Specifying and Verifying Algorithms in TLA$^+$

Stephan Merz

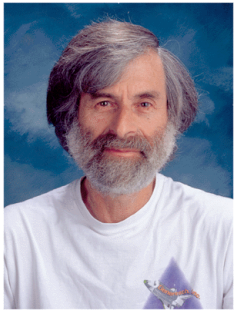
https://members.loria.fr/Stephan.Merz/

Inria Research Center at University of Lorraine & LORIA
Nancy, France

Summer School VTSA 2025
University of Liège, September 2025

Leslie Lamport

http://www.lamport.org/



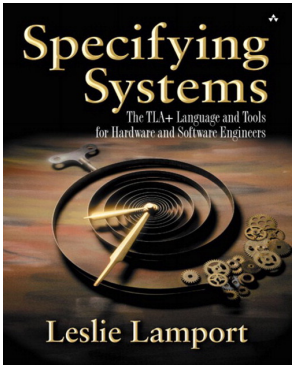
PhD 1972 (Brandeis University), Mathematics

- 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, 2001–2024

Pioneer of distributed algorithms Turing Award 2013

- Natl. Acad. of Engineering, Natl. Acad. of Sciences, American Acad. of Arts and Sciences
- PODC Influential Paper, ACM SIGOPS Hall of Fame (3x), J.C. Laprie Award (2x), LICS Award, John v. Neumann medal, E.W. Dijkstra Prize, NEC C&C Prize ...
- honorary doctorates: Rennes, Kiel, Lausanne, Lugano, Nancy, Brandeis

TLA$^+$ specification language



- describe and verify distributed and concurrent systems
- based on mathematical set theory and temporal logic TLA
- TLA$^+$ Video Course
- book: Addison-Wesley, 2003 (free download for personal use)
- TLA$^+$ home page

Some other publications

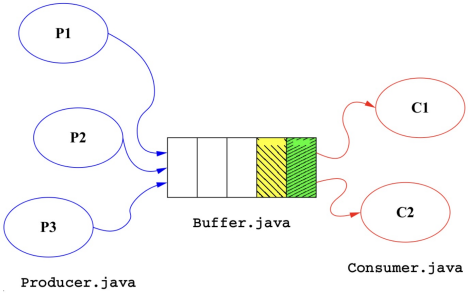
- Y. Yu, P. Manolios, L. Lamport: *Model checking TLA$^+$ Specifications*. CHARME 1999, LNCS 1703.
- D. Cousineau et al.: *TLA$^+$ Proofs*. Formal Methods (FM 2012), LNCS 7436.
- I. Konnov et al.: *TLA$^+$ Model Checking Made Symbolic.* OOPSLA 2019.
- I. Konnov, M. Kuppe, S. Merz: *Specification and Verification with the TLA$^+$ Trifecta.* LNCS 13701.

Part I

Debugging a Blocking Queue

Objectives

- Getting familiar with the TLA$^+$ language and tools
- Producer-consumer problem
  - producers blocked when queue is full
  - consumers blocked when queue is empty
- Acknowledgements
  - inspired by a blog post by Michel Charpentier
  - see also tutorial by Markus Kuppe



Java Code of the Buffer

```
public class Buffer<E> {
  private final int capacity;
  private final Queue<E> store;

  public Buffer(int capacity) {
    this.capacity = capacity;
    this.store = new ArrayDeque<E>(capacity);
  }

  public synchronized void put(E item) {
    while (store.size() == capacity) { this.wait(); }
    store.add(item);
    this.notify();
  }

  public synchronized E get() {
    while (store.size() == 0) { this.wait(); }
    E e = store.remove();
    this.notify();
    return e;
  }
}
```

naive Java implementation

low-level concurrency control

exception handling omitted

Do you see an issue?

Basic Concurrency Control in Java

- `synchronized` methods
  - disallow concurrent method executions
  - may pretend that methods execute atomically
- `Object.wait()`
  - adds the current thread to the wait set of the object
  - execution suspends until the thread is released by `notify()` or `notifyAll()`
  - useful when execution needs to await some condition to become true
- `Object.notify()` / `Object.notifyAll()`
  - allow some thread / all threads in the wait set to resume execution
  - methods invoked by a thread establishing the condition
  - resuming thread must check that condition actually holds

Modeling the Queue in TLA$^+$ / PlusCal

- PlusCal: frontend algorithmic language for TLA$^+$
  - C-like syntax, pseudo-code look and feel
  - PlusCal expressions are those of TLA$^+$
  - algorithm translated to TLA$^+$, then use TLA$^+$ verification support
- Abstractions do not match exactly
  - PlusCal has processes, which will represent producer and consumer threads
  - buffer methods represented as macros invoked by processes
  - buffer itself represented as a sequence storing the producer IDs
  - wait set holds process IDs

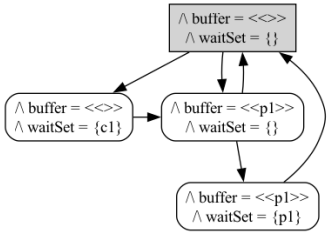
PlusCal Representation of Blocking Queue

```
–algorithm BlockingQueue {
  variables buffer = ⟨⟩, waitSet = {}
  define {
    isfull ≜ Len(buffer) = BufCapacity
    isempty ≜ Len(buffer) = 0
  }
  macro wait(id) { waitSet := waitSet ∪ {id} }
  macro notify() {
    if (waitSet ≠ {}) {
      with (p ∈ waitSet) { waitSet := waitSet \ {p}
  } }

  process (p ∈ Producers) {
    while (TRUE) {
      await (self ∉ waitSet); put(self)
  } }
  macro put(id) {
    if (isfull) { wait(id) }
    else {
      buffer := Append(buffer, id);
      notify()
  } }
  macro get(id) {
    if (isempty) { wait(id) }
    else {
      buffer := Tail(buffer);
      notify();
  } };
  process (c ∈ Consumers) {
    while (TRUE) {
      await (self ∉ waitSet); get(self)
  } }
}
```

State Space Exploration Using TLC

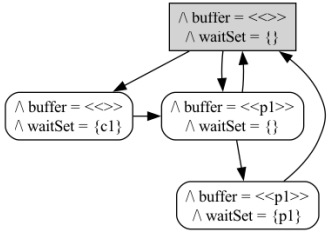
- Check concrete instances
  - first generate TLA$^+$ specification
  - define instance: configuration
  - by default, TLC generates state space and checks for deadlock



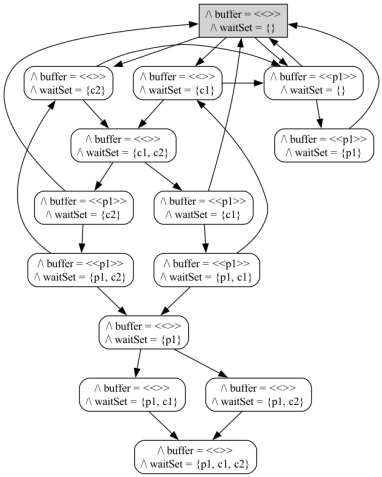
1 prod., 1 cons., buf. cap. 1

State Space Exploration Using TLC

- Check concrete instances
  - first generate TLA$^+$ specification
  - define instance: configuration
  - by default, TLC generates state space and checks for deadlock



1 prod., 1 cons., buf. cap. 1



1 prod., 2 cons., buf. cap. 1: deadlock

Fixing the Bounded Queue

- `notify()` may wake up a process that doesn't help making progress
- Distinguish between notifying producer or consumer

macro *notifyC*() {
  if ($waitSet \cap Consumers \neq \{\}$) {
    with ($c \in waitSet \cap Consumers$) { $waitSet := waitSet \setminus \{c\}$ }
} }
macro *notifyP*() {
  if ($waitSet \cap Producers \neq \{\}$) {
    with ($p \in waitSet \cap Producers$) { $waitSet := waitSet \setminus \{p\}$ }
} }

Fixing the Bounded Queue

- `notify()` may wake up a process that doesn't help making progress
- Distinguish between notifying producer or consumer

macro *notifyC*() {
  if ($waitSet \cap Consumers \neq \{\}$) {
    with ($c \in waitSet \cap Consumers$) { $waitSet := waitSet \setminus \{c\}$ }
} }
macro *notifyP*() {
  if ($waitSet \cap Producers \neq \{\}$) {
    with ($p \in waitSet \cap Producers$) { $waitSet := waitSet \setminus \{p\}$ }
} }

- Check an invariant that expresses that one "helpful" process is not blocked

$\land\ isempty \Rightarrow \exists prod \in Producers : prod \notin waitSet$
$\land\ isfull \Rightarrow \exists cons \in Consumers : cons \notin waitSet$

Part II

The Anatomy of TLA$^+$

Objectives

- Understand the syntax and semantics of TLA$^+$
  - set theory for representing data
  - temporal logic for describing executions
- Properties of executions, safety and liveness

Outline

Elementary Set Theory

- How many elements do the following sets contain?

  $\{\}$ $\quad$ $\{3, 42\}$ $\quad$ $\{1, 2, 2, 1, 2\}$ $\quad$ $\{\ \{1, 2\}, \{1, 3\}, \{2, 3, 4\}\ \}$

Elementary Set Theory

- How many elements do the following sets contain?

  $\{\}$ $\quad \{3, 42\}$ $\quad \{1, 2, 2, 1, 2\}$ $\quad \{\, \{1, 2\}, \{1, 3\}, \{2, 3, 4\} \,\}$

- How many different sets do you see?

  $\{1, 2\}$ $\quad \{2, 1\}$ $\quad \{1, 2, 2, 1\}$ $\quad \{1, 1, 1, 2, 2\}$

Elementary Set Theory

- How many elements do the following sets contain?

  $\{\}$ $\quad \{3, 42\}$ $\quad \{1, 2, 2, 1, 2\}$ $\quad \{\, \{1, 2\}, \{1, 3\}, \{2, 3, 4\} \,\}$

- How many different sets do you see?

  $\{1, 2\}$ $\quad \{2, 1\}$ $\quad \{1, 2, 2, 1\}$ $\quad \{1, 1, 1, 2, 2\}$

- Elementary operations on sets
  - union $\{1, 2\} \cup \{2, 3\}$
  - intersection $\{1, 2\} \cap \{2, 3\}$
  - set difference $\{1, 2\} \setminus \{2, 3\}$
  - subset relation $\{1, 2\} \quad \{1, 2, 3\} \qquad \{1, 2\} \quad \{2, 3\}$

Elementary Set Theory

- How many elements do the following sets contain?

  $\{\}$ $\quad \{3, 42\}$ $\quad \{1, 2, 2, 1, 2\}$ $\quad \{\, \{1, 2\}, \{1, 3\}, \{2, 3, 4\} \,\}$

- How many different sets do you see?

  $\{1, 2\}$ $\quad \{2, 1\}$ $\quad \{1, 2, 2, 1\}$ $\quad \{1, 1, 1, 2, 2\}$

- Elementary operations on sets
  - union $\quad \{1, 2\} \cup \{2, 3\} = \{1, 2, 3\}$
  - intersection $\quad \{1, 2\} \cap \{2, 3\} = \{2\}$
  - set difference $\quad \{1, 2\} \setminus \{2, 3\} = \{1\}$
  - subset relation $\quad \{1, 2\} \subseteq \{1, 2, 3\} \qquad \{1, 2\} \not\subseteq \{2, 3\}$

More Set Theory

- Generalized set union and powerset
  - UNION S union of all elements of the set (of sets) S
  - UNION $\{ \{0\}, \{0,1\}, \{0,1,2\}, \dots \} = Nat$
  - SUBSET S set of all subsets of S, a.k.a. powerset of S
  - SUBSET $\{1, 2\} = \{ \{\}, \{1\}, \{2\}, \{1, 2\} \}$

More Set Theory

- Generalized set union and powerset
  - UNION S union of all elements of the set (of sets) S
  - UNION $\{ \{0\}, \{0,1\}, \{0,1,2\}, \dots \} = Nat$
  - SUBSET S set of all subsets of S, a.k.a. powerset of S
  - SUBSET $\{1, 2\} = \{ \{\}, \{1\}, \{2\}, \{1, 2\} \}$
- Set filtering ("separation")
  - $\{x \in S : P(x)\}$ subset of S containing elements for which P holds
  - $\{x \in Nat : x \,\%\, 2 = 0\} = \{0, 2, 4, \dots\}$

More Set Theory

- Generalized set union and powerset
  - UNION S union of all elements of the set (of sets) S
  - UNION $\{ \{0\}, \{0,1\}, \{0,1,2\}, \dots \} = Nat$
  - SUBSET S set of all subsets of S, a.k.a. powerset of S
  - SUBSET $\{1, 2\} = \{ \{\}, \{1\}, \{2\}, \{1, 2\} \}$
- Set filtering ("separation")
  - $\{x \in S : P(x)\}$ subset of S containing elements for which P holds
  - $\{x \in Nat : x \,\%\, 2 = 0\} = \{0, 2, 4, \dots\}$
- Set mapping ("replacement")
  - $\{h(x) : x \in S\}$ image of S under the operator h
  - $\{2 * x : x \in 1..5\} = \{2, 4, 6, 8, 10\}$

More Set Theory

- Generalized set union and powerset
  - UNION S union of all elements of the set (of sets) S
  - UNION $\{ \{0\}, \{0,1\}, \{0,1,2\}, \dots \} = Nat$
  - SUBSET S set of all subsets of S, a.k.a. powerset of S
  - SUBSET $\{1, 2\} = \{ \{\}, \{1\}, \{2\}, \{1, 2\} \}$
- Set filtering ("separation")
  - $\{x \in S : P(x)\}$ subset of S containing elements for which P holds
  - $\{x \in Nat : x \,\%\, 2 = 0\} = \{0, 2, 4, \dots\}$
- Set mapping ("replacement")
  - $\{h(x) : x \in S\}$ image of S under the operator h
  - $\{2 * x : x \in 1..5\} = \{2, 4, 6, 8, 10\}$
- Note: $\{x : x \notin x\}$ is not a valid expression!

