A Tutorial Introduction to TLA$^+$

Stephan Merz

`http://www.loria.fr/~merz/`

INRIA Nancy & LORIA
Nancy, France

TLA$^+$ Community Event, ABZ 2014
Toulouse, June 3, 2014

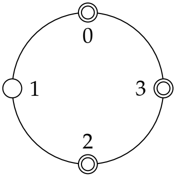
Objective

- Explain basic concepts of TLA$^+$
  - modeling systems: static and dynamic aspects
  - existing tool support for modeling and analysis
    PlusCal translator, TLC model checker, TLAPS proof platform
  - elementary aspects of system refinement
- Example-driven presentation, not trying to be exhaustive

Outline

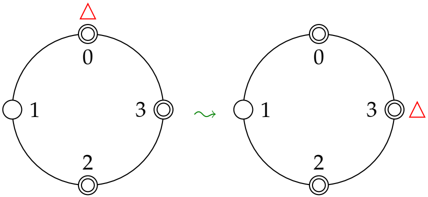
1. Modeling Systems in TLA$^+$
2. System Verification
3. The PlusCal Algorithm Language
4. Refinement in TLA$^+$

Example: Distributed Termination Detection



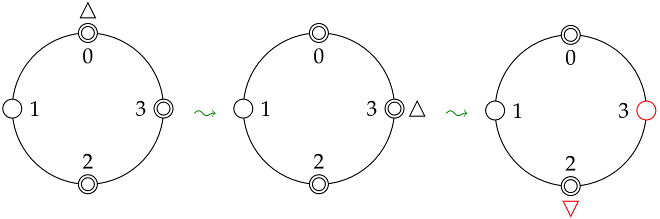
- Nodes arranged on a ring perform some computation
    - nodes can be active (double circle) or inactive
    - how can node 0 (master node) detect when all nodes are inactive?

Example: Distributed Termination Detection



- Nodes arranged on a ring perform some computation
  - nodes can be active (double circle) or inactive
  - how can node 0 (master node) detect when all nodes are inactive?
- Token-based algorithm
  - initially: token at master node, who may pass it to its neighbor

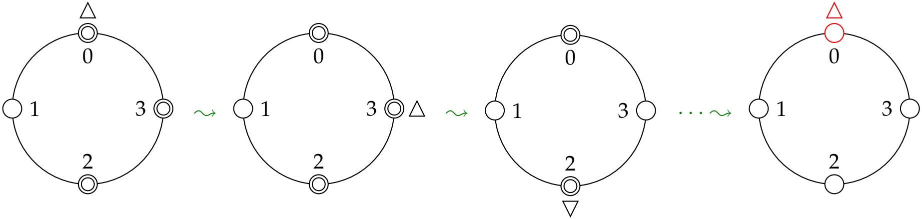
Example: Distributed Termination Detection



- Nodes arranged on a ring perform some computation
    - nodes can be active (double circle) or inactive
    - how can node 0 (master node) detect when all nodes are inactive?

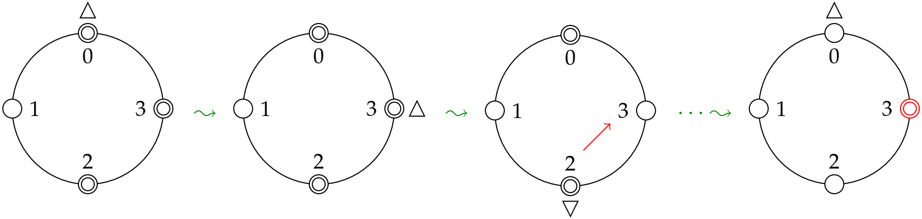
- Token-based algorithm
    - initially: token at master node, who may pass it to its neighbor
    - when a node is inactive, it passes on the token

Example: Distributed Termination Detection



- Nodes arranged on a ring perform some computation
  - nodes can be active (double circle) or inactive
  - how can node 0 (master node) detect when all nodes are inactive?
- Token-based algorithm
  - initially: token at master node, who may pass it to its neighbor
  - when a node is inactive, it passes on the token
  - termination detected when token returns to inactive master node

Example: Distributed Termination Detection

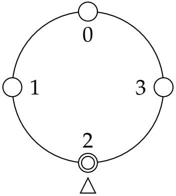


- Nodes arranged on a ring perform some computation
  - nodes can be active (double circle) or inactive
  - how can node 0 (master node) detect when all nodes are inactive?

- Token-based algorithm
  - initially: token at master node, who may pass it to its neighbor
  - when a node is inactive, it passes on the token
  - termination detected when token returns to inactive master node

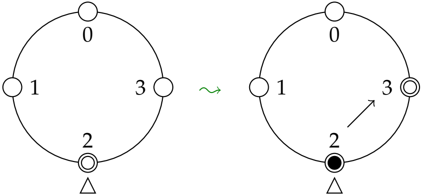
- Complication: nodes may send messages, activating receiver

Dijkstra’s Algorithm (EWD 840, 1983)



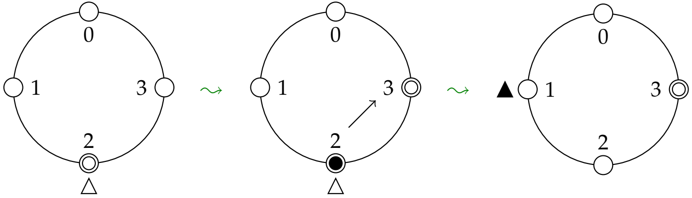
- Nodes and token colored black or white
  - master node initiates probe by sending white token

