Modeling and Verifying Distributed Algorithms Using TLA+

http://d3s.mff.cuni.cz

Courtesy of Stephan Merz

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

MODULE HourClock

EXTENDS Naturals
VARIABLE hr

\[
HCini \triangleq hr \in (0..23)
\]

\[
HCnxt \triangleq hr' = \text{IF } hr = 23 \text{ THEN } 0 \text{ ELSE } hr + 1
\]

\[
HCsafe \triangleq HCini \land \square[HCnxt]_{hr}
\]

THEOREM HCsafe \rightarrow \square HCini
HOUR CLOCK AS TRANSITION SYSTEM

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

\[ Init \land \Box [Next]_v \]

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
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

```
preparing
  └── proposeCommit
        └── all nodes are ready to commit
            └── committed

  └── proposeAbort
        └── some node wants to abort
            └── aborted
```
A First TLA\textsuperscript{+} 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
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* \(\triangleq nState = \left[ n \in Node \mapsto \text{“preparing”} \right]\)

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

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

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

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

*Spec* \(\triangleq\) *Init* \(\land \Box [\text{Next}]_{nState}\)
Remarks on the TLA⁺ Specification

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
REMARKS ON THE TLA$^+$ SPECIFICATION

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 $Init \land \Box [Next]_v$
- 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
**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. \( 1 + 1 \neq 4 \)

- \( \forall x \in \{\} : \text{“Dublin” = “Nancy”} \)  
  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</td>
<td>function domain</td>
</tr>
<tr>
<td></td>
<td>(any set)</td>
</tr>
<tr>
<td>array</td>
<td>function application</td>
</tr>
<tr>
<td>selection</td>
<td>function</td>
</tr>
<tr>
<td>$a[i]$</td>
<td>$a(i)$</td>
</tr>
</tbody>
</table>

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 $T$
- $\text{DOMAIN } f$ domain of function $f$
- $\{x \in S \mapsto e\}$ function mapping every $x \in S$ to $e$
- $f_{\text{EXCEPT}!}\{x\} = e\{y \in \text{DOMAIN } f \mapsto \text{IF } y = x \text{ THEN } e \text{ ELSE } f(x)\}$
- $(a : > x)$ finite function mapping $a$ to $x$
- $(b : > y)$ finite function mapping $b$ to $y$ (module TLC)

Refer to previous value: $f_{\text{EXCEPT}!}\{x\} = @+1$.  
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. \(\text{Decide}(p)\)
FUNCTIONAL VALUES

Functions in TLA+

programming          mathematics
array                function
index set 0..N       function domain (any set)
array selection a[i]  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 → T]                  set of functions with domain S and values in T
DOMAIN f                  domain of function f
[x ∈ S ↦ e]               function mapping every x ∈ S to e
[f EXCEPT ![x] = e]       [y ∈ DOMAIN f ↦ IF y = x THEN e ELSE f[x]]
(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. \textit{Decide}(p).

Notations used with functions

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

Refer to previous value: \[ [f \text{ EXCEPT } ![x] = @ + 1] \]
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

$$A(p) \triangleq \land guard(p, \bar{v}) \quad \text{\textit{\textbackslash * pre-condition}}$$
$$\land v'_1 = \text{exp}_1(p, \bar{v}) \quad \text{\textit{\textbackslash * variable update}}$$
$$\land v'_2 \in \text{exp}_2(p, \bar{v}) \quad \text{\textit{\textbackslash * non-determinism}}$$
$$\land \text{UNCHANGED} \langle v_3, \ldots, v_n \rangle$$

- $guard$ : 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, ...
**How To Specify Function Updates**

Cannot define action $Commit(n)$ as

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

- 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 $Commit(n)$ as
  
  \[\forall q \in Node : nState[q] \in \{\text{"readyCommit"}, \text{"committed"}\}\]
  \[\land nState[n]' = \text{"committed"}\]

