Modeling and Verifying Distributed Algorithms Using TLA$^+$

http://d3s.mff.cuni.cz

Courtesy of Stephan Merz

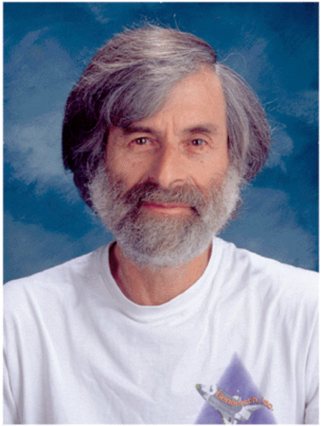
`https://members.loria.fr/Stephan.Merz/`

Department of Distributed and Dependable Systems

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

LESLIE LAMPORT

HTTP://WWW.LAMPORT.ORG/



PhD 1972 (Brandeis University), Mathematics

- Mitre Corporation, 1962–65
- Marlboro College, 1965–69
- Massachusets Computer Associates, 1970–77
- SRI International, 1977–85
- Digital Equipment Corporation / Compaq, 1985–2001
- 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 usedinESA Rosetta spacecraft
  - https://www.springer.com/gp/book/9781441997357
- Intel
  - Cache coherence protocol
  - https://dl.acm.org/doi/10.1145/1391469.1391675

TLA$^+$: INFORMAL INTRODUCTION

Example: an hour clock

MODULE *HourClock*

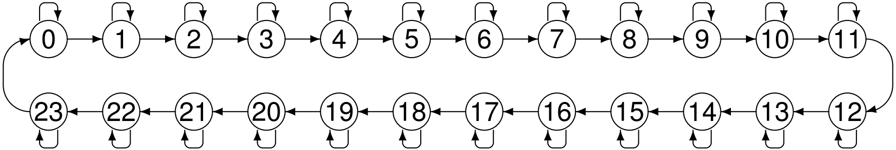
EXTENDS *Naturals*
VARIABLE *hr*

$$\begin{aligned} \mathit{HCini} &\triangleq \mathit{hr} \in (0..23) \\ \mathit{HCnxt} &\triangleq \mathit{hr}' = \text{IF } \mathit{hr} = 23 \text{ THEN } 0 \text{ ELSE } \mathit{hr} + 1 \\ \mathit{HCsafe} &\triangleq \mathit{HCini} \land \Box[\mathit{HCnxt}]_{\mathit{hr}} \end{aligned}$$

THEOREM $\mathit{HCsafe} \rightarrow \Box \mathit{HCini}$

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

HOUR CLOCK

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

HOUR CLOCK

A TLA^+ formula

$$\mathit{Init} \land \Box[\mathit{Next}]_v$$

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,

$$\mathit{HC} \;\triangleq\; \mathit{HCini} \land \Box[\mathit{HCnxt}]_{\mathit{hr}} \land \mathrm{WF}\mathit{hr}\,\mathit{HCnxt}$$

specifies an hour clock that never stops ticking.

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

Department of Distributed and Dependable Systems

PROBLEM STATEMENT

Distributed commitment.

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

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

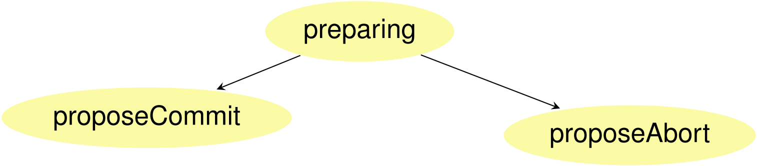
PROBLEM STATEMENT

Distributed commitment.