Dijkstra's Algorithm (EWD 840, 1983)



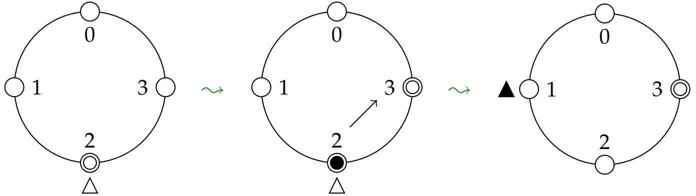
- Nodes and token colored black or white
  - master node initiates probe by sending white token
  - message to higher-numbered node stains sending node

Dijkstra's Algorithm (EWD 840, 1983)



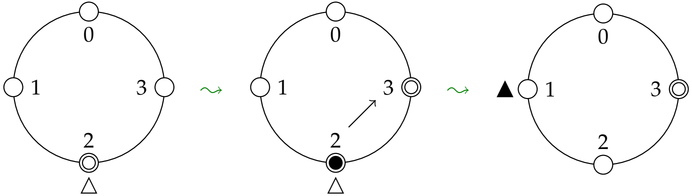
- Nodes and token colored black or white
  - master node initiates probe by sending white token
  - message to higher-numbered node stains sending node
  - when passing the token, a black node stains the token

Dijkstra's Algorithm (EWD 840, 1983)



- Nodes and token colored black or white
  - master node initiates probe by sending white token
  - message to higher-numbered node stains sending node
  - when passing the token, a black node stains the token
- Termination detection by master node
  - white token at inactive, white master node

Dijkstra's Algorithm (EWD 840, 1983)



- Nodes and token colored black or white
    - master node initiates probe by sending white token
    - message to higher-numbered node stains sending node
    - when passing the token, a black node stains the token
- Termination detection by master node
    - white token at inactive, white master node
- Required correctness properties
    - safety: termination detected only if all nodes inactive
    - liveness: when all nodes inactive, termination will be detected

TLA$^+$ Specification of EWD 840: Data Model

```
──────────────────── MODULE EWD840 ────────────────────
EXTENDS Naturals
CONSTANT N
ASSUME NAssumption ≜ N ∈ Nat \ {0}
Nodes ≜ 0 .. N − 1
Color ≜ {"white", "black"}
VARIABLES tpos, tcolor, active, color
TypeOK ≜ ∧ tpos ∈ Nodes ∧ tcolor ∈ Color
         ∧ active ∈ [Nodes → BOOLEAN] ∧ color ∈ [Nodes → Color]
───────────────────────────────────────────────────────
```

- Declaration of parameters
- Definition of operators
    - sets *Nodes* and *Color*
    - *TypeOK* documents expected values of variables
    - *active* and *color* are arrays, i.e. functions

TLA$^+$ Specification of EWD 840: Behavior (1)

$$\begin{array}{lll} \mathit{Init} & \triangleq & \land\ \mathit{tpos} \in \mathit{Nodes} \land \mathit{tcolor} = \text{“black”} \\ & & \land\ \mathit{active} \in [\mathit{Nodes} \rightarrow \textsc{boolean}] \land \mathit{color} \in [\mathit{Nodes} \rightarrow \mathit{Color}] \end{array}$$

- **Initial condition:** any “type-correct” values; token should be black

TLA$^+$ Specification of EWD 840: Behavior (1)

$$\begin{array}{l}
\mathit{Init} \triangleq \land\ \mathit{tpos} \in \mathit{Nodes} \land \mathit{tcolor} = \text{“black”} \\
\qquad\quad \land\ \mathit{active} \in [\mathit{Nodes} \rightarrow \mathrm{BOOLEAN}] \land \mathit{color} \in [\mathit{Nodes} \rightarrow \mathit{Color}] \\
\mathit{InitiateProbe} \triangleq \\
\qquad \land\ \mathit{tpos} = 0 \land (\mathit{tcolor} = \text{“black”} \lor \mathit{color}[0] = \text{“black”}) \\
\qquad \land\ \mathit{tpos}' = N - 1 \land \mathit{tcolor}' = \text{“white”} \\
\qquad \land\ \mathit{color}' = [\mathit{color}\ \mathrm{EXCEPT}\ ![0] = \text{“white”}] \\
\qquad \land\ \mathit{active}' = \mathit{active} \\
\mathit{PassToken}(i) \triangleq \\
\qquad \land\ \mathit{tpos} = i \land \neg \mathit{active}[i] \\
\qquad \land\ \mathit{tpos}' = i - 1 \\
\qquad \land\ \mathit{tcolor}' = \mathrm{IF}\ \mathit{color}[i] = \text{“black”}\ \mathrm{THEN}\ \text{“black”}\ \mathrm{ELSE}\ \mathit{tcolor} \\
\qquad \land\ \mathit{color}' = [\mathit{color}\ \mathrm{EXCEPT}\ ![i] = \text{“white”}] \\
\qquad \land\ \mathit{active}' = \mathit{active}
\end{array}$$

- Initial condition: any “type-correct” values; token should be black
- Action definitions: describe transitions of the algorithm

