Modeling and Verifying Distributed Algorithms Using TLA+

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, ...
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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$HCini$</td>
<td>$hr \in (0..23)$</td>
</tr>
<tr>
<td>$HCnxt$</td>
<td>$hr' = \text{IF } hr = 23 \text{ THEN } 0 \text{ ELSE } hr + 1$</td>
</tr>
<tr>
<td>$HCsafe$</td>
<td>$HCini \land [HCnxt]_{hr}$</td>
</tr>
</tbody>
</table>

THEOREM $HCsafe \rightarrow [HCini]$
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: \( HCsafe \rightarrow □HCini \)

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

1. **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
```
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

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

\[ \text{Decide}(n) \equiv \]
\[ \bigvee n\text{State}[n] = \text{“preparing”} \land n\text{State}' = [n\text{State} \text{EXCEPT} ![n] = \text{“proposeCommit”}] \]
\[ \bigvee n\text{State}[n] = \text{“preparing”} \land n\text{State}' = [n\text{State} \text{EXCEPT} ![n] = \text{“proposeAbort”}] \]

\[ \text{Commit}(n) \equiv \]
\[ \land \forall q \in \text{Node} : n\text{State}[q] \in \{\text{“proposeCommit”, “committed”}\} \]
\[ \land n\text{State}' = [n\text{State} \text{EXCEPT} ![n] = \text{“committed”}] \]

\[ \text{Abort}(n) \equiv \]
\[ \land \exists q \in \text{Node} : n\text{State}[q] \in \{\text{“proposeAbort”, “aborted”}\} \]
\[ \land n\text{State}' = [n\text{State} \text{EXCEPT} ![n] = \text{“aborted”}] \]

\[ \text{Next} \equiv \exists n \in \text{Node} : \text{Decide}(n) \lor \text{Commit}(n) \lor \text{Abort}(n) \]

\[ \text{Spec} \equiv \text{Init} \land \square[\text{Next}]n\text{State} \]
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
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
**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* expresses initial condition
  - *Decide*(n), *Commit*(n), *Abort*(n) represent node transitions
  - transition relation *Next*: disjunction of individual transitions

\[ \text{Init} \land \square[\text{Next}] \]
**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?

- \( \forall n \in \text{Nat} : n > 0 \) \hspace{1cm} false: \( 0 \in \text{Nat} \)
- \( \exists k \in \text{Nat} : k + k = 7 \) \hspace{1cm} false: \( k + k \) is even, for all \( k \in \text{Nat} \)
- \( \forall n \in \text{Nat} : n + n = 4 \Rightarrow n \times n = 4 \) \hspace{1cm} true: \( n + n = 4 \Rightarrow n = 2 \)
- \( \exists n \in \text{Nat} : n + n = 4 \Rightarrow n = 3 \) \hspace{1cm} true, e.g. \( 1 + 1 \neq 4 \)
- \( \forall x \in \{\} : \text{“Dublin”} = \text{“Nancy”} \) \hspace{1cm} true: trivial quantifier range
- \( \exists x \in \{\} : x = x \) \hspace{1cm} false: no \( x \in \{\} \)
- \( \neg(\exists x \in S : P(x)) \equiv (\forall x \in S : \neg P(x)) \) \hspace{1cm} true
- \( 0 \div 0 = 1 \) \hspace{1cm} unspecified
- \( 42 \land \text{“xyz”} \) \hspace{1cm} 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+

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

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

- $[S \to T]$ 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 $e$
- $[f \text{ EXCEPT } ![x] = e]$ $[y \in \text{DOMAIN } f \mapsto \text{IF } y = x \text{ THEN } e \text{ ELSE } f[x]]$
- $(a:x) @@ (b:y)$ 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 \( T \)
- \( \text{DOMAIN } f \) domain of function \( f \)
- \([ x \in S \mapsto e ]\) function mapping every \( x \in S \) to \( e \)
- \([ f \text{ EXCEPT } ![x] = e ]\) \( y \in \text{DOMAIN } f \mapsto \text{IF } y = x \text{ THEN } e \text{ ELSE } f[x] \)
- \((a:\rightarrow x) \&\& (b:\rightarrow y)\) 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

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

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

$$\forall q \in \text{Node} : n\text{State}[q] \in \{\text{“readyCommit”}, \text{“committed”}\}$$

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

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

Cannot define action $Commit(n)$ as

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

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

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

$nState' = [nState \ \texttt{EXCEPT} !n = \text{"committed"}]$
Sum up: TLA+ key concepts

- A **state** is an assignment of values to all variables
- A **step** is a pair of states
- A **stuttering step** wrt some variable leaves the variable unchanged
- An **action** is a predicate over a pair of states typically employed in the form of an operator
- If $x$ is a variable in the old state, then $x'$ is the same variable in the new state
- A **behavior** is an infinite sequence of states (with an initial state)
- A **specification** characterizes the initial state and actions
• A **state function** is a first-order logic expression
• A **state predicate** is a Boolean state function
• A **temporal formula** is an assertion about behaviors

• A **theorem** of a specification is a temporal formula that holds over every behavior of the specification

• If $S$ is a specification and $I$ is a predicate and $S \Rightarrow \square I$ is a theorem we call $I$ an **invariant** of $S$
**Verifying Properties of Distributed Commitment**

- **Type correctness**

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

  \[ TypeOK \triangleq nState \in \text{[Node} \rightarrow NState] \]

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

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

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

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

- **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**

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

**If Spec == Init \land \[\text{Next}\]_n\text{State}**

- Deadlock reached
- Liveness violated (stuttering: nState ‘ = nState)

**If Spec == Init \land \[\text{Next}\]_n\text{State} \land WF_n\text{State} (Next)**

- Deadlock reached
- Liveness preserved

**Note: Deadlock means \(\sim \[\text{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 ([], <>, ...)
- [] means globally (G), <> means eventually (F)
Lesson: Safety and Liveness in DistributedCommit

- **Safety** — nothing bad happens
  - Spec => [\] invariant
    - i.e. invariant is to be valid in all states
      - Agreement == \(\forall n \in \text{Node} : (\text{nState}[n] = "committed" \Rightarrow \forall q \in \text{Node} : \text{nState}[q] \in \{"readyCommit", "committed"\})\)
      OR
      - Agreement == (\(\exists n \in \text{Node} : \text{nState}[n] = "committed"\) \Rightarrow (\(\forall q \in \text{Node} : \text{nState}[q] \in \{"readyCommit", "committed"\}))\)

- **Liveness** — something good happens eventually
  - Spec => Liveness
    - Liveness typically a temporal formula of the form
      - <> L, []<> L, <-> L, [][](P => <> Q), (and combinations)
      - Liveness == \(\forall n \in \text{Node} : <>(\text{nState}[n] \in \{"committed", "aborted"\})\)
    - By convention: [][](P => <> Q) = P \leadsto Q ("leads to")
• **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 \( \text{Init} \).
    • 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># of states:</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>
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
**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></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

