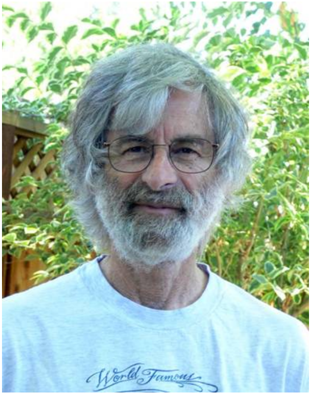
Programming Distributed Systems

Modelling and validating distributed systems with TLA+

Annette Bieniusa

FB Informatik
TU Kaiserslautern

TLA+: Specification language



Source: https://commons.wikimedia.org/wiki/File:Leslie_Lamport.jpg

- Formal language for describing and reasoning about distributed and concurrent systems
- TLA+ is a model-oriented language
  - Based on mathematical logic and set theory plus temporal logic TLA (temporal logic of actions)
  - Supported by the TLA Toolbox, an IDE that integrates model-checker and theorem prover

Overview

- Example: 1-bit clock
- TLA+ language constructs
- Safety and liveness properties
  - Executions and Traces
  - Fairness
- Example: Specifying broadcast algorithms

Goals of this Learning Path

In this learning path, you will learn how

- to read TLA+ specifications
- to encode specify safety and liveness properties in TLA
- to check specifications and find counterexamples
- to model broadcast algorithms in TLA+

Example: 1-bit Clock

First example: 1-bit Clock

- A behavior is a sequence of states, where a state is an assignment of values to variables.
- Possible behavior of 1-bit Clock:
  - `b = 1 -> b = 0 -> b = 1 -> b = 0 -> ...`
  - `b = 0 -> b = 1 -> b = 0 -> b = 1 -> ...`

Formal description:

- State variable: `b`
- Initial predicate: `b = 1 \/ b = 0`
- Next-step action (`b'` denotes the variable at the next state)
  - `\/ (b = 0) /\ (b' = 1)`
  - `\/ (b = 1) /\ (b' = 0)`
  - Meaning: `IF b = 0 THEN b' = 1 ELSE b' = 0`

1-bit Clock: TLA Specification

```
-------- MODULE OneBitClock ----------

VARIABLE b

Init == (b = 0) \/ (b = 1)

Next == \/ b = 0 /\ b' = 1
        \/ b = 1 /\ b' = 0

Spec == Init /\ [][Next]_<<b>>

======================================
```

- The initial state satisfies `Init`
- Every transition satisfies `Next` or leaves `b` unchanged
  - `[Next]_<<b>> == Next \/ (b' = b)`
- `b'` denotes value of `b` after transition

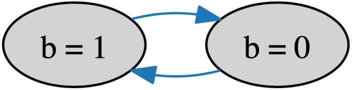
1-bit Clock: Type invariant

```
-------- MODULE OneBitClock ----------
VARIABLE b

Init == (b = 0) \/ (b = 1)

TypeInv == b \in {0,1}

Next == \/ b = 0 /\ b' = 1
        \/ b = 1 /\ b' = 0

Spec == Init /\ [][Next]_<<b>>
--------------------------------------

THEOREM Spec => []TypeInv

======================================
```

- TLA+ is untyped to keep math formulas simple
- Theorem here specifies an **invariant** property

Computing all possible behaviors

- State graph is a directed graph G
- Algorithm sketch:
  1. Put the set of all initial states into G
  2. For every state $s \in G$, compute all possible states t such that $s \to t$ is a possible step in a behaviour
  3. For every state t found in step 2 with $t \notin G$, add an edge from s to t
  4. Repeat from 2 until no new states or edges can be added to G



TLC: State model checker for TLA+

- Exhaustive breath-first search of all reachable states
- Finds (one of) the shortest path to the property violation

Diameter Number of states in the longest path of G with no repeated states

States found Total number of states it examined in step 1 and 2

Distinct states Number of states that form the set of nodes of G

Queue size Number of states s in G for which step 2 has not yet been done

Let’s check our 1-bit clock specification!