Functions in TLA$^+$

- Functions are primitive objects in TLA$^+$
  - represent mappings, for example a counter value per process
  - TLA$^+$ uses array-like notation for functions

$[x \in S \mapsto e]$	function with domain S mapping x to e
$f[e]$	application of function f to argument e
$\text{DOMAIN}\, f$	domain of function f
$[S \to T]$	set of functions with domain S and codomain T
$[f \ \text{EXCEPT}\ ![t] = e]$	function similar to f, mapping $t \in \text{DOMAIN}\, f$ to e
	$[f \ \text{EXCEPT}\ ![n] = @ + 1]$ @ refers to $f[n]$

Functions in TLA$^+$

- Functions are primitive objects in TLA$^+$
  - represent mappings, for example a counter value per process
  - TLA$^+$ uses array-like notation for functions

$[x \in S \mapsto e]$	function with domain S mapping x to e
$f[e]$	application of function f to argument e
DOMAIN f	domain of function f
$[S \to T]$	set of functions with domain S and codomain T
$[f \text{ EXCEPT } ![t] = e]$	function similar to f, mapping $t \in \text{DOMAIN } f$ to e
	$[f \text{ EXCEPT } ![n] = @ + 1]$ @ refers to $f[n]$

- Note
  - $f[x]$ is unspecified for $x \notin \text{DOMAIN } f$
  - notations extend to n-ary functions: $[x \in S, y \in T \mapsto x + y]$

Numbers in TLA$^+$

- Natural, integer, and real numbers defined in standard modules
  - axiomatic definitions, e.g. *Nat* is some set (with 0 and successor) satisfying Peano axioms
  - existence of these structures can be justified from axioms of ZFC set theory
  - set inclusions: $Nat \subseteq Int \subseteq Real$
  - compatibility of arithmetic operations: $3 - (-7) \in Nat$, $2.5 - 3.5 \in Int$

Numbers in TLA$^+$

- Natural, integer, and real numbers defined in standard modules
  - axiomatic definitions, e.g. *Nat* is some set (with 0 and successor) satisfying Peano axioms
  - existence of these structures can be justified from axioms of ZFC set theory
  - set inclusions: $Nat \subseteq Int \subseteq Real$
  - compatibility of arithmetic operations: $3 - (-7) \in Nat$, $2.5 - 3.5 \in Int$
- Standard operators on numbers
  - arithmetic operations: $+, -, *, \hat{}, \div$ and % (on integers), / (on reals), $<$
  - $i \,..\, k$ integer interval $\{j \in Int : i \leq j \wedge j \leq k\}$
- Note
  - real numbers are not supported by TLA$^+$ verifiers

Tuples and sequences

- Tuples and sequences are represented as functions
  - $\langle e_1, \ldots, e_n \rangle$ denotes $[i \in 1..n \mapsto \text{IF } i = 1 \text{ THEN } e_1 \ \ldots \ \text{ELSE } e_n]$
  - $e[i]$ denotes the i-th element of a tuple or sequence e

- Standard operators on sequences (extend standard module Sequences)

$Seq(S)$	set of sequences with elements in S
$Len(s)$	length of sequence s
$Head(s)$	first element of non-empty sequence s
$Tail(s)$	remainder of non-empty sequence s with first element removed
$s \circ t$	concatenation of sequences s and t
$Append(s, e)$	add e at the end of sequence s, equals $s \circ \langle e \rangle$
$SubSeq(s, m, n)$	subsequence of s between m-th and n-th elements (inclusive)

Strings and Records

- Strings in TLA$^+$
  - strings are sequences of characters
  - TLA$^+$ does not specify what characters are, tools use (at least) ASCII
  - strings are commonly used as constants, example: distinguish between message types
- Records are functions whose domain is a finite set of strings

record notation	instead of
$account.bal$	$account[\text{“bal”}]$
$[account \text{ EXCEPT } !.bal = @ + sum]$	$[account \text{ EXCEPT } ![\text{“bal”}] = @ + sum]$
$[owner \mapsto \text{“jones”},\ bal \mapsto 12.34]$	$[fld \in \{\text{“owner”}, \text{“bal”}\} \mapsto$ IF $fld = \text{“owner”}$ THEN “jones” ELSE $12.34]$
$[owner : \text{STRING},\ bal : Int]$	$\{f \in [\{\text{“owner”}, \text{“bal”}\} \to \text{STRING} \cup Int] :$ $f[\text{“owner”}] \in \text{STRING} \land f[\text{“bal”}] \in Int\}$

Elementary Mathematical Logic

- Atomic formulas: predicates applied to expressions

$3 > 0 \qquad z = x + 2 \qquad i \in Nat$

Elementary Mathematical Logic

- Atomic formulas: predicates applied to expressions

$$3 > 0 \qquad z = x + 2 \qquad i \in Nat$$

- Boolean combinations

TLA$^+$	programming	meaning
$\neg P$	`!P`	negation: true when P is false
$P \wedge Q$	`P && Q`	conjunction: both P and Q must be true
$P \vee Q$	`P || Q`	disjunction: at least one of P or Q must be true
$P \Rightarrow Q$	—	implication: same as $\neg P \vee Q$
$P \equiv Q$	`P == Q`	equivalence: P and Q both true, or both false

Elementary Mathematical Logic

- Atomic formulas: predicates applied to expressions

  $3 > 0 \qquad z = x + 2 \qquad i \in Nat$

- Boolean combinations

TLA$^+$	programming	meaning
$\neg P$	`!P`	negation: true when P is false
$P \wedge Q$	`P && Q`	conjunction: both P and Q must be true
$P \vee Q$	`P || Q`	disjunction: at least one of P or Q must be true
$P \Rightarrow Q$	—	implication: same as $\neg P \vee Q$
$P \equiv Q$	`P == Q`	equivalence: P and Q both true, or both false

- (Bounded) Quantification
  - $\forall x \in S : P(x)$ P is true for all values x in set S
  - $\exists x \in S : P(x)$ P is true for at least one value x in set S

Which Of The Following Formulas Are True?

- $\forall\, n \in Nat : n > 0$
- $\exists\, k \in Nat : k + k = 7$
- $\forall\, n \in Nat : n + n = 4 \Rightarrow n * n = 4$
- $\exists\, n \in Nat : n + n = 4 \Rightarrow n = 3$
- $\forall\, x \in \{\} : 0 = 1$
- $\exists\, x \in \{\} : x = x$
- $\neg(\exists\, x \in S : P(x)) \equiv (\forall\, x \in S : \neg P(x))$
- $0 \div 0 = 1$
- $42 \wedge \text{TRUE}$

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: the only $n \in Nat$ for which $n + n = 4$ is $n = 2$
- $\exists n \in Nat : n + n = 4 \Rightarrow n = 3$ — true, e.g. $1 + 1 \neq 4$
- $\forall x \in \{\} : 0 = 1$ — 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
- $42 \wedge \text{TRUE}$ — unspecified

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: the only $n \in Nat$ for which $n + n = 4$ is $n = 2$
- $\exists n \in Nat : n + n = 4 \Rightarrow n = 3$ — true, e.g. $1 + 1 \neq 4$
- $\forall x \in \{\} : 0 = 1$ — 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
- $42 \wedge \text{TRUE}$ — unspecified

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

Hilbert's Choice Operator

- CHOOSE $x : P$ and bounded variant CHOOSE $x \in S : P$
  - denotes fixed, arbitrary value x (resp., $x \in S$) satisfying P
  - if no such value exists, denotes some fixed, arbitrary value

Hilbert's Choice Operator

- CHOOSE $x : P$ and bounded variant CHOOSE $x \in S : P$
  - denotes fixed, arbitrary value x (resp., $x \in S$) satisfying P
  - if no such value exists, denotes some fixed, arbitrary value
- Powerful tool for mathematical definitions

$$\text{UNION } S \;=\; \text{CHOOSE } M : \forall x : (x \in M \;\equiv\; \exists T \in S : x \in T)$$

$$\exists x \in S : P(x) \;\equiv\; P(\text{CHOOSE } x \in S : P(x))$$

Hilbert's Choice Operator

- CHOOSE $x : P$ and bounded variant CHOOSE $x \in S : P$
  - denotes fixed, arbitrary value x (resp., $x \in S$) satisfying P
  - if no such value exists, denotes some fixed, arbitrary value
- Powerful tool for mathematical definitions

$$\text{UNION } S \;=\; \text{CHOOSE } M : \forall x : (x \in M \;\equiv\; \exists T \in S : x \in T)$$

$$\exists x \in S : P(x) \;\equiv\; P(\text{CHOOSE } x \in S : P(x))$$

- Note: choice is deterministic $(\text{CHOOSE } x : P(x)) \;=\; (\text{CHOOSE } y : P(y))$
  - in fact, CHOOSE satisfies the following determinacy law

$$(\forall x : P(x) \;\equiv\; Q(x)) \;\Rightarrow\; ((\text{CHOOSE } x : P(x)) = (\text{CHOOSE } x : Q(x)))$$

  - however, several values may be possible: $\text{CHOOSE } x \in Int : x * x = 4$

Common TLA$^+$ Idioms Involving CHOOSE

- Introduce a "null" value

$$NotAnS \triangleq \text{CHOOSE } x : x \notin S$$

  - well defined because no set can contain all values of the universe

Common TLA$^+$ Idioms Involving CHOOSE

- Introduce a "null" value

$$NotAnS \triangleq \text{CHOOSE } x : x \notin S$$

  - well defined because no set can contain all values of the universe

- Recursive function definitions

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

is shorthand for

$$fact \triangleq \text{CHOOSE } f : f = [n \in Nat \mapsto \text{IF } n = 0 \text{ THEN } 1 \text{ ELSE } n * f[n-1]]$$

  - should justify the existence and unicity of such a function

Exercises

1. Define the set $DistinctSeq(S)$ of sequences of elements in S that do not contain multiple occurrences of any element.

2. Define an operator $IsSorted(s)$ such that for any sequence $s \in DistinctSeq(Int)$, $IsSorted(s)$ is true if and only if s is sorted.

3. Define a function $sort \in [DistinctSeq(Int) \to DistinctSeq(Int)]$ that for any $s \in DistinctSeq(Int)$ returns a sorted (distinct) sequence containing the same elements as s.

4. Give a recursive definition $insort$ of the insertion sort algorithm over $DistinctSeq(Int)$. Does $sort = insort$ hold for your definitions?

Functions vs. Operators

- Functions (and records) are TLA$^+$ values
  - similar to arrays or tables in programming languages
  - typically, their value evolves during the run of an algorithm
  - functions have a domain (which is a set)
  - you may define sets of functions and quantify over them

Functions vs. Operators

- Functions (and records) are TLA$^+$ values
  - similar to arrays or tables in programming languages
  - typically, their value evolves during the run of an algorithm
  - functions have a domain (which is a set)
  - you may define sets of functions and quantify over them
- Operators live at the meta-level of TLA$^+$
  - describe operations on data
  - the arguments an operator may be applied to need not form a set
    for example, $Head(\_)$ applies to sequences over arbitrary elements
  - operators may take operators as arguments

$$Map(f(\_), s) \triangleq [i \in 1 \,..\, Len(s) \mapsto f(s[i])]$$

Outline

Constants and Variables

- TLA$^+$ formulas are built from constants and variables
- Constants
  - constant parameters declared in a module — N, *Producers*
  - constant literals representing values — 0, $\{1,2\}$, $\langle\rangle$
  - constants denote the same value in every state

Constants and Variables

- TLA$^+$ formulas are built from constants and variables
- Constants
  - constant parameters declared in a module — N, *Producers*
  - constant literals representing values — 0, $\{1, 2\}$, $\langle\rangle$
  - constants denote the same value in every state
- (State) variables
  - variable parameters declared in a module — *buffer*, *waitSet*
  - values represent current system state, may change from one state to another

Levels of TLA$^+$ Formulas

- Constant formulas $N > 0$

  do not contain state variables

Levels of TLA$^+$ Formulas

- Constant formulas $N > 0$

  do not contain state variables

- State formulas $buffer = \langle\rangle$

  contain constants and unprimed variables

Levels of TLA$^+$ Formulas

- Constant formulas $N > 0$

  do not contain state variables

- State formulas $buffer = \langle\rangle$

  contain constants and unprimed variables

- Action formulas $waitSet' = waitSet \cup \{p\}$

  may also contain primed variables

Levels of TLA$^+$ Formulas

- Constant formulas $N > 0$

  do not contain state variables

- State formulas $buffer = \langle\rangle$

  contain constants and unprimed variables

- Action formulas $waitSet' = waitSet \cup \{p\}$

  may also contain primed variables

- Temporal formulas $\Box\Diamond(waitSet = \{\})$

  built using temporal operators

Levels of TLA$^+$ Formulas

- Constant formulas $N > 0$

  do not contain state variables

- State formulas $buffer = \langle\rangle$

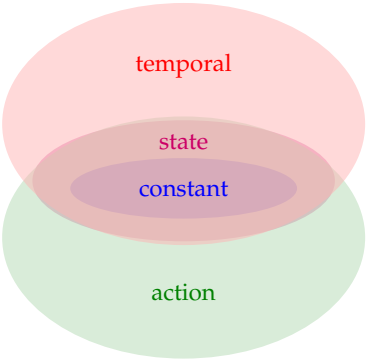
  contain constants and unprimed variables

- Action formulas $waitSet' = waitSet \cup \{p\}$

  may also contain primed variables

- Temporal formulas $\Box\Diamond(waitSet = \{\})$

  built using temporal operators



State Formulas

- States assign values to variables
  - states represent system configurations
  - formally, they assign values to all (countably many) variables
  - in practice, we only care about the variables declared in the module

State Formulas

- States assign values to variables
  - states represent system configurations
  - formally, they assign values to all (countably many) variables
  - in practice, we only care about the variables declared in the module
- State formulas represent predicates on states
  - initial conditions $buffer = \langle\rangle$
  - transition guards $p \notin waitSet$
  - invariants $isempty \Rightarrow \exists\, prod \in Producers : prod \notin waitSet$

State Formulas

- States assign values to variables
  - states represent system configurations
  - formally, they assign values to all (countably many) variables
  - in practice, we only care about the variables declared in the module
- State formulas represent predicates on states
  - initial conditions $buffer = \langle\rangle$
  - transition guards $p \notin waitSet$
  - invariants $isempty \Rightarrow \exists\, prod \in Producers : prod \notin waitSet$
- Standard first-order formulas built from constants and state variables

Action Formulas

- Actions represent state transitions

$$
\begin{aligned}
&\land \mathit{isfull} \\
&\land \mathit{waitSet}' = \mathit{waitSet} \cup \{\mathit{self}\} \\
&\land \mathit{buffer}' = \mathit{buffer}
\end{aligned}
$$

  - unprimed variables are evaluated in the state before the transition
  - primed variables are evaluated in the state after the transition
  - predicates, not statements: must also say what does not change

- Actions are the building blocks for TLA$^+$ specifications
  - typically, the next-state relation is a disjunction of actions
  - every disjunct represents one possible transition

