Modeling and Developing Systems Using TLA$^+$

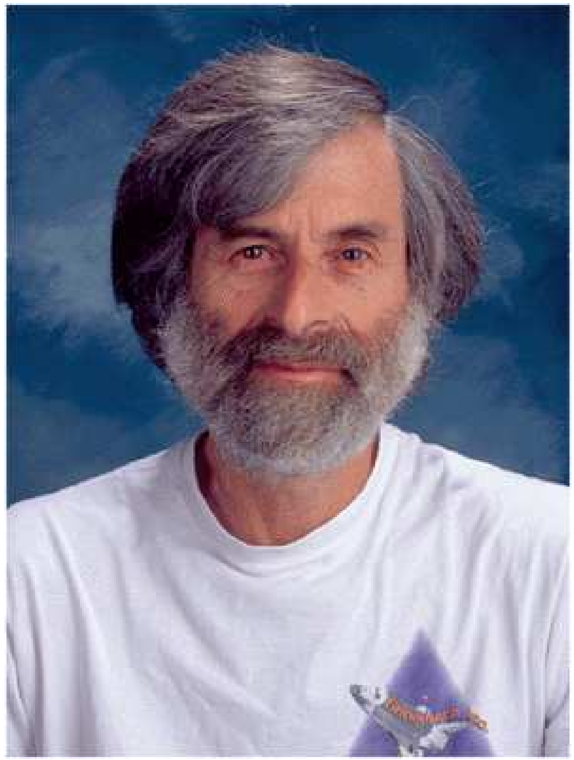
Stephan Merz

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

INRIA Lorraine & LORIA

Nancy, France

Leslie Lamport

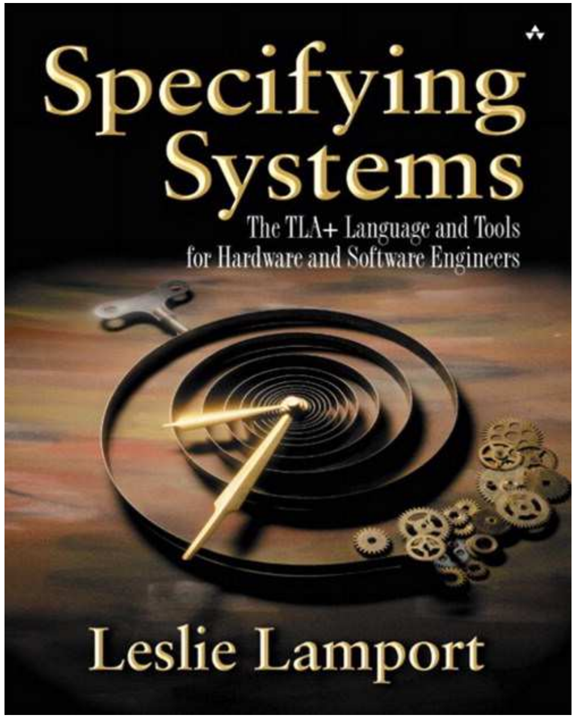


PhD 1972 (Brandeis University), Mathematics
(analytic partial differential equations)

- 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   collected works at `http://www.lamport.org/`

- National Academy of Engineering (1991)
- PODC Influential Paper Award (2000), IEEE Piore Award (2004)
- honorary doctorates (Rennes, Kiel, Lausanne)



TLA^+ specification language

- formal language for describing and reasoning about distributed and concurrent systems
- based on mathematical logic and set theory plus temporal logic TLA
- supported by tool set (TLC model checker)
- Addison-Wesley, 2003
  (free download for personal use)

Plan of these lectures

1. Transition systems and properties of runs

   $\Longrightarrow$ understand the foundations of TLA$^+$

2. System specification in TLA$^+$

   $\Longrightarrow$ read and write TLA$^+$ models

3. System verification

   $\Longrightarrow$ validate models and prove properties

4. System development

   $\Longrightarrow$ concepts of refinement and (de-)composition

5. Case study

   $\Longrightarrow$ experiment on concrete application

1 Motivation & Introduction

Why formal specifications?

- describe, analyze, and reason about systems

  algorithms, protocols, controllers, embedded systems, ...

- at different levels of abstraction

  meaningful to users, system developers, implementors, machine

- support and justify development process

  gradually introduce design decisions/architecture, relate models at different levels

Unlike programs, high-level specifications need not be executable.

- encompass software, hardware, physical environment, users
- starting point: requirements – describe "real world"
- target: executable code – enable efficient execution

Classifications of specification languages

- intended for different classes of systems
  - sequential algorithms
  - interactive systems
  - reactive & distributed systems
  - real-time & hybrid systems
  - security-sensitive systems
- based on different specification styles
  - property oriented or axiomatic: what?

    list desired (correctness) properties (cf. algebra)
  - model-based: how?

    describe system in terms of abstract model (cf. analysis, but add refinement)

TLA^{+} : model-based specification, reactive & distributed systems

TLA⁺ : Informal Introduction

Example 1.1 (an hour clock)

```
---------------------------- MODULE HourClock ----------------------------
EXTENDS Naturals
VARIABLE hr
---------------------------------------------------------------------------
HCini  ≜  hr ∈ (0..23)
HCnxt  ≜  hr' = IF hr = 23 THEN 0 ELSE hr + 1
HCsafe ≜  HCini ∧ □[HCnxt]_hr
---------------------------------------------------------------------------
THEOREM HCsafe ⇒ □HCini
===========================================================================
```

The module *HourClock* contains declarations and definitions

- *hr* a state variable
- *HCini* a state predicate
- *HCnxt* an action (built from *hr* and hr')
- *HCsafe* a temporal formula specifying that
  - the initial state satisfies *HCini*
  - every transition satisfies *HCnxt* or leaves *hr* unchanged

Module *HourClock* also asserts a theorem: $HCsafe \Rightarrow \Box HCini$

This invariant can be verified using TLC, the TLA^+ model checker.

Note:

- the hour clock may eventually stop ticking
- it must not fail in any other way

A TLA formula $\quad Init \wedge \Box[Next]_v$

specifies the initial states and the allowed transitions of a system.

It allows for transitions that do not change v: stuttering transitions.

Infinite stuttering can be excluded by asserting fairness conditions.

For example,

$$HC \triangleq HCini \wedge \Box[HCnxt]_{hr} \wedge \mathrm{WF}_{hr}(HCnxt)$$

specifies an hour clock that never stops ticking.

Fairness conditions assert that some action *A* occurs eventually
— provided it is "sufficiently often" enabled.

Two standard interpretations of "sufficiently often":

weak fairness. *A* occurs eventually if it is persistently enabled after some point

strong fairness. *A* occurs eventually if it is infinitely often enabled after some point

Note: strong fairness is strictly stronger than weak fairness

An action can be infinitely often enabled without being persistently enabled.

Identifying adequate fairness conditions for a system is often non-trivial.

Most TLA system specifications are of the form

$$Init \wedge \Box[Next]_v \wedge L$$

Init : *state formula* describing the initial state(s)

Next : *action formula* formalizing the transition relation

- usually a disjunction $A_1 \vee \ldots \vee A_n$ of possible actions (events) A_i

L : *temporal formula* asserting liveness conditions

- usually a conjunction $\mathrm{WF}_v(A_i) \wedge \ldots \wedge \mathrm{SF}_v(A_j)$ of fairness conditions

TLA system specifications formalize fair transition systems.

2 Fair transition systems, runs, and properties

Transformational systems (sequential algorithms)

- input data $\longrightarrow$ compute $\longrightarrow$ output result
- partial correctness + termination + complexity
- computational model: Turing machines, RAM, term rewriting, ...

Reactive systems (operating systems, controllers, ...)

- environment $\longleftrightarrow$ system
- safety: *something bad never happens*
- liveness: *something good eventually happens*
- computational model: transition systems

2.1 Labeled transition systems

Definition 2.1 A labeled transition system $\mathcal{T} = (Q, I, \mathcal{A}, \delta)$ is given by

- a (finite or infinite) set of *states* Q,
- a set $I \subseteq Q$ of *initial states*,
- a set $\mathcal{A}$ of *actions* (action names), and
- a *transition relation* $\delta \subseteq Q \times \mathcal{A} \times Q$.

An action $A \in \mathcal{A}$ is *enabled* at state $q \in Q$ iff $(q, A, q') \in \delta$ for some $q' \in Q$.

A *run* of $\mathcal{T}$ is a (finite or infinite) sequence $\rho = q_0 \xrightarrow{A_0} q_1 \xrightarrow{A_1} q_2 \ldots$ where $q_0 \in I$ and $(q_i, A_i, q_{i+1}) \in \delta$ holds for all i.

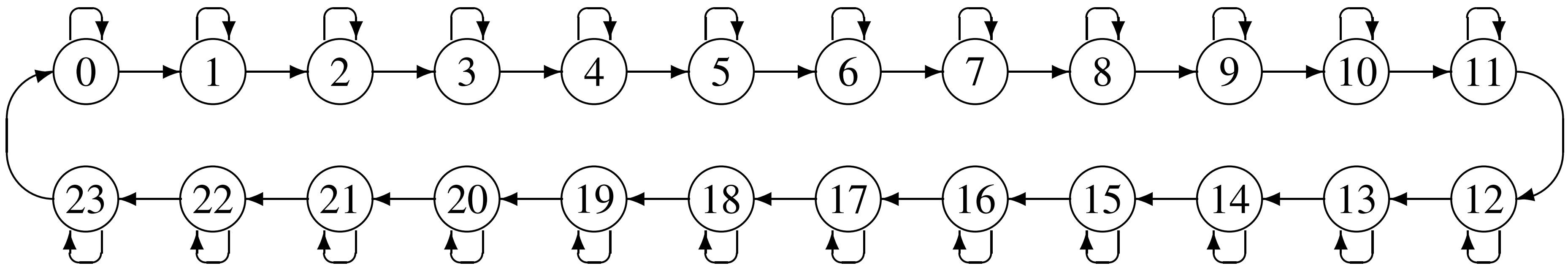
A state $q \in Q$ is *reachable* iff it appears in some run ρ of $\mathcal{T}$.

Convention. We assume that $\mathcal{A}$ contains a special "stuttering" action τ with $(q, \tau, q') \in \delta$ iff $q' = q$. Every finite run can then be extended to an infinite run by "infinite stuttering".

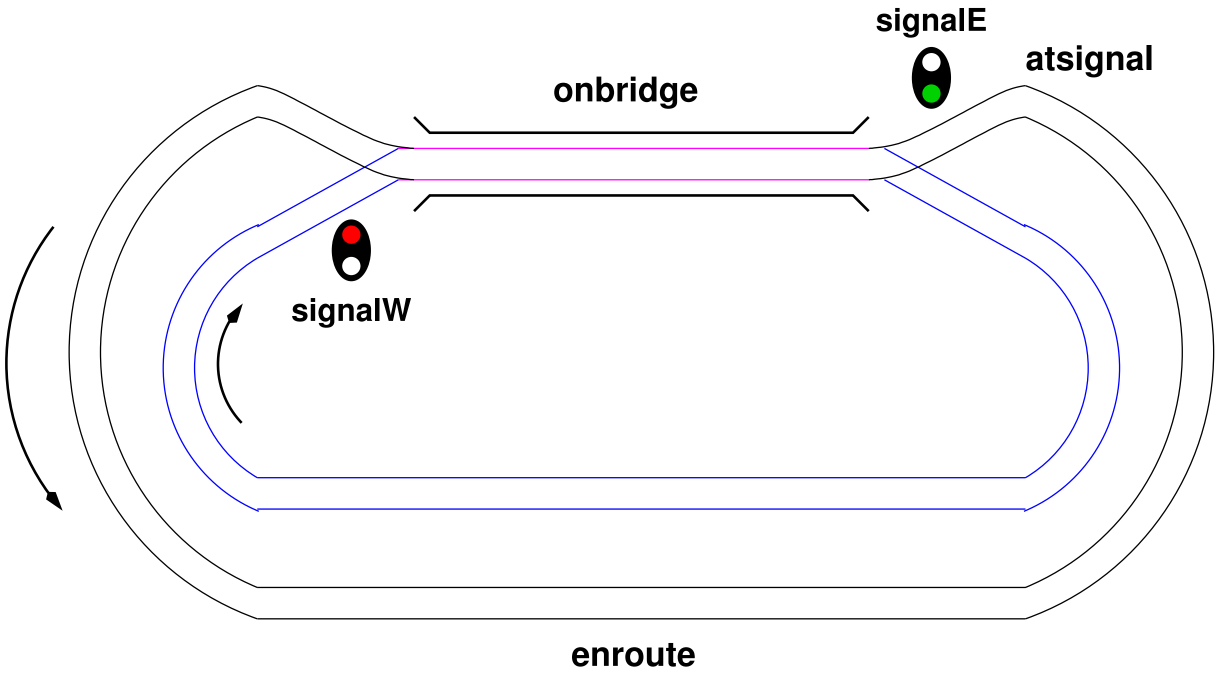
We say that $\mathcal{T}$ is *deadlocked* at q if no action except τ is enabled at q.