TLA$^+$ Specification of EWD 840: Behavior (2)

$$
\begin{array}{l}
\mathit{SendMsg}(i) \triangleq \\
\quad \land\ \mathit{active}[i] \\
\quad \land\ \exists j \in \mathit{Nodes} \setminus \{i\} : \\
\qquad \land\ \mathit{active}' = [\mathit{active}\ \mathrm{EXCEPT}\ ![j] = \mathrm{TRUE}] \\
\qquad \land\ \mathit{color}' = [\mathit{color}\ \mathrm{EXCEPT}\ ![i] = \mathrm{IF}\ j > i\ \mathrm{THEN}\ \text{“black”}\ \mathrm{ELSE}\ @] \\
\quad \land\ \mathrm{UNCHANGED}\ \langle \mathit{tpos}, \mathit{tcolor} \rangle \\
\mathit{Deactivate}(i) \triangleq \\
\quad \land\ \mathit{active}[i] \land \mathit{active}' = [\mathit{active}\ \mathrm{EXCEPT}\ ![i] = \mathrm{FALSE}] \\
\quad \land\ \mathrm{UNCHANGED}\ \langle \mathit{color}, \mathit{tpos}, \mathit{tcolor} \rangle
\end{array}
$$

- Definition of remaining actions

TLA$^+$ Specification of EWD 840: Behavior (2)

$$
\begin{array}{l}
\mathit{SendMsg}(i) \triangleq \\
\qquad \land\ \mathit{active}[i] \\
\qquad \land\ \exists j \in \mathit{Nodes} \setminus \{i\} : \\
\qquad\qquad \land\ \mathit{active}' = [\mathit{active}\ \mathrm{EXCEPT}\ ![j] = \mathrm{TRUE}] \\
\qquad\qquad \land\ \mathit{color}' = [\mathit{color}\ \mathrm{EXCEPT}\ ![i] = \mathrm{IF}\ j > i\ \mathrm{THEN}\ \text{“black”}\ \mathrm{ELSE}\ @] \\
\qquad \land\ \mathrm{UNCHANGED}\ \langle \mathit{tpos}, \mathit{tcolor} \rangle \\
\mathit{Deactivate}(i) \triangleq \\
\qquad \land\ \mathit{active}[i] \land \mathit{active}' = [\mathit{active}\ \mathrm{EXCEPT}\ ![i] = \mathrm{FALSE}] \\
\qquad \land\ \mathrm{UNCHANGED}\ \langle \mathit{color}, \mathit{tpos}, \mathit{tcolor} \rangle \\
\mathit{Next} \triangleq \\
\qquad \lor\ \mathit{InitiateProbe} \lor \exists i \in \mathit{Nodes} \setminus \{0\} : \mathit{PassToken}(i) \\
\qquad \lor\ \exists i \in \mathit{Nodes} : \mathit{SendMsg}(i) \lor \mathit{Deactivate}(i) \\
\mathit{vars} \triangleq \langle \mathit{tpos}, \mathit{tcolor}, \mathit{active}, \mathit{color} \rangle \\
\mathit{Spec} \triangleq \mathit{Init} \land \Box[\mathit{Next}]_{\mathit{vars}}
\end{array}
$$

- Definition of remaining actions
- Possible executions: initial condition, interleaving of transitions

Modeling a System in TLA$^+$

1. Describe the system configurations
    - represent the state of the system by state variables
    - mathematical abstractions: numbers, sets, functions, tuples, …

Modeling a System in TLA$^+$

1. Describe the system configurations
   - represent the state of the system by state variables
   - mathematical abstractions: numbers, sets, functions, tuples, …
2. Specify system behavior as a state machine $Init \land \Box[Next]_v$
   - initial condition: state formula identifies initial states
   - next-state relation: action formula constrains allowed transitions
   - overall spec: temporal formula defines system executions
   - $\Box[Next]_v$ every transition satisfies *Next* or leaves v unchanged

Modeling a System in TLA$^+$

1. Describe the system configurations
   - represent the state of the system by state variables
   - mathematical abstractions: numbers, sets, functions, tuples, …
2. Specify system behavior as a state machine $Init \land \Box[Next]_v$
   - initial condition: state formula — identifies initial states
   - next-state relation: action formula — constrains allowed transitions
   - overall spec: temporal formula — defines system executions
   - $\Box[Next]_v$ every transition satisfies *Next* or leaves v unchanged

- Specifications (and properties) expressed in mathematical logic
  - formally, specify a universe that contains the modeled system
  - use the power of mathematical logic to decompose the specification

Outline

1. Modeling Systems in TLA$^+$
2. System Verification
   - Safety Properties
   - Liveness Properties
3. The PlusCal Algorithm Language
4. Refinement in TLA$^+$

Outline

1. Modeling Systems in TLA$^+$
2. System Verification
   - Safety Properties
   - Liveness Properties
3. The PlusCal Algorithm Language
4. Refinement in TLA$^+$

