Weeks of debugging can save you hours of TLA$^+$

An informal introduction to a formal method

Markus A. Kuppe

Microsoft Research

June 12, 2018

Who are you?

- Show of hands who
  - ...has ever used formal methods
  - ...regularly uses formal methods
  - ...has ever used TLA$^+$

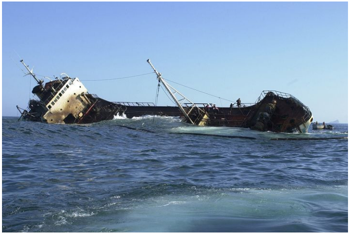
Setting the stage

Downtime of customer-facing services:
...during high-season of the year
...everybody is on vacation

=> Management unhappy because
we lost millions

We are in good company

IBM Bluemix (01/2017),
Facebook (02/2017),
Amazon AWS (02/2017),
Microsoft Azure (03/2017),
Microsoft Office 365 (03/2017),
Apple iCloud (06/2017),
...



The tale of the blocking BlockingQueue

- Post mortem analysis identifies a deadlock in BlockingQueue* as root cause
- Deadlock never showed during excessive testing
  - Despite high (unit) test coverage
- Lucky to manually reproduce the deadlock after days of testing
  - We still do not have a fix

*BlockingQueue example originally by Charpentier [2017].

The tale of the blocking BlockingQueue

```java
public final class BlockingQueue<E> {

  private final E[] store;

  public BlockingQueue(final int capacity) {
    this.store = (E[]) new Object[capacity];
  }

  public final synchronized void put(final E e) {
    while (isFull()) {
      wait();
    }
    notify();
    append(e);
  }

  public final synchronized E take() {
    while (isEmpty()) {
      wait();
    }
    notify();
    return head();
  }
  /* helper methods and some fields omitted */
}
```

TLA$^+$ to the rescue

In a presentation a colleague told us about the TLA$^+$ methodology

Demo

Let's specify BlockingQueue with TLA^+

BlockingQueue in PlusCal

module *BlockingQueuePCal*

variable *store* $= \langle\rangle$; $k = 1$; *waitset* $= \{\}$; $c = \{$"c1", "c2"$\}$; $p = \{$"p1"$\}$;

BlockingQueue in PlusCal

module *BlockingQueuePCal*

variable $store = \langle\rangle$; $k = 1$; $waitset = \{\}$; $c = \{\text{"c1"}, \text{"c2"}\}$; $p = \{\text{"p1"}\}$;

define {
$$isEmpty \triangleq Len(store) = 0$$
$$isFull \triangleq Len(store) = k$$
}

BlockingQueue in PlusCal

module *BlockingQueuePCal*

variable $store = \langle\rangle$; $k = 1$; $waitset = \{\}$; $c = \{\text{"c1"}, \text{"c2"}\}$; $p = \{\text{"p1"}\}$;

define {
$isEmpty \triangleq Len(store) = 0$
$isFull \triangleq Len(store) = k$
}

macro $wait(\;)$ { $waitset := waitset \cup \{self\}$ }

BlockingQueue in PlusCal

module *BlockingQueuePCal*

variable $store = \langle\rangle$; $k = 1$; $waitset = \{\}$; $c = \{$"c1", "c2"$\}$; $p = \{$"p1"$\}$;

define {
  $isEmpty \triangleq Len(store) = 0$
  $isFull \triangleq Len(store) = k$
}

macro $wait(\ )$ { $waitset := waitset \cup \{self\}$ }

macro $notify(\ )$ {
  if ($waitset \neq \{\}$) {
    with ($w \in waitset$) {
      $waitset := waitset \setminus \{w\}$;
    }
  }
}

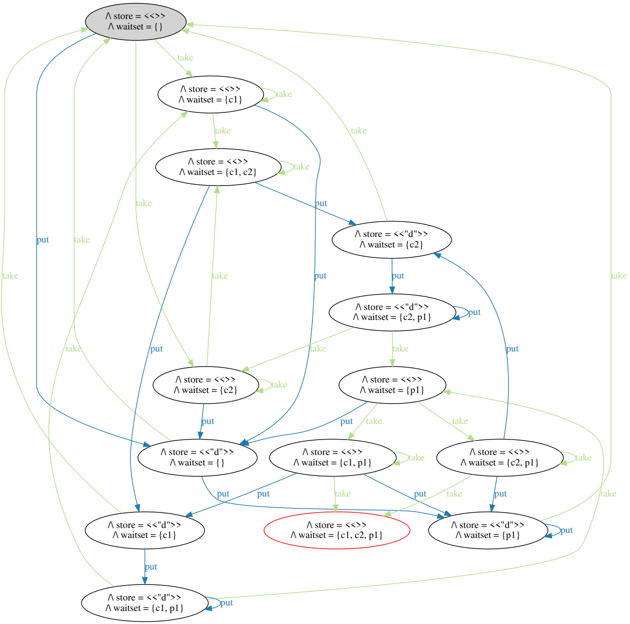
BlockingQueue in PlusCal

```
module BlockingQueuePCal
variable store = ⟨⟩ ; k = 1 ; waitset = {} ; c = {"c1", "c2"} ; p = {"p1"} ;

define {
      isEmpty ≜ Len(store) = 0
      isFull ≜ Len(store) = k
}

macro wait( ) { waitset := waitset ∪ {self} }

macro notify( ) {
      if ( waitset ≠ {} ) {
            with ( w ∈ waitset ) {
                  waitset := waitset \ {w} ;
            }
      }
}

process ( producer ∈ p ) {
      put:    while ( true ) {
                    if ( isFull ) { wait() ; }
                    else { notify() ; store := Append(store, self) ; } ;
              } ;
}
```