Priming of State Formulas

- Action formula denoting value of state formula e at successor state
  - e' is the action formula obtained by replacing every variable v by v'
  - for example, $isfull'$ stands for $Len(buffer') = BufCapacity$
  - note: binders $\forall$, $\exists$, CHOOSE are only applied to logic variables, not state variables

Priming of State Formulas

- Action formula denoting value of state formula e at successor state
  - e' is the action formula obtained by replacing every variable v by v'
  - for example, $isfull'$ stands for $Len(buffer') = BufCapacity$
  - note: binders $\forall$, $\exists$, CHOOSE are only applied to logic variables, not state variables

- UNCHANGED $e \triangleq e' = e$
  - useful macro for specifying what parts of the state remain unchanged in an action
  - often applied to tuples: UNCHANGED $\langle x, y, z \rangle \equiv x' = x \wedge y' = y \wedge z' = z$

Bracketed Action Formulas

$[A]_e \triangleq A \lor e' = e$ action A or stuttering (w.r.t. state formula e)

$\langle A \rangle_e \triangleq A \land e' \neq e$ action A and change of state formula e

- assert if "stuttering" (repetition of current state) may or must not occur
- these will be important building blocks for temporal formulas
- note duality $\neg\langle A \rangle_e \equiv [\neg A]_e \qquad \neg[A]_e \equiv \langle \neg A \rangle_e$

Enabledness of Actions

- State formula asserting if an action may occur
  - ENABLED A is obtained by existential quantification over the primed state variables in A
  - formally, substitute primed variables by logical variables,then quantify over them

$$A \triangleq \begin{array}{l} \land\ isfull \\ \land\ waitSet' = waitSet \cup \{self\} \\ \land\ buffer' = buffer \end{array} \qquad \text{ENABLED } A \equiv \exists\, wsp, bp : \begin{array}{l} \land\ isfull \\ \land\ wsp = waitSet \cup \{self\} \\ \land\ bp = buffer \end{array}$$

  - semantically, ENABLED A holds at state s if A holds for some $\langle s, t \rangle$

- TLA$^+$ also has an operator $A \cdot B$ for the sequential composition of actions

Temporal Formulas of TLA$^+$

- Linear-time logic: interpreted over behavior, i.e. ω-sequence $\sigma = s_0 s_1 \ldots$ of states
  - $\sigma, \xi \models F$ denotes that F is true of σ with valuation ξ of logical variables
  - $\models F$ means that F is valid, i.e. $\sigma \models F$ for all σ, ξ
  - write $\sigma[n..]$ for suffix $s_n s_{n+1} \ldots$

Temporal Formulas of TLA$^+$

- Linear-time logic: interpreted over behavior, i.e. ω-sequence $\sigma = s_0 s_1 \ldots$ of states
  - $\sigma, \xi \models F$ denotes that F is true of σ with valuation ξ of logical variables
  - $\models F$ means that F is valid, i.e. $\sigma \models F$ for all σ, ξ
  - write $\sigma[n..]$ for suffix $s_n s_{n+1} \ldots$

- Inductive definition of temporal formulas

state formulas	$\sigma, \xi \models P$	iff	P true at s_0 for ξ
Boolean combinations	$\sigma, \xi \models F \wedge G$	iff	$\sigma, \xi \models F$ and $\sigma, \xi \models G$ (etc.)
always	$\sigma, \xi \models \Box F$	iff	$\sigma[n..], \xi \models F$ for all $n \in \mathbb{N}$
always square A sub e	$\sigma, \xi \models \Box[A]_e$	iff	$[A]_e$ holds for all $\langle s_n, s_{n+1} \rangle$ for ξ
quantifiers	$\sigma, \xi \models \exists x : F$	iff	$\sigma, \xi' \models F$ for some $\xi' =_x \xi$

NB: will usually drop variable valuation ξ from notation

Notes on Temporal Formulas

- State formulas are temporal formulas, but actions are not
  - not even in the form $[A]_e$
  - the restriction to $\Box[A]_e$ ensures invariance under stuttering

Notes on Temporal Formulas

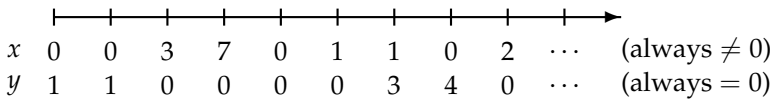
- State formulas are temporal formulas, but actions are not
  - not even in the form $[A]_e$
  - the restriction to $\Box[A]_e$ ensures invariance under stuttering
- Derived formulas
  - eventually F $\quad \Diamond F \triangleq \neg\Box\neg F$
  - F holds some time in the future, i.e. for some suffix of the behavior
  - eventually angle A sub e $\quad \Diamond\langle A\rangle_e \triangleq \neg\Box[\neg A]_e$
  - action A, modifying e, occurs at some future state transition
  - F leadsto G $\quad F \rightsquigarrow G \triangleq \Box(F \Rightarrow \Diamond G)$
  - whenever F holds, G must hold some time later

Notes on Temporal Formulas

- State formulas are temporal formulas, but actions are not
  - not even in the form $[A]_e$
  - the restriction to $\Box[A]_e$ ensures invariance under stuttering
- Derived formulas
  - eventually F $\quad \Diamond F \triangleq \neg\Box\neg F$
  - F holds some time in the future, i.e. for some suffix of the behavior
  - eventually angle A sub e $\quad \Diamond\langle A\rangle_e \triangleq \neg\Box[\neg A]_e$
  - action A, modifying e, occurs at some future state transition
  - F leadsto G $\quad F \rightsquigarrow G \triangleq \Box(F \Rightarrow \Diamond G)$
  - whenever F holds, G must hold some time later
- Comparison with standard linear-time temporal logic LTL
  - TLA has action formulas for describing state transitions
  - TLA does not have binary temporal operators such as “until”

Example: Semantics of Temporal Formulas

Which of the following formulas hold in this behavior?



- $\Box\neg(x = 0 \wedge y = 0)$
- $\Box[x = 0 \Rightarrow y' = 0]_{x,y}$
- $\Diamond(x = 7 \wedge y = 0)$
- $\Diamond\langle y = 0 \wedge x' = 0\rangle_y$
- $\Box\Diamond(y \neq 0)$
- $\Diamond\Box(x = 0 \Rightarrow y \neq 0)$
- $\Diamond\Box[\text{FALSE}]_y$

Infinitely often and eventually always

- $\Box\Diamond F$ asserts that F holds infinitely often
  - $\sigma \models \Box\Diamond F$ iff for all $m \in \mathbb{N}$ there is $n \geq m$ such that $\sigma[n..] \models F$
  - similarly: $\Box\Diamond\langle A\rangle_e$ asserts that $\langle A\rangle_e$ occurs infinitely often
- $\Diamond\Box F$ asserts that F continues to hold from some time on
  - equivalently, F is false only finitely often
  - similarly: $\Diamond\Box[A]_e$ asserts that eventually only $[A]_e$ actions occur
- Equivalences

$$\begin{array}{rcl} \neg\Box\Diamond F & \equiv & \Diamond\Box\neg F \\ \Box\Box F & \equiv & \Box F \\ \Diamond\Box\Diamond F & \equiv & \Box\Diamond F \end{array} \qquad \begin{array}{rcl} \neg\Diamond\Box F & \equiv & \Box\Diamond\neg F \\ \Diamond\Diamond F & \equiv & \Diamond F \\ \Box\Diamond\Box F & \equiv & \Diamond\Box F \end{array}$$

Fairness Conditions

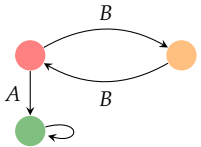
- Weak fairness $\mathrm{WF}_e(A) \triangleq \Box(\Box\mathrm{ENABLED}\ \langle A\rangle_e \Rightarrow \Diamond\langle A\rangle_e)$
  - action will be taken eventually if it is continuously enabled
  - note $\langle A\rangle_e$: cannot require fairness of stuttering actions

Fairness Conditions

- Weak fairness $\mathrm{WF}_e(A) \triangleq \Box(\Box\mathrm{ENABLED}\ \langle A\rangle_e \Rightarrow \Diamond\langle A\rangle_e)$
  - action will be taken eventually if it is continuously enabled
  - note $\langle A\rangle_e$: cannot require fairness of stuttering actions
- Strong fairness $\mathrm{SF}_e(A) \triangleq \Box(\Box\Diamond\mathrm{ENABLED}\langle A\rangle_e \Rightarrow \Diamond\langle A\rangle_e)$
  - action will be taken eventually if it is infinitely often enabled

Fairness Conditions

- Weak fairness $\mathrm{WF}_e(A) \triangleq \Box(\Box\mathrm{ENABLED}\ \langle A\rangle_e \Rightarrow \Diamond\langle A\rangle_e)$
  - action will be taken eventually if it is continuously enabled
  - note $\langle A\rangle_e$: cannot require fairness of stuttering actions
- Strong fairness $\mathrm{SF}_e(A) \triangleq \Box(\Box\Diamond\mathrm{ENABLED}\langle A\rangle_e \Rightarrow \Diamond\langle A\rangle_e)$
  - action will be taken eventually if it is infinitely often enabled
- Comparing weak and strong fairness
  - overall idea: an action must occur if it is "often enough" enabled
  - "often enough" is weaker for strong fairness, therefore the overall condition is stronger
  - consider the following state graph, which of the formulas hold?

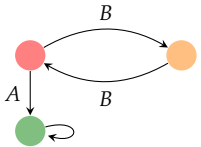


$\mathrm{WF}_v(A) \Rightarrow (red \rightsquigarrow green)$

$\mathrm{SF}_v(A) \Rightarrow (red \rightsquigarrow green)$

Fairness Conditions

- Weak fairness $\mathrm{WF}_e(A) \triangleq \Box(\Box\mathrm{ENABLED}\ \langle A\rangle_e \Rightarrow \Diamond\langle A\rangle_e)$
  - action will be taken eventually if it is continuously enabled
  - note $\langle A\rangle_e$: cannot require fairness of stuttering actions
- Strong fairness $\mathrm{SF}_e(A) \triangleq \Box(\Box\Diamond\mathrm{ENABLED}\langle A\rangle_e \Rightarrow \Diamond\langle A\rangle_e)$
  - action will be taken eventually if it is infinitely often enabled
- Comparing weak and strong fairness
  - overall idea: an action must occur if it is "often enough" enabled
  - "often enough" is weaker for strong fairness, therefore the overall condition is stronger
  - consider the following state graph, which of the formulas hold?



$\mathrm{WF}_v(A) \Rightarrow (red \rightsquigarrow green)$ no

$\mathrm{SF}_v(A) \Rightarrow (red \rightsquigarrow green)$ yes

Stuttering invariance

- Action formulas must be "bracketed" in TLA: $\Box[A]_e$, $\Diamond\langle A\rangle_e$
  - insensitive to repetitions of states in which e yields same value
  - this observation extends to all TLA formulas
- Stuttering equivalence ($\approx$)
  - Denote by $\natural\sigma$ the result of removing finite sequences of repeated states from σ
  - $\sigma \approx \tau$ iff $\natural\sigma = \natural\tau$

Stuttering invariance

- Action formulas must be "bracketed" in TLA: $\Box[A]_e$, $\Diamond\langle A\rangle_e$
  - insensitive to repetitions of states in which e yields same value
  - this observation extends to all TLA formulas
- Stuttering equivalence ($\approx$)
  - Denote by $\natural\sigma$ the result of removing finite sequences of repeated states from σ
  - $\sigma \approx \tau$ iff $\natural\sigma = \natural\tau$

Theorem (stuttering invariance)

For any temporal formula F and behaviors $\sigma \approx \tau$:

$$\sigma \models F \quad \textit{iff} \quad \tau \models F$$

System Specifications in TLA$^+$

- In TLA$^+$, systems are specified using temporal formulas
  - standard form of specification $Spec \triangleq Init \wedge \Box[Next]_{vars} \wedge L$
  - *Init* predicate describing initial states
  - *Next* transition relation, usually a disjunction of individual action formulas
  - *vars* tuple of all variables appearing in the specification
  - *L* liveness condition, usually a conjunction of fairness conditions

System Specifications in TLA$^+$

- In TLA$^+$, systems are specified using temporal formulas
  - standard form of specification $Spec \triangleq Init \wedge \Box[Next]_{vars} \wedge L$
  - $Init$ predicate describing initial states
  - $Next$ transition relation, usually a disjunction of individual action formulas
  - $vars$ tuple of all variables appearing in the specification
  - L liveness condition, usually a conjunction of fairness conditions
- The PlusCal translator generates a specification of that form
  - actions correspond to groups of statements occuring between labels
  - these statements are assumed to execute atomically

Outline

Properties of runs

- Properties are expressed as temporal formulas
  - properties are evaluated over behaviors
  - a system satisfies a property if the property holds for all system behaviors
  - in TLA$^+$, this is expressed as (validity of) $Spec \Rightarrow Prop$

Properties of runs

- Properties are expressed as temporal formulas
  - properties are evaluated over behaviors
  - a system satisfies a property if the property holds for all system behaviors
  - in TLA$^+$, this is expressed as (validity of) $Spec \Rightarrow Prop$
- Safety properties: "something bad never happens"
  - e.g., the blocking queue never deadlocks
  - if P, Q are state formulas then P, $\Box P$, $\Box(P \Rightarrow \Box Q)$ express safety properties
  - stuttering cannot break a safety property

Properties of runs

- Properties are expressed as temporal formulas
  - properties are evaluated over behaviors
  - a system satisfies a property if the property holds for all system behaviors
  - in TLA$^+$, this is expressed as (validity of) $Spec \Rightarrow Prop$
- Safety properties: "something bad never happens"
  - e.g., the blocking queue never deadlocks
  - if P, Q are state formulas then P, $\Box P$, $\Box(P \Rightarrow \Box Q)$ express safety properties
  - stuttering cannot break a safety property
- Liveness properties: "something good happens eventually"
  - e.g., whenever an item is put in the queue, it will eventually be removed
  - if P, Q are state formulas then $\Diamond P$, $\Box\Diamond P$, $\Diamond\Box P$, $P \rightsquigarrow Q$ express liveness properties
  - stuttering will not establish a liveness property

Things that TLA$^+$ Cannot Express

- Possibility properties
  - example: from every system state, the initial state can be reached
  - in branching-time logics such as CTL, this is written **AG EF** *init*
  - to some extent, such properties can be checked by asserting unreachability and obtaining a counter-example