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

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

Control flow of each node



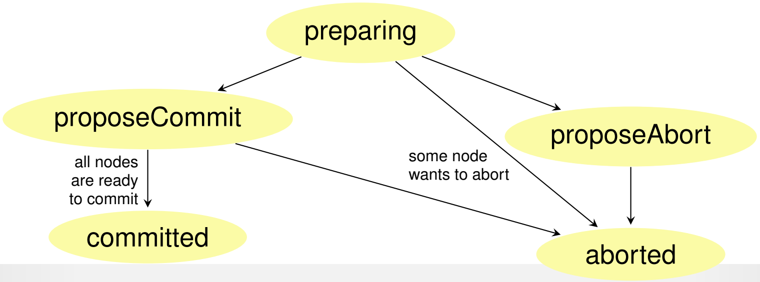
PROBLEM STATEMENT

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



A FIRST TLA$^+$ SPECIFICATION

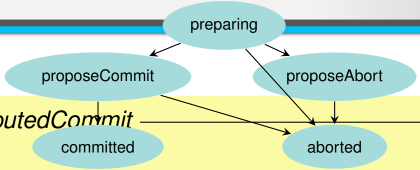
- Write a bird's eyes view specification
  - describe just how the participants' states may change
  - consider an observer that has complete information
  - don't care about distributed implementability

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*

$$\text{CONSTANT } \mathit{Node}$$

$$\text{VARIABLE } \mathit{nState}$$

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

$$\begin{aligned}
\mathit{Decide}(n) \triangleq\ & \\
& \lor\ \mathit{nState}[n] = \text{“preparing”} \land \mathit{nState}' = [\mathit{nState} \text{ EXCEPT } ![n] = \text{“proposeCommit”}] \\
& \lor\ \mathit{nState}[n] = \text{“preparing”} \land \mathit{nState}' = [\mathit{nState} \text{ EXCEPT } ![n] = \text{“proposeAbort”}]
\end{aligned}$$

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

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

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

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

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

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$^+$ $Init \land \Box[Next]_v$
  - 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

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?

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

- The last two formulas are “silly”: TLC will raise an exception
  - silly formulas are not illegal: they may occur as sub-expressions
  - $\forall n \in Nat : n \neq 0 \Rightarrow n \div n = 1$

FUNCTIONAL VALUES

- Functions in TLA$^+$

programming	mathematics
array	function
index set $0\,..\,N$	function domain (any set)
array selection $a[i]$	function application $a(i)$

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. $Decide(p)$

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

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

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

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

  - refer to previous value: $[\, f$ 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. $\text{UNCHANGED } v$) if variable v doesn't change
- Basic format of an action definition

$$\begin{array}{lll}
A(p) \triangleq & \land\ guard(p, \vec{v}) & \backslash^{*}\ \text{pre-condition} \\
 & \land\ v_1' = exp_1(p, \vec{v}) & \backslash^{*}\ \text{variable update} \\
 & \land\ v_2' \in exp_2(p, \vec{v}) & \backslash^{*}\ \text{non-determinism} \\
 & \land\ \text{UNCHANGED } \langle v_3, \ldots, v_n \rangle &
\end{array}$$

  - *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

$$\begin{aligned}&\land\ \forall\, q \in Node : nState[q] \in \{\text{“readyCommit”},\ \text{“committed”}\}\\ &\land\ nState[n]' = \text{“committed”}\end{aligned}$$

  - 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 $\mathit{Commit}(n)$ as

$$\begin{array}{l} \land\ \forall\, q \in \mathit{Node} : \mathit{nState}[q] \in \{\text{“readyCommit”},\ \text{“committed”}\} \\ \land\ \mathit{nState}[n]' = \text{“committed”} \end{array}$$

  - does not specify $\mathit{nState}[q]'$ 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 $\mathit{nState}' = [q \in \mathit{Node} \mapsto \ldots]$
  - use EXCEPT expression if only one (or a few) values are updated

$$\mathit{nState}' = [\mathit{nState}\ \ \mathrm{EXCEPT}\ ![n] = \text{“committed”}]$$

VERIFYING PROPERTIES OF DISTRIBUTED COMMITMENT

- Type correctness

$$\begin{aligned} \mathit{NState} &\triangleq \{\text{“preparing”}, \text{“proposeCommit”}, \text{“proposeAbort”}, \text{“committed”}, \text{“aborted”}\} \\ \mathit{TypeOK} &\triangleq \mathit{nState} \in [\mathit{Node} \rightarrow \mathit{NState}] \end{aligned}$$

- Nodes can commit only if all accept

$$\begin{aligned} \mathit{Agreement} \triangleq\ & \forall p \in \mathit{Node} : \mathit{nState}[p] = \text{“committed”} \\ & \Rightarrow \forall q \in \mathit{Node} : \mathit{nState}[q] \in \{\text{“proposeCommit”}, \text{“committed”}\} \end{aligned}$$

