Modeling and Verifying Distributed Algorithms Using TLA+

http://d3s.mff.cuni.cz

Courtesy of Stephan Merz

https://members.loria.fr/Stephan.Merz/
Leslie Lamport

PhD 1972 (Brandeis University), Mathematics

- Mitre Corporation, 1962–65
- Marlboro College, 1965–69
- SRI International, 1977–85
- Microsoft Research, since 2001

Pioneer of distributed algorithms  Turing Award 2013

- Natl. Acad. of Sciences, PODC Influential Paper, ACM SIGOPS Hall of Fame (3x), LICS Award, John v. Neumann medal, E.W. Dijkstra Prize, . . .
**TLA⁺ AS A FORMAL METHOD**

- Mathematical language for modeling systems
  - represent data structures as sets and functions
  - specify system dynamics and properties using temporal logic

- **TLA⁺ tools available from the TLA⁺ Toolbox**
  - TLC: explicit-state model checking
  - TLAPS: interactive theorem proving
  - PlusCal: algorithmic language, generates TLA⁺ specification

- Intended for high-level models
  - designs of distributed and concurrent algorithms
  - no link to actual implementations (so far)

- Objective: think about your design before you start implementing
INDUSTRIAL APPLICATIONS

Amazon
  - Web services
    - https://cacm.acm.org/magazines/2015/4/184701-how-amazon-web-services-uses-formal-methods/fulltext

OpenComRTOS
  - OS used in ESA Rosetta spacecraft

Intel
  - Cache coherence protocol
    - https://dl.acm.org/doi/10.1145/1391469.1391675
Example: an hour clock

```plaintext
MODULE HourClock

EXTENDS Naturals
VARIABLE hr

HCini △ hr ∈ (0..23)
HCnxt △ hr' = IF hr = 23 THEN 0 ELSE hr + 1
HCsafe △ HCini ∧ □[HCnxt]_{hr}

THEOREM HCsafe → □HCini
```
The hour clock gives rise to the following transition system:

- all states are initial
- stuttering and “tick” actions
- all states reachable, no deadlocks
The module *HourClock* contains declarations and definitions

- \( hr \) a state variable
- \( HCini \) a state predicate
- \( HCnxt \) an action (built from \( hr \) and \( hr' \))
- \( HCsafe \) a temporal formula specifying that
  - the initial state satisfies \( HCini \)
  - every transition satisfies \( HCnxt \) or leaves \( hr \) unchanged

Module *HourClock* also asserts a theorem: \( HCsafe \rightarrow \Box HCini \)

This invariant can be verified using TLC, the TLA\(^+\) model checker.

**Note:**

- the hour clock may eventually stop ticking
- it must not fail in any other way
A TLA\(^+\) formula

\[ \text{Init} \land \Box [\text{Next}]_v \]

specifies the initial states and the allowed transitions of a system. It allows for transitions that do not change \(v\): stuttering transitions. Infinite stuttering can be excluded by asserting fairness conditions. For example,

\[ HC \triangleq HC_{ini} \land \Box [HC_{nxt}]_{hr} \land WF_{hr}HC_{nxt} \]

specifies an hour clock that never stops ticking.
Distributed Commitment

The Two-Phase Commitment Protocol

Liveness Properties

More On TLA+ Expressions

Model Checking Large Specifications

Summing Up

Case Study: Distributed Computation Of A Spanning Tree
**Problem Statement**

**Distributed commitment.**

A set of nodes has to agree whether to commit or abort a transaction.

- Initially, each node decides if it wishes to commit or abort.
- The transaction is committed if all nodes wish to commit. Otherwise, it is aborted.
**PROBLEM STATEMENT**

**Distributed commitment.**
A set of nodes has to agree whether to commit or abort a transaction.
- Initially, each node decides if it wishes to commit or abort.
- The transaction is committed if all nodes wish to commit. Otherwise, it is aborted.

**Control flow of each node**

```
preparing
  └─── proposeCommit
  └─── proposeAbort
```
**Problem Statement**

**Distributed commitment.**
A set of nodes has to agree whether to commit or abort a transaction.

- Initially, each node decides if it wishes to commit or abort.
- The transaction is committed if all nodes wish to commit. Otherwise, it is aborted.

Control flow of each node

```
proposeCommit
  ▼ all nodes are ready to commit
  △ preparing

proposeAbort
  ▼ some node wants to abort
  △ preparing

committed
aborted
```
Write a bird’s eyes view specification

- describe just how the participants’ states may change
- consider an observer that has complete information
- don’t care about distributed implementability
A FIRST TLA+ SPECIFICATION

- Write a bird’s eyes view specification
  - describe just how the participants’ states may change
  - consider an observer that has complete information
  - don’t care about distributed implementability

- We’ll later “localize” the specification
  - the central view usually results in the simplest specification
  - document the externally visible behavior, however it is achieved
  - a distributed algorithm will implement the centralized specification
**BIRD’S EYES SPECIFICATION IN TLA+**

**MODULE DistributedCommit**

**CONSTANT** Node  
**VARIABLE** nState  

**Init** $\triangleq$ $nState = [n \in \text{Node} \mapsto \text{“preparing”}]$

**Decide**($n$) $\triangleq$
- $\lor nState[n] = \text{“preparing”} \land nState'[n] = [nState \text{ EXCEPT } ![n] = \text{“proposeCommit”}]$
- $\lor nState[n] = \text{“preparing”} \land nState'[n] = [nState \text{ EXCEPT } ![n] = \text{“proposeAbort”}]$

**Commit**($n$) $\triangleq$
- $\land \forall q \in \text{Node} : nState[q] \in \{\text{“proposeCommit”, “committed”}\}$
- $\land nState'[n] = [nState \text{ EXCEPT } ![n] = \text{“committed”}]$

**Abort**($n$) $\triangleq$
- $\land \exists q \in \text{Node} : nState[q] \in \{\text{“proposeAbort”, “aborted”}\}$
- $\land nState'[n] = [nState \text{ EXCEPT } ![n] = \text{“aborted”}]$

**Next** $\triangleq$ $\exists n \in \text{Node} : \text{Decide}(n) \lor \text{Commit}(n) \lor \text{Abort}(n)$

**Spec** $\triangleq$ $\text{Init} \land \Box[\text{Next}]nState$
Data model

- parameter Node represents the set of nodes
- variable nState models the state of each participant
- represented as a function (a.k.a. array) mapping nodes to states
Data model
- parameter *Node* represents the set of nodes
- variable *nState* models the state of each participant
- represented as a function (a.k.a. array) mapping nodes to states

State-based specification
- main formula *Spec* describes set of executions
- execution (behavior): infinite sequence of states
- state: assigns values to variables
**Data model**
- Parameter *Node* represents the set of nodes.
- Variable *nState* models the state of each participant.
- Represented as a function (a.k.a. array) mapping nodes to states.

