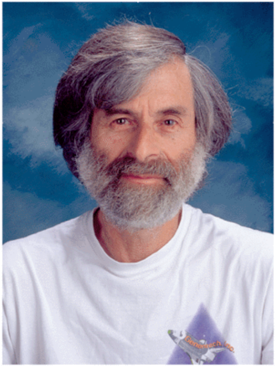
Specifying and Verifying Systems in TLA$^+$

Stephan Merz

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

INRIA Nancy & LORIA
Nancy, France

ICTAC School on Software Engineering
Natal, Brazil, August 2010



PhD 1972 (Brandeis University), Mathematics

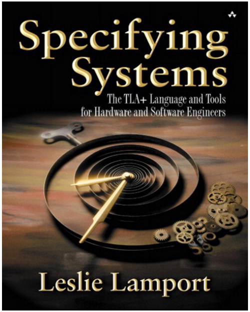
- Mitre Corporation, 1962–65
- Marlboro College, 1965–69
- Massachusets Computer Associates, 1970–77
- SRI International, 1977–85
- Digital Equipment Corporation/Compaq, 1985–2001
- Microsoft Research, since 2001

Pioneer of distributed algorithms

- Natl. Academy of Engineering, PODC Influential Paper Award, ACM SIGOPS Hall of Fame, LICS Award, IEEE John v. Neumann medal, . . .
- honorary doctorates (Rennes, Kiel, Lausanne, Lugano, Nancy)

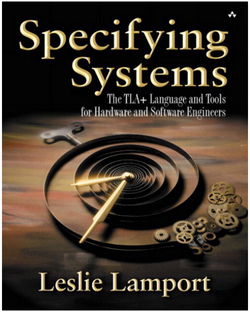
TLA$^+$ specification language

`http://tlaplus.net`



- formal language for describing and reasoning about distributed and concurrent systems
- based on mathematical logic and set theory plus linear time temporal logic TLA
- book: Addison-Wesley, 2003
  (free download for personal use)
- supported by tool set (TLA$^+$ toolbox)

TLA$^+$ specification language `http://tlaplus.net`



- formal language for describing and reasoning about distributed and concurrent systems
- based on mathematical logic and set theory plus linear time temporal logic TLA
- book: Addison-Wesley, 2003 (free download for personal use)
- supported by tool set (TLA$^+$ toolbox)

Some other publications

- Y. Yu, P. Manolios, L. Lamport: *Model checking* TLA$^+$ *Specifications*. CHARME 1999, pp. 54-66, LNCS 1703.
- S. Merz: *The Specification Language* TLA$^+$. In: Logics of Specification Languages (D. Bjørner, M. Henson, eds.), Springer 2008, pp. 401-451.
- K. Chaudhuri, D. Doligez, L. Lamport, S. Merz: *Verifying Safety Properties with the* TLA$^+$ *Proof System*. IJCAR 2010, pp. 142-148, LNCS 6173

Outline

1. Motivation and introductory example
2. Transition systems and their properties
3. System Specification in TLA$^+$
4. Verification of TLA$^+$ models
5. Structuring models: refinement and (de)composition

Why formal specifications?

- describe, analyze, and reason about systems
  - algorithms, protocols, embedded systems, ...
- at different levels of abstraction
  - requirements, high-level design, component design, code
- verify models throughout development
  - gradually introduce design decisions, detect errors early
- focus on different aspects of system under development
  - functional correctness, performance, resource usage, security, ...
- use different analysis and verification techniques
  - simulation, model checking, static analysis, theorem proving, ...

Classes of formal specification languages

- intended for different kinds of systems and properties
  - sequential algorithms
  - interactive, reactive, distributed systems
  - real-time and hybrid systems
  - security-sensitive protocols and systems
- based on different formal foundations
  - logics: first-order, higher-order, temporal, …
  - automata: transition systems, timed, hybrid automata, …
  - abstract machines: λ-calculus, rewriting, process calculi, …
- based on different specification styles
  - property-oriented: list desired correctness properties
  - model-based: describe abstract realization of intended system
- TLA$^+$: reactive systems, temporal logic, model-based

Simple TLA$^+$ specification: the hour clock

$$
\begin{array}{l}
\textsc{module } \mathit{HourClock} \\
\textsc{extends } \mathit{Naturals} \\
\textsc{variable } \mathit{hr} \\
\hline
\mathit{HCini} \triangleq \mathit{hr} \in 0..23 \\
\mathit{HCnxt} \triangleq \mathit{hr}' = \textsc{if } \mathit{hr} = 23 \textsc{ then } 0 \textsc{ else } \mathit{hr} + 1 \\
\mathit{HCsafe} \triangleq \mathit{HCini} \land \Box[\mathit{HCnxt}]_{\mathit{hr}} \\
\hline
\textsc{theorem } \mathit{HCsafe} \Rightarrow \Box \mathit{HCini}
\end{array}
$$

HourClock module: some observations

- Structure of the module
  - list of declarations, definitions, and assertions
  - *hr* — a state variable
  - *HCini* — a state predicate
  - *HCnxt* — a transition predicate (an action)
  - *HCsafe* — a temporal formula
- Formula *HCsafe* holds of a behavior if
  - initial state satisfies *HCini*
  - every transition satisfies *HCnxt* – or leaves *hr* unchanged
  - clock may stop ticking, but may not fail in any other way
- Module asserts a theorem
  - any run of the hour clock always satisfies *HCini*
  - theorem can be verified using TLC, the TLA$^+$ model checker

Non-stopping hour clock

- Formulas $Init \land \Box[Next]_v$ describe a state machine
  - initial state of system
  - allowed state transitions, including "stuttering"
- Fairness conditions rule out infinite stuttering
  - fairness: action must occur if it is possible "often enough"
  - an hour clock that never stops is specified by
    $HC \triangleq HCsafe \land \mathrm{WF}_{hr}(HCnxt)$
  - TLC verifies the theorem
    $HC \Rightarrow \forall k \in 0..23 : \Box\Diamond(hr = k)$
- Standard form of TLA$^+$ specifications: $Init \land \Box[Next]_v \land L$
  - *Init* system's initial condition
  - *Next* system's next-state relation (usually a disjunction)
  - *L* fairness conditions (for disjuncts of *Next*)

Outline

Transformational vs. reactive systems

- Transformational systems (sequential algorithms)
  - compute result from initially given input
  - semantics: relation between states before and after computation
  - partial correctness + termination
  - computational models: Turing machines, λ-calculus, ...
- Reactive systems (operating systems, controllers, ...)
  - repeated interaction between environment and system
  - semantics: (possibly) infinite sequences of states
  - safety properties: something bad never happens
  - liveness properties: something good happens eventually
  - computational models: transition systems

Outline

Transition systems

Definition

A transition system $\mathcal{T} = (Q, I, \delta)$ is given by

- a (finite or infinite) set Q of states,
- a set $I \subseteq Q$ of initial states,
- a total transition relation $\delta \subseteq Q \times Q$.

An execution of $\mathcal{T}$ is an infinite sequence $\rho = q_0 q_1 q_2 \ldots$ where $(q_i, q_{i+1}) \in \delta$ holds for all $i \in \mathbb{N}$.
A run of $\mathcal{T}$ is an execution that starts in an initial state of $\mathcal{T}$.
A state $q \in Q$ is reachable if it appears in some run of $\mathcal{T}$.

- similar to non-deterministic finite automaton; no final states
- totality of δ for technical convenience: restrict to infinite runs
- easy to ensure if δ is reflexive: stuttering closure
- deadlocks must be modeled explicitly

Variation: labeled transition systems

- Distinguish several kinds of transitions
  - transitions of system and environment
  - transitions of different processes
  - time elapse vs. system move

Definition

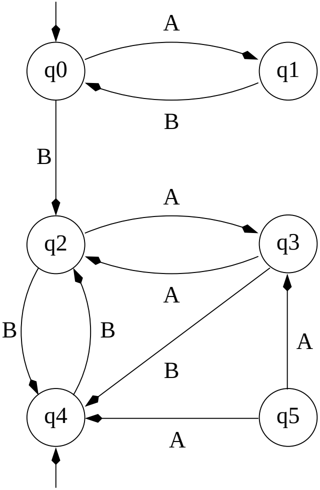
In a labeled transition system $\mathcal{T} = (\mathcal{A}, Q, I, \delta)$:

- $\mathcal{A}$ is a set of labels (actions),
- $\delta = (\delta_A)_{A \in \mathcal{A}}$ is a family of relations $\delta_A \subseteq Q \times Q$ indexed by $\mathcal{A}$.

Action A is enabled at state q if $(q, q') \in \delta_A$ for some q'.
Action A is taken at position i in run $\rho = q_0 q_1 q_2 \ldots$ if $(q_i, q_{i+1}) \in \delta_A$.

- Remarks
  - at every state, some action should be enabled
  - actions will be used later for defining fairness properties

Example



- q_0 and q_4 are initial states
- q_5 is unreachable
- A and B are enabled at q_0
- A is not enabled at q_1

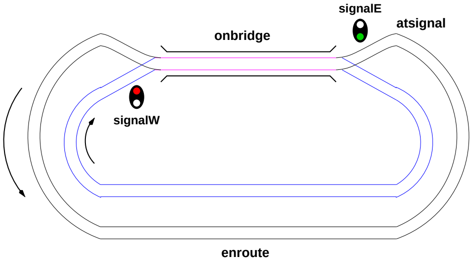
Exercises: transition system representations

1. A "stopwatch" program:

$$\begin{array}{l}\textbf{var } x, y : \text{integer} = 0, 0;\\ \textbf{cobegin}\\ \quad \alpha : \textbf{while } y = 0 \textbf{ do } \beta : x := x + 1 \textbf{ end} \quad \| \quad \gamma : y := 1\\ \textbf{coend}\end{array}$$

   Describe informally the runs of the transition system.

2. A toy railway:



   Model the behavior of the trains and the signals as a transition system.

Fairness conditions (informally)

- Transition systems are non-deterministic
  - choice between transitions of different processes
  - non-deterministic choice resulting from abstraction in the model
    e.g.: abstraction of concrete train positions by sections
  - non-determinism generates "unfair" executions
    e.g.: never execute action γ of stopwatch program
  - rule out such executions by global constraints
- Fairness conditions exclude unfair executions
  - action should be eventually taken if often enough enabled
  - weak fairness: continuously enabled
  - strong fairness: infinitely often enabled
- Note: fairness does not restrict local non-determinism

Fairness conditions (formally)

Definition (weak and strong fairness)

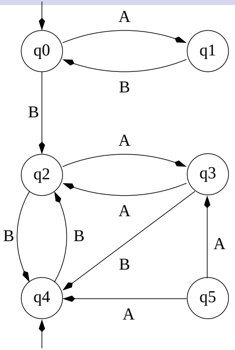
Let $\rho = q_0 q_1 q_2 \ldots$ be an execution of a labeled transition system and A be an action.

1. ρ is weakly fair w.r.t. A if for all $m \in \mathbb{N}$, if A is enabled at all states q_n for $n \geq m$ then A is taken at some position $n \geq m$.
2. ρ is strongly fair w.r.t. A if for all $m \in \mathbb{N}$, if A is enabled at infinitely many states q_n for $n \geq m$ then A is taken at some position $n \geq m$.

- Fairness conditions rule out executions
  - ρ is not weakly fair w.r.t. A if there exists some $m \in \mathbb{N}$ such that A is forever enabled from position m onward but never taken
  - ρ is not strongly fair w.r.t. A if A is infinitely often enabled but taken only finitely often

Fairness: examples and exercises

- Are the following runs fair w.r.t. *A* and *B*?
  - $\rho_1 = q_0 q_1 q_0 q_1 \cdots$
  - $\rho_2 = q_0 q_2 q_3 q_2 q_3 \cdots$
  - $\rho_2 = q_4 q_2 q_4 q_2 \cdots$



- Exercises
  - ρ is weakly fair w.r.t. A iff either A is taken infinitely often in ρ or A is infinitely often disabled.
  - ρ is strongly fair w.r.t. A iff either A is taken infinitely often in ρ or A is only finitely often enabled.
  - If ρ is strongly fair w.r.t. A then ρ is weakly fair w.r.t. A.
  - If ρ is (strongly/weakly) fair w.r.t. A then so is any suffix of ρ.

Outline

Properties of transition systems

- Properties evaluated over runs
  - two trains are never simultaneously in section onbridge
  - no train waits forever at the signal
  - if both trains wait at signal then each will enter the bridge at most once before the other
- Properties related to the transition structure
  - some initial state is reachable from all states
  - actions A and B are in conflict, resp. are independent
  - two processes can cooperate to starve a third process
- Here: consider first class of properties
  - define properties as sets of runs
  - system correctness: $Runs(\mathcal{T}) \subseteq \Phi$

Safety and liveness properties (informally)

- Safety properties: "something bad never happens"
  - trains are never simultaneously on the bridge
  - data arrives at receiver in FIFO order
- Liveness properties: "something good happens eventually"
  - train waiting at signal will eventually enter onbridge
  - every data item sent will eventually be received
  - process A executes infinitely often
- Two fundamental classes of properties of runs
  - generalization of partial correctness and termination
  - the following property is neither (pure) safety nor (pure) liveness:

    the train remains at the signal until it enters onbridge,
    which will happen eventually

Formal definitions (Alpern & Schneider 1985)

Definition

Φ is a safety property if for every infinite state sequence σ:

$$\sigma \in \Phi \quad \text{iff} \quad \text{for all } n \in \mathbb{N} \text{ there is } \tau \text{ s.t. } \sigma[..n] \circ \tau \in \Phi.$$

- σ violates a safety property if there exists a finite prefix $\sigma[..n]$ that cannot be extended to an infinite sequence satisfying Φ
- "bad things" are observable after some finite time

Definition

Φ is a liveness property if for every finite state sequence ρ there exists τ s.t. $\rho \circ \tau \in \Phi$.