Example 2.2 (Hour clock as transition system)

The hour clock (see example 1.1) gives rise to the following transition system:



- all states are initial
- stuttering and "tick" actions
- all states reachable, no deadlocks



Example 2.3 (Toy railway)

system state includes:

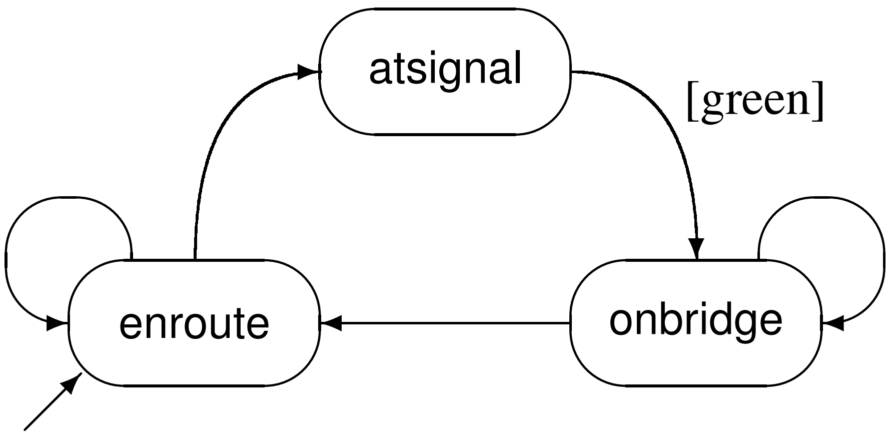
- train data (position, speed, acceleration, . . .)
- signal and switch state

state transitions:

- update of train data (sensor readings)
- signal switching

abstract (finite-state) model: divide tracks into sections

trains



signals (combinatorial)

trainW	trainE	signalW	signalE
enroute	atsignal	red	green
atsignal	enroute	green	red
atsignal	atsignal	?	?
else		red	red

entire transition system obtained as product of trains and signals

Note:
- non-determinism due to abstract model
- some transitions mapped to stuttering

Example 2.4 (parallel programs as transition systems)

var x, y : integer = 0, 0;

cobegin

α : **while** $y = 0$ **do** $\beta : x := x + 1$ **end** $\|$ $\gamma : y := 1$

coend

states: valuations of program variables (including "program counter")

actions: one action per program instruction, plus stuttering

two sample runs

run 1:

action	—	α	β	α	β	α	γ	β	α	τ	$\cdots$
x	0	0	1	1	2	2	2	3	3	3	$\cdots$
y	0	0	0	0	0	0	1	1	1	1	$\cdots$

run 2:

action	—	α	β	α	β	α	β	α	β	α	$\cdots$
x	0	0	1	1	2	2	3	3	4	4	$\cdots$
y	0	0	0	0	0	0	0	0	0	0	$\cdots$

2.2 Fairness conditions

Transition systems define the *possible* transitions.

They often admit “unfair” runs that have no counterpart in the “real” system.

Fairness problems arise from local choices (non-determinism) that are continuously resolved in one way but not the other.

Non-determinism is often due to abstraction:

- railway: abstract from exact train positions
- stopwatch program: representation of parallel execution by non-determinism

Fairness conditions specify that some actions *must* happen, provided they are “sufficiently often” enabled.

They place additional “global” constraints on runs of transition systems.

Weak fairness (justice). A run $\rho = q_0 \xrightarrow{A_0} q_1 \xrightarrow{A_1} q_2 \ldots$ is weakly fair w.r.t. an action $A \in \mathcal{A}$ iff the following condition holds:

If A is enabled at all states beyond m then $A_n = A$ for some $n \geq m$.

equivalent: If A is taken only finitely often then A is infinitely often disabled.

Strong fairness (compassion). A run $\rho = q_0 \xrightarrow{A_0} q_1 \xrightarrow{A_1} q_2 \ldots$ is strongly fair w.r.t. an action $A \in \mathcal{A}$ iff the following condition holds:

If A is enabled at infinitely many states beyond m then $A_n = A$ for some $n \geq m$.

equivalent: If A is taken only finitely often then A is only finitely often enabled.

Prove: strong fairness implies weak fairness

Any run that is strongly fair w.r.t. A is also weakly fair w.r.t. A.

Definition 2.5 A *fair transition system* $\mathcal{T}_f = (Q, I, \mathcal{A}, \delta, W, S)$ extends a transition system by sets $W, S \subseteq \mathcal{A}$.

The runs of $\mathcal{T}_f$ are those runs of the underlying transition system that are weakly fair w.r.t. all actions $A \in W$ and strongly fair w.r.t. all actions $A \in S$.

The following fairness conditions are reasonable for our examples:

hour clock: weak fairness for "tick" action *HCnxt*

toy railway:

- weak fairness for "leave bridge" (i.e., transition from onbridge to enroute)
- strong fairness for switching either signal to green in case of conflict

stopwatch program: weak fairness for each of the two processes

The choice of adequate fairness conditions is non-trivial and must be validated w.r.t. the "real world" (the system being modeled).

Assuming we have complete control over scheduling of actions, fairness conditions can be implemented. For weak fairness, a “round-robin” scheduler is sufficient.

Theorem 2.6 Let $\mathcal{T}_f = (Q, I, \mathcal{A}, \delta, \{B_0, \ldots, B_{m-1}\}, \emptyset)$ be a fair transition system without strong fairness, and let $s_0 \xrightarrow{A_0} s_1 \ldots \xrightarrow{A_{n-1}} s_n$ be a finite execution of $\mathcal{T}_f$.

Then every sequence $s_0 \xrightarrow{A_0} s_1 \ldots \xrightarrow{A_{n-1}} s_n \xrightarrow{A_n} s_{n+1} \ldots$ is a run of $\mathcal{T}_f$ provided that for all $k \geq n$ the following conditions hold:

1. $(s_k, A_k, s_{k+1}) \in \delta$ and
2. If the action $B_{k \text{ mod } m}$ is enabled at s_k then $A_k = B_{k \text{ mod } m}$.

Since we assume δ to be total (ensured by stuttering action τ) the theorem asserts that any finite execution of $\mathcal{T}_f$ can be extended to an (infinite) fair run of $\mathcal{T}_f$.

Proof (of Theorem 2.6). By condition (1), $\rho = s_0 \xrightarrow{A_0} s_1 \ldots \xrightarrow{A_{n-1}} s_n \xrightarrow{A_n} s_{n+1} \ldots$ is clearly a run of the underlying transition system $\mathcal{T}$ without fairness conditions.

It remains to prove that ρ is weakly fair for, say, action B_i.

So assume that B_i is enabled at all states s_k for $k \geq p \geq n$ (for some $p \in \mathbb{N}$).
By condition (2), we know that $A_k = B_i$ for all $k \geq p$ such that $k \bmod m = i$.

There are infinitely many such k, hence B_i appears infinitely often in ρ. Q.E.D.

A similar theorem holds for strong fairness, but it requires a priority scheduler: actions that have not been executed for a long time are prioritized.

Theorem 2.7 Let $\mathcal{T}_f = (Q, I, \mathcal{A}, \delta, \emptyset, \{B_0, \ldots, B_{m-1}\})$ be a fair transition system with strong fairness, and let $s_0 \xrightarrow{A_0} s_1 \ldots \xrightarrow{A_{n-1}} s_n$ be a finite execution of $\mathcal{T}_f$.

Then every sequence $s_0 \xrightarrow{A_0} s_1 \ldots \xrightarrow{A_{n-1}} s_n \xrightarrow{A_n} s_{n+1} \ldots$ is a run of $\mathcal{T}_f$ provided that there exists a sequence $\pi_n, \pi_{n+1}, \ldots$ of permutations π_k of $\{B_0, \ldots, B_{m-1}\}$ such that for all $k \geq n$ the following conditions hold:

1. $(s_k, A_k, s_{k+1}) \in \delta$,
2. Assume that $\pi_k = \langle C_0, \ldots, C_{m-1} \rangle$.

   If there exists i such that C_i is enabled at state s_k but all C_j where $j < i$ are disabled then $A_k = C_j$ and $\pi_{k+1} = \langle C_0, \ldots, C_{i-1}, C_{i+1}, \ldots, C_{m-1}, C_i \rangle$.

   Otherwise $A_k \in \mathcal{A}$ is arbitrary and $\pi_{k+1} = \pi_k$.

Again, any finite execution can be extended in this way to yield an infinite run.

Proof (of theorem 2.7). By condition (1), $\rho = s_0 \xrightarrow{A_0} s_1 \ldots \xrightarrow{A_{n-1}} s_n \xrightarrow{A_n} s_{n+1} \ldots$ is clearly a run of the underlying transition system $\mathcal{T}$ without fairness conditions.

It remains to prove that ρ is strongly fair for, say, action B_i. Assume not.

Then we may choose some $p \geq n$ such that B_i is enabled at infinitely many $k \geq p$ but $A_k \neq B_i$ for all $k \geq p$.

Consider the sequence $\pi_p, \pi_{p+1}, \ldots$, and in particular the positions $j_p, j_{p+1}, \ldots$ of action B_i in the π_k: because B_i is never executed, the sequence of the j_k is weakly decreasing (i.e., $j_{k+1} \leq j_k$ for all $k \geq p$), and therefore eventually stabilizes, say, $j_k = j \in \mathbb{N}$ for all $k \geq q$ (for some $q \geq p$).

By condition (2), it follows that there exist actions $C_0, \ldots, C_j = B_i$ such that for all $k \geq q$, the lists π_k are of the form $\langle C_0, \ldots, C_j, \ldots \rangle$, and none of $C_0, \ldots, C_j$ are enabled.

In particular, it follows that $C_j = B_i$ is never enabled beyond state s_q — contradiction.

Q.E.D.

Interpretation of the theorems 2.6 and 2.7.

- If runs for a transition system $\mathcal{T}$ can be generated effectively (i.e., initial and successor states are computable), then fair runs of an FTS obtained from $\mathcal{T}$ by adding some fairness conditions can be generated using schedulers.

  In fact, it is enough to use the scheduler only after an arbitrary finite prefix.

- Since strong fairness implies weak fairness, the scheduler of theorem 2.7 can also be used for FTSs with both weak and strong fairness conditions.

- However, not all fair runs, are generated in this way.

  In particular, schedulers are of no use when some actions are controlled by the environment.

- The theorems can be extended to fairness conditions on denumerable sets of actions by "diagonalization".

2.3 Properties of runs

When analysing transition systems, one is interested in properties of their runs:

- The two trains are never simultaneously in section onbridge.
- Any train waiting at the signal will eventually be on the bridge.
- The variable x will eventually remain constant.

Properties about the branching structure are occasionally also of interest:

- From any state it is possible to reach an initial state.
- Two actions A and B are in conflict, resp. are independent.
- Two processes can cooperate to starve a third process.

In the following, we restrict attention to properties of runs.

We identify a property Φ with the set of runs that satisfy Φ:

Definition 2.8 Let Q and $\mathcal{A}$ be sets of states and actions. A $(Q, \mathcal{A})$*-property* Φ is a set of ω-sequences $\sigma = s_0 \xrightarrow{A_0} s_1 \xrightarrow{A_1} \ldots$ where $s_i \in Q$ and $A_i \in \mathcal{A}$.

We interchangeably write $\sigma \in \Phi$ and $\sigma \models \Phi$.

Examples:

- set of runs of a transition system $\mathcal{T}$
- runs that are strongly fair for a given action $A \in \mathcal{A}$
- runs $s_0 \xrightarrow{A_0} s_1 \xrightarrow{A_1} \ldots$ such that $s_n(y) = 1$ for some $n \in \mathbb{N}$

Note: assertions about the existence of certain runs are not "properties" in the sense of definition 2.8!

Safety and liveness properties (Lamport 1980)

- two fundamental classes of properties, different proof principles
- generalization of partial correctness and termination of sequential programs

safety properties: something bad never happens

- trains are never simultaneously on the bridge
- data is received in the same order as it was sent

liveness properties: something good eventually happens

- trains will enter section onbridge
- every data item will eventually be received
- action γ will eventually be executed

The following is neither a safety nor a liveness property:

trains wait at signals until entering section onbridge, which will eventually occur

Definition 2.9 (Alpern, Schneider 1985)

- A property Φ is a *safety property* iff the following condition holds:

$$\sigma = s_0 \xrightarrow{A_0} s_1 \xrightarrow{A_1} s_2 \ldots \text{ is in } \Phi \text{ if and only if}$$

$$\text{every finite prefix } s_0 \xrightarrow{A_0} s_1 \ldots \xrightarrow{A_{n-1}} s_n \text{ of } \sigma \text{ can be extended}$$

$$\text{to some sequence } s_0 \xrightarrow{A_0} s_1 \ldots \xrightarrow{A_{n-1}} s_n \xrightarrow{B_n} t_{n+1} \xrightarrow{B_{n+1}} t_{n+2} \ldots \in \Phi.$$

- A property Φ is a *liveness property* iff any finite sequence $s_0 \xrightarrow{A_0} s_1 \ldots \xrightarrow{A_{n-1}} s_n$ can be extended to some sequence $s_0 \xrightarrow{A_0} s_1 \ldots \xrightarrow{A_{n-1}} s_n \xrightarrow{A_n} s_{n+1} \ldots \in \Phi$.

Connection with informal description

- A sequence σ does *not* satisfy a safety property Φ iff there exists some finite prefix of σ that cannot be extended to an infinite sequence satisfying Φ.

  The "bad thing" has thus happened after some finite time.

- Liveness properties do not exclude finite prefixes: "good thing" may occur later.

