TLA$^+$ Video Course – Lecture 10, Part 1

Leslie Lamport

IMPLEMENTATION WITH REFINEMENT

PRELIMINARIES

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

The TLA$^+$ Video Course
Lecture 10
Implementation With Refinement

The concept of implementation as implication we've been using works only when all the high-level specification's variables appear in the low-level spec. This lecture explains what implementation means when that isn't the case. It provides important insight into implementation, including what it means for a program to implement a TLA+ spec. But that comes in the second part. In this part, we discuss recursion and substitution, and then introduce our motivating example: another version of the Alternating Bit protocol.

[slide 2]

RECURSIVE DEFINITIONS

Problem:

Define an operator $RemoveX$ that removes all instances of "X" from a sequence of strings.

Suppose we need to define an operator $RemoveX$ that removes all instances of the string X from a sequence of strings.

Problem:

Define an operator $RemoveX$ that removes all instances of "X" from a sequence of strings.

Example:

$$RemoveX(\langle \text{“}Tom\text{”}, \text{“}X\text{”}, \text{“}Dick\text{”}, \text{“}Harry\text{”}, \text{“}X\text{”} \rangle)$$

Suppose we need to define an operator $RemoveX$ that removes all instances of the string X from a sequence of strings.

For example, applying $RemoveX$ to the sequence consisting of the five strings Tom, X, $Dick$, $Harry$, and X

Problem:

Define an operator $RemoveX$ that removes all instances of "X" from a sequence of strings.

Example:

$RemoveX(\langle \text{“}Tom\text{”}, \text{“}X\text{”}, \text{“}Dick\text{”}, \text{“}Harry\text{”}, \text{“}X\text{”} \rangle)$

Suppose we need to define an operator $RemoveX$ that removes all instances of the string X from a sequence of strings.

For example, applying $RemoveX$ to the sequence consisting of the five strings Tom, X, $Dick$, $Harry$, and X
yields the value obtained by removing the two Xs to obtain

Problem:

Define an operator $RemoveX$ that removes all instances of "X" from a sequence of strings.

Example:

$$RemoveX(\langle \text{“}Tom\text{”}, \text{“}X\text{”}, \text{“}Dick\text{”}, \text{“}Harry\text{”}, \text{“}X\text{”} \rangle) \\ = \langle \text{“}Tom\text{”}, \text{“}Dick\text{”}, \text{“}Harry\text{”} \rangle$$

Suppose we need to define an operator $RemoveX$ that removes all instances of the string X from a sequence of strings.

For example, applying $RemoveX$ to the sequence consisting of the five strings Tom, X, $Dick$, $Harry$, and X yields the value obtained by removing the two Xs to obtain the string Tom, $Dick$, $Harry$.

Solution: A recursive definition.

We do this with a recursive definition.

Solution: A recursive definition.

$$\begin{array}{l} RemoveX(\text{sequence}) \ \triangleq \\ \qquad \ldots\ RemoveX(\text{shorter sequence})\ \ldots \end{array}$$

We do this with a recursive definition.

A recursive definition defines $RemoveX$ of a sequence in terms of $RemoveX$ of a shorter sequence.

Solution: A recursive definition.

$RemoveX$(sequence) $\triangleq$
  $\ldots$ $RemoveX$(shorter sequence) $\ldots$

$RemoveX$(shorter sequence) $\triangleq$
  $\ldots$ $RemoveX$(still shorter sequence) $\ldots$

We do this with a recursive definition.

A recursive definition defines $RemoveX$ of a sequence in terms of $RemoveX$ of a shorter sequence.

This means that $RemoveX$ of this shorter sequence is defined to equal some expression involving $RemoveX$ of a still shorter sequence.

Solution: A recursive definition.

$RemoveX(\text{sequence}) \triangleq$
$\quad \ldots\ RemoveX(\text{shorter sequence})\ \ldots$

$RemoveX(\text{shorter sequence}) \triangleq$
$\quad \ldots\ RemoveX(\text{still shorter sequence})\ \ldots$

$$\vdots$$

We can keep going like this, obtaining expressions containing $RemoveX$ applied to shorter and shorter sequences.

Solution: A recursive definition.

$$RemoveX(\text{sequence}) \triangleq$$
$$\ldots\ RemoveX(\text{shorter sequence})\ \ldots$$

$$RemoveX(\text{shorter sequence}) \triangleq$$
$$\ldots\ RemoveX(\text{still shorter sequence})\ \ldots$$

$$\vdots$$

$$\ldots \triangleq\ \ldots\ RemoveX(\langle\rangle)\ \ldots$$

We can keep going like this, obtaining expressions containing $RemoveX$ applied to shorter and shorter sequences.

Eventually we reach an expression containing $RemoveX$ of the empty sequence

Solution: A recursive definition.

$$RemoveX(\text{sequence}) \triangleq$$
$$\quad \ldots\ RemoveX(\text{shorter sequence})\ \ldots$$

$$RemoveX(\text{shorter sequence}) \triangleq$$
$$\quad \ldots\ RemoveX(\text{still shorter sequence})\ \ldots$$

$$\vdots$$

$$\ldots \triangleq\ \ldots\ RemoveX(\langle\rangle)\ \ldots$$

$$RemoveX(\langle\rangle) \triangleq \langle\rangle$$

We can keep going like this, obtaining expressions containing $RemoveX$ applied to shorter and shorter sequences.

Eventually we reach an expression containing $RemoveX$ of the empty sequence which of course equals the empty sequence.

So we have to do two things.

Solution: A recursive definition.

$$RemoveX(\text{sequence}) \triangleq$$
$$\quad \ldots \; RemoveX(\text{shorter sequence}) \; \ldots$$

$$RemoveX(\langle\,\rangle) \triangleq \langle\,\rangle$$

Define $RemoveX$ of the empty sequence.

Solution: A recursive definition.

$$RemoveX(\text{sequence}) \triangleq$$
$$\quad \ldots\ RemoveX(\text{shorter sequence})\ \ldots$$

$$RemoveX(\langle\rangle) \triangleq \langle\rangle$$

Define $RemoveX$ of the empty sequence. And define the value of $RemoveX$ of a non-empty sequence in terms of $RemoveX$ of a shorter sequence.

Solution: A recursive definition.

$$RemoveX(\text{sequence}) \triangleq$$
$$\ldots\ RemoveX(\text{shorter sequence})\ \ldots$$

$$RemoveX(\langle \text{“}X\text{”}, \ldots \rangle) \triangleq RemoveX(\langle \ldots \rangle)$$

$$RemoveX(\langle\,\rangle) \triangleq \langle\,\rangle$$

Define $RemoveX$ of the empty sequence. And define the value of $RemoveX$ of a non-empty sequence in terms of $RemoveX$ of a shorter sequence.

$RemoveX$ of a sequence beginning with X equals $RemoveX$ applied to the rest of the sequence.

Solution: A recursive definition.

$RemoveX$(sequence) $\triangleq$
  $\ldots$ $RemoveX$(shorter sequence) $\ldots$

$RemoveX(\langle \text{“}X\text{”}, \ldots \rangle) \triangleq RemoveX(\langle \ldots \rangle)$

$RemoveX(\langle \text{“}Tom\text{”}, \ldots \rangle) \triangleq \langle \text{“}Tom\text{”} \rangle \circ RemoveX(\langle \ldots \rangle)$

$RemoveX(\langle \rangle) \triangleq \langle \rangle$

And $RemoveX$ of a sequence beginning with another value, such as Tom, equals the sequence that begins with Tom and is followed by the result of applying $RemoveX$ to the rest of the sequence.

Solution: A recursive definition.

$$RemoveX(\langle \text{“}X\text{”}, \ldots \rangle) \triangleq RemoveX(\langle \ldots \rangle)$$

$$RemoveX(\langle \text{“}Tom\text{”}, \ldots \rangle) \triangleq \langle \text{“}Tom\text{”} \rangle \circ RemoveX(\langle \ldots \rangle)$$

$$RemoveX(\langle \rangle) \triangleq \langle \rangle$$

And $RemoveX$ of a sequence beginning with another value, such as Tom, equals the sequence that begins with Tom and is followed by the result of applying $RemoveX$ to the rest of the sequence.

So we just have to write this as a single TLA+ definition.

$$RemoveX(\langle \text{“}X\text{”}, \ldots \rangle) \quad \triangleq RemoveX(\langle \ldots \rangle)$$

$$RemoveX(\langle \text{“}Tom\text{”}, \ldots \rangle) \triangleq \langle \text{“}Tom\text{”} \rangle \circ RemoveX(\langle \ldots \rangle)$$

$$RemoveX(\langle \rangle) \triangleq \langle \rangle$$

And $RemoveX$ of a sequence beginning with another value, such as Tom, equals the sequence that begins with Tom and is followed by the result of applying $RemoveX$ to the rest of the sequence.

So we just have to write this as a single TLA$^+$ definition.

$$RemoveX(\langle \text{"}X\text{"}, \ldots \rangle) \triangleq RemoveX(\langle \ldots \rangle)$$

$$RemoveX(\langle \text{"}Tom\text{"}, \ldots \rangle) \triangleq \langle \text{"}Tom\text{"} \rangle \circ RemoveX(\langle \ldots \rangle)$$

$$RemoveX(\langle \rangle) \triangleq \langle \rangle$$

And $RemoveX$ of a sequence beginning with another value, such as Tom, equals the sequence that begins with Tom and is followed by the result of applying $RemoveX$ to the rest of the sequence.

So we just have to write this as a single TLA+ definition.

$$RemoveX(\langle \text{"}X\text{"}, \ldots \rangle) \triangleq RemoveX(\langle \ldots \rangle)$$

$$RemoveX(\langle \text{"}Tom\text{"}, \ldots \rangle) \triangleq \langle \text{"}Tom\text{"} \rangle \circ RemoveX(\langle \ldots \rangle)$$

$$RemoveX(\langle\, \rangle) \triangleq \langle\, \rangle$$

$$RemoveX(seq) \triangleq$$

Here's the definition.

$$RemoveX(\langle \text{"}X\text{"}, \ldots \rangle) \triangleq RemoveX(\langle \ldots \rangle)$$

$$RemoveX(\langle \text{"}Tom\text{"}, \ldots \rangle) \triangleq \langle \text{"}Tom\text{"} \rangle \circ RemoveX(\langle \ldots \rangle)$$

$$RemoveX(\langle \rangle) \triangleq \langle \rangle$$

$$\begin{array}{l} RemoveX(seq) \triangleq \\ \quad \text{IF } seq = \langle \rangle \\ \quad\quad \text{THEN } \langle \rangle \\ \quad\quad \text{ELSE} \end{array}$$

Here's the definition.