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

The new value of the function must be specified completely

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

$$nState' = [nState \ EXCEPT ![n] = \text{"committed"}]$$
Type correctness

\[NState \triangleq \{\text{"preparing"}, \text{"proposeCommit"}, \text{"proposeAbort"}, \text{"committed"}, \text{"aborted"}\}\]

\[TypeOK \triangleq nState \in [Node \rightarrow NState]\]

Nodes can commit only if all accept

\[Agreement \triangleq \forall p \in Node : nState[p] = \text{"committed"} \Rightarrow \forall q \in Node : nState[q] \in \{\text{"proposeCommit"}, \text{"committed"}\}\]
Verifying Properties of Distributed Commitment

- **Type correctness**

  \[ NState \overset{\Delta}{=} \{ \text{“preparing”, “proposeCommit”, “proposeAbort”, “committed”, “aborted”} \} \]
  \[ TypeOK \overset{\Delta}{=} nState \in [Node \rightarrow NState] \]

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

  \[ Agreement \overset{\Delta}{=} \forall p \in Node : nState[p] = \text{“committed”} \]
  \[ \Rightarrow \forall q \in Node : nState[q] \in \{ \text{“proposeCommit”, “committed”} \} \]

- **These properties are easily verified using the TLC model checker**

  - create finite model by instantiating parameter \( Node \)
  - for example: \( Node \leftarrow \{ 1, 2, 3, 4, 5 \} \)
  - can also use model values: \( Node \leftarrow \{ alice, bob, charlie \} \)
  - check invariants \( TypeOK, Agreement \)
Lesson: Deadlock & Liveness in DistributedCommit

- **Assume**
  
  \[
  \text{Commit}(n) == \\
  \land \land \forall q \in \text{Node}: \text{nState}[q] \in \{\text{"readyCommit", "committed"}\} \\
  \land \text{nState}[n] = \text{"readyCommit"} \land \text{nState}' = [\text{nState EXCEPT ![n] = "committed"}]
  \]

- **If Spec == Init \land []\lbrack \text{Next}\rbrack \_\text{nState}**
  - Deadlock reached
  - Liveness violated (stuttering: nState ‘ = nState)

- **If Spec == Init \land []\lbrack \text{Next}\rbrack \_\text{nState} \land \text{WF}_\text{nState}(\text{Next})**
  - Deadlock reached
  - Liveness preserved

- **Note: Deadlock means ~ [] ENABLED Next**
  - i.e. at this point Spec == Init \land (nState ‘ = nState) is the only option
  - Desirable here, since to goal (all nodes aborted or committed) is reached and infinite traces are needed by LTL definition ([], <>, ...)}
Lesson: Safety and Liveness in DistributedCommit

- **Safety** – nothing bad happens
  - Spec => [] invariant
    - i.e. invariant is to be valid in all states
    - Agreement == ∀ n ∈ Node : nState[n] = "committed" => ∀ q ∈ Node : nState[q] ∈ {"readyCommit", "committed"}

- **Liveness** – something good happens eventually
  - Spec => Liveness
    - Liveness typically a temporal formula of the form
      - <> L, []<> L, <>[] L, [](P => <> Q), (and combinations)
      - Liveness == ∀ n ∈ Node : <>(nState[n] ∈ {"committed", "aborted"})
    - By convention: [](P => <> Q) = P ~> Q ("leads to")
TLC basics

- Explicit state model checker
  - It checks a model (instance) of a specification
    - Determined by Spec, choice of constants, and other parameters
  - How it checks a model:
    - It begins by generating all states satisfying the initial predicate Init.
    - Then, for each state s it generates every possible next-state t such that the pair \( \langle s, t \rangle \) satisfies Next and the Fairness constraints, looking for a state where an invariant is violated.
    - Finally, it checks temporal properties over the state space (determined by distinct t states).
Symmetry Reduction

- Sometimes exact data values are irrelevant
  - DistributedCommit: identities of participant nodes
  - Never use operation other than (dis-)equality checking

- Instantiate these values by (sets of) model values
  - Model values: anonymous constants, different from each other
  - Instantiated Node by \{a,b,c,d,e\} rather than \{1,2,3,4,5\}
  - Optionally: declare these as symmetry sets
  - TLC identifies states that differ w.r.t permutation of symmetry sets