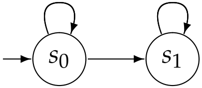
- liveness properties do not constrain finite prefixes
- the "good thing" may still occur later

Safety/liveness: examples and exercises

1. The set of runs of a transition system is a safety property.
2. Weak or strong fairness conditions are liveness properties.
3. If all Φ_i $(i \in I)$ are safety properties then $\bigcap_{i \in I} \Phi_i$ is a safety property.
4. If Φ is a liveness property and $\Phi \subseteq \Psi$ then Ψ is a liveness property.
5. The trivial property containing all state sequences is the only property that is both a safety and a liveness property.
6. Given property Φ, its safety closure
$$\mathcal{C}(\Phi) = \{\sigma : \text{for all } n \in \mathbb{N} \text{ there is } \tau \text{ s.t. } \sigma[..n] \circ \tau \in \Phi\}$$
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 $\Phi = S_\Phi \cap L_\Phi$.

Machine closure

- Specifications: transition system plus liveness properties
  - liveness properties can impose additional constraints on transitions
  - example: liveness property that requires infinitely many visits of s_0



Definition

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

- Idea: L constrains infinite, but not finite runs satisfying S
- Show: if (S, L) is machine closed and Φ is a safety property then $S \cap L \subseteq \Phi$ iff $S \subseteq \Phi$
- Theorem: specifications given by a transition system and a countable number of fairness condition are machine closed

Summary

- Transition systems: semantic basis for reactive systems
  - transition relation describes possible steps
  - fairness conditions: global constraints on non-deterministic choices
- Properties of runs: sets of state sequences
  - “linear-time” correctness properties, ignore branching properties
  - dichotomy of safety and liveness properties

Outline

The Specification Language TLA$^+$

- Temporal Logic of Actions (TLA)
  - linear-time temporal logic
  - formulas represent systems and properties
  - stuttering invariance: basis for refinement and composition
- Zermelo-Fränkel set theory: classical mathematics
  - specification of data structures in set theory
  - high levels of abstraction and expressiveness
  - explicit definitions of data and operations (e.g., communication)
- Module language
  - structuring of specifications: composition and hiding
  - standard library of common structures, user extensible

Temporal logics in computer science

- Originally: temporal relations in natural language
  - *Yesterday she said that she'd come tomorrow, so she should come today.*
  - A. Prior: *Past, present, and future.* Oxford University Press, 1967
- 1977: express properties of reactive systems
  - A. Pnueli: *The temporal logic of programs.* FOCS'77
- 1981: automatic verification of finite-state systems
  - E.M. Clarke, E.A. Emerson: *Synthesis of synchronization skeletons for branching time temporal logic*. Logics of Programs, 1981
  - J.P. Queille, J. Sifakis: *Specification and verification of concurrent systems in Cesar*. Intl. Symp. Programming, LNCS 137

System	satisfies	property
	formalized as	
Transition system	is model of	temporal formula

Outline

Anatomy of TLA

- Variables and constants
  - (state) variables $v, hr, queue$ represent system state
  - constants $x, k, nProcs$ represent system parameters
- Action formulas: predicates over (pairs of) states
  - represent initial condition, invariants, system transitions

    $hr \in 0..23$

    $hr' = \text{IF } hr = 23 \text{ THEN } 0 \text{ ELSE } hr + 1$

    $\exists p \in 1..nProcs : Request(p)$
- Temporal formulas: predicates over state sequences
  - represent system specifications or properties

    $Init \wedge \Box[Next]_{vars} \wedge \mathrm{WF}_{vars}(Exit_1 \vee Exit_2)$

    $\forall p \in 1..nProcs : request[p] \leadsto grant[p]$

Syntax of action formulas

- First-order formulas (over given signature) containing
  - constants $x, y, k, \ldots \in \mathcal{X}$
  - state variables $v, w, \ldots \in \mathcal{V}$
  - primed state variables $v', w', \ldots \in \mathcal{V}'$
- Examples
  - $v = 0, \quad v > 0 \land v' = v - 1, \quad \exists k : v - w' < x + 3 * k$
  - free and bound variables, substitution: as in first-order logic
- State and transition formulas
  - terms: transition functions
  - formulas: transition predicates
  - formulas without free primed flexible variables: state formulas
  - formulas without flexible variables: constant formulas

Semantics of action formulas

- First-order interpretation $\mathcal{I}$ for underlying signature
  - provides a non-empty universe $|\mathcal{I}|$ of values
  - interprets function and predicate symbols: $0, +, <, \in, \ldots$
- Interpretations of constants and variables
  - valuations of state variables: states $s : \mathcal{V} \rightarrow |\mathcal{I}|$
  - valuation of constants $\xi : \mathcal{X} \rightarrow |\mathcal{I}|$
- Interpretation of action formulas $[\![A]\!]^{\xi}_{s,t} \in \{\mathsf{tt}, \mathsf{ff}\}$
  - standard inductive definition
  - s, t interpret unprimed and primed flexible variables
  - ξ interprets rigid variables
- Interpretation of state formulas does not depend on second state

Notations for action formulas (1)

- "Priming" of state formulas e
  - e' action formula obtained by replacing every variable v by v'
  - rename bound variables as necessary to avoid confusion

$$\begin{array}{rcl} (v+1)' & \equiv & v'+1 \\ (\exists k : v = k + w)' & \equiv & \exists k : v' = k + w' \\ (\exists v' : v = v' + w)' & \equiv & \exists vp : v' = vp + w' \end{array}$$

- UNCHANGED $e \triangleq e' = e$

- Bracketed action formulas

$$\begin{array}{rcll} [A]_e & \triangleq & A \lor e' = e & \text{A or stuttering (w.r.t. e)} \\ \langle A \rangle_e & \triangleq & A \land \neg(e' = e) & \text{A and change of e} \end{array}$$

  - Note: $\langle A \rangle_e \equiv \neg[\neg A]_e$ $\quad \neg\langle A \rangle_e \equiv [\neg A]_e$
    $[A]_e \equiv \neg\langle \neg A \rangle_e$ $\quad \neg[A]_e \equiv \langle \neg A \rangle_e$

Notations for action formulas (2)

- Enabledness of actions
  - $\text{ENABLED } A \;\triangleq\; \exists v_1', \ldots, v_n' : A$

    ($v_1', \ldots, v_n'$ all free primed state variables in A)
  - $[\![\text{ENABLED } A]\!]_s^\xi = \text{tt}$ iff $[\![A]\!]_{s,t}^\xi = \text{tt}$ for some state t
  - will be used to define fairness conditions in TLA
- Sequential composition of actions (atomic)
  - $$A \cdot B \;\equiv\; \exists v_1'', \ldots, v_n'' : A[v_1''/v_1', \ldots, v_n''/v_n'] \wedge B[v_1''/v_1, \ldots, v_n''/v_n]$$

    ($v_1, \ldots, v_n$ all free state variables in A or B)
  - $[\![A \cdot B]\!]_{s,t}^\xi = \text{tt}$ iff $[\![A]\!]_{s,u}^\xi = \text{tt}$ and $[\![B]\!]_{u,t}^\xi = \text{tt}$ for some state u
  - sequential composition of A and B as single atomic action

Temporal formulas of TLA (to be completed)

- interpreted over ω-sequence $\sigma = s_0 s_1 \ldots$ of states and valuation ξ
  - $\sigma, \xi \models F$ if F holds over (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 $[\![P]\!]^{\xi}_{s_0} = \mathrm{tt}$
Boolean combinations	$\sigma, \xi \models F \wedge G$ iff $\sigma, \xi \models F$ and $\sigma, \xi \models G$, etc.
first-order quantifiers	$\sigma, \xi \models \exists x : F$ iff $\sigma, \zeta \models F$ for some $\zeta \sim_x \xi$
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 $[\![[A]_e]\!]^{\xi}_{s_n, s_{n+1}} = \mathrm{tt}$ for all $n \in \mathbb{N}$

  - Note: $[A]_e$ and $\Box A$ are not temporal formulas (for action A)
  - if ξ is unimportant, just write $\sigma \models F$

Notations for temporal formulas

- "eventually" operator $\quad \Diamond F \ \triangleq\ \neg\Box\neg F$
  - semantics: $\sigma \models \Diamond F$ iff $\sigma[n..] \models F$ for some $n \in \mathbb{N}$
- "eventually angle A sub e" $\quad \Diamond\langle A\rangle_e \ \triangleq\ \neg\Box[\neg A]_e$
  - some future state transition satisfies $\langle A\rangle_e$
  - again, $\langle A\rangle_e$ and $\Diamond A$ are not temporal formulas
- "leadsto" operator $\quad F \leadsto G \ \equiv\ \Box(F \Rightarrow \Diamond G)$
  - every occurrence of F is followed by some occurrence of G

Example: semantics of temporal formulas

Which of the following formulas hold in this behavior?

x	0	0	3	7	0	1	1	0	2	$\cdots$ (always $\neq 0$)
y	1	1	0	0	0	0	3	4	0	$\cdots$ (always $= 0$)

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

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

$$\neg\Box\Diamond F \equiv \Diamond\Box\neg F \qquad \Diamond\Box\Diamond F \equiv \Box\Diamond F$$
$$\neg\Diamond\Box F \equiv \Box\Diamond\neg F \qquad \Box\Diamond\Box F \equiv \Diamond\Box F$$

Fairness of actions in TLA

- Weak fairness
  - action will be taken eventually if it is continuously enabled

$$\mathrm{WF}_e(A) \triangleq \Box(\Box\textsc{Enabled}\ \langle A\rangle_e \Rightarrow \Diamond\langle A\rangle_e)$$

- Strong fairness
  - action will be taken eventually if it is infinitely often enabled

$$\mathrm{SF}_e(A) \triangleq \Box(\Box\Diamond\textsc{Enabled}\ \langle A\rangle_e \Rightarrow \Diamond\langle A\rangle_e)$$

- Equivalent conditions

$$\begin{aligned}
\mathrm{WF}_e(A) &\equiv \Diamond\Box\textsc{Enabled}\ \langle A\rangle_e \Rightarrow \Box\Diamond\langle A\rangle_e\\
\mathrm{WF}_e(A) &\equiv \Box\Diamond\neg\textsc{Enabled}\ \langle A\rangle_e \vee \Box\Diamond\langle A\rangle_e\\
\mathrm{SF}_e(A) &\equiv \Box\Diamond\textsc{Enabled}\ \langle A\rangle_e \Rightarrow \Box\Diamond\langle A\rangle_e\\
\mathrm{SF}_e(A) &\equiv \Diamond\Box\neg\textsc{Enabled}\ \langle A\rangle_e \vee \Box\Diamond\langle A\rangle_e
\end{aligned}$$

Example: "stopwatch" program in TLA$^+$

$$
\begin{array}{l}
\textsf{MODULE } \textit{Stopwatch} \\
\textsf{EXTENDS } \textit{Naturals} \\
\textsf{VARIABLES } pc_1, pc_2, x, y \\
\hline
\end{array}
$$

$$
\begin{array}{lcl}
\textit{Init} & \triangleq & pc_1 = \text{"alpha"} \land pc_2 = \text{"gamma"} \land x = 0 \land y = 0 \\
A & \triangleq & \land\ pc_1 = \text{"alpha"} \\
 & & \land\ pc_1' = \text{IF } y = 0 \text{ THEN "beta" ELSE "stop"} \\
 & & \land\ \text{UNCHANGED } \langle pc_2, x, y \rangle \\
B & \triangleq & \land\ pc_1 = \text{"beta"} \land pc_1' = \text{"alpha"} \\
 & & \land\ x' = x + 1 \land \text{UNCHANGED } \langle pc_2, y \rangle \\
G & \triangleq & \land\ pc_2 = \text{"gamma"} \land pc_2' = \text{"stop"} \\
 & & \land\ y' = 1 \land \text{UNCHANGED } \langle pc_1, x \rangle \\
\textit{vars} & \triangleq & \langle pc_1, pc_2, x, y \rangle \\
\textit{Spec} & \triangleq & \textit{Init} \land \Box[A \lor B \lor G]_{\textit{vars}} \land \mathrm{WF}_{\textit{vars}}(A \lor B) \land \mathrm{WF}_{\textit{vars}}(G)
\end{array}
$$

var x, y : integer = 0, 0;
cobegin
    α : **while** $y = 0$ **do**
        $\beta : x := x + 1$
    end
$\|$
    $\gamma : y := 1$
coend

- explicit encoding of control structure
- process structure not obvious from TLA$^+$ representation

Aside: PlusCal (Lamport, 2006)

```
-- algorithm Peterson {
variables req  = [id ∈ Node ↦ FALSE],
          turn = 0
process (Proc ∈ Node) {
  ncs: while (TRUE) {
         skip;
  rq:    req[self] := TRUE; turn := other(self);
  try:   await (turn = self ∨ ¬lock[other(self)]);
  cs:    skip;
  lv:    req[self] := FALSE
} } }
```

- High-level language for multi-process algorithms
  - atomicity indicated through labels
  - algorithm embedded in TLA$^+$ module
- Translation to TLA$^+$: simulation and model checking

Stuttering invariance

- Action formulas must be "protected" in TLA: $\Box[A]_t,\ \Diamond\langle A\rangle_t$
  - finite repetition of identical states (up to t) preserves truth
  - this observation extends to all TLA formulas
- Stuttering equivalence ($\approx$)
  - $$\natural(s_0 s_1 \ldots) \triangleq \begin{array}{l} \text{IF } \forall k \in \mathbb{N} : s_k = s_0 \\ \text{THEN } s_0 s_1 \ldots \\ \text{ELSE } \begin{array}[t]{ll} \text{LET} & m = \min\{k \in \mathbb{N} : s_k \neq s_0\} \\ \text{IN} & s_0 \circ \natural(s_m s_{m+1} \ldots) \end{array} \end{array}$$
  - $\sigma \approx \tau$ iff $\natural\sigma = \natural\tau$