If seq is the empty sequence, then $RemoveX$ of seq equals the empty sequence.

$$RemoveX(\langle \text{“}X\text{”}, \ldots \rangle) \triangleq RemoveX(\langle \ldots \rangle)$$

$$RemoveX(\langle \text{“}Tom\text{”}, \ldots \rangle) \triangleq \langle \text{“}Tom\text{”} \rangle \circ RemoveX(\langle \ldots \rangle)$$

$$RemoveX(\langle \rangle) \triangleq \langle \rangle$$

$$
\begin{array}{l}
RemoveX(seq) \triangleq \\
\quad \text{IF } seq = \langle \rangle \\
\quad\quad \text{THEN } \langle \rangle \\
\quad\quad \text{ELSE IF } Head(seq) = \text{“}X\text{”} \\
\quad\quad\quad\quad\quad \text{THEN} \\
\quad\quad\quad\quad\quad \text{ELSE}
\end{array}
$$

Here's the definition.

If seq is the empty sequence, then $RemoveX$ of seq equals the empty sequence.

Otherwise if the head of seq (its first element) equals the string X,

$$RemoveX(\langle \text{“}X\text{”}, \ldots \rangle) \triangleq RemoveX(\langle \ldots \rangle)$$

$$RemoveX(\langle \text{“}Tom\text{”}, \ldots \rangle) \triangleq \langle \text{“}Tom\text{”} \rangle \circ RemoveX(\langle \ldots \rangle)$$

$$RemoveX(\langle\,\rangle) \triangleq \langle\,\rangle$$

$$
\begin{array}{l}
RemoveX(seq) \triangleq \\
\quad \text{IF } seq = \langle\,\rangle \\
\quad\quad \text{THEN } \langle\,\rangle \\
\quad\quad \text{ELSE IF } Head(seq) = \text{“}X\text{”} \\
\quad\quad\quad\quad\quad \text{THEN } RemoveX(Tail(seq)) \\
\quad\quad\quad\quad\quad \text{ELSE}
\end{array}
$$

Here’s the definition.

If seq is the empty sequence, then $RemoveX$ of seq equals the empty sequence.

Otherwise if the head of seq (its first element) equals the string X, then $RemoveX$ of seq equals $RemoveX$ of the tail of seq.

[slide 24]

$$RemoveX(\langle \text{"}X\text{"}, \ldots \rangle) \triangleq RemoveX(\langle \ldots \rangle)$$

$$RemoveX(\langle \text{"}Tom\text{"}, \ldots \rangle) \triangleq \langle \text{"}Tom\text{"} \rangle \circ RemoveX(\langle \ldots \rangle)$$

$$RemoveX(\langle \, \rangle) \triangleq \langle \, \rangle$$

$$
\begin{array}{l}
RemoveX(seq) \triangleq \\
\quad \text{IF } seq = \langle \, \rangle \\
\quad\quad \text{THEN } \langle \, \rangle \\
\quad\quad \text{ELSE IF } Head(seq) = \text{"}X\text{"} \\
\quad\quad\quad\quad \text{THEN } RemoveX(Tail(seq)) \\
\quad\quad\quad\quad \text{ELSE } \langle Head(seq) \rangle \circ RemoveX(Tail(seq))
\end{array}
$$

Else, it equals the sequence obtained by prepending the head of seq to the front of $RemoveX$ of the tail of seq.

$$
\begin{array}{l}
RemoveX(seq) \triangleq \\
\quad \text{IF } seq = \langle\,\rangle \\
\quad\quad \text{THEN } \langle\,\rangle \\
\quad\quad \text{ELSE IF } Head(seq) = \text{“}X\text{”} \\
\quad\quad\quad\quad\quad \text{THEN } RemoveX(Tail(seq)) \\
\quad\quad\quad\quad\quad \text{ELSE } \langle Head(seq)\rangle \circ RemoveX(Tail(seq))
\end{array}
$$

Else, it equals the sequence obtained by prepending the head of seq to the front of $RemoveX$ of the tail of seq.

This is a recursive definition because the symbol we're defining appears in its definition.

$$
\begin{array}{l}
\textsf{RECURSIVE } RemoveX(\_) \\[4pt]
RemoveX(seq) \triangleq \\
\quad \textsf{IF } seq = \langle\,\rangle \\
\quad\quad \textsf{THEN } \langle\,\rangle \\
\quad\quad \textsf{ELSE IF } Head(seq) = \text{“}X\text{”} \\
\qquad\qquad\qquad \textsf{THEN } RemoveX(Tail(seq)) \\
\qquad\qquad\qquad \textsf{ELSE } \langle Head(seq)\rangle \circ RemoveX(Tail(seq))
\end{array}
$$

Such a definition must be preceded by a RECURSIVE declaration of the symbol being defined, with its arguments indicated by underscore characters.

$$
\begin{array}{l}
\textsf{RECURSIVE } RemoveX(\_) \\
RemoveX(seq) \triangleq \\
\quad \textsf{IF } seq = \langle\,\rangle \\
\qquad \textsf{THEN } \langle\,\rangle \\
\qquad \textsf{ELSE IF } Head(seq) = \text{“}X\text{”} \\
\qquad\qquad\qquad \textsf{THEN } RemoveX(Tail(seq)) \\
\qquad\qquad\qquad \textsf{ELSE } \langle Head(seq)\rangle \circ RemoveX(Tail(seq))
\end{array}
$$

Such a definition must be preceded by a RECURSIVE declaration of the symbol being defined, with its arguments indicated by underscore characters.

This is the complete definition of $RemoveX$.

If you’ve used a “functional” programming language, recursive definitions will seem natural.

If you’ve used a “functional” programming language, recursive definitions will seem natural.

If you've used a "functional" programming language, recursive definitions will seem natural.

If not, think of using a recursive definition when implementing the operator with a program requires a loop.

If you've used a "functional" programming language, recursive definitions will seem natural.

If not, think of using a recursive definition when implementing the operator with a program requires a loop.

SUBSTITUTION

Substitution is a fundamental operation of mathematics.

Substitution is a fundamental operation of mathematics.

Substitution is a fundamental operation of mathematics.

There’s no standard notation for expressing it.

Substitution is a fundamental operation of mathematics.

But mathematicians have no standard notation for expressing it.

Substitution is a fundamental operation of mathematics.

There's no standard notation for expressing it.

In this lecture I will write the expression obtained
by substituting an expression e for the symbol v
in an expression f like this

Substitution is a fundamental operation of mathematics.

But mathematicians have no standard notation for expressing it.

In this lecture I will write the expression obtained by substituting an expression e for the symbol v in an expression f like this,

Substitution is a fundamental operation of mathematics.

There's no standard notation for expressing it.

In this lecture I will write the expression obtained
by substituting an expression e for the symbol v
in an expression f like this

$$f \;\; \textsf{WITH} \;\; v \leftarrow e$$

Substitution is a fundamental operation of mathematics.

But mathematicians have no standard notation for expressing it.

In this lecture I will write the expression obtained by substituting an expression e for the symbol v in an expression f like this, which I'll read as "f with e substituted for v."

[slide 35]

Substitution is a fundamental operation of mathematics.

There's no standard notation for expressing it.

In this lecture I will write the expression obtained
by substituting an expression e for the symbol v
in an expression f like this

$$f \ \text{WITH} \ v \leftarrow e$$

For example $(y^3 - y) \ \text{WITH} \ y \leftarrow x + 2$

For example $y^3 - y$ with $x + 2$ substituted for y

Substitution is a fundamental operation of mathematics.

There's no standard notation for expressing it.

In this lecture I will write the expression obtained
by substituting an expression e for the symbol v
in an expression f like this

$$f \ \text{WITH} \ v \leftarrow e$$

For example $(y^3 - y)$ WITH $y \leftarrow x + 2$

equals $(x+2)^3 - (x+2)$.

For example $y^3 - y$ with $x + 2$ substituted for y
equals the expression $(x+2)$ cubed minus the expression $(x+2)$.

Substitution is a fundamental operation of mathematics.

There's no standard notation for expressing it.

In this lecture I will write the expression obtained by substituting an expression e for the symbol v in an expression f like this

$$f \ \textsf{WITH} \ v \leftarrow e$$

This is not TLA^{+} notation.

For example $y^3 - y$ with $x + 2$ substituted for y equals the expression $(x + 2)$ cubed minus the expression $(x + 2)$.

This is not TLA^{+} notation. I'm using it only for this lecture.

A Fundamental Law of Ordinary Math

There's a fundamental law of substitution in ordinary math.

It says that

A Fundamental Law of Ordinary Math

For any variable v and expressions e and f

There's a fundamental law of substitution in ordinary math.

It says that for any variable v and expressions e and f,

A Fundamental Law of Ordinary Math

For any variable v and expressions e and f

$$v = e \quad \textbf{implies} \quad f = (f \ \text{WITH} \ v \leftarrow e)$$

There's a fundamental law of substitution in ordinary math.

It says that for any variable v and expressions e and f,
v equals e implies that f equals the expression f with e substituted for v.

A Fundamental Law of Ordinary Math

For any variable v and expressions e and f

$$v = e \quad \textbf{implies} \quad f = (f \text{ WITH } v \leftarrow e)$$

Let's write it as

$$\text{THEOREM } (v = e) \Rightarrow (f = (f \text{ WITH } v \leftarrow e))$$

There's a fundamental law of substitution in ordinary math.

It says that for any variable v and expressions e and f,
v equals e implies that f equals the expression f with e substituted for v.

Let's write it as this theorem.

A Fundamental Law of Ordinary Math

For any variable v and expressions e and f

$$v = e \ \textbf{implies} \ f = (f \ \text{WITH} \ v \leftarrow e)$$

Let's write it as

$$\text{THEOREM} \ (v = e) \Rightarrow (f = (f \ \text{WITH} \ v \leftarrow e))$$

I'll call this the **Simple Substitution Law**

There's a fundamental law of substitution in ordinary math.

It says that for any variable v and expressions e and f, v equals e implies that f equals the expression f with e substituted for v.

Let's write it as this theorem.

I'll call this law the *Simple Substitution Law*, though it's not what mathematicians call it.

Ordinary math corresponds to the constant expressions of TLA$^+$.

Ordinary math corresponds to the constant expressions of TLA$^+$.

Ordinary math corresponds to the constant expressions of TLA+.

Mathematicians' variables are the CONSTANTS of TLA+.

Ordinary math corresponds to the constant expressions of TLA+.

Mathematicians' variables are the declared constants of TLA+.

Ordinary math corresponds to the constant expressions of TLA+.

Mathematicians' variables are the CONSTANTS of TLA+.

Nothing in ordinary math corresponds to the VARIABLES and non-constant operators of TLA+.

Ordinary math corresponds to the constant expressions of TLA+.

Mathematicians' variables are the declared constants of TLA+.