Properties and finite sequences

- Given a sequence $\sigma = s_0 \xrightarrow{A_0} s_1 \ldots \xrightarrow{A_1} s_2 \ldots$, we write $\sigma[..n]$ to denote the prefix $s_0 \xrightarrow{A_0} s_1 \ldots \xrightarrow{A_{n-1}} s_n$.
- For sequences $\rho = s_0 \xrightarrow{A_0} s_1 \ldots \xrightarrow{A_{n-1}} s_n$ and $\sigma = s_n \xrightarrow{A_n} s_{n+1} \xrightarrow{A_{n+1}} s_{n+2} \ldots$, we write $\rho \circ \sigma$ for the concatenation $s_0 \xrightarrow{A_0} s_1 \ldots \xrightarrow{A_{n-1}} s_n \xrightarrow{A_n} s_{n+1} \xrightarrow{A_{n+1}} s_{n+2} \ldots$.
- For a property Φ and a finite sequence $\rho = s_0 \xrightarrow{A_0} s_1 \ldots \xrightarrow{A_{n-1}} s_n$, we write $\rho \models \Phi$ iff $\rho \circ \sigma \in \Phi$ for some infinite sequence σ (ρ optimistically satisfies Φ).

Reformulation of characteristic conditions:

- Φ is a safety property iff for any infinite sequence σ:

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

- Φ is a liveness property iff $\sigma[..n] \models \Phi$ for all σ and all $n \in \mathbb{N}$.

Examples

- The set $\mathcal{R}$ of runs of a transition system $\mathcal{T} = (Q, I, \mathcal{A}, \delta)$ with a total transition relation, but without fairness conditions, is a safety property:

  Let $\sigma = s_0 \xrightarrow{A_0} s_1 \xrightarrow{A_1} s_2 \ldots$

  $$\begin{array}{rl} & \sigma[..n] \models \mathcal{R} \text{ for all } n \in \mathbb{N} \\ \Rightarrow & s_0 \in I \text{ and } (s_i, s_{i+1}) \in \delta \text{ for all } i < n, \text{ for all } n \in \mathbb{N} \\ \Rightarrow & \sigma \in \mathcal{R} \end{array}$$

- Weak or strong fairness conditions are liveness properties:

  Using the constructions of theorems 2.6 and 2.7, any finite sequence can be extended to some sequence satisfying a fairness property.

Theorem 2.10 (safety and liveness: fundamental results)

1. If Φ_i is a safety property, for all $i \in I$, then so is $\bigcap_{i \in I} \Phi_i$.

2. If Φ is a liveness property then so is any $\Psi \supseteq \Phi$.

3. The trivial property containing all sequences is the only property that is both a safety and a liveness property.

4. For any property Φ, the property

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

   is the smallest safety property containing Φ, called the *safety closure* of Φ.

   - Φ is a safety property iff $\mathcal{C}(\Phi) = \Phi$.
   - If Φ is arbitrary and Ψ is a safety property then: $\Phi \subseteq \Psi$ iff $\mathcal{C}(\Phi) \subseteq \Psi$.

5. $\Phi \subseteq \Psi \quad \Longrightarrow \quad \mathcal{C}(\Phi) \subseteq \mathcal{C}(\Psi)$.

6. For any property Φ there is a safety property S_Φ and a liveness property L_Φ such that $\Phi = S_\Phi \cap L_\Phi$.

Proof. 1–3, 5: exercise!

4. Clearly, we have $\Phi \subseteq \mathcal{C}(\Phi)$ for any Φ. Moreover, $\mathcal{C}(\Phi)$ is a safety property:

$$
\begin{array}{lll}
 & \sigma[..n] \models \mathcal{C}(\Phi) \text{ for all } n \in \mathbb{N} & \\
\Rightarrow & \text{for all } n \in \mathbb{N} \text{ there is } \tau \text{ such that } \sigma[..n] \circ \tau \in \mathcal{C}(\Phi) & [\text{def. } \sigma[..n] \models \mathcal{C}(\Phi)] \\
\Rightarrow & \sigma[..n] \models \Phi \text{ for all } n \in \mathbb{N} & [\text{def. } \mathcal{C}(\Phi)] \\
\Rightarrow & \sigma \in \mathcal{C}(\Phi) & [\text{def. } \mathcal{C}(\Phi)]
\end{array}
$$

$\mathcal{C}(\Phi) \subseteq S$ for any safety property S such that $\Phi \subseteq S$:

$$
\begin{array}{lll}
 & \sigma \in \mathcal{C}(\Phi) & \\
\Rightarrow & \sigma[..n] \models \Phi \text{ for all } n \in \mathbb{N} & [\text{def. } \mathcal{C}(\Phi)] \\
\Rightarrow & \text{for all } n \in \mathbb{N} \text{ exists } \tau \text{ such that } \sigma[..n] \circ \tau \in \Phi & [\text{def. } \sigma[..n] \models \Phi] \\
\Rightarrow & \text{for all } n \in \mathbb{N} \text{ exists } \tau \text{ such that } \sigma[..n] \circ \tau \in S & [\Phi \subseteq S] \\
\Rightarrow & \sigma[..n] \models S \text{ for all } n \in \mathbb{N} & [\text{def. } \sigma[..n] \models S] \\
\Rightarrow & \sigma \in S & [S \text{ safety property}]
\end{array}
$$

6. Let $S_\Phi = \mathcal{C}(\Phi)$ and $L_\Phi = \{\sigma : \sigma \notin \mathcal{C}(\Phi) \text{ or } \sigma \in \Phi\}$: exercise!

Example 2.11 (see also "stopwatch" example 2.4)
Let Φ be the set of all sequences $s_0 \xrightarrow{A_0} s_1 \xrightarrow{A_1} s_2 \ldots$ such that for some $n \in \mathbb{N}$,

$$s_i(y) = 0 \text{ for all } i \leq n \text{ and } s_i(y) = 1 \text{ for all } i > n$$

The safety closure $\mathcal{C}(\Phi)$ contains the sequences $\sigma = s_0 \xrightarrow{A_0} s_1 \xrightarrow{A_1} s_2 \ldots$ such that

- either $\sigma \in \Phi$ or
- $s_i(y) = 0$ for all $i \in \mathbb{N}$.

Exercise 2.12
Let $\mathcal{T} = (Q, I, \mathcal{A}, \delta, W, S)$ be a fair transition system with $W, S \subseteq \mathcal{A}$ and $\mathcal{A}$ finite.

Determine the safety closure of the set of (fair) runs of $\mathcal{T}$.

By theorem 2.10(6), any property can be written as a pair (S, L) where S is a safety property and L is a liveness property.

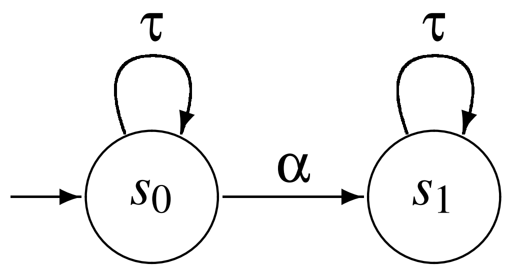
It is often desirable that S alone provides all constraints on finite sequences ρ:

$$\rho \models S \quad \Longrightarrow \quad \rho \circ \sigma \models S \cap L \text{ for some } \sigma$$

Definition 2.13 Let S be a safety property and L be any property.

The pair (S, L) is machine closed iff $\mathcal{C}(S \cap L) = S$.

Example 2.14 (non-machine-closed specification)
Let S denote the set of all runs of the transition system



and let L be the set of sequences that contain the state s_0 infinitely often.

The finite run $s_0 \xrightarrow{\alpha} s_1$ can be extended to a run in S, but not in $S \cap L$.

If (S,L) is machine-closed and Φ is a safety property then the runs satisfying (S,L) satisfy Φ iff $S \subseteq \Phi$ holds:

$$
\begin{array}{lll}
 & S \cap L \subseteq \Phi & \\
\Leftrightarrow & \mathcal{C}(S \cap L) \subseteq \Phi & \text{[Theorem 2.10(4)]} \\
\Leftrightarrow & S \subseteq \Phi & [(S,L) \text{ machine closed}]
\end{array}
$$

The liveness property L can thus be ignored for the proof of safety properties.

Notes:

- If (S,L) is a system specification then it should usually be machine closed: otherwise they require unbounded look-ahead and are *non-implementable*.
- Some formalisms ensure that all system specifications are machine closed.
- Theorems 2.6 and 2.7 imply that fair transition systems yield machine closed specifications. (They can be generalized for countably many fairness conditions.)

Summary

- Transition systems: semantics of reactive and distributed systems
- Fairness conditions constrain local non-determinism
- Properties of runs formalized as sets of (infinite) state-action sequences
- Rich theory of safety and liveness properties
- Every property is the intersection of a safety and a liveness property
- Machine closure prerequisite for implementability of system specifications

3 System specification in TLA$^+$

Temporal logics: a short history

- Middle Ages: understand temporal relations in natural language

  Yesterday she said that she'd come tomorrow, so she should come today.

- 20th century: formalisation of modal and temporal logics

  temporal primitives: always, eventually, until, since, . . .

  A. Prior: *Past, present, and future.* Oxford University Press, 1967

- 1977: Pnueli uses temporal logic to express properties of reactive systems

  A. Pnueli: *The temporal logic of programs.* FOCS'77

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

Temporal Logic of Actions (TLA) (L. Lamport, TOPLAS 1994)

- uniform language : transition system *and* properties represented as formulas
- mathematical abstraction : basis for description and analysis of reactive and distributed systems
- logical connectives express structural concepts (composition, refinement, hiding)
- avoid temporal logic : first-order proof obligations whenever possible

Keep it as simple as possible, but no simpler

3.1 Anatomy of TLA

TLA defines two levels of syntax: *action formulas* and *temporal formulas*.

- action formulas describe states and state transitions
- temporal formulas describe state sequences

Formally, assume given:

- a first-order signature (function and predicate symbols),
- disjoint sets $\mathcal{X}_r$ and $\mathcal{X}_f$ of *rigid* and *flexible* (or *state*) variables.

Rigid variables denote values as in first-order logic.

Flexible variables represent state components (program variables).

Action formulas are evaluated over pairs of states

They are ordinary first-order formulas built from

- rigid variables $x \in \mathcal{X}_r$,
- (unprimed) flexible variables $v \in \mathcal{X}_f$, and
- primed flexible variables v' for $v \in \mathcal{X}_f$.

Examples: $hr \in (0..23)$, $hr' = hr + 1$, $\exists k : n + m' < 3 * k$, ...

Terms are called *transition functions*, formulas *transition predicates* or *actions*.

Action formulas without free primed variables are called *state formulas*.

Actions are not primitive in TLA!

Semantics of action formulas

- *first-order interpretation* I (for the underlying signature)
  - provides a universe $|I|$ of values
  - interprets function and predicate symbols: $0,\ +,\ <,\ \in,\ \ldots$
- *state* : valuation of flexible variables $s : \mathcal{X}_f \to |I|$
- *valuation* of rigid variables $\xi : \mathcal{X}_r \to |I|$

$[\![A]\!]^{\xi}_{s,t} \in \{\mathbf{tt}, \mathbf{ff}\}$ given by standard inductive definition

- s and t interpret unprimed and primed flexible variables
- ξ interprets rigid variables

Note: semantics of state formulas independent of second state

Notations (for action formulas)

- For a state formula e, write e' for the action formula obtained by "priming" all free flexible variables (rename bound variables as necessary).

  Examples:
  $$
  \begin{aligned}
  (v+1)' &\equiv v'+1 \\
  (\exists x : n = x+m)' &\equiv \exists x : n' = x+m' \\
  (\exists n' : n = n'+m)' &\equiv \exists np : n' = np+m'
  \end{aligned}
  $$

- For an action A and a state function t write
  $$
  \begin{aligned}
  [A]_t &\equiv A \vee t' = t \\
  \langle A \rangle_t &\equiv A \wedge \neg(t' = t)
  \end{aligned}
  $$

  Note:
  $$
  \begin{aligned}
  \langle A \rangle_t &\equiv \neg[\neg A]_t \qquad & \neg\langle A \rangle_t &\equiv [\neg A]_t \\
  [A]_t &\equiv \neg\langle \neg A \rangle_t \qquad & \neg[A]_t &\equiv \langle \neg A \rangle_t
  \end{aligned}
  $$

- For an action A define the state formula (!)

$$\text{ENABLED}\ A \;\equiv\; \exists v'_1, \ldots, v'_n : A$$

where $v'_1, \ldots, v'_n$ are all free primed flexible variables in A.

ENABLED A holds at s iff there is some state t such that A holds of (s,t).

- For two actions A and B define

$$A \cdot B \;\equiv\; \exists v''_1, \ldots, v''_n : A[v''_1/v_1, \ldots, v''_n/v_n] \wedge B[v''_1/v'_1, \ldots, v''_n/v_n]$$

$A \cdot B$ holds of (s,t) iff for some state u, A holds of (s,u) and B of (u,t).

It represents the sequential composition of A and B as a single atomic action.

Temporal formulas are evaluated over (infinite) state sequences

Definition 3.1 (syntax and semantics of temporal formulas)

Let $\sigma = s_0 s_1 \ldots$ be a sequence of states and ξ be a valuation of the rigid variables.

- Every state formula P is a formula.

  $\sigma, \xi \models P$ iff $[\![P]\!]^{\xi}_{s_0} = \mathbf{tt}$

- For an action A and a state function t, $\Box[A]_t$ (*"always square A sub t"*) is a formula.

  $\sigma, \xi \models \Box[A]_t$ iff for all $n \in \mathbb{N}$, $[\![A]\!]^{\xi}_{s_n, s_{n+1}} = \mathbf{tt}$ or $[\![t]\!]^{\xi}_{s_n} = [\![t]\!]^{\xi}_{s_{n+1}}$

- If F is a formula then so is $\Box F$ (*"always F"*).

