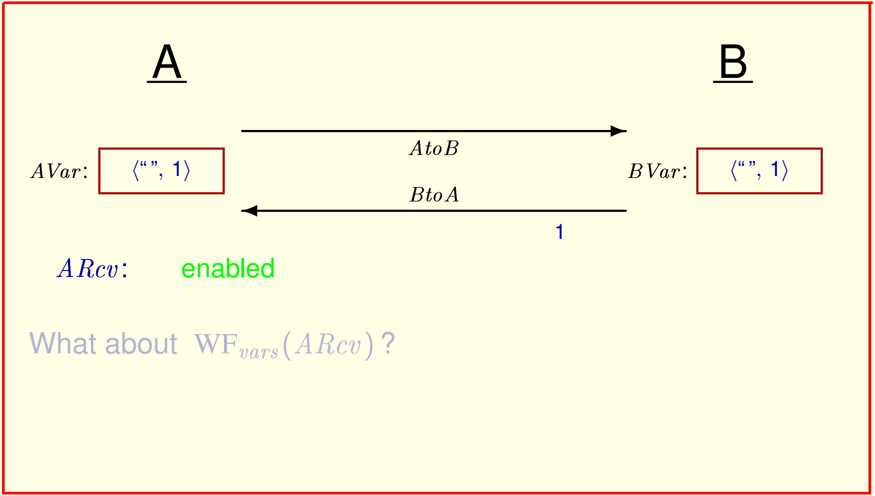
What about $\mathrm{WF}_{vars}(ARcv)$?

Weak fairness of A-send and B-send are true for this behavior because A-send and B-send steps keep occurring.

What about weak fairness of A-receive?

A-receive is not enabled in the initial state, since $BtoA$ contains no messages.

It becomes enabled when B sends a message.

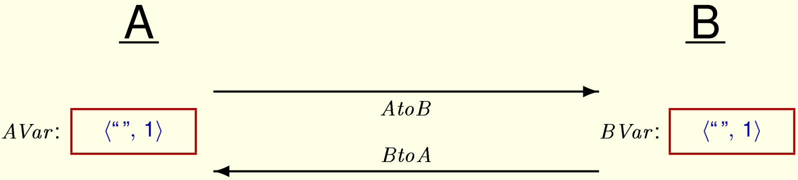


Weak fairness of A-send and B-send are true for this behavior because A-send and B-send steps keep occurring.

What about weak fairness of A-receive?

A-receive is not enabled in the initial state, since $BtoA$ contains no messages.

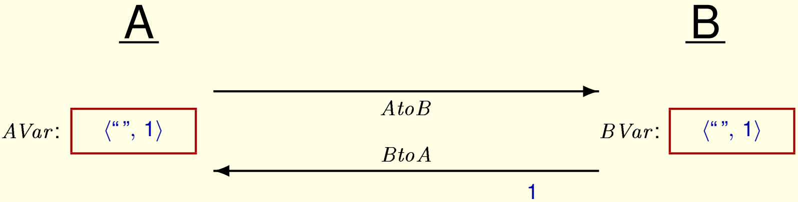
It becomes enabled when B sends a message.



$ARcv$: not enabled

What about $\mathrm{WF}_{vars}(ARcv)$?

It becomes disabled when that message is lost.

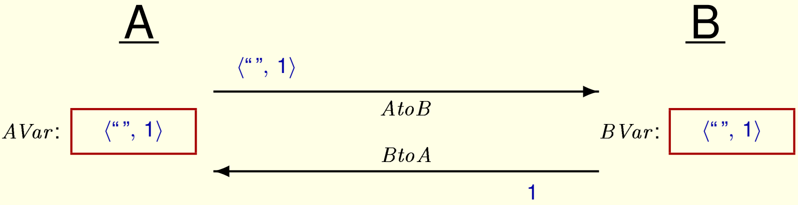


$ARcv$: enabled

What about $\mathrm{WF}_{vars}(ARcv)$?

It becomes disabled when that message is lost.

It becomes enabled again when B sends another message.