Safety Properties in TLA$^+$

1. Prove type correctness
    - invariant of the specification:

      THEOREM $Spec \Rightarrow \Box TypeOK$

    - asserts that *TypeOK* is always true during any execution of *Spec*

Safety Properties in TLA$^+$

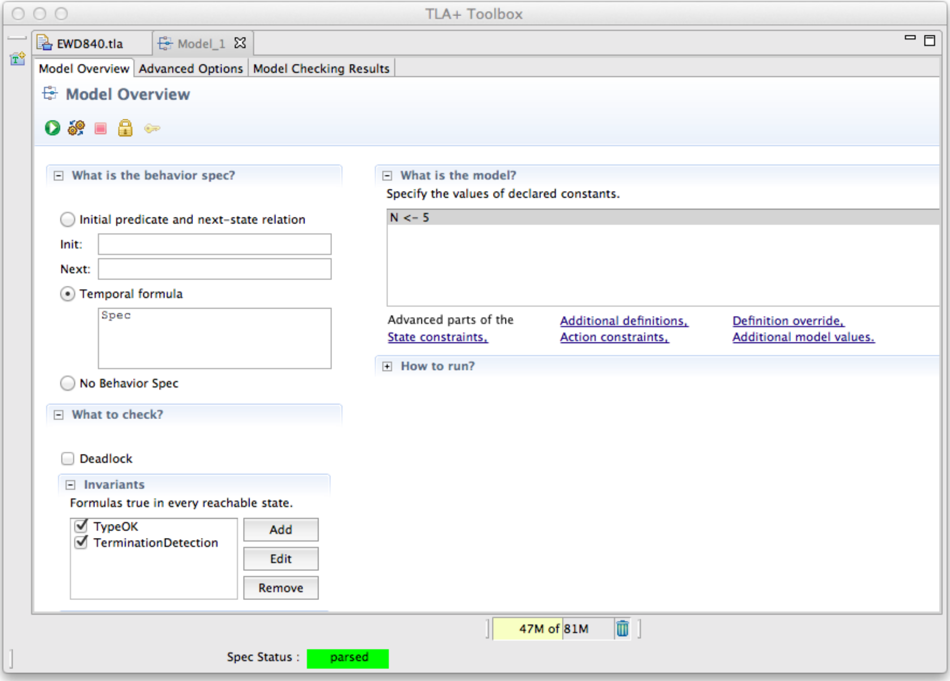
1. Prove type correctness
   - invariant of the specification: $\text{THEOREM } \mathit{Spec} \Rightarrow \Box \mathit{TypeOK}$
   - asserts that *TypeOK* is always true during any execution of *Spec*
2. Termination detection implies that all nodes are inactive
   - termination detected when white token at inactive, white node 0

$$
\begin{array}{l}
\mathit{terminationDetected} \triangleq \\
\qquad \mathit{tpos} = 0 \land \mathit{tcolor} = \text{“white”} \land \neg \mathit{active}[0] \land \mathit{color}[0] = \text{“white”} \\
\mathit{TerminationDetection} \triangleq \\
\qquad \mathit{terminationDetected} \Rightarrow \forall i \in \mathit{Nodes} : \neg \mathit{active}[i] \\
\text{THEOREM } \mathit{Spec} \Rightarrow \Box \mathit{TerminationDetection}
\end{array}
$$

   - formally again expressed as an invariant

Model Checking Using TLC

- Define a model: finite instance of TLA$^+$ specification
  - instantiate system parameters by concrete values
    for example, create instance for $N = 5$
  - indicate operator corresponding to system specification
    in our example, $Spec$
  - indicate invariants to verify
    formulas $TypeOK$ and $TerminationDetection$
  - run TLC on this model and for these properties
- Fully integrated into the TLA$^+$ Toolbox
  - Eclipse IDE for developing and analyzing TLA$^+$ specifications
- Use TLC also for validating the specification



Model Overview

What is the behavior spec?

○ Initial predicate and next-state relation

Init:

Next:

● Temporal formula

`Spec`

○ No Behavior Spec

What to check?

☐ Deadlock

Invariants

Formulas true in every reachable state.

☑ TypeOK
☑ TerminationDetection

What is the model?

Specify the values of declared constants.

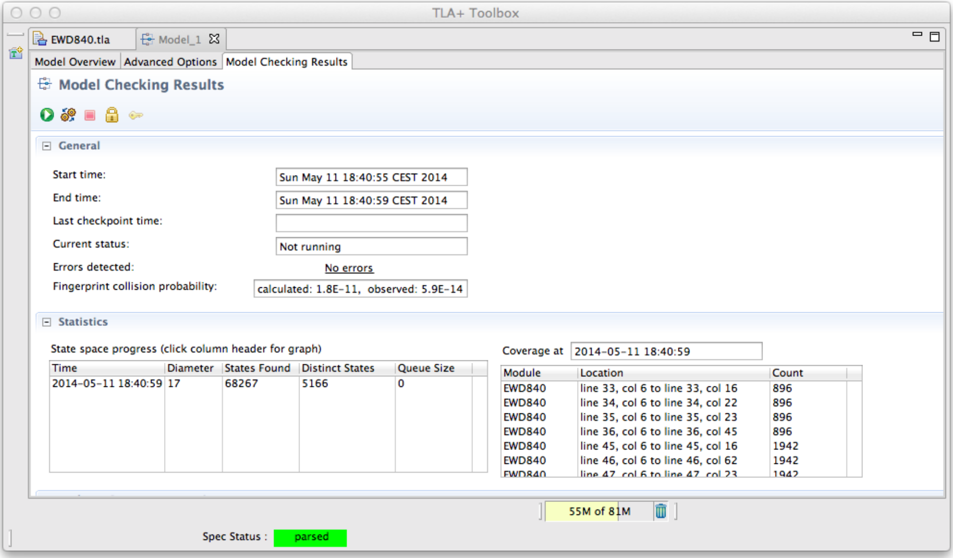
N <- 5

Advanced parts of the model: Additional definitions, Definition override, State constraints, Action constraints, Additional model values.