  $\sigma, \xi \models \Box F$ iff $\sigma[n..], \xi \models F$ for all $n \in \mathbb{N}$

- Boolean combinations of formulas are formulas, as are $\exists x : F$ and $\forall x : F$ for $x \in \mathcal{X}_r$ (with obvious semantics).

Notations (for temporal formulas)

- If F is a temporal formula then $\Diamond F$ (*"eventually F"*, *"finally F"*) abbreviates

$$\Diamond F \;\equiv\; \neg\Box\neg F \quad : \quad \sigma,\xi \models \Diamond F \text{ iff } \sigma[n..],\xi \models F \text{ for some } n \in \mathbb{N}$$

- Similarly we define $\Diamond\langle A\rangle_t$ (*"eventually angle A sub t"*)

$$\Diamond\langle A\rangle_t \;\equiv\; \neg\Box[\neg A]_t \quad : \quad \sigma,\xi \models \Diamond\langle A\rangle_t \text{ iff } [\![\langle A\rangle_t]\!]^{\xi}_{s_n,s_{n+1}} = \mathbf{tt} \text{ for some } n \in \mathbb{N}$$

- $F \leadsto G$ (*"F leads to G"*) is defined as

$$F \leadsto G \;\equiv\; \Box(F \Rightarrow \Diamond G)$$

It asserts that every suffix satisfying F is followed by some suffix satisfying G.

Infinitely often and eventually always

- The formula $\Box\Diamond F$ asserts that F holds infinitely often over σ:

  $$\sigma, \xi \models \Box\Diamond F \text{ iff for all } m \in \mathbb{N} \text{ there is } n \geq m \text{ such that } \sigma[n..], \xi \models F$$

  Similarly, the formula $\Box\Diamond\langle A\rangle_t$ asserts that the action $\langle A\rangle_t$ occurs infinitely often.

- The formula $\Diamond\Box F$ asserts that F holds from a certain suffix onward.

  Equivalently, F is false only finitely often.

  The formula $\Diamond\Box[A]_t$ asserts that only $[A]_t$ actions occur after some initial time.

Equivalences:

$\neg\Box\Diamond F \equiv \Diamond\Box\neg F$ $\quad$ $\Diamond\Box\Diamond F \equiv \Box\Diamond F$

$\neg\Diamond\Box F \equiv \Box\Diamond\neg F$ $\quad$ $\Box\Diamond\Box F \equiv \Diamond\Box F$

Example 3.2 (semantics of temporal formulas)

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 $= 0$)

Which of the following formulas hold of this behavior?

$\Box\neg(x = 0 \wedge y = 0)$

$\Box[x = 0 \Rightarrow y' = 0]_{x,y}$

$\Diamond(x = 7 \wedge y = 0)$

$\Diamond\langle y = 0 \wedge x' = 0\rangle_y$

$\Box\Diamond(y \neq 0)$

$\Diamond\Box(x = 0 \Rightarrow y \neq 0)$

$\Diamond\Box[\text{FALSE}]_y$

Representing fairness in TLA

Recall definitions of weak and strong fairness conditions:

- A run is weakly fair for some action A iff A occurs infinitely often provided that it is eventually always enabled.
- A run is strongly fair for some action A iff A occurs infinitely often provided that it is infinitely often enabled.

For actions $\langle A \rangle_t$ this can be written in TLA:

$$\begin{aligned} \mathrm{WF}_t(A) &\equiv \Diamond\Box \textsc{Enabled}\ \langle A \rangle_t \Rightarrow \Box\Diamond\langle A \rangle_t \\ \mathrm{SF}_t(A) &\equiv \Box\Diamond \textsc{Enabled}\ \langle A \rangle_t \Rightarrow \Box\Diamond\langle A \rangle_t \end{aligned}$$

Equivalent conditions:

$$\mathrm{WF}_t(A) \equiv \Box\Diamond\neg\textsc{Enabled}\ \langle A \rangle_t \lor \Box\Diamond\langle A \rangle_t \qquad \mathrm{SF}_t(A) \equiv \Diamond\Box\neg\textsc{Enabled}\ \langle A \rangle_t \lor \Box\Diamond\langle A \rangle_t$$

$$\mathrm{WF}_t(A) \equiv \Box(\Box\textsc{Enabled}\ \langle A \rangle_t \Rightarrow \Diamond\langle A \rangle_t) \qquad \mathrm{SF}_t(A) \equiv \Box(\Box\Diamond\textsc{Enabled}\ \langle A \rangle_t \Rightarrow \Diamond\langle A \rangle_t)$$

Example 3.3 (stopwatch as a TLA$^+$ module)

MODULE *Stopwatch*

EXTENDS *Naturals*

VARIABLES pc_1, pc_2, x, y

$$Init \triangleq pc_1 = \text{"alpha"} \land pc_2 = \text{"gamma"} \land x = 0 \land y = 0$$

$$A \triangleq \begin{array}{l} \land\, pc_1 = \text{"alpha"} \land pc_1' = \text{IF } y = 0 \text{ THEN "beta" ELSE "stop"} \\ \land\, \text{UNCHANGED } \langle pc_2, x, y \rangle \end{array}$$

$$B \triangleq \begin{array}{l} \land\, pc_1 = \text{"beta"} \land pc_1' = \text{"alpha"} \\ \land\, x' = x + 1 \land \text{UNCHANGED } \langle pc_2, y \rangle \end{array}$$

$$G \triangleq \begin{array}{l} \land\, pc_2 = \text{"gamma"} \land pc_2' = \text{"stop"} \\ \land\, y' = 1 \land \text{UNCHANGED } \langle pc_1, x \rangle \end{array}$$

$$vars \triangleq \langle pc_1, pc_2, x, y \rangle$$

$$Spec \triangleq Init \land \Box[A \lor B \lor G]_{vars} \land \text{WF}_{vars}(A \lor B) \land \text{WF}_{vars}(G)$$

Note:
- explicit encoding of control structure
- process structure lost

Stuttering invariance

Actions in TLA formulas must be "guarded" : $\Box[A]_t,\ \Diamond\langle A\rangle_t$

These formulas allow for finitely many state repetitions, and this observation extends to arbitrary TLA formulas.

Definition 3.4 Stuttering equivalence ($\approx$) is the smallest equivalence relation that identifies behaviors

$$s_0 s_1 \ldots s_n s_{n+1} s_{n+2} \ldots \quad \text{and} \quad s_0 s_1 \ldots s_n s_n s_{n+1} s_{n+2} \ldots$$

Theorem 3.5 For any TLA formula F and stuttering equivalent behaviors $\sigma \approx \tau$:

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

TLA formulas cannot distinguish stuttering equivalent behaviors.

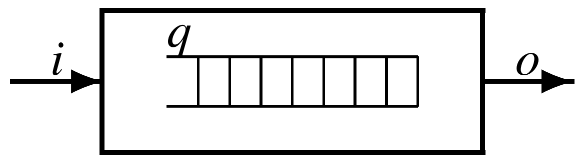
3.2 Representing system paradigms in TLA

Recall: a system specification is usually of the form

$$Init \wedge \Box[Next]_v \wedge L$$

- state components (e.g., program variables, communication channels) explicitly represented as flexible variables
- synchronization and communication encoded explicitly by appropriate actions
- different classes of systems characterized by different specification styles
- in the following: example specifications of FIFO channels

Example 3.6 (lossy FIFO)



MODULE *LossyQueue*

EXTENDS *Sequences*

VARIABLES *i,o,q*

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

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

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

$$LQNext \triangleq LQEnq \vee LQDeq$$

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

$$LQSpec \triangleq LQInit \wedge \Box[LQNext]_{q,o} \wedge LQLive$$

- i and o represent interface, q is (unbounded) internal buffer
- buffer can enqueue same input value several times, or not at all

Simple interleaving specifications are of the form

$$Init \wedge \Box[Next]_{v,o} \wedge L$$

i, o, v : input, output and internal variables of the system

Next : action formula describing the possible transitions

Only o and v appear in the index: the system allows for arbitrary changes to the input variables ("environment actions").

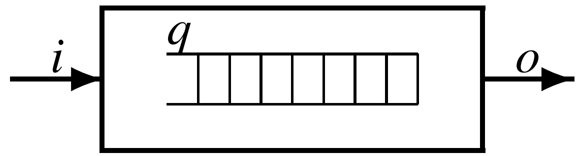
The system should not change the input variables (interleaving model):

$$Next \Rightarrow i' = i$$

L : conjunction of fairness conditions $\mathrm{WF}_{v,o}(A)$ or $\mathrm{SF}_{v,o}(A)$

Usually, *Next* is a disjunction $A_1 \vee \ldots \vee A_n$, and L asserts fairness of several A_i.

Example 3.7 (synchronous communication, interleaving)



MODULE *SyncInterleavingQueue*

EXTENDS *Sequences*

VARIABLES *i,o,q*

$$SIQInit \triangleq q = \langle\rangle \wedge i = o$$

$$SIQEnq \triangleq i' \neq i \wedge q' = Append(q, i') \wedge o' = o$$

$$SIQDeq \triangleq q \neq \langle\rangle \wedge o' = Head(q) \wedge q' = Tail(q) \wedge i' = i$$

$$SIQNext \triangleq SIQEnq \vee SIQDeq$$

$$SIQLive \triangleq \mathrm{WF}_{i,q,o}(SIQDeq)$$

$$SIQSpec \triangleq SIQInit \wedge \Box[SIQNext]_{i,q,o} \wedge SIQLive$$

- *i* appears in the index: “synchronous” reaction to changes of input
- interleaving model: *SIQEnq* and *SIQDeq* mutually exclusive
- every run of *SIQSpec* also satisfies *LQSpec*

Interleaving specifications with synchronous communication

$$Init \wedge \Box[Next]_{i,v,o} \wedge L$$

Next : disjunction $Env \vee Sys$

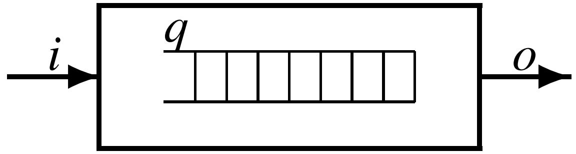
- *Sys* describes system actions (internal or output)
- *Env* describes environment actions and their effect on system state

$$Sys \Rightarrow i' = i \qquad \text{and} \qquad Env \Rightarrow o' = o$$

- no action changes both input and output: interleaving model
- input variables appear in the index to ensure reaction to their change
- closed system specifications

L : asserts fairness conditions of system actions

Example 3.8 (asynchronous communication, interleaving)



MODULE *AsyncInterleavingQueue*

EXTENDS *Sequences*

VARIABLES i, o, q, sig

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

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

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

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

$$AQNext \triangleq AQEnv \lor AQEnq \lor SIQDeq$$

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

$$AQSpec \triangleq AQInit \land \Box[AQNext]_{i,q,o,sig} \land AQLive$$

- explicit model of “handshake” protocol for enqueuing values (*AQEnv*, *AQEnq*)
- fairness condition on *AQEnq* ensures that system reacts to new inputs
- every run of *AQSpec* also satisfies *LQSpec*

Asynchronous communication has to be modeled explicitly

Environment actions A are represented as two separate actions A_{env} and A_{sys} :

- A_{env} models proper environment step

$$A_{env} \Rightarrow \text{UNCHANGED } \langle v, o \rangle$$

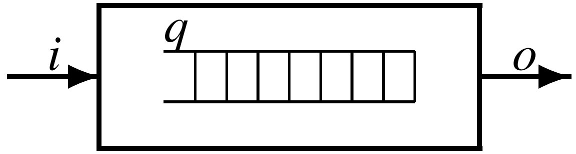
- A_{sys} represents system reaction to environment step

$$A_{sys} \Rightarrow \text{UNCHANGED } \langle i, o \rangle$$

- handshake variables (like *sig*) ensure alternation of A_{sys} and A_{env}
- fairness conditions for A_{sys} ensure (eventual) system reaction

Mostly: interleaving specifications, synchronous or asynchronous communication.

Example 3.9 (synchronous communication, non-interleaving)



MODULE *SyncNonInterleavingQueue*

EXTENDS *Sequences*

VARIABLES i, o, q

$$SNQInit \triangleq q = \langle\rangle \wedge i = o$$

$$d(v) \triangleq \text{IF } v' = v \text{ THEN } \langle\rangle \text{ ELSE } \langle v' \rangle$$

$$SNQEnq \triangleq i' \neq i \wedge q \circ d(i) = d(o) \circ q'$$

$$SNQDeq \triangleq q \neq \langle\rangle \wedge o' = Head(q) \wedge q \circ d(i) = d(o) \circ q'$$

$$SNQLive \triangleq \mathrm{WF}_{i,q,o}(SNQDeq)$$

$$SNQSpec \triangleq \wedge\ SNQInit \wedge \Box[SNQEnq]_i \wedge \Box[SNQDeq]_o$$
$$\qquad\qquad \wedge\ \Box[SNQEnq \vee SNQDeq]_q \wedge SNQLive$$

- one next-state relation per variable
- non-interleaving: input and output may occur simultaneously
- every run of *SIQSpec* also satisfies *SNQSpec*

Non-interleaving specifications simultaneous actions of system and environment

They can be written in the form

$$Init \wedge \Box[Env]_i \wedge \Box[Int]_v \wedge \Box[Out]_o \wedge L$$

- Env, Int, Out describe environment, internal, and output actions
- synchronization by common variables as necessary
- "transition invariants" ensure consistent modifications of state components
- L specifies fairness conditions for subactions of *Int* and *Out*

Observations:

- Non-interleaving specifications are usually harder to write.
- NI specifications may be a more faithful model of the real system.
- NI specifications are easier to compose.