Things that TLA$^+$ Cannot Express

- Possibility properties
  - example: from every system state, the initial state can be reached
  - in branching-time logics such as CTL, this is written **AG EF** *init*
  - to some extent, such properties can be checked by asserting unreachability and obtaining a counter-example
- Probabilistic properties
  - example: a system malfunction occurs with probability $\leq 10^{-6}$
  - TLA$^+$ can only speak about what is true for all runs

Things that TLA$^+$ Cannot Express

- Possibility properties
  - example: from every system state, the initial state can be reached
  - in branching-time logics such as CTL, this is written **AG EF** *init*
  - to some extent, such properties can be checked by asserting unreachability and obtaining a counter-example
- Probabilistic properties
  - example: a system malfunction occurs with probability $\leq 10^{-6}$
  - TLA$^+$ can only speak about what is true for all runs
- Hyperproperties
  - example: an attacker cannot distinguish two runs that differ in the value of the secret
  - TLA$^+$ formulas cannot compare different runs of a system
  - hyperproperties can be expressed using quantification over state variables (see later)

Digression: Classes of Linear-Time Properties

- Identify properties and sets of behaviors
  - define a property φ to be a set of ω-sequences of states
  - write $\sigma \models \varphi$ for $\sigma \in \varphi$
- Additional notation
  - for $\sigma = s_0 s_1 \ldots$ write $\sigma[..n] = s_0 s_1 \ldots s_n$
  - for $\sigma = s_0 \ldots s_n$ and $\tau = t_0 t_1 \ldots$ write $\sigma \circ \tau = s_0 \ldots s_n t_0 t_1 \ldots$
  - for a property φ and a finite sequence $\sigma = s_0 \ldots s_n$ write
    $\sigma \models \varphi$ if $\sigma \circ \tau \models \varphi$ for some τ
  - "optimistic interpretation": σ can still be extended to a behavior satisfying φ

Safety and Liveness (Alpern & Schneider 1985)

Definition

A property φ is a safety property if for every behavior σ:

$$\sigma \models \varphi \quad \text{iff} \quad \sigma[..n] \models \varphi \ \text{ for all } n \in \mathbb{N}.$$

- σ does not satisfy a safety property if there is a finite prefix $\sigma[..n]$ that cannot be extended to an infinite behavior satisfying φ
- "bad things" are observable after some finite time

Safety and Liveness (Alpern & Schneider 1985)

Definition

A property φ is a safety property if for every behavior σ:

$$\sigma \models \varphi \quad \text{iff} \quad \sigma[..n] \models \varphi \ \text{ for all } n \in \mathbb{N}.$$

- σ does not satisfy a safety property if there is a finite prefix $\sigma[..n]$ that cannot be extended to an infinite behavior satisfying φ
- "bad things" are observable after some finite time

Definition

A property φ is a liveness property if $\sigma \models \varphi$ holds for all finite behaviors σ.

- liveness properties do not constrain finite prefixes
- "good things" may still occur in the future

Safety and Liveness: Examples and Exercises

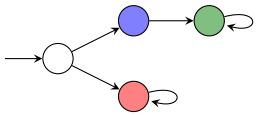
1. The formula $Init \wedge \Box[Next]_{vars}$ describes a safety property.
2. Weak and strong fairness conditions are liveness properties.
3. If all φ_i ($i \in I$) are safety properties then $\bigcap_{i \in I} \varphi_i$ is a safety property.
4. If φ is a liveness property and $\psi \supseteq \varphi$ then ψ is a liveness property.
5. The set of all behaviors is the only property that is both safety and liveness.
6. Given a property φ, its safety closure

   $$\mathcal{C}(\varphi) \triangleq \{\sigma : \sigma[..n] \models \varphi \text{ for all } n \in \mathbb{N}\}$$

   is the smallest safety property containing φ.
7. For any property φ there exist a safety property S_φ and a liveness property L_φ such that $\varphi = S_\varphi \cap L_\varphi$.

Machine Closure

- Specifications: state machine plus liveness property L
  - state machine may "paint itself into a corner" by reaching a state from which it can no longer satisfy L
  - conversely, L may constrain the transitions of the state machine



together with $\Diamond green$ satisfies $\Box\neg red$

Definition

Let S be a safety property and L be an arbitrary property.
The pair (S, L) is machine closed if $\mathcal{C}(S \cap L) = S$.

- Idea: L does not add any safety conditions to S

Machine-Closed Specifications

- If (S, L) is machine closed and φ is a safety property then $S \cap L \subseteq \varphi$ iff $S \subseteq \varphi$
  - may ignore the liveness property when proving safety properties of a specification

Machine-Closed Specifications

- If (S, L) is machine closed and φ is a safety property then $S \cap L \subseteq \varphi$ iff $S \subseteq \varphi$
  - may ignore the liveness property when proving safety properties of a specification
- Machine closed system specifications
  - $Init \wedge \Box[Next]_{vars} \wedge L$ is machine closed if L is a countable conjunction of (weak or strong) fairness conditions on disjuncts of $Next$

Machine-Closed Specifications

- If (S, L) is machine closed and φ is a safety property then $S \cap L \subseteq \varphi$ iff $S \subseteq \varphi$
  - may ignore the liveness property when proving safety properties of a specification
- Machine closed system specifications
  - $Init \wedge \Box[Next]_{vars} \wedge L$ is machine closed if L is a countable conjunction of (weak or strong) fairness conditions on disjuncts of $Next$
- In general, one should aim at writing machine closed specifications
  - exceptions exist for some very high-level properties such as linearizability

Part III

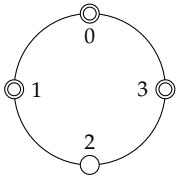
Specifying and Verifying Systems

Objectives

- Write a high-level specification of a state machine
- Use the TLA$^+$ tools to verify properties of the specification
  - explicit-state model checking using TLC
  - symbolic model checking using Apalache
  - theorem proving using TLAPS
- Concrete example: (synchronous) distributed termination detection

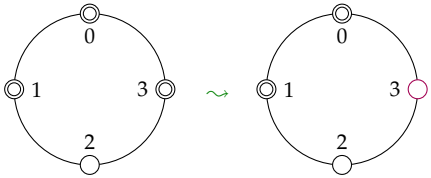
Outline

The Problem of Distributed Termination Detection



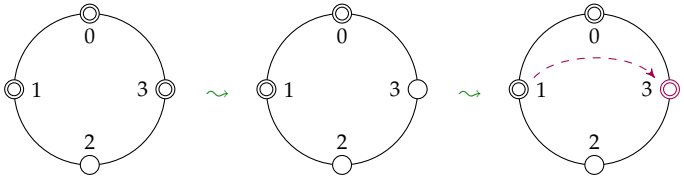
- Nodes in a distributed system perform some computation
  - nodes can be active (double circle) or inactive (simple circle)

The Problem of Distributed Termination Detection



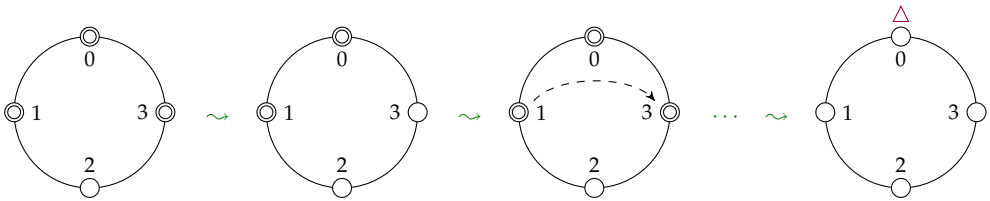
- Nodes in a distributed system perform some computation
  - nodes can be active (double circle) or inactive (simple circle)
- Possible transitions
  - an active node may locally terminate its computation

The Problem of Distributed Termination Detection



- Nodes in a distributed system perform some computation
  - nodes can be active (double circle) or inactive (simple circle)
- Possible transitions
  - an active node may locally terminate its computation
  - an active node may send a message to another node, activating the receiver

The Problem of Distributed Termination Detection



- Nodes in a distributed system perform some computation
  - nodes can be active (double circle) or inactive (simple circle)
- Possible transitions
  - an active node may locally terminate its computation
  - an active node may send a message to another node, activating the receiver
- Eventually, all nodes may have terminated
  - if this happens, it should eventually be signalled

Modeling Termination Detection in TLA$^+$: System Configurations

MODULE *TerminationDetection*

EXTENDS *Naturals*
CONSTANT N
ASSUME $NAssumption \triangleq N \in Nat \setminus \{0\}$
$Node \triangleq 0..N-1$
VARIABLES $active, termDetected$
$TypeOK \triangleq active \in [Node \to \text{BOOLEAN}] \land termDetected \in \text{BOOLEAN}$
$vars \triangleq \langle active, termDetected \rangle$
$terminated \triangleq \forall n \in Node : active[n] = \text{FALSE}$

- Import standard module *Naturals*
- Declare constant parameters and variables (untyped)
- Define some basic operators

Modeling Termination Detection in TLA$^+$: State Machine

- Initial condition

$$\begin{aligned} Init \triangleq\ & \land\ active \in [Node \to \text{BOOLEAN}] \\ & \land\ termDetected \in \{\text{FALSE}, terminated\} \end{aligned}$$

Modeling Termination Detection in TLA$^+$: State Machine

- Initial condition

$$\begin{aligned} Init \triangleq\ & \land\ active \in [Node \to \text{BOOLEAN}] \\ & \land\ termDetected \in \{\text{FALSE}, terminated\} \end{aligned}$$

- State transitions

$$\begin{aligned} Terminate(i) \triangleq\ & \land\ active[i] \land active' = [active \text{ EXCEPT } ![i] = \text{FALSE}] \\ & \land\ termDetected' \in \{termDetected, terminated'\} \\ Message(i,j) \triangleq\ & \land\ active[i] \land active' = [active \text{ EXCEPT } ![j] = \text{TRUE}] \\ & \land\ \text{UNCHANGED } termDetected \\ SignalTerminated \triangleq\ & \land\ terminated \land termDetected' = \text{TRUE} \\ & \land\ \text{UNCHANGED } active \end{aligned}$$

Modeling Termination Detection in TLA$^+$: State Machine

- Initial condition

$$\begin{aligned} Init \triangleq\ & \land\ active \in [Node \to \text{BOOLEAN}] \\ & \land\ termDetected \in \{\text{FALSE}, terminated\} \end{aligned}$$

- State transitions

$$\begin{aligned} Terminate(i) \triangleq\ & \land\ active[i] \land active' = [active \text{ EXCEPT } ![i] = \text{FALSE}] \\ & \land\ termDetected' \in \{termDetected, terminated'\} \\ Message(i,j) \triangleq\ & \land\ active[i] \land active' = [active \text{ EXCEPT } ![j] = \text{TRUE}] \\ & \land\ \text{UNCHANGED } termDetected \\ SignalTerminated \triangleq\ & \land\ terminated \land termDetected' = \text{TRUE} \\ & \land\ \text{UNCHANGED } active \end{aligned}$$

- Overall specification

$$\begin{aligned} Next &\triangleq \exists i, j \in Node : Terminate(i) \lor Message(i,j) \lor SignalTerminated \\ Spec &\triangleq Init \land \Box[Next]_{vars} \land \text{WF}_{vars}(SignalTerminated) \end{aligned}$$

Expressing Correctness Properties

1. Safety properties
   - type correctness

     $Spec \Rightarrow \Box TypeOK$

     TypeOK is true at all states, in any execution of *Spec*

Expressing Correctness Properties

1. Safety properties
   - type correctness
     $Spec \Rightarrow \Box TypeOK$
     TypeOK is true at all states, in any execution of *Spec*
   - correct termination detection
     $Spec \Rightarrow \Box(termDetected \Rightarrow terminated)$
     again expressed as an invariant

Expressing Correctness Properties

1. Safety properties
   - type correctness
     $Spec \Rightarrow \Box TypeOK$
     TypeOK is true at all states, in any execution of *Spec*
   - correct termination detection
     $Spec \Rightarrow \Box(termDetected \Rightarrow terminated)$
     again expressed as an invariant
   - quiescence of the system
     $Spec \Rightarrow \Box(terminated \Rightarrow \Box terminated)$

Expressing Correctness Properties

1. Safety properties
   - type correctness
     $Spec \Rightarrow \Box TypeOK$
     TypeOK is true at all states, in any execution of *Spec*
   - correct termination detection
     $Spec \Rightarrow \Box(termDetected \Rightarrow terminated)$
     again expressed as an invariant
   - quiescence of the system
     $Spec \Rightarrow \Box(terminated \Rightarrow \Box terminated)$
2. Liveness properties
   - eventual detection
     $Spec \Rightarrow \Box(terminated \Rightarrow \Diamond termDetected)$
     remember: the system isn't guaranteed to terminate

Outline

Verification Using TLC

- Create a model: finite instance of TLA$^+$ specification
  - instantiate constant parameters

    `CONSTANT` N=4
  - indicate operator corresponding to system specification

    `SPECIFICATION` Spec
  - indicate properties to verify and run TLC

    `INVARIANTS` TypeOK TDCorrect
    `PROPERTIES` Quiescence Liveness
  - TLC reports 17 distinct states (65 for $N = 6$)
- Push-button verification technique
  - TLC can be run from IDEs (TLA$^+$ Toolbox and VS Code Extension)

Basic Working of TLC: Initial States

- TLC expects a specification in standard form $Init \wedge \Box[Next]_{vars} \wedge L$
- Interpretation of *Init* for generating initial states
  - TLC evaluates *Init* from left to right
  - $x = t$ assigns the value of t to x: all variables in t must be defined
  - $x \in S$ where S evaluates to a finite set generates one initial states per element of S
  - later predicates involving x are evaluated and may suppress a state

Basic Working of TLC: Initial States

- TLC expects a specification in standard form $Init \wedge \Box[Next]_{vars} \wedge L$
- Interpretation of *Init* for generating initial states
  - TLC evaluates *Init* from left to right
  - $x = t$ assigns the value of t to x: all variables in t must be defined
  - $x \in S$ where S evaluates to a finite set generates one initial states per element of S
  - later predicates involving x are evaluated and may suppress a state
- Examples

$x \in 0..3$	TLC generates four initial states
$x \in Nat \wedge x \leq 3$	cannot be evaluated: *Nat* is infinite
$x \in \{n \in Nat : n \leq 3\}$	cannot be evaluated: filter applied to infinite set

Basic Working of TLC: Computing Successor States

- Interpretation of *Next* for generating successor states
  - *Next* is split into disjuncts, then successor states are generated for each disjunct
  - same principle as for initial states: evaluation from left to right
  - $x' = t$ and $x' \in S$ interpreted as assignments when x' is undefined and S is finite
  - other predicates evaluate to truth value and may block generation of successor states