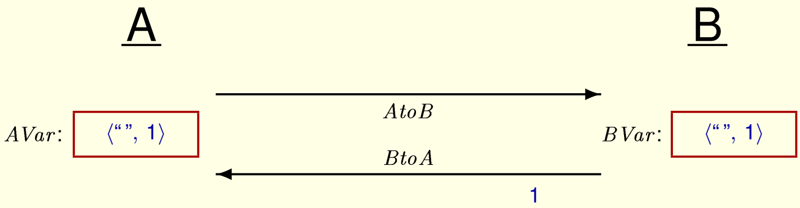


$ARcv$: enabled

What about $\mathrm{WF}_{vars}(ARcv)$?

It becomes disabled when that message is lost.

It becomes enabled again when B sends another message.

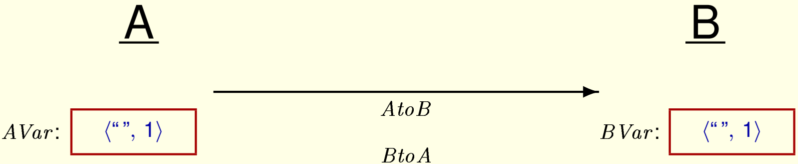


$ARcv$: enabled

What about $\mathrm{WF}_{vars}(ARcv)$?

It becomes disabled when that message is lost.

It becomes enabled again when B sends another message.



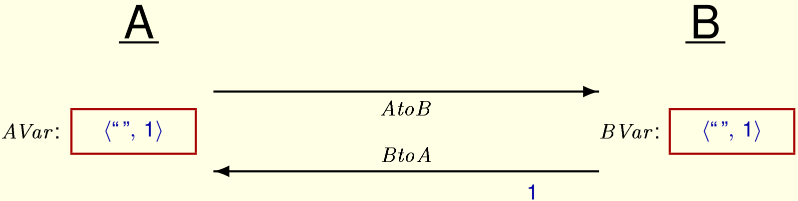
$ARcv$: not enabled

What about $\mathrm{WF}_{vars}(ARcv)$?

It becomes disabled when that message is lost.

It becomes enabled again when B sends another message.

It is disabled again when that message is lost.



$ARcv$: enabled

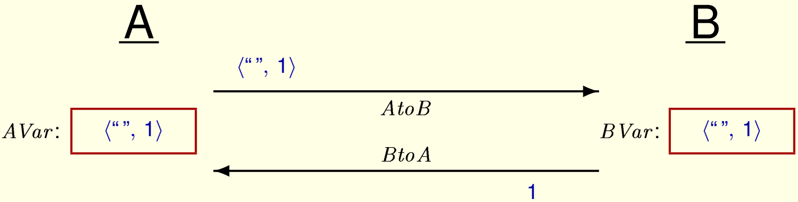
What about $\mathrm{WF}_{vars}(ARcv)$?

It becomes disabled when that message is lost.

It becomes enabled again when B sends another message.

It is disabled again when that message is lost.

It becomes enabled again when B sends yet another message.



$ARcv$: enabled

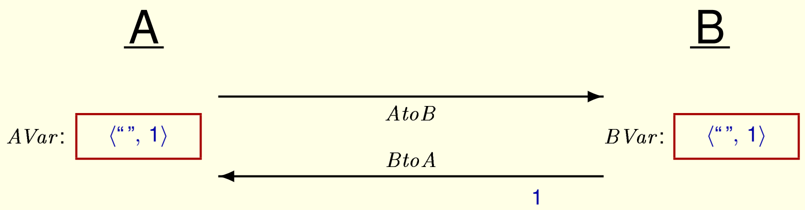
What about $\mathrm{WF}_{vars}(ARcv)$?

It becomes disabled when that message is lost.

It becomes enabled again when B sends another message.

It is disabled again when that message is lost.

It becomes enabled again when B sends yet another message.



$ARcv$: enabled

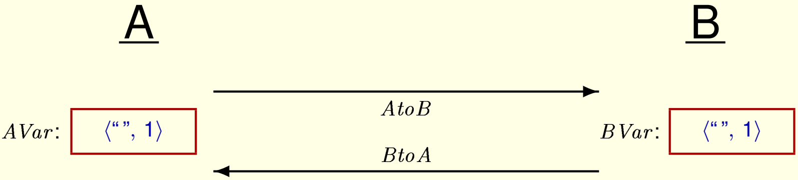
What about $\mathrm{WF}_{vars}(ARcv)$?