Summary

- TLA: system specification *and* properties are formulas
- action formulas (states and transitions) vs. temporal formulas (behaviors)
- actions must be "guarded": $\Box[A]_v$, $\Diamond\langle A\rangle_v$ entails stuttering invariant semantics
- fairness properties definable as TLA formulas
- different specification styles represent different system paradigms
- interleaving vs. non-interleaving representations

4 System verification and validation

Formal models of systems are the basis for formal analysis.

Validation: are we building the right system ?

- compare model against (informal!) user requirements
- animation, prototyping, run test cases

Verification: are we building the system right ?

- compare model against (formal) correctness properties or abstract model
- theorem proving, model checking, equivalence checking

4.1 Deductive verification in TLA

Systems as well as properties are represented as TLA formulas.

System described by *Spec* satisfies property *Prop*
iff
Prop holds of every run of *Spec*
iff
formula $Spec \Rightarrow Prop$ is valid : $\models Spec \Rightarrow Prop$

System verification reduces to provability of TLA formulas.

Next: verification rules for standard correctness properties

4.1.1 Invariants

formulas $\Box I$ for state predicate I

- characterize the set of reachable states of a system
- express intuitive correctness of algorithm
- basis for proving more advanced properties

Basic proof rule: (INV1)

$$\frac{I \wedge Next \Rightarrow I' \qquad I \wedge v' = v \Rightarrow I'}{I \wedge \Box[Next]_v \Rightarrow \Box I}$$

Justification:

- hypothesis ensures that every transition (stuttering or not) preserves I
- thus, if I holds initially, it will hold throughout the run

Generalization: (INV1_m)

$$\frac{I \wedge [N_1]_{v_1} \wedge \ldots \wedge [N_k]_{v_k} \Rightarrow I'}{I \wedge \Box[N_1]_{v_1} \wedge \ldots \wedge \Box[N_k]_{v_k} \Rightarrow \Box I}$$

Example 4.1 (invariant for the hour clock, see example 1.1)

MODULE *HourClock*

EXTENDS *Naturals*

VARIABLE *hr*

$$HCini \triangleq hr \in (0..23)$$

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

$$HC \triangleq HCini \land \Box[HCnxt]_{hr} \land \mathrm{WF}_{hr}(HCnxt)$$

Prove $HC \Rightarrow \Box HCini$: by (INV1) and propositional logic, it suffices to show

$$hr \in (0..23) \land HCnxt \quad \Rightarrow \quad hr' \in (0..23)$$

$$hr \in (0..23) \land hr' = hr \quad \Rightarrow \quad hr' \in (0..23)$$

Both implications are clearly valid.

(INV1) can be used to prove *inductive* invariants.

Usually, an invariant has to be strengthened for the proof, using the derived rule

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

- J : inductive invariant that implies I
- Finding inductive invariants requires creativity.
- Its proof is entirely schematic and doesn't need temporal logic.
- Some formal methods document inductive invariants as part of the model.

Exercise 4.2

For the interleaving FIFO with synchronous communication, prove that any two consecutive elements in the queue are different:

$$\mathit{SIQSpec} \Rightarrow \Box(\forall i \in (1..\mathit{Len}(q)-1) : q[i] \neq q[i+1])$$

Excursion: For an action A and a state predicate P define

$$\mathbf{wp}(P,A) \stackrel{\triangle}{=} \forall v_1',\ldots,v_n' : A \Rightarrow P' \qquad \big(\equiv \neg\text{ENABLED }(A \wedge \neg P')\big)$$

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

$\mathbf{wp}(P,A)$ is called the *weakest precondition* of P w.r.t. A.

It defines the set of states all of whose A-successors satisfy P.

Examples:

$$\begin{aligned}
\mathbf{wp}(x=5, x'=x+1) &\equiv \forall x' : x'=x+1 \Rightarrow x'=5 \\
&\equiv x=4 \\
\mathbf{wp}(y \in S, S'=S \cup T \wedge y'=y) &\equiv \forall y', S' : S'=S \cup T \wedge y'=y \Rightarrow y' \in S' \\
&\equiv y \in S \vee y \in T \\
\mathbf{wp}(x>0, x'=0) &\equiv \forall x' : x'=0 \Rightarrow x'>0 \\
&\equiv \text{FALSE}
\end{aligned}$$

Using the **wp** notation, (INV) can be rewritten as follows:

$$\frac{Init \Rightarrow J \qquad J \Rightarrow \mathbf{wp}(J, Next \lor v' = v) \qquad J \Rightarrow I}{Init \land \Box[Next]_v \implies \Box I}$$

The following heuristic can help finding inductive invariants:

1. Start with the target invariant: $J_0 \triangleq I$.
2. Try proving $J_i \land A \Rightarrow J_i'$ for each subaction A of $[Next]_v$.

   If the proof fails, set $J_{i+1} \triangleq J_i \land \mathbf{wp}(J_i, A)$.
3. Repeat step 2 until
   - either all sub-proofs succeed and $Init \Rightarrow J_i$ holds: J_i is an inductive invariant
   - or J_i is not implied by *Init*; then I is not an invariant.

This heuristic need not terminate: must also generalize appropriately.

4.1.2 Liveness from fairness

Fairness conditions ensure that actions do occur eventually.

Liveness from weak fairness

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

The hypotheses of (WF1) are again non-temporal formulas.

Example: weak fairness for *HCnxt* ensures that the clock keeps ticking

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

Using (WF1) and first-order logic, this can be reduced to

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

These formulas are again valid.

Exercise 4.3

show that elements advance in the lossy queue, i.e.

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

Correctness of (WF1): assume $\sigma, \xi \models \Box[Next]_v \wedge \mathrm{WF}_v(A)$ where $\sigma = s_0 s_1 \ldots$

To prove that $\sigma, \xi \models P \leadsto Q$ assume that $[\![P]\!]^{\xi}_{s_n} = \mathbf{tt}$ for some $n \in \mathbb{N}$.

For a contradiction, assume also that $[\![Q]\!]^{\xi}_{s_m} = \mathbf{ff}$ for all $m \geq n$.

1. $[\![P]\!]^{\xi}_{s_n} = \mathbf{tt}$ for all $m \geq n$.

   $\surd$ proof by induction on $m \geq n$ using hypothesis $P \wedge [Next]_v \Rightarrow P' \vee Q'$

2. $[\![\text{ENABLED}\ \langle A \rangle_v]\!]^{\xi}_{s_m} = \mathbf{tt}$ for all $m \geq n$.

   $\surd$ from (1) and hypothesis $P \Rightarrow \text{ENABLED}\ \langle A \rangle_v$

3. $[\![\langle A \rangle_v]\!]^{\xi}_{s_m} = \mathbf{tt}$ for some $m \geq n$.

   $\surd$ from (2) and weak fairness assumption for $\langle A \rangle_v$

4. $[\![Q]\!]^{\xi}_{s_{m+1}} = \mathbf{tt}$ for some $m \geq n$.

   $\surd$ from (3), (1), and hypothesis $P \wedge \langle Next \wedge A \rangle_v \Rightarrow Q'$

5. Q.E.D. (by contradiction)

Liveness from strong fairness

For (WF1), the “helpful action” $\langle A \rangle_v$ must remain executable as long as P holds.

This assumption is too strong for actions with strong fairness, which ensures eventual execution if the action is infinitely often (but not necessarily persistently) enabled.

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

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

Example 4.4 (mutual exclusion with semaphores)

Pseudo-code:

semaphore s = 1;

loop			**loop**	
$ncrit_1$:	(* non-critical *)	‖	$ncrit_2$:	(* non-critical *)
try_1 :	P(s)		try_2 :	P(s)
$crit_1$:	(* critical *)		$crit_2$:	(* critical *)
	V(s)			V(s)
endloop			**endloop**	

TLA$^+$ module:

MODULE *Mutex*

VARIABLES *s, pc1, pc2*

$$Init \triangleq s = 1 \land pc1 = \text{"ncrit"} \land pc2 = \text{"ncrit"}$$

$$Ncrit(pc, oth) \triangleq pc = \text{"ncrit"} \land pc' = \text{"try"} \land \text{UNCHANGED } \langle oth, s \rangle$$

$$Enter(pc, oth) \triangleq pc = \text{"try"} \land s = 1 \land pc' = \text{"crit"} \land s' = 0 \land oth' = oth$$

$$Exit(pc, oth) \triangleq pc = \text{"crit"} \land pc' = \text{"ncrit"} \land s' = 1 \land oth' = oth$$

$$Proc1 \triangleq Ncrit(pc1, pc2) \lor Enter(pc1, pc2) \lor Exit(pc1, pc2)$$

$$Proc2 \triangleq Ncrit(pc2, pc1) \lor Enter(pc2, pc1) \lor Exit(pc2, pc1)$$

$$vars \triangleq \langle s, pc1, pc2 \rangle$$

TLA$^+$ module: (continued)

$$Live \triangleq \land \mathrm{SF}_{vars}(Enter(pc1, pc2)) \land \mathrm{SF}_{vars}(Enter(pc2, pc1))$$
$$\qquad \land \mathrm{WF}_{vars}(Exit(pc1, pc2)) \land \mathrm{WF}_{vars}(Exit(pc2, pc1))$$

$$Mutex \triangleq Init \land \Box[Proc1 \lor Proc2]_{vars} \land Live$$

$$Inv \triangleq \lor\ s = 1 \land \{pc1, pc2\} \subseteq \{\text{"ncrit"}, \text{"try"}\}$$
$$\qquad \lor\ s = 0 \land pc1 = \text{"crit"} \land pc2 \in \{\text{"ncrit"}, \text{"try"}\}$$
$$\qquad \lor\ s = 0 \land pc2 = \text{"crit"} \land pc1 \in \{\text{"ncrit"}, \text{"try"}\}$$

THEOREM $Mutex \Rightarrow Inv$

THEOREM $Mutex \Rightarrow (pc1 = \text{"try"} \leadsto pc1 = \text{"crit"})$

The proof of the invariant is straightforward using (INV1) : exercise!

Our goal is to establish liveness:

- The two process can compete for entry to the critical section.
- The helpful action $Enter(pc1, pc2)$ is disabled while $pc2 = \text{"crit"}$.

We use (SF1) to show

$$\Box[Proc1 \lor Proc2]_{vars} \land \mathrm{SF}_{vars}(Enter(pc1, pc2)) \land \Box\mathrm{WF}_{vars}(Exit(pc2, pc1))$$
$$\Rightarrow \; ((pc1 = \text{“try”} \land Inv) \leadsto pc1 = \text{“crit”})$$

The first and second hypotheses of (SF1) pose no problem.

For the third hypothesis, we use (WF1) to show

$$\Box[Proc1 \lor Proc2]_{vars} \land \mathrm{WF}_{vars}(Exit(pc2, pc1))$$
$$\Rightarrow \; ((pc1 = \text{“try”} \land Inv \land s \neq 1) \leadsto (pc1 = \text{“try”} \land s = 1))$$

Simple temporal reasoning (see later) implies

$$\Box(pc1 = \text{“try”} \land Inv) \land \Box[Proc1 \lor Proc2]_{vars} \land \Box\mathrm{WF}_{vars}(Exit(pc2, pc1))$$
$$\Rightarrow \; \Diamond \quad \underbrace{(pc1 = \text{“try”} \land s = 1)}_{\text{ENABLED } \langle Enter(pc1, pc2)\rangle_{vars}}$$

4.1.3 Liveness from well-founded relations

Rules (WF1) and (SF1) prove elementary liveness properties:

- clock will eventually display next hour
- elements in queue will advance by one step, first element will be output

Really, want to prove complex properties:

- clock will eventually display noon

$$HC \Rightarrow \Box\Diamond(hr = 12)$$

- any element in the queue will eventually be output

$$LQSpec \Rightarrow \big((\exists k \in 1..Len(q) : q[k] = x) \leadsto o = x\big)$$

Informal argumentation: repeat elementary liveness argument

- every tick of the clock brings us closer to noon
- every output action will move the element closer to the head of the queue; the following output action will actually put it on the output channel

Definition 4.5 A binary relation $\prec \subseteq D \times D$ is well-founded iff there is no infinite descending chain $d_0 \succ d_1 \succ d_2 \succ \ldots$ of elements $d_i \in D$.

Note: – well-founded relations are irreflexive and asymmetric.

– Every non-empty subset of $(D, \prec)$ contains a minimal element.

Example 4.6 (Well-founded relations)

- $<$ is well-founded over $\mathbb{N}$, but also over ordinal numbers.
- Lexicographic ordering on fixed-size lists is well-founded.
- Lexicographic ordering is not well-founded over $\{a,b\}^*$: $b \succ ab \succ aab \succ aaab \succ \ldots$

The following rule can be used to combine "leadsto" properties:

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

where d and e are rigid variables and d does not have free occurrences in G

(WFO) requires proving another "leads-to" property, typically by (WF1) or (SF1).

The premise "$(D, \prec)$ well-founded" is verified semantically (or in the host logic).

Exercise 4.7
Formally justify the correctness of the rule (WFO).

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

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

$$dist(d) \triangleq \text{IF } d \leq 12 \text{ THEN } 12 - d \text{ ELSE } 36 - d$$

$$d \prec e \triangleq dist(d) < dist(e)$$

Using (WFO), we have to prove

$$HC \wedge d \in 0..23 \Rightarrow (hr = d \wedge hr \neq 12 \leadsto hr = 12 \vee \exists e \in 0..23 : e \prec d \wedge hr = e)$$

This follows from the formula

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

shown earlier.