How to run?

47M of 81M

Spec Status : parsed



Model Checking Results

General

Start time: Sun May 11 18:40:55 CEST 2014

End time: Sun May 11 18:40:59 CEST 2014

Last checkpoint time:

Current status: Not running

Errors detected: No errors

Fingerprint collision probability: calculated: 1.8E-11, observed: 5.9E-14

Statistics

State space progress (click column header for graph)

Time	Diameter	States Found	Distinct States	Queue Size
2014-05-11 18:40:59	17	68267	5166	0

Coverage at 2014-05-11 18:40:59

Module	Location	Count
EWD840	line 33, col 6 to line 33, col 16	896
EWD840	line 34, col 6 to line 34, col 22	896
EWD840	line 35, col 6 to line 35, col 23	896
EWD840	line 36, col 6 to line 36, col 45	896
EWD840	line 45, col 6 to line 45, col 16	1942
EWD840	line 46, col 6 to line 46, col 62	1942
EWD840	line 47, col 6 to line 47, col 23	1942

55M of 81M

Spec Status : parsed

Using TLAPS to Prove Safety of EWD 840

- TLAPS: proof assistant for verifying TLA$^+$ specifications
  - interesting specifications cannot be verified fully automatically (for arbitrary instances)
  - user interaction guides verification
  - automatic back-end provers discharge leaf obligations

Using TLAPS to Prove Safety of EWD 840

- TLAPS: proof assistant for verifying TLA$^+$ specifications
  - interesting specifications cannot be verified fully automatically (for arbitrary instances)
  - user interaction guides verification
  - automatic back-end provers discharge leaf obligations

- Proving a simple invariant in TLAPS

$$\begin{array}{l}
\text{THEOREM } \mathit{TypeOK\_inv} \triangleq \mathit{Spec} \Rightarrow \Box \mathit{TypeOK} \\
\langle 1 \rangle 1.\ \mathit{Init} \Rightarrow \mathit{TypeOK} \\
\langle 1 \rangle 2.\ \mathit{TypeOK} \land [\mathit{Next}]_{\mathit{vars}} \Rightarrow \mathit{TypeOK}' \\
\langle 1 \rangle 3.\ \text{QED} \quad \text{BY } \langle 1 \rangle 1, \langle 1 \rangle 2, \mathit{PTL} \text{ DEF } \mathit{Spec}
\end{array}$$

  - hierarchical proof language represents proof tree
  - steps can be proved in any order: usually start with QED step
  - invariant follows from steps ⟨1⟩1 and ⟨1⟩2 by temporal logic

Simple Proofs

- Prove that *Init* implies *TypeOK*

$$\begin{array}{l}\langle 1 \rangle 1.\ \mathit{Init} \Rightarrow \mathit{TypeOK} \\ \quad \textsc{by}\ \mathit{NAssumption}\ \textsc{defs}\ \mathit{Init}, \mathit{TypeOK}, \mathit{Node}, \mathit{Color}\end{array}$$

  - definitions and facts must be cited explicitly
  - this helps manage the size of the search space for backend provers

Simple Proofs

- Prove that *Init* implies *TypeOK*

$$\langle 1 \rangle 1.\ \mathit{Init} \Rightarrow \mathit{TypeOK}$$
$$\quad \text{BY } \mathit{NAssumption} \text{ DEFS } \mathit{Init}, \mathit{TypeOK}, \mathit{Node}, \mathit{Color}$$

  - definitions and facts must be cited explicitly
  - this helps manage the size of the search space for backend provers

- Attempt similar proof for step $\langle 1 \rangle 2$

$$\langle 1 \rangle 2.\ \mathit{TypeOK} \land [\mathit{Next}]_{\mathit{vars}} \Rightarrow \mathit{TypeOK}'$$
$$\quad \text{BY } \mathit{NAssumption} \text{ DEFS } \mathit{TypeOK}, \mathit{Next}, \mathit{vars}, \mathit{InitiateProbe}, \ldots$$

  - back-end provers don't prove this automatically
  - use TLC to ensure that *TypeOK* is an inductive invariant

$$\mathit{TypeOK} \land \Box[\mathit{Next}]_{\mathit{vars}} \Rightarrow \Box \mathit{TypeOK}$$

  - decompose proof obligation into simpler steps

Hierarchical Proofs

```
⟨1⟩2. TypeOK ∧ [Next]_vars ⇒ TypeOK'
  ⟨2⟩ USE DEF TypeOK, Node, Color
  ⟨2⟩ SUFFICES ASSUME TypeOK, Next
               PROVE  TypeOK'
    BY DEFS TypeOK, vars
  ⟨2⟩1. CASE InitiateProbe
    BY ⟨2⟩1 DEF InitiateProbe
  ⟨2⟩2. ASSUME NEW i ∈ Node \ {0}, PassToken(i)
        PROVE  TypeOK'
    BY ⟨2⟩2, NAssumption DEF PassToken
  ... similar for remaining actions ...
  ⟨2⟩ QED  BY ⟨2⟩1, ⟨2⟩2, ... DEF Next
```