<table>
<thead>
<tr>
<th></th>
<th>No symmetry</th>
<th>symmetry</th>
</tr>
</thead>
<tbody>
<tr>
<td>N=3</td>
<td>71</td>
<td>23</td>
</tr>
<tr>
<td>N=5</td>
<td>1055</td>
<td>61</td>
</tr>
<tr>
<td>N=7</td>
<td>16511</td>
<td>127</td>
</tr>
</tbody>
</table>

# of states:
Outline

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
IMPLEMENTING DISTRIBUTED COMMITMENT

- The current specification cannot be directly implemented
  - nodes in a distributed system cannot access states of other nodes
  - introduce explicit communication by message passing
IMPLEMENTING DISTRIBUTED COMMITMENT

- 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>
</table>
Implementing Distributed Commitment

- 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>
</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
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 \((\text{“name”} \mapsto \text{“fred”}) @@ (\text{“age”} \mapsto 23)\)
**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: \( [\text{name} \mapsto \text{"fred"}, \text{age} \mapsto 23] \)
  - equals ("name" :> "fred") @@ ("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: \( [\text{rec EXCEPT } !.\text{age} = @ + 1] \)
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 (\(\text{“name”} : \text{> “fred”}\) @@ (\(\text{“age”} : \text{> 23}\))

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: \([\text{rec EXCEPT !.age} = @ + 1]\)

\(n\)-tuples (sequences) are also represented as functions
- \(\langle 42, \{\}, \text{“abc”} \rangle\) is a function with domain \(1..3\)
- \(\langle \rangle\) denotes the empty tuple
- use function application for projection, e.g. \(\text{seq}[2]\)
- cf. frequent idiom in action definitions \(\text{UNCHANGED } \langle x, y, z \rangle\)
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
A function definition can be written $f[x \in S] \triangleq e(x)$

- recursive definitions: $e(x)$ may contain $f$

\[
\text{fact}[x \in \text{Nat}] \triangleq \text{IF } x = 0 \text{ THEN } 1 \text{ ELSE } x \times \text{fact}[x - 1]
\]

- such functions are well-defined if termination is ensured
**Two-Phase Commit in TLA⁺ (1)**

**MODULE TwoPhaseCommit**

```plaintext
CONSTANT Node
VARIABLES cState, nState, committed, msgs
vars △ (cState, nState, committed, msgs)
Message △ [ kind : {"commit", "abort"}, node : Node ]
∪ [ kind : {"doCommit", "doAbort"} ]
commit(n) △ [ kind ↦ "commit", node ↦ n ]
abort(n) △ [ kind ↦ "abort", node ↦ n ]
doCommit △ [ kind ↦ "doCommit" ]
doAbort △ [ kind ↦ "doAbort" ]
```

46
**Two-Phase Commit in TLA⁺ (1)**

**Module TwoPhaseCommit**

**Constants**

- `Node`  

**Variables**

- `cState`, `nState`, `committed`, `msgs`

**Definition**

- `vars`  
- `Message`  
- `commit(n)`  
- `abort(n)`  
- `doCommit`  
- `doAbort`  

**Init**

- `cState` = "preparing"  
- `nState` = `[ n ∈ Node ↦ “preparing” ]`  
- `committed` = `{}`  
- `msgs` = `{}`

**Decide(n)**

- `nState'[n]` = "preparing"  
- `msgs'[n]` = `msgs` ∪ `{commit(n)}`  
- `msgs'[n]` = `msgs` ∪ `{abort(n)}`
Two-Phase Commit in TLA⁺ (2)

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

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

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

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

\[ \text{Spec} \triangleq \text{Init} \land \Box [\text{Next}]_{\text{vars}} \]
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
  \[\text{Spec} \Rightarrow \text{Prop} \quad \text{Impl} \Rightarrow \text{Spec}\]

- every execution of \text{Spec} satisfies property \text{Prop}
- every execution of \text{Impl} corresponds to an execution of \text{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 \text{Spec} satisfies property \text{Prop}
- every execution of \text{Impl} corresponds to an execution of \text{Spec}