Department of Distributed and Dependable Systems

VERIFYING PROPERTIES OF DISTRIBUTED COMMITMENT

- Type correctness

$$\begin{aligned} \mathit{NState} &\triangleq \{\text{“preparing”},\ \text{“proposeCommit”},\ \text{“proposeAbort”},\ \text{“committed”},\ \text{“aborted”}\} \\ \mathit{TypeOK} &\triangleq \mathit{nState} \in [\mathit{Node} \to \mathit{NState}] \end{aligned}$$

- Nodes can commit only if all accept

$$\begin{aligned} \mathit{Agreement} \triangleq\ & \forall p \in \mathit{Node} : \mathit{nState}[p] = \text{“committed”} \\ & \quad \Rightarrow \forall q \in \mathit{Node} : \mathit{nState}[q] \in \{\text{“proposeCommit”},\ \text{“committed”}\} \end{aligned}$$

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

Lesson: Deadlock & Liveness in DistributedCommit

- Assume

```
Commit(n) ==
   /\ \A q \in Node : nState[q] \in {"readyCommit", "committed"}
  /\nState[n]="readyCommit" /\ nState' = [nState EXCEPT ![n] = "committed"]
```

- If Spec == Init /\ [][Next]_nState
  - Deadlock reached
  - Liveness violated (stuttering: nState ' = nState)
- If Spec == Init /\ [][Next]_nState /\ WF_nState(Next)
  - Deadlock reached
  - Liveness preserved
- Note: Deadlock means ~ [] ENABLED Next
  - i.e. at this point Spec == Init /\ (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$_i$
        - i.e. invariant is to be valid in all states
            - Agreement == \A n \in Node : nState[n] = "committed" => \A q \in Node : 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 == \A n \in Node : <>(nState[n] \in {"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 〈*s,t*〉 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) .

TLC basics (cont.)

- 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

# of states:

	No symmetry	symmetry
N=3	71	23
N=5	1055	61
N=7	16511	127

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

Department of
Distributed and
Dependable
Systems

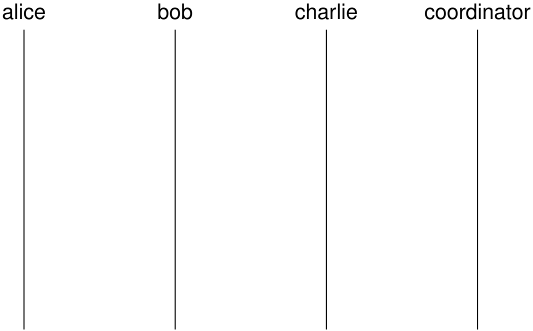
IMPLEMENTING DISTRIBUTED COMMITMENT

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

Department of Distributed and Dependable Systems

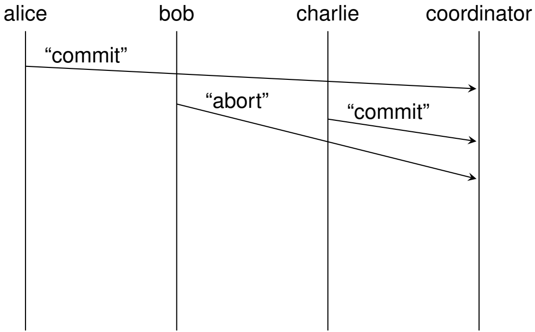
IMPLEMENTING DISTRIBUTED COMMITMENT

- 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



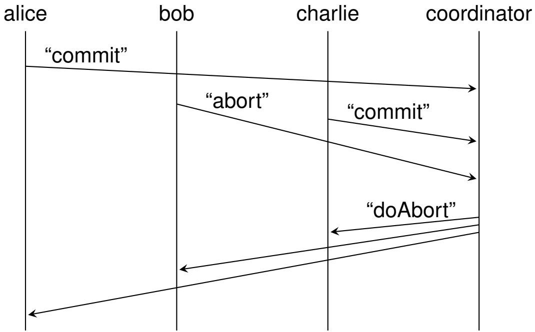
IMPLEMENTING DISTRIBUTED COMMITMENT

- 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



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



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

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: $[\mathit{name} \mapsto \text{“fred”},\ \mathit{age} \mapsto 23]$
  - equals $(\text{“name”} :> \text{“fred”})\ @@\ (\text{“age”} :> 23)$