Stuttering invariance

- Action formulas must be "protected" in TLA: $\Box[A]_t, \ \Diamond\langle A\rangle_t$
  - finite repetition of identical states (up to t) preserves truth
  - this observation extends to all TLA formulas
- Stuttering equivalence ($\approx$)
  - $$\natural(s_0 s_1 \ldots) \triangleq \begin{array}{l} \text{IF } \forall k \in \mathbb{N} : s_k = s_0 \\ \text{THEN } s_0 s_1 \ldots \\ \text{ELSE } \text{LET } m = \min\{k \in \mathbb{N} : s_k \neq s_0\} \\ \qquad\quad \text{IN } s_0 \circ \natural(s_m s_{m+1} \ldots) \end{array}$$
  - $\sigma \approx \tau \quad \text{iff} \quad \natural\sigma = \natural\tau$

Theorem (stuttering invariance)

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

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

- Fundamental for refinement and composition

Example: hour-minute clock

- Extension of hour clock by a minute hand

MODULE *HourMinuteClock*

EXTENDS *Naturals*, *HourClock*
VARIABLE *min*

$$HMCini \triangleq HCini \land min \in 0..59$$
$$Min \triangleq min' = \text{IF } min = 59 \text{ THEN } 0 \text{ ELSE } min + 1$$
$$Hr \triangleq (min = 59 \land HCnxt) \lor (min < 59 \land hr' = hr)$$
$$HMCnxt \triangleq Min \land Hr$$
$$HMC \triangleq HMCini \land \Box[HMCnxt]_{\langle hr,min \rangle} \land \mathrm{WF}_{\langle hr,min \rangle}(HMCnxt)$$

THEOREM $HMC \Rightarrow HC$

  - stuttering invariance is essential for refinement as implication
  - no formal difference between specifications and properties

Outline

Logical language underlying TLA$^+$

- Specification language TLA$^+$: based on ZF set theory
  - standard logical basis for formalizing mathematics
  - high levels of abstraction and expressiveness

    e.g., Riemann integral in 15 lines of TLA$^+$
  - set-theoretical language, no types

    $5 = \{\}$ or $17 \land \text{“abc”}$ are well-formed formulas
    but (of course!) we don't know if they are true

- Formally: logical basis
  - binary predicate symbol $\in$
  - choice operator (binder) $\text{CHOOSE } x : P$

- Hilbert's choice operator
  - $\text{CHOOSE } x : P$ denotes an arbitrary value satisfying P
  - or an arbitrary, fixed value if no such value exists

Set-theoretic constructions

- Characteristic axioms of Hilbert's choice

$$(\exists x : P(x)) \;\equiv\; P(\text{CHOOSE } x : P(x)) \qquad \text{[choice]}$$

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

- Examples: set-theoretical constructions and theorems
  - generalized union

    $$\text{UNION } S \;\triangleq\; \text{CHOOSE } M : \forall x : x \in M \;\equiv\; \exists T \in S : x \in T$$
  - no set contains all elements

    $$(\text{CHOOSE } x : x \notin S) \notin S$$
  - Russell's paradoxical set is not well-defined

    $$(\text{CHOOSE } S : \forall x : x \in S \equiv x \notin x) = (\text{CHOOSE } x : \mathsf{ff})$$

- Exercises: define the following constructions
  - subset relation $S \subseteq T$
  - set intersection $S \cap T$
  - powerset $\text{SUBSET } S$: set containing all subsets of S
  - set comprehensions $\{x \in S : P(x)\}$ and $\{e(x) : x \in S\}$

Choice vs. non-determinism

- Two formulas for specifying resource allocation

$$\begin{array}{l} Alloc_{nd} \triangleq \\ \land\ owner = NoProcess \\ \land\ waiting \neq \{\} \\ \land\ owner' \in waiting \\ \land\ waiting' = waiting \setminus \{owner'\} \end{array}$$

$$\begin{array}{l} Alloc_{ch} \triangleq \\ \land\ owner = NoProcess \\ \land\ waiting \neq \{\} \\ \land\ owner' = \text{CHOOSE } p : p \in waiting \\ \land\ waiting' = waiting \setminus \{owner'\} \end{array}$$

- enabledness: resource is free, some process is waiting
- effect: assign the resource to some waiting process
- $Alloc_{nd}$ produces $Card(waiting)$ successor states
- $Alloc_{ch}$ produces single successor state for some fixed process

CHOOSE operator is deterministic: "arbitrary, but fixed element"

Functions in TLA$^+$

- TLA$^+$ introduces functions as primitive values

$[S \rightarrow T]$	set of functions with domain S and codomain T
$\text{DOMAIN}\, f$	domain of function f
$f[e]$	application of function f to argument e
$[x \in S \mapsto e]$	function with domain S mapping x to e
$[f \text{ EXCEPT } ![t] = e]$	function override: like f, but argument t maps to e $[f \text{ EXCEPT } ![t] = @ + e] = [f \text{ EXCEPT } ![t] = f[t] + e]$

- Characteristic property of functional values

$$f = [x \in \text{DOMAIN}\, f \mapsto f[x]]$$

- Note
  - value $f[x]$ unspecified if $x \notin \text{DOMAIN}\, f$
  - notations extend to several arguments: $[x \in S, y \in T \mapsto x + y]$

Recursive functions

- Recursive functions can be defined using choice

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

- Abbreviated notation

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

- Potential pitfalls
  - existence of such a function should be justified
  - no implicit commitment to least fixed point semantics

Numbers in TLA⁺ (1)

- Natural numbers defined using choice from Peano axioms

$$
\begin{array}{lll}
\multicolumn{3}{l}{PeanoAxioms(N, Z, Sc) \ \triangleq} \\
\quad \land\ Z \in N & & \\
\quad \land\ Sc \in [N \rightarrow N] & & \\
\quad \land\ \forall n \in N : (n \neq Z) \equiv (\exists m \in N : n = Sc[m]) & & \\
\quad \land\ \forall S \in \text{SUBSET}\ N : Z \in S \land (\forall n \in S : Sc[n] \in S) \Rightarrow S = N & & \\
Succ & \triangleq & \text{CHOOSE}\ Sc : \exists N, Z : PeanoAxioms(N, Z, Sc) \\
Nat & \triangleq & \text{DOMAIN}\ Succ \\
Zero & \triangleq & \text{CHOOSE}\ Z : PeanoAxioms(Nat, Z, Succ)
\end{array}
$$

  - existence of such values can be justified from set theory

Numbers in TLA$^+$ (2)

- Arithmetic operations defined recursively (simplified)

$$
\begin{array}{lcl}
pred(x) & \triangleq & \text{CHOOSE } y : x = Succ[y] \\
plus[x \in Nat, y \in Nat] & \triangleq & \text{IF } y = 0 \text{ THEN } x \text{ ELSE } Succ[plus[x, pred(y)]] \\
x + y & \triangleq & plus[x, y] \\
x \leq y & \triangleq & \exists z : y = x + z
\end{array}
$$

- Numbers and intervals

$$
\begin{array}{l}
0 \triangleq Zero, \;\; 1 \triangleq Succ[0], \;\; 2 \triangleq Succ[1], \;\; \ldots \\
i..j \triangleq \{n \in Nat : i \leq n \wedge n \leq j\}
\end{array}
$$

- Integer and real numbers
  - defined similarly as supersets of *Nat*
  - arithmetic operations agree: $3.5 + 2.5 \in Nat$

Tuples and sequences

- Tuples and sequences are represented as functions

$$\langle e_1, \ldots, e_n \rangle \triangleq [i \in 1..n \mapsto \text{IF } i = 1 \text{ THEN } e_1 \ \ldots \ \text{ELSE } e_n]$$

- Standard operators involving sequences

$$\begin{array}{lcl}
Seq(S) & \triangleq & \text{UNION } \{[1..n \rightarrow S] : n \in Nat\} \\
Len(s) & \triangleq & \text{CHOOSE } n \in Nat : \text{DOMAIN } s = 1..n \\
Head(s) & \triangleq & s[1] \\
Tail(s) & \triangleq & [i \in 1..(Len(s) - 1) \mapsto s[i+1]] \\
s \circ t & \triangleq & [i \in 1..(Len(s) + Len(t)) \mapsto \\
 & & \quad \text{IF } i \leq Len(s) \text{ THEN } s[i] \text{ ELSE } t[i - Len(s)]] \\
Append(s, e) & \triangleq & s \circ \langle e \rangle
\end{array}$$

- Question: What are $Head(\langle\rangle)$ and $Tail(\langle\rangle)$?

Exercises

1. Define an operator $IsSorted(s)$ such that for any sequence s of numbers, $IsSorted(s)$ is true iff s is sorted.

2. Define a function $sort \in [Seq(Real) \rightarrow Seq(Real)]$ such that $sort[s]$ is a sorted sequence containing the same elements as s.

3. Give a recursive definition of the $mergesort$ function.
   Does $sort = mergesort$ hold for your definitions? Why (why not)?

4. Define operators $IsFiniteSet(S)$ and $card(S)$ such that $IsFiniteSet(S)$ holds iff S is a finite set and that $card(S)$ denotes the cardinality of S if S is finite.

Strings and records

- String representations
  - strings are sequences of characters
  - standard operations on sequences apply to strings:
    $\text{"th"} \circ \text{"is"} = \text{"this"}$

- Records: functions whose domain is a finite set of strings

short notation	instead of
$account.bal$	$account[\text{"bal"}]$
$[account \text{ EXCEPT } !.bal = @ + sum]$	$[account \text{ EXCEPT } ![\text{"bal"}] = @ + sum]$
$[num \mapsto 1234567, bal \mapsto -321.45]$	$[fld \in \{\text{"num"}, \text{"bal"}\} \mapsto$ $\quad \text{IF } fld = \text{"num" THEN } 1234567$ $\quad \text{ELSE } -321.45]$
$[x : S, y : T]$	$[fld \in \{\text{"x"}, \text{"y"}\} \to$ $\quad \text{IF } fld = \text{"x" THEN } S \text{ ELSE } T]$

Outline

TLA$^+$ modules: basic ideas

- A TLA$^+$ module essentially contains a sequence of
  - declarations of constant and variable parameters
  - definitions of operators (and functions)
  - assertions of assumptions and theorems
- Module hierarchy helps structuring specifications
  - scoping of parameter and operator names
  - module extension: import all parameters and definitions
  - module instantiation: import definitions, must provide parameters
  - local modules: hide auxiliary definitions
- Meaning of modules
  - accumulate assumptions: hypotheses for theorems
  - operators: replace defined symbols by definition body

Principle of unique names

- Identifiers active in a scope cannot be reused
  - redeclaration or redefinition is illegal
  - active identifiers must not be used as bound variables

MODULE *IllegalModule*

EXTENDS *Naturals*
CONSTANTS x, y

$m + n$	$\triangleq$	$\ldots$	(* + defined in module *Naturals* *)
$Foo(y, z)$	$\triangleq$	$\exists x : \ldots$	(* x and y active in current scope *)
Nat	$\triangleq$	LET $y \triangleq \ldots$ IN $\ldots$	(* clashes of *Nat* and y *)

- Import of same module via different paths is allowed
- Definitions can be protected from export: LOCAL keyword

Module extension

```
---------------- MODULE Foo ----------------
EXTENDS Bar, Baz
CONSTANTS Data, Compare(_)
---------------------------------------------
...
=============================================
```

- Module *Foo* exports
  - symbols declared or defined in module *Foo*
  - (non-local) symbols exported by modules *Bar*, *Baz*
- Within module *Foo*
  - may use, but not redefine or redeclare symbols exported by *Bar*, *Baz*
- Equivalent to textual inclusion of modules *Bar*, *Baz*

Module instantiation: import with renaming

$$
\begin{array}{lcl}
\multicolumn{3}{c}{\textsc{module}\ \mathit{Component}} \\
\mathit{InChan} & \triangleq & \textsc{instance}\ \mathit{Channel}\ \textsc{with}\ \mathit{Data} \leftarrow \mathit{Message}, \mathit{chan} \leftarrow \mathit{in} \\
\mathit{Chan}(c) & \triangleq & \textsc{instance}\ \mathit{Channel}\ \textsc{with}\ \mathit{Data} \leftarrow \mathit{Message}, \mathit{chan} \leftarrow c
\end{array}
$$

- Use of operators defined in instantiated module

  $\mathit{InChan}!\mathit{Send}(d)$ resp. $\mathit{Chan}(\mathit{in})!\mathit{Send}(d)$

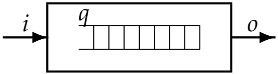
- Special cases
  - identity renaming can be omitted in instantiation
  - instance name can be omitted if only one copy needed
  - LOCAL instantiation is possible

- Operators defined in instantiating module
  - non-local instances are exported for use within later modules
  - symbols declared or defined in module *Channel* are not exported

Outline

Formally describe a FIFO queue

- Basic system component



  - input elements over input "wire" i
  - output over "wire" o, in same order
  - simplifying assumption: subsequent elements are different
  - also assume unbounded queue capacity

- Explore possible specification styles in TLA$^+$
  - specify transition system: represent state components
  - encode synchronization and communication

First attempt: lossy queue

MODULE *LossyQueue*

EXTENDS *Sequences*
VARIABLES i, q, o

$$LQInit \triangleq q = \langle\rangle \land i = o$$

$$LQEnq \triangleq q' = Append(q, i') \land o' = o$$

$$LQDeq \triangleq q \neq \langle\rangle \land o' = Head(q) \land q' = Tail(q) \land i' = i$$

$$LQLive \triangleq \mathrm{WF}_{q,o}(SQDeq)$$

$$LQSpec \triangleq LQInit \land \Box[LQEnq \lor LQDeq]_{q,o} \land LQLive$$

- i and o represent interface, q is internal buffer
- interleaving: enqueue and dequeue cannot happen at once
- fairness for dequeue: every element should eventually be output