It becomes disabled when that message is lost.

It becomes enabled again when B sends another message.

It is disabled again when that message is lost.

It becomes enabled again when B sends yet another message.



$ARcv$: not enabled

What about $\mathrm{WF}_{vars}(ARcv)$?

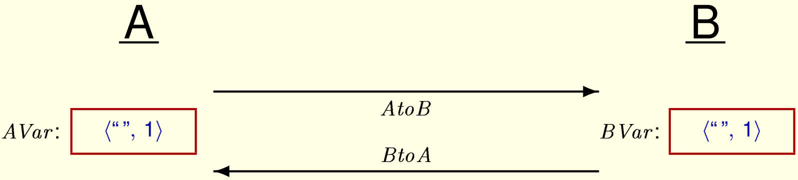
It becomes disabled when that message is lost.

It becomes enabled again when B sends another message.

It is disabled again when that message is lost.

It becomes enabled again when B sends yet another message.

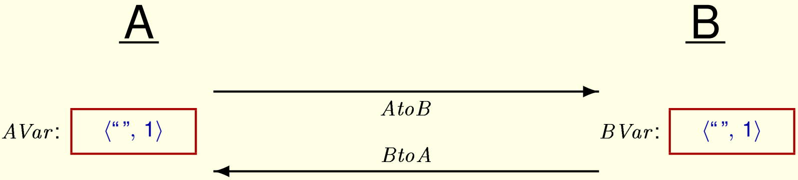
It's disabled again when that message is lost. And so on.



$ARcv$: not enabled

What about $\mathrm{WF}_{vars}(ARcv)$?

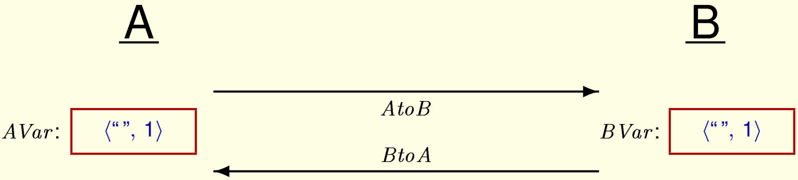
So weak fairness of A-receive



$ARcv$: not enabled

What about $\mathrm{WF}_{vars}(ARcv)$? True

So weak fairness of A-receive is true on this behavior

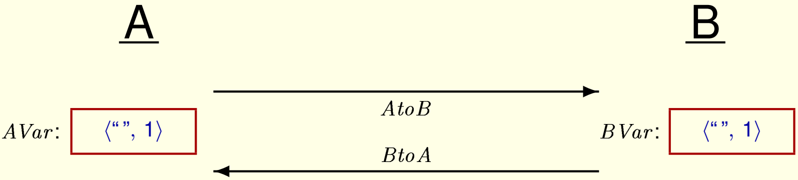


$ARcv$: not enabled

What about $\mathrm{WF}_{vars}(ARcv)$? True
because $ARcv$ never continuously enabled.

So weak fairness of A-receive is true on this behavior

because A-receive keeps getting disabled after it's enabled, and it's never continuously enabled.



$ARcv$: not enabled

$\mathrm{WF}_{vars}(BRcv)$ is also true.

So weak fairness of A-receive is true on this behavior

because A-receive keeps getting disabled after it's enabled, and it's never continuously enabled.

Weak fairness of B-receive is also true on this behavior for the same reason.

The behavior satisfies $FairSpec$, defined by:

$$FairSpec \triangleq Spec \land \mathrm{WF}_{vars}(ARcv) \land \mathrm{WF}_{vars}(BRcv) \land \mathrm{WF}_{vars}(ASnd) \land \mathrm{WF}_{vars}(BSnd)$$

The behavior satisfies $FairSpec$, when it's defined like this.

The behavior satisfies $FairSpec$, defined by:

$$FairSpec \triangleq Spec \land \mathrm{WF}_{vars}(ARcv) \land \mathrm{WF}_{vars}(BRcv) \land \mathrm{WF}_{vars}(ASnd) \land \mathrm{WF}_{vars}(BSnd)$$

but doesn't satisfy $ABS!FairSpec$.