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

TLA$^+$ RECORDS AND TUPLES

- A TLA$^+$ record corresponds to a struct in C
  - represented as a function whose domain is a set of strings
  - a record with two fields: $[name \mapsto \text{“fred”},\ age \mapsto 23]$
  - equals $(\text{“name”} :> \text{“fred”})\ @@\ (\text{“age”} :> 23)$
- Notation used with records
  - set of records of certain shape: $[name : \mathrm{STRING},\ age : 0\,..\,120]$
  - record access: $rec.name$ abbreviates $rec[\text{“name”}]$
  - record update: $[rec\ \mathrm{EXCEPT}\ !.age = @ + 1]$
- *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. $seq[2]$
  - cf. frequent idiom in action definitions $\mathrm{UNCHANGED}\ \langle x, y, z\rangle$

Functions Versus Operators

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

$$F(x) \triangleq e(x) \qquad \text{vs.} \qquad f[x \in S] \triangleq e(x)$$

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

(Recursive) Function Definitions

- A function definition can be written $f[x \in S] \triangleq e(x)$
  - recursive definitions: $e(x)$ may contain f

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

  - such functions are well-defined if termination is ensured

TWO-PHASE COMMIT IN TLA$^+$ (1)

MODULE *TwoPhaseCommit*

CONSTANT *Node*

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

$$
\begin{array}{lcl}
vars & \triangleq & \langle cState, nState, committed, msgs \rangle \\
Message & \triangleq & \quad [\,kind : \{\text{“commit”}, \text{“abort”}\},\ node : Node\,] \\
 & & \cup\ [\,kind : \{\text{“doCommit”}, \text{“doAbort”}\}\,] \\
commit(n) & \triangleq & [\,kind \mapsto \text{“commit”},\ node \mapsto n\,] \\
abort(n) & \triangleq & [\,kind \mapsto \text{“abort”},\ node \mapsto n\,] \\
doCommit & \triangleq & [\,kind \mapsto \text{“doCommit”}\,] \\
doAbort & \triangleq & [\,kind \mapsto \text{“doAbort”}\,]
\end{array}
$$

Two-Phase Commit in TLA$^+$ (1)

MODULE *TwoPhaseCommit*

CONSTANT *Node*

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

$$
\begin{aligned}
&vars \triangleq \langle cState, nState, committed, msgs \rangle \\
&Message \triangleq \quad [\, kind : \{\text{“commit”}, \text{“abort”}\},\ node : Node \,] \\
&\qquad\qquad\quad \cup [\, kind : \{\text{“doCommit”}, \text{“doAbort”}\} \,] \\
&commit(n) \triangleq [\, kind \mapsto \text{“commit”},\ node \mapsto n \,] \\
&abort(n) \triangleq [\, kind \mapsto \text{“abort”},\ node \mapsto n \,] \\
&doCommit \triangleq [\, kind \mapsto \text{“doCommit”} \,] \\
&doAbort \triangleq [\, kind \mapsto \text{“doAbort”} \,]
\end{aligned}
$$

$$
\begin{aligned}
&Init \triangleq \land\ cState = \text{“preparing”} \land nState = [\, n \in Node \mapsto \text{“preparing”} \,] \\
&\qquad\quad \land\ committed = \{\} \land msgs = \{\} \\
&Decide(n) \triangleq \land\ nState[n] = \text{“preparing”} \\
&\qquad\qquad\quad \land \lor \land\ nState' = [nState \text{ EXCEPT } ![n] = \text{“proposeCommit”}] \\
&\qquad\qquad\qquad\quad\ \ \land\ msgs' = msgs \cup \{commit(n)\} \\
&\qquad\qquad\qquad \lor \land\ nState' = [nState \text{ EXCEPT } ![n] = \text{“proposeAbort”}] \\
&\qquad\qquad\qquad\quad\ \ \land\ msgs' = msgs \cup \{abort(n)\}
\end{aligned}
$$