First attempt: lossy queue

MODULE *LossyQueue*

EXTENDS *Sequences*
VARIABLES i, q, o

$$
\begin{aligned}
LQInit &\triangleq q = \langle\rangle \wedge i = o \\
LQEnq &\triangleq q' = Append(q, i') \wedge o' = o \\
LQDeq &\triangleq q \neq \langle\rangle \wedge o' = Head(q) \wedge q' = Tail(q) \wedge i' = i \\
LQLive &\triangleq \mathrm{WF}_{q,o}(SQDeq) \\
LQSpec &\triangleq LQInit \wedge \Box[LQEnq \vee LQDeq]_{q,o} \wedge LQLive
\end{aligned}
$$

- i and o represent interface, q is internal buffer
- interleaving: enqueue and dequeue cannot happen at once
- fairness for dequeue: every element should eventually be output
- buffer can enqueue same input several times, or not at all

Synchronous communication, interleaving

MODULE *SyncQueue*

EXTENDS *Sequences*
VARIABLES i, q, o

$$
\begin{aligned}
SQInit &\triangleq q = \langle\rangle \land i = o \\
SQEnq &\triangleq i' \neq i \land q' = Append(q, i') \land o' = o \\
SQDeq &\triangleq q \neq \langle\rangle \land o' = Head(q) \land q' = Tail(q) \land i' = i \\
SQLive &\triangleq \mathrm{WF}_{i,q,o}(SQDeq) \\
SQSpec &\triangleq SQInit \land \Box[SQEnq \lor SQDeq]_{i,q,o} \land SQLive
\end{aligned}
$$

- change of input wire triggers enqueue: i appears in index
- interleaving style and fairness as before
- standard specification style in TLA$^+$
- every run of *SQSpec* also satisfies *LQSpec*

Asynchronous communication, interleaving

MODULE *AsyncQueue*

EXTENDS *Sequences*
VARIABLES i, q, o, sig

$$AQInit \triangleq q = \langle\rangle \land i = o \land sig = 0$$

$$AQEnv \triangleq sig = 0 \land sig' = 1 \land \text{UNCHANGED } \langle q, o \rangle$$

$$AQEnq \triangleq sig = 1 \land sig' = 0 \land q' = Append(q, i') \land \text{UNCHANGED } \langle i, o \rangle$$

$$AQDeq \triangleq q \neq \langle\rangle \land o' = Head(q) \land q' = Tail(q) \land \text{UNCHANGED } \langle i, sig \rangle$$

$$AQLive \triangleq \mathrm{WF}_{i,q,o,sig}(AQEnq) \land \mathrm{WF}_{i,q,o,sig}(AQDeq)$$

$$AQSpec \triangleq AQInit \land \Box[AQEnv \lor AQEnq \lor AQDeq]_{i,q,o,sig} \land AQLive$$

- decouple sender input and acceptance of input by the queue
- explicit "handshake protocol" for enqueueing values
- fairness condition on *AQEnq* ensures system reaction
- every run of *AQSpec* also satisfies *LQSpec*

Synchronous communication, non-interleaving

MODULE *SyncNIQueue*

EXTENDS *Sequences*
VARIABLES i, q, o

$$
\begin{array}{lcl}
SNQInit & \triangleq & q = \langle\rangle \land i = o \\
d(v) & \triangleq & \text{IF } v' = v \text{ THEN } \langle\rangle \text{ ELSE } \langle v' \rangle \\
SNQEnq & \triangleq & i' \neq i \land q \circ d(i) = d(o) \circ q' \\
SNQDeq & \triangleq & q \neq \langle\rangle \land o' = Head(q) \land q \circ d(i) = d(o) \circ q' \\
SNQLive & \triangleq & \mathrm{WF}_{q,o}(SNQDeq) \\
SNQSpec & \triangleq & \land\ SNQInit \land \Box[SNQEnq]_i \land \Box[SNQDeq]_o \\
 & & \land\ \Box[SNQEnq \lor SNQDeq]_q \land SNQLive
\end{array}
$$

- enqueue and dequeue may happen simultaneously if q non-empty
- separate next-state relation per (group of) variable(s)
- every run of *SQSpec* also satisfies *SNQSpec*

Outline

Informal requirements

- A set of clients compete for a (fixed, finite) set of resources.
- Whenever a client holds no resources and has no outstanding requests, he can request a set of resources.
- The allocator may allocate a set of available resources to a client that requested them, possibly without completely satisfying the client's request.
- Clients may return resources they hold at any time.
- A client that received all resources he requested must eventually return them (not necessarily at once).

Objectives:

- Clients have exclusive access to resources they hold.
- Every request is eventually satisfied.

A first solution in TLA$^+$ (1/4)

MODULE *SimpleAllocator*

EXTENDS *FiniteSet*

CONSTANTS *Clients*, *Resources*
ASSUME *IsFiniteSet*(*Resources*)

VARIABLES
unsat, (* *unsat*[*c*] denotes the outstanding requests of client *c* *)
alloc (* *alloc*[*c*] denotes the resources allocated to client *c* *)

$$TypeInvariant \triangleq$$
$$\land\ unsat \in [Clients \rightarrow \text{SUBSET } Resources]$$
$$\land\ alloc \in [Clients \rightarrow \text{SUBSET } Resources]$$

$available \triangleq$ (* set of resources free for allocation *)
$$Resources \setminus (\text{UNION } \{alloc[c] : c \in Clients\})$$

A first solution in TLA$^+$ (2/4)

$Init \triangleq$ (* initially, no resources have been requested or allocated *)

$\quad \land\ unsat = [Clients \rightarrow \{\}]$

$\quad \land\ alloc = [Clients \rightarrow \{\}]$

(* Client c requests set S of resources, provided it has no outstanding requests and no a

$Request(c,S) \triangleq$

$\quad \land\ unsat[c] = \{\} \land alloc[c] = \{\}$

$\quad \land\ S \neq \{\} \land unsat' = [unsat \text{ EXCEPT } ![c] = S]$

$\quad \land\ \text{UNCHANGED } alloc$

(* Allocation of a set of available resources to a client that requested them. *)

$Allocate(c,S) \triangleq$

$\quad \land\ S \neq \{\} \land S \subseteq available \cap unsat[c]$

$\quad \land\ alloc' = [alloc \text{ EXCEPT } ![c] = @ \cup S]$

$\quad \land\ unsat' = [unsat \text{ EXCEPT } ![c] = @ \setminus S]$

A first solution in TLA$^+$ (3/4)

$Return(c,S) \triangleq$ (* Client c returns a set of resources that it holds. *)

$$\begin{aligned}
&\land S \neq \{\} \land S \subseteq alloc[c] \\
&\land alloc' = [alloc \text{ EXCEPT } ![c] = @ \setminus S] \\
&\land \text{UNCHANGED } unsat
\end{aligned}$$

$Next \triangleq$ (* The next-state relation. *)

$$\begin{aligned}
&\exists c \in Clients, S \in \text{SUBSET } Resources : \\
&\quad Request(c,S) \lor Allocate(c,S) \lor Return(c,S)
\end{aligned}$$

$vars \triangleq \langle unsat, alloc \rangle$

$SimpleAllocator \triangleq$ (* The complete high-level specification. *)

$$\begin{aligned}
&\land Init \land \Box[Next]_{vars} \\
&\land \forall c \in Clients : \mathrm{WF}_{vars}(Return(c, alloc[c])) \\
&\land \forall c \in Clients : \mathrm{SF}_{vars}(\exists S \in \text{SUBSET } Resources : Allocate(c,S))
\end{aligned}$$

A first solution in TLA$^+$ (4/4)

(* Definition of system properties *)

$$ResourceMutex \triangleq \forall c_1, c_2 \in Clients : c_1 \neq c_2 \Rightarrow alloc[c_1] \cap alloc[c_2] = \{\}$$

$$ClientsWillObtain \triangleq \forall c \in Clients, r \in Resources : (r \in unsat[c]) \leadsto (r \in alloc[c])$$

$$ClientsWillReturn \triangleq \forall c \in Clients : unsat[c] = \{\} \leadsto alloc[c] = \{\}$$

$$InfOftenHappy \triangleq \forall c \in Clients : \Box\Diamond(unsat[c] = \{\})$$

THEOREM $SimpleAllocator \Rightarrow \Box ResourceMutex$

THEOREM $SimpleAllocator \Rightarrow ClientsWillObtain$

THEOREM $SimpleAllocator \Rightarrow ClientsWillReturn$

THEOREM $SimpleAllocator \Rightarrow InfOftenHappy$

We will later see how these theorems can be verified in TLA$^+$

Summary

- specification language based on TLA and set theory
- TLA: stuttering invariant fragment of LTL
- specifications and properties represented as formulas
- TLA$^+$: untyped ZF for specifying data structures
- module structure for hierarchical specifications
- system specification: explicit encoding of transition system
- TLA$^+$ supports different specification styles

Outline

Verification and validation

- Validation: are we building the right system?
  - compare formal model against informal user requirements
  - techniques: animation, prototyping, testing, reviews
- Verification: are we building the system right?
  - use formal techniques to establish properties of a model
  - compare two formal models at different levels of abstraction
  - techniques: theorem proving, model and equivalence checking
- In the following: focus on verification

Verification by model checking and theorem proving

- Algorithmic verification – state space exploration $\mathcal{T} \models \Phi$
  - effectively construct state graph of transition system $\mathcal{T}$
  - use semantic definitions to evaluate property over $\mathcal{T}$
  - effective mechanization for certain properties over finite transition systems (and for some classes of infinite ones)
  - TLC: model checker for finite TLA$^+$ models

- Deductive verification – theorem proving $\mathcal{T} \vdash \Phi$
  - define logic for expressing system properties
  - establish proof rules for relating system and properties
  - implement proof system within a theorem prover
  - TLAPS: interactive proof support for TLA$^+$

Outline

Invariants

Definition

A state formula J is an invariant of a transition system $\mathcal{T} = (Q, I, \delta)$ if $[\![J]\!]_q = \mathrm{tt}$ holds for every reachable state q.

- Invariants: fundamental correctness properties
  - exclusive access of processes to resources
  - non-corruption of messages
  - in-order delivery of messages
- Special case: inductive invariants
  - J is an inductive invariant if $[\![J]\!]_q = \mathrm{tt}$ for every $q \in I$ and if for all $(q, q') \in \delta$, if $[\![J]\!]_q = \mathrm{tt}$ then $[\![J]\!]_{q'} = \mathrm{tt}$
  - Note: second condition required even for unreachable q
  - inductive invariants: basis of deductive system verification
  - some methods (such as B or Z) document inductive invariants

Invariant checking

- Idea of the algorithm
  - enumerate reachable states of $\mathcal{T}$ (finite state!)
  - verify that all reachable states satisfy J
- Pseudo-code

```
boolean verifyInvariant(TransSystem ts, Predicate inv) {
  Set visited = new Set();
  Set todo = ts.getInitials();
  while (!todo.isEmpty()) {
    State s = todo.removeSomeElement()
    if (!visited.contains(s)) {
      if (!s.satisfies(inv))
        return false;
      visited.add(s);
      todo.addAll(ts.getSuccessors(s));
  } }
  return true;
}
```

Invariant checking: implementation

- Depth-first search: stack `todo`
  - stack contains counter-example in case invariant does not hold
  - counter-examples may be unnecessarily long
- Breadth-first search: queue `todo`
  - ensures counter-examples of minimal length
  - counter-example generation needs additional back-pointers
- Problem: large state spaces (beyond 10^6–10^7 states)
  - use disk to store set `visited`
  - compress: store signatures instead of complete states
    $\leadsto$ incomplete search in case of collisions
  - reduce number of states or transitions to explore

Beyond invariant checking

- Basic idea extends to more complicated properties
  - beyond mere reachability (of state violating invariant)
  - apply graph algorithms to determine validity of properties
- Example: "process A will eventually acquire resource"
  - enumerate strongly connected components of state graph (Tarjan)
  - check if in all states of SCC, process A tries to acquire resource without ever holding it
  - complexity linear in size of state graph
  - alternative: "double reachability" search (Courcoubetis et al. 1992)
- On-the-fly model checking
  - properties of runs represented by finite automata over ω-words: $\mathcal{A}$ accepts state sequences that violate property
  - general graph algorithms for solving language emptiness of $\mathcal{T} \times \mathcal{A}$
  - many implementations, e.g. Spin: `http://spinroot.com`

Model checking using TLC

- Define finite instance for model checking
  - instantiate system parameters to fixed values

    ```
    N = 5, Clients = {c1, c2, c3}, ...
    ```

  - indicate formula defining system specification

    ```
    SPECIFICATION SimpleAllocator
    ```

  - indicate properties to verify

    ```
    INVARIANTS TypeInvariant, ResourceMutex
    TEMPORAL ClientsWillObtain, InfOftenHappy
    ```

  - easiest done by creating a “model” in the TLA$^+$ toolbox

Model checking using TLC

- Define finite instance for model checking
  - instantiate system parameters to fixed values

