Temporal Logic of Actions (TLA)
(a brief introduction)

Shmuel Katz
Computer Science Department
The Technion
Basic Idea

• Combine transitions with temporal logic
• One notation that allows easy reasoning
• Powerful enough for both safety and liveness
• Appropriate for open systems: describing a component that has other components “competing” on shared memory
• Key idea: do certain actions OR keep certain parts of the state unchanged
• Weak and strong fairness assumptions are used to guarantee needed liveness properties (but do not influence safety properties)
Open versus Closed systems

• A Closed system has all components inside the specification, and a simple interface to an Environment

• An Open system has “unknown” assignments, messages, events at any time. Much harder to specify and show correct, but sometimes necessary.

• When do operations take effect?
Overlapping Can Be Tricky

- Initially $x = 0$
- Execute $x := x+1$ in one component, in parallel with $x := x+2$ in another.

- What are the possible results?

- The answer can depend on what is considered ‘atomic’

- Many systems today have multiple processes (sometimes with time sharing)
Shared Memory Concurrency

• Multiple processes that change the same variables (operate over the same state)

• Have to consider what happens DURING an implementation....input/output is not enough.

• New issues:
  > waiting,
  > deadlock,
  > mutual exclusion,
  > starvation
Stuttering states in TLA

• Actions outside the “group of interest” can change part of the state, but leave “critical parts” unchanged—useful for specifying open systems

• Also good for refinement---many implementation steps might not change the abstract state, until a critical moment when the state changes
  > Inserting can rearrange array implementing a set, without changing the set, and then put in a new element that DOES change the abstract set.

• When there are overlapping inserts, deletes, and membership operations---need critical moment when a new element is considered “in”
Two kinds of “variables” and TLA basics

• **Flexible** variables: represent the system state, can assume different values in different states

• **Static** variables: like in mathematics...have an unknown but fixed value (for constants and logical variables that remember initial values)

• **Action A**: like an operation schema, but just a formula

\[ A = v \in \mathbb{Z} \land v' \in \mathbb{Z} \land v' < 0 \]

• Use regular box $\Box$ or diamond $\Diamond$ with linear view

• **ENABLED A** is just like “pre A” of Z (hide all primed flexible variables)
More TLA notations

• \([A]_t\) transition that either satisfies A, or keeps value of t unchanged…equivalent to \(A \lor (t' = t)\)

• \(\langle A\rangle_t\) transition that does an action of A and changes the value of t …..\(A \land t' \neq t\)

• Can use the above along with regular \(\Box\) or \(\Diamond\)
  \(\Box P\) or \(\Box [A]_t\) (=“only A changes t’”)

\(\Box [ q' = \text{Append}(q, \text{in}) ] \_ q\)
On liveness assumptions and fairness

- **Basic liveness assumption**: if actions are ENABLED, one of them will occur (something happens somewhere—true even in sequential)

- **Weak fairness for A**, denoted $WF_t(A)$ is
  \[
  \square \Diamond \neg \text{ENABLED } \langle A \rangle_t \lor \square \Diamond \langle A \rangle_t
  \]
  \[
  \Diamond \square \text{ENABLED } \langle A \rangle_t \Rightarrow \square \Diamond \langle A \rangle_t
  \]
  > Common assumption in parallel systems...means each processor has basic liveness assumption
  > Even in uniprocessors, most schedulers guarantee it
  > Really means, “enabled until taken” is enough....

- **Strong fairness for A**, denoted $SF_t(A)$, is
  \[
  \square \Diamond \text{ENABLED } \langle A \rangle_t \Rightarrow \square \Diamond \langle A \rangle_t
  \]
Fairness and Safety Properties

• Claim: A fairness assumption can help for proving liveness properties, but does not help with safety properties!

• A strange situation: A fairness assumption removes some computations from consideration, but not any of the states or prefixes in those computations..

• Consider a repeated op b in one processor, in parallel with a second processor that stops execution of both by doing a. WF on a guarantees termination.
• Without it, b’s might occur forever....
A FIFO queue

• Connecting two channels \textit{in} and \textit{out} with an internal queue \( q \)
• \( \text{NoMsg} = \text{CHOOSE} x : x \notin \text{Message} \)
• \( \text{Init} \equiv q = \langle \rangle \land \text{in} = \text{NoMsg} \land \text{out} = \text{NoMsg} \)
• \( \text{Enq} \equiv \text{in} \in \text{Message} \land q' = \text{Append}(q, \text{in}) \land \text{in'} = \text{NoMsg} \land \text{out'} = \text{out} \)
• \( \text{Deq} \equiv q \neq \langle \rangle \land \text{out'} = \text{Head}(q) \land q' = \text{Tail}(q) \land \text{in'} = \text{in} \)
• \( \text{Next} \equiv \text{Enq} \lor \text{Deq} \)
• \( \text{vars} \equiv \langle \text{in}, \text{out}, q \rangle \)
• \( \text{InternalFIFO} \equiv \text{Init} \land \Box[\text{Next}]_{\text{vars}} \land \text{WF}_{\text{vars}}(\text{Deq}) \)
On liveness assertions

- Note: the FIFO queue only requires Weak Fairness on the Deq operation. Is this correct?
- +: Disallows only adding to the queue forever, even when Deq can continuously be done
- -: Hard to see whether Enq must occur (reasoning: Deq can only occur a finite number of times before the queue is empty, and then either nothing happens or Enq occurs)
- The WF property allows showing that whenever the queue is nonempty and Deq is enabled (never withdrawn), it will eventually be done.
- □ ( ENABLED [Deq] => ◊ ⟨Deq⟩_{var} )
Faults in Spec.: A Lossy Queue

• Exactly like a regular one, but with one more allowed change:

\[
\text{Erase} \equiv q \neq \langle \rangle \land q' = \text{Tail}(q) \land in' = in \land out' = out
\]

Erase models a fault that loses messages
Now have: Next \(\equiv\) Enq \(\lor\) Deq \(\lor\) Erase

• Problem: the liveness property of Deq no longer follows from WF(Deq) : Maybe we always erase elements…(so Deq is not continuously enabled)
A Compromise

• “No liveness constraint on Deq” leads to a useless lossy queue.