- Examples: assume $x = 1$ and $y = 1$ in current state

$x' = x + y \land y' = x'$	results in successor where $x = y = 2$
$x' = x \land y' \in 0..x$	results in two successors where $x = 1$ and $y = 0$ resp. $y = 1$
$x < y \land x' = x \land y' \in 0..x$	produces no successor

Basic Working of TLC: Computing the State Graph

- TLC maintains a queue Q of unexplored states and a set S of seen states
- Initially, Q contains the initial states, S is empty
- As long as Q is non-empty:
  - remove the state s at the head of Q
  - if $s \notin S$, add all successors of s at the end of Q
  - add s to set S of explored states

Basic Working of TLC: Computing the State Graph

- TLC maintains a queue Q of unexplored states and a set S of seen states
- Initially, Q contains the initial states, S is empty
- As long as Q is non-empty:
  - remove the state s at the head of Q
  - if $s \notin S$, add all successors of s at the end of Q
  - add s to set S of explored states
- Observations
  - breadth-first exploration of the state graph
  - besides the states themselves, also record edges for evaluating temporal properties
  - state comparison: use hash function to determine if a state is new
  - when TLC runs out of main memory, it swaps to disk

TLC: Summing Up

- Invariants are checked during state computation
  - by default, TLC stops when a state violating an invariant is found
  - breadth-first search ensures minimal length of counter-example
- Temporal properties are checked on entire state graph
  - for large state spaces, TLC performs intermediate checks for counter-examples
  - due to stuttering transitions, liveness properties require fairness assumptions

TLC: Summing Up

- Invariants are checked during state computation
  - by default, TLC stops when a state violating an invariant is found
  - breadth-first search ensures minimal length of counter-example
- Temporal properties are checked on entire state graph
  - for large state spaces, TLC performs intermediate checks for counter-examples
  - due to stuttering transitions, liveness properties require fairness assumptions
- Tips for using TLC effectively
  - TLC-friendly specifications do not require computations of large sets
    e.g., $[S \to T]$ has $|T|^{|S|}$ elements, but checking $f \in [S \to T]$ is fast
  - start small and scale up gradually: state spaces are larger than you think!
  - be suspicious of success: counter-examples are most informative result
    look at number of distinct states, check non-properties

Outline

Bounded Model Checking Using Apalache

- Apalache: symbolic (SMT-based) model checker for TLA$^+$
  - check safety properties for bounded prefixes of executions
  - like TLC, checks properties of finite instances
  - relies on constraint solving rather than state enumeration

Bounded Model Checking Using Apalache

- Apalache: symbolic (SMT-based) model checker for TLA$^+$
  - check safety properties for bounded prefixes of executions
  - like TLC, checks properties of finite instances
  - relies on constraint solving rather than state enumeration
- Apalache requires type annotations for constants and variables

```
CONSTANT
  \* @type:  Int;
  N
VARIABLES
  \* @type:  Int → Bool;
  active,
  \* @type:  Bool;
  termDetected
```

- types help Apalache encode the spec for SMT solvers
- type checker makes sure that type annotations are respected

Running Apalache on the Model

- Apalache can use the same configuration files as TLC

  ```
  apalache-mc check --length k --config=XX.cfg XX.tla
  ```

  - invariants checked for all states reachable in k (default 10) steps
  - any temporal properties are ignored

- Complementary verification technique compared to explicit state enumeration
  - depending on the size of the state space, one or the other may be more efficient
  - Apalache can only check states reachable up to the bound
  - Apalache can sometimes handle infinite state spaces

Basic Working of Apalache

- For simplicity, assume that there are only integer variables
  - introduce $k+1$ copies $\vec{x}_0, \vec{x}_1, \ldots, \vec{x}_k$ of the state variables
  - assert that an invariant violation is reachable in k steps

    $$Init(\vec{x}_0) \wedge Next(\vec{x}_0, \vec{x}_1) \wedge \cdots \wedge Next(\vec{x}_{k-1}, \vec{x}_k) \wedge \neg Inv(\vec{x}_k)$$

  - use a constraint (SMT) solver for checking if this formula is satisfiable
  - if it is, the model represents a counter-example

Basic Working of Apalache

- For simplicity, assume that there are only integer variables
  - introduce $k+1$ copies $\vec{x}_0, \vec{x}_1, \ldots, \vec{x}_k$ of the state variables
  - assert that an invariant violation is reachable in k steps

    $$Init(\vec{x}_0) \wedge Next(\vec{x}_0, \vec{x}_1) \wedge \cdots \wedge Next(\vec{x}_{k-1}, \vec{x}_k) \wedge \neg Inv(\vec{x}_k)$$

  - use a constraint (SMT) solver for checking if this formula is satisfiable
  - if it is, the model represents a counter-example

- This idea extends to more complex specifications
  - encoding variables of complex types requires additional variables
  - complexity increases (exponentially) with the number of variables and the length k but is not directly related to the number of reachable states

Checking Inductive Invariants Using Apalache

- Inductive Invariants *Inv*
  - a predicate that is true initially ... $Init \Rightarrow Inv$
  - ... and is preserved by all transitions $Inv \land Next \Rightarrow Inv'$
  - note: not all invariants are inductive

Checking Inductive Invariants Using Apalache

- Inductive Invariants *Inv*
  - a predicate that is true initially ... $Init \Rightarrow Inv$
  - ... and is preserved by all transitions $Inv \land Next \Rightarrow Inv'$
  - note: not all invariants are inductive

- Checking inductive invariants such as $TypeOK \land (termDetected \Rightarrow terminated)$
  - the two conditions can be checked by Apalache for prefixes of length 0 and 1
  - the invariant then holds throughout any execution
  - verification easily scales up to $N = 100$
  - quiescence property can be checked similarly

Checking Inductive Invariants Using Apalache

- Inductive Invariants *Inv*
  - a predicate that is true initially ... $Init \Rightarrow Inv$
  - ... and is preserved by all transitions $Inv \land Next \Rightarrow Inv'$
  - note: not all invariants are inductive
- Checking inductive invariants such as $TypeOK \land (termDetected \Rightarrow terminated)$
  - the two conditions can be checked by Apalache for prefixes of length 0 and 1
  - the invariant then holds throughout any execution
  - verification easily scales up to $N = 100$
  - quiescence property can be checked similarly
- Apalache is particularly useful for checking inductive invariants

Outline

Using TLAPS to Prove Safety Properties

- TLAPS: proof assistant for verifying TLA$^+$ specifications
  - not limited to fixed instances, unlike model checkers
  - relies on user interaction to guide verification

Using TLAPS to Prove Safety Properties

- TLAPS: proof assistant for verifying TLA$^+$ specifications
  - not limited to fixed instances, unlike model checkers
  - relies on user interaction to guide verification
- Proving a simple inductive invariant in TLAPS

THEOREM $TypeCorrect \triangleq Spec \Rightarrow \Box TypeOK$
$\langle 1 \rangle 1.\ Init \Rightarrow TypeOK$
$\langle 1 \rangle 2.\ TypeOK \wedge [Next]_{vars} \Rightarrow TypeOK'$
$\langle 1 \rangle 3.$ QED BY $\langle 1 \rangle 1, \langle 1 \rangle 2, PTL$ DEF $Spec$

  - hierarchical proof language represents proof tree
  - steps can be proved in any order: usually start with QED step
  - invariant follows from steps $\langle 1 \rangle 1$ and $\langle 1 \rangle 2$ by temporal logic (PTL)

Simple Proofs

- Prove that *Init* implies *TypeOK*

  $\langle 1 \rangle 1$. $Init \Rightarrow TypeOK$
  BY *NAssumption* DEFS *Init*, *TypeOK*, *Node*, *terminated*

  - relevant definitions and facts must be cited explicitly
  - this helps manage the size of the search space for proof tools

Simple Proofs

- Prove that *Init* implies *TypeOK*

  $\langle 1 \rangle 1$. $Init \Rightarrow TypeOK$
  BY *NAssumption* DEFS *Init*, *TypeOK*, *Node*, *terminated*

  - relevant definitions and facts must be cited explicitly
  - this helps manage the size of the search space for proof tools

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

  $\langle 1 \rangle 2$. $TypeOK \wedge [Next]_{vars} \Rightarrow TypeOK'$
  BY *NAssumption* DEFS *TypeOK*, *Next*, *vars*, *Terminate*, ...

  - decompose proof into smaller steps when brute force fails

Hierarchical Proofs

$\langle 1 \rangle 2.\ \mathit{TypeOK} \wedge [\mathit{Next}]_{\mathit{vars}} \Rightarrow \mathit{TypeOK}'$
  $\langle 2 \rangle$ SUFFICES ASSUME $\mathit{TypeOK}, [\mathit{Next}]_{\mathit{vars}}$
    PROVE TypeOK'
    OBVIOUS
  $\langle 2 \rangle$ USE *NAssumption* DEF *Node*, *TypeOK*
  $\langle 2 \rangle 1.$ CASE *DetectTermination*
    BY $\langle 2 \rangle 1$ DEF *DetectTermination*
  $\langle 2 \rangle 2.$ ASSUME NEW $i \in \mathit{Node}$, $\mathit{Terminate}(i)$
    PROVE TypeOK'
    BY $\langle 2 \rangle 2$ DEF *Terminate*, *terminated*
  ... similarly for the remaining actions ...
  $\langle 2 \rangle$ QED BY $\langle 2 \rangle 1, \langle 2 \rangle 2, \ldots$ DEF *Next*

- SUFFICES steps represent backward chaining

Proof of Main Safety Property

$TDCorrect \triangleq termDetected \Rightarrow terminated$

- Apalache suggested that *TDCorrect* is inductive relative to *TypeOK*

THEOREM $Safety \triangleq Spec \Rightarrow \Box TDCorrect$
$\langle 1 \rangle 1.\ Init \Rightarrow TDCorrect$
$\langle 1 \rangle 2.\ TypeOK \wedge TDCorrect \wedge [Next]_{vars} \Rightarrow TDCorrect'$
$\langle 1 \rangle 4.$ QED BY $\langle 1 \rangle 1$, $\langle 1 \rangle 2$, $\langle 1 \rangle 3$, *TypeCorrect*, *PTL* DEF *Spec*

  - proofs of steps $\langle 1 \rangle 1$ and $\langle 1 \rangle 2$ are similar as before

Proof of Main Safety Property

$TDCorrect \triangleq termDetected \Rightarrow terminated$

- Apalache suggested that *TDCorrect* is inductive relative to *TypeOK*

THEOREM $Safety \triangleq Spec \Rightarrow \Box TDCorrect$
$\langle 1 \rangle 1.\ Init \Rightarrow TDCorrect$
$\langle 1 \rangle 2.\ TypeOK \wedge TDCorrect \wedge [Next]_{vars} \Rightarrow TDCorrect'$
$\langle 1 \rangle 4.$ QED BY $\langle 1 \rangle 1$, $\langle 1 \rangle 2$, $\langle 1 \rangle 3$, *TypeCorrect*, *PTL* DEF *Spec*

  - proofs of steps $\langle 1 \rangle 1$ and $\langle 1 \rangle 2$ are similar as before

- Proof of quiescence is similar
  - proofs of safety properties require minimal temporal logic
  - automation of TLA$^+$ set theory is main concern

Liveness Proof (1)

- Liveness properties require fairness hypotheses
  - safety specification $Init \wedge \Box[Next]_{vars}$ allows for arbitrary stuttering steps
  - fairness conditions rule out infinite stuttering

$$\mathrm{WF}_{vars}(A) \triangleq \Box\big((\Box\mathrm{ENABLED}\ \langle A\rangle_{vars}) \Rightarrow \Diamond\langle A\rangle_{vars}\big)$$

  - an $\langle A\rangle_{vars}$ action must eventually occur if it remains enabled

Liveness Proof (1)

- Liveness properties require fairness hypotheses
  - safety specification $Init \wedge \Box[Next]_{vars}$ allows for arbitrary stuttering steps
  - fairness conditions rule out infinite stuttering

$$\mathrm{WF}_{vars}(A) \triangleq \Box\big((\Box\text{ENABLED } \langle A\rangle_{vars}) \Rightarrow \Diamond\langle A\rangle_{vars}\big)$$

  - an $\langle A\rangle_{vars}$ action must eventually occur if it remains enabled

- First establish enabledness conditions of actions

LEMMA *EnabledDT* $\triangleq$ ASSUME *TypeOK*
PROVE $(\text{ENABLED } \langle \mathit{SignalTerminated}\rangle_{vars}) \equiv \mathit{terminated} \wedge \neg \mathit{termDetected}$
BY *ExpandENABLED* DEF *TypeOK*, *SignalTerminated*, *terminated*, *vars*

  - proof method *ExpandENABLED* replaces ENABLED by its definition

Liveness Proof (2)

- Now prove the liveness property

THEOREM $Liveness \triangleq Spec \Rightarrow Liveness$
$\langle 1 \rangle$ DEFINE $P \triangleq terminated \wedge \neg termDetected \quad Q \triangleq termDetected$
$\langle 1 \rangle 1.\ TypeOK \wedge P \wedge [Next]_{vars} \Rightarrow P' \vee Q'$
$\langle 1 \rangle 2.\ TypeOK \wedge P \wedge \langle SignalTerminated \rangle_{vars} \Rightarrow Q'$
$\langle 1 \rangle 3.\ TypeOK \wedge P \Rightarrow$ ENABLED $\langle SignalTerminated \rangle_{vars}$
$\langle 1 \rangle 4.$ QED BY $\langle 1 \rangle 1, \langle 1 \rangle 2, \langle 1 \rangle 3, PTL$ DEF $Spec, Liveness$

  - steps $\langle 1 \rangle 1$ – $\langle 1 \rangle 3$ again only require action-level reasoning

Liveness Proof (2)