**Two-phase commit implements distributed commitment**

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

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

- enter \text{DC}!\text{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
IMPLEMENTATION AS IMPLICATION

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
- □[(\textit{Next})\textsubscript{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
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 Init
- all transitions that change vars must respect action Next
So far we have only specified what may (not) happen

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

- 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

- **safety**: nothing bad ever happens
- a system that does nothing never does something bad
- the above specification allows for (even infinite) stuttering
So far we have only specified what may (not) happen

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

- 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

- \textbf{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

- \textbf{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]v and ◊⟨A⟩v
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 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 $Init \land \Box[Next] vars$ 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 $A$

\[
RcvCommit(n) \overset{\Delta}{=} \\
\land n \notin committed \land commit(n) \in msgs \\
\land committed' = committed \cup \{n\} \land nState' = nState \\
\land IF committed' = Node THEN \land cState' = “committed” \\
\land msgs' = msgs \cup \{doCommit\} \\
ELSE UNCHANGED \langle cState, msgs \rangle
\]

- enabled if $n \notin committed$ and $commit(n) \in msgs$
**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

- **Enablement of an action** \( A \) at state \( s \)
  - there exists some state \( t \) such that \( \langle s, t \rangle \) satisfies \( A \)

\[
\text{RcvCommit}(n) \overset{\Delta}{=} \\
\land n \notin \text{committed} \land \text{commit}(n) \in \text{msgs} \\
\land \text{committed}' = \text{committed} \cup \{n\} \land n\text{State}' = n\text{State} \\
\land \text{IF committed}' = \text{Node} \text{ THEN } \land c\text{State}' = \text{“committed”} \\
\land \text{msgs}' = \text{msgs} \cup \{\text{doCommit}\} \\
\text{ELSE UNCHANGED } \langle c\text{State}, \text{msgs}\rangle
\]

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

\[
\text{ENABLED } A \overset{\Delta}{=} \exists \text{vars}' : A \quad \text{(quantification over all primed variables)}
\]
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: $\Box (\Box \text{ENABLED } \langle A \rangle_{vars} \Rightarrow \Diamond \langle A \rangle_{vars})$
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 $\langle A \rangle_{vars}$ occurring is weaker
  - hence strong fairness is a stronger condition

Standard form of TLA+ specifications

$Init \land \forall i \in W: WF_{vars}(A(i)) \land \forall j \in S: SF_{vars}(B(j))$ 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

- $\text{SF}_{\text{vars}}(A)$ 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

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

- 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)
**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 $\text{TLA}^+$ specifications

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

- actions $A(i), B(j)$ occur as disjuncts of $\text{Next}$
- $\text{WF}$: the system should not stop when the action may occur
- $\text{SF}$: 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!
LIVENESS CHECKING FOR TWO-PHASE COMMIT

Simple fairness hypothesis

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

\[ \forall n \in \text{Node} : \land WF_{\text{vars}}(\text{Decide}(n)) \land WF_{\text{vars}}(\text{Execute}(n)) \land WF_{\text{vars}}(\text{RcvCommit}(n)) \land WF_{\text{vars}}(\text{RcvAbort}(n)) \]
LIVENESS CHECKING FOR TWO-PHASE COMMIT

Simple fairness hypothesis

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

\[
\forall n \in \text{Node} : \bigwedge WF_{vars}(\text{Decide}(n)) \land WF_{vars}(\text{Execute}(n)) \\
\land WF_{vars}(\text{RcvCommit}(n)) \land WF_{vars}(\text{RcvAbort}(n))
\]

Verify liveness properties

- each participant will eventually abort or commit

\[
\text{Liveness} \triangleq \forall n \in \text{Node} : \Diamond (n\text{State}[n] \in \{\text{"committed"}, \text{"aborted"}\})
\]

- similarly, add fairness condition \(WF_{nState}(\text{Next})\) to DC!Spec
- verify that implementation still holds
SUMMING UP

- 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
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