• “Always succeeding with Deq if there is an element in q” is too demanding.

• Suggested compromise: strong fairness

\[
(\Box \Diamond \text{ENABLED} [\text{Deq}]) \Rightarrow \Box \Diamond \langle \text{Deq} \rangle_{\text{out}}
\]

• If the queue **repeatedly** is nonempty, eventually the Deq will succeed.
A Channel is a Lossy Queue

• Replace Enq by Snd (send)

• Replace Deq by Rcv (receive)

• The Snd action is initiated in one computer connected to the channel and the Rcv in another at the other end, with the new allowed transition modeling the possible fault of losing a message in transmission.

• Strong fairness means we won’t always lose messages.
A Key Property

- In a channel (a lossy queue), if we keep transmitting the same message, it will get through!

\[ (\Box (\Box (\Diamond \langle \text{Snd} \rangle_{\text{var}} \land \Box (\text{in}' = m) \land \\
\Box \Diamond \text{ENABLED}[\text{Rcv}] ) ) \Rightarrow \Diamond \langle \text{Rcv} \rangle_{\text{out}} \land \text{out}' = m ) \]

- Can’t be stuck waiting for Rcv, because then the queue must be empty, contradicting the liveness of Snd. Thus complete Rcv repeatedly, and each time remove an element. Must get to m eventually.....
Implementing Fault Tolerance

• A **reliable channel**: specified by original queue with SEND for Enq and RECEIVE for Deq

• **Problem**: Implement a reliable channel by imposing a Communication Protocol on two lossy queues *(code review in the tirgul)*

• **Example uses** *alternating bit protocol*

• Sender module has a value to send in val, and uses msgQ to send (val, bit) and ackQ to receive bit.

• Receive module uses msgQ to receive (val,bit) pairs, and ackQ to send bit.
Alternating Bit Protocol

SEND

msgQ

rcvd

sBit

sAck

ACK

rcvd

rBit

Sender

Receiver

lossy queue

lossy queue

reliable queue
Connecting two levels

- Exernal user initiates SEND (at one end) and RECEIVE (at the other) for a correct FIFO channel
- Protocol uses two lossy channels to get a single message “across the faulty gap”
- Sender module puts messages from SEND in a local queue sq [in tirgul—of size 1, called sent]
- Receiver module puts messages that crossed the gap in a local queue rq waiting for RECEIVE requests [in tirgul—of size 1, called rcvd]
- Note: version in tirgul just treats getting a single message across the “faulty gap”
Hints to Understanding

• sq has the values SENT that have not yet been moved across the lossy channel.

• rq has the values received after crossing the lossy channel, before RECEIVE operations remove them.

sBit equals the bit currently being sent

rBit equals the bit currently being received on the other side.
Hints (continued)

• **Sender**: Keep sending the same (value, bit) pair from the head of sq until you get back an ACK with that bit (into sAck), then switch your bit, erase the value from sq and start sending the new bit with the next value.

• **Receiver**: When you get a pair with a new bit put the value in the local rq, switch your bit to the new one, and send an ACK back. Ignore additional copies with the same bit you have, but send back more ACKs with your bit.

• Initially, the bits are different on each side.
The Mapping

• Key to understanding:

\[ q \equiv \text{if } s\text{Bit} \neq r\text{Bit} \]

\[ \text{then } rq \triangleq sq \quad (=\text{sending a new bit}) \]

\[ \text{else } rq \triangleq \text{tail}(sq) \quad (=\text{arrived, but still sending}) \]

• The mapping changes whenever \( s\text{Bit} \) or \( r\text{Bit} \) changes-- but the RESULT of the mapping will stay the same!

(check cases where that happens)

• SEND and RECEIVE change the abstract queue

• Using the lossy implementation does NOT
Liveness of the queue

• Need to show that SEND and RECEIVE satisfy the usual queue liveness properties

• Therefore, need that elements move across the lossy queue, so sq has room, and rq can provide elements when queue has room/ isn’t empty (resp.)

• Use reasoning by contradiction on the temporal properties.
Summary on Fault-Tolerance

• Can model faults as ‘possible changes’

• Temporal logic gives global liveness properties not expressed in other ways.

• Reasoning can be confusing, need care in choosing temporal assertions and mappings.

• Open systems are especially hard to specify and verify, need overlap and repeating states.
TLA Today

• An active Microsoft project, with
  > TLA+ for easier specifications
  > PlusCal algorithm language (C-like) that can be translated into TLA by a compiler
  > TLAPS a Proof System for TLA
  > TLA Toolbox an integrated IDE with all of the above

• Intended for “delicate” multiprocessor key algorithms and protocols, and open systems

• To find resources, Google for: Temporal Logic of Actions