**State-based specification**
- Main formula *Spec* describes set of executions.
- Execution (behavior): infinite sequence of states.
- State: assigns values to variables.

**Describing a state machine in TLA+**
- Formula *Init* expresses initial condition.
- *Decide*(n), *Commit*(n), *Abort*(n) represent node transitions.
- Transition relation *Next*: disjunction of individual transitions.
**VALUES IN TLA+**

- TLA+ is an untyped, set-based formalism
  - we don’t have to specify that Node is a set
  - in fact, every value of TLA+ is a set
  - even numbers and strings are sets
    - but we don’t care what the elements of these sets are
  - (not just) in this respect, TLA+ follows classical mathematics

What about type errors?

"silly" expressions such as 42 + {} are accepted by the parser
the value of such expressions is not specified
TLC will report an error when it tries to evaluate a silly expression

Deemed acceptable: specifications are short (200 – 800 lines)
**Values in TLA⁺**

- TLA⁺ is an untyped, set-based formalism
  - we don’t have to specify that *Node* is a set
  - in fact, every value of TLA⁺ is a set
  - even numbers and strings are sets
    - but we don’t care what the elements of these sets are
  - (not just) in this respect, TLA⁺ follows classical mathematics

- What about type errors?
  - “silly” expressions such as $42 + \{\}$ are accepted by the parser
  - the value of such expressions is not specified
  - TLC will report an error when it tries to evaluate a silly expression

- Deemed acceptable: specifications are short (200–800 lines)
Which of the following formulas are true?

- $\forall n \in \text{Nat} : n > 0$  
  
  false: $0 \in \text{Nat}$

- $\exists k \in \text{Nat} : k + k = 7$  
  
  false: $k + k$ is even, for all $k \in \text{Nat}$

- $\forall n \in \text{Nat} : n + n = 4 \Rightarrow n \times n = 4$  
  
  true: $n + n = 4 \Rightarrow n = 2$

- $\exists n \in \text{Nat} : n + n = 4 \Rightarrow n = 3$  
  
  true, e.g. $1 + 1 \neq 4$

- $\forall x \in \{\} : \text{“Dublin”} = \text{“Nancy”}$  
  
  true: trivial quantifier range

- $\exists x \in \{\} : x = x$  
  
  false: no $x \in \{\}$

- $\neg (\exists x \in S : P(x)) \equiv (\forall x \in S : \neg P(x))$  
  
  true

- $0 \div 0 = 1$  
  
  unspecified

- $42 \land \text{“xyz”}$  
  
  unspecified

The last two formulas are “silly”: TLC will raise an exception

- silly formulas are not illegal: they may occur as sub-expressions

- $\forall n \in \text{Nat} : n \neq 0 \Rightarrow n \div n = 1$
**FUNCTIONAL VALUES**

### Functions in TLA^+

<table>
<thead>
<tr>
<th>Programming</th>
<th>Mathematics</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>function</td>
</tr>
<tr>
<td>index set 0..N</td>
<td>function domain (any set)</td>
</tr>
<tr>
<td>array selection a[i]</td>
<td>function application a(i)</td>
</tr>
</tbody>
</table>
FUNCTIONAL VALUES

Functions in TLA+

- **programming**
  - array
  - index set $0..N$
  - array selection $a[i]$

- **mathematics**
  - function
  - function domain (any set)
  - function application $a(i)$

- TLA$^+$ is mathematics, but writes $a[i]$ for function application
- parentheses are used for operator application, e.g. $Decide(p)$
FUNCTIONAL VALUES

Functions in TLA⁺

- programming vs. mathematics
- array function
- index set 0..N
- array selection a[i]
- function domain (any set)
- function application a(i)

TLA⁺ is mathematics, but writes a[i] for function application
- parentheses are used for operator application, e.g. Decide(p)

Notations used with functions

- \[ S \rightarrow T \] set of functions with domain S and values in T
- \[ x \in S \mapsto e \] function mapping every \( x \in S \) to e
- \[ f \text{ EXCEPT } ![x] = e \] function mapping every \( x \in S \) to e
- \((a:x) @@ (b:y)\) finite function mapping a to x, b to y (module TLC)
FUNCTIONAL VALUES

- **Functions in TLA⁺**
  - **programming**
    - array
    - index set $0..N$
    - array selection $a[i]$
  - **mathematics**
    - function
    - function domain (any set)
    - function application $a(i)$
  - TLA⁺ is mathematics, but writes $a[i]$ for function application
  - parentheses are used for operator application, e.g. $\text{Decide}(p)$

- **Notations used with functions**
  - $[S \rightarrow T]$ set of functions with domain $S$ and values in $T$
  - `DOMAIN f` domain of function $f$
  - $[x \in S \mapsto e]$ function mapping every $x \in S$ to $e$
  - $[f \text{ EXCEPT } ![x] = e]$ $[y \in \text{DOMAIN } f \mapsto \text{IF } y = x \text{ THEN } e \text{ ELSE } f[x]]$
  - $(a:x) @@ (b:y)$ finite function mapping $a$ to $x$, $b$ to $y$ (module TLC)
  - refer to previous value: $[f \text{ EXCEPT } ![x] = @ + 1]$
SPECIFYING ACTIONS

- Actions must completely specify the successor states
  - relation between pre-state and post-state (primed variables)
  - write \( v' = v \) (a.k.a. UNCHANGED \( v \)) if variable \( v \) doesn’t change

- Basic format of an action definition

\[
A(p) \triangleq \wedge guard(p, \bar{v}) \quad \triangleright \text{pre-condition}
\wedge v'_1 = exp_1(p, \bar{v}) \quad \triangleright \text{variable update}
\wedge v'_2 \in exp_2(p, \bar{v}) \quad \triangleright \text{non-determinism}
\wedge \text{UNCHANGED} \langle v_3, \ldots, v_n \rangle
\]

- \( guard \): state predicate, determines when action can be taken
- \( exp_i \): state function, computes new value(s) of variable \( v_i \)
- more complicated actions: case distinction, quantifiers, . . .
Cannot define action $\text{Commit}(n)$ as

\[ \forall q \in \text{Node} : nState[q] \in \{ \text{"readyCommit"}, \text{"committed"} \} \]
\[ nState[n]' = \text{"committed"} \]

- does not specify $nState[q]'$ for $q \neq n$
- does not even say that $nState'$ is a function
How To Specify Function Updates