```
        alice    bob    charlie    coordinator
```

```
- “commit”
- “abort”
- “commit”
- “doAbort”
```
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: \( \{ \text{name} \mapsto \text{"fred"}, \text{age} \mapsto 23 \} \)
- equals \( (\text{name} : \text{"fred"}) \oplus (\text{age} : > 23) \)
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: \([name \mapsto \text{“fred”}, \ age \mapsto 23]\)
- equals \((\text{“name”} : \mapsto \text{“fred”}) \oplus (\text{“age”} : \mapsto 23)\)

Notation used with records
- set of records of certain shape: \([name : \text{STRING}, \ age : 0..120]\)
- record access: \(\text{rec.name}\) abbreviates \(\text{rec[“name”]}\)
- record update: \([\text{rec} \text{EXCEPT !.age} = @ + 1]\)
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: $[name \mapsto \text{“fred”},\ age \mapsto 23]$ 
- equals $(\text{“name”} : \mapsto \text{“fred”}) @ (\text{“age”} : \mapsto 23)$

Notation used with records
- set of records of certain shape: $[name : \text{STRING},\ age : 0..120]$ 
- 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 $f[x]$?

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

- functions have a fixed domain, operators do not
- operators are not values: cannot be stored in variables
A function definition can be written \( f[x \in S] \triangleq e(x) \)

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

\[
fact[x \in Nat] \triangleq \begin{cases} 
1 & \text{if } x = 0 \\
x \times fact[x - 1] & \text{otherwise}
\end{cases}
\]

- such functions are well-defined if termination is ensured

Exercise:
- define an operator for inserting an integer into a sorted sequence of integers
  - using a recursive function
  - using only elementary TLA constructs

In practice, specifications contain few recursive functions
**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" ]
```