    ```
    N = 5, Clients = {c1, c2, c3}, ...
    ```

  - indicate formula defining system specification

    ```
    SPECIFICATION SimpleAllocator
    ```

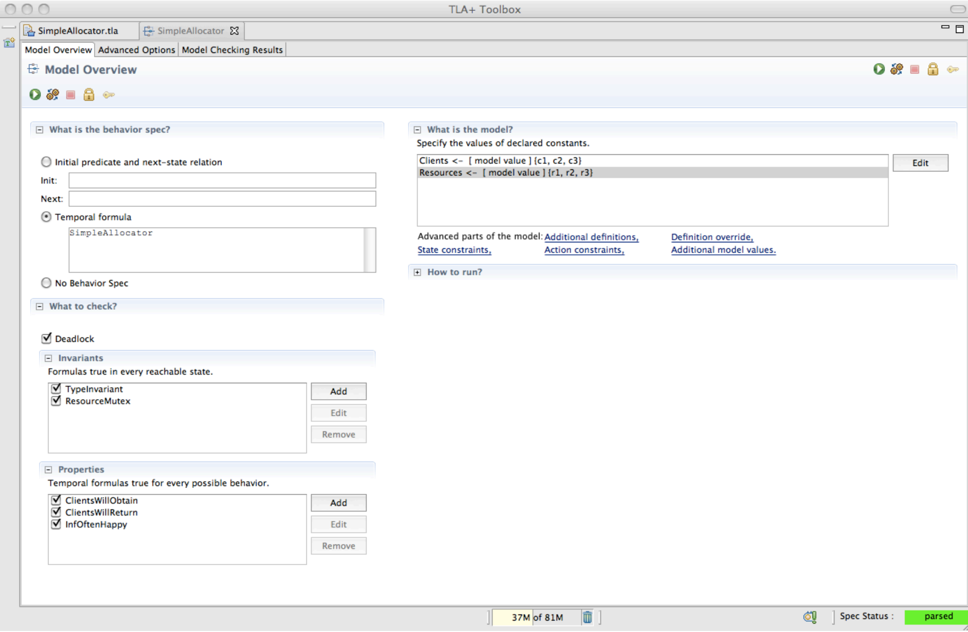
  - indicate properties to verify

    ```
    INVARIANTS TypeInvariant, ResourceMutex
    TEMPORAL ClientsWillObtain, InfOftenHappy
    ```

  - easiest done by creating a "model" in the TLA$^+$ toolbox
- Some limitations of TLC (see Lamport's book for details)
  - all values must be finite (integers, finite sets, arrays, ...)
  - specifications must be in standard form $Init \wedge \Box[Next]_v \wedge L$
  - primed state variables must be "assigned to" at first occurrence

    $x' = [x \text{ EXCEPT } ![3] = @ + 1]$ $\qquad x[3]' = x[3] + 1$

    $v' \in 1..nProcs$ $\qquad 1 \leq v' \wedge v' \leq nProcs$


TLA+ Toolbox
SimpleAllocator.tla
SimpleAllocator
Model Overview
Advanced Options
Model Checking Results
Model Overview
What is the behavior spec?
Initial predicate and next-state relation
Init:
Next:
Temporal formula
SimpleAllocator
No Behavior Spec
What to check?
Deadlock
Invariants
Formulas true in every reachable state.
TypeInvariant
ResourceMutex
Add
Edit
Remove
Properties
Temporal formulas true for every possible behavior.
ClientsWillObtain
ClientsWillReturn
InfOftenHappy
Add
Edit
Remove
What is the model?
Specify the values of declared constants.
Clients <- [model value] {c1, c2, c3}
Resources <- [model value] {r1, r2, r3}
Edit
Advanced parts of the model:
Additional definitions,
Definition override,
State constraints,
Action constraints,
Additional model values.
How to run?
37M of 81M
Spec Status :
parsed

Outline

Principle of deductive verification

- System and properties represented as TLA formulas

system described by *Spec* has property *Prop*
iff
formula *Prop* holds of every run of *Spec*
iff
implication $Spec \Rightarrow Prop$ is valid

- System verification reduces to TLA provability
- No restriction to finite-state specifications
- Now: verification rules for standard correctness properties

Proving invariants

- Invariant: formula $\Box I$ for state predicate I
  - characterize the set of reachable states of a system
  - basis for proving further correctness properties
  - deductive verification relies on inductive invariants
- Basic proof rule: (INV1) $$\frac{I \land Next \Rightarrow I' \quad I \land v' = v \Rightarrow I'}{I \land \Box[Next]_v \Rightarrow \Box I}$$
  - every transition (stuttering or not) preserves I
  - thus, if I holds initially, it will hold forever
- Reduce temporal conclusion to non-temporal hypotheses

Invariant for the hour clock

MODULE *HourClock*

EXTENDS *Naturals*
VARIABLE *hr*

$$
\begin{array}{lcl}
HCini & \triangleq & hr \in 0..23 \\
HCnxt & \triangleq & hr' = \text{IF } hr = 23 \text{ THEN } 0 \text{ ELSE } hr + 1 \\
HC & \triangleq & HCini \land \Box[HCnxt]_{hr} \land \mathrm{WF}_{hr}(HCnxt)
\end{array}
$$

- Prove $HC \Rightarrow \Box HCini$
  - using (INV1) and the definition of HC we must prove

$$
\begin{array}{lcl}
hr \in 0..23 \land HCnxt & \Rightarrow & hr' \in 0..23 \\
hr \in 0..23 \land hr' = hr & \Rightarrow & hr' \in 0..23
\end{array}
$$

  - both implications are clearly valid

Finding inductive invariants

- Rule (INV1) can be used to prove inductive invariants
  - in general, need to strengthen proposed invariant
  - use derived proof rule

$$\text{(INV)} \quad \frac{Init \Rightarrow J \qquad J \wedge (Next \vee v' = v) \Rightarrow J' \qquad J \Rightarrow I}{Init \wedge \Box[Next]_v \Rightarrow \Box I}$$

  - J : inductive invariant that implies I

- Inductive invariants
  - finding inductive invariants is often difficult
  - once found, proving them is easy (in principle)
  - inductive invariants contain algorithmic idea: document them!
  - there are systematic techniques for strengthening invariants

Example: strengthen invariant

MODULE *SyncQueue*

EXTENDS *Sequences*

VARIABLES i, q, o

$$SQInit \triangleq q = \langle\rangle \land i = o$$

$$SQEnq \triangleq i' \neq i \land q' = Append(q, i') \land o' = o$$

$$SQDeq \triangleq q \neq \langle\rangle \land o' = Head(q) \land q' = Tail(q) \land i' = i$$

$$SQLive \triangleq \mathrm{WF}_{i,q,o}(SQDeq)$$

$$SQSpec \triangleq SQInit \land \Box[SQEnq \lor SQDeq]_{i,q,o} \land SQLive$$

- Elements are enqueued only if input changes
- Therefore any two consecutive queue elements are different
- Exercise: formally prove

$$SQSpec \Rightarrow \Box(\forall i \in (1..Len(q) - 1) : q[i] \neq q[i+1])$$

Excursion: weakest preconditions (1)

- For an action A and a state predicate P define

  $$\mathbf{wp}(P, A) \;\triangleq\; \forall v_1', \ldots, v_n' : A \Rightarrow P' \qquad (\equiv \;\; \neg\textsc{Enabled}\,(A \land \neg P'))$$

  where $v_1', \ldots, v_n'$ are all free primed variables in A or P'

- $\mathbf{wp}(P, A)$: weakest precondition of P w.r.t. A
  - true in those states all of whose A-successors satisfy P

- Examples

$$\begin{array}{rcl}
\mathbf{wp}(x = 5, x' = x + 1) & \equiv & \forall x' : x' = x + 1 \Rightarrow x' = 5 \\
 & \equiv & x = 4 \\
\mathbf{wp}(y \in S, S' = S \cup T \land y' = y) & \equiv & \forall y', S' : S' = S \cup T \land y' = y \Rightarrow y' \in S' \\
 & \equiv & y \in S \cup T \\
\mathbf{wp}(x > 0, x' = 0) & \equiv & \forall x' : x' = 0 \Rightarrow x' > 0 \\
 & \equiv & \mathsf{ff}
\end{array}$$

Excursion: weakest preconditions (2)

- Rule (INV) can be rewritten as follows

$$\frac{Init \Rightarrow J \quad J \Rightarrow \mathbf{wp}(J, Next \vee v' = v) \quad J \Rightarrow I}{Init \wedge \Box[Next]_v \Rightarrow \Box I}$$

- Following heuristic can help finding inductive invariants

1. start with invariant to prove: $J := I$
2. try proving $J \wedge A \Rightarrow J'$ for each disjunct A of $[Next]_v$
   if this proof fails, set $J := J \wedge \mathbf{wp}(J, A)$
3. repeat step 2 until
   - ★ either all proofs succeed and $Init \Rightarrow J$ holds:
     $\Rightarrow$ J is an inductive invariant
   - ★ or J is not implied by *Init*:
     $\Rightarrow$ I is not an invariant

- Heuristic need not terminate: generalize candidate invariant

Liveness 1: use fairness conditions

- Fairness conditions ensure that actions occur eventually
- Following rule derives liveness from weak fairness

$$\text{(WF1)}\quad \frac{\begin{array}{c} P \wedge [Next]_v \Rightarrow P' \vee Q' \\ P \wedge \langle Next \wedge A \rangle_v \Rightarrow Q' \\ P \Rightarrow \textsc{Enabled}\ \langle A \rangle_v \end{array}}{\Box[Next]_v \wedge \mathrm{WF}_v(A) \Rightarrow (P \leadsto Q)}$$

- hypotheses of (WF1) are again non-temporal

Examples

- Fairness of *HCnxt* ensures that clock keeps ticking

  $$HC \;\Rightarrow\; \forall k \in 0..23 : hr = k \leadsto hr = (k+1)\%24$$

  - by (WF1) and predicate logic it suffices to show

    $$k \in 0..23 \land hr = k \land [HCnxt]_{hr} \Rightarrow hr' = k \lor hr' = (k+1)\%24$$
    $$k \in 0..23 \land hr = k \land \langle HCnxt \rangle_{hr} \Rightarrow hr' = (k+1)\%24$$
    $$k \in 0..23 \land hr = k \Rightarrow \text{ENABLED } \langle HCnxt \rangle_{hr}$$

  - these formulas are again valid

- Exercise: show that elements advance in queue

  $$SQSpec \Rightarrow \forall k \in 1..Len(q) : \forall x : q[k] = x \leadsto (o = x \lor q[k-1] = x)$$

Correctness of rule (WF1)

Assumptions

1. $\sigma \models \Box[Next]_v \wedge \mathrm{WF}_v(A)$ where $\sigma = s_0 s_1 \ldots$
2. To prove that $\sigma \models P \leadsto Q$ assume that $[\![P]\!]_{s_n} = \mathsf{tt}$ for some $n \in \mathbb{N}$
3. Assume also that $[\![Q]\!]_{s_m} = \mathsf{ff}$ for all $m \geq n$

Show: contradiction

1. $[\![P]\!]_{s_m} = \mathsf{tt}$ for all $m \geq n$
   [induction on $m \geq n$ using A1, A2, and hypothesis $P \wedge [Next]_v \Rightarrow P' \vee Q'$]
2. $[\![\text{ENABLED } \langle A \rangle_v]\!]_{s_m} = \mathsf{tt}$ for all $m \geq n$
   [from (1) and hypothesis $P \Rightarrow \text{ENABLED } \langle A \rangle_v$]
3. $[\![\langle A \rangle_v]\!]_{s_m, s_{m+1}} = \mathsf{tt}$ for some $m \geq n$
   [from (2) and A1: $\mathrm{WF}_v(A)$]
4. $[\![Q]\!]_{s_{m+1}} = \mathsf{tt}$ for some $m \geq n$
   [from (3), (1), A1, and hypothesis $P \wedge \langle Next \wedge A \rangle_v \Rightarrow Q'$]
5. Q.E.D. (contradiction with A3)

Liveness from strong fairness

- For (WF1), the action $\langle A \rangle_v$ must remain enabled forever
  - strong fairness only requires "infinitely often enabled"
  - appropriate proof rule

$$
\text{(SF1)} \quad \frac{\begin{array}{c} P \land [Next]_v \Rightarrow P' \lor Q' \\ P \land \langle Next \land A \rangle_v \Rightarrow Q' \\ \Box P \land \Box[Next]_v \land \Box F \Rightarrow \Diamond \text{ENABLED}\ \langle A \rangle_v \end{array}}{\Box[Next]_v \land \text{SF}_v(A) \land \Box F \Rightarrow (P \leadsto Q)}
$$

- Observations
  - first two hypotheses are as in (WF1)
  - third hypothesis is a temporal formula: F can be a conjunction of
    - fairness conditions: $\text{WF}_v(B) \equiv \Box\text{WF}_v(B)$, $\text{SF}_v(B) \equiv \Box\text{SF}_v(B)$
    - auxiliary "leadsto" formulas, invariants, ...

Complex liveness properties

- Rules (WF1) and (SF1) prove elementary liveness properties
  - clock eventually displays next hour
  - elements in queue will advance by one position
- How can we prove more complex properties?
  - clock will eventually display 12 o'clock

    $HC \Rightarrow \Box\Diamond(hr = 12)$
  - enqueued values will eventually be output

    $SQSpec \Rightarrow \big((\exists k \in 1..Len(q) : q[k] = x) \leadsto o = x\big)$
- Informal argument: repeat elementary liveness property
  - every tick of the clock brings us closer to noon
  - every output action moves the element closer to the output channel

Well-founded relations

Definition

A binary relation $\prec \subseteq D \times D$ is well-founded if there is no infinite descending chain $d_0 \succ d_1 \succ d_2 \succ \ldots$ of elements $d_i \in D$.

