INTRODUCTION TO TLA

Presented by: Kevin Yeh
What is TLA+

- Specification Language for modelling complex or concurrent systems
- TLA+ toolbox performs model checks to check for correctness
- PlusCAL
What can TLA+ do for you?

• Modelling of ALGORITHMS prior to implementation
• Meant as a supplement to traditional test/verification
• Very powerful bug detection
What can TLA+ do for you?

- Been used successfully at Amazon, HP, and Intel
- Two weeks before value was added

### Applying TLA+ to some of our more complex systems

<table>
<thead>
<tr>
<th>System</th>
<th>Components</th>
<th>Line count (excl. comments)</th>
<th>Benefit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Background redistribution of data</td>
<td>645 PlusCal</td>
<td>Found 1 bug, and found a bug in the first proposed fix.</td>
</tr>
<tr>
<td>DynamoDB</td>
<td>Replication &amp; group-membership system</td>
<td>939 TLA+</td>
<td>Found 3 bugs, some requiring traces of 35 steps</td>
</tr>
<tr>
<td>EBS</td>
<td>Volume management</td>
<td>102 PlusCal</td>
<td>Found 3 bugs.</td>
</tr>
<tr>
<td>Internal distributed</td>
<td>Lock-free data structure</td>
<td>223 PlusCal</td>
<td>Improved confidence. Failed to find a liveness bug as we did not check liveness.</td>
</tr>
<tr>
<td>lock manager</td>
<td>Fault tolerant replication and reconfiguration algorithm</td>
<td>318 TLA+</td>
<td>Found 1 bug. Verified an aggressive optimization.</td>
</tr>
</tbody>
</table>
Intangibles

• Requires up-front system understanding
• Adds value even after production release
TLA+ an Overview

• 4 parts to a specification
  • Initial predicate
  • Possible “Next” states
  • Safety Properties
  • Liveness Properties
Alternating One-bit Clock

• Initial Predicate
  • \((b = 0) \lor (b = 1)\)

• Next States
  • \(((b = 0) \land (b' = 1)) \lor ((b = 1) \land (b' = 0))\)
Alternating One-bit Clock

- **Initial Predicate**
  - \((b = 0) \lor (b = 1)\)

- **Next States**
  - \(((b = 0) \land (b' = 1)) \lor ((b = 1) \land (b' = 0))\)

\[
\begin{align*}
\text{VARIABLE } & b \\
\text{Init} == & (b = 0) \lor (b = 1) \\
\text{Next} == & \lor /\ b = 0 \\
& /\ b' = 1 \\
& \lor /\ b = 1 \\
& /\ b' = 0
\end{align*}
\]
Die Hard Problem

- What you have: 3-gallon jug, 5-gallon jug, and a faucet
- Goal: Measure 4 gallons
Die Hard Problem

VARIABLES big, small

Init == \big = 0
\small = 0

Next == \FillSmall
\FillBig
\EmptySmall
\EmptyBig
\SmallToBig
\BigToSmall
Die Hard Problem

VARIABLES big, small

Init == /
\ big = 0
\ small = 0

Next == \ FillSmall
\ FillBig
\ EmptySmall
\ EmptyBig
\ SmallToBig
\ BigToSmall

FillSmall == \ small’ = 3
\ big’ = big

SmallToBig == V \ big+ small >5
\ big’ = 5
\ small’ = small – (5-big)
V \ big + small <= 5
\ big’ = big + small
\ small’ = 0
Model Checker

• Builds up a Directed Graph of all possible states.
Die Hard - Solution

**Invariants**
Formulas true in every reachable state.

- big # 4

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Initial predicate&gt;</td>
<td>State (num = 1)</td>
</tr>
<tr>
<td>big</td>
<td>0</td>
</tr>
<tr>
<td>small</td>
<td>0</td>
</tr>
<tr>
<td>&lt;Action line 15, col 12 State (num = 2)</td>
<td></td>
</tr>
<tr>
<td>big</td>
<td>5</td>
</tr>
<tr>
<td>small</td>
<td>0</td>
</tr>
<tr>
<td>&lt;Action line 35, col 7 State (num = 3)</td>
<td></td>
</tr>
<tr>
<td>big</td>
<td>2</td>
</tr>
<tr>
<td>small</td>
<td>3</td>
</tr>
<tr>
<td>&lt;Action line 18, col 15 State (num = 4)</td>
<td></td>
</tr>
<tr>
<td>big</td>
<td>2</td>
</tr>
<tr>
<td>small</td>
<td>0</td>
</tr>
<tr>
<td>&lt;Action line 35, col 7 State (num = 5)</td>
<td></td>
</tr>
<tr>
<td>big</td>
<td>0</td>
</tr>
<tr>
<td>small</td>
<td>2</td>
</tr>
<tr>
<td>&lt;Action line 15, col 12 State (num = 6)</td>
<td></td>
</tr>
<tr>
<td>big</td>
<td>5</td>
</tr>
</tbody>
</table>

Select line in Error Trace to show its value here.
Safety/Liveness Properties

- Safety Property – Define a correct behavior of your procedure
  - Partial Correctness: (terminated) => (Correct_Output)

- Liveness Property – Define a correct behavior that must eventually hold
  - Termination
Euclid’s Algorithm – a high level view

• Find the Greatest Common Divisor of two numbers
• General Procedure:
  • PlusCAL -> TLA+
  • Write the definition of GCD using set logic: GCD(m,n)
  • Use definition to write Safety/Liveness Properties
• This is how TLA+ is used in industry
Euclid’s Algorithm

• PlusCAL code:

```plaintext
--algorithm Euclid
variables x = M, y = N;
while (x ≠ y) {
  if(x<y) { y := y-x }
  else { x := x - y }
}

};
```
/* BEGIN TRANSLATION
VARIABLES x, y, pc

vars == << x, y, pc >>

Init == (* Global variables *)
\x = M
\y = N
\pc = "Lbl_1"

Lb1_1 == \pc = "Lb1_1"
\ IF x ≠ y
  THEN \ IF x<y
   THEN \ y' = y-x
     \ x' = x
     \ y' = y
  ELSE \ x' = x - y
  ELSE \ pc' = "Lb1_1"
ELSE \ pc' = "Done"
\ UNCHANGED "<< x, y >>"
Next == Lb1_1

(* Disjunct to prevent deadlock on termination *)
(pc = "Done" \ UNCHANGED vars)
Model Checking

Safety

- **Invariants**
  Formulas true in every reachable state.
  
  \( (pc = "Done") \implies (x = y) \land (x = \text{GCD}(M,N)) \)

Liveness

- **Properties**
  Temporal formulas true for every possible behavior.
  
  - **Termination**

0 BUGS!
Questions?