- Now prove the liveness property

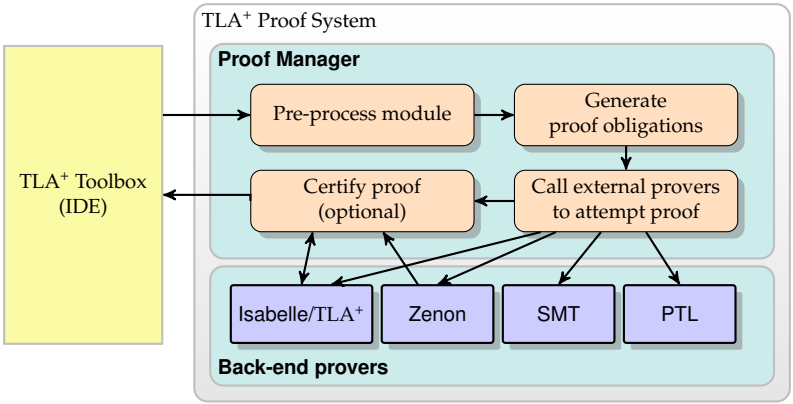
THEOREM $Liveness \triangleq Spec \Rightarrow Liveness$
$\langle 1 \rangle$ DEFINE $P \triangleq terminated \wedge \neg termDetected \quad Q \triangleq termDetected$
$\langle 1 \rangle 1.\ TypeOK \wedge P \wedge [Next]_{vars} \Rightarrow P' \vee Q'$
$\langle 1 \rangle 2.\ TypeOK \wedge P \wedge \langle SignalTerminated \rangle_{vars} \Rightarrow Q'$
$\langle 1 \rangle 3.\ TypeOK \wedge P \Rightarrow$ ENABLED $\langle SignalTerminated \rangle_{vars}$
$\langle 1 \rangle 4.$ QED BY $\langle 1 \rangle 1, \langle 1 \rangle 2, \langle 1 \rangle 3, PTL$ DEF $Spec, Liveness$

  - steps $\langle 1 \rangle 1$ – $\langle 1 \rangle 3$ again only require action-level reasoning

- Interactive theorem proving
  - applicable to arbitrary instances of specifications
  - requires human ingenuity for guiding the proof
  - use model checkers to weed out errors before embarking on proofs

Outline

TLAPS Architecture



- Isabelle/TLA$^+$: faithful encoding of TLA$^+$ in Isabelle's meta-logic
- Zenon: tableau prover for set theory
- PTL: decision procedure for propositional temporal logic

Encoding TLA$^+$: Boolean Expressions

- TLA$^+$ is untyped: no distinction between terms and formulas

$(42 = \text{TRUE}) \land \text{“abc”}$ denotes some (undetermined) value

Encoding TLA$^+$: Boolean Expressions

- TLA$^+$ is untyped: no distinction between terms and formulas

  $(42 = \text{TRUE}) \land \text{“abc”}$ denotes some (undetermined) value

- Two-valued semantics with underspecification
  - operators such as $=, \in, \land, \forall$ always evaluate to TRUE or FALSE
  - $(p = q) \Rightarrow (p \Leftrightarrow q)$ holds, but $(p \Leftrightarrow q) \Rightarrow (p = q)$ does not

Encoding TLA$^+$: Boolean Expressions

- TLA$^+$ is untyped: no distinction between terms and formulas

  $(42 = \text{TRUE}) \land \text{“abc”}$ denotes some (undetermined) value

- Two-valued semantics with underspecification
  - operators such as $=, \in, \land, \forall$ always evaluate to TRUE or FALSE
  - $(p = q) \Rightarrow (p \Leftrightarrow q)$ holds, but $(p \Leftrightarrow q) \Rightarrow (p = q)$ does not

- Most standard laws of logic remain true

$$
\begin{array}{lll}
(\neg P) = (P \Rightarrow \text{FALSE}) & \neg(P \land Q) = (\neg P \lor \neg Q) & \mathit{boolify}(P) \triangleq P = \text{TRUE} \\
(P \land \text{TRUE}) = \mathit{boolify}(P) & \mathit{boolify}(x = y) = (x = y) & \mathit{boolify}(Q \land R) = (Q \land R) \\
P(t) \Rightarrow (\exists x : P(x)) & (\forall x : P(x)) \Rightarrow P(x) &
\end{array}
$$

Encoding Set Theory

- Reduce set-theoretic constructions to first-order logic
  - define set-theoretic operators in terms of uninterpreted binary predicate $\in$

$$
\begin{aligned}
& e \in (S \cup T) \;\equiv\; (e \in S) \vee (e \in T) \\
& S \subseteq T \;\equiv\; \forall x \in S : x \in T \\
& e \in \{\, x \in S : P(x) \,\} \;\equiv\; (e \in S) \wedge P(e) \\
& e \in \{\, f(x) : x \in S \,\} \;\equiv\; \exists x \in S : e = f(x)
\end{aligned}
$$

Encoding Set Theory

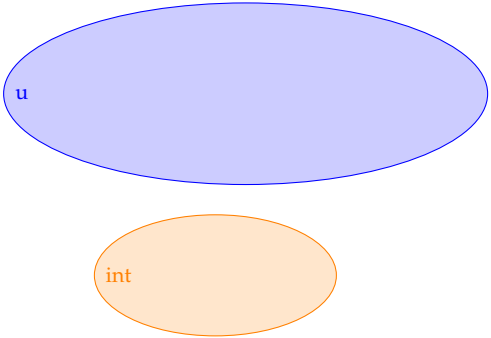
- Reduce set-theoretic constructions to first-order logic
  - define set-theoretic operators in terms of uninterpreted binary predicate $\in$

$$
\begin{aligned}
& e \in (S \cup T) \;\equiv\; (e \in S) \vee (e \in T) \\
& S \subseteq T \;\equiv\; \forall x \in S : x \in T \\
& e \in \{\, x \in S : P(x) \,\} \;\equiv\; (e \in S) \wedge P(e) \\
& e \in \{\, f(x) : x \in S \,\} \;\equiv\; \exists x \in S : e = f(x)
\end{aligned}
$$

- Two implementations of this encoding
  - first implementation relies on extensive rewriting
  - second implementation uses axioms with well-chosen triggers
  - no significant performance difference, but rewriting is brittle
- TLA$^+$ functions encoded in a similar way

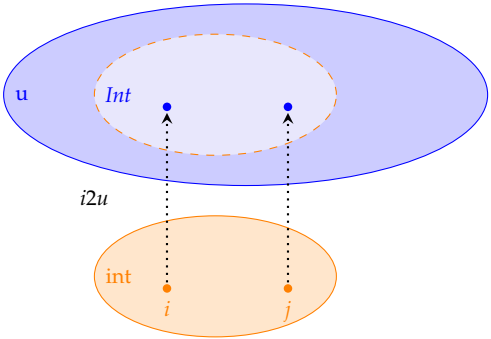
Untyped Logic: Theory Reasoning

- SMT solvers provide automation for interpreted theories
- Untyped embedding: inject interpreted sorts into TLA$^+$ universe



Untyped Logic: Theory Reasoning

- SMT solvers provide automation for interpreted theories
- Untyped embedding: inject interpreted sorts into TLA$^+$ universe



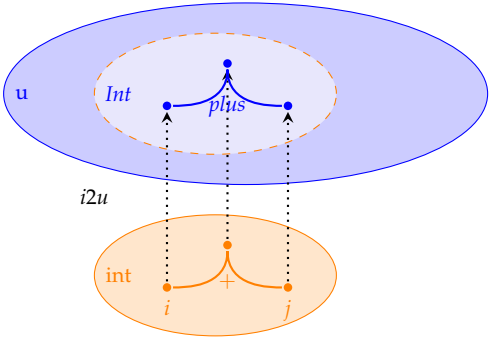
Characteristic axioms

$\forall i, j : i2u(i) = i2u(j) \Rightarrow i = j$

$\forall u : u \in Int \equiv \exists i : u = i2u(i)$

Untyped Logic: Theory Reasoning

- SMT solvers provide automation for interpreted theories
- Untyped embedding: inject interpreted sorts into TLA$^+$ universe



Characteristic axioms

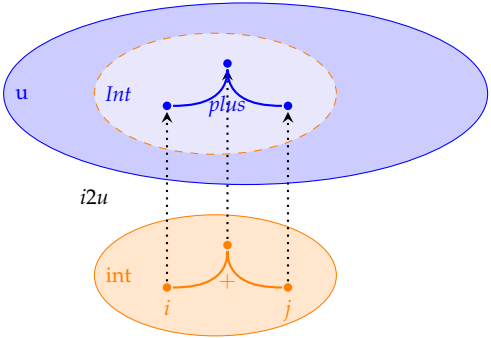
$\forall i, j : i2u(i) = i2u(j) \Rightarrow i = j$

$\forall u : u \in Int \equiv \exists i : u = i2u(i)$

$\forall i, j : plus(i2u(i), i2u(j)) = i2u(i + j)$

Untyped Logic: Theory Reasoning

- SMT solvers provide automation for interpreted theories
- Untyped embedding: inject interpreted sorts into TLA$^+$ universe



Characteristic axioms

$\forall i, j : i2u(i) = i2u(j) \Rightarrow i = j$

$\forall u : u \in Int \equiv \exists i : u = i2u(i)$

$\forall i, j : plus(i2u(i), i2u(j)) = i2u(i + j)$

- Use of triggers again makes this work in practice

Support for Temporal Logic Reasoning

- Temporal logic breaks natural deduction
  - $F \vdash G$ cannot be identified with $\vdash F \Rightarrow G$
  - for example, have $F \vdash \Box F$ but not $\vdash F \Rightarrow \Box F$

Support for Temporal Logic Reasoning

- Temporal logic breaks natural deduction
  - $F \vdash G$ cannot be identified with $\vdash F \Rightarrow G$
  - for example, have $F \vdash \Box F$ but not $\vdash F \Rightarrow \Box F$
  - But: $\Box F \vdash G$ can be identified with $\vdash \Box F \Rightarrow G$

Support for Temporal Logic Reasoning

- Temporal logic breaks natural deduction
  - $F \vdash G$ cannot be identified with $\vdash F \Rightarrow G$
  - for example, have $F \vdash \Box F$ but not $\vdash F \Rightarrow \Box F$
  - But: $\Box F \vdash G$ can be identified with $\vdash \Box F \Rightarrow G$
- Arrange temporal reasoning so that all hypotheses are boxed
  - formula F is boxed if $F \Leftrightarrow \Box F$
  - syntactic approximation: constant formulas, $\Box F$, $\Diamond \Box F$, $\mathrm{WF}_v(A)$, conjunction, ...
  - implicit generalization of formulas derived in boxed context
  - requires disciplined, but quite natural use of temporal reasoning

Temporal Logic Reasoning in Practice

- Quickly reduce temporal conclusions to action-level hypotheses

$$\frac{P \wedge [N]_v \Rightarrow P' \vee Q' \quad P \wedge \langle A \rangle_v \Rightarrow Q' \quad P \Rightarrow \text{ENABLED } \langle A \rangle_v}{\Box[N]_v \wedge \text{WF}_v(A) \;\Rightarrow\; \Box(P \Rightarrow \Diamond Q)}$$

- Temporal reasoning is mostly propositional
  - use PTL decision procedure rather than hard-wired rules

Temporal Logic Reasoning in Practice

- Quickly reduce temporal conclusions to action-level hypotheses

$$\frac{P \wedge [N]_v \Rightarrow P' \vee Q' \quad P \wedge \langle A \rangle_v \Rightarrow Q' \quad P \Rightarrow \text{ENABLED } \langle A \rangle_v}{\Box[N]_v \wedge \text{WF}_v(A) \;\Rightarrow\; \Box(P \Rightarrow \Diamond Q)}$$

- Temporal reasoning is mostly propositional
  - use PTL decision procedure rather than hard-wired rules
- Support for first-order temporal reasoning by on-the-fly abstraction
  - hide operators that are not part of PTL, and vice versa
  - during pre-processing, commute $\forall$ and $\Box$ as well as $\exists$ and $\Diamond$

First-Order Temporal Reasoning

THEOREM $Init \wedge \Box[Next]_v \Rightarrow \forall p \in Proc : \Box Safe(p)$
$\langle 1 \rangle$. SUFFICES ASSUME NEW $p \in Proc$
PROVE $Init \wedge \Box[Next]_v \Rightarrow \Box Safe(p)$
OBVIOUS
$\langle 1 \rangle 1$. $Init \Rightarrow Safe(p)$ BY DEF $Init$, $Safe$, ...
$\langle 1 \rangle 2$. $Safe(p) \wedge [Next]_v \Rightarrow Safe(p)'$ BY DEF $Safe$, $Next$, v, ...
$\langle 1 \rangle 3$. QED BY $\langle 1 \rangle 1$, $\langle 1 \rangle 2$, PTL

- Mix of first-order and temporal reasoning
  - first-order provers vs. PTL decision procedure
  - prime "modality" handled by pre-processing at action level
- What is really going on here?

Coalescing: Basic Idea

- Abstract subformulas from different sublogic
  - in the SUFFICES step, the FOL prover sees the proof obligation

$$\frac{p \in \mathit{Proc} \qquad \mathit{Init} \wedge \boxed{\Box[\mathit{Next}]_v} \Rightarrow \boxed{\Box \mathit{Safe}}(p)}{\mathit{Init} \wedge \boxed{\Box[\mathit{Next}]_v} \Rightarrow \forall p \in \mathit{Proc} : \boxed{\Box \mathit{Safe}}(p)}$$

Coalescing: Basic Idea

- Abstract subformulas from different sublogic
  - in the SUFFICES step, the FOL prover sees the proof obligation

$$\frac{p \in Proc \qquad Init \wedge \boxed{\Box[Next]_v} \Rightarrow \boxed{\Box Safe}(p)}{Init \wedge \boxed{\Box[Next]_v} \Rightarrow \forall p \in Proc : \boxed{\Box Safe}(p)}$$

  - in the QED step, the PTL decision procedure sees

$$\frac{\boxed{Init} \Rightarrow \boxed{Safe(p)} \qquad \boxed{Safe(p)} \wedge \boxed{[Next]_v} \Rightarrow \circ \boxed{Safe(p)}}{\boxed{Init} \wedge \Box \boxed{[Next]_v} \Rightarrow \Box \boxed{Safe(p)}}$$

  - the formulas in boxes are introduced as ad-hoc operators
- Sound combination of temporal and first-order reasoning

Part IV

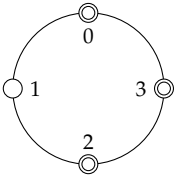
Refinement of Specifications

Objectives

- Compare specifications written at different levels of abstraction
- TLA$^+$ constructs used for refining specifications
  - quantification over state variables
  - refinement mappings and module instantiation
- Illustration: an algorithm for distributed termination detection

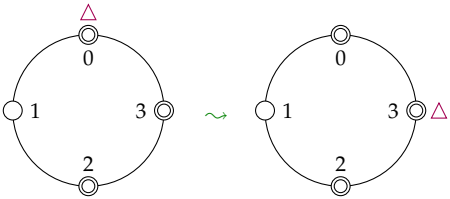
Outline

Initial Idea: Token Passing



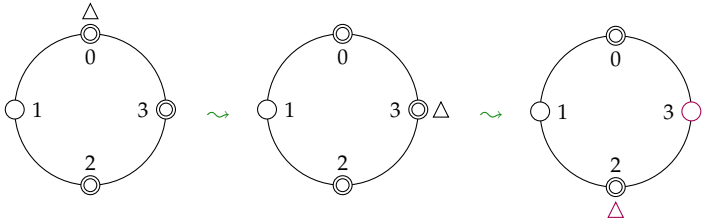
- Remember the problem of termination detection
  - nodes can be active (double circle) or inactive (simple circle)
  - "master node" 0 in charge of detecting when all nodes are inactive