TWO-PHASE COMMIT IN TLA$^+$ (2)

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

$$
\begin{aligned}
\mathit{RcvAbort}(n) \triangleq\ & \land\ \mathit{abort}(n) \in \mathit{msgs} \land \mathit{cState}' = \text{"aborted"} \\
& \land\ \mathit{msgs}' = \mathit{msgs} \cup \{\mathit{doAbort}\} \\
& \land\ \text{UNCHANGED } \langle \mathit{nState}, \mathit{committed} \rangle
\end{aligned}
$$

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

$$\mathit{Next} \triangleq \exists n \in \mathit{Node} : \mathit{Decide}(n) \lor \mathit{RcvCommit}(n) \lor \mathit{RcvAbort}(n) \lor \mathit{Execute}(n)$$

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

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

VERIFYING IMPLEMENTATION

- Specifications and properties are both TLA$^+$ formulas
  - consider theorems of the following forms

$$\mathit{Spec} \Rightarrow \mathit{Prop} \qquad \mathit{Impl} \Rightarrow \mathit{Spec}$$

  - 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

$$Spec \Rightarrow Prop \qquad Impl \Rightarrow Spec$$

  - every execution of *Spec* satisfies property *Prop*
  - every execution of *Impl* corresponds to an execution of *Spec*
- Two-phase commit implements distributed commitment

$$DC \triangleq \text{INSTANCE } DistributedCommit$$
$$\text{THEOREM } Spec \Rightarrow DC!Spec$$

  - enter *DC*!*Spec* as a temporal property and run TLC
  - TLC verifies that the implementation is correct

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

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

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
  - $\Box[Next]_{vars}$: all transitions satisfy *Next* or leave *vars* unchanged
  - *DC*!*Spec* allows arbitrary steps that do not change *nState*

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

SAFETY VS. LIVENESS

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

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

  - executions must start in a state satisfying predicate *Init*
  - all transitions that change *vars* must respect action *Next*

SAFETY VS. LIVENESS

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

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

  - executions must start in a state satisfying predicate *Init*
  - all transitions that change *vars* must respect action *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

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

  - executions must start in a state satisfying predicate *Init*
  - all transitions that change *vars* must respect action *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

- $\Box$ ("box") means "always"
  - $\Box(nState \in [Node \to State])$ state invariant
  - $\Box[A]_{vars}$ action invariant
- $\Diamond$ ("diamond") means "eventually"
  - $\forall p \in Node : \Diamond(nState[p] \in \{\text{"committed"}, \text{"aborted"}\})$
  - $\exists p \in Node : \Diamond\langle Decide(p)\rangle_{vars}$
  - $\langle A\rangle_e$ means $A \wedge (e' \neq e)$
- Combinations
  - $P \leadsto Q \triangleq \Box(P \Rightarrow \Diamond Q)$ P is eventually followed by Q
  - $\Box\Diamond F$ F is true infinitely often
  - $\Diamond\Box F$ 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 $\Box[A]_v$ and $\Diamond\langle A\rangle_v$

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*

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

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

ENABLEDNESS OF ACTIONS

- Executions specified by $Init \land \Box[Next]_{vars}$ may stop
  - i.e., perform only transitions satisfying $\textsc{unchanged}\ vars$
  - 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

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

  - enabled if $n \notin committed$ and $commit(n) \in msgs$
- $\textsc{enabled}\ A \triangleq \exists\, vars' : A$ (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

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 $\mathrm{WF}_{vars}(A)$
  - if $\langle A \rangle_{vars}$ is continuously enabled then it eventually occurs
  - in symbols: $\Box(\Box\textsf{ENABLED}\ \langle A \rangle_{vars} \Rightarrow \Diamond\langle A \rangle_{vars})$

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 $\mathrm{WF}_{vars}(A)$
  - if $\langle A \rangle_{vars}$ is continuously enabled then it eventually occurs
  - in symbols: $\Box(\Box\mathsf{ENABLED}\ \langle A \rangle_{vars} \Rightarrow \Diamond\langle A \rangle_{vars})$
- Strong fairness $\mathrm{SF}_{vars}(A)$
  - if $\langle A \rangle_{vars}$ is repeatedly enabled then it eventually occurs
  - in symbols: $\Box(\Box\Diamond\mathsf{ENABLED}\ \langle A \rangle_{vars} \Rightarrow \Diamond\langle A \rangle_{vars})$
  - note: $\langle A \rangle_{vars}$ may also be disabled repeatedly