- Cannot define action $\text{Commit}(n)$ as

$$\land \land q \in \text{Node} : n\text{State}[q] \in \{\text{"readyCommit"}, \text{"committed"}\}$$

$$\land n\text{State}[n]' = \text{"committed"}$$

- does not specify $n\text{State}[q]'$ for $q \neq n$
- does not even say that $n\text{State}'$ is a function

The new value of the function must be specified completely

- in general, write $n\text{State}' = [q \in \text{Node} \mapsto \ldots]$
- use EXCEPT expression if only one (or a few) values are updated

$$n\text{State}' = [n\text{State} \text{ EXCEPT } !n = \text{"committed"}]$$
Verifying Properties of Distributed Commitment

- **Type correctness**

  \[ NState \overset{\Delta}{=} \{ \text{"preparing"}, \text{"proposeCommit"}, \text{"proposeAbort"}, \text{"committed"}, \text{"aborted"} \} \]

  \[ TypeOK \overset{\Delta}{=} nState \in [Node \rightarrow NState] \]

- **Nodes can commit only if all accept**

  \[ Agreement \overset{\Delta}{=} \forall p \in Node : nState[p] = \text{"committed"} \]

  \[ \Rightarrow \forall q \in Node : nState[q] \in \{ \text{"proposeCommit"}, \text{"committed"} \} \]
VERIFYING PROPERTIES OF DISTRIBUTED COMMITMENT

- **Type correctness**
  
  \[ NState \triangleq \{ \text{“preparing”, “proposeCommit”, “proposeAbort”, “committed”, “aborted”} \} \]
  
  \[ TypeOK \triangleq nState \in [Node \rightarrow NState] \]

- **Nodes can commit only if all accept**
  
  \[ Agreement \triangleq \forall p \in Node : nState[p] = \text{“committed”} \]
  
  \[ \Rightarrow \forall q \in Node : nState[q] \in \{ \text{“proposeCommit”, “committed”} \} \]

- **These properties are easily verified using the TLC model checker**
  
  - create finite model by instantiating parameter Node
  - for example: Node ← \{1, 2, 3, 4, 5\}
  - can also use model values: Node ← \{alice, bob, charlie\}
  - check invariants TypeOK, Agreement
Distributed Commitment

The Two-Phase Commitment Protocol

Liveness Properties

More On TLA+ Expressions

Model Checking Large Specifications

Summing Up

Case Study: Distributed Computation Of A Spanning Tree
The current specification cannot be directly implemented
- nodes in a distributed system cannot access states of other nodes
- introduce explicit communication by message passing
The current specification cannot be directly implemented
- nodes in a distributed system cannot access states of other nodes
- introduce explicit communication by message passing

Standard solution: two-phase commitment
- make use of a coordinator who centralizes agreement

<table>
<thead>
<tr>
<th>alice</th>
<th>bob</th>
<th>charlie</th>
<th>coordinator</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The current specification cannot be directly implemented
- nodes in a distributed system cannot access states of other nodes
- introduce explicit communication by message passing

Standard solution: two-phase commitment
- make use of a coordinator who centralizes agreement
The current specification cannot be directly implemented
- nodes in a distributed system cannot access states of other nodes
- introduce explicit communication by message passing

**Standard solution: two-phase commitment**
- make use of a coordinator who centralizes agreement

```
<table>
<thead>
<tr>
<th>alice</th>
<th>bob</th>
<th>charlie</th>
<th>coordinator</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;commit&quot;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&quot;abort&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;commit&quot;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>&quot;doAbort&quot;</td>
</tr>
</tbody>
</table>
```
MODELING COMMUNICATION IN TLA⁺

- TLA⁺ has no built-in primitives for message passing
  - no unique, generally accepted communication model
  - message loss and duplication, ordering guarantees etc.

- Use a variable that explicitly models the communication network
  - for example: sets vs. sequences for (un)ordered communication
  - different communication models can be provided by libraries

- For two-phase commit protocol
  - represent messages as records of message kind and additional data
  - represent network as set of messages: no ordering is assumed
  - messages are sent once, assume no message loss
A TLA+ record corresponds to a struct in C

- represented as a function whose domain is a set of strings
- a record with two fields: \([name \mapsto \text{"fred"}, \ age \mapsto 23]\)
- equals ("name" : >  "fred") @@ ("age" : > 23)
A TLA+ record corresponds to a struct in C
- represented as a function whose domain is a set of strings
- a record with two fields: \([\text{name} \mapsto \text{fred}, \text{age} \mapsto 23]\)
- equals \((\text{name} :> \text{fred}) \&\& (\text{age} :> 23)\)

Notation used with records
- set of records of certain shape: \([\text{name} : \text{STRING}, \text{age} : 0..120]\)
- record access: \(\text{rec.name}\) abbreviates \(\text{rec[“name”]}\)
- record update: \([\text{rec EXCEPT !.age = @ + 1}]\)
**TLA⁺ RECORDS AND TUPLES**

- **A TLA⁺ record** corresponds to a struct in C
  - represented as a function whose domain is a set of strings
  - a record with two fields: 
    - `[name ↦ “fred”, age ↦ 23]`
  - equals ("name" ↦ "fred") @@ (“age” ↦ 23)

- **Notation used with records**
  - set of records of certain shape: 
    - `[name : STRING, age : 0..120]`
  - record access: `rec.name` abbreviates `rec[“name”]`
  - record update: `[rec EXCEPT !.age = @ + 1]`

- **n-tuples (sequences)** are also represented as functions
  - `<42, {}, “abc”>` is a function with domain 1..3
  - `<>` denotes the empty tuple
  - use function application for projection, e.g. `seq[2]`
  - cf. frequent idiom in action definitions `UNCHANGED ⟨x, y, z⟩`
Functions Versus Operators

What’s the difference between \( F(x) \) and \( f[x] \)?

\[
F(x) \overset{\Delta}{=} e(x) \quad \text{vs.} \quad f[x \in S] \overset{\Delta}{=} e(x)
\]

- functions have a fixed domain, operators do not
- operators are not values: cannot be stored in variables

Operators (and functions) can be passed as arguments

\[
\text{IsPartialOrder}(S, R(x, y)) \overset{\Delta}{=} \land \forall x \in S: R(x, x) \land \forall x, y \in S: R(x, y) \land R(y, x) \Rightarrow x = y \land \forall x, y, z \in S: R(x, y) \land R(y, z) \Rightarrow R(x, z)
\]
Functions Versus Operators

- What’s the difference between $F(x)$ and $f[x]$?

  $F(x) \triangleq e(x)$  vs.  $f[x \in S] \triangleq e(x)$

  - functions have a fixed domain, operators do not
  - operators are not values: cannot be stored in variables

- When should you prefer one over the other?