4.1.4 Simple temporal logic

The application of the verification rules is supported by laws of

- first-order logic,
- theories formalizing the data (set theory, arithmetic, graph theory, ...),
- and laws of temporal logic such as the following:

(STL1) $$\frac{F}{\Box F}$$

(STL2) $\Box F \Rightarrow F$

(STL3) $\Box\Box F \equiv \Box F$

(STL4) $\Box(F \Rightarrow G) \Rightarrow (\Box F \Rightarrow \Box G)$

(STL5) $\Box(F \wedge G) \equiv (\Box F \wedge \Box G)$

(STL6) $\Diamond\Box(F \wedge G) \equiv \Diamond\Box F \wedge \Diamond\Box G$

(TLA1) $$\frac{P \wedge t' = t \Rightarrow P'}{\Box P \equiv P \wedge \Box[P \Rightarrow P']_t}$$

(TLA2) $$\frac{I \wedge I' \wedge [A]_t \Rightarrow [B]_u}{\Box I \wedge \Box[A]_t \Rightarrow \Box[B]_u}$$

Note: validity of propositional temporal logic is mechanically decidable.

4.2 Algorithmic verification

Interactive theorem provers can assist deductive system verification.

☺ applicable in principle to arbitrary TLA specifications

☹ tedious to apply, needs much expertise

Finite-state models can be analyzed using state-space exploration
by running the TLA$^+$ model checker TLC.

The model is defined by a TLA$^+$ specification and a configuration file:

```
SPECIFICATION HC
INVARIANT HCini
PROPERTY HClive
```

```
TLC Version 2.0 of Mar 17, 2004

Model-checking

Parsing file HourClock.tla

Parsing file /.../Naturals.tla

Semantic processing of module Naturals

Semantic processing of module HourClock

Implied-temporal checking-satisfiability problem has 1 branches.

Finished computing initial states:  24 distinct states generated.

- - Checking temporal properties for the complete state space...

Model checking completed.  No error has been found.

   Estimates of the probability that TLC did not check all reachable states

   because two distinct states had the same fingerprint:

     calculated (optimistic):  3.122502256758253E-17

     based on the actual fingerprints:  6.063116922924556E-18

48 states generated, 24 distinct states found, 0 states left on queue.

The depth of the complete state graph search is 1.
```

Invariant checking systematically generate all reachable states

```
Set seen = new Set(); Set todo = new Set().addAll(Spec.getInitial());
while !todo.isEmpty() {
  State s = todo.removeSomeElement();
  if !seen.contains(s) {
    seen.add(s);
    if !s.satisfies(inv) { throw new InvariantViolated(s); }
    todo.addAll(Spec.getSuccessors(s));
} }
```

- depth-first search: organize `todo` as stack
  - $\leadsto$ stack contains counter-example if invariant is violated
- breadth-first search: organize `todo` as a queue (TLC implementation)
  - $\leadsto$ remember predecessor states to obtain shortest-length counter-example

Property checking : generalize to search for "acceptance cycles"

Syntactic restrictions for TLC

- TLC must be able to compute initial and successor states:
  - Action formulas are evaluated "from left to right".
  - The first occurrence of a primed flexible variables must be an "assignment"

    $x' = e$ or $x' \in S$ where S evaluates to a finite set

  - All flexible variables must be "assigned" some value.
  - Quantifiers must range over finite sets: $\forall x \in S : P$, $\exists x \in S : P$
- Analogous conditions apply to the initial state predicate.
- Module parameters must be instantiated by finite sets.

See Lamport's book for a detailed description.

What if the model is not finite-state or too large ?

- test: analyze small, finite instances
- approximate: write higher-level model
- abstract: soundness-preserving finite-state abstraction

Model checking : debugging rather than verification

Summary

- Validation: compare model to informal requirements (review, simulation)
- Verification: establish properties of formal model
- Deductive verification: invariants, fairness, well-founded relations
- TLA: most proof obligations are non-temporal formulas
- Combine verification steps using rules of temporal logic
- Algorithmic verification using TLC
  finite-state instances, counter-example on failure, great for debugging

5 The language TLA$^+$

TLA$^+$ is a specification language based on TLA that adds

- module structure (declarations, extension, instantiation)
- fixed first-order language and interpretation, based on set theory

TLA$^+$ is untyped: e.g., $5 = \text{TRUE}$ and $17 \wedge \text{“abc”}$ are well-formed formulas

— but we don’t know if they are true or false

Now: brief presentation of concepts necessary to understand TLA$^+$ models

See Lamport’s book for detailed exposition.

5.1 Specifying data in TLA$^+$

Every TLA$^+$ value is a set — cf. set-theoretical foundations of mathematics (ZFC)

From a logical perspective, the language of TLA$^+$ contains

- the binary predicate symbol $\in$
  (actually, TLA$^+$ also considers functions as primitive — see later)

- and the term formation operator $\text{CHOOSE}\ x : P$ (a.k.a. Hilbert's ε-operator) that denotes some (arbitrary, but fixed) value satisfying P if such a value exists — and any value otherwise

The choice operator

TLA$^+$ assumes a first-order interpretation with an (unspecified) choice function ε :

$$[\![\text{CHOOSE } x : P]\!]^{\xi}_{s,t} \quad = \quad \varepsilon(\{d : [\![P]\!]^{\xi[x:=d]}_{s,t} = \mathbf{tt}\})$$

Characteristic axioms:

$$(\exists x : P(x)) \quad \Rightarrow \quad P(\text{CHOOSE } x : P(x))$$

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

Examples:

$$(\text{CHOOSE } x : x \notin \mathit{ProcId}) \quad \notin \quad \mathit{ProcId}$$

$$(\text{CHOOSE } n : n \in \mathit{Nat} \wedge (n/2) * 2 = n) \quad = \quad (\text{CHOOSE } x : \exists k \in \mathit{Nat} : x = 2 * k)$$

$$(\text{CHOOSE } S : \forall z : z \in S \equiv z \notin z) \quad = \quad (\text{CHOOSE } x : x \in \{\}) \qquad \text{(cf. Russell's paradox)}$$

Choice vs. non-determinism

Consider the following actions specifying resource allocation:

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

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

- Both are enabled in precisely those states where the resource is free and there is some waiting process.
- $Alloc_{nd}$ produces as many successor states as there are waiting processes.
- $Alloc_{ch}$ produces a single successor state: it chooses some fixed process.

Constructions of elementary set theory in TLA+

$$
\begin{array}{rcl}
S \subseteq T & \triangleq & \forall x : x \in S \Rightarrow x \in T \\
\{e_1, \ldots, e_n\} & \triangleq & \text{CHOOSE } M : \forall x : x \in M \;\equiv\; (x = e_1 \vee \ldots \vee x = e_n) \\
\text{UNION } S & \triangleq & \text{CHOOSE } M : \forall x : x \in M \;\equiv\; (\exists T \in S : x \in T) \\
S \cup T & \triangleq & \text{UNION } \{S, T\} \\
S \cap T & \triangleq & \text{CHOOSE } M : x \in M \;\equiv\; (x \in S \wedge x \in T) \\
\text{SUBSET } S & \triangleq & \text{CHOOSE } M : \forall x : x \in M \;\equiv\; x \subseteq S \\
\{x \in S : P\} & \triangleq & \text{CHOOSE } M : \forall x : x \in M \;\equiv\; (x \in S \wedge P) \\
\{t : x \in S\} & \triangleq & \text{CHOOSE } M : \forall x : x \in M \;\equiv\; (\exists y \in S : x = t)
\end{array}
$$

- existence of these sets ensured by rules of ZF set theory

Functional values in TLA⁺

Some sets represent functions — TLA⁺ does not specify how

$[S \rightarrow T]$	set of functions with domain S and codomain T
$\text{DOMAIN}\, f$	domain of functional value f
$f[e]$	application of functional value f to expression e
$[x \in S \mapsto e]$	function with domain S mapping x to e
$[f \text{ EXCEPT } ![t] = e]$	function update
	$[f \text{ EXCEPT } ![t] = @ + e] = [f \text{ EXCEPT } ![t] = f[t] + e]$

f is a functional value iff $f = [x \in \text{DOMAIN}\, f \mapsto f[x]]$

Value of $f[x]$ is unspecified for $x \notin \text{DOMAIN}\, f$.

Recursive functions can be defined using choice, e.g.

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

This can be abbreviated to

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

- should justify existence of such a function
- no implicit commitment to, e.g., least fixed point semantics

Natural numbers defined using choice from Peano axioms

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

Predefined notation: $0 \triangleq \mathit{Zero},\ 1 \triangleq \mathit{Succ}[0],\ \ldots$

$$i..j \triangleq \{n \in \mathit{Nat} : i \leq n \land n \leq j\}$$

Integers and reals similarly defined as supersets of *Nat*, arithmetic operations agree

Tuples and sequences represented as functions

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

Some standard operations on sequences

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

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

Exercise 5.1

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

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

3. Give a recursive definition of the $\mathit{mergesort}$ function.

   Does $\mathit{sort} = \mathit{mergesort}$ hold for your definitions? Why (why not)?

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

Representation of strings: sequences of characters

standard operations on sequences apply to strings, e.g. $\text{“th”} \circ \text{“is”} = \text{“this”}$

Records: functions whose domain is a finite set of strings

short notation	instead of
$account.bal$	$account[\text{“bal”}]$
$[account \text{ EXCEPT } !.bal = @ + sum]$	$[account \text{ EXCEPT } ![\text{“bal”}] = @ + sum]$
$[num \mapsto 1234567, bal \mapsto -321.45]$	$[f \in \{\text{“num”}, \text{“bal”}\} \mapsto$ $\text{IF } f = \text{“num” THEN } 1234567$ $\text{ELSE } -321.45]$

5.2 TLA$^+$ modules

A TLA$^+$ module consists of a sequence of

– declarations   of constant and variable parameters
– definitions   of operators (non-recursive)
– assertions   of assumptions and theorems

- Modules serve as units of structuring: they provide scopes for identifiers.
- They form a hierarchy by extending or instantiating other modules.
- The meaning of any symbol is obtained by replacing definitions by their bodies.

Principle of unique names

Identifiers that are active in the current scope cannot be redeclared or redefined — not even as bound variables.

MODULE *IllegalModule*

EXTENDS *Naturals*

CONSTANTS x, y

$m + n$	$\triangleq$	$\ldots$	* attempt to redefine operator $+$ defined in *Naturals*
$Foo(y, z)$	$\triangleq$	$\exists x : \ldots$	* x and y already declared as constant parameters
Nat	$\triangleq$	LET $y \triangleq \ldots$ IN $\ldots$	* clashes of *Nat* (from *Naturals*) and y (parameter)

Import of the same module via different paths is allowed.

Definitions can be protected from export by the LOCAL keyword.

Module extension

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

Module *Foo* exports

- the symbols declared or defined in module *Foo* *and*
- the symbols (of global scope) exported by modules *Bar*, *Baz*

Module *Foo* may use, but not redefine or declare symbols exported by *Bar*, *Baz*.

Module instantiation allows for import with renaming:

$$\textsc{module}\ \mathit{Component}$$

$$\mathit{InChan} \triangleq \textsc{instance}\ \mathit{Channel}\ \textsc{with}\ \mathit{Data} \leftarrow \mathit{Message}, \mathit{chan} \leftarrow \mathit{in}$$

$$\mathit{Chan}(c) \triangleq \textsc{instance}\ \mathit{Channel}\ \textsc{with}\ \mathit{Data} \leftarrow \mathit{Message}, \mathit{chan} \leftarrow c$$

The operators defined in module *Channel* can be used in *Component* as follows:

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

- Identity renaming can be omitted
- Name for the instantiation can be omitted if only one copy is needed
- LOCAL instantiation is possible

Module *Component* does not export symbols declared or defined in module *Channel*.

Summary

- TLA^+ : complete specification language based on TLA and set theory
- untyped formalism: every value is a set
- rich data structures definable via set-theoretic constructions
- module structure to decompose specifications

6 Refinement, hiding, and composition

So far: specifications of components at a single level of abstraction.

This chapter:

- compare different levels of abstraction: refinement of runs
- composition of components to build a system
- hiding (encapsulation) of internal state components

These concepts are represented by logical connectives in TLA^+

6.1 Refinement

Example 6.1 (hour and minute clock, see example 1.1)

MODULE *HourMinuteClock*

EXTENDS *Naturals*, *HourClock*

VARIABLE *min*

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

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

$$Hr \triangleq (min = 59 \land HCnxt) \lor (min < 59 \land hr' = hr)$$

$$HMCnxt \triangleq Min \land Hr$$

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

THEOREM $HMC \Rightarrow HC$

HMC implies the hour clock specification *HC* : stuttering invariance.