WEAK FAIRNESS VS. STRONG FAIRNESS

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

WEAK FAIRNESS VS. STRONG FAIRNESS

- $\mathrm{SF}_{vars}(A)$ implies $\mathrm{WF}_{vars}(A)$
  - the assumption for $\langle A \rangle_{vars}$ occurring is weaker
  - hence strong fairness is a stronger condition
- Standard form of TLA^{+} specifications

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

  - 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)

WEAK FAIRNESS VS. STRONG FAIRNESS

- $\mathrm{SF}_{vars}(A)$ implies $\mathrm{WF}_{vars}(A)$
  - the assumption for $\langle A \rangle_{vars}$ occurring is weaker
  - hence strong fairness is a stronger condition
- Standard form of TLA^+ specifications

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

  - 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!

LIVENESS CHECKING FOR TWO-PHASE COMMIT

- Simple fairness hypothesis $\mathrm{WF}_{vars}(Next)$
  - stop only if no action can be performed
  - usually the weakest reasonable fairness condition
  - other choices are possible, such as

$$\begin{aligned}\forall n \in Node : &\land \mathrm{WF}_{vars}(Decide(n)) \land \mathrm{WF}_{vars}(Execute(n)) \\ &\land \mathrm{WF}_{vars}(RcvCommit(n)) \land \mathrm{WF}_{vars}(RcvAbort(n))\end{aligned}$$

LIVENESS CHECKING FOR TWO-PHASE COMMIT

- Simple fairness hypothesis $\mathrm{WF}_{vars}(Next)$
  - stop only if no action can be performed
  - usually the weakest reasonable fairness condition
  - other choices are possible, such as

$$\begin{aligned}\forall n \in Node : &\land \mathrm{WF}_{vars}(Decide(n)) \land \mathrm{WF}_{vars}(Execute(n))\\ &\land \mathrm{WF}_{vars}(RcvCommit(n)) \land \mathrm{WF}_{vars}(RcvAbort(n))\end{aligned}$$

- Verify liveness properties
  - each participant will eventually abort or commit

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

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

- 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
- 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 $\{ A \}$ **or** $\{ B \}$ **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

either $\{ A \}$ **or** $\{ B \}$ **with** $x \in S \; \{ A \}$

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

req: $\mathit{try}[\mathit{self}] := \text{TRUE};$

Example: Alternating-Bit Protocol in PlusCal

```
-------------------------- MODULE AlternatingBit --------------------------
EXTENDS Naturals, Sequences
CONSTANT Data
noData ≜ CHOOSE x : x ∉ Data
(****
--algorithm AlternatingBit {
    variables sndC = ⟨⟩, ackC = ⟨⟩;
    process (send = “sender”)
        . . .
    process (rcv = “receiver”)
        . . .
    process (err = “error”)
        . . .
}
****)
\* BEGIN TRANSLATION
\* END TRANSLATION
===========================================================================
```

Example: Alternating-Bit Protocol in PlusCal

```
──────────────── MODULE AlternatingBit ────────────────
EXTENDS Naturals, Sequences
CONSTANT Data
noData ≜ CHOOSE x : x ∉ 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 ≜ 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

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

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

PlusCal translator generates TLA$^+$ specification here

PlusCal Code of Sender Process

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

PlusCal Code of Sender Process