  - must use functions for values manipulated by the specification
    - analogous to arrays in programming
  - must use operators if you cannot specify the domain
    - e.g., operators applicable to arbitrary sequences or sets
Two-Phase Commit in TLA$^+$ (1)

MODULE TwoPhaseCommit

CONSTANT Node
VARIABLES cState, nState, committed, msgs

vars $\triangleq$ $\langle$cState, nState, committed, msgs$\rangle$

Message $\triangleq$ $\{kind : \{"commit", "abort"\}, node : Node\}$

$\cup$ $\{kind : \{"doCommit", "doAbort"\}\}$

commit$(n)$ $\triangleq$ $\{kind \mapsto \"commit\", node \mapsto n\}$

abort$(n)$ $\triangleq$ $\{kind \mapsto \"abort\", node \mapsto n\}$

doCommit $\triangleq$ $\{kind \mapsto \"doCommit\"\}$

doAbort $\triangleq$ $\{kind \mapsto \"doAbort\"\}$
Two-Phase Commit in TLA$^+$ (1)

**Module TwoPhaseCommit**

**CONSTANT** Node

**VARIABLES** cState, nState, committed, msgs

vars $\triangleq \langle cState, nState, committed, msgs \rangle$

**Message** $\triangleq [ \text{kind} : \{ \text{“commit”}, \text{“abort”} \}, \text{node} : \text{Node} ]$

$\triangleq [ \text{kind} : \{ \text{“doCommit”}, \text{“doAbort”} \} ]$

commit($n$) $\triangleq [ \text{kind} \mapsto \text{“commit”}, \text{node} \mapsto n ]$

abort($n$) $\triangleq [ \text{kind} \mapsto \text{“abort”}, \text{node} \mapsto n ]$

$\text{doCommit} \triangleq [ \text{kind} \mapsto \text{“doCommit”} ]$

$\text{doAbort} \triangleq [ \text{kind} \mapsto \text{“doAbort”} ]$

**Init** $\triangleq \land cState = \text{“preparing”} \land nState = [ n \in \text{Node} \mapsto \text{“preparing”} ]$

$\land \text{committed} = \{} \land \text{msgs} = \{}$

**Decide($n$)** $\triangleq \land nState[n] = \text{“preparing”}$

$\lor \land nState'[n] = [ nState \text{ EXCEPT } ![n] = \text{“proposeCommit”} ]$

$\land \text{msgs}' = \text{msgs} \cup \{ \text{commit($n$)} \}$

$\lor \land nState'[n] = [ nState \text{ EXCEPT } ![n] = \text{“proposeAbort”} ]$

$\land \text{msgs}' = \text{msgs} \cup \{ \text{abort($n$)} \}$
Two-Phase Commit in TLA⁺ (2)

\[ RcvCommit(n) \triangleq n \notin \text{committed} \land \text{commit}(n) \in \text{msgs} \]
\[ \land \text{committed}' = \text{committed} \cup \{n\} \land n\text{State}' = n\text{State} \]
\[ \land \text{IF committed}' = \text{Node} \]
\[ \text{THEN cState}' = "\text{committed}" \land \text{msgs}' = \text{msgs} \cup \{\text{doCommit}\} \]
\[ \text{ELSE UNCHANGED } \langle \text{cState}, \text{msgs} \rangle \]

\[ RcvAbort(n) \triangleq \land \text{abort}(n) \in \text{msgs} \land \text{cState}' = "\text{aborted}" \]
\[ \land \text{msgs}' = \text{msgs} \cup \{\text{doAbort}\} \]
\[ \land \text{UNCHANGED } \langle \text{nState}, \text{committed} \rangle \]

\[ \text{Execute}(n) \triangleq \land \lor \land \text{doCommit} \in \text{msgs} \]
\[ \land \text{nState}' = [\text{nState} \text{EXCEPT } ![n] = "\text{committed}""] \]
\[ \lor \land \text{doAbort} \in \text{msgs} \]
\[ \land \text{nState}' = [\text{nState} \text{EXCEPT } ![n] = "\text{aborted}""] \]
\[ \land \text{UNCHANGED } \langle \text{cState}, \text{committed}, \text{msgs} \rangle \]

\[ \text{Next} \triangleq \exists n \in \text{Node} : \text{Decide}(n) \lor RcvCommit(n) \lor RcvAbort(n) \lor \text{Execute}(n) \]

\[ \text{Spec} \triangleq \text{Init} \land \Box[\text{Next}]_{\text{vars}} \]
EXERCISE: VERIFYING PROPERTIES OF THE PROTOCOL

- State the following properties as TLA+ formulas
  - type correctness: variables take expected values
  - the coordinator does not send conflicting orders
  - if a “doCommit” message has been sent then
    1. all participants are in state “readyCommit” or “committed”
    2. no “abort” message has been sent

- Use the TLC model checker
  - verify the above properties for finite instances
  - note the size of the corresponding state spaces

- Check deadlock freedom and explain the result
Specifications and properties are both TLA\(^+\) formulas

- consider theorems of the following forms
  
  \[ Spec \Rightarrow Prop \quad Impl \Rightarrow Spec \]

- every execution of \( Spec \) satisfies property \( Prop \)
- every execution of \( Impl \) corresponds to an execution of \( Spec \)
Specifications and properties are both TLA$^+$ formulas

- consider theorems of the following forms

\[ \text{Spec} \Rightarrow \text{Prop} \quad \text{Impl} \Rightarrow \text{Spec} \]

- every execution of Spec satisfies property Prop
- every execution of Impl corresponds to an execution of Spec

Two-phase commit implements distributed commitment

\[ DC \overset{\Delta}{=} \text{INSTANCE} \text{ DistributedCommit} \]

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

- enter \( DC!Spec \) as a temporal property and run TLC
- TLC verifies that the implementation is correct
How can this be true?

- TwoPhaseCommit uses more variables than DistributedCommit
- every action of DistributedCommit changes variable nState
- actions like RcvCommit of TwoPhaseCommit leave nState unchanged
How can this be true?

- *TwoPhaseCommit* uses more variables than *DistributedCommit*
- every action of *DistributedCommit* changes variable *nState*
- actions like *RcvCommit* of *TwoPhaseCommit* leave *nState* unchanged

**TLA**\(^+\) specification do not fix the state space

- formulas are interpreted over all (infinitely many) variables
- of course, only the variables of interest are constrained
- may compare specifications using different sets of variables
How can this be true?