More on TLA+

Structure of TLA+ Modules - Part 1

```
------------------- MODULE M ---------------------------
EXTENDS M1,..., Mn
\* Incorporates the declarations, definitions, assumptions,
\* and theorems from the modules named M1,...,Mn into the
\* current module.

CONSTANTS C1,..., Cn
\* Declares the C1,..., Cn to be constant parameters.

ASSUME P
\* Asserts P as an assumption.

VARIABLES x1,..., xn
\* Declares x1,..., xn as variables.
```

Structure of TLA+ Modules - Part 2

```
TypeInv == exp \* Declares the types of variables x1,..., xn.

Init == exp \* Initializes variables x1,..., xn.

F(x1,..., xn) == exp
\* Defines F to be an operator such that
\* F(e1,...,en) equals exp with each identifier xk replaced by ek.

f[x \in S] == exp
\* Defines f to be the function with domain S such that
\* f[x] = exp for all x in S.
\* The symbol f may occur in exp, allowing a recursive definition.

THEOREM P
\* Asserts that P can be proved from the definitions and
\* assumptions of the current module.
=========================================================
```

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Propositional and Predicate Logic

```
TRUE
FALSE

~(a /\ b \/ c)
a => b
Next == b' = 0

\A x \in {1, 2, 3, 4, 5} : x >= 0
\E x \in {1, 2, 3, 4, 5} : x % 2 = 0
```

Functions

```
[i \in {2,3,5,9} |-> i - 7]
    = (2 :> -5 @@ 3 :> -4 @@ 5 :> -2 @@ 9 :> 2)

DOMAIN [i \in {2,3,5,9} |-> i - 7]
    = {2, 3, 5, 9}

[ [i \in {2,3,5,9} |-> i - 7][3] = -4

[ {2,4} -> { "a", "b" } ]
    = { (2 :> "a" @@ 4 :> "a"), (2 :> "a" @@ 4 :> "b"),
        (2 :> "b" @@ 4 :> "a"), (2 :> "b" @@ 4 :> "b") }

[ [i \in {2,3,5,9} |-> i - 7] EXCEPT ![2]= 12 ]
    = (2 :> 12 @@ 3 :> -4 @@ 5 :> -2 @@ 9 :> 2)
```

Records

```
[node |-> "n1", edge |-> "e1"]

[node |-> "n1", edge |-> "e1"].edge = "e1"

[nodes : {"n1","n2"}, edges : {"e1","e2"}]

[node |-> "n1", edge |-> "e1"] EXCEPT !.edge = "xpto"]
    = [node |-> "n1", edge |-> "xpto"]
```

Tuples

```
<<"ana", 32, 37495>>

<<"ana",32>>[2] = 32

<<"ana",32>>[1] = "ana"

{1,2,3} \times {"a","b"}
    = { <<1, "a">>, <<1, "b">>, <<1, "c">>,
        <<2, "a">>, <<2, "b">>, <<2, "c">>,
        <<3, "a">>, <<3, "b">>, <<3, "c">> }
```

Technische Universität Kaiserslautern

Sets

```
S = {1, 2, 3}

S /= {1, 2, 3}      S # {1, 2, 3}

x \in S
x \notin S

S \union {1, 2, 3}

{ n \in {1, 2, 3, 4, 5} : n % 2 != 0 } = {1, 3, 5}
{ 2*n+1 : n \in {1, 2, 3, 4, 5} } = {3, 5, 7, 9, 11}

UNION { {1, 2}, {2, 3}, {3, 4} } = {1, 2, 3, 4}
SUBSET {1, 2} = {{}, {1}, {2}, {1, 2}}
```

Sequences

```
-------- MODULE Sequences -------------------------
LOCAL INSTANCE Naturals

Seq(S) == UNION {[1..n -> S] : n \in Nat}

Len(s) == CHOOSE n \in Nat : DOMAIN s = 1..n

s \o t == [i \in 1..(Len(s) + Len(t)) |->
    IF i \leq Len(s) THEN s[i] ELSE t[i-Len(s)]]

Append(s, e) == s \o <<e>>

Head(s) == s[1]

Tail(s) == [i \in 1..(Len(s)-1) |-> s[i+1]]

SubSeq(s, m, n) == [i \in 1..(1+n-m) |-> s[i+m-1]]
===================================================
```

