TLA$^+$ Video Course – Lecture 9, Part 1

Leslie Lamport

THE ALTERNATING BIT PROTOCOL

THE HIGH LEVEL SPECIFICATION

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

The TLA$^+$ Video Course
Lecture 9
The Alternating Bit Protocol

Up until now, we have been specifying what a system *may* do. The main purpose of this lecture is to explain how to specify what a system *must* do. It's based on a single example: the Alternating Bit Protocol — a simple algorithm for sending data across a channel that can lose messages.

In Part 1, we specify *what* the protocol should do. We will specify *how* it does it in Part 2.

But before we get to the protocol, we learn about the TLA+ operators for using a very important data structure: finite sequences, which programmers often call *lists*.
[slide 2]

FINITE SEQUENCES

Finite sequence is another name for *tuple*.

Finite sequence is just another name for *tuple*.

Finite sequence is another name for *tuple*.

$\langle -3, \text{“}xyz\text{”}, \{0, 2\} \rangle$ is a sequence of length 3.

Finite sequence is just another name for *tuple*.

So this tuple is a sequence of length 3.

Finite sequence is another name for *tuple*.

$\langle\ -3\ ,\ \text{“}xyz\text{”}\ ,\ \{0,2\}\ \rangle$ is a sequence of length 3.

Finite sequence is just another name for *tuple*.

So this tuple is a sequence of length 3.

Remember that this tuple

Finite sequence is another name for *tuple*.

$\langle\ -3\ ,\ \text{"}xyz\text{"}\ ,\ \{0, 2\}\ \rangle$ is a sequence of length 3.

```
<< -3, "xyz", {0,2} >>
```

Finite sequence is just another name for *tuple*.

So this tuple is a sequence of length 3.

Remember that this tuple is typed like this.

Finite sequence is another name for *tuple*.

⟨ ⟩

<< >>

Finite sequence is just another name for *tuple*.

So this tuple is a sequence of length 3.

Remember that this tuple is typed like this.

Where the angle brackets are typed double less-than and double greater-than.

[slide 8]

Finite sequence is another name for *tuple*.

$\langle -3, \text{“}xyz\text{”}, \{0,2\} \rangle$ is a sequence of length 3.

Finite sequence is just another name for *tuple*.

So this tuple is a sequence of length 3.

Remember that this tuple is typed like this.

Where the angle brackets are typed double less-than and double greater-than.

Finite sequence is another name for *tuple*.

$\langle -3, \text{"}xyz\text{"}, \{0, 2\} \rangle$ is a sequence of length 3.

A sequence of length N is a function with domain $1 \,..\, N$.

A sequence of length N is a function whose domain is the set of integers from 1 through N.

Finite sequence is another name for *tuple*.

$\langle -3, \text{"}xyz\text{"}, \{0, 2\} \rangle$ is a sequence of length 3.

A sequence of length N is a function with domain $1 \,..\, N$.

$\langle -3, \text{"}xyz\text{"}, \{0, 2\} \rangle[1]$

A sequence of length N is a function whose domain is the set of integers from 1 through N.

This sequence of length 3 applied to the number one

Finite sequence is another name for *tuple*.

$\langle -3, \text{“}xyz\text{”}, \{0, 2\} \rangle$ is a sequence of length 3.

A sequence of length N is a function with domain $1 \,..\, N$.

$$\langle -3, \text{“}xyz\text{”}, \{0, 2\} \rangle[1] \;=\; -3$$

A sequence of length N is a function whose domain is the set of integers from 1 through N.

This sequence of length 3 applied to the number one equals its first element.

Finite sequence is another name for *tuple*.

$\langle -3, \text{“}xyz\text{”}, \{0, 2\} \rangle$ is a sequence of length 3.

A sequence of length N is a function with domain $1 \,..\, N$.

$\langle -3, \text{“}xyz\text{”}, \{0, 2\} \rangle[1] \;=\; -3$

$\langle -3, \text{“}xyz\text{”}, \{0, 2\} \rangle[2]$

A sequence of length N is a function whose domain is the set of integers from 1 through N.

This sequence of length 3 applied to the number one equals its first element.

The sequence applied to the number two

Finite sequence is another name for *tuple*.

$\langle -3, \text{“}xyz\text{”}, \{0, 2\} \rangle$ is a sequence of length 3.

A sequence of length N is a function with domain $1 \,..\, N$.

$\langle -3, \text{“}xyz\text{”}, \{0, 2\} \rangle[1] = -3$

$\langle -3, \text{“}xyz\text{”}, \{0, 2\} \rangle[2] = \text{“}xyz\text{”}$

A sequence of length N is a function whose domain is the set of integers from 1 through N.

This sequence of length 3 applied to the number one equals its first element.

The sequence applied to the number two equals its second element.

Finite sequence is another name for *tuple*.

$\langle -3, \text{“}xyz\text{”}, \{0, 2\} \rangle$ is a sequence of length 3.

A sequence of length N is a function with domain $1 \mathrel{..} N$.

$\langle -3, \text{“}xyz\text{”}, \{0, 2\} \rangle[1] = -3$

$\langle -3, \text{“}xyz\text{”}, \{0, 2\} \rangle[2] = \text{“}xyz\text{”}$

$\langle -3, \text{“}xyz\text{”}, \{0, 2\} \rangle[3]$

A sequence of length N is a function whose domain is the set of integers from 1 through N.

This sequence of length 3 applied to the number one equals its first element.

The sequence applied to the number two equals its second element.

And applied to the number three

Finite sequence is another name for *tuple*.

$\langle -3, \text{"}xyz\text{"}, \{0, 2\} \rangle$ is a sequence of length 3.

A sequence of length N is a function with domain $1 \,..\, N$.

$\langle -3, \text{"}xyz\text{"}, \{0, 2\} \rangle[1] = -3$

$\langle -3, \text{"}xyz\text{"}, \{0, 2\} \rangle[2] = \text{"}xyz\text{"}$

$\langle -3, \text{"}xyz\text{"}, \{0, 2\} \rangle[3] = \{0, 2\}$

A sequence of length N is a function whose domain is the set of integers from 1 through N.

This sequence of length 3 applied to the number one equals its first element.

The sequence applied to the number two equals its second element.

And applied to the number three equals its third element.

Finite sequence is another name for *tuple*.

$\langle -3, \text{"}xyz\text{"}, \{0,2\} \rangle$ is a sequence of length 3.

A sequence of length N is a function with domain $1 \,..\, N$.

$$\langle -3, \text{"}xyz\text{"}, \{0,2\} \rangle[1] \;=\; -3$$

$$\langle -3, \text{"}xyz\text{"}, \{0,2\} \rangle[2] \;=\; \text{"}xyz\text{"}$$

$$\langle -3, \text{"}xyz\text{"}, \{0,2\} \rangle[3] \;=\; \{0,2\}$$

A sequence of length N is a function whose domain is the set of integers from 1 through N.

This sequence of length 3 applied to the number one equals its first element.

The sequence applied to the number two equals its second element.

And applied to the number three equals its third element.

The sequence $\langle 1, 4, 9, \ldots, N^2 \rangle$ is the function such that

The sequence of the squares of the first N positive integers is the function which

The sequence $\langle 1, 4, 9, \ldots, N^2 \rangle$ is the function such that

$$\langle 1, 4, 9, \ldots, N^2 \rangle [i] = i^2$$

The sequence of the squares of the first N positive integers is the function which

when applied to the number i, equals i squared

The sequence $\langle 1, 4, 9, \ldots, N^2 \rangle$ is the function such that

$$\langle 1, 4, 9, \ldots, N^2 \rangle [i] = i^2$$

for all i in $1 \,..\, N$.

The sequence of the squares of the first N positive integers is the function which
when applied to the number i, equals i squared
for all i in its domain, the integers from 1 through N.

The sequence $\langle 1, 4, 9, \ldots, N^2 \rangle$ is the function such that

$$\langle 1, 4, 9, \ldots, N^2 \rangle [i] = i^2$$

for all i in $1 \,..\, N$.

It is written $[i \in 1 \,..\, N \mapsto i^2]$.

The sequence of the squares of the first N positive integers is the function which
when applied to the number i, equals i squared
for all i in its domain, the integers from 1 through N.

That function is usually written this way

The sequence $\langle 1, 4, 9, \ldots, N^2 \rangle$ is the function such that

$$\langle 1, 4, 9, \ldots, N^2 \rangle [i] = i^2$$

for all i in $1 \,..\, N$.

It is written $[i \in 1 \,..\, N \mapsto i^2]$.

typed `i^2`

The sequence of the squares of the first N positive integers is the function which
when applied to the number i, equals i squared
for all i in its domain, the integers from 1 through N.

That function is usually written this way
where the exponentiation operator is represented by the caret character.

The $Sequences$ Module

The standard $Sequences$ module

The *Sequences* Module

Defines useful operators.

The standard *Sequences* module defines some useful operators on finite sequences.

The $\mathit{Sequences}$ Module

$\mathit{Tail}(\langle s_1, \ldots, s_n \rangle)$ equals $\langle s_2, \ldots, s_n \rangle$.

The standard $\mathit{Sequences}$ module defines some useful operators on finite sequences.

The tail of a non-empty sequence equals the sequence obtained 1by chopping off its first element

The $Sequences$ Module

$Tail(\langle s_1, \ldots, s_n \rangle)$ equals $\langle s_2, \ldots, s_n \rangle$.

$$Head(seq) \triangleq seq[1]$$

The standard $Sequences$ module defines some useful operators on finite sequences.

The tail of a non-empty sequence equals the sequence obtained 1by chopping off its first element

And since it would be funny to have a tail without a head, we call the first element its head.

The $Sequences$ Module

$Tail(\langle s_1, \ldots, s_n \rangle)$ equals $\langle s_2, \ldots, s_n \rangle$.

$Head(seq) \triangleq seq[1]$

$\circ$ (concatenation)

The concatenation operator

The *Sequences* Module

$Tail(\langle s_1, \ldots, s_n \rangle)$ equals $\langle s_2, \ldots, s_n \rangle$.

$Head(seq) \triangleq seq[1]$

$\circ$ (concatenation)

$\backslash o$

The concatenation operator which we type backslash lower-case Oh, concatenates two sequences

The $Sequences$ Module

$Tail(\langle s_1, \ldots, s_n \rangle)$ equals $\langle s_2, \ldots, s_n \rangle$.

$Head(seq) \triangleq seq[1]$

- $\circ$ (concatenation)

  $\langle 3, 2, 1 \rangle \circ \langle \text{“}a\text{”}, \text{“}b\text{”} \rangle = \langle 3, 2, 1, \text{“}a\text{”}, \text{“}b\text{”} \rangle$

The concatenation operator which we type backslash lower-case Oh, concatenates two sequences as in this example.

The $Sequences$ Module

$Tail(\langle s_1, \ldots, s_n \rangle)$ equals $\langle s_2, \ldots, s_n \rangle$.

$Head(seq) \triangleq seq[1]$

- $\circ$ (concatenation)

  If $seq \neq \langle \, \rangle$ then $seq = \langle Head(seq) \rangle \circ Tail(seq)$

The concatenation operator which we type backslash lower-case Oh, concatenates two sequences as in this example.

Any non-empty sequence is the concatenation of the one-element sequence containing only its head, with its tail.

The $Sequences$ Module

$Tail(\langle s_1, \ldots, s_n \rangle)$ equals $\langle s_2, \ldots, s_n \rangle$.

$Head(seq) \triangleq seq[1]$

$\circ$ (concatenation)

$Append(seq, e) \triangleq seq \circ \langle e \rangle$

The concatenation operator which we type backslash lower-case Oh, concatenates two sequences as in this example.

Any non-empty sequence is the concatenation of the one-element sequence containing only its head, with its tail.

The append operator appends an element to the end of a sequence.

$Len(seq)$ equals the length of sequence seq.

The operator L-E-N applied to a sequence equals the sequence's length.

$Len(seq)$ equals the length of sequence seq.

The domain of seq is $1 \, .. \, Len(seq)$.

The operator L-E-N applied to a sequence equals the sequence's length.

Note that the domain of a sequence is the set of integers from 1 to the sequence's length.

$Len(seq)$ equals the length of sequence seq.

The domain of seq is $1 \,..\, Len(seq)$.

$1 \,..\, 0 = \{\}$, which is the domain of $\langle \rangle$.

The operator L-E-N applied to a sequence equals the sequence's length.

Note that the domain of a sequence is the set of integers from 1 to the sequence's length.

Note also that one dot-dot zero is the empty set, which is the domain of the empty sequence.

$Len(seq)$ equals the length of sequence seq.

$Seq(S)$ is the set of all sequences with elements in S.

The S-E-Q operator applied to a set equals the set of all finite sequences formed from the elements of that set.

$Len(seq)$ equals the length of sequence seq.

$Seq(S)$ is the set of all sequences with elements in S.

$$Seq(\{3\}) = \{\langle\rangle, \langle 3\rangle, \langle 3, 3\rangle, \langle 3, 3, 3\rangle, \ldots\}.$$

The S-E-Q operator applied to a set equals the set of all finite sequences formed from the elements of that set.

For example, S-E-Q applied to the set containing the single element 3 equals this infinite set of sequences.

Let's define $Remove(i, seq)$ to be the sequence obtained by removing the i^{th} element from the sequence seq .

For later use, let's now define the $Remove$ operator so $Remove$ of i, $seek$ is the sequence obtained by removing the i^{th} element from the sequence $seek$.