Nothing in ordinary math corresponds to the declared variables and non-constant operators of TLA+.

Ordinary math corresponds to the constant expressions of TLA^{+}.

Mathematicians' variables are the CONSTANTS of TLA^{+}.

Nothing in ordinary math corresponds to the VARIABLES and non-constant operators of TLA^{+}.

They belong to temporal logic,

Ordinary math corresponds to the constant expressions of TLA^{+}.

Mathematicians' variables are the declared constants of TLA^{+}.

Nothing in ordinary math corresponds to the declared variables and non-constant operators of TLA^{+}.

They belong to temporal logic,

Ordinary math corresponds to the constant expressions of TLA$^+$.

Mathematicians' variables are the CONSTANTS of TLA$^+$.

Nothing in ordinary math corresponds to the VARIABLES and non-constant operators of TLA$^+$.

They belong to temporal logic, a special kind of math that's not as simple as ordinary math.

Ordinary math corresponds to the constant expressions of TLA$^+$.

Mathematicians' variables are the declared constants of TLA$^+$.

Nothing in ordinary math corresponds to the declared variables and non-constant operators of TLA$^+$.

They belong to temporal logic, a special kind of math that's not as simple as ordinary math.

Ordinary math corresponds to the constant expressions of TLA+.

Mathematicians' variables are the CONSTANTS of TLA+.

Nothing in ordinary math corresponds to the VARIABLES and non-constant operators of TLA+.

Temporal Logic of Actions

And T-L-A stands for the temporal logic of actions.

THEOREM $(v = e) \Rightarrow (f = (f \text{ WITH } v \leftarrow e))$

The Simple Substitution Law

THEOREM $(v = e) \Rightarrow (f = (f \text{ WITH } v \leftarrow e))$

This is not true if v is a variable or e is a non-constant expression.

The Simple Substitution Law is not true if v is a variable or e is a non-constant expression.

THEOREM $(v = e) \Rightarrow (f = (f \text{ WITH } v \leftarrow e))$

This is not true if v is a variable or e is a non-constant expression.

Example

The Simple Substitution Law is not true if v is a variable or e is a non-constant expression.

Here's an example that shows it's not true.

THEOREM $(v = e) \Rightarrow (f = (f \text{ WITH } v \leftarrow e))$

This is not true if v is a variable or e is a non-constant expression.

Example $v \leftarrow y$, $e \leftarrow x + 2$, $f \leftarrow y'$ where x and y are variables

The Simple Substitution Law is not true if v is a variable or e is a non-constant expression.

Here's an example that shows it's not true.

Let's substitute y for v, $x + 2$ for e, and y prime for f in the law – where x and y are variables.

THEOREM $(v = e) \Rightarrow (f = (f \text{ WITH } v \leftarrow e))$

This is not true if v is a variable or e is a non-constant expression.

Example $v \leftarrow y,\ e \leftarrow x + 2,\ f \leftarrow y'$ where x and y are variables

THEOREM

The law states that

THEOREM $(v = e) \Rightarrow (f = (f$ WITH $v \leftarrow e))$

This is not true if v is a variable or e is a non-constant expression.

Example $v \leftarrow y,\ e \leftarrow x + 2,\ f \leftarrow y'$ where x and y are variables

THEOREM $(y = x + 2) \Rightarrow$

The law states that

“v equals e”, which is “y equals $x + 2$”, implies that

THEOREM $(v = e) \Rightarrow (f = (f$ WITH $v \leftarrow e))$

This is not true if v is a variable or e is a non-constant expression.

Example $v \leftarrow y$, $e \leftarrow x + 2$, $f \leftarrow y'$ where x and y are variables

THEOREM $(y = x + 2) \Rightarrow (y' =$

The law states that

"v equals e", which is "y equals $x + 2$", implies that

f, which is y prime, equals

$\text{THEOREM}\ (v = e) \Rightarrow (f = (f\ \text{WITH}\ v \leftarrow e))$

This is not true if v is a variable or e is a non-constant expression.

Example $v \leftarrow y$, $e \leftarrow x + 2$, $f \leftarrow y'$ where x and y are variables

$$\text{THEOREM}\ (y = x + 2) \Rightarrow (y' = (x + 2)')$$

The law states that

"v equals e", which is "y equals $x + 2$", implies that

f, which is y prime, equals

"the expression f with $x + 2$ substituted for y", which is $x + 2$ prime.

THEOREM $(v = e) \Rightarrow (f = (f \text{ WITH } v \leftarrow e))$

This is not true if v is a variable or e is a non-constant expression.

Example $v \leftarrow y,\ e \leftarrow x + 2,\ f \leftarrow y'$ where x and y are variables

THEOREM $\boxed{(y = x + 2) \ \Rightarrow\ (y' = (x + 2)')}$

This is an assertion about a behavior.

This formula is an assertion about a behavior, and the theorem asserts that it's true for all behaviors.

THEOREM $(v = e) \Rightarrow (f = (f \text{ WITH } v \leftarrow e))$

This is not true if v is a variable or e is a non-constant expression.

Example $v \leftarrow y,\ e \leftarrow x + 2,\ f \leftarrow y'$ where x and y are variables

THEOREM $\boxed{(y = x + 2)} \Rightarrow (y' = (x + 2)')$

This is an assertion about a behavior.

Asserts $y = x + 2$ in first state of behavior.

This formula is an assertion about a behavior, and the theorem asserts that it's true for all behaviors.

This is a state formula, so it asserts that $y = x + 2$ is true in the first state of the behavior.

THEOREM $(v = e) \Rightarrow (f = (f \text{ WITH } v \leftarrow e))$

This is not true if v is a variable or e is a non-constant expression.

Example $v \leftarrow y,\ e \leftarrow x + 2,\ f \leftarrow y'$ where x and y are variables

THEOREM $(y = x + 2) \Rightarrow \boxed{(y' = (x + 2)')}$

This is an assertion about a behavior.

Asserts $y = x + 2$ in first state of behavior.

This action formula asserts that the value of y in the second state of the behavior equals the value of x plus two in that second state –

THEOREM $(v = e) \Rightarrow (f = (f \text{ WITH } v \leftarrow e))$

This is not true if v is a variable or e is a non-constant expression.

Example $v \leftarrow y,\ e \leftarrow x + 2,\ f \leftarrow y'$ where x and y are variables

THEOREM $(y = x + 2) \Rightarrow \boxed{(y' = (x + 2)')}$

This is an assertion about a behavior.

Asserts $y = x + 2$ in first state of behavior.

Asserts $y = x + 2$ in second state of behavior.

This action formula asserts that the value of y in the second state of the behavior equals the value of x plus two in that second state – in other words, that $y = x + 2$ is true in the second state of the behavior.

THEOREM $(v = e) \Rightarrow (f = (f$ WITH $v \leftarrow e))$

This is not true if v is a variable or e is a non-constant expression.

Example $v \leftarrow y,\ e \leftarrow x + 2,\ f \leftarrow y'$ where x and y are variables

THEOREM $\boxed{(y = x + 2) \Rightarrow (y' = (x + 2)')}$

Asserts $y = x + 2$ in the first state
implies $y = x + 2$ in the second state.

This action formula asserts that the value of y in the second state of the behavior equals the value of x plus two in that second state – in other words, that $y = x + 2$ is true in the second state of the behavior.

So this formula asserts that $y = x + 2$ true in the first state implies that it's also true in the second state.

THEOREM $(v = e) \Rightarrow (f = (f \text{ WITH } v \leftarrow e))$

This is not true if v is a variable or e is a non-constant expression.

Example $v \leftarrow y,\ e \leftarrow x + 2,\ f \leftarrow y'$ where x and y are variables

~~THEOREM $(y = x + 2) \Rightarrow (y' = (x + 2)')$~~

Asserts $y = x + 2$ in the first state
implies $y = x + 2$ in the second state.

Not true for all behaviors

This action formula asserts that the value of y in the second state of the behavior equals the value of x plus two in that second state – in other words, that $y = x + 2$ is true in the second state of the behavior.

So this formula asserts that $y = x + 2$ true in the first state implies that it's also true in the second state.

Which is not true for all behaviors.

THEOREM $(v = e) \Rightarrow (f = (f \text{ WITH } v \leftarrow e))$

This is not true if v is a variable or e is a non-constant expression.

Example $v \leftarrow y,\ e \leftarrow x + 2,\ f \leftarrow y'$ where x and y are variables

THEOREM $(y = x + 2) \Rightarrow (y' = (x + 2)')$

Asserts $y = x + 2$ in the first state
implies $y = x + 2$ in the second state.

Not true for all behaviors

The law is not true if v is a variable or e is a non-constant expression.

The Temporal Substitution Law

To obtain the substitution law for temporal logic, which I will call the *Temporal Substitution Law*

The Temporal Substitution Law

THEOREM $(v = e) \Rightarrow (f = (f \text{ WITH } v \leftarrow e))$

To obtain the substitution law for temporal logic, which I will call the *Temporal Substitution Law*

We change the Simple Substitution Law

The Temporal Substitution Law

THEOREM $\Box(v = e) \Rightarrow (f = (f \text{ WITH } v \leftarrow e))$

To obtain the substitution law for temporal logic, which I will call the *Temporal Substitution Law*

We change the Simple Substitution Law by adding this *always* operator.

The Temporal Substitution Law

$\text{THEOREM } (\Box\,(v = e)) \Rightarrow \ (f \ = \ (f \ \text{ WITH } \ v \leftarrow e))$

To obtain the substitution law for temporal logic, which I will call the *Temporal Substitution Law*

We change the Simple Substitution Law by adding this *always* operator.

The statement of the theorem is parsed like this,

The Temporal Substitution Law

$$\text{THEOREM } \Box(v = e) \Rightarrow (f = (f \text{ WITH } v \leftarrow e))$$

To obtain the substitution law for temporal logic, which I will call the *Temporal Substitution Law*

We change the Simple Substitution Law by adding this *always* operator.

The statement of the theorem is parsed like this,

The Temporal Substitution Law

THEOREM $\Box(v = e) \Rightarrow (f = (f \text{ WITH } v \leftarrow e))$

Asserts that, for every behavior:

To obtain the substitution law for temporal logic, which I will call the *Temporal Substitution Law*
We change the Simple Substitution Law by adding this *always* operator.

The statement of the theorem is parsed like this,

So the law now asserts that, for every behavior:

The Temporal Substitution Law

THEOREM $\Box(v = e) \Rightarrow (f = (f \text{ WITH } v \leftarrow e))$

Asserts that, for every behavior:

if $v = e$ is true in all states of the behavior

To obtain the substitution law for temporal logic, which I will call the *Temporal Substitution Law*

We change the Simple Substitution Law by adding this *always* operator.