- SUFFICES steps represent backward chaining
- trivial case UNCHANGED *vars* handled during decomposition
- Toolbox IDE helps with hierarchical decomposition

Verifying Safety: Summing Up

- Model checking for finite instances
  - TLC computes reachable state graph, checking invariants on the fly
  - for termination, the set of reachable states must be finite
  - updates of state variables: $v' = e$ or $v' \in E$
    for "computable" expressions e, E (E must evaluate to finite set)
  - finite bounds for quantifiers, set comprehensions, function domains

Verifying Safety: Summing Up

- Model checking for finite instances
  - TLC computes reachable state graph, checking invariants on the fly
  - for termination, the set of reachable states must be finite
  - updates of state variables: $v' = e$ or $v' \in E$
    for "computable" expressions *e*, *E* (*E* must evaluate to finite set)
  - finite bounds for quantifiers, set comprehensions, function domains

- Theorem proving for arbitrary instances
  - explicit, hierarchical proofs
  - inductive invariant must be provided by the user
  - main proof effort at action level: supported by automatic backends
  - PTL decision procedure for simple temporal reasoning

Outline

1. Modeling Systems in TLA$^+$
2. System Verification
   - Safety Properties
   - Liveness Properties
3. The PlusCal Algorithm Language
4. Refinement in TLA$^+$

Verifying Liveness (1)

- When all nodes are inactive, termination will be detected
  - expressed in TLA$^+$ using a leadsto-formula $F \leadsto G \triangleq \Box(F \Rightarrow \Diamond G)$

$$\begin{array}{l} \mathit{Liveness} \triangleq (\forall i \in \mathit{Nodes} : \neg \mathit{active}[i]) \leadsto \mathit{terminationDetected} \\ \text{THEOREM } \mathit{Spec} \Rightarrow \mathit{Liveness} \end{array}$$

  - verification using TLC, for 5 nodes

Verifying Liveness (1)

- When all nodes are inactive, termination will be detected
  - expressed in TLA$^+$ using a leadsto-formula $F \leadsto G \triangleq \Box(F \Rightarrow \Diamond G)$

$$\begin{array}{l} \mathit{Liveness} \triangleq (\forall i \in \mathit{Nodes} : \neg \mathit{active}[i]) \leadsto \mathit{terminationDetected} \\ \text{THEOREM } \mathit{Spec} \Rightarrow \mathit{Liveness} \end{array}$$

  - verification using TLC, for 5 nodes
  - TLC produces a counter-example that ends in infinite stuttering
  - $\Box[\mathit{Next}]_{\mathit{vars}}$ allows for steps that do not change *vars*

- Use fairness constraint to ensure progress
  - fairness: action will be taken, provided it is long enough enabled

$$\mathit{Spec} \triangleq \mathit{Init} \land \Box[\mathit{Next}]_{\mathit{vars}} \land \mathrm{WF}_{\mathit{vars}}(\mathit{Next})$$

  - don't stutter indefinitely as long as some transition is possible

Verifying Liveness (2)

- Verify liveness property for redefined specification
  - TLC verifies that the property is now satisfied

Verifying Liveness (2)

- Verify liveness property for redefined specification
  - TLC verifies that the property is now satisfied
- TLC also verifies an undesired liveness property

$$\begin{array}{l} \mathit{AllNodesTerminateIfNoMessages} \triangleq \\ \qquad \Diamond\Box[\neg\exists i \in \mathit{Nodes} : \mathit{SendMsg}(i)]_{\mathit{vars}} \Rightarrow \Diamond(\forall i \in \mathit{Nodes} : \neg\mathit{active}[i]) \end{array}$$

  - our fairness requirement is too strong!
  - weaken fairness constraint: only ensure termination detection

$$\begin{array}{l} \mathit{System} \triangleq \mathit{InitiateProbe} \lor \exists i \in \mathit{Nodes} \setminus \{0\} : \mathit{PassToken}(i) \\ \mathit{Spec} \triangleq \mathit{Init} \land \Box[\mathit{Next}]_{\mathit{vars}} \land \mathrm{WF}_{\mathit{vars}}(\mathit{System}) \end{array}$$

  - TLC now verifies liveness, but no longer termination
- Too strong fairness constraints are a frequent specification error!

Verifying Liveness: Summing Up

- Specification of fair state machine $Init \wedge \Box[Next]_v \wedge F$
  - F can be a conjunction of weak or strong fairness conditions

$$\begin{array}{lcl} \text{ENABLED}\, A & \triangleq & \exists var' : A \quad (var' \text{ contains all primed variables in } A) \\ WF_v(A) & \triangleq & \Box(\Box\, \text{ENABLED}\, \langle A \rangle_v \Rightarrow \Diamond \langle A \rangle_v) \\ SF_v(A) & \triangleq & \Box(\Box\Diamond\, \text{ENABLED}\, \langle A \rangle_v \Rightarrow \Diamond \langle A \rangle_v) \end{array}$$

  - typically: *Next* is a disjunction, fairness assumed for some disjuncts
  - don't assume too strong fairness conditions: use TLC for validation