- *TwoPhaseCommit* uses more variables than *DistributedCommit*
- every action of *DistributedCommit* changes variable *nState*
- actions like *RcvCommit* of *TwoPhaseCommit* leave *nState* unchanged

**TLA**+ specification do not fix the state space

- formulas are interpreted over all (infinitely many) variables
- of course, only the variables of interest are constrained
- may compare specifications using different sets of variables

**TLA**+ formulas are insensitive to finite stuttering

- cannot observe changes to variables other than those of interest
- \( \Box [\text{Next}]_{vars} \): all transitions satisfy *Next* or leave *vars* unchanged
- *DC!Spec* allows arbitrary steps that do not change *nState*
Distributed Commitment

The Two-Phase Commitment Protocol

Liveness Properties

More On TLA+ Expressions

Model Checking Large Specifications

Summing Up

Case Study: Distributed Computation Of A Spanning Tree
So far we have only specified what may (not) happen

\[ \text{Init} \land \Box [\text{Next}]_{\text{vars}} \]

- executions must start in a state satisfying predicate \text{Init}
- all transitions that change \text{vars} must respect action \text{Next}
Safety vs. Liveness

So far we have only specified what may (not) happen

\[ \text{Init} \land \Box [\text{Next}]_{\text{vars}} \]

- executions must start in a state satisfying predicate \( \text{Init} \)
- all transitions that change \( \text{vars} \) must respect action \( \text{Next} \)

These formulas assert safety properties

- **safety**: nothing bad ever happens
- a system that does nothing never does something bad
- the above specification allows for (even infinite) stuttering
SAFETY VS. LIVENESS

- So far we have only specified what may (not) happen
  \[ \text{Init} \land \Box\[\text{Next}]_{\text{vars}} \]
  - executions must start in a state satisfying predicate \textit{Init}
  - all transitions that change \textit{vars} must respect action \textit{Next}

- These formulas assert safety properties
  - \textit{safety}: nothing bad ever happens
    - a system that does nothing never does something bad
    - the above specification allows for (even infinite) stuttering

- A full specification should also say what must happen
  - \textit{liveness}: something good happens eventually
    - cannot tell that it’s false by looking at a finite prefix
    - example: participants will eventually commit or abort
Box and Diamond

- □ ("box") means "always"
  - □ (nState ∈ [Node → PState]) state invariant
  - □[A]vars action invariant

- ◊ ("diamond") means "eventually"
  - ∀p ∈ Node : ◊(nState[p] ∈ {"committed", "aborted"})
  - ∃p ∈ Node : ◊⟨Decide(p)⟩vars
  - ⟨A⟩e means A ∧ (e′ ≠ e)

Combinations
- P ⇸ Q ≜ □(P ⇒ ◊Q) P is eventually followed by Q
- □◊F F is true infinitely often
- ◊□F F eventually stays true (is false only finitely often)
- note: ¬□F ≡ ◊¬F, ¬◊F ≡ □¬F, similar for □[A]vars and ◊⟨A⟩vars

Combinations
Executions specified by $Init \land \Box[Next]_{vars}$ may stop

- i.e., perform only transitions satisfying UNCHANGED $vars$
- this may happen even if some action could be taken
ENABLEDNESS OF ACTIONS

- Executions specified by $Init \land \Box[Next]_{vars}$ may stop
  - i.e., perform only transitions satisfying UNCHANGED $vars$
  - this may happen even if some action could be taken

- Enabledness of an action $A$ at state $s$
  - there exists some state $t$ such that $\langle s, t \rangle$ satisfies $A$
ENABLEDNESS OF ACTIONS

- Executions specified by $Init \land \Box [\text{Next}]_{\text{vars}}$ may stop
  - i.e., perform only transitions satisfying UNCHANGED $\text{vars}$
  - this may happen even if some action could be taken

- Enabledness of an action $A$ at state $s$
  - there exists some state $t$ such that $\langle s, t \rangle$ satisfies $A$

$$RcvCommit(n) \trianglerighteq$$

\begin{align*}
&\land n \notin \text{committed} \land \text{commit}(n) \in \text{msgs} \\
&\land \text{committed}' = \text{committed} \cup \{n\} \land n\text{State}' = n\text{State} \\
&\land \text{IF committed}' = \text{Node} \text{ THEN } \land \text{cState}' = "\text{committed}" \\
&\text{ELSE UNCHANGED } \langle \text{cState, msgs} \rangle
\end{align*}

- enabled if $n \notin \text{committed}$ and $\text{commit}(n) \in \text{msgs}$
**ENABLEDNESS OF ACTIONS**

- Executions specified by $\text{Init} \land \square[\text{Next}]_{\text{vars}}$ may stop
  - i.e., perform only transitions satisfying UNCHANGED $\text{vars}$
  - this may happen even if some action could be taken

- Enabledness of an action $A$ at state $s$
  - there exists some state $t$ such that $\langle s, t \rangle$ satisfies $A$

$$RcvCommit(n) \triangleq$$

$$\begin{align*}
&\land n \notin \text{committed} \land \text{commit}(n) \in \text{msgs} \\
&\land \text{committed}' = \text{committed} \cup \{n\} \land \text{nState}' = \text{nState} \\
&\land \text{IF committed}' = \text{Node} \text{ THEN } \land \text{cState}' = \text{"committed"} \\
&\land \text{msgs}' = \text{msgs} \cup \{\text{doCommit}\} \\
&\text{ELSE UNCHANGED } \langle \text{cState}, \text{msgs} \rangle
\end{align*}$$

- enabled if $n \notin \text{committed}$ and $\text{commit}(n) \in \text{msgs}$

**ENABLED** $A \triangleq \exists \text{vars}' : A$ (quantification over all primed variables)
Fairness hypotheses

Express that an action must occur if it is sufficiently often enabled

- different interpretations of “sufficiently often”
- temporal logic is useful for making this precise
- note: finite stuttering is still allowed
FAIRNESS HYPOTHESES

Express that an action must occur if it is sufficiently often enabled
- different interpretations of “sufficiently often”
- temporal logic is useful for making this precise
- note: finite stuttering is still allowed

Weak fairness $WF_{vars}(A)$
- if $\langle A \rangle_{vars}$ is continuously enabled then it eventually occurs
- in symbols: $\square(\square ENABLED \langle A \rangle_{vars} \Rightarrow \Diamond \langle A \rangle_{vars})$
Fairness Hypotheses

- Express that an action must occur if it is sufficiently often enabled
  - different interpretations of “sufficiently often”
  - temporal logic is useful for making this precise
  - note: finite stuttering is still allowed

**Weak fairness** $WF_{vars}(A)$
- if $\langle A \rangle_{vars}$ is continuously enabled then it eventually occurs
- in symbols: $\Box (\Box \text{ENABLED } \langle A \rangle_{vars} \Rightarrow \Diamond \langle A \rangle_{vars})$