- Observations
  - well-founded relations are irreflexive and asymmetric
  - every non-empty subset of D contains a minimal element
- Examples
  - $<$ is well-founded over $\mathbb{N}$ (also over ordinal numbers)
  - lexicographic ordering on sequences over bounded size
  - lexicographic ordering on $Seq(\{a, b\})$ is not well-founded:

    $b \succ ab \succ aab \succ aaab \succ \ldots$

Liveness from well-founded relations

- Additional proof rule

$$(\text{WFO})\quad \frac{\begin{array}{c}(D,\prec)\ \text{well-founded}\\ F \wedge d \in D \Rightarrow \big(H(d)\ \leadsto\ G \vee (\exists e \in D : e \prec d \wedge H(e))\big)\end{array}}{F \Rightarrow \big((\exists d \in D : H(d)) \leadsto G\big)}$$

(d does not occur in G)

- Observations
  - must prove another "leadsto" property, typically based on fairness
  - first premise should be verified by reasoning about data

- Exercise: prove the correctness of rule (WFO)

Example

Prove $HC \Rightarrow ((\exists d \in 0..23 : hr = d) \leadsto hr = 12)$

- Define the well-founded relation $\prec$ on 0..23 by

$$\begin{array}{lcll} dist(d) & \triangleq & \text{IF } d \leq 12 \text{ THEN } 12 - d \text{ ELSE } 36 - d & \text{(* distance from 12 *)} \\ d \prec e & \triangleq & dist(d) < dist(e) & \end{array}$$

- Applying (WFO) we have to prove

$$HC \land d \in 0..23 \Rightarrow (hr = d \leadsto (hr = 12 \lor \exists e \in 0..23 : e \prec d \land hr = e))$$

- This follows from the formula

$$HC \;\Rightarrow\; \forall k \in (0..23) : hr = k \leadsto hr = (k+1) \mod 24$$

shown earlier, using the definition of $\prec$

Combining properties: simple temporal logic

Application of verification rules is supported by the laws of

- first-order logic,
- theories formalizing the data (in particular, set theory),
- and laws of temporal logic such as

$$\text{(STL1)}\quad \frac{F}{\Box F} \qquad\qquad \text{(STL2)}\quad \Box F \Rightarrow F$$

$$\text{(STL3)}\quad \Box\Box F \equiv \Box F \qquad\qquad \text{(STL4)}\quad \Box(F \Rightarrow G) \Rightarrow (\Box F \Rightarrow \Box G)$$

$$\text{(STL5)}\quad \Box(F \wedge G) \equiv (\Box F \wedge \Box G) \qquad\qquad \text{(STL6)}\quad \Diamond\Box(F \wedge G) \equiv \Diamond\Box F \wedge \Diamond\Box G$$

$$\text{(TLA1)}\quad \frac{P \wedge t' = t \Rightarrow P'}{\Box P \equiv P \wedge \Box[P \Rightarrow P']_t} \qquad\qquad \text{(TLA2)}\quad \frac{I \wedge I' \wedge [A]_t \Rightarrow [B]_u}{\Box I \wedge \Box[A]_t \Rightarrow \Box[B]_u}$$

Validity of propositional temporal logic is mechanically decidable

Proof support: the TLA$^+$ proof system

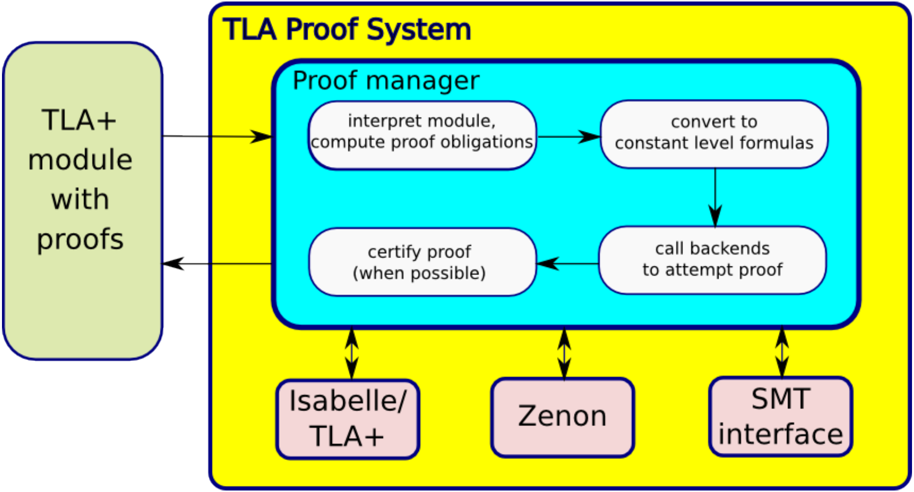
- Assist user in writing proofs
  - hierarchical, declarative proof language
  - skeleton of invariant proof

$$
\begin{array}{l}
\text{THEOREM } \mathit{Spec} \Rightarrow \Box \mathit{Inv} \\
\langle 1 \rangle 1.\ \ \mathit{Init} \Rightarrow \mathit{Inv} \\
\langle 1 \rangle 2.\ \ \text{ASSUME } \mathit{Inv}, \mathit{Next} \\
\qquad\quad \text{PROVE } \ \mathit{Inv}' \\
\langle 1 \rangle 3.\ \ \text{QED} \\
\quad \text{BY } \langle 1 \rangle 1, \langle 1 \rangle 2, \mathit{INV}1 \text{ DEF } \mathit{Spec}
\end{array}
$$

  - proofs of steps $\langle 1 \rangle 1$ and $\langle 1 \rangle 2$ to be filled in

- Verification of leaf proofs by theorem provers
- Philosophy: decompose proof steps until they become "trivial"
- First release: `http://msr-inria.inria.fr/~doligez/tlaps/`

Architecture of the TLAPS



Summary

- model checking: push-button verification of finite instances
- deductive approach: reduce system verification to logical validity
- proof rules for invariants and for liveness properties
- TLA: quickly reduce to non-temporal verification conditions
- machine assistance: theorem proving through the TLAPS

Outline

Modeling "in the large"

- So far: specifications at single level of abstraction
- This chapter:
  - compare models at different levels of abstraction: refinement
  - hiding (encapsulation) of internal state
  - composition from and decomposition into system component
- These concepts are represented in TLA$^+$ by logical connectives

Outline

Reminder: refining the hour clock

MODULE *HourMinuteClock*

EXTENDS *Naturals*, *HourClock*

VARIABLE *min*

$$HMCini \triangleq HCini \land min \in 0..59$$

$$Min \triangleq min' = \text{IF } min = 59 \text{ THEN } 0 \text{ ELSE } min + 1$$

$$Hr \triangleq \text{IF } min = 59 \text{ THEN } HCnxt \text{ ELSE } hr' = hr$$

$$HMCnxt \triangleq Min \land Hr$$

$$HMC \triangleq HMCini \land \Box[HMCnxt]_{\langle hr,min \rangle} \land \mathrm{WF}_{\langle hr,min \rangle}(HMCnxt)$$

THEOREM $HMC \Rightarrow HC$

- Every run of *HMC* also satisfies the hour-clock specification
- Stuttering invariance essential for allowing minute ticks

Proving the implication $HMC \Rightarrow HC$

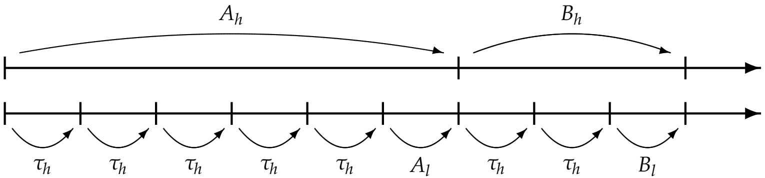
- Initial condition $\quad HMCini \Rightarrow HCini$
  - obvious from definition of $HMCini$
- Next-state relation $\quad HMC \Rightarrow \Box[HCnxt]_{hr}$
  - easy by definition of $HMCnxt$, formally supported by rule (TLA2)
- Fairness condition $\quad HMC \Rightarrow \mathrm{WF}_{hr}(HCnxt)$
  - unfold definition of $\mathrm{WF}_{hr}(HCnxt)$ or use derived proof rule

$$\text{(WF2)} \quad \frac{\begin{array}{c}\langle N \land P \land A\rangle_v \Rightarrow \langle B\rangle_w \\ P \land \textsc{Enabled}\ \langle B\rangle_w \Rightarrow \textsc{Enabled}\ \langle A\rangle_v \\ \Box[N \land [\neg B]_w]_v \land \mathrm{WF}_v(A) \land \Box F \land \Diamond\Box\textsc{Enabled}\ \langle B\rangle_w \Rightarrow \Diamond\Box P\end{array}}{\Box[N]_v \land \mathrm{WF}_v(A) \land \Box F \Rightarrow \mathrm{WF}_w(B)}$$

- Exercise: verify the refinement using TLC

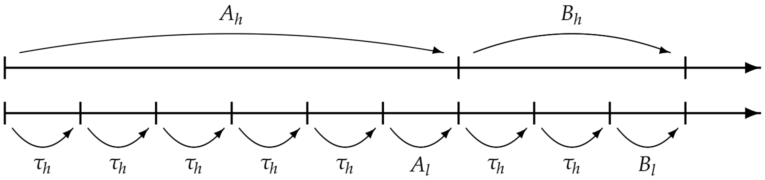
Refinement as implication (trace inclusion)

- Refinement may add state variables ("implementation detail")
- High-level actions implemented by several low-level steps
  - actions except last one do not affect high-level state
  - final action corresponds to high-level effect



Refinement as implication (trace inclusion)

- Refinement may add state variables ("implementation detail")
- High-level actions implemented by several low-level steps
  - actions except last one do not affect high-level state
  - final action corresponds to high-level effect



- Stuttering invariance is crucial for refinement as implication
  - must also preserve high-level fairness conditions
  - implementation inherits all high-level properties
- Implementation may reduce non-determinism
  - branching structure of high-level model need not be preserved

Resource allocator revisited

- Consider the following scenario
  - clients c_1 and c_2 request resources r_1 and r_2
  - allocator gives r_1 to c_1 and r_2 to c_2
- Is the system deadlocked?
- Why didn't we see the problem when running TLC?

Resource allocator revisited

- Consider the following scenario
  - clients c_1 and c_2 request resources r_1 and r_2
  - allocator gives r_1 to c_1 and r_2 to c_2
- Is the system deadlocked?
  - No: c_1 may return the resource r_1, unblocking the system
- Why didn't we see the problem when running TLC?
  - in fact, c_1 must eventually return r_1 due to the fairness condition

    $$\forall c \in \mathit{Clients} : \mathrm{WF}_{\mathit{vars}}(\mathit{Return}(c, \mathit{alloc}[c]))$$
  - therefore, all required liveness properties hold in the model

Resource allocator revisited

- Consider the following scenario
  - clients c_1 and c_2 request resources r_1 and r_2
  - allocator gives r_1 to c_1 and r_2 to c_2
- Is the system deadlocked?
  - No: c_1 may return the resource r_1, unblocking the system
- Why didn't we see the problem when running TLC?
  - in fact, c_1 must eventually return r_1 due to the fairness condition

    $$\forall c \in \mathit{Clients} : \mathrm{WF}_{\mathit{vars}}(\mathit{Return}(c, \mathit{alloc}[c]))$$
  - therefore, all required liveness properties hold in the model
- The specification *SimpleAllocator* is wrong!

Weakening the fairness condition

- Intended fairness requirement (cf. informal description)
  - clients must eventually return all resources it holds
    after having received all resources it requested
  - this can be expressed in TLA$^+$ as

    $$\forall c \in Clients : \mathrm{WF}_{vars}(unsat[c] = \{\} \wedge Return(c, alloc[c]))$$

- With this weaker condition the liveness properties no longer hold
  - TLC reports the expected counter-example

Weakening the fairness condition

- Intended fairness requirement (cf. informal description)
  - clients must eventually return all resources it holds
    after having received all resources it requested
  - this can be expressed in TLA$^+$ as

$$\forall c \in Clients : \mathrm{WF}_{vars}(unsat[c] = \{\} \land Return(c, alloc[c]))$$

- With this weaker condition the liveness properties no longer hold
  - TLC reports the expected counter-example
- Lessons learnt
  1. specifying fairness can be tricky
  2. verification is not a panacea: specifications need to be validated
- How can we correct the specification?

Resource allocator: second solution

- Idea: allocator maintains a schedule of clients to satisfy
  - ensure that all requests can be satisfied, even in the worst case

- Principle of operation
  - allocate resource r to client c only if c appears in the schedule and no client c' that is scheduled before c requests r
  - upon receiving a new request from client c, put c into a pool of clients awaiting scheduling
  - eventually append clients from the pool to the schedule (in arbitrary order: can be refined later)

- In the following: formulation in TLA$^+$

Scheduling allocator in TLA$^+$ (1/5)

MODULE *SchedulingAllocator*

EXTENDS *FiniteSets*, *Sequences*, *Naturals*

CONSTANTS *Clients*, *Resources*
ASSUME *IsFiniteSet*(*Resources*)

VARIABLES
unsat, (* *unsat*[*c*] denotes the outstanding requests of client *c* *)
alloc, (* *alloc*[*c*] denotes the resources allocated to client *c* *)
pool, (* clients with unsatisfied requests that have not been scheduled *)
sched (* schedule (sequence of clients) *)

$$
\begin{array}{l}
TypeInvariant \triangleq \\
\quad \land\ unsat \in [Clients \rightarrow \text{SUBSET}\ Resources] \\
\quad \land\ alloc \in [Clients \rightarrow \text{SUBSET}\ Resources] \\
\quad \land\ pool \in \text{SUBSET}\ Clients \\
\quad \land\ sched \in Seq(Clients)
\end{array}
$$

Scheduling allocator in TLA$^+$ (2/5)

$PermSeqs(S) \triangleq$ (* Permutation sequences of finite set S. *)
$\text{LET } perms[ss \in \text{SUBSET } S] \triangleq$
$\quad \text{IF } ss = \{\} \text{ THEN } \langle\rangle$
$\quad \text{ELSE LET } ps \triangleq [x \in ss \mapsto \{Append(sq, x) : sq \in perms[ss \setminus \{x\}]\}]$
$\quad\quad \text{IN } \text{UNION } \{ps[x] : x \in ss\}$
$\text{IN } perms[S]$

$Drop(seq,i) \triangleq$ (* remove element at position i from sequence seq *)
$\quad SubSeq(seq, 1, i-1) \circ SubSeq(seq, i+1, Len(seq))$