The statement of the theorem is parsed like this,

So the law now asserts that, for every behavior: if $v = e$ is true in all states of the behavior,

The Temporal Substitution Law

THEOREM $\Box(v = e) \Rightarrow (f = (f \text{ WITH } v \leftarrow e))$

Asserts that, for every behavior:

if $v = e$ is true in all states of the behavior

then $f = (f \text{ WITH } v \leftarrow e)$ is true on the behavior

To obtain the substitution law for temporal logic, which I will call the *Temporal Substitution Law*

We change the Simple Substitution Law by adding this *always* operator.

The statement of the theorem is parsed like this,

So the law now asserts that, for every behavior: if $v = e$ is true in all states of the behavior, then the formula "f equals f with e substituted for v" is true on the behavior.

[slide 72]

The Temporal Substitution Law

THEOREM $\Box(v = e) \;\Rightarrow\; (f \;=\; (f \;\text{WITH}\; v \leftarrow e))$

Asserts that, for every behavior:

if $v = e$ is true in all states of the behavior

then $f \;=\; (f \;\text{WITH}\; v \leftarrow e)$ is true on the behavior

Example

Let's look at the same example as before.

The Temporal Substitution Law

THEOREM $\Box(v = e) \Rightarrow (f = (f \text{ WITH } v \leftarrow e))$

Asserts that, for every behavior:

if $v = e$ is true in all states of the behavior

then $f = (f \text{ WITH } v \leftarrow e)$ is true on the behavior

Example $v \leftarrow y$, $e \leftarrow x + 2$, $f \leftarrow y'$ where x and y are variables

Let's look at the same example as before.

With y substituted for v, $x + 2$ substituted for e and y prime substituted for f, where x and y are variables.

The Temporal Substitution Law

THEOREM $\Box(v = e) \Rightarrow (f = (f \text{ WITH } v \leftarrow e))$

Asserts that, for every behavior:

if $v = e$ is true in all states of the behavior

then $f = (f \text{ WITH } v \leftarrow e)$ is true on the behavior

Example $v \leftarrow y$, $e \leftarrow x + 2$, $f \leftarrow y'$ where x and y are variables

THEOREM $\Box(y = x + 2) \Rightarrow (y' = (x + 2)')$

Let's look at the same example as before.

With y substituted for v, $x + 2$ substituted for e and y prime substituted for f, where x and y are variables.

The law now

The Temporal Substitution Law

THEOREM $\Box(v = e) \Rightarrow (f = (f \text{ WITH } v \leftarrow e))$

Asserts that, for every behavior:

if $v = e$ is true in all states of the behavior

then $f = (f \text{ WITH } v \leftarrow e)$ is true on the behavior

Example $v \leftarrow y,\ e \leftarrow x + 2,\ f \leftarrow y'$ where x and y are variables

THEOREM $\boxed{\Box(y = x + 2)} \Rightarrow (y' = (x + 2)')$

Asserts: **if** $y = x + 2$ in all states of a behavior

Let's look at the same example as before.

With y substituted for v, $x + 2$ substituted for e and y prime substituted for f, where x and y are variables.

The law now asserts that if y equals $x + 2$ in **all** states of a behavior

The Temporal Substitution Law

THEOREM $\Box(v = e) \Rightarrow (f = (f \text{ WITH } v \leftarrow e))$

Asserts that, for every behavior:

if $v = e$ is true in all states of the behavior

then $f = (f \text{ WITH } v \leftarrow e)$ is true on the behavior

Example $v \leftarrow y,\ e \leftarrow x + 2,\ f \leftarrow y'$ where x and y are variables

THEOREM $\Box(y = x + 2) \Rightarrow \boxed{(y' = (x + 2)')}$

Asserts: **if** $y = x + 2$ in all states of a behavior

then $y = x + 2$ in the second state of the behavior

then y equals $x + 2$ in the second state of the behavior.

The Temporal Substitution Law

THEOREM $\Box(v = e) \Rightarrow (f = (f \text{ WITH } v \leftarrow e))$

Asserts that, for every behavior:

if $v = e$ is true in all states of the behavior

then $f = (f \text{ WITH } v \leftarrow e)$ is true on the behavior

Example $v \leftarrow y,\ e \leftarrow x + 2,\ f \leftarrow y'$ where x and y are variables

THEOREM $\Box(y = x + 2) \Rightarrow (y' = (x + 2)')$

Asserts: **if** $y = x + 2$ in all states of a behavior

then $y = x + 2$ in the second state of the behavior

then y equals $x + 2$ in the second state of the behavior.

Which is obviously true of all behaviors.

The Temporal Substitution Law

THEOREM $\Box(v = e) \Rightarrow (f = (f \text{ WITH } v \leftarrow e))$

then y equals $x + 2$ in the second state of the behavior.

Which is obviously true of all behaviors.

The Temporal Substitution Law

$\text{THEOREM} \;\; \Box(v = e) \;\Rightarrow\; (f \;=\; (f \;\text{WITH}\; v \leftarrow e))$

True when v a VARIABLE and e a state expression.

This law is true when v is a declared VARIABLE and e is a state expression.

The Temporal Substitution Law

THEOREM $\Box (v = e) \Rightarrow (f = (f \text{ WITH } v \leftarrow e))$

True when v a VARIABLE and e a state expression.

Also true when v a CONSTANT and e a constant expression

This law is true when v is a declared VARIABLE and e is a state expression.

The law is also true when v is a declared CONSTANT and e is a constant expression,

The Temporal Substitution Law

THEOREM $\Box(v = e) \Rightarrow (f = (f \text{ WITH } v \leftarrow e))$

True when v a VARIABLE and e a state expression.

Also true when v a CONSTANT and e a constant expression,
in which case their values don't depend on the state

This law is true when v is a declared VARIABLE and e is a state expression.

The law is also true when v is a declared CONSTANT and e is a constant expression,
in which case the values of v and e don't depend on the state,

The Temporal Substitution Law

THEOREM $\Box(v = e) \;\Rightarrow\; (f \;=\; (f \;\text{ WITH }\; v \leftarrow e))$

True when v a VARIABLE and e a state expression.

Also true when v a CONSTANT and e a constant expression,
in which case their values don't depend on the state
and in any behavior

This law is true when v is a declared VARIABLE and e is a state expression.

The law is also true when v is a declared CONSTANT and e is a constant expression,
in which case the values of v and e don't depend on the state,
and therefore, in any behavior,

The Temporal Substitution Law

THEOREM $\square(v = e) \;\Rightarrow\; (f \;=\; (f \;\text{WITH}\; v \leftarrow e))$

True when v a VARIABLE and e a state expression.

Also true when v a CONSTANT and e a constant expression,
in which case their values don't depend on the state
and in any behavior $v = e$ is true

This law is true when v is a declared VARIABLE and e is a state expression.

The law is also true when v is a declared CONSTANT and e is a constant expression,
in which case the values of v and e don't depend on the state,
and therefore, in any behavior, $v = e$ is true in the initial state

The Temporal Substitution Law

THEOREM $\Box(v = e) \Rightarrow (f = (f \text{ WITH } v \leftarrow e))$

True when v a VARIABLE and e a state expression.

Also true when v a CONSTANT and e a constant expression,
in which case their values don't depend on the state
and in any behavior $v = e$ is true iff $\Box(v = e)$ is.

This law is true when v is a declared VARIABLE and e is a state expression.

The law is also true when v is a declared CONSTANT and e is a constant expression,
in which case the values of v and e don't depend on the state,
and therefore, in any behavior, $v = e$ is true in the initial state
if and only if it's true in all states of the behavior.

The Temporal Substitution Law

THEOREM $\Box(v = e) \Rightarrow (f = (f \text{ WITH } v \leftarrow e))$

True when v a VARIABLE and e a state expression.

Also true when v a CONSTANT and e a constant expression, in which case their values don't depend on the state and in any behavior $v = e$ is true iff $\Box(v = e)$ is.

So we get the Ordinary Substitution Law:

THEOREM $(v = e) \Rightarrow (f = (f \text{ WITH } v \leftarrow e))$

So The Temporal Substitution Law becomes the Ordinary Substitution Law.

The General Temporal Substitution Law

$\text{THEOREM} \quad \Box(v = e) \;\Rightarrow\; (f \;=\; (f \;\; \text{WITH} \;\; v \leftarrow e))$

There's a straightforward generalization of the Temporal Substitution Law . . .

The General Temporal Substitution Law

THEOREM $\Box(v = e) \;\Rightarrow\; (f = (f \text{ WITH } v \leftarrow e))$

THEOREM $\Box((v_1 = e_1) \wedge (v_2 = e_2))$
$\qquad \Rightarrow (f = (f \text{ WITH } v_1 \leftarrow e_1,\, v_2 \leftarrow e_2))$

There's a straightforward generalization of the Temporal Substitution Law . . .
to substitution for two variables.

The General Temporal Substitution Law

$$\text{THEOREM} \quad \Box(v = e) \Rightarrow (f = (f \text{ WITH } v \leftarrow e))$$

$$\text{THEOREM} \quad \Box((v_1 = e_1) \land (v_2 = e_2)) \\ \Rightarrow (f = \boxed{(f \text{ WITH } v_1 \leftarrow e_1,\, v_2 \leftarrow e_2)})$$

There's a straightforward generalization of the Temporal Substitution Law . . . to substitution for two variables.

The meaning of this WITH expression should be obvious, except perhaps for the fact that

The General Temporal Substitution Law

THEOREM $\Box(v = e) \Rightarrow (f = (f \text{ WITH } v \leftarrow e))$

THEOREM $\Box((v_1 = e_1) \wedge (v_2 = e_2))$
$$\Rightarrow (f = \boxed{(f \text{ WITH } v_1 \leftarrow e_1,\ v_2 \leftarrow e_2)})$$

simultaneous substitution

the substitutions for v-one and v-two have to be done simultaneously, not one after the other. (This makes a difference if v2 appears in expression e1.)

The General Temporal Substitution Law

THEOREM $\Box(v = e) \Rightarrow (f = (f \text{ WITH } v \leftarrow e))$

THEOREM $\Box((v_1 = e_1) \land (v_2 = e_2))$
$$\Rightarrow (f = (f \text{ WITH } v_1 \leftarrow e_1,\ v_2 \leftarrow e_2))$$

the substitutions for v-one and v-two have to be done simultaneously, not one after the other. (This makes a difference if $v2$ appears in expression $e1$.)

The General Temporal Substitution Law

$$\text{THEOREM} \quad \Box(v = e) \;\Rightarrow\; (f = (f \text{ WITH } v \leftarrow e))$$

$$\text{THEOREM} \quad \Box((v_1 = e_1) \land (v_2 = e_2)) \\ \Rightarrow\; (f = (f \text{ WITH } v_1 \leftarrow e_1,\, v_2 \leftarrow e_2))$$

$$\text{THEOREM} \quad \Box((v_1 = e_1) \land (v_2 = e_2) \land \ldots) \\ \Rightarrow\; (f = (f \text{ WITH } v_1 \leftarrow e_1,\, v_2 \leftarrow e_2, \ldots))$$

the substitutions for v-one and v-two have to be done simultaneously, not one after the other. (This makes a difference if $v2$ appears in expression $e1$.)