The behavior satisfies $FairSpec$, when it's defined like this.

but it doesn't satisfy the high level fair spec in module $ABSpec$ because no values are ever sent from A to B.

The behavior satisfies $FairSpec$, defined by:

$$FairSpec \triangleq Spec \land \mathrm{WF}_{vars}(ARcv) \land \mathrm{WF}_{vars}(BRcv) \land \mathrm{WF}_{vars}(ASnd) \land \mathrm{WF}_{vars}(BSnd)$$

but doesn't satisfy $ABS!FairSpec$.

~~THEOREM $FairSpec \Rightarrow ABS!FairSpec$~~

The behavior satisfies $FairSpec$, when it's defined like this.

but it doesn't satisfy the high level fair spec in module $ABSpec$ because no values are ever sent from A to B.

So this theorem is not true.

$$
\begin{aligned}
FairSpec \;\triangleq\; Spec \wedge\; & \mathrm{WF}_{vars}(ARcv) \wedge \mathrm{WF}_{vars}(BRcv) \;\wedge \\
& \mathrm{WF}_{vars}(ASnd) \wedge \mathrm{WF}_{vars}(BSnd)
\end{aligned}
$$

The behavior satisfies $FairSpec$, when it's defined like this.

but it doesn't satisfy the high level fair spec in module $ABSpec$ because no values are ever sent from A to B.

So this theorem is not true.

$$FairSpec \triangleq Spec \land \mathrm{WF}_{vars}(ARcv) \land \mathrm{WF}_{vars}(BRcv) \land \mathrm{WF}_{vars}(ASnd) \land \mathrm{WF}_{vars}(BSnd)$$

The problem is that

$$FairSpec \triangleq Spec \land \boxed{\mathrm{WF}_{vars}(ARcv)} \land \boxed{\mathrm{WF}_{vars}(BRcv)} \land \mathrm{WF}_{vars}(ASnd) \land \mathrm{WF}_{vars}(BSnd)$$

Don't imply $ARcv$ or $BRcv$ steps ever occur, because actions keep getting disabled.

The problem is that

these weak fairness conditions don't imply that any A-receive or B-recieve steps ever occur, because those actions keep getting disabled.

Weak fairness of action A asserts of a behavior:

If A ever remains continuously enabled,
then an A step must eventually occur.

Remember that weak fairness of A means if A ever remains continuously enabled, then an A step must eventually occur.

Strong

~~Weak~~ fairness of action A asserts of a behavior:

is repeatedly

If A ever ~~remains continuously~~ enabled,
then an A step must eventually occur.

Remember that weak fairness of A means if A ever remains continuously enabled, then an A step must eventually occur.

Strong fairness of A means that if A ever *is repeatedly* enabled, then an A step must eventually occur.

Strong

~~Weak~~ fairness of action A asserts of a behavior:

is repeatedly

If A ever ~~remains continuously~~ enabled, then an A step must eventually occur.

$\cdots \rightarrow s_{42} \rightarrow s_{43} \rightarrow s_{44} \rightarrow s_{45} \rightarrow s_{46} \rightarrow s_{47} \rightarrow s_{48} \rightarrow s_{49} \rightarrow s_{50} \rightarrow \cdots$

For example, suppose we have a behavior,

Strong

~~Weak~~ fairness of action A asserts of a behavior:

is repeatedly

If A ever ~~remains continuously~~ enabled,
then an A step must eventually occur.

$\cdots \rightarrow s_{42} \rightarrow s_{43} \rightarrow s_{44} \rightarrow s_{45} \rightarrow s_{46} \rightarrow s_{47} \rightarrow s_{48} \rightarrow s_{49} \rightarrow s_{50} \rightarrow \cdots$

A enabled:

For example, suppose we have a behavior, and A enabled is

Strong

~~Weak~~ fairness of action A asserts of a behavior:

is repeatedly

If A ever ~~remains continuously~~ enabled,
then an A step must eventually occur.