Let's define $Remove(i, seq)$ to be the sequence obtained by removing the i^{th} element from the sequence seq .

$$Len(Remove(i, seq)) = Len(seq) - 1$$

For later use, let's now define the $Remove$ operator so $Remove$ of i, $seek$ is the sequence obtained by removing the i^{th} element from the sequence $seek$.

The length of $Remove$ of i, $seek$ should be one less than the length of $seek$.

Let's define $Remove(i, seq)$ to be the sequence obtained by removing the i^{th} element from the sequence seq.

$Len(Remove(i, seq)) = Len(seq) - 1$, so

$$Remove(i, seq) \triangleq [j \in 1 \,..\, (Len(seq) - 1) \mapsto \\ \quad \ldots \quad]$$

For later use, let's now define the $Remove$ operator so $Remove$ of i, $seek$ is the sequence obtained by removing the i^{th} element from the sequence $seek$.

The length of $Remove$ of i, $seek$ should be one less than the length of $seek$. so $Remove$ of i, $seek$ should be defined like this to be a function whose domain is the set of integers from one to the length of $seek$ minus one.

[slide 39]

Let's define $Remove(i, seq)$ to be the sequence obtained by removing the i^{th} element from the sequence seq.

$Len(Remove(i, seq)) = Len(seq) - 1$, so

$$Remove(i, seq) \triangleq [j \in 1 \,..\, (Len(seq) - 1) \mapsto \boxed{\ldots}\ ]$$

We just have to fill in the dot-dot-dot.

Let's define $Remove(i, seq)$ to be the sequence obtained by removing the i^{th} element from the sequence seq.

$Len(Remove(i, seq)) = Len(seq) - 1$, so

$$
\begin{aligned}
Remove(i, seq) \;\triangleq\; [j \in 1 \,..\, (Len(seq) - 1) \;\mapsto\; & \text{IF } j < i \text{ THEN } seq[j] \\
& \qquad\quad\;\; \text{ELSE } seq[j+1] \;]
\end{aligned}
$$

We just have to fill in the dot-dot-dot.

A little thought shows that the definition should be this.

Well, a little thought when you're more used to writing specs. It might be a lot of thought now.

Let's define $Remove(i, seq)$ to be the sequence obtained by removing the i^{th} element from the sequence seq.

$Len(Remove(i, seq)) = Len(seq) - 1$, so

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

Let's check this.

We just have to fill in the dot-dot-dot.

A little thought shows that the definition should be this.

Well, a little thought when you're more used to writing specs. It might be a lot of thought now.

So we should check this definition. Here's how.

Create a new spec with this body, which you can copy from the Web page:

$$\text{EXTENDS } Integers, Sequences$$

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

Create a new spec with this body, which you can copy from the Web page.

Create a new spec with this body, which
you can copy from the Web page:

$$\text{EXTENDS } Integers, Sequences$$

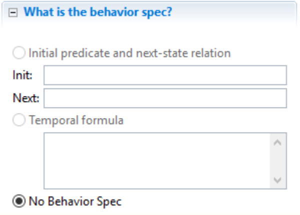
$$Remove(i, seq) \triangleq [j \in 1 \,..\, (Len(seq) - 1) \mapsto \text{IF } j < i \text{ THEN } seq[j] \text{ ELSE } seq[j+1]]$$

Create a new model.

Create a new spec with this body, which
you can copy from the Web page.

Now create a new model.

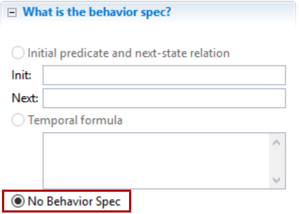
The *Model Overview* page will show



The model's *Model Overview* page will show

[slide 45]

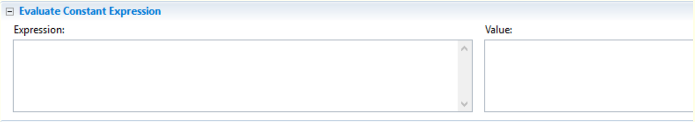
The *Model Overview* page will show



The model's *Model Overview* page will show that there are no behaviors to be checked.

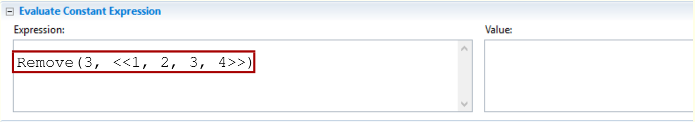
(TLC can still check assumptions.)

On the *Model Checking Results* page



On the Model Checking Results page

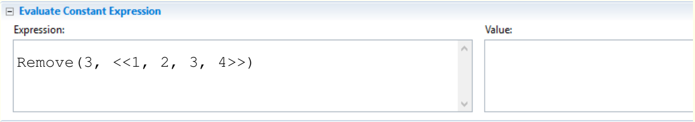
On the *Model Checking Results* page



Enter an expression to check.

On the Model Checking Results page Enter an expression to check, such as this one.

On the *Model Checking Results* page



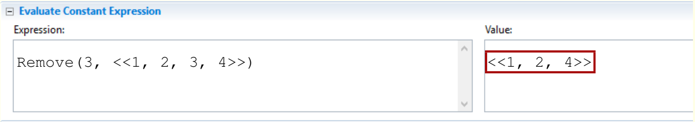
Enter an expression to check.

Run TLC on the model.

On the Model Checking Results page Enter an expression to check, such as this one.

Run TLC on the model.

On the *Model Checking Results* page



Enter an expression to check.

TLC computes its value.

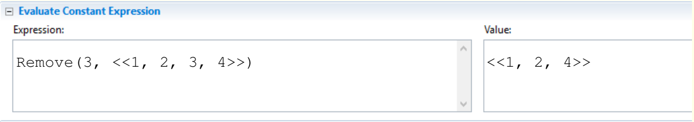
Run TLC on the model.

On the Model Checking Results page Enter an expression to check, such as this one.

Run TLC on the model.

TLC will compute the value of the expression.

On the *Model Checking Results* page



Enter an expression to check.

TLC computes its value.

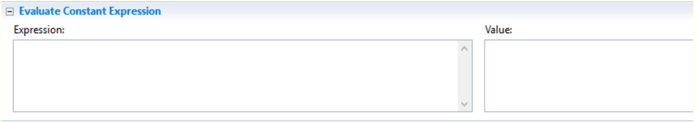
Run TLC on the model.

On the Model Checking Results page Enter an expression to check, such as this one.

Run TLC on the model.

TLC will compute the value of the expression.
in this case checking that $Remove$ has the correct value for these arguments.

You can evaluate a constant expression on any model of any spec.



You can evaluate a constant expression on any model, with or without a behavioral spec.

The Cartesian Product

The Cartesian Product

The Cartesian Product

For any sets S and T

$$S \times T = \text{ the set of all } \langle a, b \rangle \text{ with}$$
$$a \in S \text{ and } b \in T.$$

The Cartesian Product

For any sets S and T their cartesian product S cross T equals the set of all pairs a, b with a in S and b in T.

The Cartesian Product

For any sets S and T

$$S \times T \ = \ \{\langle a, b\rangle : a \in S,\ b \in T\}$$

The Cartesian Product

For any sets S and T their cartesian product S cross T equals the set of all pairs a, b with a in S and b in T.

That set can also be written like this.

The Cartesian Product

For any sets S and T

$$S \times T \;=\; \{\langle a, b \rangle : a \in S,\; b \in T\}$$

ASCII: \X

The Cartesian Product

For any sets S and T their cartesian product S cross T equals the set of all pairs a, b with a in S and b in T.

That set can also be written like this.

The *cross* operator is typed *backslash upper-case X*.

The Cartesian Product

For any sets S and T

$$S \times T \; = \; \{\langle a, b \rangle : a \in S, \; b \in T\}$$

Let TLC compute $(1 \,..\, 3) \times \{\text{“}a\text{”}, \text{“}b\text{”}\}$.

Stop the video and let TLC compute this 6-element set.

The Cartesian Product

For any sets S and T

$$S \times T \;=\; \{\langle a, b\rangle : a \in S,\; b \in T\}$$

Let TLC compute $1\,..\,3 \;\times \{\text{“}a\text{”}, \text{“}b\text{”}\}$.

Stop the video and let TLC compute this 6-element set.

Now see what happens if you remove the parentheses.

The Cartesian Product

For any sets S and T

$$S \times T \;=\; \{\langle a, b \rangle : a \in S,\; b \in T\}$$

Let TLC compute $1 \,..\, 3 \;\times \{\text{"}a\text{"}, \text{"}b\text{"}\}$.

It's parsed as $1 \,..\, (3 \times \{\text{"}a\text{"}, \text{"}b\text{"}\})$.

Stop the video and let TLC compute this 6-element set.

Now see what happens if you remove the parentheses.

You get an error because this is how that expression is parsed.

The Cartesian Product

For any sets S, T, and U

$$S \times T = \{\langle a, b\rangle : a \in S,\ b \in T\}$$

$$S \times T \times U = \{\langle a, b, c\rangle : a \in S,\ b \in T\ \ c \in U\}$$

The cross product of three sets is the obvious set of triples.

The Cartesian Product

For any sets S, T, and U

$$S \times T = \{\langle a, b\rangle : a \in S,\ b \in T\}$$

$$S \times T \times U = \{\langle a, b, c\rangle : a \in S,\ b \in T\ \ c \in U\}$$

$$\vdots$$

The cross product of three sets is the obvious set of triples.

And so on for the cross product of any number of sets.

WHAT THE PROTOCOL SHOULD ACCOMPLISH

In the Alternating Bit protocol

In the Alternating Bit protocol

In the AB protocol

In the Alternating Bit protocol We abbreviate “alternating bit” as A-B.

In the AB protocol a sender A sends a sequence of data items to a receiver B.

In the Alternating Bit protocol We abbreviate “alternating bit” as A-B.

In the AB protocol, a sender A sends a sequence of data items to a receiver B.

In the AB protocol a sender A sends a sequence of strings to a receiver B.

In the Alternating Bit protocol We abbreviate “alternating bit” as A-B.

In the AB protocol, a sender A sends a sequence of data items to a receiver B.

Let’s suppose for now that those data items are strings.

In the AB protocol a sender A sends a sequence of strings to a receiver B.

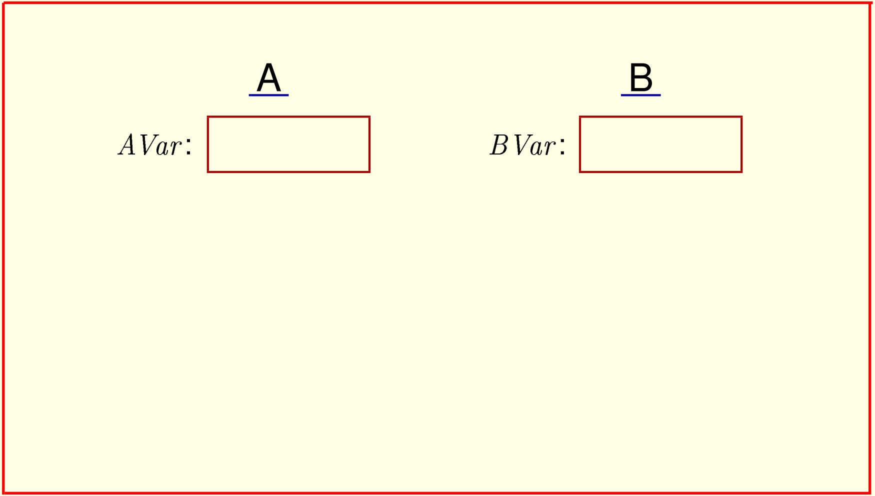
Here's an obvious way to represent this.

In the Alternating Bit protocol We abbreviate "alternating bit" as A-B.

In the AB protocol, a sender A sends a sequence of data items to a receiver B.

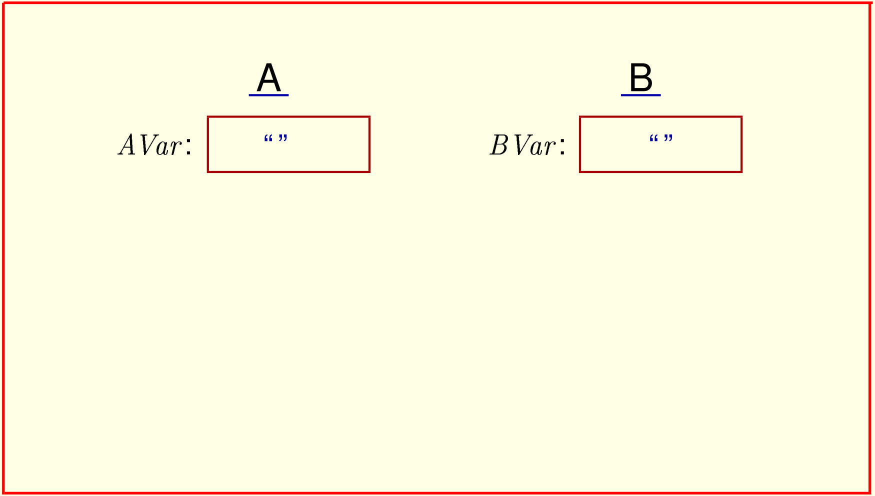
Let's suppose for now that those data items are strings.

Here's an obvious way to represent this.



The states of A and B are represented by two variables, $AVar$ and $BVar$.