```
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
```

initialize local variables

PlusCal Code of Sender Process

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

initialize local variables

prepare new data

PlusCal Code of Sender Process

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

initialize local variables

prepare new data

while not acknowledged, either (re)send data or receive acknowledgement

PlusCal Code of Sender Process

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

initialize local variables

prepare new data

while not acknowledged, either (re)send data or receive acknowledgement

- Familiar "look and feel" of imperative code

PlusCal Code of Other Processes

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

PlusCal Code of Other Processes

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

receive data item and send acknowledgement

record new data item

PlusCal Code of Other Processes

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

process (err = "error") {
e0:   while (TRUE) {
         either {
            await (Len(sndC) > 0); sndC := Tail(sndC);
         } or  {
            await (Len(ackC) > 0); ackC := Tail(ackC);
      } }
}    \* end process err
```

receive data item and send acknowledgement

record new data item

drop message from the data or the acknowledgement channel

Translation to TLA$^+$: System State

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

$$
\begin{array}{l}
\textsc{variables}\ \mathit{sndC},\ \mathit{ackC},\ \mathit{pc},\ \mathit{sending},\ \mathit{sBit},\ \mathit{lastAck},\ \mathit{rcvd},\ \mathit{rBit} \\
\mathit{ProcSet} \triangleq \{\text{“sender”}\} \cup \{\text{“receiver”}\} \cup \{\text{“error”}\} \\
\mathit{Init} \triangleq \\
\quad \land\ \mathit{sndC} = \langle\rangle \land \mathit{ackC} = \langle\rangle \\
\quad \land\ \mathit{sending} = \mathit{noData} \land \mathit{sBit} = 0 \land \mathit{lastAck} = 0 \\
\quad \land\ \mathit{rcvd} = \mathit{noData} \land \mathit{rBit} = 0 \\
\quad \land\ \mathit{pc} = [\mathit{self} \in \mathit{ProcSet} \mapsto \textsc{case}\ \mathit{self} = \text{“sender”} \to \text{“s0”} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \Box\ \mathit{self} = \text{“receiver”} \to \text{“r0”} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \Box\ \mathit{self} = \text{“error”} \to \text{“e0”}]
\end{array}
$$

Translation to TLA$^+$: Transitions

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

$s1 \triangleq$

Translation to TLA$^+$: Transitions

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

```
s1 ≜
  ∧ pc["sender"] = "s1"
  ∧ IF lastAck ≠ sBit
      THEN ∧ ∨ ∧ sndC' = Append(sndC, ⟨sending, sBit⟩)
               ∧ UNCHANGED ⟨ackC, lastAck⟩
             ∨ ∧ Len(ackC) > 0
               ∧ lastAck' = Head(ackC)
               ∧ ackC' = Tail(ackC)
               ∧ sndC' = sndC
           ∧ pc' = [pc EXCEPT ![“sender”] = “s1”]
      ELSE ∧ pc' = [pc EXCEPT ![“sender”] = “s0”]
           ∧ UNCHANGED ⟨sndC, ackC, lastAck⟩
  ∧ UNCHANGED ⟨sending, sBit, rcvd, rBit⟩
```

Fairly direct translation from PlusCal block to TLA$^+$ action

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

$$send \triangleq s0 \lor s1 \qquad rcv \triangleq r0 \lor r1 \qquad err \triangleq e0$$
$$Next \triangleq send \lor rcv \lor err$$

Translation to TLA$^+$: Tying It All Together

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

$$
\begin{array}{lll}
send \triangleq s0 \lor s1 & rcv \triangleq r0 \lor r1 & err \triangleq e0 \\
Next \triangleq send \lor rcv \lor err & &
\end{array}
$$

- Define the overall TLA$^+$ specification

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

Translation to TLA$^+$: Tying It All Together

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

$$\begin{array}{lll} \mathit{send} \triangleq s0 \lor s1 & \mathit{rcv} \triangleq r0 \lor r1 & \mathit{err} \triangleq e0 \\ \mathit{Next} \triangleq \mathit{send} \lor \mathit{rcv} \lor \mathit{err} & & \end{array}$$

- Define the overall TLA$^+$ specification

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

- Extension: fairness conditions per process or label

$$\begin{array}{ll} \textit{fair process } (\mathit{send} = \text{“sender”}) & \mathit{Spec} \triangleq \ldots \land \mathrm{WF}_{\mathit{vars}}(\mathit{send}) \\ \mathit{s1}\text{:+} \quad \textit{while } (\mathit{lastAck} \neq \mathit{sBit}) \ldots & \mathit{Spec} \triangleq \ldots \land \mathrm{SF}_{\mathit{vars}}(s1) \end{array}$$

PlusCal: Summing Up

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