$\cdots \rightarrow s_{42} \rightarrow s_{43} \rightarrow s_{44} \rightarrow s_{45} \rightarrow s_{46} \rightarrow s_{47} \rightarrow s_{48} \rightarrow s_{49} \rightarrow s_{50} \rightarrow \cdots$

A enabled: false

For example, suppose we have a behavior, and A enabled is false in this state,

Strong

~~Weak~~ fairness of action A asserts of a behavior:

is repeatedly

If A ever ~~remains continuously~~ enabled,
then an A step must eventually occur.

$\cdots \to s_{42} \to s_{43} \to s_{44} \to s_{45} \to s_{46} \to s_{47} \to s_{48} \to s_{49} \to s_{50} \to \cdots$

A enabled: false true

For example, suppose we have a behavior, and A enabled is false in this state, then true,

Strong

~~Weak~~ fairness of action A asserts of a behavior:

is repeatedly

If A ever ~~remains continuously~~ enabled,
then an A step must eventually occur.

$\cdots \rightarrow s_{42} \rightarrow s_{43} \rightarrow s_{44} \rightarrow s_{45} \rightarrow s_{46} \rightarrow s_{47} \rightarrow s_{48} \rightarrow s_{49} \rightarrow s_{50} \rightarrow \cdots$

A enabled: false true false

For example, suppose we have a behavior, and A enabled is false in this state, then true, the false again,

Strong

~~Weak~~ fairness of action A asserts of a behavior:

is repeatedly

If A ever ~~remains continuously~~ enabled,
then an A step must eventually occur.

$\cdots \rightarrow s_{42} \rightarrow s_{43} \rightarrow s_{44} \rightarrow s_{45} \rightarrow s_{46} \rightarrow s_{47} \rightarrow s_{48} \rightarrow s_{49} \rightarrow s_{50} \rightarrow \cdots$

A enabled: false true false true

For example, suppose we have a behavior, and A enabled is false in this state, then true, the false again, then true,

Strong

~~Weak~~ fairness of action A asserts of a behavior:

is repeatedly

If A ever ~~remains continuously~~ enabled,

then an A step must eventually occur.

$\cdots \rightarrow s_{42} \rightarrow s_{43} \rightarrow s_{44} \rightarrow s_{45} \rightarrow s_{46} \rightarrow s_{47} \rightarrow s_{48} \rightarrow s_{49} \rightarrow s_{50} \rightarrow \cdots$

A enabled: false true false true false

For example, suppose we have a behavior, and A enabled is false in this state, then true, the false again, then true, then false

Strong

~~Weak~~ fairness of action A asserts of a behavior:

is repeatedly

If A ever ~~remains continuously~~ enabled,
then an A step must eventually occur.

$\cdots \rightarrow s_{42} \rightarrow s_{43} \rightarrow s_{44} \rightarrow s_{45} \rightarrow s_{46} \rightarrow s_{47} \rightarrow s_{48} \rightarrow s_{49} \rightarrow s_{50} \rightarrow \cdots$

A enabled: false true false true false true

For example, suppose we have a behavior, and A enabled is false in this state, then true, the false again, then true, then false and so on,

Strong

~~Weak~~ fairness of action A asserts of a behavior:

is repeatedly

If A ever ~~remains continuously~~ enabled, then an A step must eventually occur.

$\cdots \rightarrow s_{42} \rightarrow s_{43} \rightarrow s_{44} \rightarrow s_{45} \rightarrow s_{46} \rightarrow s_{47} \rightarrow s_{48} \rightarrow s_{49} \rightarrow s_{50} \rightarrow \cdots$

A enabled: false true false true false true false

For example, suppose we have a behavior, and A enabled is false in this state, then true, the false again, then true, then false and so on,

Strong

~~Weak~~ fairness of action A asserts of a behavior:

is repeatedly

If A ever ~~remains continuously~~ enabled,
then an A step must eventually occur.

$\cdots \rightarrow s_{42} \rightarrow s_{43} \rightarrow s_{44} \rightarrow s_{45} \rightarrow s_{46} \rightarrow s_{47} \rightarrow s_{48} \rightarrow s_{49} \rightarrow s_{50} \rightarrow \cdots$

A enabled: false true false true false true false false

For example, suppose we have a behavior, and A enabled is false in this state, then true, the false again, then true, then false and so on,

Strong

~~Weak~~ fairness of action A asserts of a behavior:

If A ever ~~remains continuously~~ is repeatedly enabled,
then an A step must eventually occur.