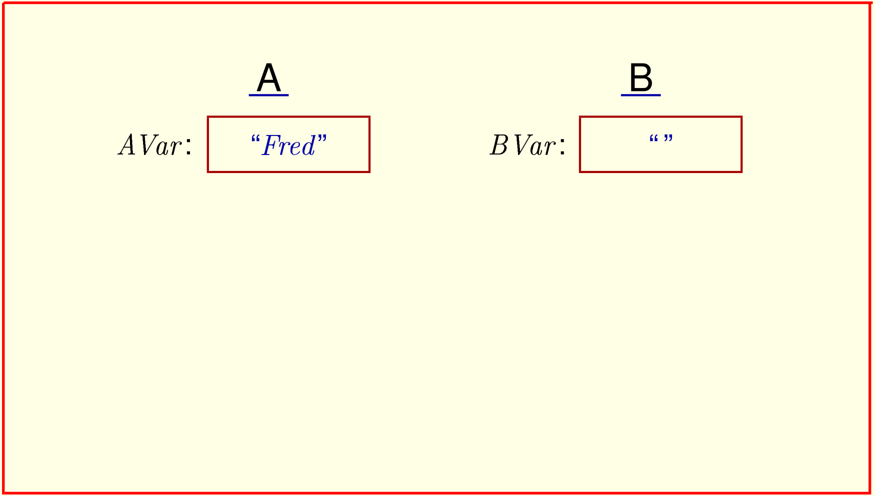
They're initially set to some default value,



The states of A and B are represented by two variables, $AVar$ and $BVar$.

They're initially set to some default value, say the empty string.

If A wants to send a string, say the string Fred,

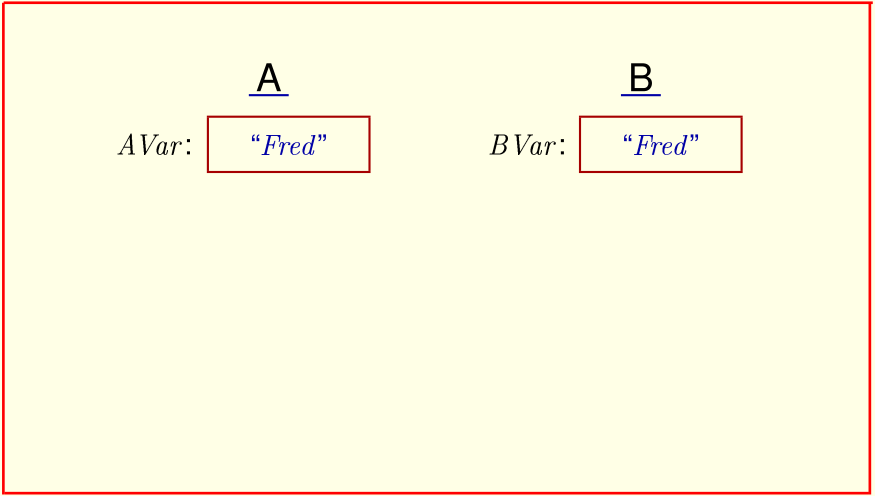


The states of A and B are represented by two variables, $AVar$ and $BVar$.

They're initially set to some default value, say the empty string.

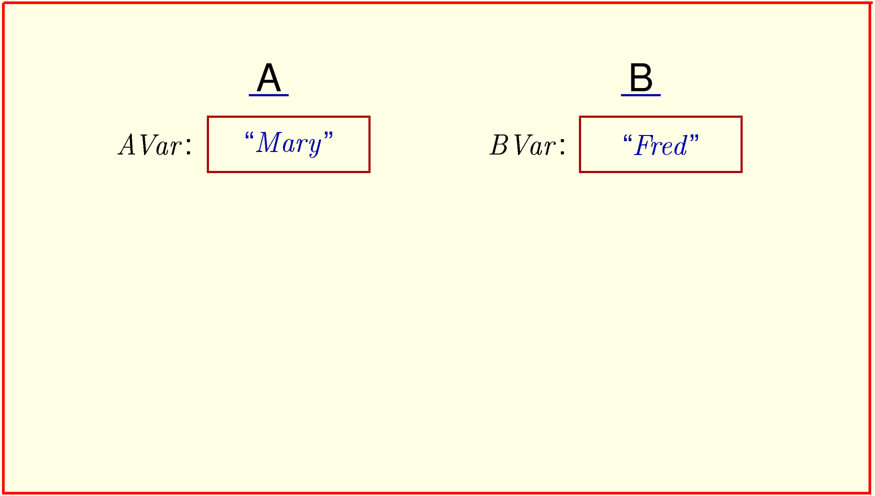
If A wants to send a string, say the string Fred,
it sets $AVar$ to that value.

B must eventually receive that string



by setting $BVar$ equal to it.

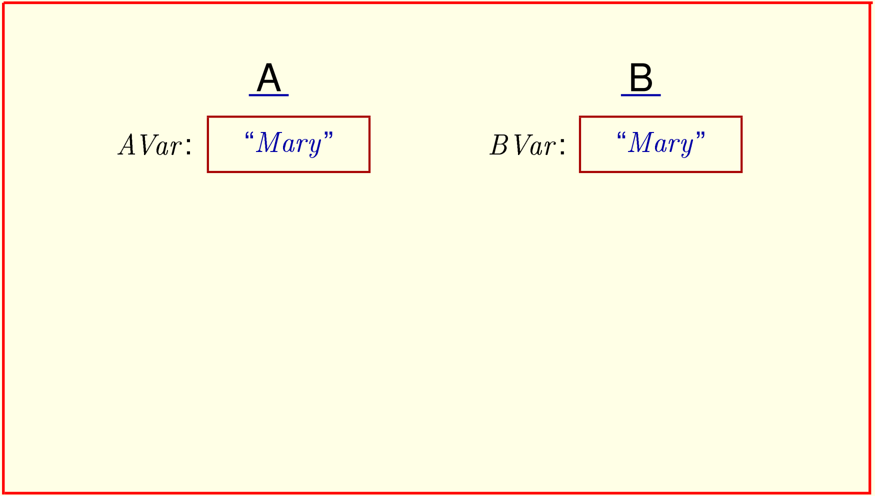
A chooses a new value, say $Mary$



by setting $BVar$ equal to it.

A chooses a new value, say $Mary$

which it sends and B receives.

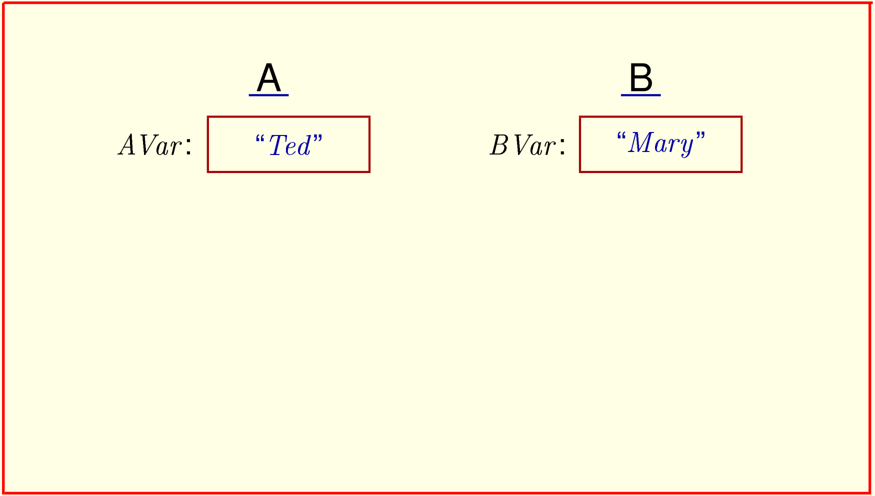


by setting $BVar$ equal to it.

A chooses a new value, say $Mary$

which it sends and B receives.

and so on.

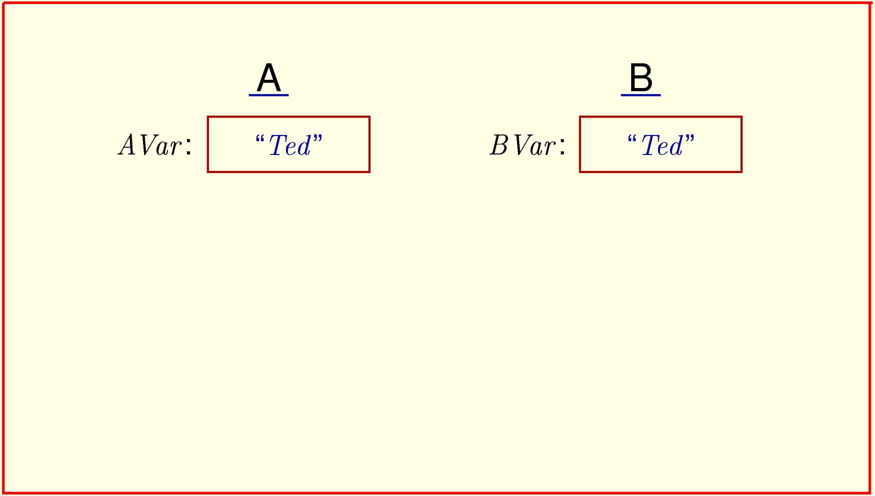


by setting $BVar$ equal to it.

A chooses a new value, say $Mary$

which it sends and B receives.

and so on.

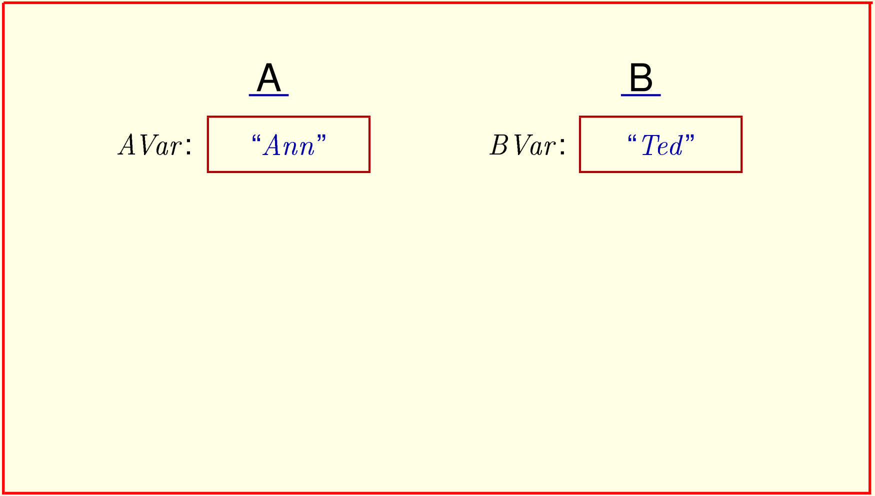


by setting $BVar$ equal to it.

A chooses a new value, say $Mary$

which it sends and B receives.

and so on.

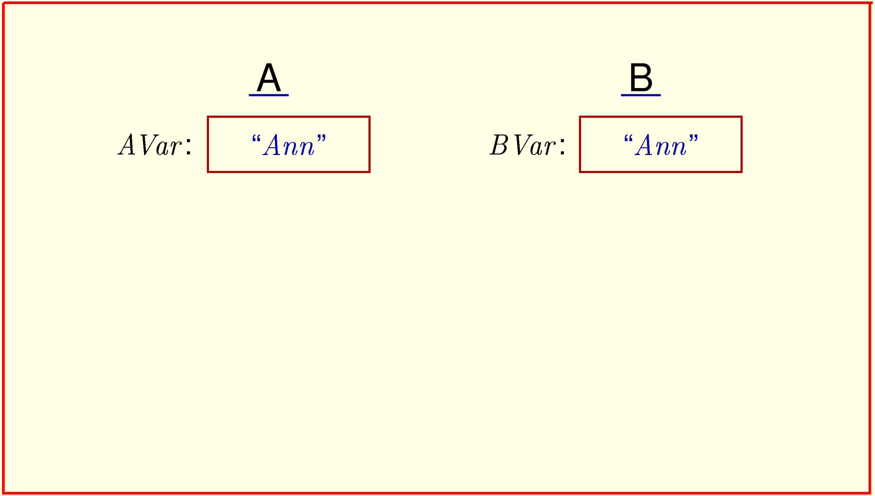


by setting $BVar$ equal to it.

A chooses a new value, say $Mary$

which it sends and B receives.

and so on.

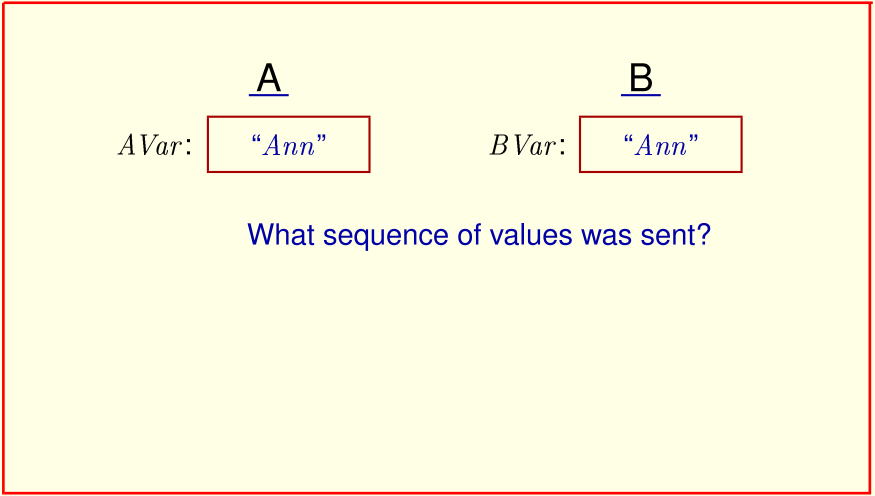


by setting $BVar$ equal to it.

A chooses a new value, say $Mary$

which it sends and B receives.

and so on.



What sequence of values was sent?

A **B**

$AVar$: "Ann" $BVar$: "Ann"

What sequence of values was sent?