**Strong fairness** $SF_{vars}(A)$
- if $\langle A \rangle_{vars}$ is repeatedly enabled then it eventually occurs
- in symbols: $\Box (\Box \Diamond \text{ENABLED } \langle A \rangle_{vars} \Rightarrow \Diamond \langle A \rangle_{vars})$
- note: $\langle A \rangle_{vars}$ may also be disabled repeatedly
**Weak Fairness vs. Strong Fairness**

- $SF_{vars}(A)$ implies $WF_{vars}(A)$
  - the assumption for $⟨A⟩_{vars}$ occurring is weaker
  - hence strong fairness is a stronger condition

Standard form of TLA+

$Init \land \neg ∃\neg (\exists x. \neg x) \land (\forall i : WF_{vars}(A(i)) \land (\forall j : SF_{vars}(B(j)))$ actions $A(i), B(j)$ occur as disjuncts of $Next$: the system should not stop when the action may occur $SF$: the action should eventually be performed, even if a different action is possible $no$ fairness: the action is not required to occur (e.g., a request from the environment)

Choosing appropriate fairness conditions can be tricky!
**Weak Fairness vs. Strong Fairness**

- $SF_{\text{vars}}(A)$ implies $WF_{\text{vars}}(A)$
  - the assumption for $\langle A \rangle_{\text{vars}}$ occurring is weaker
  - hence strong fairness is a stronger condition

- Standard form of TLA$^+$ specifications

\[
\text{Init} \land \Box [\text{Next}]_{\text{vars}} \land (\forall i \in W : WF_{\text{vars}}(A(i)) \land (\forall j \in S : SF_{\text{vars}}(B(j)))
\]

- actions $A(i), B(j)$ occur as disjuncts of $\text{Next}$
- WF: the system should not stop when the action may occur
- SF: the action should eventually be performed, even if a different action is possible
- no fairness: the action is not required to occur (e.g., a request from the environment)

Choosing appropriate fairness conditions can be tricky!
Weak Fairness vs. Strong Fairness

- $\text{SF}_{\text{vars}}(A)$ implies $\text{WF}_{\text{vars}}(A)$
  - the assumption for $\langle A \rangle_{\text{vars}}$ occurring is weaker
  - hence strong fairness is a stronger condition

- Standard form of TLA$^+$ specifications

\[
\text{Init} \land \Box \lbrack \text{Next} \rbrack_{\text{vars}} \land (\forall i \in W : \text{WF}_{\text{vars}}(A(i)) \land (\forall j \in S : \text{SF}_{\text{vars}}(B(j))
\]

- actions $A(i), B(j)$ occur as disjuncts of $\text{Next}$
- WF: the system should not stop when the action may occur
- SF: the action should eventually be performed, even if a different action is possible
- no fairness: the action is not required to occur (e.g., a request from the environment)

Choosing appropriate fairness conditions can be tricky!
Simple fairness hypothesis

- stop only if no action can be performed
- usually the weakest reasonable fairness condition
- other choices are possible, such as

\[ \forall n \in \text{Node} : \land \text{WF}_{\text{vars}}(\text{Decide}(n)) \land \text{WF}_{\text{vars}}(\text{Execute}(n)) \land \text{WF}_{\text{vars}}(\text{RcvCommit}(n)) \land \text{WF}_{\text{vars}}(\text{RcvAbort}(n)) \]
**Liveness Checking for Two-Phase Commit**

- **Simple fairness hypothesis**
  - \(WF_{vars}(Next)\)
  - stop only if no action can be performed
  - usually the weakest reasonable fairness condition
  - other choices are possible, such as
    \[
    \forall n \in \text{Node} : \land WF_{vars}(\text{Decide}(n)) \land WF_{vars}(\text{Execute}(n)) \\
    \land WF_{vars}(\text{RcvCommit}(n)) \land WF_{vars}(\text{RcvAbort}(n))
    \]

- **Verify liveness properties**
  - each participant will eventually abort or commit
  - \(Liveness \triangleq \forall n \in \text{Node} : \Diamond (n\text{State}[n] \in \{\text{"committed"}, \text{"aborted"}\})\)
  - similarly, add fairness condition \(WF_{n\text{State}}(Next)\) to \(DC!Spec\)
  - verify that implementation still holds
Specify algorithms as state machines
- initial condition, next-state relation, possibly fairness
- use the model checker for gaining confidence
- check non-properties and analyze counter-examples

Look for high-level abstractions
- model data using sets and functions
- exploit the power of mathematics for crisp definitions
- focus on high-level design, do not try to mimic the source code

Verify correctness by refinement when you can
- high-level specification describes intended behavior
- gradually introduce implementation detail
Outline

1. Modeling Systems in TLA+
2. System Verification
3. The TLA+ Language
4. The PlusCal Algorithm Language
5. Refinement in TLA+
6. V2X Case Study in TLA+
7. Conclusion
TLA⁺: algorithms specified by logical formulas

- data model represented in set theory
- fair state machine specified in temporal logic

Conventional descriptions of algorithms by pseudo-code

- familiar presentations, using imperative-style language
- (obviously) effective for conveying algorithmic ideas
- neither executable nor mathematically precise

PlusCal: pseudo-code flavor, but precise and more expressive
Modeling Algorithms: TLA$^+$ vs. Pseudo-Code

- **TLA$^+$**: algorithms specified by logical formulas
  - data model represented in set theory
  - fair state machine specified in temporal logic

- Conventional descriptions of algorithms by pseudo-code
  - familiar presentations, using imperative-style language
  - (obviously) effective for conveying algorithmic ideas
  - neither executable nor mathematically precise

- **PlusCal**: pseudo-code flavor, but precise and more expressive
PlusCal: Elements of an Algorithm Language

- Language for modeling algorithms, not programming
- High-level abstractions, precise semantics

- Familiar control structure + non-determinism

- Concurrency: indicate grain of atomicity
PlusCal: Elements of an Algorithm Language

- Language for modeling algorithms, not programming
- High-level abstractions, precise semantics
  - use TLA\(^+\) expressions for modeling data
  - simple translation of PlusCal to TLA\(^+\) specification
- Familiar control structure + non-determinism

- Concurrency: indicate grain of atomicity
PlusCal: Elements of an Algorithm Language

- Language for modeling algorithms, not programming
- High-level abstractions, precise semantics
  - use TLA$^+$ expressions for modeling data
  - simple translation of PlusCal to TLA$^+$ specification
- Familiar control structure + non-determinism
  - flavor of imperative language: assignment, loop, conditional, ...
  - special constructs for non-deterministic choice

```
either \{ A \} \text{ or } \{ B \} \quad \text{with } x \in S \{ A \}
```