Refinement is represented in TLA as (validity of) implication: $\models HMC \Rightarrow HC$

- $HMC \Rightarrow HCini$: obvious
- $HMC \Rightarrow \Box[HCnxt]_{hr}$: immediate from definition, formal proof via (TLA2)
- $HMC \Rightarrow \mathrm{WF}_{hr}(HCnxt)$: informal argument obvious, formally supported by rule

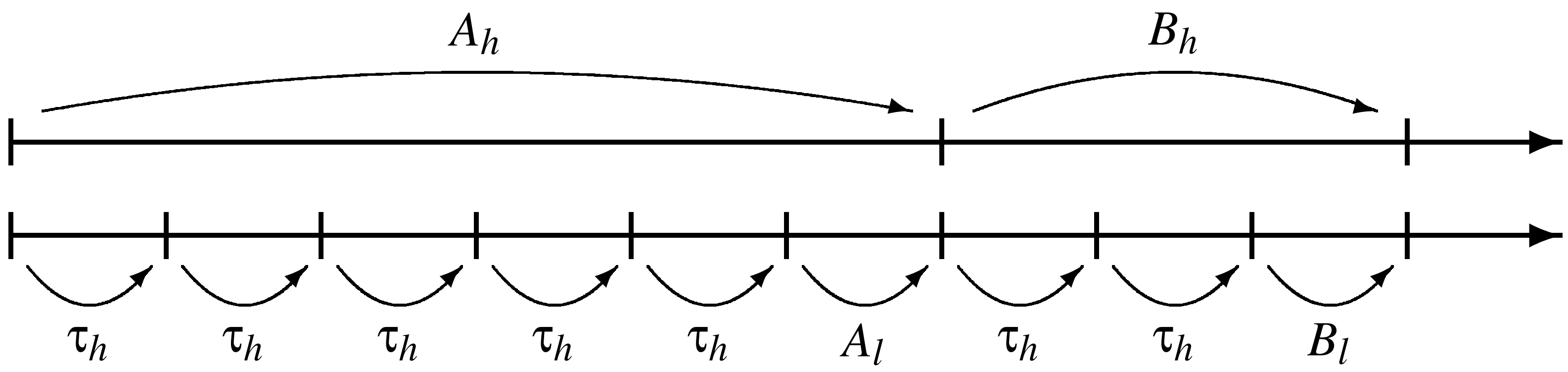
$$\text{(WF2)}\quad \frac{\begin{array}{c}\langle N \wedge P \wedge A\rangle_v \Rightarrow \langle B\rangle_w \\ P \wedge \textsc{Enabled}\ \langle B\rangle_w \Rightarrow \textsc{Enabled}\ \langle A\rangle_v \\ \Box[N \wedge [\neg B]_w]_v \wedge \mathrm{WF}_v(A) \wedge \Box F \wedge \Diamond\Box\textsc{Enabled}\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

Exercise:

1. formally prove that the hour-minute clock refines the hour clock
2. verify the refinement using TLC

Refinement as implication (trace inclusion)

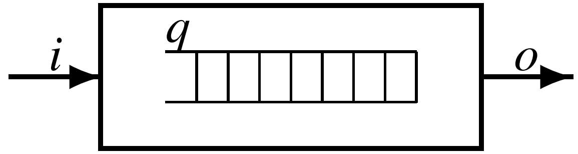
- refinement may add state variables ("implementation detail")
- high-level actions decomposed into sequence of low-level actions:
  - actions except last one do not affect high-level state
  - final action corresponds to high-level effect



- stuttering invariance is crucial to make this work
- fairness condition must also be preserved to ensure liveness properties
- branching structure not preserved: implementation may reduce non-determinism

6.2 Hiding of state components

Reminder: queue specification



MODULE *SyncInterleavingQueue*

EXTENDS *Sequences*

VARIABLES *i,o,q*

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

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

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

$$SIQNext \triangleq SIQEnq \lor SIQDeq$$

$$SIQLive \triangleq \mathrm{WF}_{i,q,o}(SIQDeq)$$

$$SIQSpec \triangleq SIQInit \land \Box[SIQNext]_{i,q,o} \land SIQLive$$

The internal queue q is an "implementation detail", not part of the interface.

The FIFO should behave *as if there were* an internal queue.

Hiding expressed in TLA by existential quantification over flexible variables:

MODULE *SIQueue*

VARIABLES i, o

$IntQueue(q) \triangleq$ INSTANCE *SyncInterleavingQueue*

$SIQueue \triangleq \exists\!\!\exists\, q : IntQueue(q)!SIQSpec$

We therefore extend the syntax of temporal formulas:

- If F is a formula and v is a flexible variable then $\exists\!\!\exists\, v : F$ is a formula.

Intuitive meaning:

The TLA formula $\exists\!\!\exists\, v : F$ holds of behavior σ
if F holds of some τ that differs from σ only in the valuations of v.

Naive semantics of flexible quantification

$$\sigma, \xi \models \exists v : F \text{ iff } \tau, \xi \models F \text{ for some } \tau =_v \sigma$$

Problem: $F \triangleq v = 0 \land \Box[v = 1]_w$

F asserts that v has changed before first change of w.

F holds of τ, and therefore $\exists v : F$ holds of σ.

$\exists v : F$ would not hold of behavior obtained from σ by removing second state.

Violation of stuttering invariance!

Correct semantics of quantification: "build in" stuttering invariance

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

- Both σ and τ satisfy $\exists v : v = 0 \wedge \Box[v = 1]_w$

  In fact, this formula is valid!

- For the clock example, we have $\models HC \Rightarrow \exists min : HMC$

  Intuition: the implementation of an hour clock can use a hidden minute display

Although the semantics is more complicated, the usual proof rules are sound:

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

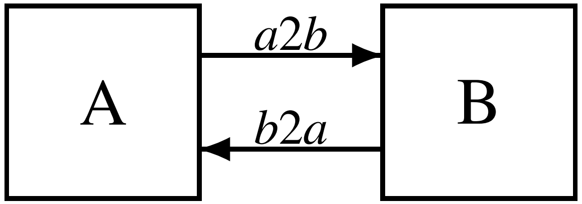
$$(\exists\text{-E})\quad \frac{F \Rightarrow G}{(\exists v : F) \Rightarrow G}\quad (v \text{ not free in } G)$$

For completeness, more introduction rules needed ("history", "prophecy" variables).

6.3 Composition of specifications

Reactive and distributed systems: parallel composition of components

Common variables $a2b$ and $b2a$ represent interface (rename internal variables).



Assuming that runs of components are described by $ASpec$ and $BSpec$, the formula

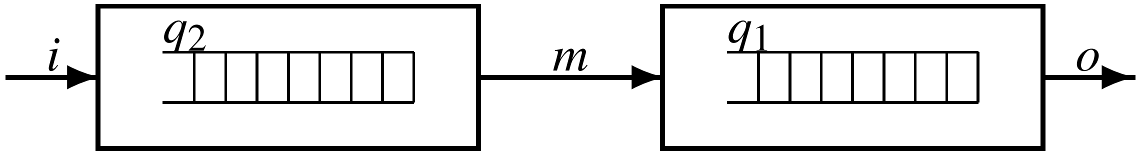
$$ASpec \wedge BSpec$$

specifies the two components running in parallel.

Problem: each formula must allow changes to interface due to other component.

$\Longrightarrow$ Sometimes need additional conjuncts to express “synchronization”.

Example 6.2 (composition of two FIFOs)



MODULE *TwoQueues*

VARIABLES i, m, o

$LeftQ \triangleq$ INSTANCE *SIQueue* WITH $o \leftarrow m$

$RightQ \triangleq$ INSTANCE *SIQueue* WITH $i \leftarrow m$

$LongQ \triangleq LeftQ!SIQueue \land RightQ!SIQueue$

INSTANCE *SIQueue*

THEOREM $LongQ \Rightarrow SIQueue$??

Problem: *LongQ* allows for simultaneous enqueueing and dequeueing, but *SIQueue* does not. Interleaving assumption has to be asserted explicitly.

Proof: $LongQ \land \Box[i' = i \lor o' = o]_{i,o} \Rightarrow SIQueue$ (outline)

$$
\begin{aligned}
LongQ \quad &\equiv \quad \land\ \exists q : IntQueue(i,q,m)!SIQSpec \\
& \qquad \land\ \exists q : IntQueue(m,q,o)!SIQSpec \\
SIQueue \quad &\equiv \quad \exists q : IntQueue(i,q,o)!SIQSpec
\end{aligned}
$$

Using rules ($\exists$-E) and ($\exists$-I), we have to show

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

for some state function t. The proof succeeds for $t \triangleq q_1 \circ q_2$.

Exercise: formally carry out this proof.

Summary

- Refinement: successively add implementation detail during system development
- TLA: represent refinement as implication (stuttering invariance!)
- Hiding of internal state components via existential quantification
- Composition of sub-systems represented as conjunction

7 Case study: a resource allocator

- A set of clients compete for a (finite) set of resources.
- Whenever a client holds no resources and has no outstanding requests, he can request a set of resources. (No request may exceed the entire set of resources.)
- The allocator can allocate a set of available resources to a client that requested them, possibly without completely satisfying the client's request.
- Clients can 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.

7.1 A first solution

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

$Resources \setminus (\text{UNION } \{alloc[c] : c \in Clients\})$

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

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

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

* Client c requests set S of resources, provided it has no outstanding requests and no allocated resources.

$Request(c,S) \triangleq$

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

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

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

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

$Allocate(c,S) \triangleq$

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

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

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

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

$\land\ S \neq \{\} \land S \subseteq alloc[c]$

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

$\land$ UNCHANGED $unsat$

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

$\exists c \in Clients, S \in \text{SUBSET } Resources :$

$Request(c,S) \lor Allocate(c,S) \lor Return(c,S)$

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

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

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

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

$\land\ \forall c \in Clients : \mathrm{SF}_{vars}(\exists S \in \text{SUBSET } Resources : Allocate(c,S))$

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

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

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

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

THEOREM $SimpleAllocator \Rightarrow \Box ResourceMutex$

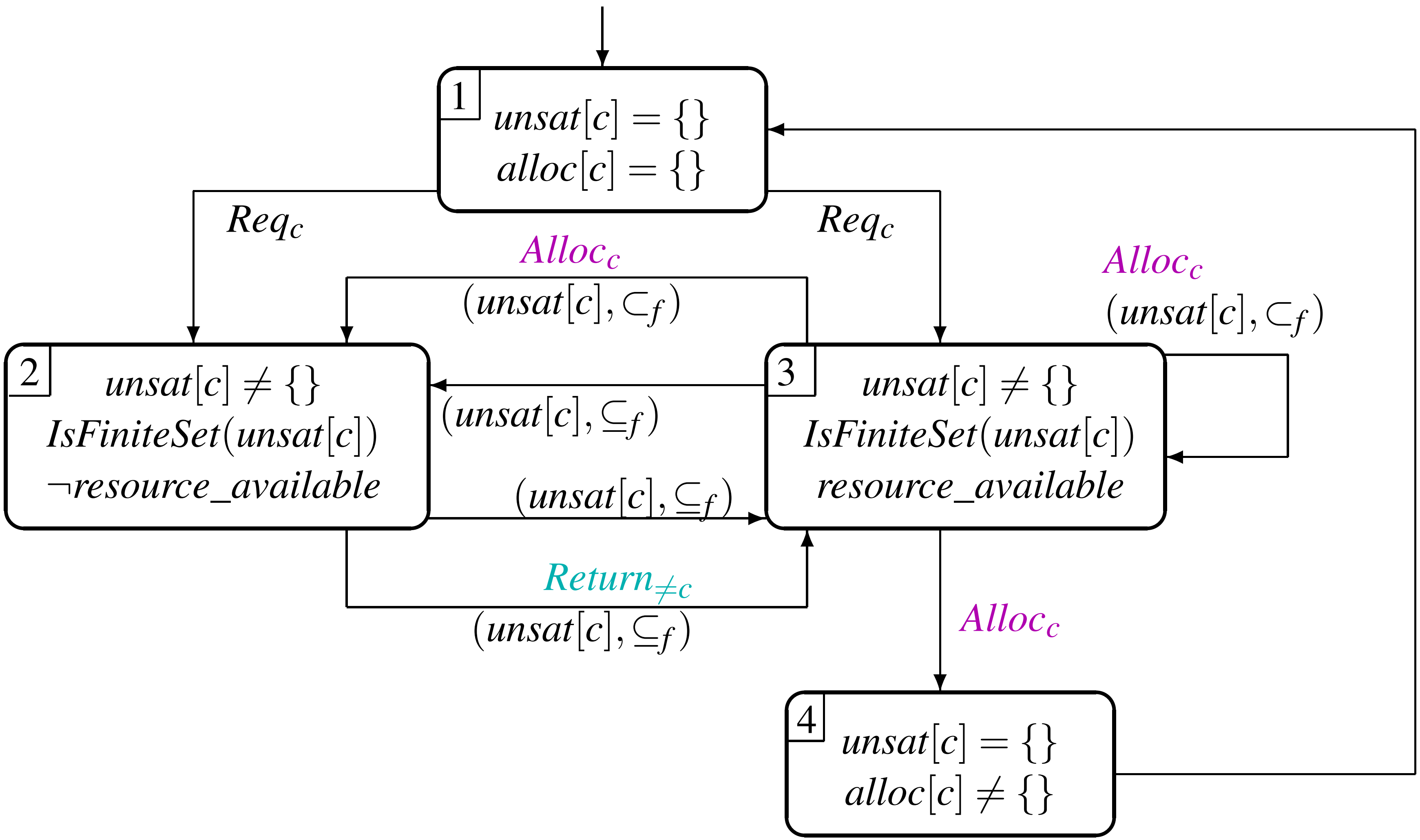
THEOREM $SimpleAllocator \Rightarrow ClientsWillFree$

THEOREM $SimpleAllocator \Rightarrow ClientsWillObtain$

THEOREM $SimpleAllocator \Rightarrow InfOftenSatisfied$

All three theorems are verified by TLC (for small sets *Clients* and *Resources*).