"$Fred$", "$Mary$", "Ted", "Ann"

What sequence of values was sent?

Obviously, the sequence Fred, Mary, Ted, and Ann.

A

B

$AVar$: "Ann"

$BVar$: "Ann"

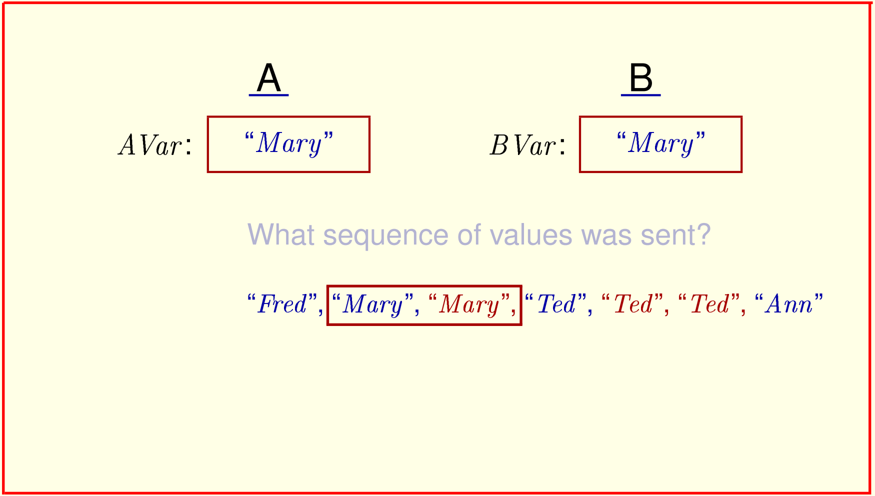
What sequence of values was sent?

"$Fred$", "$Mary$", "$Mary$", "Ted", "Ted", "Ted", "Ann"

What sequence of values was sent?

Obviously, the sequence Fred, Mary, Ted, and Ann.

No, it was actually this sequence.



What sequence of values was sent?

Obviously, the sequence Fred, Mary, Ted, and Ann.

No, it was actually this sequence.

Didn't you see $AVar$ change from $Mary$ to $Mary$, and $BVar$ do the same thing?

[slide 81]

A | **B**

$AVar$: "*Ted*" | $BVar$: "*Ted*"

What sequence of values was sent?

"*Fred*", "*Mary*", "*Mary*", "*Ted*", "*Ted*", "*Ted*", "*Ann*"

And didn't you see them changing from Ted to Ted twice?

A

B

$AVar$: "Ted"

$BVar$: "Ted"

What sequence of values was sent?

"$Fred$", "$Mary$", "Ted", "Ann"

And didn't you see them changing from Ted to Ted twice?

Of course not. A value can't have been sent if nothing changed.

A

B

$AVar$: “Ted”

$BVar$: “Ted”

What sequence of values was sent?

“$Fred$”, “$Mary$”, “$Mary$”, “Ted”, “Ted”, “Ted”, “Ann”

How can this sequence of values be sent?

And didn’t you see them changing from Ted to Ted twice?

Of course not. A value can’t have been sent if nothing changed.

How can we let the same value be sent twice in a row?

A	B
$AVar$: "*Ted*"	$BVar$: "*Ted*"

What sequence of values was sent?

"*Fred*", "*Mary*", "*Mary*", "*Ted*", "*Ted*", "*Ted*", "*Ann*"

How can this sequence of values be sent?

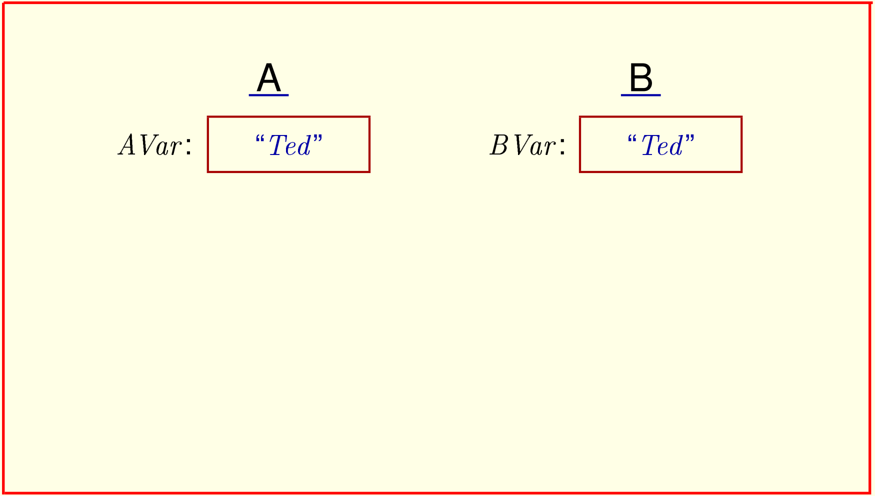
With additional state that can change.

And didn't you see them changing from Ted to Ted twice?

Of course not. A value can't have been sent if nothing changed.

How can we let the same value be sent twice in a row?

By adding something to the state that can change when the value is sent for the second time.



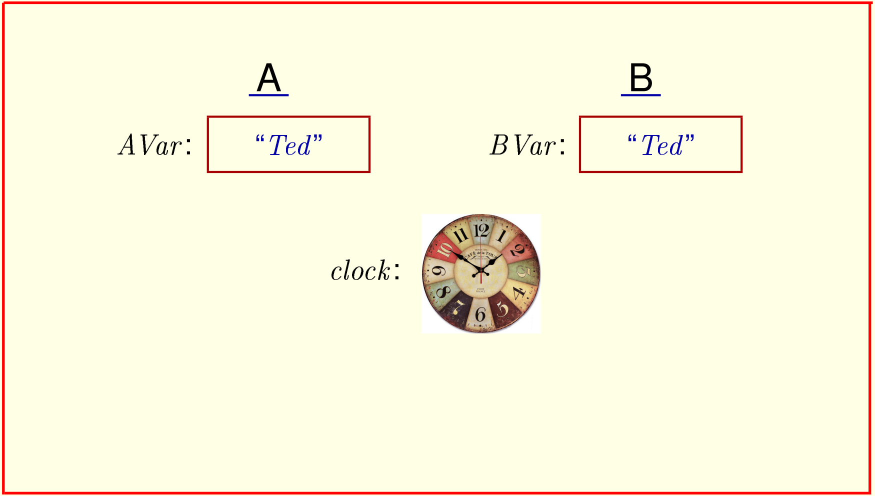
We could add a variable $clock$

A B

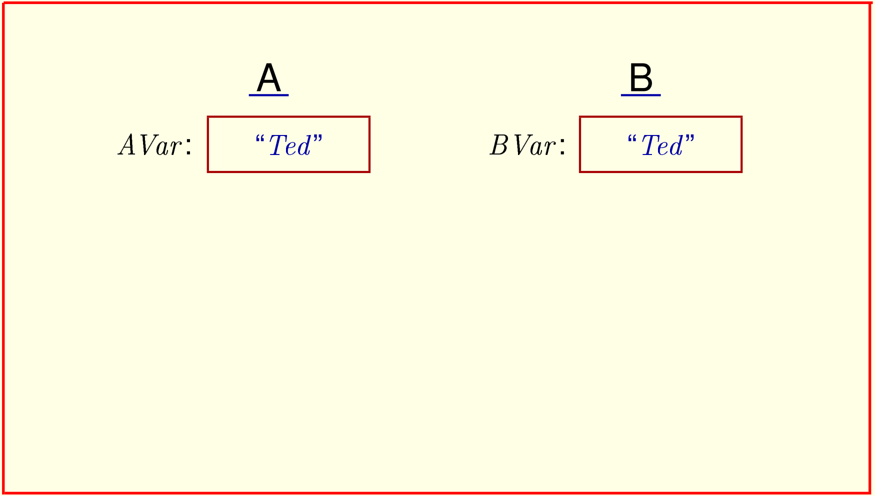
$AVar$: "Ted" $BVar$: "Ted"

$clock$: 

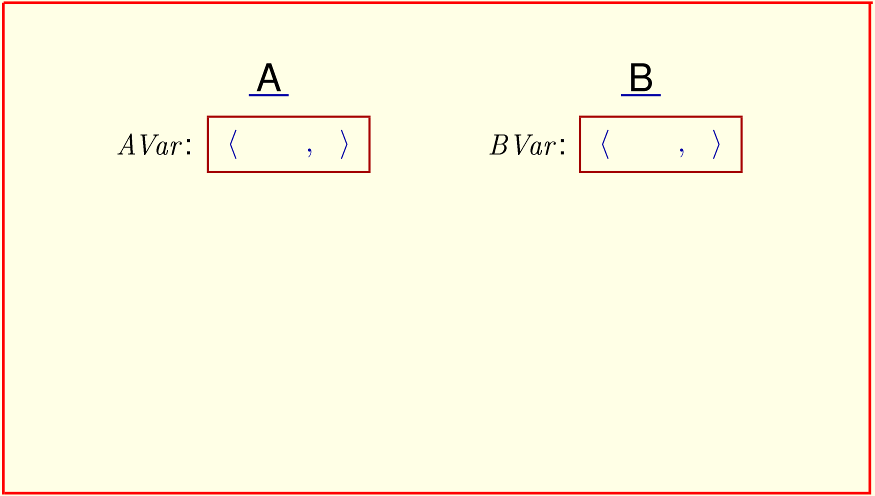
We could add a variable $clock$ And let the value in $AVar$ be sent again when



We could add a variable $clock$ And let the value in $AVar$ be sent again when the value of $clock$ changes.



We could add a variable *clock* And let the value in *AVar* be sent again when the value of *clock* changes. But we'll take a different approach



We could add a variable $clock$ And let the value in $AVar$ be sent again when the value of $clock$ changes. But we'll take a different approach

We'll let the values of $AVar$ and $BVar$ be ordered pairs,

A

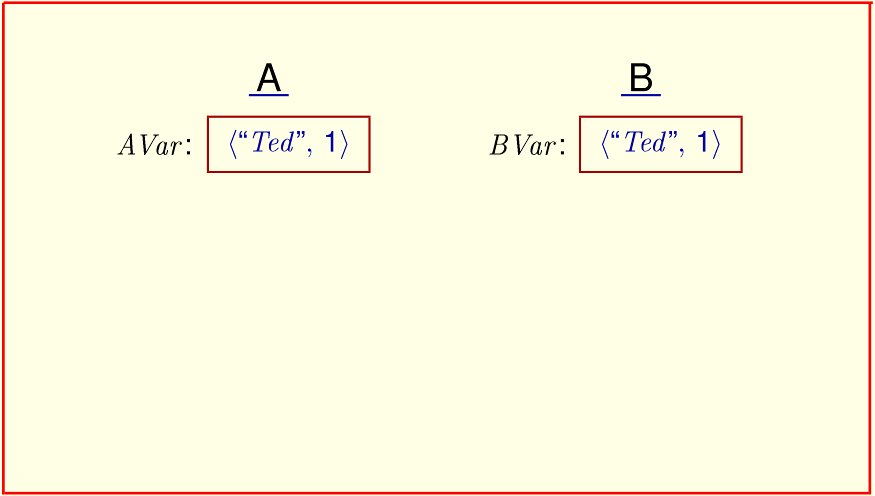
$AVar$: $\langle$"Ted", $\rangle$

B

$BVar$: $\langle$"Ted", $\rangle$

We could add a variable $clock$ And let the value in $AVar$ be sent again when the value of $clock$ changes. But we'll take a different approach

We'll let the values of $AVar$ and $BVar$ be ordered pairs, the first element of which is the value being sent



We could add a variable $clock$ And let the value in $AVar$ be sent again when the value of $clock$ changes. But we'll take a different approach

We'll let the values of $AVar$ and $BVar$ be ordered pairs, the first element of which is the value being sent and the second element is a one-bit value

A

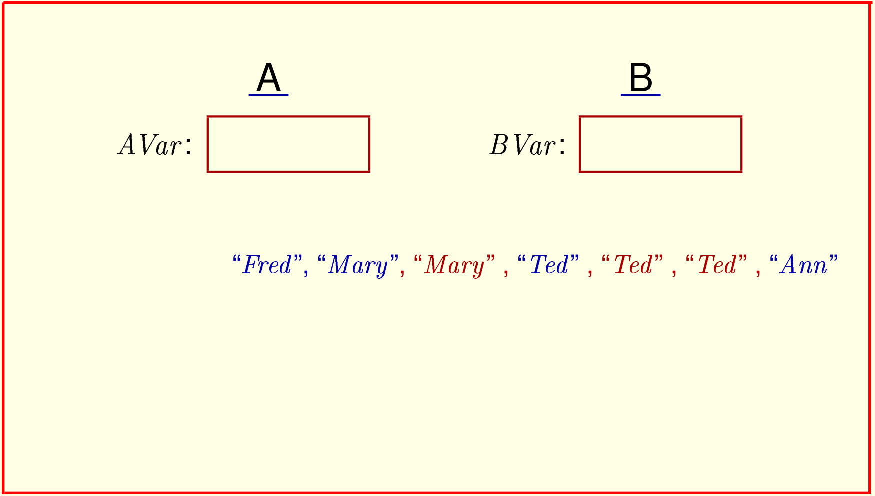
$AVar$: $\langle$“Ann”, 0$\rangle$

B

$BVar$: $\langle$“Ted”, 1$\rangle$

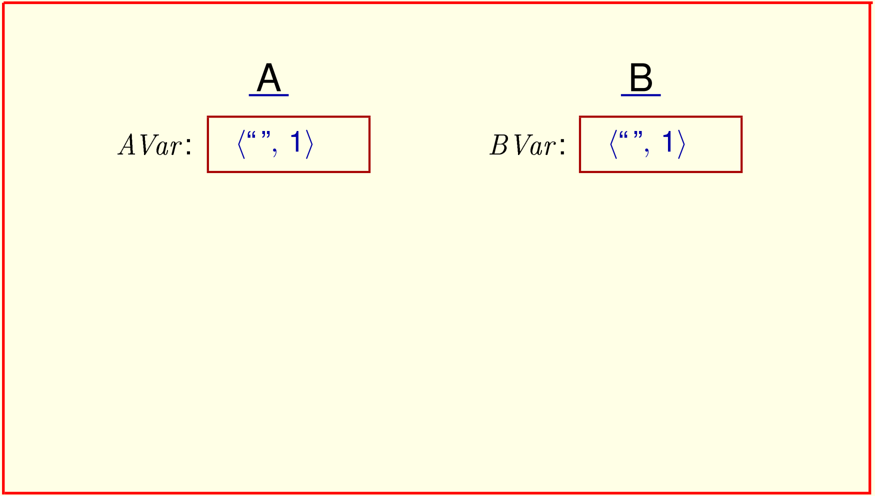
We could add a variable $clock$ And let the value in $AVar$ be sent again when the value of $clock$ changes. But we’ll take a different approach