- Concurrency: indicate grain of atomicity
PlusCal: Elements of an Algorithm Language

- Language for modeling algorithms, not programming
- High-level abstractions, precise semantics
  - use TLA\(^+\) expressions for modeling data
  - simple translation of PlusCal to TLA\(^+\) specification

- Familiar control structure + non-determinism
  - flavor of imperative language: assignment, loop, conditional, …
  - special constructs for non-deterministic choice

  
  \[
  \text{either} \{ A \} \text{ or } \{ B \} \quad \text{with} \quad x \in S \{ A \}
  \]

- Concurrency: indicate grain of atomicity
  - statements may be labeled
  - statements between two labels are executed atomically

  
  \[\text{req: } \text{try}[\text{self}] := \text{TRUE;}\]
Example: Alternating-Bit Protocol in PlusCal

---

**MODULE AlternatingBit**

EXTENDS Naturals, Sequences

CONSTANT Data

noData ≡ CHOOSE x : x /∈ Data

****

algorithm AlternatingBit {

variables sndC = ⟨⟩, ackC = ⟨⟩;

process (send = “sender”)

...

process (rcv = “receiver”)

...

process (err = “error”)

...

}

****

\* BEGIN TRANSLATION

\* END TRANSLATION

---
Example: Alternating-Bit Protocol in PlusCal

---

**MODULE AlternatingBit**

EXTENDS Naturals, Sequences

CONSTANT Data

noData ∆= CHOOSE x : x \( \notin \) Data

(****

--algorithm AlternatingBit {

variables sndC = ⟨⟩, ackC = ⟨⟩;

process (send = “sender”) …

process (rcv = “receiver”) …

process (err = “error”) …

}

****)

\* BEGIN TRANSLATION

\* END TRANSLATION

---

Stephan Merz (INRIA Nancy)  TLA+ Tutorial  Twente, September 2014  61 / 100
Example: Alternating-Bit Protocol in PlusCal

```plaintext
MODULE AlternatingBit

EXTENDS Naturals, Sequences

CONSTANT Data

noData △ CHOOSE x : x /∈ Data

(* **

--algorithm AlternatingBit {
    variables sndC = ⟨⟩, ackC = ⟨⟩;
    process (send = “sender”)
        ...
    process (rcv = “receiver”)
        ...
    process (err = “error”)
        ...
}

****)

\* BEGIN TRANSLATION
\* END TRANSLATION
```

PlusCal algorithm embedded within TLA+ module

Global variable declarations

Stephan Merz (INRIA Nancy)
Example: Alternating-Bit Protocol in PlusCal

--- MODULE AlternatingBit ---

EXTENDS Naturals, Sequences
CONSTANT Data

noData \triangleq CHOICE x : x \notin Data

(* ***)
--algorithm AlternatingBit {
  variables sndC = \langle \rangle, ackC = \langle \rangle;
  process (send = "sender")
  ...
  process (rcv = "receiver")
  ...
  process (err = "error")
  ...
}

(***)

\* BEGIN TRANSLATION
\* END TRANSLATION

PlusCal algorithm embedded within TLA+ module

global variable declarations

three parallel processes — code to be filled in
Example: Alternating-Bit Protocol in PlusCal

---

**MODULE AlternatingBit**

**EXTENDS Naturals, Sequences**

**CONSTANT Data**

\[ noData \triangleq \text{CHOOSE } x : x \notin Data \]

(****

--algorithm AlternatingBit {

  **variables** sndC = ⟨⟩, ackC = ⟨⟩;

  **process** (send = “sender”)  
  ...

  **process** (rcv = “receiver”)  
  ...

  **process** (err = “error”)  
  ...

}

****)

\* BEGIN TRANSLATION

\* END TRANSLATION

---
process (send = "sender")

variables sending = noData, sBit = 0, lastAck = 0;

s0: while (TRUE) {
    with (d ∈ Data) {
        sending := d; sBit := 1 - sBit;
    }
}

s1: while (lastAck ≠ sBit) {
    either {
        sndC := Append(sndC, ⟨sending, sBit⟩);
    }
    or {
        await (Len(ackC) > 0);
        lastAck := Head(ackC); ackC := Tail(ackC);
    }
}

/* end process send */
process (send = “sender”)

variables sending = noData, sBit = 0, lastAck = 0;

{s0: while (TRUE) {
    with (d ∈ Data) { sending := d; sBit := 1 − sBit }
}

s1: while (lastAck ≠ sBit) {
    either {
        sndC := Append(sndC, ⟨sending, sBit⟩);
    }
    or {
        await (Len(ackC) > 0);
        lastAck := Head(ackC); ackC := Tail(ackC);
    }
}

\* end process send
process (send = "sender")
  variables sending = noData, sBit = 0, lastAck = 0; 
  
  s0: while (TRUE) {
    with (d ∈ Data) { sending := d; sBit := 1 − sBit };
  }

  s1: while (lastAck ≠ sBit) {
    either {
      sndC := Append(sndC, ⟨sending, sBit⟩);
    } or {
      await (Len(ackC) > 0);
      lastAck := Head(ackC); ackC := Tail(ackC);
    }
  }

/* end process send */
PlusCal Code of Sender Process

process (send = “sender”)
    variables sending = noData, sBit = 0, lastAck = 0; {
    s0:   while (TRUE) {
        with (d ∈ Data) { sending := d; sBit := 1 − sBit };
    }
    s1:   while (lastAck ≠ sBit) {
        either {
            sndC := Append(sndC, ⟨sending, sBit⟩);
        } or {
            await (Len(ackC) > 0);
            lastAck := Head(ackC); ackC := Tail(ackC);
        }
    }
} /* end process send */
PlusCal Code of Sender Process

```plaintext
process (send = "sender")
variables sending = noData, sBit = 0, lastAck = 0; {
  s0:  while (TRUE) {
      with (d ∈ Data) { sending := d; sBit := 1 − sBit }
  }
  s1:  while (lastAck ≠ sBit) {
      either {
        sndC := Append(sndC, ⟨sending, sBit⟩);
      } or {
        await (Len(ackC) > 0);
        lastAck := Head(ackC); ackC := Tail(ackC);
      }
  }

  /* end process send
```