And this obviously generalizes to substitution for any number of variables.

The Temporal Substitution Law

THEOREM $\Box(v = e) \;\Rightarrow\; (f = (f \text{ WITH } v \leftarrow e))$

THEOREM $\Box((v_1 = e_1) \wedge (v_2 = e_2))$
$\Rightarrow\; (f = (f \text{ WITH } v_1 \leftarrow e_1,\, v_2 \leftarrow e_2))$

THEOREM $\Box((v_1 = e_1) \wedge (v_2 = e_2) \wedge \ldots)$
$\Rightarrow\; (f = (f \text{ WITH } v_1 \leftarrow e_1,\, v_2 \leftarrow e_2, \ldots))$

the substitutions for v-one and v-two have to be done simultaneously, not one after the other. (This makes a difference if $v2$ appears in expression $e1$.)

And this obviously generalizes to substitution for any number of variables.

It's this general version that I'll refer to as the Temporal Substitution Law.
[slide 93]

THE AB2 PROTOCOL

We now come to the motivating example of this lecture, the AB2 protocol.

The AB2 protocol is AB protocol with one simple modification:

The AB2 protocol is the same as the Alternating Bit protocol of Lecture 9 except for one simple modification:

The AB2 protocol is AB protocol with one simple modification:

Messages are detectably corrupted rather than lost.

The AB2 protocol is the same as the Alternating Bit protocol of Lecture 9 except for one simple modification:

Messages are detectably corrupted rather than lost.

The AB2 protocol is AB protocol with one simple modification:

Messages are detectably corrupted rather than lost.

A corrupted message is represented by a value Bad unequal to any message that can be sent.

The AB2 protocol is the same as the Alternating Bit protocol of Lecture 9 except for one simple modification:

Messages are detectably corrupted rather than lost.

A corrupted message is represented by a special value Bad that doesn't equal any message that can be sent.

The AB2 protocol is AB protocol with one simple modification:

Messages are detectably corrupted rather than lost.

A corrupted message is represented by a value Bad unequal to any message that can be sent.

The specification is in module $AB2$

The specification is in module $AB2$,

The AB2 protocol is AB protocol with one simple modification:

Messages are detectably corrupted rather than lost.

A corrupted message is represented by a value Bad unequal to any message that can be sent.

The specification is in module $AB2$, which you can now download.

The specification is in module $AB2$, which you can now download.

Module AB2 is obtained by modifying module AB.

Module AB2 is obtained by making simple modifications to module AB.

If starts like AB by

Module AB2 is obtained by modifying module AB.

EXTENDS $Integers, Sequences$

Module AB2 is obtained by making simple modifications to module AB.

If starts like AB by extending the $Integers$ and $Sequences$ modules, and

Module AB2 is obtained by modifying module AB.

EXTENDS $Integers, Sequences$

CONSTANT $Data$

Module AB2 is obtained by making simple modifications to module AB.

If starts like AB by extending the $Integers$ and $Sequences$ modules, and declaring the constant $Data$, the set of possible data values that can be sent.

It also declares

Module AB2 is obtained by modifying module AB.

EXTENDS $Integers, Sequences$

CONSTANTS $Data, Bad$

Module AB2 is obtained by making simple modifications to module AB.

If starts like AB by extending the $Integers$ and $Sequences$ modules, and declaring the constant $Data$, the set of possible data values that can be sent.

It also declares the constant Bad, and adds the assumption

Module AB2 is obtained by modifying module AB.

EXTENDS $Integers, Sequences$

CONSTANTS $Data, \boxed{Bad}$

ASSUME $Bad \notin$ the set of possible messages

that Bad is not an element of the set of possible messages that can be sent, which equals

Module AB2 is obtained by modifying module AB.

EXTENDS $Integers,\ Sequences$

CONSTANTS $Data,\ Bad$

ASSUME $Bad \notin \boxed{(Data \times \{0, 1\})}$

the set of possible messages
from A to B

that Bad is not an element of the set of possible messages that can be sent, which equals the set of possible messages that can be sent from A to B

Module AB2 is obtained by modifying module AB.

EXTENDS *Integers*, *Sequences*

CONSTANTS *Data*, *Bad*

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

the set of possible messages from B to A

that Bad is not an element of the set of possible messages that can be sent, which equals the set of possible messages that can be sent from A to B

union with the set of possible messages that can be sent from B to A, which contains the two values 0 and 1.

Module AB2 is obtained by modifying module AB.

EXTENDS $Integers, \ Sequences$

CONSTANTS $Data, \ Bad$

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

that Bad is not an element of the set of possible messages that can be sent, which equals the set of possible messages that can be sent from A to B

union with the set of possible messages that can be sent from B to A, which contains the two values 0 and 1.

VARIABLES $AVar, BVar,$

The variables $AVar$ and $BVar$ are the same as as in module AB, but

VARIABLES $AVar, BVar, AtoB, BtoA$

The variables $AVar$ and $BVar$ are the same as as in module AB, but the message sequences $AtoB$ and $BtoA$ are renamed

VARIABLES $AVar$, $BVar$, $AtoB2$, $BtoA2$

The variables $AVar$ and $BVar$ are the same as as in module AB, but the message sequences $AtoB$ and $BtoA$ are renamed $AtoB2$ and $BtoA2$.

VARIABLES $AVar$, $BVar$, $AtoB2$, $BtoA2$

$vars \triangleq \langle AVar, BVar, AtoB2, BtoA2 \rangle$

The variables $AVar$ and $BVar$ are the same as as in module AB, but the message sequences $AtoB$ and $BtoA$ are renamed $AtoB2$ and $BtoA2$.

$vars$ is again defined to be the tuple of all variables.

VARIABLES $AVar, \ BVar, \ AtoB2, \ BtoA2$

$$vars \triangleq \langle AVar, BVar, AtoB2, BtoA2 \rangle$$

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

The variables $AVar$ and $BVar$ are the same as as in module AB, but the message sequences $AtoB$ and $BtoA$ are renamed $AtoB2$ and $BtoA2$.

$vars$ is again defined to be the tuple of all variables.

Here's the definition of $TypeOK$ from AB.

VARIABLES $AVar, \; BVar, \; AtoB2, \; BtoA2$

$$vars \triangleq \langle AVar, BVar, AtoB2, BtoA2 \rangle$$

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

The type assertions for $AVar$ and $BVar$ are the same as in AB

$$\text{VARIABLES} \quad AVar,\ BVar,\ AtoB2,\ BtoA2$$

$$vars \triangleq \langle AVar, BVar, AtoB2, BtoA2 \rangle$$

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

The type assertions for $AVar$ and $BVar$ are the same as in AB

In module AB, the variable $AtoB$ equals a sequence of $Data$, bit pairs,

VARIABLES $AVar,\ BVar,\ AtoB2,\ BtoA2$

$$vars \triangleq \langle AVar,\ BVar,\ AtoB2,\ BtoA2 \rangle$$

$$
\begin{aligned}
TypeOK \triangleq\ & \land\ AVar \in Data \times \{0,1\} \\
& \land\ BVar \in Data \times \{0,1\} \\
& \land\ AtoB2 \in Seq((Data \times \{0,1\}) \cup \{Bad\}) \\
& \land\ BtoA \in Seq(\{0,1\})
\end{aligned}
$$

The type assertions for $AVar$ and $BVar$ are the same as in AB

In module AB, the variable $AtoB$ equals a sequence of $Data$, bit pairs, while the elements of the sequence $AtoB2$ are either $Data$, bit pairs or else equal to Bad.

VARIABLES $AVar,\ BVar,\ AtoB2,\ BtoA2$

$$vars \triangleq \langle AVar, BVar, AtoB2, BtoA2 \rangle$$

$$\begin{aligned} TypeOK \triangleq\ & \land\ AVar \in Data \times \{0, 1\} \\ & \land\ BVar \in Data \times \{0, 1\} \\ & \land\ AtoB2 \in Seq((Data \times \{0, 1\}) \cup \{Bad\}) \\ & \land\ BtoA \in Seq(\{0, 1\}) \end{aligned}$$

The type assertions for $AVar$ and $BVar$ are the same as in AB

In module AB, the variable $AtoB$ equals a sequence of $Data$, bit pairs, while the elements of the sequence $AtoB2$ are either $Data$, bit pairs or else equal to Bad.

Stop the video and make sure you understand this formula.

$$\text{VARIABLES} \quad AVar,\ BVar,\ AtoB2,\ BtoA2$$

$$vars \triangleq \langle AVar,\ BVar,\ AtoB2,\ BtoA2 \rangle$$

$$\begin{aligned} TypeOK \triangleq\ & \land\ AVar \in Data \times \{0, 1\} \\ & \land\ BVar \in Data \times \{0, 1\} \\ & \land\ AtoB2 \in Seq((Data \times \{0, 1\}) \cup \{Bad\}) \\ & \land\ BtoA \in Seq(\{0, 1\}) \end{aligned}$$

Similarly, where $BtoA$ of module AB is a sequence of zeros or ones,

$$\text{VARIABLES } AVar,\ BVar,\ AtoB2,\ BtoA2$$

$$vars \triangleq \langle AVar, BVar, AtoB2, BtoA2 \rangle$$

$$
\begin{aligned}
TypeOK \triangleq\ & \land\ AVar \in Data \times \{0, 1\} \\
& \land\ BVar \in Data \times \{0, 1\} \\
& \land\ AtoB2 \in Seq((Data \times \{0, 1\}) \cup \{Bad\}) \\
& \land\ BtoA2 \in Seq(\{0, 1, Bad\})
\end{aligned}
$$

Similarly, where $BtoA$ of module AB is a sequence of zeros or ones, $BtoA2$ is a sequence of the values zero, one, or Bad.

$$
\text{VARIABLES} \quad AVar,\ BVar,\ AtoB2,\ BtoA2
$$

$$
vars \triangleq \langle AVar, BVar, AtoB2, BtoA2 \rangle
$$

$$
\begin{aligned}
TypeOK \triangleq\ & \land\ AVar \in Data \times \{0, 1\} \\
& \land\ BVar \in Data \times \{0, 1\} \\
& \land\ AtoB2 \in Seq((Data \times \{0, 1\}) \cup \{Bad\}) \\
& \land\ BtoA2 \in Seq(\{0, 1, Bad\})
\end{aligned}
$$

Similarly, where $BtoA$ of module AB is a sequence of zeros or ones, $BtoA2$ is a sequence of the values zero, one, or Bad.