We’ll let the values of $AVar$ and $BVar$ be ordered pairs, the first element of which is the value being sent and the second element is a one-bit value that is changed when a value is chosen. So we can send this sequence of values



We could add a variable $clock$ And let the value in $AVar$ be sent again when the value of $clock$ changes. But we'll take a different approach

We'll let the values of $AVar$ and $BVar$ be ordered pairs, the first element of which is the value being sent and the second element is a one-bit value that is changed when a value is chosen. So we can send this sequence of values Like this [15 $\times$ (1 per second) pause]



We could add a variable $clock$ And let the value in $AVar$ be sent again when the value of $clock$ changes. But we'll take a different approach

We'll let the values of $AVar$ and $BVar$ be ordered pairs, the first element of which is the value being sent and the second element is a one-bit value that is changed when a value is chosen. So we can send this sequence of values Like this [15 $\times$ (1 per second) pause]

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

We could add a variable $clock$ And let the value in $AVar$ be sent again when the value of $clock$ changes. But we'll take a different approach

We'll let the values of $AVar$ and $BVar$ be ordered pairs, the first element of which is the value being sent and the second element is a one-bit value that is changed when a value is chosen. So we can send this sequence of values

Like this [15 $\times$ (1 per second) pause]

A

B

$AVar$: $\langle$"*Fred*", 0$\rangle$

$BVar$: $\langle$"*Fred*", 0$\rangle$

"*Fred*"

We could add a variable $clock$ And let the value in $AVar$ be sent again when the value of $clock$ changes. But we'll take a different approach

We'll let the values of $AVar$ and $BVar$ be ordered pairs, the first element of which is the value being sent and the second element is a one-bit value that is changed when a value is chosen. So we can send this sequence of values Like this [15 $\times$ (1 per second) pause]

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

"$Fred$"

We could add a variable $clock$ And let the value in $AVar$ be sent again when the value of $clock$ changes. But we'll take a different approach

We'll let the values of $AVar$ and $BVar$ be ordered pairs, the first element of which is the value being sent and the second element is a one-bit value that is changed when a value is chosen. So we can send this sequence of values

Like this [15 $\times$ (1 per second) pause]

A **B**

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

"$Fred$", "$Mary$"

We could add a variable $clock$ And let the value in $AVar$ be sent again when the value of $clock$ changes. But we'll take a different approach

We'll let the values of $AVar$ and $BVar$ be ordered pairs, the first element of which is the value being sent and the second element is a one-bit value that is changed when a value is chosen. So we can send this sequence of values Like this [15 $\times$ (1 per second) pause]

A

$AVar$: $\langle$"$Mary$", 0$\rangle$

B

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

"$Fred$", "$Mary$"

We could add a variable $clock$ And let the value in $AVar$ be sent again when the value of $clock$ changes. But we'll take a different approach

We'll let the values of $AVar$ and $BVar$ be ordered pairs, the first element of which is the value being sent and the second element is a one-bit value that is changed when a value is chosen. So we can send this sequence of values Like this [15 $\times$ (1 per second) pause]

A **B**

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

"$Fred$", "$Mary$", "$Mary$"

We could add a variable $clock$ And let the value in $AVar$ be sent again when the value of $clock$ changes. But we'll take a different approach

We'll let the values of $AVar$ and $BVar$ be ordered pairs, the first element of which is the value being sent and the second element is a one-bit value that is changed when a value is chosen. So we can send this sequence of values Like this [15 $\times$ (1 per second) pause]

A **B**

$AVar$: ⟨"Ted", 1⟩ $BVar$: ⟨"$Mary$", 0⟩

"$Fred$", "$Mary$", "$Mary$"

We could add a variable $clock$ And let the value in $AVar$ be sent again when the value of $clock$ changes. But we'll take a different approach

We'll let the values of $AVar$ and $BVar$ be ordered pairs, the first element of which is the value being sent and the second element is a one-bit value that is changed when a value is chosen. So we can send this sequence of values Like this [15 $\times$ (1 per second) pause]

A

B

$AVar$: $\langle$"Ted", 1$\rangle$

$BVar$: $\langle$"Ted", 1$\rangle$

"$Fred$", "$Mary$", "$Mary$" , "Ted"

We could add a variable $clock$ And let the value in $AVar$ be sent again when the value of $clock$ changes. But we'll take a different approach

We'll let the values of $AVar$ and $BVar$ be ordered pairs, the first element of which is the value being sent and the second element is a one-bit value that is changed when a value is chosen. So we can send this sequence of values Like this [15 $\times$ (1 per second) pause]

<u>A</u> <u>B</u>

$AVar$: $\langle$"Ted", 0$\rangle$ $BVar$: $\langle$"Ted", 1$\rangle$

"$Fred$", "$Mary$", "$Mary$" , "Ted"

We could add a variable $clock$ And let the value in $AVar$ be sent again when the value of $clock$ changes. But we'll take a different approach

We'll let the values of $AVar$ and $BVar$ be ordered pairs, the first element of which is the value being sent and the second element is a one-bit value that is changed when a value is chosen. So we can send this sequence of values Like this [15 $\times$ (1 per second) pause]

A

B

$AVar$: $\langle$"Ted", 0$\rangle$

$BVar$: $\langle$"Ted", 0$\rangle$

"$Fred$", "$Mary$", "$Mary$" , "Ted" , "Ted"

We could add a variable $clock$ And let the value in $AVar$ be sent again when the value of $clock$ changes. But we'll take a different approach

We'll let the values of $AVar$ and $BVar$ be ordered pairs, the first element of which is the value being sent and the second element is a one-bit value that is changed when a value is chosen. So we can send this sequence of values Like this [15 $\times$ (1 per second) pause]

A **B**

$AVar$: $\langle$ "Ted", 1 $\rangle$ $BVar$: $\langle$ "Ted", 0 $\rangle$

"$Fred$", "$Mary$", "$Mary$" , "Ted" , "Ted"

We could add a variable $clock$ And let the value in $AVar$ be sent again when the value of $clock$ changes. But we'll take a different approach

We'll let the values of $AVar$ and $BVar$ be ordered pairs, the first element of which is the value being sent and the second element is a one-bit value that is changed when a value is chosen. So we can send this sequence of values Like this [15 $\times$ (1 per second) pause]

A

B

$AVar$: $\langle$“*Ted*”, 1$\rangle$

$BVar$: $\langle$“*Ted*”, 1$\rangle$

“*Fred*”, “*Mary*”, “*Mary*” , “*Ted*” , “*Ted*” , “*Ted*”

We could add a variable $clock$ And let the value in $AVar$ be sent again when the value of $clock$ changes. But we'll take a different approach

We'll let the values of $AVar$ and $BVar$ be ordered pairs, the first element of which is the value being sent and the second element is a one-bit value that is changed when a value is chosen. So we can send this sequence of values Like this [15 $\times$ (1 per second) pause]

A B

$AVar$: ⟨"Ann", 0⟩ $BVar$: ⟨"Ted", 1⟩

"$Fred$", "$Mary$", "$Mary$" , "Ted" , "Ted" , "Ted"

We could add a variable $clock$ And let the value in $AVar$ be sent again when the value of $clock$ changes. But we'll take a different approach

We'll let the values of $AVar$ and $BVar$ be ordered pairs, the first element of which is the value being sent and the second element is a one-bit value that is changed when a value is chosen. So we can send this sequence of values Like this [15 $\times$ (1 per second) pause]

[slide 108]

A **B**

$AVar$: ⟨"Ann", 0⟩ $BVar$: ⟨"Ann", 0⟩

"$Fred$", "$Mary$", "$Mary$" , "Ted" , "Ted" , "Ted" , "Ann"

We could add a variable $clock$ And let the value in $AVar$ be sent again when the value of $clock$ changes. But we'll take a different approach

We'll let the values of $AVar$ and $BVar$ be ordered pairs, the first element of which is the value being sent and the second element is a one-bit value that is changed when a value is chosen. So we can send this sequence of values Like this [15 $\times$ (1 per second) pause]

THE HIGH LEVEL SPEC

MODULE $ABSpec$

The spec of what the AB protocol is supposed to accomplish is in a module named $ABSpec$.

MODULE *ABSpec*

EXTENDS *Integers*

The spec of what the AB protocol is supposed to accomplish is in a module named $ABSpec$.

As usual, it extends the $Integers$ module.

MODULE *ABSpec*

EXTENDS *Integers*

CONSTANT *Data* The set of values that can be transmitted.

The spec of what the AB protocol is supposed to accomplish is in a module named $ABSpec$.

As usual, it extends the $Integers$ module.

And it declares the constant $Data$, which is the set of all values that can be transmitted.

MODULE *ABSpec*

EXTENDS *Integers*

CONSTANT *Data*

VARIABLES *AVar*, *BVar*

We declare the spec's two variables

MODULE *ABSpec*

EXTENDS *Integers*

CONSTANT *Data*

VARIABLES *AVar*, *BVar*

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

AVar and *BVar* are
⟨data, 0 or 1⟩ pairs.

We declare the spec's two variables and the type correctness invariant asserting that both variables are pairs whose first element is in the set $Data$, and whose second element is either zero or one.

$$\text{MODULE } ABSpec$$

$\text{EXTENDS } Integers$

$\text{CONSTANT } Data$

$\text{VARIABLES } AVar, BVar$

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

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

We declare the spec's two variables and the type correctness invariant asserting that both variables are pairs whose first element is in the set $Data$, and whose second element is either zero or one.

It's convenient to define $vars$ to be the tuple of all variables.

$$Init \triangleq$$

The initial-state formula asserts that

$$Init \triangleq \land AVar \in Data \times \{1\}$$

$AVar$ can equal $\langle$any element of $Data$, 1$\rangle$.

The initial-state formula asserts that

$AVar$ can equal any pair whose first element is in $Data$ and whose second element is 1.

$$Init \triangleq \land AVar \in Data \times \{1\}$$
$$\qquad\quad \land BVar = AVar$$

The initial-state formula asserts that $AVar$ can equal any pair whose first element is in $Data$ and whose second element is 1.

And $BVar$ must equal $AVar$.

$A \triangleq$ A chooses a new value to send.

We're going to define A to be the action in which the sender A chooses a new value to send.

$A \triangleq$

$B \triangleq$ B receives a value.

We're going to define A to be the action in which the sender A chooses a new value to send.

And we're going to define B to be the action in which the receiver B receives a value.

$$A \triangleq$$

$$B \triangleq$$

$$Next \triangleq A \lor B$$

We're going to define A to be the action in which the sender A chooses a new value to send.

And we're going to define B to be the action in which the receiver B receives a value.

The next-state action permits an A step or a B step.

$$A \triangleq$$

$$B \triangleq$$

$$Next \triangleq A \lor B$$

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

And here's the complete spec.

$$A \triangleq$$

$$B \triangleq$$

$$Next \triangleq A \vee B$$

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

And here's the complete spec.

Remember that $vars$ was defined to be the tuple of all variables.

$A \triangleq$

$B \triangleq$

$Next \triangleq A \lor B$

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

Now for the definition of action A.

$$A \triangleq \land AVar = BVar$$

$$B \triangleq$$

$$Next \triangleq A \lor B$$

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

Now for the definition of action A.

The action can be taken when $AVar$ equals $BVar$.

$$A \triangleq \land\ AVar = BVar$$
$$\qquad \land\ \exists\, d \in Data : AVar' = \langle d, \qquad\qquad \rangle$$

$$B \triangleq$$

$$Next \triangleq A \lor B$$

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

Now for the definition of action A.

The action can be taken when $AVar$ equals $BVar$.

The new value of $AVar$ is a pair that can have any data value as its first component

$$A \triangleq \land AVar = BVar$$
$$\land \exists d \in Data : AVar' = \langle d, \boxed{} \rangle$$

the complement of $AVar[2]$

$$B \triangleq$$

$$Next \triangleq A \lor B$$

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

Now for the definition of action A.

The action can be taken when $AVar$ equals $BVar$.