$available \triangleq$ (* set of resources free for allocation *)
$\quad Resources \setminus (\text{UNION } \{alloc[c] : c \in Clients\})$

Scheduling allocator in TLA$^+$ (3/5)

$$
\begin{array}{l}
Init \triangleq \\
\quad \land\ unsat = [Clients \rightarrow \{\}] \land alloc = [Clients \rightarrow \{\}] \\
\quad \land\ pool = \{\} \land sched = \langle\rangle
\end{array}
$$

$Request(c,S) \triangleq$ (* Client c requests set S of resources. *)

$$
\begin{array}{l}
\quad \land\ unsat[c] = \{\} \land alloc[c] = \{\} \\
\quad \land\ S \neq \{\} \land unsat' = [unsat \text{ EXCEPT } ![c] = S] \\
\quad \land\ pool' = pool \cup \{c\} \\
\quad \land\ \text{UNCHANGED } \langle alloc, sched\rangle
\end{array}
$$

$Return(c,S) \triangleq$ (* Client c returns a set of resources that it holds. *)

$$
\begin{array}{l}
\quad \land\ S \neq \{\} \land S \subseteq alloc[c] \\
\quad \land\ alloc' = [alloc \text{ EXCEPT } ![c] = @ \setminus S] \\
\quad \land\ \text{UNCHANGED } \langle unsat, pool, sched\rangle
\end{array}
$$

Scheduling allocator in TLA$^+$ (4/5)

$Schedule \triangleq$ (* Extend allocator schedule by the processes from the pool. *)

$\land\ pool \neq \{\}$

$\land\ \exists sq \in PermSeqs(pool) : sched' = sched \circ sq$

$\land\ pool' = \{\}$

$\land\ \text{UNCHANGED } \langle unsat, alloc \rangle$

$Allocate(c,S) \triangleq$ (* Allocate set S of available resources to client c. *)

$\land\ S \neq \{\} \land S \subseteq available \cap unsat[c]$

$\land\ \exists i \in 1..Len(sched) :$

$\qquad \land\ sched[i] = c$

$\qquad \land\ \forall j \in 1..i-1 : unsat[sched[j]] \cap S = \{\}$

$\qquad \land\ sched' = \text{IF } S = unsat[c] \text{ THEN } Drop(sched, i) \text{ ELSE } sched$

$\land\ alloc' = [alloc \text{ EXCEPT } ![c] = @ \cup S]$

$\land\ unsat' = [unsat \text{ EXCEPT } ![c] = @ \setminus S]$

$\land\ \text{UNCHANGED } pool$

Scheduling allocator in TLA$^+$ (5/5)

$$
\begin{array}{l}
Next \triangleq \qquad \text{(* The next-state relation. *)} \\
\quad \lor \exists c \in Clients, S \in \text{SUBSET } Resources : \\
\qquad\qquad Request(c, S) \lor Allocate(c, S) \lor Return(c, S) \\
\quad \lor Schedule
\end{array}
$$

$$
vars \triangleq \langle unsat, alloc, pool, sched \rangle
$$

$$
\begin{array}{l}
SchedulingAllocator \triangleq \\
\qquad \land Init \land \Box[Next]_{vars} \\
\qquad \land \forall c \in Clients : \mathrm{WF}_{vars}(unsat[c] = \{\} \land Return(c, alloc[c])) \\
\qquad \land \forall c \in Clients : \mathrm{WF}_{vars}(\exists S \in \text{SUBSET } Resources : Allocate(c, S)) \\
\qquad \land \mathrm{WF}_{vars}(Schedule)
\end{array}
$$

Verifying the scheduling allocator

- Scheduling allocator satisfies all correctness requirements
- Crucial invariant
  - full request of any scheduled client can be satisfied from the resources that will be available after all previously scheduled clients released all resources they held or requested

$$\begin{array}{l} \forall i \in 1..Len(sched) : unsat[sched[i]] \subseteq \\ \qquad available \\ \qquad \cup \text{UNION } \{unsat[sched[j]] \cup alloc[sched[j]] : j \in 1..i-1\} \\ \qquad \cup \text{UNION } \{alloc[c] : c \in Clients\} \end{array}$$

Verifying the scheduling allocator

- Scheduling allocator satisfies all correctness requirements
- Crucial invariant
  - full request of any scheduled client can be satisfied from the resources that will be available after all previously scheduled clients released all resources they held or requested

$$\begin{array}{l}\forall i \in 1..Len(sched) : unsat[sched[i]] \subseteq \\ \qquad available \\ \qquad \cup \text{UNION } \{unsat[sched[j]] \cup alloc[sched[j]] : j \in 1..i-1\} \\ \qquad \cup \text{UNION } \{alloc[c] : c \in Clients\}\end{array}$$

- Successful verification using TLC (for small instances)
- Do you believe that the model is now correct?

Scheduling allocator as a refinement

- In fact, *SchedulingAllocator* refines *SimpleAllocator*

```
MODULE AllocatorRefinement
EXTENDS SchedulingAllocator
Simple ≜ INSTANCE SimpleAllocator
THEOREM SchedulingAllocator ⇒ Simple!SimpleAllocator
```

  - the weaker fairness conditions of the scheduling allocator suffice thanks to scheduling
  - the allocator does no longer require the cooperation of the clients for ensuring that they will eventually receive all resources

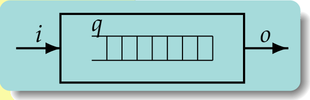
- Correctness properties are inherited by *SchedulingAllocator*

Outline

Reminder: specification of a FIFO component

MODULE *SyncQueue*

EXTENDS *Sequences*
VARIABLES i, q, o



$$SQInit \triangleq q = \langle\rangle \land i = o$$
$$SQEnq \triangleq i' \neq i \land q' = Append(q, i') \land o' = o$$
$$SQDeq \triangleq q \neq \langle\rangle \land o' = Head(q) \land q' = Tail(q) \land i' = i$$
$$SQLive \triangleq \mathrm{WF}_{i,q,o}(SQDeq)$$
$$SQSpec \triangleq SQInit \land \Box[SQEnq \lor SQDeq]_{i,q,o} \land SQLive$$

- The internal queue q is an "implementation detail"
  - it should not be part of the interface
  - FIFO component should function as if there were a queue
  - do not expose implementation detail, especially for refinement

Hiding in TLA$^+$

- TLA uses existential quantification to represent hiding

```
──────────── MODULE FIFO ────────────
VARIABLES i, o
Queue(q) ≜ INSTANCE SyncQueue
─────────────────────────────────────
FIFO ≜ ∃ q : Queue(q)!SQSpec
─────────────────────────────────────
```

  - module *FIFO* contains the “external” specification of the queue
  - separate interface and implementation

Hiding in TLA$^+$

- TLA uses existential quantification to represent hiding

```
---------------- MODULE FIFO ----------------
VARIABLES i, o
Queue(q) ≜ INSTANCE SyncQueue
----------------------------------------------
FIFO ≜ ∃∃ q : Queue(q)!SQSpec
==============================================
```

  - module *FIFO* contains the "external" specification of the queue
  - separate interface and implementation

- Extend syntax of TLA formulas
  - if F is a formula and v a state variable then $\exists\!\!\exists\, v : F$ is a formula
  - intuitive meaning: $\sigma \models \exists\!\!\exists\, v : F$ if $\tau \models F$ for some τ that differs from σ only in the valuations of v

Naive semantics of quantification

- First attempt at defining semantics

$$\sigma, \xi \models \exists\, v : F \quad \text{iff} \quad \tau, \xi \models F \text{ for some } \tau \text{ s.t. } \tau_n =_v \sigma_n \text{ for all } n \in \mathbb{N}$$

Naive semantics of quantification

- First attempt at defining semantics

  $$\sigma, \xi \models \exists\!\!\exists\, v : F \quad \text{iff} \quad \tau, \xi \models F \;\text{ for some } \tau \text{ s.t. } \tau_n =_v \sigma_n \text{ for all } n \in \mathbb{N}$$

- Consider $F \triangleq v = 0 \wedge \Box[v = 1]_w$

τ				
v	0	1	1	$\cdots$
w	0	0	1	$\cdots$

σ				
v	0	0	1	$\cdots$
w	0	0	1	$\cdots$

  - F essentially asserts that v has changed value before w changes
  - clearly, $\tau \models F$, and therefore $\sigma \models \exists\!\!\exists\, v : F$
  - however, $\exists\!\!\exists\, v : F$ would not hold of σ with second state removed

- Violation of stuttering invariance

Formal semantics of quantification

- Solution: define semantics with stuttering invariance "built in"

  $\sigma, \xi \models \exists\!\!\exists\, v : F \quad$ iff $\quad$ there exist $\rho \approx \sigma$ and $\tau =_v \rho$ s.t. $\tau, \xi \models F$

  - both σ and its variant satisfy $\exists\!\!\exists\, v : v = 0 \land \Box[v = 1]_w$
  - in fact, this formula is valid

Formal semantics of quantification

- Solution: define semantics with stuttering invariance "built in"

  $$\sigma, \xi \models \exists\, v : F \quad \text{iff} \quad \text{there exist } \rho \approx \sigma \text{ and } \tau =_v \rho \text{ s.t. } \tau, \xi \models F$$

  - both σ and its variant satisfy $\exists\, v : v = 0 \land \Box[v = 1]_w$
  - in fact, this formula is valid

- Clock example: have $HC \Rightarrow \exists\, min : HMC$

  - implementation of hour clock may contain hidden minute hand

- Verification: standard proof rules are sound

  $(\exists\text{-I}) \quad F(t) \Rightarrow \exists\, v : F(v) \quad$ (t state function: "refinement mapping")

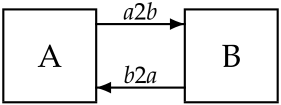
  $$(\exists\text{-E}) \quad \frac{F \Rightarrow G}{(\exists\, v : F) \Rightarrow G} \quad (v \text{ not free in } G)$$

  - for completeness, need more introduction rules
  - TLC does not support $\exists$: provide explicit refinement mapping

Outline

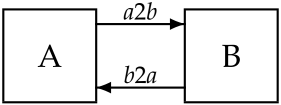
1. Motivation and introductory example
2. Transition systems and their properties
3. System Specification in TLA$^+$
4. Verification of TLA$^+$ models
5. Structuring models: refinement and (de)composition
   - Refinement of specifications
   - Hiding (encapsulation) of internal state
   - Composition and decomposition

Parallel composition of components



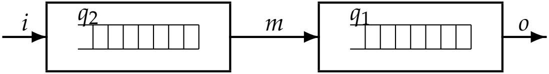
- Assume: components A and B modeled by *ASpec* and *BSpec*
  - interface represented by the shared variables *a2b* and *b2a*
  - both specifications accomodate changes due to other component
  - assume: internal variables hidden (or renamed to be disjoint)

Parallel composition of components



- Assume: components A and B modeled by *ASpec* and *BSpec*
  - interface represented by the shared variables *a2b* and *b2a*
  - both specifications accomodate changes due to other component
  - assume: internal variables hidden (or renamed to be disjoint)
- System containing both components must satisfy $ASpec \wedge BSpec$
  - stuttering invariance is again important here
- System model may require additional constraints
  - typical example: interleaving assumption

Example: composition of two FIFO channels



MODULE *TwoQueues*

VARIABLES i, m, o

$LeftFIFO \triangleq$ INSTANCE $FIFO$ WITH $o \leftarrow m$

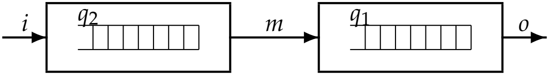
$RightFIFO \triangleq$ INSTANCE $FIFO$ WITH $i \leftarrow m$

$LongFIFO \triangleq LeftFIFO!FIFO \wedge RightFIFO!FIFO$

$Single \triangleq$ INSTANCE $FIFO$

THEOREM $LongFIFO \Rightarrow Single!FIFO$??

Example: composition of two FIFO channels



MODULE *TwoQueues*

VARIABLES i, m, o

$LeftFIFO \triangleq$ INSTANCE $FIFO$ WITH $o \leftarrow m$

$RightFIFO \triangleq$ INSTANCE $FIFO$ WITH $i \leftarrow m$

$LongFIFO \triangleq LeftFIFO!FIFO \land RightFIFO!FIFO$

$Single \triangleq$ INSTANCE $FIFO$

THEOREM $LongFIFO \Rightarrow Single!FIFO$??

- *LongFIFO* allows for simultaneous enqueueing and dequeueing
  - *Single*!*FIFO* does not, due to interleaving assumption
  - assert interleaving when composing

Interleaving composition

- Interleaving composition: add a "synchronizer"

$$\text{THEOREM } \mathit{LongFIFO} \land \Box[i' = i \lor o' = o]_{i,o} \Rightarrow \mathit{Spec}!\mathit{FIFO} \quad \text{where}$$

$$\begin{array}{lcl} \mathit{LongFIFO} & \equiv & \land\ \exists\, q : \mathit{SyncQueue}(i, q, m)!\mathit{SQSpec} \\ & & \land\ \exists\, q : \mathit{SyncQueue}(m, q, o)!\mathit{SQSpec} \\ \mathit{Spec}!\mathit{FIFO} & \equiv & \exists\, q : \mathit{SyncQueue}(i, q, o)!\mathit{SQSpec} \end{array}$$

Interleaving composition

- Interleaving composition: add a "synchronizer"

  $$\text{THEOREM } \mathit{LongFIFO} \land \Box[i' = i \lor o' = o]_{i,o} \Rightarrow \mathit{Spec!FIFO} \quad \text{where}$$

  $$\begin{aligned} \mathit{LongFIFO} &\equiv \land\ \exists\, q : \mathit{SyncQueue}(i,q,m)!\mathit{SQSpec} \\ &\ \land\ \exists\, q : \mathit{SyncQueue}(m,q,o)!\mathit{SQSpec} \\ \mathit{Spec!FIFO} &\equiv \exists\, q : \mathit{SyncQueue}(i,q,o)!\mathit{SQSpec} \end{aligned}$$

- Proof (outline)

  - using rules ($\exists$-E) and ($\exists$-I) it is enough to prove

    $$\begin{aligned} &\land\ \mathit{SyncQueue}(i,q_2,m)!\mathit{SQSpec} \land \mathit{SyncQueue}(m,q_1,o)!\mathit{SQSpec} \\ &\land\ \Box[i' = i \lor o' = o]_{i,o} \\ &\Rightarrow \mathit{SyncQueue}(i,t,o)!\mathit{SQSpec} \end{aligned}$$

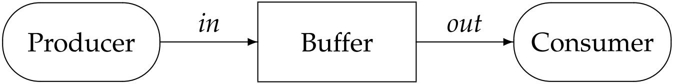
    for some state function t. Choose $t \triangleq q_1 \circ q_2$.