CHOOSE operator

```
CHOOSE x \in S : P(x)
\* Equals some value v in S such that P(v) equals true, if such a
    value exists.
\* Its value is unspecified if no such v exists.

CHOOSE x \in {1, 2, 3, 4, 5} : TRUE
CHOOSE x \in {1, 2, 3, 4, 5} : x % 2 = 0
```

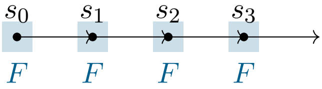
Specifying Safetey and Liveness Properties with Temporal Logic

Temporal Properties

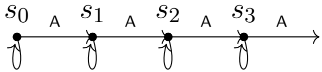
- Examples:
  - Does an algorithm always terminate?
  - If disrupted, will a system return to a stable state eventually?
- Amir Pnueli introduced in 1977 the use of temporal logic for describing system behaviors
- TLA is a variant tailored for systems
  - *Action formulas* describe states and state transitions
  - *Temporal formulas* describe state sequences (traces)
- Temporal operators
  - `[] F` : F is always true
  - `<> F` : F is eventually true
  - `F ~> G` : F leads to G

[] F : F is always true

- Formula `[]F`, where `F` is a state predicate, is true iff `F` is true in every state of the behavior

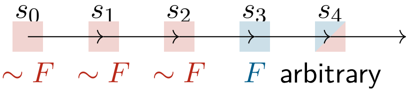


- Recall: Formula `[][A]_<<e>>`, where `A` is an action and `e` a state function, is true iff every successive step is an `[A]_<<e>>` step



<> F : F is eventually true

- Formula `<>F`, where `F` is a state predicate, is true iff `F` will be true in some state
- P is not always false
    - `<>P == ~[](~P)`



F ∼> G: F leads to G

- Whenever F is true, then G is eventually true
  - `F ~> G == [](F => <>G)`

- Every request leads to a response: `request ~> response`

Examples

- `[]<>F` : Infinitely often ⇒ Progress
  - At all times, `F` is true then or at some later time
  - e.g. the traffic light is green infinitely often
- `<>[]F` : Eventually always ⇒ Stability
  - Eventually, `F` becomes true and remains true from then on
  - e.g. eventually all messages are delivered

Fairness

Fairness

- To prove liveness properties, it is necessary to make some assumptions about the system environment
- If a transition is “often enough” enabled, it should at some point happen (**fairness**)
- TLA has two forms of fairness:
  - Strong fairness for action A: `SF_<<e>> (A)`
  - Weak fairness for action A: `WF_<<e>> (A)`

Weak Fairness `WF_<<e>> (A)`

- `(<>[] ENABLED <A>_<<e>>) => ([]<> <A>_<<e>>)`
- If A ever becomes forever enabled, then an A step must eventually occur
- Weak fairness of A asserts that an A step must eventually occur if A is continuously enabled
  - "continuously" = without interruption
- Example: Traffic light
  - If the traffic light is weakly fair, it will eventually turn green, the red, etc.
  - But if the car waiting for the light is only weakly fair, it might never move!

Strong Fairness SF_<<e>> (A)

- ([]<> ENABLED <A>_<<e>>) => ([]<> <A>_<<e>>)
- If A is infinitely often enabled, then infinitely many A steps occur
- Strong fairness of A asserts that an A step must eventually occur if A is continually enabled
  - “continually” = repeatedly, possible with interruptions
- Example: Traffic light
  - A strongly fair car will eventually move even if the light keeps switching
  - Beware: Requires the light to be weakly fair!

In practice

- Temporal properties are powerful, but can be confusing
  - Using adhoc formulas is error prone
  - Use uniform way with fairness properties
- Checking liveness properties is slow
  - Invariant checks can be parallelized by TLC
  - Restrict your model to small instances
- Liveness properties are often not needed, but having TLA+ as tool is handy!