$Init$, $ASnd$, $BSnd$ are the same except with

$AtoB \leftarrow AtoB2 \quad BtoA \leftarrow BtoA2$

The initial-state formula and the actions in which A and B send messages are the same except for renaming the variables $AtoB$ and $BtoA$.

$Init$, $ASnd$, $BSnd$ are the same except with

$$AtoB \leftarrow AtoB2 \qquad BtoA \leftarrow BtoA2$$

$$Init \triangleq \begin{array}{l} \land\ AVar \in Data \times 1 \\ \land\ BVar = AVar \\ \land\ AtoB2 = \langle\,\rangle \\ \land\ BtoA2 = \langle\,\rangle \end{array}$$

The initial-state formula and the actions in which A and B send messages are the same except for renaming the variables $AtoB$ and $BtoA$.

Here's the initial-state formula.

$Init$, $ASnd$, $BSnd$ are the same except with

$$AtoB \leftarrow AtoB2 \qquad BtoA \leftarrow BtoA2$$

$$\begin{aligned} Init \;\triangleq\; & \land\; AVar \in Data \times 1 \\ & \land\; BVar = AVar \\ & \land\; AtoB2 = \langle\,\rangle \\ & \land\; BtoA2 = \langle\,\rangle \end{aligned}$$

$$\begin{aligned} ASnd \;\triangleq\; & \land\; AtoB2' = Append(AtoB2, AVar) \\ & \land\; \text{UNCHANGED } \langle AVar, BtoA2, BVar \rangle \end{aligned}$$

The initial-state formula and the actions in which A and B send messages are the same except for renaming the variables $AtoB$ and $BtoA$.

Here's the initial-state formula.

A's send-message action.

$Init$, $ASnd$, $BSnd$ are the same except with

$$AtoB \leftarrow AtoB2 \qquad BtoA \leftarrow BtoA2$$

$$\begin{aligned} Init \triangleq\ & \land\ AVar \in Data \times 1 \\ & \land\ BVar = AVar \\ & \land\ AtoB2 = \langle\,\rangle \\ & \land\ BtoA2 = \langle\,\rangle \end{aligned}$$

$$\begin{aligned} ASnd \triangleq\ & \land\ AtoB2' = Append(AtoB2, AVar) \\ & \land\ \text{UNCHANGED } \langle AVar, BtoA2, BVar \rangle \end{aligned}$$

$$\begin{aligned} BSnd \triangleq\ & \land\ BtoA2' = Append(BtoA2, BVar[2]) \\ & \land\ \text{UNCHANGED } \langle AVar, BVar, AtoB2 \rangle \end{aligned}$$

The initial-state formula and the actions in which A and B send messages are the same except for renaming the variables $AtoB$ and $BtoA$.

Here's the initial-state formula.

A's send-message action.

And B's send-message action.

$ARcv$ and $BRcv$ must ignore Bad messages.

The receive actions of A and B must ignore corrupted messages, which equal Bad.

$ARcv$ and $BRcv$ must ignore Bad messages.

$ARcv$ is the same as before.

$$
\begin{aligned}
ARcv \;\triangleq\; & \land\; BtoA2 \neq \langle\,\rangle \\
& \land\; \text{IF } Head(BtoA2) = AVar[2] \\
& \qquad \text{THEN } \exists\, d \in Data : AVar' = \langle d, 1 - AVar[2] \rangle \\
& \qquad \text{ELSE } AVar' = AVar \\
& \land\; BtoA2' = Tail(BtoA2) \\
& \land\; \text{UNCHANGED } \langle AtoB2,\, BVar \rangle
\end{aligned}
$$

The receive actions of A and B must ignore corrupted messages, which equal Bad.

A's receive action is the same as before, except for the change of variables.

$ARcv$ and $BRcv$ must ignore Bad messages.

$ARcv$ is the same as before.

If $Head(BtoA2) = Bad$

$$
\begin{aligned}
ARcv \triangleq\ & \land\ BtoA2 \neq \langle\, \rangle \\
& \land\ \text{IF } Head(BtoA2) = AVar[2] \\
& \qquad \text{THEN } \exists\, d \in Data : AVar' = \langle d, 1 - AVar[2] \rangle \\
& \qquad \text{ELSE } AVar' = AVar \\
& \land\ BtoA2' = Tail(BtoA2) \\
& \land\ \text{UNCHANGED } \langle AtoB2, BVar \rangle
\end{aligned}
$$

That's because if the message being received, which is at the head of the sequence of messages sent by B, equals Bad,

$ARcv$ and $BRcv$ must ignore Bad messages.

$ARcv$ is the same as before.

If $Head(BtoA2) = Bad$

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

$$
\begin{aligned}
ARcv \triangleq\ & \land\ BtoA2 \neq \langle\,\rangle \\
& \land\ \text{IF } Head(BtoA2) = AVar[2] \\
& \qquad \text{THEN } \exists\, d \in Data : AVar' = \langle d, 1 - AVar[2] \rangle \\
& \qquad \text{ELSE } AVar' = AVar \\
& \land\ BtoA2' = Tail(BtoA2) \\
& \land\ \text{UNCHANGED } \langle AtoB2, BVar \rangle
\end{aligned}
$$

That's because if the message being received, which is at the head of the sequence of messages sent by B, equals Bad,

then our assumption means that Bad doesn't equal 0 or 1,

$ARcv$ and $BRcv$ must ignore Bad messages.

$ARcv$ is the same as before.

If $Head(BtoA2) = Bad$

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

$$
\begin{array}{lll}
ARcv \triangleq & \wedge & BtoA2 \neq \langle \rangle \\
& \wedge & \text{IF } Head(BtoA2) = AVar[2] \\
& & \quad \text{THEN } \exists\, d \in Data : AVar' = \langle d, 1 - AVar[2] \rangle \\
& & \quad \text{ELSE } AVar' = AVar \\
& \wedge & BtoA2' = Tail(BtoA2) \\
& \wedge & \text{UNCHANGED } \langle AtoB2, BVar \rangle
\end{array}
$$

That's because if the message being received, which is at the head of the sequence of messages sent by B, equals Bad,

then our assumption means that Bad doesn't equal 0 or 1,

but $AVar[2]$ does equal either 0 or 1

$ARcv$ and $BRcv$ must ignore Bad messages.

$ARcv$ is the same as before.

If $Head(BtoA2) = Bad$

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

$$
\begin{aligned}
ARcv \triangleq\ & \land\ BtoA2 \neq \langle\ \rangle \\
& \land\ \text{IF } \boxed{Head(BtoA2) = AVar[2]} \quad \textbf{FALSE} \\
& \qquad \text{THEN } \exists\, d \in Data : AVar' = \langle d, 1 - AVar[2] \rangle \\
& \qquad \text{ELSE } AVar' = AVar \\
& \land\ BtoA2' = Tail(BtoA2) \\
& \land\ \text{UNCHANGED } \langle AtoB2, BVar \rangle
\end{aligned}
$$

That's because if the message being received, which is at the head of the sequence of messages sent by B, equals Bad,

then our assumption means that Bad doesn't equal 0 or 1,

but $AVar[2]$ does equal either 0 or 1

So the **if** condition is false, and the action leaves $AVar$ unchanged, meaning that A ignores the message.

$ARcv$ and $BRcv$ must ignore Bad messages.

$BRcv$ must be modified.

To ignore corrupted messages, $BRcv$ must be modified beyond just renaming the variables $AtoB$ and $BtoA$.

$ARcv$ and $BRcv$ must ignore Bad messages.

$BRcv$ must be modified.

$$
\begin{array}{lll}
BRcv & \triangleq & \land\ AtoB2 \neq \langle\ \rangle \\
 & & \land\ \text{IF } (Head(AtoB2) \neq Bad) \land (Head(AtoB2)[2] \neq BVar[2]) \\
 & & \quad\ \ \text{THEN } BVar' = Head(AtoB2) \\
 & & \quad\ \ \text{ELSE } BVar' = BVar \\
 & & \land\ AtoB2' = Tail(AtoB2) \\
 & & \land\ \text{UNCHANGED } \langle AVar,\ BtoA2 \rangle
\end{array}
$$

To ignore corrupted messages, $BRcv$ must be modified beyond just renaming the variables $AtoB$ and $BtoA$.

Here's the new definition.

$ARcv$ and $BRcv$ must ignore Bad messages.

$BRcv$ must be modified.

$$
\begin{array}{lll}
BRcv & \triangleq & \land\ AtoB2 \neq \langle\,\rangle \\
 & & \land\ \text{IF } (Head(AtoB2) \neq Bad) \land (Head(AtoB2)[2] \neq BVar[2]) \\
 & & \quad \text{THEN } BVar' = Head(AtoB2) \\
 & & \quad \text{ELSE } BVar' = BVar \\
 & & \land\ AtoB2' = Tail(AtoB2) \\
 & & \land\ \text{UNCHANGED } \langle AVar,\ BtoA2 \rangle
\end{array}
$$

To ignore corrupted messages, $BRcv$ must be modified beyond just renaming the variables $AtoB$ and $BtoA$.

Here's the new definition.

In the **if** formula

$ARcv$ and $BRcv$ must ignore Bad messages.

$BRcv$ must be modified.

$$
\begin{aligned}
BRcv \triangleq\ & \land\ AtoB2 \neq \langle\,\rangle \\
& \land\ \text{IF } (Head(AtoB2) \neq Bad) \land (Head(AtoB2)[2] \neq BVar[2]) \\
& \qquad \text{THEN } BVar' = Head(AtoB2) \\
& \qquad \text{ELSE } BVar' = BVar \\
& \land\ AtoB2' = Tail(AtoB2) \\
& \land\ \text{UNCHANGED } \langle AVar, BtoA2 \rangle
\end{aligned}
$$

To ignore corrupted messages, $BRcv$ must be modified beyond just renaming the variables $AtoB$ and $BtoA$.

Here's the new definition.

In the **if** formula this conjunct has been added to the test.

$ARcv$ and $BRcv$ must ignore Bad messages.

$BRcv$ must be modified.

Message ignored if $Head(BtoA2) = Bad$.

$$
\begin{array}{lll}
BRcv \triangleq & \land & AtoB2 \neq \langle\,\rangle \\
& \land & \text{IF } (Head(AtoB2) \neq Bad) \land (Head(AtoB2)[2] \neq BVar[2]) \\
& & \quad \text{THEN } BVar' = Head(AtoB2) \\
& & \quad \text{ELSE } BVar' = BVar \\
& \land & AtoB2' = Tail(AtoB2) \\
& \land & \text{UNCHANGED } \langle AVar, BtoA2 \rangle
\end{array}
$$

To ignore corrupted messages, $BRcv$ must be modified beyond just renaming the variables $AtoB$ and $BtoA$.