---
TWO-PHASE COMMIT IN TLA⁺ (1)

MODULE TwoPhaseCommit

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” ]

Init ≜ ∧ cState = “preparing” ∧ nState = [ n ∈ Node ↦ “preparing” ]
∧ committed = {} ∧ msgs = {}

Decide(n) ≜ ∧ nState[n] = “preparing”
∧ ∨ ∧ nState′ = [nState EXCEPT ![n] = “proposeCommit”]
∧ msgs′ = msgs ∪ {commit(n)}
∧ nState′ = [nState EXCEPT ![n] = “proposeAbort”]
∧ msgs′ = msgs ∪ {abort(n)}
Two-Phase Commit in TLA$^+$ (2)

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

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

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

\[ Next \overset{\Delta}{=} \exists n \in Node: Decide(n) \lor RcvCommit(n) \lor RcvAbort(n) \lor Execute(n) \]

\[ Spec \overset{\Delta}{=} Init \land \Box[Next]_{vars} \]
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 \( Spec \)
**Verifying Implementation**

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

\[
\text{DC} \triangleq \text{INSTANCE DistributedCommit}
\]

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

- enter \(\text{DC!Spec}\) 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

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

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

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

- 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

A full specification should also say what must happen

- 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”
  - □ (\(nState \in [\text{Node} \rightarrow \text{PState}]\)) state invariant
  - □\([A]_{\text{vars}}\) action invariant

- ◊ (“diamond”) means “eventually”
  - ∀\(p \in \text{Node}\) : ◊ (\(nState[p] \in \{\text{committed}, \text{aborted}\}\))
  - ∃\(p \in \text{Node}\) : ◊ ⟨\(\text{Decide}(p)\)⟩\(\text{vars}\)
  - ⟨\(A⟩_e\) means \(A \land (e' \neq e)\)

**Combinations**

- \(P \leadsto Q \overset{\triangleleft}{=} \Box (P \Rightarrow \Diamond Q)\) \(P\) is eventually followed by \(Q\)
- □◊\(F\) \(F\) is true infinitely often
- ◊□\(F\) \(F\) eventually stays true (is false only finitely often)
- note: \(\neg \Box F \equiv \Diamond \neg F, \neg \Diamond F \equiv \Box \neg F\), similar for □\([A]_{\text{vars}}\) and ◊⟨\(A⟩\)\(\text{vars}\)
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 $Init \land \square[Next] vars$ 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 $\text{Init} \land \Box [\text{Next}]_{\text{vars}}$ 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 $A$

$$RcvCommit(n) \triangleq$$

$$\land n \notin \text{committed} \land \text{commit}(n) \in \text{msgs}$$

$$\land \text{committed}' = \text{committed} \cup \{n\} \land n\text{State}' = n\text{State}$$

$$\land \text{IF committed}' = \text{Node} \text{ THEN } \land \text{cState}' = \text{"committed"}$$

$$\land \text{msg}' = \text{msgs} \cup \{\text{doCommit}\}$$

ELSE UNCHANGED $\langle \text{cState}, \text{msgs} \rangle$

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

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

- enabled if $n \notin committed$ and $commit(n) \in msgs$
- $\text{ENABLED } A \triangleq \exists vars' : A$ (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
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 $WF_{\text{vars}} (A)$
- if $<A>_{\text{vars}}$ is continuously enabled then it eventually occurs
- in symbols: $◊ □ \text{ENABLED} (<A>_{\text{vars}}) \Rightarrow □ ◊ <A>_{\text{vars}}$

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

\[ SF_{\text{vars}}(A) \text{ implies } WF_{\text{vars}}(A) \]