The new value of $AVar$ is a pair that can have any data value as its first component and whose second component is the complement of $AVar$'s original second component.

$$A \triangleq \land\ AVar = BVar$$
$$\qquad \land\ \exists\, d \in Data : AVar' = \langle d, \boxed{} \rangle$$

the complement of $AVar[2]$

IF $AVar[2] = 0$ THEN 1
ELSE 0

$$B \triangleq$$

$$Next \triangleq A \lor B$$

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

A programmer might write that complement this way.

$$A \triangleq \land\ AVar = BVar$$
$$\qquad \land\ \exists\, d \in Data : AVar' = \langle d, \boxed{} \rangle$$

the complement of $AVar$[2]

IF $AVar[2] = 0$ THEN 1
ELSE 0

$(AVar[2] + 1)$ % 2

$$B \triangleq$$

$$Next \triangleq A \lor B$$

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

A programmer might write that complement this way.

A mathematician might write it this way

$A \triangleq \land AVar = BVar$
$\qquad \land \exists d \in Data : AVar' = \langle d, \boxed{} \rangle$

the complement of $AVar[2]$

IF $AVar[2] = 0$ THEN 1
ELSE 0

$(AVar[2] + 1) \boxed{\%} 2$

$B \triangleq$

$Next \triangleq A \lor B$

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

A programmer might write that complement this way.

A mathematician might write it this way where percent is the modulus operator used in many programming languages.

TLC will show you how it's defined for negative arguments.

$$A \triangleq \land AVar = BVar$$
$$\qquad \land \exists d \in Data : AVar' = \langle d, \boxed{1 - AVar[2]} \rangle$$

the complement of $AVar[2]$

$$B \triangleq$$

$$Next \triangleq A \lor B$$

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

We'll write it like this, the way a bright child might.

$$A \triangleq \land AVar = BVar$$
$$\quad \land \exists d \in Data : AVar' = \langle d, 1 - AVar[2] \rangle$$

$$B \triangleq$$

$$Next \triangleq A \lor B$$

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

We'll write it like this, the way a bright child might.

$$A \triangleq \land AVar = BVar$$
$$\quad \land \exists\, d \in Data : AVar' = \langle d, 1 - AVar[2] \rangle$$
$$\quad \land BVar' = BVar$$

$$B \triangleq$$

$$Next \triangleq A \lor B$$

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

We'll write it like this, the way a bright child might.

The action leaves $BVar$ unchanged.

$$A \triangleq \land AVar = BVar$$
$$\land \exists d \in Data : AVar' = \langle d, 1 - AVar[2] \rangle$$
$$\land BVar' = BVar$$

$$B \triangleq$$

$$Next \triangleq A \lor B$$

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

We now define action B, in which a message is received.

$$A \triangleq \land AVar = BVar$$
$$\land \exists d \in Data : AVar' = \langle d, 1 - AVar[2] \rangle$$
$$\land BVar' = BVar$$

$$B \triangleq \land AVar \neq BVar$$

$$Next \triangleq A \lor B$$

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

We now define action B, in which a message is received.

A B step can be taken when the values of $AVar$ and $BVar$ are unequal.

$$
\begin{aligned}
A \;\triangleq\;& \land\ AVar = BVar \\
& \land\ \exists\, d \in Data : AVar' = \langle d,\ 1 - AVar[2] \rangle \\
& \land\ BVar' = BVar
\end{aligned}
$$

$$
\begin{aligned}
B \;\triangleq\;& \land\ AVar \neq BVar \\
& \land\ BVar' = AVar
\end{aligned}
$$

$$
Next \;\triangleq\; A \lor B
$$

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

We now define action B, in which a message is received.

A B step can be taken when the values of $AVar$ and $BVar$ are unequal.

The step sets the value of $BVar$ to that of $AVar$

[slide 137]

$$
\begin{aligned}
A \;\triangleq\;& \land\ AVar = BVar \\
& \land\ \exists\, d \in Data : AVar' = \langle d,\ 1 - AVar[2] \rangle \\
& \land\ BVar' = BVar
\end{aligned}
$$

$$
\begin{aligned}
B \;\triangleq\;& \land\ AVar \neq BVar \\
& \land\ BVar' = AVar \\
& \land\ AVar' = AVar
\end{aligned}
$$

$$
Next \;\triangleq\; A \lor B
$$

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

We now define action B, in which a message is received.

A B step can be taken when the values of $AVar$ and $BVar$ are unequal.

The step sets the value of $BVar$ to that of $AVar$

And it leaves $AVar$ unchanged.

$$
\begin{array}{lcl}
A & \triangleq & \land\ AVar = BVar \\
 & & \land\ \exists\, d \in Data : AVar' = \langle d, 1 - AVar[2] \rangle \\
 & & \land\ BVar' = BVar \\[1em]
B & \triangleq & \land\ AVar \neq BVar \\
 & & \land\ BVar' = AVar \\
 & & \land\ AVar' = AVar \\[1em]
Next & \triangleq & A \lor B \\[1em]
Spec & \triangleq & Init \land \Box[Next]_{vars}
\end{array}
$$

This completes the definition of the specification $Spec$.

– Stop the video.

– Download $ABSpec$.

– Open it in the Toolbox.

Stop the video now to download $ABSpec$ and open it in the Toolbox.

– Stop the video.

– Download $ABSpec$.

– Open it in the Toolbox.

– Create a model that substitutes a small set of model values for $Data$.

Stop the video now to download $ABSpec$ and open it in the Toolbox.

Create a model that substitutes a small set of model values, perhaps containing 3 values, for $Data$.

– Stop the video.

– Download $ABSpec$.

– Open it in the Toolbox.

– Create a model that substitutes a small set of model values for $Data$.

– Run TLC on the model to check invariance of $TypeOK$.

Stop the video now to download $ABSpec$ and open it in the Toolbox.

Create a model that substitutes a small set of model values, perhaps containing 3 values, for $Data$.

And run TLC on the model to check that $TypeOK$ is an invariant.

Type correctness doesn't mean
the spec is correct.

Type correctness doesn't mean that a specification is correct.

Type correctness doesn't mean
the spec is correct.

To find errors, check that formulas
which should be invariants are.

Type correctness doesn't mean that a specification is correct.

To find errors, we should check that formulas which should be invariants actually are invariants.

Type correctness doesn't mean the spec is correct.

To find errors, check that formulas which should be invariants are.

Here's one such formula defined in $ABSpec$:

$$Inv \triangleq (AVar[2] = BVar[2]) \Rightarrow (AVar = BVar)$$

Type correctness doesn't mean that a specification is correct.

To find errors, we should check that formulas which should be invariants actually are invariants.

Here's one such formula defined in the $ABSpec$ module.

Type correctness doesn't mean the spec is correct.

To find errors, check that formulas which should be invariants are.

Here's one such formula defined in $ABSpec$:

$$Inv \triangleq (AVar[2] = BVar[2]) \Rightarrow (AVar = BVar)$$

Convince yourself that it should be an invariant and have TLC check that it is.

Convince yourself that it should be an invariant and have TLC check that it actually is.

Formula $Spec$ asserts what may happen.

Like all the specifications we've written so far, formula $Spec$ asserts only what *may* happen.

Formula $Spec$ asserts what may happen.

We now specify what must happen.

Like all the specifications we've written so far, formula $Spec$ asserts only what *may* happen.

We will now specify what *must* happen.

Formula $Spec$ asserts what may happen.

We now specify what must happen.

Exactly what do may and must mean?

Like all the specifications we've written so far, formula $Spec$ asserts only what *may* happen.

We will now specify what *must* happen.

But first, we look at exactly what *may* and *must* mean.

SAFETY AND LIVENESS

Safety Formula

A safety formula is a temporal formula

Safety Formula

Asserts what may happen.

A safety formula is a temporal formula that asserts only what may happen.

More precisely, it's a temporal formula that

Safety Formula

Asserts what may happen.

Any behavior that violates it

A safety formula is a temporal formula that asserts only what may happen.

More precisely, it's a temporal formula that if a behavior violates it – meaning that if the formula is false on the behavior,

Safety Formula

Asserts what may happen.

Any behavior that violates it does so at some point.

A safety formula is a temporal formula that asserts only what may happen.

More precisely, it's a temporal formula that if a behavior violates it – meaning that if the formula is false on the behavior, then that violation occurs at some particular point in the behavior.

Safety Formula

Asserts what may happen.

Any behavior that violates it does so at some point.

Nothing past that point makes any difference.

A safety formula is a temporal formula that asserts only what may happen.

More precisely, it's a temporal formula that if a behavior violates it – meaning that if the formula is false on the behavior, then that violation occurs at some particular point in the behavior.

And nothing in the behavior past that point can prevent the violation.

Safety Formula

Asserts what may happen.

Any behavior that violates it does so at some point.

Example: $Init \ \wedge \ \Box[Next]_{vars}$ can be violated either:

For example the kind of specification we've been writing can be violated by a behavior only if either

Safety Formula

Asserts what may happen.

Any behavior that violates it does so at some point.

Example: $\boxed{Init} \land \Box[Next]_{vars}$ can be violated either:

at an initial state not satisfying $Init$

For example the kind of specification we've been writing can be violated by a behavior only if either The initial formula is false on the behavior's first state, or

Safety Formula

Asserts what may happen.

Any behavior that violates it does so at some point.

Example: $Init \ \wedge \ \Box [Next]_{vars}$ can be violated either:

at an initial state not satisfying $Init$

or at a step not satisfying $[Next]_{vars}$.

For example the kind of specification we've been writing can be violated by a behavior only if either The initial formula is false on the behavior's first state, or the action $Next$ sub vars is false on some step.

Remember that this action false on a step means

Safety Formula

Asserts what may happen.

Any behavior that violates it does so at some point.

Example: $Init \ \wedge \ \Box [Next]_{vars}$ can be violated either:

at an initial state not satisfying $Init$

or at a step not satisfying $[Next]_{vars}$.

The step neither satisfies $Next$ nor leaves $vars$ unchanged.

For example the kind of specification we've been writing can be violated by a behavior only if either The initial formula is false on the behavior's first state, or the action $Next$ sub vars is false on some step.

Remember that this action false on a step means that the step neither satisfies the action $Next$ nor leaves the tuple $vars$ of variables unchanged.

Safety Formula

Asserts what may happen.

Any behavior that violates it does so at some point.

Example: $Init \ \wedge \ \Box [Next]_{vars}$ can be violated either:

at an initial state not satisfying $Init$

or at a step not satisfying $[Next]_{vars}$.

Nothing past that point can make any difference.

And nothing in the behavior past that point of violation can cause the formula to be true.

Liveness Formula

A liveness formula is a temporal formula

Liveness Formula

Asserts what must happen.

A liveness formula is a temporal formula that asserts only what *must* happen.

More precisely, it's a temporal formula for which

Liveness Formula

Asserts what must happen.

A behavior can **not** violate it at any point.

A liveness formula is a temporal formula that asserts only what *must* happen.

More precisely, it's a temporal formula for which a behavior can *not* violate it at any particular point.

Liveness Formula

Asserts what must happen.

A behavior can **not** violate it at any point.

The rest of the behavior can always make it true.

At any point in a behavior, there's a way to complete the behavior so it satisfies the formula.

Liveness Formula

Asserts what must happen.

A behavior can **not** violate it at any point.

Example: $x = 5$ on some state of the behavior.

At any point in a behavior, there's a way to complete the behavior so it satisfies the formula.

An example of a liveness formula is one asserting that x equals 5 on some state of the behavior.

Liveness Formula

Asserts what must happen.

A behavior can **not** violate it at any point.

Example: $x = 5$ on some state of the behavior.

We'll see later how to write a formula that asserts this.

At any point in a behavior, there's a way to complete the behavior so it satisfies the formula.

An example of a liveness formula is one asserting that x equals 5 on some state of the behavior.

We'll see in a minute how to write a formula that asserts this.

Liveness Formula

Asserts what must happen.

A behavior can **not** violate it at any point.

Example: $x = 5$ on some state of the behavior.

At any point, it's always possible for a later state to satisfy $x = 5$.

At any point in a behavior, it's always possible for x to equal 5 in some later state.

Liveness Formula

Asserts what must happen.

A behavior can **not** violate it at any point.

Example: $x = 5$ on some state of the behavior.

At any point, it's always possible for a later state to satisfy $x = 5$.

At any point in a behavior, it's always possible for x to equal 5 in some later state.

So the behavior isn't violated at that point.

Liveness Formula

Asserts what must happen.

A behavior can **not** violate it at any point.

Example: $x = 5$ on some state of the behavior.

At any point, it's always possible for a later state to satisfy $x = 5$.

A behavior is any infinite sequence of states.

At any point in a behavior, it's always possible for x to equal 5 in some later state.

So the behavior isn't violated at that point.

Remember that a behavior is any infinite sequence of states. We're not talking only about behaviors that satisfy some specification.

$x = 5$ on some state of the behavior

"x equals 5 is true on some state of the behavior"

$x = 5$ on some state of the behavior

asserted by $\diamond(x = 5)$

"x equals 5 is true on some state of the behavior" is asserted by this temporal formula.

$x = 5$ on some state of the behavior

asserted by $\Diamond(x = 5)$

typed < >

"x equals 5 is true on some state of the behavior" is asserted by this temporal formula.

where this symbol is typed *less-than greater than*

$x = 5$ on some state of the behavior

asserted by $\Diamond(x = 5)$

typed < >

pronounced *eventually*

"x equals 5 is true on some state of the behavior" is asserted by this temporal formula.

where this symbol is typed *less-than greater than* and pronounced *eventually*.

$x = 5$ on some state of the behavior

asserted by $\diamond(x = 5)$

"x equals 5 is true on some state of the behavior" is asserted by this temporal formula.

where this symbol is typed *less-than greater than* and pronounced *eventually*.

The only liveness property sequential programs must satisfy is termination.

The only liveness property sequential programs must satisfy is termination.