Stephan Merz (INRIA Nancy)
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

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

- Concurrency: indicate grain of atomicity

Stephan Merz (INRIA Nancy)  
TLA\(^+\) Tutorial  
Twente, September 2014  
60 / 100
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

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

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

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

```plaintext
MODULE AlternatingBit

EXTENDS Naturals, Sequences

CONSTANT Data

noData Δ= 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
Example: Alternating-Bit Protocol in PlusCal

--- MODULE AlternatingBit ---

EXTENDS Naturals, Sequences
CONSTANT Data

\[ noData \triangleq \text{CHOOSE } x : x \not\in \text{Data} \]

(* * * * *
--algorithm AlternatingBit {
  variables sndC = \langle \rangle, ackC = \langle \rangle;
  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 ∉ 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

Three parallel processes — code to be filled in

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

--- MODULE AlternatingBit ---

EXTENDS Naturals, Sequences

CONSTANT Data

noData \(\triangleq \) CHOOSE \(x : x \notin Data\)

(*\*)

--algorithm AlternatingBit {
  variables sndC = \langle \rangle, ackC = \langle \rangle;

  process (send = "sender")
  
  ...

  process (rcv = "receiver")
  
  ...

  process (err = "error")
  
  ...

}

(*\*)

\* BEGIN TRANSLATION
\* END TRANSLATION

---
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 */
process (send = “sender”) 
    variables sending = noData, sBit = 0, lastAck = 0; {   
        initialize local variables
        s0: while (TRUE) {                     
            prepare new data
            with (d ∈ Data) { sending := d; sBit := 1 − sBit }; 
        } } } } 
    \* end process send 

\begin{itemize}
    \item initialize local variables
    \item prepare new data
    \item while not acknowledged, either (re)send data or receive acknowledgement
\end{itemize}
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

receive data item and send acknowledgement
record new data item
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

```plaintext
VARIABLES sndC, ackC, pc, sending, sBit, lastAck, rcvd, rBit
ProcSet △ {“sender”} ∪ {“receiver”} ∪ {“error”}
Init △
  ∧ sndC = ⟨⟩ ∧ ackC = ⟨⟩
  ∧ sending = noData ∧ sBit = 0 ∧ lastAck = 0
  ∧ rcvd = noData ∧ rBit = 0
  ∧ pc = [self ∈ ProcSet ↦ CASE self = “sender” → “s0”
           □ self = “receiver” → “r0”
           □ self = “error” → “e0”]
```
Translation to TLA⁺: Transitions

\[ s1 \triangleq \]

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

\[ s1 \triangleq \]
\[ \begin{align*}
& \land pc["sender"] = "s1" \\
& \land \text{IF lastAck} \neq \text{sBit} \\
& \qquad \text{THEN} \land \forall \land sndC' = \text{Append}(sndC, \langle \text{sending}, \text{sBit} \rangle) \\
& \qquad \land \text{UNCHANGED } \langle \text{ackC}, \text{lastAck} \rangle \\
& \qquad \lor \land \text{Len(ackC)} > 0 \\
& \qquad \land \text{lastAck}' = \text{Head}(\text{ackC}) \\
& \qquad \land \text{ackC}' = \text{Tail}(\text{ackC}) \\
& \qquad \land \text{sndC}' = \text{sndC} \\
& \qquad \land pc' = [pc \ \text{EXCEPT } !["sender"] = "s1"] \\
& \text{ELSE} \land pc' = [pc \ \text{EXCEPT } !["sender"] = "s0"] \\
& \qquad \land \text{UNCHANGED } \langle \text{sndC, ackC, lastAck} \rangle \\
& \land \text{UNCHANGED } \langle \text{sending, sBit, rcvd, rBit} \rangle
\end{align*} \]
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

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

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

\[
\text{fair process (send = "sender")} \quad \text{Spec} \triangleq \ldots \land \text{WF}_{\text{vars}}(\text{send})
\]
\[
\text{s1:+ while (lastAck \neq sBit) \ldots} \quad \text{Spec} \triangleq \ldots \land \text{SF}_{\text{vars}}(s1)
\]
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