$\cdots \rightarrow s_{42} \rightarrow s_{43} \rightarrow s_{44} \rightarrow s_{45} \rightarrow s_{46} \rightarrow s_{47} \rightarrow s_{48} \rightarrow s_{49} \rightarrow s_{50} \rightarrow \cdots$

A enabled: false true false true false true false false true

For example, suppose we have a behavior, and A enabled is false in this state, then true, the false again, then true, then false and so on,

Strong

~~Weak~~ fairness of action A asserts of a behavior:

is repeatedly

If A ever ~~remains continuously~~ enabled,
then an A step must eventually occur.

$\cdots \to s_{42} \to s_{43} \to s_{44} \to s_{45} \to s_{46} \to s_{47} \to s_{48} \to s_{49} \to s_{50} \to \cdots$

A enabled:	false	true	false	true	false	true	false	false	true

where it keeps being re-enabled after it becomes disabled.

Then an A step must eventually occur.

Strong

~~Weak~~ fairness of action A asserts of a behavior:

is repeatedly

If A ever ~~remains continuously~~ enabled,
then an A step must eventually occur.

$\cdots \rightarrow s_{42} \rightarrow s_{43} \rightarrow s_{44} \rightarrow s_{45} \rightarrow s_{46} \rightarrow s_{47} \rightarrow s_{48} \rightarrow s_{49} \rightarrow s_{50} \rightarrow \cdots$

A enabled:	false	true	false	true	false	true	false	false	true

Or equivalently:

A cannot be repeatedly enabled forever
without another A step occurring.

where it keeps being re-enabled after it becomes disabled.

Then an A step must eventually occur.

An equivalent way of saying this is that A cannot be repeatedly enabled forever without another A step occurring.

$$FairSpec \triangleq Spec \land \mathrm{WF}_{vars}(ARcv) \land \mathrm{WF}_{vars}(BRcv) \land \mathrm{WF}_{vars}(ASnd) \land \mathrm{WF}_{vars}(BSnd)$$

We need to change the definition of $FairSpec$ to what it was originally

$$FairSpec \triangleq Spec \land \boxed{\mathrm{WF}_{vars}(ARcv)} \land \boxed{\mathrm{WF}_{vars}(BRcv)} \land \\ \mathrm{WF}_{vars}(ASnd) \land \mathrm{WF}_{vars}(BSnd)$$

We need to change the definition of $FairSpec$ to what it was originally
changing these weak fairness conditions

$$FairSpec \triangleq Spec \land \boxed{\mathrm{SF}_{vars}(ARcv)} \land \boxed{\mathrm{SF}_{vars}(BRcv)} \land \mathrm{WF}_{vars}(ASnd) \land \mathrm{WF}_{vars}(BSnd)$$

We need to change the definition of $FairSpec$ to what it was originally changing these weak fairness conditions to strong fairness.

$$FairSpec \triangleq Spec \wedge \mathrm{SF}_{vars}(ARcv) \wedge \mathrm{SF}_{vars}(BRcv) \wedge \mathrm{WF}_{vars}(ASnd) \wedge \boxed{\mathrm{WF}_{vars}(BSnd)}$$

B must keep sending messages

We need to change the definition of $FairSpec$ to what it was originally changing these weak fairness conditions to strong fairness.

Since the B-send action is always enabled, weak fairness of B-send implies that B keeps sending messages.

$$FairSpec \triangleq Spec \land \boxed{\mathrm{SF}_{vars}(ARcv)} \land \mathrm{SF}_{vars}(BRcv) \land$$
$$\mathrm{WF}_{vars}(ASnd) \land \boxed{\mathrm{WF}_{vars}(BSnd)}$$

B must keep sending messages
which implies A must eventually
receive those messages.

We need to change the definition of $FairSpec$ to what it was originally changing these weak fairness conditions to strong fairness.