Here's the new definition.

In the **if** formula this conjunct has been added to the test.

So $BVar$ is left unchanged and the message being received is ignored if it equals Bad.

$LoseMsg$ is replaced by $CorruptMsg$

Finally, the $LoseMsg$ action is replaced by a $CurruptMsg$ action,

$LoseMsg$ is replaced by $CorruptMsg$, which changes messages in $AtoB2$ and $BtoA2$ to Bad instead of removing them.

Finally, the $LoseMsg$ action is replaced by a $CurruptMsg$ action, which changes messages in $AtoB2$ and $BtoA2$ to Bad instead of removing them.

$LoseMsg$ is replaced by $CorruptMsg$, which changes messages in $AtoB2$ and $BtoA2$ to Bad instead of removing them.

$$
\begin{aligned}
CorruptMsg \triangleq\ & \land \lor \land \exists\, i \in 1..Len(AtoB2) : \\
& \qquad\qquad AtoB2' = [AtoB2 \text{ EXCEPT } ![i] = Bad] \\
& \qquad \land BtoA2' = BtoA2 \\
& \quad \lor \land \exists\, i \in 1..Len(BtoA2) : \\
& \qquad\qquad BtoA2' = [BtoA2 \text{ EXCEPT } ![i] = Bad] \\
& \qquad \land AtoB2' = AtoB2 \\
& \land \text{UNCHANGED } \langle AVar, BVar \rangle
\end{aligned}
$$

Finally, the $LoseMsg$ action is replaced by a $CurruptMsg$ action, which changes messages in $AtoB2$ and $BtoA2$ to Bad instead of removing them.

Here is the definition, which is the same as the $LoseMsg$ action,

$LoseMsg$ is replaced by $CorruptMsg$, which changes messages in $AtoB2$ and $BtoA2$ to Bad instead of removing them.

$$
\begin{array}{lll}
CorruptMsg \triangleq & \land \lor \land \exists i \in 1..Len(AtoB2) : & \\
 & \qquad\qquad AtoB2' = [AtoB2 \text{ EXCEPT } ![i] = Bad] & \\
 & \qquad \land BtoA2' = BtoA2 & \\
 & \quad \lor \land \exists i \in 1..Len(BtoA2) : & \\
 & \qquad\qquad BtoA2' = [BtoA2 \text{ EXCEPT } ![i] = Bad] & \\
 & \qquad \land AtoB2' = AtoB2 & \\
 & \land \text{UNCHANGED } \langle AVar, BVar \rangle &
\end{array}
$$

Finally, the $LoseMsg$ action is replaced by a $CurruptMsg$ action, which changes messages in $AtoB2$ and $BtoA2$ to Bad instead of removing them.

Here is the definition, which is the same as the $LoseMsg$ action, except for these parts that describe the change to $AtoB2$ or $BtoA2$.

The definitions of $Next$ and the safety specification $Spec$ are straightforward.

The definitions of $Next$ and of the safety specification $Spec$

The definitions of $Next$ and the safety specification $Spec$ are straightforward.

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

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

The definitions of $Next$ and of the safety specification $Spec$

are what you should expect.

The definitions of $Next$ and the safety specification $Spec$ are straightforward.

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

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

Liveness is discussed later.

The definitions of $Next$ and of the safety specification $Spec$ are what you should expect.

I'll discuss liveness later.

The $AB2$ protocol is essentially the same as the AB protocol.

The $AB2$ protocol is essentially the same as the ordinary alternating bit protocol of module AB.

The $AB2$ protocol is essentially the same as the AB protocol.

It too implements the high-level safety specification in module $ABSpec$.

The $AB2$ protocol is essentially the same as the ordinary alternating bit protocol of module AB.

As we expect, it too implements the high-level safety specification of the protocol in module $ABSpec$.

This is expressed in module $AB2$ the same as in module AB,

The $AB2$ protocol is essentially the same as the AB protocol.

It too implements the high-level safety specification in module $ABSpec$.

$$ABS \triangleq \text{INSTANCE } ABSpec$$

by importing module $ABSpec$ with renaming

The $AB2$ protocol is essentially the same as the AB protocol.

It too implements the high-level safety specification in module $ABSpec$.

$$ABS \triangleq \text{INSTANCE } ABSpec$$

$$\text{THEOREM } Spec \Rightarrow ABS!Spec$$

by importing module $ABSpec$ with renaming

and stating this theorem.

CHECKING AB2

Now check that the $AB2$ protocol implements the high-level safety spec of module $ABSpec$.

You should now check that the $AB2$ protocol implements the high-level safety spec of module $ABSpec$.

Now check that the $AB2$ protocol implements the high-level safety spec of module $ABSpec$.

As with the AB spec, a model must provide:

As with the AB spec, a model must provide two things:

Now check that the $AB2$ protocol implements the high-level safety spec of module $ABSpec$.

As with the AB spec, a model must provide:

– A value for the constant $Data$.

As with the AB spec, a model must provide two things:

First, it has to provide A value for the constant $Data$.

Now check that the $AB2$ protocol implements the high-level safety spec of module $ABSpec$.

As with the AB spec, a model must provide:

– A value for the constant $Data$.

Use a set $\{d1, d2, d3\}$ of model values.

As with the AB spec, a model must provide two things:

First, it has to provide A value for the constant $Data$.
For example, you can use this set of three model values.

Now check that the $AB2$ protocol implements the high-level safety spec of module $ABSpec$.

As with the AB spec, a model must provide:

– A value for the constant $Data$.

Use a set $\{d1, d2, d3\}$ of model values.

– A state constraint to bound the lengths of $AtoB2$ and $BtoA2$.

As with the AB spec, a model must provide two things:

First, it has to provide A value for the constant $Data$.
For example, you can use this set of three model values.

Second, it must provide a state constraint to bound the lengths of the sequences $AtoB2$ and $BtoA2$.

Now check that the $AB2$ protocol implements the high-level safety spec of module $ABSpec$.

As with the AB spec, a model must provide:

- A value for the constant $Data$.

  Use a set $\{d1, d2, d3\}$ of model values.

- A state constraint to bound the lengths of $AtoB2$ and $BtoA2$.

  Use $(AtoB2 < 4) \land (BtoA2 < 4)$.

As with the AB spec, a model must provide two things:

First, it has to provide A value for the constant $Data$.
For example, you can use this set of three model values.

Second, it must provide a state constraint to bound the lengths of the sequences $AtoB2$ and $BtoA2$.
You can constrain them both to have length less than four.

A model of $AB2$ most also specify a value for Bad.

A model of specification $AB2$ most also specify a value for the constant Bad.

A model of $AB2$ most also specify a value for Bad.

It must satisfy

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

A model of specification $AB2$ most also specify a value for the constant Bad.

That value must satisfy the module's assumption,

A model of $AB2$ most also specify a value for Bad.

It must satisfy

$$\text{ASSUME } Bad \notin (\overset{\{\texttt{d1,d2,d3}\}}{\cancel{Data}} \times \{0,1\}) \cup \{0,1\}$$

A model of specification $AB2$ most also specify a value for the constant Bad.

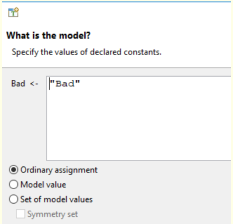
That value must satisfy the module's assumption,
when $Data$ also equals the value the model assigns to it.

A model of $AB2$ most also specify a value for Bad.

It must satisfy

$$\text{ASSUME } Bad \notin (\overset{\{\texttt{d1,d2,d3}\}}{\cancel{Data}} \times \{0,1\}) \cup \{0,1\}$$

An obvious choice:



A model of specification $AB2$ most also specify a value for the constant Bad.

That value must satisfy the module's assumption,
when $Data$ also equals the value the model assigns to it.

An obvious choice is to let the model assign the string quote-bad to the constant Bad.

Running the model produces this TLC error:

```
Attempted to check equality of integer 0 with
non-integer: "Bad"
```

But running the model produces this TLC error: Attempted to check equality of integer 0 with non-integer quote-bad.

Running the model produces this TLC error:

```
Attempted to check equality of integer 0 with
value I don’t know to be an integer: "Bad"
```

But running the model produces this TLC error: Attempted to check equality of integer 0 with non-integer quote-bad.

What TLC really means is that it tried to check if 0 equals the value quote-bad, and it doesn’t even know whether or not that value is an integer.

Running the model produces this TLC error:

```
Attempted to check equality of integer 0 with
non-integer: "Bad"
```

We think that “Bad” and 0 are different

We naturally think that “Bad” and 0 are different,

Running the model produces this TLC error:

```
Attempted to check equality of integer 0 with
non-integer: "Bad"
```

We think that "Bad" and 0 are different, but the semantics of TLA$^+$ don't say that they are.

We naturally think that "Bad" and 0 are different,

But the semantics of TLA$^+$ doesn't specify that they're different. So TLC doesn't know whether or not they're equal.

Running the model produces this TLC error:

```
Attempted to check equality of integer 0 with
non-integer: "Bad"
```

We think that "Bad" and 0 are different, but the semantics of TLA+ don't say that they are.

What value of Bad satisfies

$$Bad \notin (\overset{\{\texttt{d1,d2,d3}\}}{\cancel{Data}} \times \{0, 1\}) \cup \{0, 1\}$$?

We naturally think that "Bad" and 0 are different,
But the semantics of TLA+ doesn't specify that they're different. So TLC doesn't know whether or not they're equal.

What value of Bad does satisfy this condition?

We don't know, and we don't need to know. To define the model, all we need to know is:

Running the model produces this TLC error:

```
Attempted to check equality of integer 0 with
non-integer: "Bad"
```

We think that "Bad" and 0 are different, but the semantics of TLA+ don't say that they are.

What value of Bad does TLC think satisfies

$$Bad \notin (\underset{\{\texttt{d1,d2,d3}\}}{\cancel{Data}} \times \{0, 1\}) \cup \{0, 1\} \quad ?$$

What value does *TLC* ***think*** satisfies the condition? And the answer to *that* question is:

Running the model produces this TLC error:

```
Attempted to check equality of integer 0 with
non-integer: "Bad"
```

We think that "Bad" and 0 are different, but the semantics of TLA+ don't say that they are.

What value of Bad does TLC think satisfies

$$Bad \notin (\overset{\{\texttt{d1,d2,d3}\}}{\cancel{Data}} \times \{0, 1\}) \cup \{0, 1\}$$?

A model value.

What value does *TLC* ***think*** satisfies the condition? And the answer to *that* question is:

A model value.

TLC assumes a model value does not equal any value that you might expect it to be different from.

TLC assumes a model value does not equal any value that you would expect it to be different from.

You don't need to know precisely what that means.