Verifying Liveness: Summing Up

- Specification of fair state machine $Init \land \Box[Next]_v \land F$
  - F can be a conjunction of weak or strong fairness conditions

$$\begin{aligned}
\text{ENABLED}\, A &\triangleq \exists var' : A \quad (var' \text{ contains all primed variables in } A)\\
WF_v(A) &\triangleq \Box(\Box\, \text{ENABLED}\, \langle A\rangle_v \Rightarrow \Diamond\langle A\rangle_v)\\
SF_v(A) &\triangleq \Box(\Box\Diamond\, \text{ENABLED}\, \langle A\rangle_v \Rightarrow \Diamond\langle A\rangle_v)
\end{aligned}$$

  - typically: *Next* is a disjunction, fairness assumed for some disjuncts
  - don't assume too strong fairness conditions: use TLC for validation

- Verifying liveness through model checking
  - temporal properties such as $\Box\Diamond P$, $\Diamond\Box P$, $P \leadsto Q$ and combinations
  - verification using TLC is more complex, but still automatic

- Verification through theorem proving
  - not yet supported by TLAPS: needs quantified temporal logic

Outline

1. Modeling Systems in TLA$^+$
2. System Verification
3. The PlusCal Algorithm Language
4. Refinement in TLA$^+$

Languages for Describing Algorithms

- TLA$^+$: algorithms specified by logical formulas
  - set-theoretical language for modeling data
  - fair state machine specified in temporal logic

Languages for Describing Algorithms

- TLA$^+$: algorithms specified by logical formulas
  - set-theoretical language for modeling data
  - 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, not programming concurrent algorithms
- High-level abstractions, precise semantics

- Familiar control structure + non-determinism

- Concurrency: indicate grain of atomicity

PlusCal: Elements of an Algorithm Language

- Language for modeling, not programming concurrent algorithms
- 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, not programming concurrent algorithms
- 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, not programming concurrent algorithms
- 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

$\textbf{either}\ \{\ A\ \}\ \textbf{or}\ \{\ B\ \} \qquad \textbf{with}\ x \in S\ \{\ A\ \}$

- Concurrency: indicate grain of atomicity
  - statements may be labeled
  - statements from one label to the next one are executed atomically

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

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 inserts TLA$^+$ between these lines

PlusCal Code of Processes

```
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 Processes

```
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 Processes

```
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 Processes

```
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 Processes

```
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

- Code of the two other processes is similar
- Familiar “look and feel” of imperative code

Translation to TLA$^+$: System State

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

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

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

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

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

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

- 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

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

- Define the overall TLA$^+$ specification

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

- Extension: fairness conditions per process or label

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

PlusCal: Summing Up

- A gateway drug for programmers (C. Newcombe, Amazon)
    - retain familiar look and feel of pseudo-code
    - abstractness and expressiveness through embedded TLA$^+$
    - precision through simple translation to TLA$^+$
    - formal verification via standard TLA$^+$ tool set

PlusCal: Summing Up

- A gateway drug for programmers (C. Newcombe, Amazon)
  - retain familiar look and feel of pseudo-code
  - abstractness and expressiveness through embedded TLA^+
  - precision through simple translation to TLA^+
  - formal verification via standard TLA^+ tool set
- Simplicity of translation induces some limitations
  - single level of processes can make modeling unnatural
  - translation dictates rules on where labels must and cannot go
  - properties must be written in TLA^+ (probably a feature)

PlusCal: Summing Up

- A gateway drug for programmers (C. Newcombe, Amazon)
  - retain familiar look and feel of pseudo-code
  - abstractness and expressiveness through embedded TLA$^+$
  - precision through simple translation to TLA$^+$
  - formal verification via standard TLA$^+$ tool set
- Simplicity of translation induces some limitations
  - single level of processes can make modeling unnatural
  - translation dictates rules on where labels must and cannot go
  - properties must be written in TLA$^+$ (probably a feature)
- Algorithm language: much better than pseudo-code

Outline

1. Modeling Systems in TLA$^+$
2. System Verification
3. The PlusCal Algorithm Language
4. **Refinement in TLA$^+$**

Refinement of System Specifications

- Refining (implementing) specification *Spec* by *Impl*
  - every behavior allowed by *Impl* is a possible execution of *Spec*
  - TLA$^+$ formalization $\boxed{Impl \Rightarrow Spec}$
  - systems and properties represented as formulas

Refinement of System Specifications

- Refining (implementing) specification *Spec* by *Impl*
  - every behavior allowed by *Impl* is a possible execution of *Spec*
  - TLA$^+$ formalization $\boxed{\mathit{Impl} \Rightarrow \mathit{Spec}}$
  - systems and properties represented as formulas
- Problem: *Impl* will include detail not present in *Spec*
  - additional variables + extra transitions
  - many steps of *Impl* will be meaningless for *Spec*

Refinement of System Specifications

- Refining (implementing) specification *Spec* by *Impl*
  - every behavior allowed by *Impl* is a possible execution of *Spec*
  - TLA$^+$ formalization $Impl \Rightarrow Spec$
  - systems and properties represented as formulas
- Problem: *Impl* will include detail not present in *Spec*
  - additional variables + extra transitions
  - many steps of *Impl* will be meaningless for *Spec*
- Stuttering invariance to the rescue!
  - extra transitions of *Impl* stutter w.r.t. the variables in *Spec*
  - stuttering steps are allowed by next-state relation $\Box[Next]_{vars}$
  - infinite stuttering ruled out by fairness conditions

Example: Specifying Data Transmission

MODULE *Transmission*

$$
\begin{array}{l}
\textsc{constant}\ Data \\
noData \triangleq \textsc{choose}\ x : x \notin Data \\
\textsc{variables}\ sending, rcvd \\
Init \triangleq sending = noData \land rcvd = noData \\
Send \triangleq \land\ rcvd = sending \\
\qquad\qquad \land\ sending' \in Data \\
\qquad\qquad \land\ rcvd' = rcvd \\
Receive \triangleq \land\ rcvd \neq sending \\
\qquad\qquad\quad \land\ rcvd' = sending \\
\qquad\qquad\quad \land\ sending' = sending \\
vars \triangleq \langle sending, rcvd \rangle \\
Spec \triangleq Init \land \Box[Send \lor Receive]_{vars}
\end{array}
$$

- Simple handshake protocol specified as a state machine
  - new data may be sent when previous one has been received
  - no explicit mechanism for transferring data

Implementation Through Alternating Bit Protocol