Verification of *InfOftenSatisfied* via Boolean abstraction



MODULE *SimpleAllocatorDiagram*

EXTENDS *SimpleAllocator*

CONSTANTS c * Skolem constant for verification

ASSUME $c \in Clients$

$$unsat\_c\_empty \triangleq unsat[c] = \{\}$$

$$unsat\_c\_finite \triangleq IsFiniteSet(unsat[c])$$

$$alloc\_c\_empty \triangleq alloc[c] = \{\}$$

$$resource\_available \triangleq unsat[c] \cap available \neq \{\}$$

$$Req_c \triangleq \exists S \in \text{SUBSET } Resources : Request(c, S)$$

$$Alloc_c \triangleq \exists S \in \text{SUBSET } Resources : Allocate(c, S)$$

$$Return_c \triangleq Return(c, alloc[c])$$

$$\begin{aligned} Return_{\neq c} \triangleq\ & \wedge unsat[c] \neq \{\} \wedge \neg resource\_available \\ & \wedge \exists d \in Clients : d \neq c \wedge alloc[d] \cap unsat[c] \neq \{\} \wedge Return(d, alloc[d]) \end{aligned}$$

$$S \subseteq_f T \triangleq IsFiniteSet(S) \wedge S \subseteq T$$

$$S \subset_f T \triangleq S \subseteq_f T \wedge S \neq T$$

The specification *SimpleAllocator* is wrong.

The fairness condition

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

requires clients to return resources even if their entire request has not been satisfied.

The specification *SimpleAllocator* is wrong.

The fairness condition

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

requires clients to return resources even if their entire request has not been satisfied.

Solution: weaken fairness condition and require

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

With the new fairness condition, the implication

$$SimpleAllocator \Rightarrow ClientsWillObtain$$

is no longer valid (and TLC produces a counter-example).

7.2 Second solution

Idea: allocator keeps a schedule of clients with pending requests
such that all requests can be completely satisfied, under worst-case assumptions.

- Resource *r* will be allocated to client *c* only if *c* appears in the schedule and if no client that appears before *c* in the schedule requires it.
- Upon issuing a request, clients are put in a pool of clients with pending requests.
- The allocator eventually appends its schedule with clients from the pool
  (in arbitrary order)

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)

$TypeInvariant \triangleq$

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

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

$\land\ pool \in \text{SUBSET}\ Clients$

$\land\ sched \in Seq(Clients)$

$PermSeqs(S) \triangleq$ * Permutation sequences of finite set S.

$\text{LET } perms[ss \in \text{SUBSET } S] \triangleq$

$\quad\quad \text{IF } ss = \{\} \text{ THEN } \langle\rangle$

$\quad\quad \text{ELSE LET } ps \triangleq \left[x \in ss \mapsto \left\{Append(sq, x) : sq \in perms[ss \setminus \{x\}]\right\}\right]$

$\quad\quad\quad\quad \text{IN } \quad \text{UNION } \{ps[x] : x \in ss\}$

$\text{IN } \quad perms[S]$

$Drop(seq, i) \triangleq$ * remove element at position i from sequence seq

$SubSeq(seq, 1, i-1) \circ SubSeq(seq, i+1, Len(seq))$

$available \triangleq$ * set of resources free for allocation

$Resources \setminus (\text{UNION } \{alloc[c] : c \in Clients\})$

$Init \triangleq$

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

$\land\ pool = \{\} \land sched = \langle\rangle$

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

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

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

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

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

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

$\land\ S \neq \{\} \land S \subseteq alloc[c]$

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

$\land\ \text{UNCHANGED } \langle unsat, pool, sched \rangle$

$Allocate(c,S) \triangleq$

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

$$
\begin{aligned}
&\land S \neq \{\} \land S \subseteq available \cap unsat[c] \\
&\land \exists i \in 1..Len(sched) : \; \land sched[i] = c \\
&\qquad\qquad\qquad\qquad\quad \land \forall j \in 1..i-1 : unsat[sched[j]] \cap S = \{\} \\
&\qquad\qquad\qquad\qquad\quad \land sched' = \text{IF } S = unsat[c] \text{ THEN } Drop(sched, i) \text{ ELSE } sched \\
&\land alloc' = [alloc \text{ EXCEPT } ![c] = @ \cup S] \\
&\land unsat' = [unsat EXCEPT ![c] = @ \setminus S] \\
&\land \text{UNCHANGED } pool
\end{aligned}
$$

$Schedule \triangleq$ * The allocator extends its schedule by the processes from the pool.

$$
\begin{aligned}
&\land pool \neq \{\} \\
&\land \exists sq \in PermSeqs(pool) : sched' = sched \circ sq \\
&\land pool' = \{\} \\
&\land \text{UNCHANGED } \langle unsat, alloc \rangle
\end{aligned}
$$

$$Next \triangleq \quad \text{* The next-state relation.}$$

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

$$\lor Schedule$$

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

$$SchedulingAllocator \triangleq$$

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

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

$$\land \forall c \in Clients : \mathrm{WF}_{vars}(\exists S \in \text{SUBSET } Resources : Allocate(c, S))$$

$$\land \mathrm{WF}_{vars}(Schedule)$$

The scheduling allocator satisfies the correctness requirements.

Crucial invariant: request of any scheduled client can be satisfied from the resources that will be available after previously scheduled clients released the resources they held:

$$\forall i \in 1..Len(sched) :$$
$$\begin{aligned} unsat[sched[i]] \subseteq \quad & available \\ \cup \quad & \text{UNION } \{unsat[sched[j]] \cup alloc[sched[j]] : j \in 1..i-1\} \\ \cup \quad & \text{UNION } \{alloc[c] : c \in UnscheduledClients\} \end{aligned}$$

where $\quad UnscheduledClients \;\triangleq\; Clients \setminus \{sched[i] : i \in 1..Len(sched)\}$

In fact, the scheduling allocator is a refinement of the simple allocator.

MODULE *AllocatorRefinement*

EXTENDS *SchedulingAllocator*

$Simple \triangleq$ INSTANCE *SimpleAllocator*
$SimpleAllocator \triangleq Simple!SimpleAllocator$

THEOREM $SchedulingAllocator \Rightarrow SimpleAllocator$

Due to the schedule, the weaker fairness requirement of the clients implies the original one because the allocator can guarantee that each client will eventually receive the resources it asked for.

7.3 Towards an implementation

Next goals:

- distinguish local states of clients and allocator
- introduce explicit message passing between processes

Idea:

- variables $unsat, alloc, pool, sched$ represent allocator state
- add variables
  - $requests \in [Clients \rightarrow \text{SUBSET } Resources]$
  - $holding \in [Clients \rightarrow \text{SUBSET } Resources]$

  to represent clients' view of system state
- distinguish originating and receiving part of actions

Scheduling approach as before, now focus on distribution and communication

MODULE *AllocatorImplementation*

EXTENDS *FiniteSets, Sequences, Naturals*

CONSTANTS *Clients, Resources*

ASSUME *IsFiniteSet(Resources)*

VARIABLES

```
    unsat,      \* unsat[c] : allocator's view of pending requests of client c
    alloc,      \* alloc[c] : allocator's view of resources allocated to c
    pool,       \* set of clients with pending requests that have not been scheduled
    sched,      \* schedule (sequence of clients)
    requests,   \* request[c] : client c's view of pending requests
    holding,    \* holding[c] : client c's view of allocated resources
    network     \* set of messages in transit
```

Sched $\triangleq$ INSTANCE *SchedulingAllocator*

$$Messages \triangleq [type : \{\text{"request"}, \text{"allocate"}, \text{"return"}\},$$
$$clt : Clients,$$
$$rsrc : \text{SUBSET } Resources]$$

$$TypeInvariant \triangleq$$
$$\land\ Sched!TypeInvariant$$
$$\land\ requests \in [Clients \rightarrow \text{SUBSET } Resources]$$
$$\land\ holding \in [Clients \rightarrow \text{SUBSET } Resources]$$
$$\land\ network \in \text{SUBSET } Messages$$

$$Init \triangleq$$
$$\land\ Sched!Init$$
$$\land\ requests = [c \in Clients \mapsto \{\}]$$
$$\land\ holding = [c \in Clients \mapsto \{\}]$$
$$\land\ network = \{\}$$

$$Request(c, S) \triangleq$$

$$\land\ requests[c] = \{\} \land holding[c] = \{\}$$

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

$$\land\ network' = network \cup \{[type \mapsto \text{“request”}, clt \mapsto c, rsrc \mapsto S]\}$$

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

$$Rreq(m) \triangleq$$

$$\land\ m \in network \land m.type = \text{“request”}$$

$$\land\ unsat' = [unsat \text{ EXCEPT } ![m.clt] = m.rsrc]$$

$$\land\ pool' = pool \cup \{m.clt\}$$

$$\land\ network' = network \setminus \{m\}$$

$$\land\ \text{UNCHANGED } \langle alloc, sched, requests, holding \rangle$$

$$Allocate(c, S) \triangleq$$
$$\land\ Sched!Allocate(c, S)$$
$$\land\ network' = network \cup \{[type \mapsto \text{"allocate"}, clt \mapsto c, rsrc \mapsto S]\}$$
$$\land\ \text{UNCHANGED}\ \langle requests, holding \rangle$$

$$RAlloc(m) \triangleq$$
$$\land\ m \in network \land m.type = \text{"allocate"}$$
$$\land\ holding' = [holding\ \text{EXCEPT}\ ![m.clt] = @ \cup m.rsrc]$$
$$\land\ requests' = [requests\ \text{EXCEPT}\ ![m.clt] = @ \setminus m.rsrc]$$
$$\land\ network' = network \setminus \{m\}$$
$$\land\ \text{UNCHANGED}\ \langle unsat, alloc, pool, sched \rangle$$

$$Return(c, S) \triangleq \ \ldots$$
$$RRet(m) \triangleq \ \ldots$$
$$Schedule \triangleq Sched!Schedule \land \text{UNCHANGED}\ \langle requests, holding, network \rangle$$

$$Next \triangleq$$

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

$$\lor \exists m \in network : RReq(m) \lor RAlloc(m) \lor RRet(m)$$

$$\lor Schedule$$

$$vars \triangleq \langle unsat, alloc, pool, sched, requests, holding, network \rangle$$

$$Specification \triangleq$$

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

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

$$\land \forall c \in Clients : \mathrm{WF}_{vars}(\exists S \in \text{SUBSET } Resources : Allocate(c, S))$$

$$\land \mathrm{WF}_{vars}(Schedule)$$

$$\land \forall m \in Messages : \mathrm{WF}_{vars}(RReq(m)) \land \mathrm{WF}_{vars}(RAlloc(m)) \land \mathrm{WF}_{vars}(RRet(m))$$

THEOREM $Specification \Rightarrow Sched!SchedulingAllocator$

TLC produces a counter-example:

1. Client $c1$ returns a resource it is holding
2. Client $c1$ requests the same resource again
3. Allocator handles the request before the return message

This error reflects a typical race condition!

Possible solutions:

- use FIFO communication between processes
- strengthen pre-condition of $RReq(m)$ by conjunct

$$alloc[m.clt] = \{\}$$

Correctness of refinement relies on following invariant:

$$
\begin{aligned}
&\mathit{RequestsInTransit}(c) \triangleq \quad \text{* requests sent by } c \text{ but not yet received}\\
&\quad \{msg.rsrc : msg \in \{m \in network : m.type = \text{“request”} \land m.clt = c\}\}\\
&\mathit{AllocsInTransit}(c) \triangleq \quad \text{* allocations sent to } c \text{ but not yet received}\\
&\quad \{msg.rsrc : msg \in \{m \in network : m.type = \text{“allocate”} \land m.clt = c\}\}\\
&\mathit{ReturnsInTransit}(c) \triangleq \quad \text{* return messages sent by } c \text{ but not yet received}\\
&\quad \{msg.rsrc : msg \in \{m \in network : m.type = \text{“return”} \land m.clt = c\}\}\\
&\mathit{Invariant} \triangleq \forall c \in Clients :\\
&\quad \land\ Cardinality(RequestsInTransit(c)) \leq 1\\
&\quad \land\ requests[c] = unsat[c] \cup (\text{UNION } RequestsInTransit(c)) \cup (\text{UNION } AllocsInTransit(c))\\
&\quad \land\ alloc[c] = holding[c] \cup (\text{UNION } AllocsInTransit(c)) \cup (\text{UNION } ReturnsInTransit(c))
\end{aligned}
$$

Exercise: verify this invariant

The last specification describes a distributed system, distinguishing between local state of different processes and assigning responsability for actions.

However, it is not written as a composition of specifications!

- In principle, this can be done in TLA^+ (exercise!)
- Some issues:
  - state representation (function vs. collection of scalar variables)
  - interleaving vs. non-interleaving composition
- Moreover, TLC does not (yet) support multiple next-state relations.

The process structure is in the eye of the beholder

Summary

- Development of a not-so-small case study
- Verification of properties does not ensure correctness of specification
- Fairness can be tricky (especially for environment)
- Refinement helps to focus on a single problem at a time
- Delay decomposition and communication after main algorithm
- Writing “monolithic” models is easiest in TLA^+

Summary

- TLA formulas express specifications and properties of (transition) systems.
- System verification reduces to proof of formulas.

  Verification rules try to reduce temporal conclusions to non-temporal hypotheses.
- Structural concepts are represented using logical connectives:

refinement	implication
composition	conjunction
hiding	existential quantification

  Stuttering invariant semantics makes this representation possible.
- TLA$^+$: specification language designed around TLA and ZFC set theory.
- Tool support: TLC, model checker for high-level specifications

Thank you!