- Familiar “look and feel” of imperative code
process (rcv = "receiver")
    variables rcvd = noData, rBit = 0;

r0: while (TRUE) {
    r1: await (Len(sndC) > 0);
        with (d = Head(sndC)[1], b = Head(sndC)[2]) {
            sndC := Tail(sndC); ackC := Append(ackC, b);
            if (b ≠ rBit) { rcvd := d; rBit := b; }
        }
} /* end process rcv */
process (rcv = “receiver”)
  variables rcvd = noData, rBit = 0; {
  r0:   while (TRUE) {
  r1:    await (Len(sndC) > 0);
    with (d = Head(sndC)[1], b = Head(sndC)[2]) {
      sndC := Tail(sndC); ackC := Append(ackC, b);
      if (b ≠ rBit) { rcvd := d; rBit := b; }
    }
  }  /* end process rcv */
PlusCal Code of Other Processes

process (rcv = “receiver”)
    variables rcvd = noData, rBit = 0; {
    r0: while (TRUE) {
        r1: await (Len(sndC) > 0);
            with (d = Head(sndC)[1], b = Head(sndC)[2]) {
                sndC := Tail(sndC); ackC := Append(ackC, b);
                if (b ≠ rBit) { rcvd := d; rBit := b; }
            }
    } }

    \* end process rcv

process (err = “error”) {
    e0: while (TRUE) {
        either {
            await (Len(sndC) > 0); sndC := Tail(sndC);
        } or {
            await (Len(ackC) > 0); ackC := Tail(ackC);
        }
    }

    \* end process err
Translation to TLA$^+$: System State

- **TLA$^+$ variables**
  - variables corresponding to those declared in PlusCal algorithm
  - “program counter” stores current point of program execution

\[
\begin{align*}
\text{VARIABLES } & \text{sndC, ackC, pc, sending, } s\text{Bit, } \text{lastAck, } rcvd, r\text{Bit} \\
\text{ProcSet } & \triangleq \{ \text{"sender"} \} \cup \{ \text{"receiver"} \} \cup \{ \text{"error"} \} \\
\text{Init } & \triangleq \\
& \quad \land \text{sndC} = \langle \rangle \land \text{ackC} = \langle \rangle \\
& \quad \land \text{sending} = \text{noData} \land s\text{Bit} = 0 \land \text{lastAck} = 0 \\
& \quad \land \text{rcvd} = \text{noData} \land r\text{Bit} = 0 \\
& \quad \land \text{pc} = \left[ \text{self} \in \text{ProcSet} \mapsto \text{CASE } \text{self} = \text{"sender"} \rightarrow \text{"s0"} \right. \\
& \quad \quad \left. \quad \text{self} = \text{"receiver"} \rightarrow \text{"r0"} \right. \\
& \quad \quad \left. \text{self} = \text{"error"} \rightarrow \text{"e0"} \right]
\end{align*}
\]
Translation to TLA⁺: Transitions

\[ s1 \triangleq \]

\[ s1: \text{while } (\text{lastAck} \neq \text{sBit}) \{ \]
\[ \text{either } \{ \]
\[ \text{sndC} := \text{Append}(\text{sndC}, \langle \text{sending}, \text{sBit} \rangle); \]
\[ \} \text{ or } \{ \]
\[ \text{await } (\text{Len}(\text{ackC}) > 0); \]
\[ \text{lastAck} := \text{Head}(\text{ackC}); \text{ackC} := \text{Tail}(\text{ackC}); \]
\[ \} \} \]
Translation to TLA\(^+\): Transitions

\[ s1 \triangleq \\
\land pc[\text{“sender”}] = \text{“s1”} \\
\land \text{IF } lastAck \neq sBit \\
\quad \text{THEN } \lor \land \text{sndC'} = \text{Append}(\text{sndC}, \langle \text{sending}, sBit \rangle) \\
\quad \land \text{UNCHANGED } \langle \text{ackC}, lastAck \rangle \\
\quad \lor \land \text{Len}(\text{ackC}) > 0 \\
\quad \land lastAck' = \text{Head}(\text{ackC}) \\
\quad \land \text{ackC'} = \text{Tail}(\text{ackC}) \\
\quad \land \text{sndC'} = \text{sndC} \\
\land pc' = [pc \ EXCEPT ![\text{“sender”}] = \text{“s1”}] \\
\lor \land \text{UNCHANGED } \langle \text{sndC}, \text{ackC}, lastAck \rangle \\
\land \text{UNCHANGED } \langle \text{sending}, sBit, rcvd, rBit \rangle \]
Define the transition relation of the algorithm

- transition relation of process: disjunction of individual transitions
- overall next-state relation: disjunction of processes
- generalizes to multiple instances of same process type

\[
\begin{align*}
\text{send} & \triangleq s_0 \lor s_1 \\
\text{rcv} & \triangleq r_0 \lor r_1 \\
\text{err} & \triangleq e_0
\end{align*}
\]

\[
\text{Next} \triangleq \text{send} \lor \text{rcv} \lor \text{err}
\]
Translation to TLA⁺: Tying It All Together

- Define the transition relation of the algorithm
  - transition relation of process: disjunction of individual transitions
  - overall next-state relation: disjunction of processes
  - generalizes to multiple instances of same process type

\[
\begin{align*}
\text{send} & \triangleq s_0 \lor s_1 \\
\text{rcv} & \triangleq r_0 \lor r_1 \\
\text{err} & \triangleq e_0 \\
\text{Next} & \triangleq \text{send} \lor \text{rcv} \lor \text{err}
\end{align*}
\]

- Define the overall TLA⁺ specification

\[
\text{Spec} \triangleq \text{Init} \land \square[\text{Next}]_{vars}
\]
Translation to TLA⁺: Tying It All Together

- Define the transition relation of the algorithm
  - transition relation of process: disjunction of individual transitions
  - overall next-state relation: disjunction of processes
  - generalizes to multiple instances of same process type

\[
\begin{align*}
\text{send} & \triangleq s0 \lor s1 \\
\text{rcv} & \triangleq r0 \lor r1 \\
\text{err} & \triangleq e0 \\
\text{Next} & \triangleq \text{send} \lor \text{rcv} \lor \text{err}
\end{align*}
\]

- Define the overall TLA⁺ specification

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

- Extension: fairness conditions per process or label

\[
\begin{align*}
\text{fair process} \ (\text{send} = \text{“sender”}) & \quad \text{Spec} \triangleq \ldots \land \WF_{\text{vars}}(\text{send}) \\
\text{s1:} & + \quad \text{while} \ (\text{lastAck} \neq \text{sBit}) \ldots & \text{Spec} \triangleq \ldots \land \SF_{\text{vars}}(s1)
\end{align*}
\]
PlusCal: Summing Up

- A gateway drug for programmers  (C. Newcombe, Amazon)
  - retain familiar look and feel of pseudo-code
  - high level of abstraction due to TLA\(^+\) expression language
  - simple translation to TLA\(^+\) fixes formal semantics
  - standard TLA\(^+\) tool set provides verification capabilities