Initial Idea: Token Passing



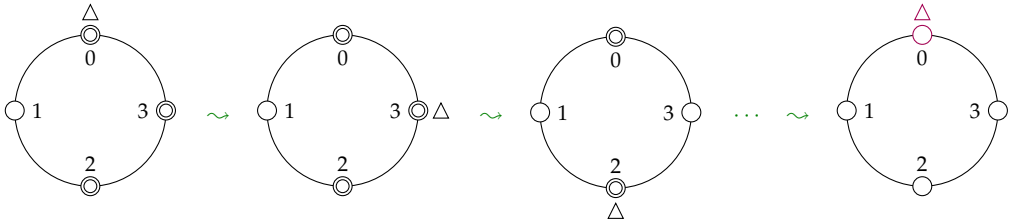
- Remember the problem of termination detection
  - nodes can be active (double circle) or inactive (simple circle)
  - "master node" 0 in charge of detecting when all nodes are inactive
- Token-based algorithm
  - initially: token at master node, who may pass it to its neighbor

Initial Idea: Token Passing



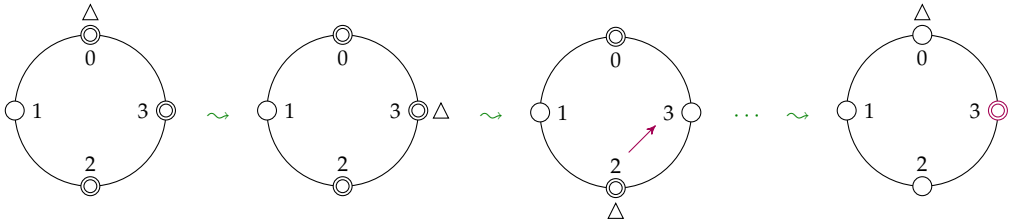
- Remember the problem of termination detection
  - nodes can be active (double circle) or inactive (simple circle)
  - "master node" 0 in charge of detecting when all nodes are inactive
- Token-based algorithm
  - initially: token at master node, who may pass it to its neighbor
  - when (non-master) node is inactive, the token moves to its neighbor

Initial Idea: Token Passing



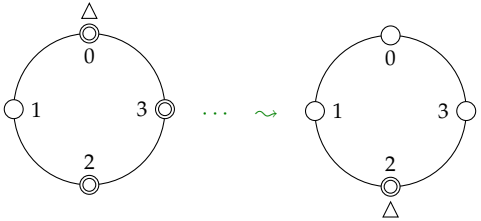
- Remember the problem of termination detection
  - nodes can be active (double circle) or inactive (simple circle)
  - "master node" 0 in charge of detecting when all nodes are inactive
- Token-based algorithm
  - initially: token at master node, who may pass it to its neighbor
  - when (non-master) node is inactive, the token moves to its neighbor
  - termination detected when token returns to inactive master node

Initial Idea: Token Passing



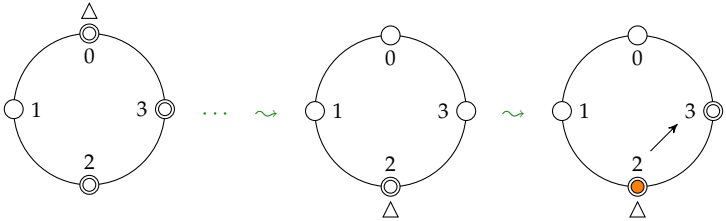
- Remember the problem of termination detection
  - nodes can be active (double circle) or inactive (simple circle)
  - "master node" 0 in charge of detecting when all nodes are inactive
- Token-based algorithm
  - initially: token at master node, who may pass it to its neighbor
  - when (non-master) node is inactive, the token moves to its neighbor
  - termination detected when token returns to inactive master node
- Complication: nodes may send messages, activating receiver

Dijkstra's Algorithm (EWD 840, 1983)



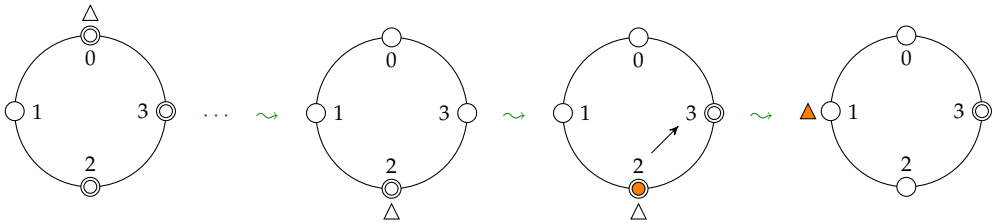
- Nodes and token colored orange ("dirty") or white ("clean")
  - master node initiates probe by sending white token

Dijkstra's Algorithm (EWD 840, 1983)



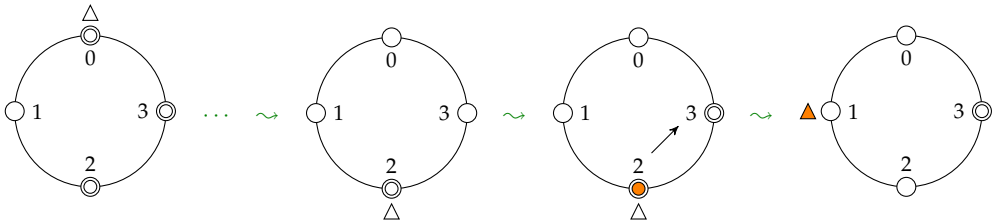
- Nodes and token colored orange ("dirty") or white ("clean")
  - master node initiates probe by sending white token
  - node becomes orange when sending a message to a higher-numbered node

Dijkstra's Algorithm (EWD 840, 1983)



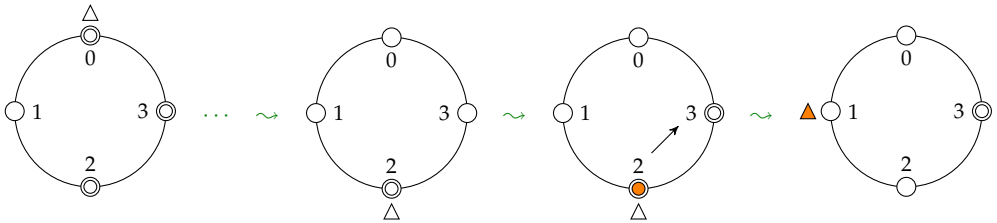
- Nodes and token colored orange ("dirty") or white ("clean")
  - master node initiates probe by sending white token
  - node becomes orange when sending a message to a higher-numbered node
  - when passing the token, an orange node transfers the stain to the token

Dijkstra's Algorithm (EWD 840, 1983)



- Nodes and token colored orange ("dirty") or white ("clean")
  - master node initiates probe by sending white token
  - node becomes orange when sending a message to a higher-numbered node
  - when passing the token, an orange node transfers the stain to the token
- Master node detects termination when it is inactive, white, and holds white token

Dijkstra's Algorithm (EWD 840, 1983)



- Nodes and token colored orange ("dirty") or white ("clean")
  - master node initiates probe by sending white token
  - node becomes orange when sending a message to a higher-numbered node
  - when passing the token, an orange node transfers the stain to the token
- Master node detects termination when it is inactive, white, and holds white token
- Safety: termination detected only if all nodes are inactive
- Liveness: when all nodes inactive, termination will eventually be detected

TLA$^+$ Specification of EWD 840: System Configurations

MODULE *EWD840*

EXTENDS *Naturals*
CONSTANT N
ASSUME $NAssumption \triangleq N \in Nat \setminus \{0\}$
$Nodes \triangleq 0..N-1$
$Color \triangleq \{\text{“white”}, \text{“orange”}\}$
VARIABLES $tpos, tcolor, active, color$
$TypeOK \triangleq \land tpos \in Nodes \land tcolor \in Color$
$\land active \in [Nodes \to \text{BOOLEAN}] \land color \in [Nodes \to Color]$
$terminated \triangleq \forall n \in Nodes : active[n] = \text{FALSE}$

- Similar to the description of termination detection
  - nodes: represent activation status and color
  - token: current position and color

TLA$^+$ Specification of EWD 840: Initiation and System Transitions

$$Init \triangleq \land tpos \in Nodes \land tcolor = \text{“orange”}$$
$$\land active \in [Nodes \to \text{BOOLEAN}] \land color \in [Nodes \to Color]$$

- Initial condition: any “type-correct” values; token is initially orange

TLA$^+$ Specification of EWD 840: Initiation and System Transitions

$$
\begin{aligned}
&\mathit{Init} \triangleq \land\ \mathit{tpos} \in \mathit{Nodes} \land \mathit{tcolor} = \text{“orange”} \\
&\qquad\ \land\ \mathit{active} \in [\mathit{Nodes} \to \text{BOOLEAN}] \land \mathit{color} \in [\mathit{Nodes} \to \mathit{Color}] \\
&\mathit{InitiateProbe} \triangleq \\
&\quad \land\ \mathit{tpos} = 0 \land (\mathit{tcolor} = \text{“orange”} \lor \mathit{color}[0] = \text{“orange”}) \\
&\quad \land\ \mathit{tpos}' = N - 1 \land \mathit{tcolor}' = \text{“white”} \\
&\quad \land\ \mathit{color}' = [\mathit{color} \text{ EXCEPT } ![0] = \text{“white”}] \\
&\quad \land\ \mathit{active}' = \mathit{active} \\
&\mathit{PassToken}(i) \triangleq \\
&\quad \land\ \mathit{tpos} = i \land (\neg \mathit{active}[i] \lor \mathit{color}[i] = \text{“orange”} \lor \mathit{tcolor} = \text{“orange”}) \\
&\quad \land\ \mathit{tpos}' = i - 1 \\
&\quad \land\ \mathit{tcolor}' = \text{IF } \mathit{color}[i] = \text{“orange” THEN “orange” ELSE } \mathit{tcolor} \\
&\quad \land\ \mathit{color}' = [\mathit{color} \text{ EXCEPT } ![i] = \text{“white”}] \\
&\quad \land\ \mathit{active}' = \mathit{active} \\
&\mathit{System} \triangleq \mathit{InitiateProbe} \lor \exists i \in \mathit{Nodes} \setminus \{0\} : \mathit{PassToken}(i)
\end{aligned}
$$

- Initial condition: any “type-correct” values; token is initially orange
- Transitions of detection algorithm: token passing

TLA$^+$ Specification of EWD 840: Environment Transitions

$$
\begin{aligned}
&Message(i,j) \triangleq \\
&\quad \land\ active[i] \\
&\quad \land\ active' = [active \text{ EXCEPT } ![j] = \text{TRUE}] \\
&\quad \land\ color' = [color \text{ EXCEPT } ![i] = \text{IF } j > i \text{ THEN “orange” ELSE } @] \\
&\quad \land\ \text{UNCHANGED } \langle tpos, tcolor \rangle \\
&Terminate(i) \triangleq \\
&\quad \land\ active[i] \land active' = [active \text{ EXCEPT } ![i] = \text{FALSE}] \\
&\quad \land\ \text{UNCHANGED } \langle color, tpos, tcolor \rangle \\
&Env \triangleq \exists i, j \in Nodes : Message(i,j) \lor Terminate(i)
\end{aligned}
$$

- Definition of actions not controlled by the algorithm

TLA$^+$ Specification of EWD 840: Environment Transitions

$$
\begin{aligned}
&Message(i,j) \triangleq \\
&\quad \land\ active[i] \\
&\quad \land\ active' = [active \text{ EXCEPT } ![j] = \text{TRUE}] \\
&\quad \land\ color' = [color \text{ EXCEPT } ![i] = \text{IF } j > i \text{ THEN “orange” ELSE } @] \\
&\quad \land\ \text{UNCHANGED } \langle tpos, tcolor \rangle \\
&Terminate(i) \triangleq \\
&\quad \land\ active[i] \land active' = [active \text{ EXCEPT } ![i] = \text{FALSE}] \\
&\quad \land\ \text{UNCHANGED } \langle color, tpos, tcolor \rangle \\
&Env \triangleq \exists i, j \in Nodes : Message(i,j) \lor Terminate(i) \\
&Next \triangleq System \lor Env \\
&vars \triangleq \langle tpos, tcolor, active, color \rangle \\
&Spec \triangleq Init \land \Box[Next]_{vars}
\end{aligned}
$$

- Definition of actions not controlled by the algorithm
- Standard form of overall specification

Outline

High-Level Safety Properties of EWD 840

- Type correctness

THEOREM $Spec \Rightarrow \Box TypeOK$

High-Level Safety Properties of EWD 840

- Type correctness

  THEOREM $Spec \Rightarrow \Box TypeOK$

- All nodes are inactive when master node detects termination
  - master signals termination when it is white and inactive and holds a white token

$terminationDetected \triangleq tpos = 0 \land tcolor = \text{“white”} \land color[0] = \text{“white”} \land active[0] = \text{FALSE}$

$TDCorrect \triangleq terminationDetected \Rightarrow terminated$

THEOREM $Spec \Rightarrow \Box TDCorrect$

High-Level Safety Properties of EWD 840

- Type correctness — THEOREM $Spec \Rightarrow \Box TypeOK$
- All nodes are inactive when master node detects termination
  - master signals termination when it is white and inactive and holds a white token

$terminationDetected \triangleq tpos = 0 \land tcolor = \text{“white”} \land color[0] = \text{“white”} \land active[0] = \text{FALSE}$
$TDCorrect \triangleq terminationDetected \Rightarrow terminated$
THEOREM $Spec \Rightarrow \Box TDCorrect$

- These properties can be verified using the familiar TLA$^+$ tools
  - explicit-state model checking using TLC
  - bounded symbolic model checking using Apalache
  - interactive proof using TLAPS requires an inductive invariant

Dijkstra's Inductive Invariant

- Dijkstra provides the following inductive invariant

$$
\begin{array}{lll}
Inv \triangleq & \lor \forall i \in Nodes : i > tpos \Rightarrow \neg active[i] & \text{all nodes behind the token are inactive} \\
& \lor \exists j \in Node : j \in 0 \,..\, tpos \land color[j] = \text{“orange”} & \text{some node ahead is dirty} \\
& \lor tcolor = \text{“orange”} & \text{the token is dirty}
\end{array}
$$