```
MODULE AlternatingBit
. . .
INSTANCE Transmission
THEOREM Spec ⇒ Transmission!Spec
```

- Implementation checked by TLC
  - for fixed instances of *Data* and constraints on channel size
  - exercise: extend this to fair data transmission

    hint: ensure that the channels do not lose all data or acknowledgements
- Stuttering invariance is essential here
  - most transitions of alternating bit protocol stutter on $\langle sending, rcvd \rangle$

Information Hiding

- TLA$^+$ specifications describe state machines
  - often introduce “implementation detail” for controlling transitions
  - example: program counter generated by PlusCal translator
  - internal detail should be hidden from “interface”

Information Hiding

- TLA$^+$ specifications describe state machines
  - often introduce “implementation detail” for controlling transitions
  - example: program counter generated by PlusCal translator
  - internal detail should be hidden from “interface”

- In logic, hiding corresponds to existential quantification

$$\begin{aligned} \mathit{Inner} &\triangleq \mathit{Init} \land \Box[\mathit{Next}]_{\mathit{vars}} \land F \\ \mathit{Spec} &\triangleq \exists\!\!\exists\, x : \mathit{Inner} \end{aligned}$$

  - behaves like inner specification, but with variables x hidden

Information Hiding

- TLA$^+$ specifications describe state machines
  - often introduce "implementation detail" for controlling transitions
  - example: program counter generated by PlusCal translator
  - internal detail should be hidden from "interface"
- In logic, hiding corresponds to existential quantification

$$
\begin{aligned}
Inner &\triangleq Init \land \Box[Next]_{vars} \land F \\
Spec &\triangleq \boldsymbol{\exists}\, x : Inner
\end{aligned}
$$

  - behaves like inner specification, but with variables x hidden
- Refinement under information hiding
  - prove $Impl \Rightarrow Inner[t/x]$ for showing $Impl \Rightarrow Spec$
  - refinement mapping t : computed from implementation variables

Summing Up

- TLA$^+$: Specify systems in logic, from first principles
    - describe system behavior at appropriate level of abstraction
    - mathematical logic is flexible and expressive
    - set theory plus state machine plus temporal logic
    - no formal distinction between systems and properties
    - experience shows that this approach scales to practical systems
- Support tools
    - TLA$^+$ Toolbox: editor, syntax/semantic analysis, pretty printer
    - TLC: explicit-state model checker, checkpointing, parallelization
    - TLAPS: interactive proof platform with powerful theorem provers
    - PlusCal translator for generating TLA$^+$ specification
- Community: Google group, this workshop!

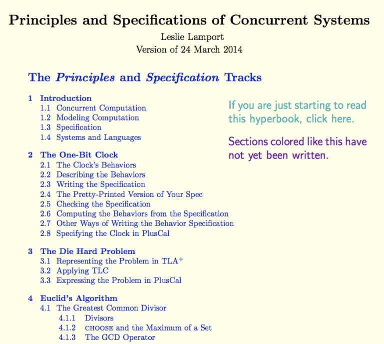
Going Further

- The TLA$^+$ Web page

  `http://research.microsoft.com/en-us/um/people/lamport/tla/tla.html`

- Detailed presentations

The Hyperbook



Principles and Specifications of Concurrent Systems

Leslie Lamport
Version of 24 March 2014

The *Principles* and *Specification* Tracks

- **1 Introduction**
  - 1.1 Concurrent Computation
  - 1.2 Modeling Computation
  - 1.3 Specification
  - 1.4 Systems and Languages
- **2 The One-Bit Clock**
  - 2.1 The Clock's Behaviors
  - 2.2 Describing the Behaviors
  - 2.3 Writing the Specification
  - 2.4 The Pretty-Printed Version of Your Spec
  - 2.5 Checking the Specification
  - 2.6 Computing the Behaviors from the Specification
  - 2.7 Other Ways of Writing the Behavior Specification
  - 2.8 Specifying the Clock in PlusCal
- **3 The Die Hard Problem**
  - 3.1 Representing the Problem in TLA$^+$
  - 3.2 Applying TLC
  - 3.3 Expressing the Problem in PlusCal
- **4 Euclid's Algorithm**
  - 4.1 The Greatest Common Divisor
    - 4.1.1 Divisors
    - 4.1.2 CHOOSE and the Maximum of a Set
    - 4.1.3 The GCD Operator

If you are just starting to read this hyperbook, click here.

Sections colored like this have not yet been written.

Specifying Systems