Since the B-send action is always enabled, weak fairness of B-send implies that B keeps sending messages. This keeps enabling A-receive which, by strong fairness implies that A-receive steps must eventually occur to receive those messages — even if $Lose$-message actions keep disabling A-receive.

$$FairSpec \triangleq Spec \wedge \mathrm{SF}_{vars}(ARcv) \wedge \mathrm{SF}_{vars}(BRcv) \wedge$$
$$\boxed{\mathrm{WF}_{vars}(ASnd)} \wedge \mathrm{WF}_{vars}(BSnd)$$

A must keep sending messages

Similarly, A must keep sending messages

$$FairSpec \triangleq Spec \land \mathrm{SF}_{vars}(ARcv) \land \boxed{\mathrm{SF}_{vars}(BRcv)} \land \boxed{\mathrm{WF}_{vars}(ASnd)} \land \mathrm{WF}_{vars}(BSnd)$$

A must keep sending messages
that B must eventually receive.

Similarly, A must keep sending messages that B must eventually receive.

$$FairSpec \triangleq Spec \land \mathrm{SF}_{vars}(ARcv) \land \mathrm{SF}_{vars}(BRcv) \land \mathrm{WF}_{vars}(ASnd) \land \mathrm{WF}_{vars}(BSnd)$$

Similarly, A must keep sending messages that B must eventually receive.

With this definition,

$$FairSpec \triangleq Spec \land \mathrm{SF}_{vars}(ARcv) \land \mathrm{SF}_{vars}(BRcv) \land \mathrm{WF}_{vars}(ASnd) \land \mathrm{WF}_{vars}(BSnd)$$

THEOREM $FairSpec \Rightarrow ABS!FairSpec$

Similarly, A must keep sending messages that B must eventually receive.

With this definition, the theorem is true.

$$FairSpec \triangleq Spec \land \mathrm{SF}_{vars}(ARcv) \land \mathrm{SF}_{vars}(BRcv) \land \mathrm{WF}_{vars}(ASnd) \land \mathrm{WF}_{vars}(BSnd)$$

THEOREM $FairSpec \Rightarrow ABS!FairSpec$

TLC will now find no error.

Similarly, A must keep sending messages that B must eventually receive.

With this definition, the theorem is true.

You can change the definition of $FairSpec$ in the module and rerun the model, and TLC will now find no error.

What Good is Liveness?

What Good is Liveness?

What Good is Liveness?

What good is knowing that something eventually happens?

What Good is Liveness?

What good is knowing that something eventually happens?

What Good is Liveness?

What good is knowing that something eventually happens – if it could be 10^6 years from now?

What Good is Liveness?

What good is knowing that something eventually happens?
If it could be a million years from now when it happens.

What Good is Liveness?

What good is knowing that something eventually happens – if it could be 10^6 years from now?

How can we ensure strong fairness of the $ARcv$ and $BRcv$ actions?

How can we ensure strong fairness of the $ARcv$ and $BRcv$ actions?

What Good is Liveness?

What good is knowing that something eventually happens – if it could be 10^6 years from now?

How can we ensure strong fairness of the $ARcv$ and $BRcv$ actions? Or ever know that it's not satisfied?

How can we ensure strong fairness of the $ARcv$ and $BRcv$ actions?
Or ever know that it's not satisfied? Since it would take forever to be sure that it's not.

A specification is an abstraction.

A specification is an abstraction.

A specification is an abstraction.

It’s a compromise between our desires for accuracy and simplicity.

A specification is an abstraction.

It’s a compromise between our desires for accuracy and simplicity.

A specification is an abstraction.

It’s a compromise between our desires for
accuracy and simplicity.

We’d like to require that a message is received within 4.7 ms.

A specification is an abstraction.

It’s a compromise between our desires for accuracy and simplicity.

We’d like to require that a message is received within 4.7 milliseconds of when it’s sent.

A specification is an abstraction.

It's a compromise between our desires for accuracy and simplicity.

We'd like to require that a message is received within 4.7 ms.

But that would require specifying:

But that would require specifying:

A specification is an abstraction.

It’s a compromise between our desires for
accuracy and simplicity.

We’d like to require that a message is received within 4.7 ms.

But that would require specifying:

– How long it can take a message to be received.