TLC assumes a model value does not equal any value that you might expect it to be different from.

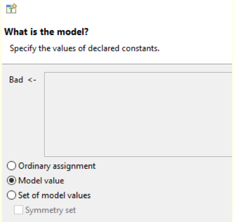
It's convenient to let the constant Bad equal the model value `Bad`.

It's convenient to have the model assign to the constant Bad the model value of the same name.

TLC assumes a model value does not equal any value that you might expect it to be different from.

It's convenient to let the constant Bad equal the model value `Bad`.

Here's how:



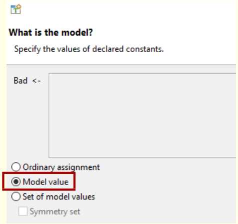
It's convenient to have the model assign to the constant Bad the model value of the same name.

To do that, in the window for assigning a value to the constant,

TLC assumes a model value does not equal any value that you might expect it to be different from.

It's convenient to let the constant Bad equal the model value `Bad`.

Here's how:



It's convenient to have the model assign to the constant Bad the model value of the same name.

To do that, in the window for assigning a value to the constant, just select the model value option

You can now run TLC to check that the $AB2$ specification implements the specification of module $ABSpec$.

You can now run TLC to check that the $AB2$ specification implements the specification of module $ABSpec$.

LIVENESS OF AB2

Module $AB2$ next defines $FairSpec$ to be the obvious analogue of $FairSpec$ of AB.

Module $AB2$ next defines $FairSpec$ to be the obvious analogue of formula $FairSpec$ of module AB.

Module $AB2$ next defines $FairSpec$ to be the obvious analogue of $FairSpec$ of AB.

But it doesn't implement $ABS!FairSpec$.

Module $AB2$ next defines $FairSpec$ to be the obvious analogue of formula $FairSpec$ of module AB.

But this specification $FairSpec$ doesn't implement the high-level specification $FairSpec$ of module $ABSpec$.

Module $AB2$ next defines $FairSpec$ to be the obvious analogue of $FairSpec$ of AB .

But it doesn't implement $ABS!FairSpec$.

Fairness requirements on subactions of $Next$ can't guarantee that any messages are received before they're corrupted.

Module $AB2$ next defines $FairSpec$ to be the obvious analogue of formula $FairSpec$ of module AB .

But this specification $FairSpec$ doesn't implement the high-level specification $FairSpec$ of module $ABSpec$.

I believe that fairness requirements on subactions of $Next$ cannot guarantee that any messages are received before they're corrupted.

Module $AB2$ next defines $FairSpec$ to be the obvious analogue of $FairSpec$ of AB.

But it doesn't implement $ABS!FairSpec$.

Fairness requirements on subactions of $Next$ can't guarantee that any messages are received before they're corrupted.

To do that, we change the safety spec.

To guarantee that, we change the safety spec.

Sending a message adds something to the state that determines if the message can be corrupted.

We let sending a message add something to the state that determines if the message can be corrupted.

Sending a message adds something to the state
that determines if the message can be corrupted.

We could add a component to each message.

We let sending a message add something to the state that determines if the message can be corrupted.

We could add a component to each message. For example

Sending a message adds something to the state that determines if the message can be corrupted.

We could add a component to each message.

$\langle$"Tom", 0, TRUE$\rangle$

message *cannot* be corrupted

We let sending a message add something to the state that determines if the message can be corrupted.

We could add a component to each message. For example We could let a component with value TRUE mean that the message *cannot* be corrupted.

Sending a message adds something to the state that determines if the message can be corrupted.

We could add a component to each message.

$\langle$"Tom", 0, TRUE$\rangle$
message *cannot* be corrupted

$\langle$"Tom", 0, FALSE$\rangle$
message *can* be corrupted

We let sending a message add something to the state that determines if the message can be corrupted.

We could add a component to each message. For example We could let a component with value TRUE mean that the message *cannot* be corrupted. And let a component with value FALSE mean that the message *can* be corrupted.

Sending a message adds something to the state
that determines if the message can be corrupted.

We could add a component to each message.

An imaginary component that’s not meant to be implemented

It’s an imaginary component that’s not meant to be implemented

Sending a message adds something to the state
that determines if the message can be corrupted.

We could add a component to each message.

An imaginary component that’s not meant to be implemented
and serves only to specify liveness.

It’s an imaginary component that’s not meant to be implemented

and serves only to specify liveness.

It’s best to keep the real and imaginary
parts of the state separate

It’s best to keep the real and imaginary parts of the state separate

It's best to keep the real and imaginary parts of the state separate by putting them in different variables.

It's best to keep the real and imaginary parts of the state separate

by putting them in different variables.

It's best to keep the real and imaginary parts of the state separate by putting them in different variables.

Instead of

$AtoB2: \quad \langle \langle \text{“}Tom\text{”}, 0, \text{TRUE} \rangle, \langle \text{“}Tom\text{”}, 0, \text{FALSE} \rangle, \langle \text{“}Fred\text{”}, 0, \text{FALSE} \rangle \rangle$

It's best to keep the real and imaginary parts of the state separate by putting them in different variables.

Instead of adding an imaginary component to the messages in $AtoB2$,

It's best to keep the real and imaginary parts of the state separate by putting them in different variables.

Instead of

$AtoB2$: $\langle\, \langle \text{“}Tom\text{”}, 0, \text{TRUE} \rangle,\ \langle \text{“}Tom\text{”}, 0, \text{FALSE} \rangle,\ \langle \text{“}Fred\text{”}, 0, \text{FALSE} \rangle \,\rangle$

we have

$AtoB2$: $\langle\, \langle \text{“}Tom\text{”}, 0 \rangle,\ \langle \text{“}Tom\text{”}, 0 \rangle,\ \langle \text{“}Fred\text{”}, 1 \rangle \,\rangle$

It's best to keep the real and imaginary parts of the state separate

by putting them in different variables.

Instead of adding an imaginary component to the messages in $AtoB2$,

We have the same messages in $AtoB2$

It's best to keep the real and imaginary parts of the state separate by putting them in different variables.

Instead of

$AtoB2$: $\langle \langle \text{“}Tom\text{”}, 0, \text{TRUE} \rangle, \langle \text{“}Tom\text{”}, 0, \text{FALSE} \rangle, \langle \text{“}Fred\text{”}, 0, \text{FALSE} \rangle \rangle$

we have

$AtoB2$: $\langle \langle \text{“}Tom\text{”}, 0 \rangle, \langle \text{“}Tom\text{”}, 0 \rangle, \langle \text{“}Fred\text{”}, 1 \rangle \rangle$

$AtoBgood$: $\langle \text{TRUE}, \text{FALSE}, \text{FALSE} \rangle$

It's best to keep the real and imaginary parts of the state separate

by putting them in different variables.

Instead of adding an imaginary component to the messages in $AtoB2$,

We have the same messages in $AtoB2$ and put the sequence of their imaginary components into a separate variable $AtoBgood$.

It’s best to keep the real and imaginary parts of the state separate by putting them in different variables.

And we similarly have $BtoA2$ and $BtoAgood$.

And we similarly have $BtoA2$ and the imaginary variable $BtoAgood$.

The resulting safety specification $SpecP$ is defined in a module named $AB2P$, which EXTENDS module $AB2$.

The resulting specification $SpecP$ is defined in a module named $AB2P$, which EXTENDS module $AB2$.

The resulting safety specification $SpecP$ is defined in a module named $AB2P$, which EXTENDS module $AB2$.

$AtoBgood$ and $BtoAgood$ are imaginary variables, not meant to be implemented.

The resulting specification $SpecP$ is defined in a module named $AB2P$, which EXTENDS module $AB2$.

The variables $AtoBgood$ and $BtoAgood$ are imaginary variables; they're not meant to be implemented.

The resulting safety specification $SpecP$ is defined in a module named $AB2P$, which EXTENDS module $AB2$.

$AtoBgood$ and $BtoAgood$ are imaginary variables, not meant to be implemented. They are used only for defining the fairness requirements.

The resulting specification $SpecP$ is defined in a module named $AB2P$, which EXTENDS module $AB2$.

The variables $AtoBgood$ and $BtoAgood$ are imaginary variables; they're not meant to be implemented.

They are used only for defining the fairness requirements.

The resulting safety specification $SpecP$ is defined in a module named $AB2P$, which EXTENDS module $AB2$.

$AtoBgood$ and $BtoAgood$ are imaginary variables, not meant to be implemented. They are used only for defining the fairness requirements.

Deciding in advance if a message can be deleted doesn't change the values the variables of $AB2$ can assume.

Deciding in advance if a message can be deleted doesn't change the values that the variables of $AB2$ can assume.

The resulting safety specification $SpecP$ is defined in a module named $AB2P$, which EXTENDS module $AB2$.

$AtoBgood$ and $BtoAgood$ are imaginary variables, not meant to be implemented. They are used only for defining the fairness requirements.

Deciding in advance if a message can be deleted doesn't change the values the variables of $AB2$ can assume.

So if we ignore the values of $AtoBgood$ and $BtoAgood$,

Deciding in advance if a message can be deleted doesn't change the values that the variables of $AB2$ can assume.

So if we ignore the values of the imaginary variables $AtoBgood$ and $BtoAgood$,

The resulting safety specification $SpecP$ is defined in a module named $AB2P$, which EXTENDS module $AB2$.

$AtoBgood$ and $BtoAgood$ are imaginary variables, not meant to be implemented. They are used only for defining the fairness requirements.

Deciding in advance if a message can be deleted doesn't change the values the variables of $AB2$ can assume.

So if we ignore the values of $AtoBgood$ and $BtoAgood$, then $Spec$ and $SpecP$ allow the same behaviors.

Deciding in advance if a message can be deleted doesn't change the values that the variables of $AB2$ can assume.

So if we ignore the values of the imaginary variables $AtoBgood$ and $BtoAgood$, then specifications $Spec$ and $SpecP$ allow the same behaviors.

You can read the definitions of $SpecP$ and of specification $FairSpecP$ with fairness requirements in module $AB2P$.

You can read the definitions of $SpecP$ and of the specification $FairSpecP$ with fairness requirements in module $AB2P$.

You can read the definitions of $SpecP$ and of specification $FairSpecP$ with fairness requirements in module $AB2P$.

Stop the video and download it now.

You can read the definitions of $SpecP$ and of the specification $FairSpecP$ with fairness requirements in module $AB2P$.

Stop the video and download that module now.

Our discussion of liveness of the AB2 protocol stops here. The second part of this lecture considers only the protocol's safety spec, explaining the precise sense in which it implements the safety spec of the AB protocol, and how to check that it does. Imaginary variables will appear again.

End of Lecture 10, Part 1

IMPLEMENTATION WITH REFINEMENT

PRELIMINARIES