The only liveness property sequential programs must satisfy is termination.

$$\Diamond \; Terminated$$

The only liveness property sequential programs must satisfy is termination.

It's expressed by the formula *eventually* $Terminated$, for a state formula $Terminated$ which asserts that the program has reached a terminated state.

The only liveness property sequential programs must satisfy is termination.

$\Diamond \; Terminated$

Concurrent systems can have a wide variety of liveness requirements.

The only liveness property sequential programs must satisfy is termination.

It's expressed by the formula *eventually* $Terminated$, for a state formula $Terminated$ which asserts that the program has reached a terminated state.

Concurrent systems can have a wide variety of liveness requirements.

Liveness property for $ABSpec$:

Here's a liveness property we might like the AB protocol to ensure.

Liveness property for $ABSpec$:

If $AVar = \langle$"hi"$, 0\rangle$ in some state

Here's a liveness property we might like the AB protocol to ensure.

If $AVar$ equals the pair "hi" zero in some state

Liveness property for $ABSpec$:

If $AVar = \langle \text{“}hi\text{”}, 0 \rangle$ in some state

A is sending $\langle \text{“}hi\text{”}, 0 \rangle$

Here's a liveness property we might like the AB protocol to ensure.

If $AVar$ equals the pair “hi” zero in some state
which means it's a state in which A is sending that pair to B

Liveness property for $ABSpec$:

If $AVar = \langle \text{“}hi\text{”}, 0 \rangle$ in some state

then $BVar = \langle \text{“}hi\text{”}, 0 \rangle$ in that state or a later state.

Here's a liveness property we might like the AB protocol to ensure.

If $AVar$ equals the pair “hi” zero in some state
which means it's a state in which A is sending that pair to B

then $BVar$ equals this pair either in that state or in a later state.

Liveness property for $ABSpec$:

If $AVar = \langle \text{"}hi\text{"}, 0 \rangle$ in some state

then $BVar = \langle \text{"}hi\text{"}, 0 \rangle$ in that state or a later state.

B has received $\langle \text{"}hi\text{"}, 0 \rangle$

Here's a liveness property we might like the AB protocol to ensure.

If $AVar$ equals the pair "hi" zero in some state
which means it's a state in which A is sending that pair to B

then $BVar$ equals this pair either in that state or in a later state.
which means it's a state in which B has received the pair.
That property is expressed in TLA^+

Liveness property for $ABSpec$:

If $AVar = \langle \text{“}hi\text{”}, 0 \rangle$ in some state

then $BVar = \langle \text{“}hi\text{”}, 0 \rangle$ in that state or a later state.

$$(AVar = \langle \text{“}hi\text{”}, 0 \rangle) \leadsto (BVar = \langle \text{“}hi\text{”}, 0 \rangle)$$

by this temporal formula, where

Liveness property for $ABSpec$:

If $AVar = \langle \text{“}hi\text{”}, 0 \rangle$ in some state
then $BVar = \langle \text{“}hi\text{”}, 0 \rangle$ in that state or a later state.

$$(AVar = \langle \text{“}hi\text{”}, 0 \rangle) \boxed{\leadsto} (BVar = \langle \text{“}hi\text{”}, 0 \rangle)$$

pronounced *leads to*

by this temporal formula, where

this symbol is read *leads to*

Liveness property for $ABSpec$:

If $AVar = \langle \text{"}hi\text{"}, 0\rangle$ in some state

then $BVar = \langle \text{"}hi\text{"}, 0\rangle$ in that state or a later state.

$$(AVar = \langle \text{"}hi\text{"}, 0\rangle) \leadsto (BVar = \langle \text{"}hi\text{"}, 0\rangle)$$

typed ~>

by this temporal formula, where

this symbol is read *leads to*

and is typed in ascii as *tilde greater-than*.

More generally:

In general, we'd like the AB protocol to satisfy this property:

More generally:

Any value being sent by A is eventually received by B.

In general, we'd like the AB protocol to satisfy this property:
Any value being sent by A is eventually received by B.

This is expressed as follows:

More generally:

Any value being sent by A is eventually received by B.

$$\forall\, v \in Data \times \{0, 1\}:$$

In general, we'd like the AB protocol to satisfy this property:
Any value being sent by A is eventually received by B.

This is expressed as follows:

For all v in this set,

More generally:

Any value being sent by A is eventually received by B.

$\forall\, v \in \boxed{Data \times \{0, 1\}}$:

the possible values of $AVar$ and $BVar$

In general, we'd like the AB protocol to satisfy this property:
Any value being sent by A is eventually received by B.

This is expressed as follows:

For all v in this set, which is the set of all possible values of $AVar$ and $BVar$.

More generally:

Any value being sent by A is eventually received by B.

$$\forall\, v \in Data \times \{0, 1\} : (AVar = v) \leadsto (BVar = v)$$

$AVar$ equals v leads to $BVar$ equals v.

More generally:

Any value being sent by A is eventually received by B.

$$\forall\, v \in Data \times \{0, 1\} : (AVar = v) \leadsto (BVar = v)$$

Exercise: Convince yourself that $\Diamond P$ is equivalent to $\neg \Box \neg P$.

$AVar$ equals v leads to $BVar$ equals v.

As an exercise, convince yourself that *eventually* P is equivalent to *not always not P*.

WEAK FAIRNESS

Enabled

Enabled.

Enabled

An action A is *enabled* in a state s iff there is a state t such that A is true on $s \rightarrow t$.

Let A be an arbitrary action. A is said to be *enabled* in a state s if and only if there is some next state t such that A is true on the step from s to t.

Enabled

An action A is *enabled* in a state s iff there is a state t such that A is true on $s \rightarrow t$.

Let A be an arbitrary action. A is said to be *enabled* in a state s if and only if there is some next state t such that A is true on the step from s to t.

Instead of saying A is true on the step s to t,

Enabled

An action A is *enabled* in a state s iff there is a state t such that $s \rightarrow t$ is an A step.

Let A be an arbitrary action. A is said to be *enabled* in a state s if and only if there is some next state t such that A is true on the step from s to t.

Instead of saying A is true on the step s to t, we often say that s to t is an A *step*.

Enabled

An action A is *enabled* in a state s iff there is a state t such that $s \rightarrow t$ is an A step.

For example, action A of $ABSpec$

$$
\begin{aligned}
A \triangleq\ & \land\ AVar = BVar \\
& \land\ \exists\, d \in Data : AVar' = \langle d, 1 - AVar[2] \rangle \\
& \land\ BVar' = BVar
\end{aligned}
$$

Let A be an arbitrary action. A is said to be *enabled* in a state s if and only if there is some next state t such that A is true on the step from s to t.

Instead of saying A is true on the step s to t, we often say that s to t is an A *step*.

As an example, remember action A of $ABSpec$ which is defined like this.

Enabled

An action A is *enabled* in a state s iff there is a state t such that $s \rightarrow t$ is an A step.

For example, action A of $ABSpec$

$$
\begin{aligned}
A \;\triangleq\; & \land\; AVar = BVar \\
& \land\; \exists\, d \in Data : AVar' = \langle d, 1 - AVar[2] \rangle \\
& \land\; BVar' = BVar
\end{aligned}
$$

is enabled iff

For it to be enabled

Enabled

An action A is *enabled* in a state s iff there is a state t such that $s \rightarrow t$ is an A step.

For example, action A of $ABSpec$

$$A \triangleq \land \boxed{AVar = BVar}$$
$$\land \exists d \in Data : AVar' = \langle d, 1 - AVar[2] \rangle$$
$$\land BVar' = BVar$$

is enabled iff $AVar = BVar$

For it to be enabled The first conjunct must be true.

A conjunct with no primes is an assertion about the first state, so it's an *enabling condition* for an action.

Enabled

An action A is *enabled* in a state s iff there is a state t such that $s \rightarrow t$ is an A step.

For example, action A of $ABSpec$

$$
\begin{aligned}
A \triangleq\ & \land\ AVar = BVar \\
& \land\ \exists\, d \in Data : \boxed{AVar' = \langle d, 1 - AVar[2] \rangle} \\
& \land\ \boxed{BVar' = BVar}
\end{aligned}
$$

is enabled iff $AVar = BVar$

For it to be enabled The first conjunct must be true.
A conjunct with no primes is an assertion about the first state, so it's an *enabling condition* for an action.

We can obviously choose values of $AVar$ and $BVar$ in the next state to make these two conjuncts true –

Enabled

An action A is *enabled* in a state s iff there is a state t such that $s \rightarrow t$ is an A step.

For example, action A of $ABSpec$

$$
\begin{aligned}
A \;\triangleq\; & \land\; AVar = BVar \\
& \land\; \boxed{\exists\, d \in Data} : AVar' = \langle d, 1 - AVar[2] \rangle \\
& \land\; BVar' = BVar
\end{aligned}
$$

is enabled iff $AVar = BVar$

For it to be enabled The first conjunct must be true.
A conjunct with no primes is an assertion about the first state, so it's an *enabling condition* for an action.

We can obviously choose values of $AVar$ and $BVar$ in the next state to make these two conjuncts true –
except that the second conjunct is false if $Data$ is the empty set,

Enabled

An action A is *enabled* in a state s iff there is a state t such that $s \rightarrow t$ is an A step.

For example, action A of $ABSpec$

$$\begin{aligned} A \triangleq\ & \land\ AVar = BVar \\ & \land\ \exists\, d \in Data : AVar' = \langle d, 1 - AVar[2] \rangle \\ & \land\ BVar' = BVar \end{aligned}$$

is enabled iff $AVar = BVar$ and $Data \neq \{\}$.

For it to be enabled The first conjunct must be true.
A conjunct with no primes is an assertion about the first state, so it's an *enabling condition* for an action.

We can obviously choose values of $AVar$ and $BVar$ in the next state to make these two conjuncts true –
except that the second conjunct is false if $Data$ is the empty set, so $Data$ must be non-empty for A to be enabled.

Weak fairness of action A asserts of a behavior:

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

Weak fairness of action A asserts of a behavior: that if A ever remains continuously enabled, then an A step must eventually occur.

Weak fairness of action A asserts of a behavior:

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

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

Weak fairness of action A asserts of a behavior: that if A ever remains continuously enabled, then an A step must eventually occur.

For example, suppose we have a behavior,

Weak fairness of action A asserts of a behavior:

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:

Weak fairness of action A asserts of a behavior: that if A ever remains continuously enabled, then an A step must eventually occur.

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

Weak fairness of action A asserts of a behavior:

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

Weak fairness of action A asserts of a behavior: that if A ever remains continuously enabled, then an A step must eventually occur.

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

Weak fairness of action A asserts of a behavior:

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

Weak fairness of action A asserts of a behavior: that if A ever remains continuously enabled, then an A step must eventually occur.

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

Weak fairness of action A asserts of a behavior:

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

Weak fairness of action A asserts of a behavior: that if A ever remains continuously enabled, then an A step must eventually occur.

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

Weak fairness of action A asserts of a behavior:

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

Weak fairness of action A asserts of a behavior: that if A ever remains continuously enabled, then an A step must eventually occur.

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

Weak fairness of action A asserts of a behavior:

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 true

Weak fairness of action A asserts of a behavior: that if A ever remains continuously enabled, then an A step must eventually occur.

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

Weak fairness of action A asserts of a behavior:

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 true true

Weak fairness of action A asserts of a behavior: that if A ever remains continuously enabled, then an A step must eventually occur.

For example, suppose we have a behavior, And A enabled is false in this state, then true, then false again, then true, and it remains continuously

Weak fairness of action A asserts of a behavior:

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	true	true	true		

Weak fairness of action A asserts of a behavior: that if A ever remains continuously enabled, then an A step must eventually occur.

For example, suppose we have a behavior, And A enabled is false in this state, then true, then false again, then true, and it remains continuously true

Weak fairness of action A asserts of a behavior:

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	true	true	true	true	

Weak fairness of action A asserts of a behavior: that if A ever remains continuously enabled, then an A step must eventually occur.

For example, suppose we have a behavior, And A enabled is false in this state, then true, then false again, then true, and it remains continuously true

[slide 213]

Weak fairness of action A asserts of a behavior:

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	true	true	true	true	true

Weak fairness of action A asserts of a behavior: that if A ever remains continuously enabled, then an A step must eventually occur.

For example, suppose we have a behavior, And A enabled is false in this state, then true, then false again, then true, and it remains continuously true

[slide 214]

Weak fairness of action A asserts of a behavior:

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	true	true	true	true	true

Then an A step must occur in this green part of the behavior.

After which, A need not remain enabled.

Weak fairness of action A asserts of a behavior:

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	true	true	true	true	true

Or equivalently:

A cannot remain enabled forever
without another A step occurring.

Then an A step must occur in this green part of the behavior.

After which, A need not remain enabled.

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

Weak fairness of A is written as the temporal formula $\mathrm{WF}_{vars}(A)$, where $vars$ is the tuple of all the spec's variables.

Weak fairness of A is written as this temporal formula, where $vars$ is the tuple of all the spec's variables.

Weak fairness of A is written as the temporal formula $\boxed{\mathrm{WF}_{vars}(A)}$, where $vars$ is the tuple of all the spec's variables.

`WF_vars(A)` in ASCII

Weak fairness of A is written as this temporal formula, where $vars$ is the tuple of all the spec's variables.

It's typed as WF underscore $vars$ parentheses A in ASCII.

Weak fairness of A is written as the temporal formula $\boxed{\mathrm{WF}_{vars}(A)}$, where $vars$ is the tuple of all the spec's variables.

`WF_vars(A)` in ASCII

Pronounced "WF of A"

Weak fairness of A is written as this temporal formula, where $vars$ is the tuple of all the spec's variables.