- Exercise: formally carry out this proof

Decomposition of system specification

- Obtain component specifications from high-level system model
  - more in line with refinement approach to system development
  - technically often simpler than composing independent components
- Partition system state & identify component responsibilities
  - assume that no variable can be written by two different components
  - variables can be read by any number of components
  - generalization possible, but technically more involved
- Example: development of a producer-consumer system

High-level specification



- First abstraction: only specify the buffer
  - ignore nature of products and how they are produced/consumed
  - do not specify communication between buffer and environment
- Assume FIFO buffer of fixed capacity

High-level specification in TLA$^+$

MODULE *AbstractPC*

EXTENDS *Sequences*
CONSTANTS *Product*, *None*, *buffCap*
VARIABLES *buffer*, *in*, *out*

$$Init \triangleq buffer = \langle\rangle \land in = None \land out = None$$

$$\begin{aligned} Get \triangleq\ & \land Len(buffer) < buffCap \\ & \land buffer' = Append(buffer, in) \\ & \land \text{UNCHANGED } \langle in, out\rangle \end{aligned}$$

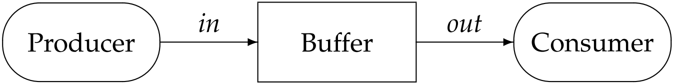
$$\begin{aligned} Put \triangleq\ & \land Len(buffer) > 0 \\ & \land out' = Head(buffer) \land buffer' = Tail(buffer) \\ & \land \text{UNCHANGED } in \end{aligned}$$

$$vars \triangleq \langle in, buffer, out\rangle$$

$$Spec \triangleq Init \land \Box[Get \lor Put]_{vars} \land \mathrm{WF}_{vars}(Put)$$

THEOREM $Spec \Rightarrow \Box(\Diamond\langle Get\rangle_{vars} \Rightarrow \Diamond\langle Put\rangle_{vars})$

Refinement: introducing communication



- Explicitly model communication between components
  - assume asynchronous communication for distributed system
- Synchronize sender and receiver over channel
  - signal availability of new value and successful reception
  - in particular, block producer when the buffer is full
  - avoid variables written by two different components

Channel specification in TLA$^+$

- Generic channel specification with "split bit" handshake

$$tmt \;\text{———}\; \boxed{ch} \;\text{———}\; rcv$$

  - channel content represented by *ch*, control bits *tmt* and *rcv*
  - channel empty (i.e., value may be sent) when $tmt = rcv$
  - channel full (i.e., value may be received) when $tmt \neq rcv$

MODULE *Channel*

CONSTANTS *Value*, *NoValue*

VARIABLES *ch*, *tmt*, *rcv*

$$
\begin{array}{lcl}
Init & \triangleq & ch = NoValue \land tmt = \text{FALSE} \land rcv = \text{FALSE} \\
Snd(v) & \triangleq & tmt = rcv \land ch' = v \land tmt' = \neg tmt \land rcv' = rcv \\
Rcv & \triangleq & tmt \neq rcv \land rcv' = \neg rcv \land \text{UNCHANGED } \langle ch, tmt \rangle \\
Next & \triangleq & (\exists v \in Value : Snd(v)) \lor Rcv \\
vars & \triangleq & \langle ch, tmt, rcv \rangle \\
Chan & \triangleq & Init \land \Box[Next]_{vars}
\end{array}
$$

Refined specification of producer/consumer (1/2)

$$
\begin{array}{l}
\textsc{module}\ \mathit{RefinedPC} \\
\textsc{extends}\ \mathit{Sequences} \\
\textsc{constants}\ \mathit{Product}, \mathit{None}, \mathit{buffCap} \\
\textsc{variables}\ \mathit{buffer}, \mathit{in}, \mathit{itmt}, \mathit{ircv}, \mathit{out}, \mathit{otmt}, \mathit{orcv} \\
\mathit{InChan} \triangleq \textsc{instance}\ \mathit{Channel}\ \textsc{with} \\
\qquad \mathit{Value} \leftarrow \mathit{Product}, \mathit{NoValue} \leftarrow \mathit{None}, \mathit{ch} \leftarrow \mathit{in}, \mathit{tmt} \leftarrow \mathit{itmt}, \mathit{rcv} \leftarrow \mathit{ircv} \\
\mathit{OutChan} \triangleq \textsc{instance}\ \mathit{Channel}\ \textsc{with} \\
\qquad \mathit{Value} \leftarrow \mathit{Product}, \mathit{NoValue} \leftarrow \mathit{None}, \mathit{ch} \leftarrow \mathit{out}, \mathit{tmt} \leftarrow \mathit{otmt}, \mathit{rcv} \leftarrow \mathit{orcv}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{Init} & \triangleq & \mathit{buffer} = \langle\rangle \land \mathit{InChan}!\mathit{Init} \land \mathit{OutChan}!\mathit{Init} \\
\mathit{Produce}(v) & \triangleq & \mathit{InChan}!\mathit{Snd}(v) \land \textsc{unchanged}\ \langle \mathit{buffer}, \mathit{OutChan}!\mathit{vars} \rangle \\
\mathit{Get} & \triangleq & \land\ \mathit{Len}(\mathit{buffer}) < \mathit{buffCap} \\
 & & \land\ \mathit{InChan}!\mathit{Rcv} \land \mathit{buffer}' = \mathit{Append}(\mathit{buffer}, \mathit{in}) \\
 & & \land\ \textsc{unchanged}\ \mathit{OutChan}!\mathit{vars} \\
\mathit{Put} & \triangleq & \land\ \mathit{Len}(\mathit{buffer}) > 0 \\
 & & \land\ \mathit{OutChan}!\mathit{Snd}(\mathit{Head}(\mathit{buffer})) \land \mathit{buffer}' = \mathit{Tail}(\mathit{buffer}) \\
 & & \land\ \textsc{unchanged}\ \mathit{InChan}!\mathit{vars} \\
\mathit{Consume} & \triangleq & \mathit{OutChan}!\mathit{Rcv} \land \textsc{unchanged}\ \langle \mathit{buffer}, \mathit{InChan}!\mathit{vars} \rangle
\end{array}
$$

Refined specification of producer/consumer (2/2)

$$
\begin{array}{lcl}
\mathit{Next} & \triangleq & (\exists v \in \mathit{Product} : \mathit{Produce}(v)) \lor \mathit{Get} \lor \mathit{Put} \lor \mathit{Consume} \\
\mathit{vars} & \triangleq & \langle \mathit{buffer}, \mathit{InChan}!\mathit{vars}, \mathit{OutChan}!\mathit{vars} \rangle \\
\mathit{Fairness} & \triangleq & \mathrm{WF}_{\mathit{vars}}(\mathit{Get}) \land \mathrm{WF}_{\mathit{vars}}(\mathit{Put}) \land \mathrm{WF}_{\mathit{vars}}(\mathit{Consume}) \\
\mathit{Spec} & \triangleq & \mathit{Init} \land \Box[\mathit{Next}]_{\mathit{vars}} \land \mathit{Fairness}
\end{array}
$$

LOCAL INSTANCE *AbstractPC*
THEOREM $\mathit{Spec} \Rightarrow \mathit{AbstractPC}!\mathit{Spec}$

- Proof of refinement theorem
  - safety part quite obvious
  - in particular, new actions do not modify variable *buffer*
  - fairness condition on new *Consume* action necessary for implementing high-level fairness constraint on *Put*

- TLC can again verify refinement for small instances

Decomposition: partition variables and actions

- Decompose system into three components
  - identify variables accessed by each component
  - assign responsibilities for performing actions to components

	Producer	Buffer	Consumer
writes	*in*, *itmt*	*ircv*, *buffer*, *out*, *otmt*	*orcv*
reads	*ircv*	*in*, *itmt*, *orcv*	*out*, *otmt*
performs	*Produce*	*Get*, *Put*	*Consume*

- Derive component specifications from this decomposition
  - no variable written by two different components
  - unique assignment of actions to components

Producer and Consumer components in TLA$^+$

MODULE *Producer*

$$
\begin{array}{l}
\textsc{constants}\ \mathit{Product}, \mathit{None} \\
\textsc{variables}\ \mathit{in}, \mathit{itmt}, \mathit{ircv} \\
\mathit{PInit} \triangleq \mathit{in} = \mathit{None} \land \mathit{itmt} = \text{FALSE} \\
\mathit{Produce}(v) \triangleq \begin{array}{l} \land\ \mathit{itmt} = \mathit{ircv} \\ \land\ \mathit{in}' = v \land \mathit{itmt}' = \neg \mathit{itmt} \land \mathit{ircv}' = \mathit{ircv} \end{array} \\
\mathit{Producer} \triangleq \mathit{PInit} \land \Box[\exists v \in \mathit{Product} : \mathit{Produce}(v)]_{\langle \mathit{in}, \mathit{itmt} \rangle}
\end{array}
$$

MODULE *Consumer*

$$
\begin{array}{l}
\textsc{variables}\ \mathit{orcv}, \mathit{out}, \mathit{otmt} \\
\mathit{CInit} \triangleq \mathit{orcv} = \text{FALSE} \\
\mathit{Consume} \triangleq \begin{array}{l} \land\ \mathit{otmt} \neq \mathit{orcv} \\ \land\ \mathit{orcv}' = \neg \mathit{orcv} \land \text{UNCHANGED}\ \langle \mathit{out}, \mathit{otmt} \rangle \end{array} \\
\mathit{Consumer} \triangleq \mathit{CInit} \land \Box[\mathit{Consume}]_{\mathit{orcv}} \land \mathrm{WF}_{\mathit{orcv}}(\mathit{Consume})
\end{array}
$$

Buffer component in TLA$^+$

$$\textsc{module}\ \mathit{Buffer}$$

EXTENDS *Sequences*
CONSTANTS *Product*, *None*, *buffCap*
VARIABLES *in*, *itmt*, *ircv*, *buffer*, *out*, *otmt*, *orcv*

$$\mathit{BInit} \triangleq \mathit{ircv} = \text{FALSE} \land \mathit{buffer} = \langle\rangle \land \mathit{out} = \mathit{None} \land \mathit{otmt} = \text{FALSE}$$

$$\begin{aligned}\mathit{Get} \triangleq\ & \land \mathit{Len}(\mathit{buffer}) < \mathit{buffCap} \land \mathit{itmt} \neq \mathit{ircv} \\ & \land \mathit{buffer}' = \mathit{Append}(\mathit{buffer}, \mathit{in}) \land \mathit{ircv}' = \neg \mathit{ircv} \\ & \land \text{UNCHANGED}\ \langle \mathit{in}, \mathit{itmt}, \mathit{out}, \mathit{otmt}, \mathit{orcv} \rangle\end{aligned}$$

$$\begin{aligned}\mathit{Put} \triangleq\ & \land \mathit{Len}(\mathit{buffer}) > 0 \land \mathit{otmt} = \mathit{orcv} \\ & \land \mathit{out}' = \mathit{Head}(\mathit{buffer}) \land \mathit{otmt}' = \neg \mathit{otmt} \land \mathit{buffer}' = \mathit{Tail}(\mathit{buffer}) \\ & \land \text{UNCHANGED}\ \langle \mathit{in}, \mathit{itmt}, \mathit{ircv}, \mathit{orcv} \rangle\end{aligned}$$

$$\mathit{bvars} \triangleq \langle \mathit{ircv}, \mathit{buffer}, \mathit{out}, \mathit{otmt} \rangle$$

$$\mathit{Buffer} \triangleq \mathit{BInit} \land \Box[\mathit{Get} \lor \mathit{Put}]_{\mathit{bvars}} \land \mathrm{WF}_{\mathit{bvars}}(\mathit{Get}) \land \mathrm{WF}_{\mathit{bvars}}(\mathit{Put})$$

Summary: decomposition

MODULE *Refinement*

EXTENDS *Producer, Consumer, Buffer*
LOCAL INSTANCE *RefinedPC*
THEOREM $Producer \land Buffer \land Consumer \Rightarrow RefinedPC!Spec$

- Overall approach to decompose system specification
  - partition system variables according to who modifies them
  - identify actions performed by each component
  - declare all variables read or written in component specifications
  - only written variables appear in subscripts (next-state, fairness)
  - add UNCHANGED clauses for read-only variables
- What if no suitable variable partitioning can be found?
  - try to split (auxiliary) variables to separate responsibilities
  - allow for shared actions in different system models

Summary: structuring models

- Refinement: successively add implementation detail
  - represented in TLA as (validity of) implication
  - stuttering invariance allows for additional low-level steps
- Hiding of internal state components: exhibit system interface
  - represented in TLA by existential quantification over state variables
  - non-standard definition ensures stuttering invariance
  - standard proof rules remain sound: refinement mappings
- Composing and decomposing specifications
  - composition represented as conjunction of specifications
  - stuttering invariance allows for local component steps
  - decomposition derives components from system specification

Thank you!