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

---

**Standard Form of TLA+ Specifications**

\[ \text{Init} \land \exists [\text{Next}] \land (\forall i \in W: WF_{\text{vars}}(A(i)) \land (\forall j \in S: SF_{\text{vars}}(B(j))) \land (A(i), B(j) \text{ occur as disjuncts of } \text{Next}) \]

\( WF: \) 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

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

- actions $A(i), B(j)$ occur as disjuncts of 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

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

- 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

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

Verify liveness properties

- each participant will eventually abort or commit

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

- similarly, add fairness condition \( WF_{nState}(Next) \) 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: $\text{TLA}^+$ vs. Pseudo-Code

- $\text{TLA}^+$: algorithms specified by logical formulas
  - data model represented in set theory
  - fair state machine specified in temporal logic
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
  - either
    - \{A\}
    - \{B\}
    - with \(x \in S\) \\
- 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

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

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

- 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 Δ 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 ≜ 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

---

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
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

---

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

---

**MODULE AlternatingBit**

EXTENDS Naturals, Sequences

CONSTANT Data

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

(*.*)

--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; { initialize local variables
    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; {

\[s0: \text{while } (\text{TRUE}) \{\]

\[\text{with } (d \in Data) \{ \text{sending := } d; \text{sBit := } 1 - \text{sBit} \};\]

\[s1: \text{while } (\text{lastAck } \neq \text{sBit}) \{\]

\[\text{either } \{\]

\[\text{sndC := Append(sndC, } \langle \text{sending, sBit} \rangle \rangle;\]

\[\text{or } \{\]

\[\text{await } (\text{Len(ackC) } > \text{0});\]

\[\text{lastAck := Head(ackC); ackC := Tail(ackC);}\]

\[\text{\} \} \} \}

\} \] /* 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; }
        }
        initialize local variables
    }
    prepare new data
    s1: while (lastAck ≠ sBit) {
        either {
            sndC := Append(sndC, ⟨sending, sBit⟩);
        } or {
            await (Len(ackC) > 0);
            lastAck := Head(ackC); ackC := Tail(ackC);
        }
    }
    while not acknowledged, either (re)send data or receive acknowledgement

    */ 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 \( rcv = \text{"receiver"} \)

variables \( rcvd = \text{noData}, rBit = 0; \) { 

r0: while (TRUE) { 

r1: await (\( \text{Len}(sndC) > 0 \)) ; 

with (d = \text{Head}(sndC)[1], b = \text{Head}(sndC)[2]) { 

sndC := \text{Tail}(sndC); ackC := \text{Append}(ackC,b); 

if (b \neq 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

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 \triangleq \{“sender”\} \cup \{“receiver”\} \cup \{“error”\}
Init \triangleq
\land sndC = \langle \rangle \land ackC = \langle \rangle
\land sending = noData \land sBit = 0 \land lastAck = 0
\land rcvd = noData \land rBit = 0
\land pc = [self \in ProcSet \mapsto \text{CASE } self = “sender” \rightarrow “s0”
  \hspace{1em} \square self = “receiver” \rightarrow “r0”
  \hspace{1em} \square self = “error” \rightarrow “e0”]
```
\[
s_1 \triangleq \\
\text{while } (\text{lastAck} \neq \text{sBit}) \{ \\
\text{either } \{ \\
\text{sndC} := \text{Append}(\text{sndC}, \langle \text{sending}, \text{sBit} \rangle); \\
\} \text{ or } \{ \\
\text{await } (\text{Len}(\text{ackC}) > 0); \\
\text{lastAck} := \text{Head}(\text{ackC}); \text{ackC} := \text{Tail}(\text{ackC}); \\
\} \}
\]
Translation to TLA\(^+\): Transitions

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

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

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

- Define the overall TLA\(^+\) specification

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

\[
\begin{align*}
\text{fair process (send = “sender”) Spec} & \triangleq \ldots \land \text{WF}_{\text{vars}}(\text{send}) \\
\text{s1:+ while (lastAck \neq sBit) … Spec} & \triangleq \ldots \land \text{SF}_{\text{vars}}(s1)
\end{align*}
\]
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