But that would require specifying:

How long it can take a message to be received.

A specification is an abstraction.

It’s a compromise between our desires for accuracy and simplicity.

We’d like to require that a message is received within 4.7 ms.

But that would require specifying:

– How long it can take a message to be received.

– How often messages can be lost.

But that would require specifying:

How long it can take a message to be received.

How often messages can be lost.

A specification is an abstraction.

It's a compromise between our desires for accuracy and simplicity.

We'd like to require that a message is received within 4.7 ms.

But that would require specifying:

- How long it can take a message to be received.
- How often messages can be lost.
- How frequently messages are retransmitted.

But that would require specifying:

How long it can take a message to be received.

How often messages can be lost.

And how frequently messages are retransmitted.

It’s simpler to require that a message is eventually received.

It’s simpler to require that a message is eventually received.

It's simpler to require that a message is eventually received.

If it's not eventually received, it can't be received within 4.7 ms.

It's simpler to require that a message is eventually received.

And if it's not eventually received, it certainly can't be received within 4.7 milliseconds.

It’s simpler to require that a message is eventually received.

If it’s not eventually received, it can’t be received within 4.7 ms.

For systems without hard real-time response requirements,

It’s simpler to require that a message is eventually received.

And if it’s not eventually received, it certainly can’t be received within 4.7 milliseconds.

For systems without hard real-time response requirements,

It’s simpler to require that a message is eventually received.

If it’s not eventually received, it can’t be received within 4.7 ms.

For systems without hard real-time response requirements, liveness checking is a useful way to find errors that prevent things from happening.

It’s simpler to require that a message is eventually received.

And if it’s not eventually received, it certainly can’t be received within 4.7 milliseconds.

For systems without hard real-time response requirements,

liveness checking is a useful way to find errors that prevent things from happening.

Many systems use timeouts only to ensure that something must happen.

Many systems use timeouts only to ensure that something must happen.

Many systems use timeouts only to ensure that something must happen.

Many systems use timeouts only to ensure that something must happen.

By using timeouts **only** for that purpose, I mean that

Many systems use timeouts only to ensure that something must happen.

Correctness of such a system does not depend on how long it takes the timeouts to occur.

Many systems use timeouts only to ensure that something must happen.
By using timeouts **only** for that purpose, I mean that correctness of such a system does not depend on how long it takes the timeouts to occur.

Many systems use timeouts only to ensure that something must happen.

Correctness of such a system does not depend on how long it takes the timeouts to occur.

That can influence only performance.

Many systems use timeouts only to ensure that something must happen. By using timeouts **only** for that purpose, I mean that correctness of such a system does not depend on how long it takes the timeouts to occur.

That can influence only performance.

Many systems use timeouts only to ensure that
something must happen.

Correctness of such a system does not depend
on how long it takes the timeouts to occur.

Specifications of these systems can describe timeouts as actions
with no time constraints, only weak fairness conditions.

Specifications of these systems can describe timeouts as actions with no time constraints, only weak fairness conditions.

Many systems use timeouts only to ensure that something must happen.

Correctness of such a system does not depend on how long it takes the timeouts to occur.

Specifications of these systems can describe timeouts as actions with no time constraints, only weak fairness conditions.

This is true for most systems with no bounds on how long it can take an enabled operation (such as receiving a message) to occur.

Specifications of these systems can describe timeouts as actions with no time constraints, only weak fairness conditions.

This is true for most systems with no bounds on how long it can take an enabled operation (such as receiving a message) to occur.

In the first eight lectures, you learned about writing the safety part of a TLA+ spec. Now you know how to specify liveness. You simply add weak and strong fairness conditions. Simple, yes. Easy, no. Liveness is inherently subtle. TLA+ is the simplest way I know to express it, and it's still hard.

But don't worry if you have trouble with liveness. The safety part is by far the largest part and almost always the most important part of a spec. A major reason to add liveness is to catch errors in the safety part. If your fairness conditions don't imply the eventually or leads-to properties you expect to hold, it could be because the safety part doesn't allow behaviors that it should.

[slide 281]

End of Lecture 9, Part 2

THE ALTERNATING BIT PROTOCOL

THE PROTOCOL