It's typed as WF underscore $vars$ parentheses A in ASCII.

It's usually read "WF of A", omitting the $vars$.

Weak fairness of A is written as the temporal formula $\mathrm{WF}_{vars}(A)$, where $vars$ is the tuple of all the spec's variables.

`WF_vars(A)` in ASCII

Pronounced "WF of A"

I'll explain the $vars$ later.

Weak fairness of A is written as this temporal formula, where $vars$ is the tuple of all the spec's variables.

It's typed as WF underscore $vars$ parentheses A in ASCII.

It's usually read "WF of A", omitting the $vars$.

I'll explain the $vars$ later.

Weak fairness of A is written as the temporal formula $\mathrm{WF}_{vars}(A)$, where $vars$ is the tuple of all the spec's variables.

It's a liveness property because it can always be made true by an A step or a state in which A is not enabled.

WF of A is a liveness property because, at any point in a behavior, it can be made true by an A step or a state in which A is not enabled.

Weak fairness of A is written as the temporal formula $\mathrm{WF}_{vars}(A)$, where $vars$ is the tuple of all the spec's variables.

It's a liveness property because it can always be made true by an A step or a state in which A is not enabled.

Later, we'll see the strong fairness formula $\mathrm{SF}_{vars}(A)$.

WF of A is a liveness property because, at any point in a behavior, it can be made true by an A step or a state in which A is not enabled.

Later, in the second part of this lecture, we'll see the strong fairness formula SF of A.

ADDING LIVENESS TO A SPEC

A spec with liveness is written

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

A TLA^{+} spec with liveness is written in this form

A spec with liveness is written

$$Init \land \Box[Next]_{vars} \land \boxed{Fairness}$$

A conjunction of $\mathrm{WF}_{vars}(A)$ and $\mathrm{SF}_{vars}(A)$ formulas

A TLA+ spec with liveness is written in this form
where $Fairness$ is a conjunction of one or more WF and/or SF of A formulas

A spec with liveness is written

$$Init \land \Box[Next]_{vars} \land \boxed{Fairness}$$

A conjunction of $\mathrm{WF}_{vars}(A)$ and $\mathrm{SF}_{vars}(A)$ formulas where each A is a subaction of $Next$.

A TLA+ spec with liveness is written in this form where $Fairness$ is a conjunction of one or more WF and/or SF of A formulas and each A is a subaction of $Next$

A spec with liveness is written

$$Init \land \Box[Next]_{vars} \land \boxed{Fairness}$$

A conjunction of $\mathrm{WF}_{vars}(A)$ and $\mathrm{SF}_{vars}(A)$ formulas where each $\boxed{A \text{ is a subaction of } Next}$.

Every A step is a $Next$ step.

A TLA$^+$ spec with liveness is written in this form where $Fairness$ is a conjunction of one or more WF and/or SF of A formulas and each A is a subaction of $Next$ Which means that every possible A step is a $Next$ step.

A spec with liveness is written

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

Module $ABSpec$ defines

$$FairSpec \triangleq Init \land \Box[Next]_{vars} \land \mathbf{WF}_{vars}(Next)$$

Module $ABSpec$ defines $FairSpec$ to be this specification,

A spec with liveness is written

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

Module $ABSpec$ defines

$$FairSpec \triangleq Init \land \Box[Next]_{vars} \land \boxed{\mathrm{WF}_{vars}(Next)}$$

Asserts that a behavior keeps taking $Next$ steps as long as $Next$ is enabled.

Module $ABSpec$ defines $FairSpec$ to be this specification, Where WF of $Next$ asserts that a behavior keeps taking $Next$ steps as long as $Next$ is enabled.

A spec with liveness is written

$$Init \wedge \Box[Next]_{vars} \wedge Fairness$$

Module $ABSpec$ defines

$$FairSpec \triangleq Init \wedge \Box[Next]_{vars} \wedge \boxed{\mathrm{WF}_{vars}(Next)}$$

Asserts that a behavior keeps taking $Next$ steps

as long as $Next$ is enabled.

not in a deadlocked / terminated state

Module $ABSpec$ defines $FairSpec$ to be this specification, Where WF of $Next$ asserts that a behavior keeps taking $Next$ steps as long as $Next$ is enabled.

Which means as long as the system is not in a deadlocked or terminated state.

A spec with liveness is written

$$Init \wedge \Box[Next]_{vars} \wedge Fairness$$

Module $ABSpec$ defines

$$FairSpec \triangleq \boxed{Init \wedge \Box[Next]_{vars}} \wedge \mathbf{WF}_{vars}(Next)$$

Asserts that a behavior keeps taking $Next$ steps
as long as $Next$ is enabled.

not in a ~~deadlocked / terminated state~~

And the safety part of the spec implies that such a state cannot be reached.

A spec with liveness is written

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

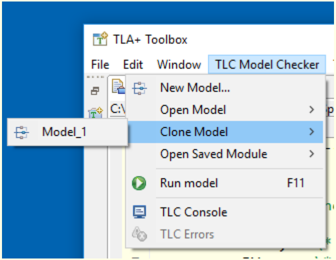
Module $ABSpec$ defines

$$FairSpec \triangleq Init \land \Box[Next]_{vars} \land \mathbf{WF}_{vars}(Next)$$

Asserts that a behavior keeps taking $Next$ steps as long as $Next$ is enabled – which means it keeps sending and receiving values forever.

And the safety part of the spec implies that such a state cannot be reached.

So the behavior must keep taking $Next$ steps, with A sending and B receiving values forever.

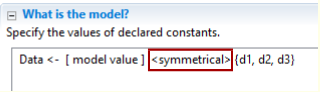


Clone the model you've created for $ABSpec$

For liveness checking, your model must not have any symmetry set.

For liveness checking, your model must not have any symmetry set.

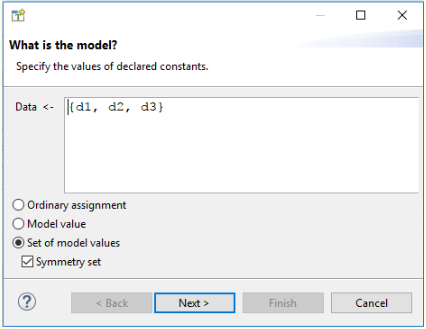
For liveness checking, your model must not have any symmetry set.



For liveness checking, your model must not have any symmetry set.

If it does,

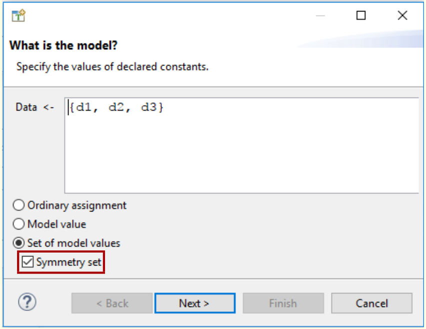
For liveness checking, your model must not have any symmetry set.



For liveness checking, your model must not have any symmetry set.

If it does, change it.

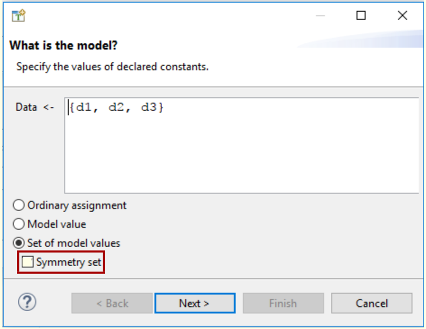
For liveness checking, your model must not have any symmetry set.



For liveness checking, your model must not have any symmetry set.

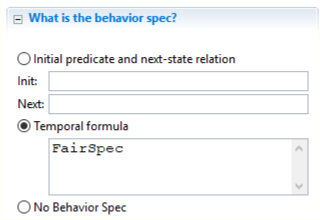
If it does, change it.

For liveness checking, your model must not have any symmetry set.



For liveness checking, your model must not have any symmetry set.

If it does, change it.



Set its behavior spec to $FairSpec$.

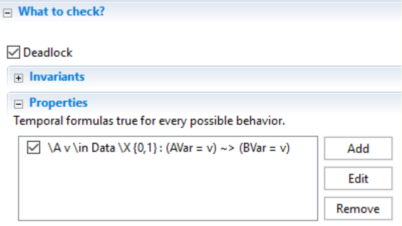
Have TLC check this temporal property:

$$\forall v \in Data \times \{0, 1\} : (AVar = v) \leadsto (BVar = v)$$

Have TLC check that $FairSpec$ satisfies this liveness property, which we looked at before.

Have TLC check this temporal property:

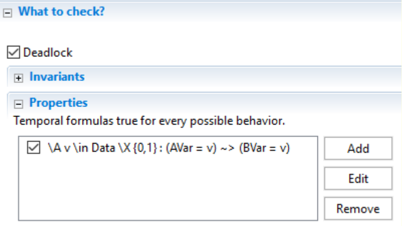
$$\forall v \in Data \times \{0, 1\} : (AVar = v) \leadsto (BVar = v)$$



Have TLC check that $FairSpec$ satisfies this liveness property, which we looked at before.

Have TLC check this temporal property:

$$\forall v \in Data \times \{0, 1\} : (AVar = v) \leadsto (BVar = v)$$



You can copy it from the Web page.

Have TLC check that *FairSpec* satisfies this liveness property, which we looked at before.

You can copy it from the Web page.

Another possible high-level spec of the AB protocol:

$$Init \wedge \Box[Next]_{vars} \wedge \mathrm{WF}_{vars}(B)$$

Here's another possible high-level spec of the AB protocol.

Another possible high-level spec of the AB protocol:

$$Init \wedge \Box[Next]_{vars} \wedge \boxed{\mathrm{WF}_{vars}(B)}$$

Here's another possible high-level spec of the AB protocol.
which has this fairness requirement.

Another possible high-level spec of the AB protocol:

$$Init \wedge \Box[Next]_{vars} \wedge \mathrm{WF}_{vars}(B)$$

Action B is enabled when the sender has sent a value that hasn't been received.

Here's another possible high-level spec of the AB protocol.
which has this fairness requirement.

Action B is enabled when the sender has sent a value that hasn't been received.

Another possible high-level spec of the AB protocol:

$$Init \wedge \Box[Next]_{vars} \wedge \mathrm{WF}_{vars}(B)$$

Action B is enabled when the sender has sent a value that hasn't been received.

It remains enabled until a B step occurs.

Here's another possible high-level spec of the AB protocol.
which has this fairness requirement.

Action B is enabled when the sender has sent a value that hasn't been received.
And it remains enabled until a B step occurs.

Another possible high-level spec of the AB protocol:

$$Init \wedge \Box[Next]_{vars} \wedge \mathrm{WF}_{vars}(B)$$

Action B is enabled when the sender has sent a value that hasn't been received.

It remains enabled until a B step occurs.

This spec requires every sent value to be received,
but allows the sender to stop sending.

This spec requires every sent value to be received,
but allows the sender to stop sending at any time.

Exercise Explain why these two formulas are equivalent, when $Init$, $Next$, ... are defined as in module $ABSpec$:

$$Init \land \Box[Next]_{vars} \land \mathrm{WF}_{vars}(Next)$$

$$Init \land \Box[Next]_{vars} \land \mathrm{WF}_{vars}(A) \land \mathrm{WF}_{vars}(B)$$

Use TLC to check their equivalence.

Here's an exercise for you. Explain why these two formulas are equivalent, when $Init$, $Next$, and so on are defined as they are in module $ABSpec$.

And use TLC to check that they really are equivalent.

The $vars$ Subscript

Here's what that $vars$ subscript is all about.

The $vars$ Subscript

Weak fairness of A asserts of a behavior:

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

Here's what that $vars$ subscript is all about.

Remember our definition of weak fairness of an action A.

The $vars$ Subscript

$\mathrm{WF}_{vars}(A)$ asserts of a behavior:

If $A \wedge (vars' \neq vars)$ ever remains continuously enabled, then an $A \wedge (vars' \neq vars)$ step must eventually occur.

Here's what that $vars$ subscript is all about.

Remember our definition of weak fairness of an action A.

WF of A means weak fairness of the action A and $vars$ prime not equal to $vars$.

The $vars$ Subscript

$\mathrm{WF}_{vars}(A)$ asserts of a behavior:

> If $A \wedge (vars' \neq vars)$ ever remains continuously enabled, then an $A \wedge (vars' \neq vars)$ step must eventually occur.

An $A \wedge (vars' \neq vars)$ step is a non-stuttering A step.

Here's what that $vars$ subscript is all about.

Remember our definition of weak fairness of an action A.

WF of A means weak fairness of the action A and $vars$ prime not equal to $vars$.

A step of that action is a non-stuttering A step.

The $vars$ Subscript

$\mathrm{WF}_{vars}(A)$ asserts of a behavior:

If $A \wedge (vars' \neq vars)$ ever remains continuously enabled, then an $A \wedge (vars' \neq vars)$ step must eventually occur.

An $A \wedge (vars' \neq vars)$ step is a non-stuttering A step.

It makes no sense to require a stuttering step to occur.

We add the non-stuttering requirement because it makes no sense to require a stuttering step to occur, since there's no way of telling whether it did.

You now know what the AB protocol is supposed to do, but you still don't know how it does it. And what is this mysterious strong fairness? Tune in to the second exciting part of this lecture to find out.

Meanwhile, you'll be happy to learn that sequences are the last of the commonly used TLA+ data types that you need to know. And you've seen almost all of the built-in TLA+ operators on those data types.

End of Lecture 9, Part 1

THE ALTERNATING BIT PROTOCOL

THE HIGH LEVEL SPECIFICATION