Dijkstra's Inductive Invariant

- Dijkstra provides the following inductive invariant

$$
\begin{array}{lll}
Inv \triangleq & \lor \forall i \in Nodes : i > tpos \Rightarrow \neg active[i] & \text{all nodes behind the token are inactive} \\
& \lor \exists j \in Node : j \in 0 \,..\, tpos \land color[j] = \text{“orange”} & \text{some node ahead is dirty} \\
& \lor tcolor = \text{“orange”} & \text{the token is dirty}
\end{array}
$$

  - Apalache confirms that this invariant is inductive relative to type correctness
  - routine proof using TLAPS

Dijkstra's Inductive Invariant

- Dijkstra provides the following inductive invariant

$$
\begin{array}{lll}
Inv \triangleq & \lor \forall i \in Nodes : i > tpos \Rightarrow \neg active[i] & \text{all nodes behind the token are inactive} \\
& \lor \exists j \in Node : j \in 0 \,..\, tpos \land color[j] = \text{“orange”} & \text{some node ahead is dirty} \\
& \lor tcolor = \text{“orange”} & \text{the token is dirty}
\end{array}
$$

  - Apalache confirms that this invariant is inductive relative to type correctness
  - routine proof using TLAPS

- Main correctness theorem is now a simple corollary
  - Dijkstra's invariant implies correctness of termination detection
  - TLAPS proves $Inv \Rightarrow TDCorrect$

Outline

Towards Liveness of the Algorithm

- Global termination should eventually be detected

  $Liveness \triangleq terminated \leadsto terminationDetected$

  - same formula as the one used for the termination detection problem

Towards Liveness of the Algorithm

- Global termination should eventually be detected

$$Liveness \triangleq terminated \rightsquigarrow terminationDetected$$

  - same formula as the one used for the termination detection problem

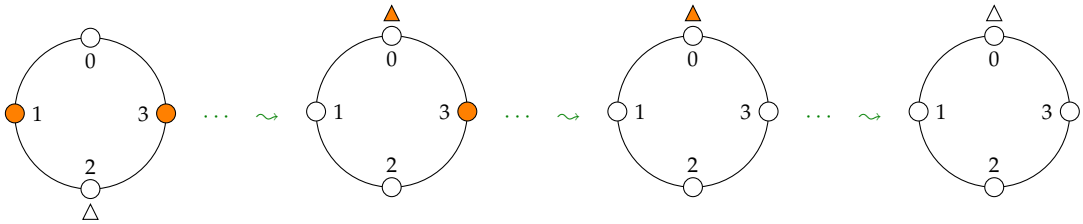
- Need appropriate fairness condition to ensure liveness
  - require that token-passing transitions eventually occur

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

  - can you explain why $\mathrm{WF}_{vars}(Next)$ would be too strong?
  - TLC confirms that liveness holds for small configurations

Idea of the Liveness Proof

- Detecting termination may require three rounds of the token
  - the first round brings the token back to node 0 (and all nodes remain inactive)
  - the second round removes any stain from nodes
  - the third round confirms that all nodes are clean and brings back a clean token



Formal Proof of Liveness

- Formally, set up a proof by contradiction

$$TSpec \triangleq \Box TypeOK \wedge \Box Inv \wedge \Box(\neg terminationDetected) \wedge \Box[Next]_{vars} \wedge \mathrm{WF}_{vars}(System)$$

  - since termination may occur in any reachable state, do not include initial condition but assume invariants
  - in view of a contradiction, assume that termination is never detected

- Prove the following three lemmas

$$allWhite \triangleq \forall n \in Node : color[n] = \text{“white”}$$
$$\textsc{lemma}\ Round1 \triangleq TSpec \Rightarrow (terminated \leadsto (terminated \wedge tpos = 0))$$
$$\textsc{lemma}\ Round2 \triangleq TSpec \Rightarrow (terminated \wedge tpos = 0 \leadsto terminated \wedge tpos = 0 \wedge allWhite)$$
$$\textsc{lemma}\ Round3 \triangleq TSpec \Rightarrow (terminated \wedge tpos = 0 \wedge allWhite \leadsto terminated \wedge tpos = 0 \wedge allWhite \wedge tcolor = \text{“white”})$$

Outline of the Proofs of the Lemmas

- Set up an induction on the current position of the token

LEMMA $Round1 \triangleq TSpec \Rightarrow (terminated \leadsto (terminated \land tpos = 0))$
$\langle 1 \rangle$. DEFINE $P(n) \triangleq terminated \land n \in Node \land tpos = n$
$Q \triangleq P(0)$
$R(n) \triangleq TSpec \Rightarrow \Box(P(n) \Rightarrow \Diamond Q)$
$\langle 1 \rangle 1$. $\forall n \in Nat : R(n)$
$\langle 2 \rangle 1$. $R(0)$ * trivial
$\langle 2 \rangle 2$. ASSUME NEW $n \in Nat$ PROVE $R(n) \Rightarrow R(n+1)$
$\langle 2 \rangle$. QED BY $\langle 2 \rangle 1$, $\langle 2 \rangle 2$, *NatInduction*
$\langle 1 \rangle 2$. $TSpec \Rightarrow \Box((\exists n \in Nat : P(n)) \Rightarrow \Diamond Q)$ * consequence of $\langle 1 \rangle 1$
$\langle 1 \rangle 3$. $TypeOK \Rightarrow (terminated \Rightarrow \exists n \in Nat : P(n))$ * trivial
$\langle 1 \rangle$. QED BY $\langle 1 \rangle 2$, $\langle 1 \rangle 3$, *PTL* DEF *TSpec*

- The proofs of the other lemmas are very similar

Finishing the Liveness Proof

- We now have all ingredients for proving liveness

THEOREM $Live \triangleq \Box TypeOK \wedge \Box Inv \wedge \Box[Next]_{vars} \wedge \mathrm{WF}_{vars}(System) \Rightarrow Liveness$
$\langle 1 \rangle$. $terminated \wedge tpos = 0 \wedge allWhite \wedge tcolor =$ "white" $\Rightarrow terminationDetected$
$\langle 1 \rangle$. QED * conclude by contradiction
BY $Round1$, $Round2$, $Round3$, PTL DEF $TSpec$, $Liveness$

THEOREM $SpecLive \triangleq Spec \Rightarrow Liveness$
BY $Live$, $TypeCorrect$, $Invariant$, PTL DEF $Spec$

- Proofs of liveness properties require more effort
  - less stylized and less automated than safety proofs
  - non-trivial interaction between first-order and temporal logic reasoning

Outline

Reflecting on the Proof of EWD 840

- We proved the same properties for EWD 840 and for the specification of the problem
  - could we have done the proof differently?
  - can we formally relate the two specifications?

Reflecting on the Proof of EWD 840

- We proved the same properties for EWD 840 and for the specification of the problem
  - could we have done the proof differently?
  - can we formally relate the two specifications?

- In TLA$^+$, there is no formal distinction between a specification and its properties
  - both are represented by temporal formulas
  - every execution of EWD 840 also satisfies the specification of termination detection

THEOREM $Refinement \triangleq EWD840!Spec \Rightarrow TerminationDetection!Spec$ * (approximately)

  - stuttering invariance is essential here: token passing is invisible at the abstract level

Information Hiding

- TLA$^+$ specifications describe state machines
  - contain “implementation details” necessary for describing state transitions
  - example: program counter generated by PlusCal translator
  - these “details” are irrelevant when implementing the specification
  - the variables representing these details should be hidden from the interface

Information Hiding

- TLA$^+$ specifications describe state machines
  - contain "implementation details" necessary for describing state transitions
  - example: program counter generated by PlusCal translator
  - these "details" are irrelevant when implementing the specification
  - the variables representing these details should be hidden from the interface

- In logic, hiding corresponds to (existential) quantification

$$\begin{aligned} \mathit{Inner} &\triangleq \mathit{Init} \land \Box[\mathit{Next}]_v \land L \\ \mathit{Spec} &\triangleq \exists\, x : \mathit{Inner} \end{aligned}$$

  - *Spec* describes the same system as *Inner*, but with variables x hidden

Semantics of Quantification over Variables

- $\exists$ quantifies over state variables, not values
  - naive semantics: $\sigma, \xi \models \exists x : F$ iff $\tau, \xi \models F$ for some $\tau =_x \sigma$
    where $\tau =_x \sigma$ means $\tau_i(y) = \sigma_i(y)$ for all variables except x
  - this definition does not preserve stuttering invariance

    $$\exists x : x = 0 \land y = 0 \land \Box[(x = 0 \land x' = 1 \land y' = y) \lor (x = 1 \land y' = y + 1 \land x' = x)]_{\langle x,y \rangle}$$

  - real semantics: $\sigma, \xi \models \exists x : F$ iff $\tau, \xi \models F$ for some $\tau \approx_x \sigma$
    where $\tau \approx_x \sigma$ means $\tau \approx \tau' =_x \sigma' \approx \sigma$ for some σ', τ'

Semantics of Quantification over Variables

- $\exists$ quantifies over state variables, not values
  - naive semantics: $\sigma, \xi \models \exists x : F$ iff $\tau, \xi \models F$ for some $\tau =_x \sigma$
    where $\tau =_x \sigma$ means $\tau_i(y) = \sigma_i(y)$ for all variables except x
  - this definition does not preserve stuttering invariance

    $$\exists x : x = 0 \land y = 0 \land \Box[(x = 0 \land x' = 1 \land y' = y) \lor (x = 1 \land y' = y + 1 \land x' = x)]_{\langle x,y \rangle}$$

  - real semantics: $\sigma, \xi \models \exists x : F$ iff $\tau, \xi \models F$ for some $\tau \approx_x \sigma$
    where $\tau \approx_x \sigma$ means $\tau \approx \tau' =_x \sigma' \approx \sigma$ for some σ', τ'
- The universal quantifier $\forall$ is dual to $\exists$
- The standard quantifier rules hold for $\forall$ and $\exists$

Refinement Mappings

- Proving refinement requires reasoning about $\exists$

  $(\exists\, x : \mathit{Impl}) \Rightarrow (\exists\, y : \mathit{Spec})$

  - using standard quantifier rules, this reduces to proving $\mathit{Impl} \Rightarrow \exists\, y : \mathit{Spec}$
  - existential quantifiers are usually introduced using "witness terms" $\mathit{Impl} \Rightarrow \mathit{Spec}[t/y]$

Refinement Mappings

- Proving refinement requires reasoning about $\exists$

  $(\exists\, x : \mathit{Impl}) \Rightarrow (\exists\, y : \mathit{Spec})$

  - using standard quantifier rules, this reduces to proving $\mathit{Impl} \Rightarrow \exists\, y : \mathit{Spec}$
  - existential quantifiers are usually introduced using "witness terms" $\mathit{Impl} \Rightarrow \mathit{Spec}[t/y]$

- In this context, the witness (state function) is called a refinement mapping
  - compute the high-level state from the state of the low-level specification
  - must define a suitable refinement mapping when proving refinement

Refinement Mappings in TLA$^+$

- TLC, Apalache, and TLAPS do not support reasoning about $\exists$
- Use module instantiation for defining refinement mappings

MODULE *Concrete*

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

$Abs \triangleq$ INSTANCE *Abstract* WITH $y \leftarrow \ldots$ * *defines refinement mapping*

THEOREM *Refinement* $\triangleq Spec \Rightarrow Abs!Spec$

  - *Abs*!*Spec* denotes the formula *Spec* defined in module *Abstract* after substitution

Refinement Mappings in TLA$^+$

- TLC, Apalache, and TLAPS do not support reasoning about $\exists$
- Use module instantiation for defining refinement mappings

MODULE *Concrete*

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

$Abs \triangleq$ INSTANCE *Abstract* WITH $y \leftarrow \ldots$ * *defines refinement mapping*

THEOREM *Refinement* $\triangleq Spec \Rightarrow Abs!Spec$

  - $Abs!Spec$ denotes the formula $Spec$ defined in module *Abstract* after substitution

- Concrete example: EWD 840

$TD \triangleq$ INSTANCE *TerminationDetection* WITH *termDetected* $\leftarrow$ *terminationDetected*

THEOREM *Refinement* $\triangleq Spec \Rightarrow TD!Spec$

Proving Refinement for EWD 840

- We must prove three facts
  - refinement of initialization $Init \Rightarrow TD!Init$
  - refinement of transitions $[Next]_{vars} \Rightarrow [TD!Next]_{TD!vars}$
  - preservation of fairness $Spec \Rightarrow \mathrm{WF}_{TD!vars}(TD!SignalTerminated)$

Proving Refinement for EWD 840

- We must prove three facts
  - refinement of initialization $Init \Rightarrow TD!Init$
  - refinement of transitions $[Next]_{vars} \Rightarrow [TD!Next]_{TD!vars}$
  - preservation of fairness $Spec \Rightarrow \mathrm{WF}_{TD!vars}(TD!SignalTerminated)$
- How can we prove this?
  - proof of correct initialization is immediate
  - show that every action of EWD 840 corresponds to a high-level step or stuttering $\Rightarrow$ in fact, use the invariants of EWD 840 in this proof
  - show that $TD!SignalTerminated$ cannot remain enabled forever in a run of EWD 840
  - by the liveness property, we cannot have $\Box(terminated \wedge \neg terminationDetected)$

Beyond Refinement Mappings

- Refinement mappings do not always exist
  - constructing a witness may require referring to past or future states
  - use specific rules to introduce auxiliary information
  - then construct a refinement mapping from the enriched state

- Auxiliary variables
  - history variables record information from past states
    $\Rightarrow$ cf. ghost variables in program verification
  - prophecy variables predict future states
  - stuttering variables enforce insertion of stuttering transitions

Lamport, Merz: Prophecy Made Simple. ACM TOPLAS 44(2), 6:1-6:27, 2022.

Part V

Conclusion

Summing Up

- TLA$^+$: mathematical language for specifying systems
  - highly expressive and flexible language encourages abstract descriptions
  - state machine specifications represent system behavior
  - no distinction between systems and properties
  - refinement (and composition) reflected in logic
- Support tools
  - IDEs: VS Code Extension for TLA$^+$ / TLA$^+$ Toolbox
  - TLC: explicit-state model checker, checkpointing, parallelization
  - Apalache: bounded model checking based on SMT encoding
  - TLAPS: interactive proof platform, automatic proof back-ends
  - PlusCal: front-end for generating TLA$^+$ from "pseudo code" language
- More information

  `http://lamport.azurewebsites.net/tla/tla.html` Google discussion group