BlockingQueue in PlusCal

```
module BlockingQueuePCal
variable store = ⟨⟩ ; k = 1 ; waitset = {} ; c = {"c1", "c2"} ; p = {"p1"} ;

define {
      isEmpty ≜ Len(store) = 0
      isFull ≜ Len(store) = k
 }

macro wait( ) { waitset := waitset ∪ {self} }

macro notify( ) {
        if ( waitset ≠ {} ) {
               with ( w ∈ waitset ) {
                     waitset := waitset \ {w} ;
                }
         }
 }

process ( producer ∈ p ) {
        put :     while ( true ) {
                           if ( isFull ) { wait() ; }
                           else { notify() ; store := Append(store, self) ; } ;
                  } ;
 }

process ( consumer ∈ c ) {
      take :     while ( true ) {
                          if ( isEmpty ) { wait() ; }
                          else { notify() ; store := Tail(store) ; } ;
                 } ;
 }
```

State Graph


/\ store = <<>> /\ waitset = {}
/\ store = <<>> /\ waitset = {c1}
/\ store = <<>> /\ waitset = {c1, c2}
/\ store = <<"d">> /\ waitset = {c2}
/\ store = <<"d">> /\ waitset = {c2, p1}
/\ store = <<>> /\ waitset = {c2}
/\ store = <<>> /\ waitset = {p1}
/\ store = <<"d">> /\ waitset = {}
/\ store = <<>> /\ waitset = {c1, p1}
/\ store = <<>> /\ waitset = {c2, p1}
/\ store = <<"d">> /\ waitset = {c1}
/\ store = <<>> /\ waitset = {c1, c2, p1}
/\ store = <<"d">> /\ waitset = {p1}
/\ store = <<"d">> /\ waitset = {c1, p1}
take
put

Invariants

module *BlockingQueuePCal*

```
variable store = ⟨⟩ ; k = 1 ; waitset = {} ; c = {"c1", "c2"} ; p = {"p1"} ;

define {
    Inv ≜ waitset ≠ (c ∪ p)      <= UP HERE!

    isEmpty ≜ Len(store) = 0
    isFull ≜ Len(store) = k
}

macro wait( ) { waitset := waitset ∪ {self} }

macro notify( ) {
    if ( waitset ≠ {} ) {
        with ( w ∈ waitset ) {
            waitset := waitset \ {w} ;
        }
    }
}

process ( producer ∈ p ) {
    put:    while ( true ) {
                if ( isFull ) { wait() ; }
                else { notify() ; store := Append(store, self) ; } ;
            } ;
}

process ( consumer ∈ c ) {
    take:   while ( true ) {
                if ( isEmpty ) { wait() ; }
                else { notify() ; store := Tail(store) ; } ;
            } ;
}
```

Error Trace

TLC Errors

Model_1

```
Invariant waitset # ProcSet is violated.
```

Error-Trace Exploration

Error-Trace

Name	Value
<Initial predicate>	State (num = 1)
buffer	<< >>
waitset	{}
<take line 70, col 15 to line 81, c	State (num = 2)
buffer	<< >>
waitset	{c1}
<take line 70, col 15 to line 81, c	State (num = 3)
buffer	<< >>
waitset	{c1, c2}
<put line 55, col 14 to line 66, co	State (num = 4)
buffer	<<"d">>
waitset	{c2}
<put line 55, col 14 to line 66, co	State (num = 5)
buffer	<<"d">>
waitset	{c2, p1}
<take line 70, col 15 to line 81, c	State (num = 6)
buffer	<< >>
waitset	{p1}
<take line 70, col 15 to line 81, c	State (num = 7)
buffer	<< >>
waitset	{c1, p1}
<take line 70, col 15 to line 81, c	State (num = 8)
buffer	<< >>
waitset	{c1, c2, p1}

Input Space

```
module BlockingQueuePCal
variable store = ⟨⟩ ; k ∈ 1 . . 6 ; waitset = {} ;
         c ∈ subset {"c1", "c2", "c3", "c4"} \ {{}} ;   This
         p ∈ subset {"p1", "p2", "p3", "p4"} \ {{}} ;   This

define {
    Inv ≜ waitset ≠ (c ∪ p)

    isEmpty ≜ Len(store) = 0
    isFull ≜ Len(store) = k
}
macro wait( ) { waitset := waitset ∪ {self} }
    macro notify( ) {
     if ( waitset ≠ {} ) {
        with ( w ∈ waitset ) {
            waitset := waitset \ {w} ;
        }
     }
}
process ( producer ∈ p ) {
    put:    while ( true ) {
                 if ( isFull ) { wait() ; }
                 else { notify() ; store := Append(store, self) ; } ;
            } ;
}
process ( consumer ∈ c ) {
    take:   while ( true ) {
                 if ( isEmpty ) { wait() ; }
                 else { notify() ; store := Tail(store) ; } ;
            } ;
}
```

Relation Capacity K, Consumer C and Producer P

Deadlock iff:

$$2K < |C| + |P|$$

TLA$^+$

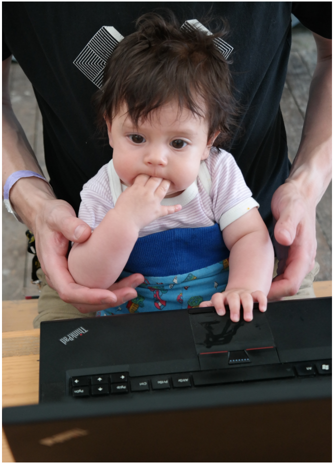
- TLA$^+$ - Temporal Logic of Actions - is a formal specification language developed by Leslie Lamport
- Design, model, document, and verify concurrent and distributed systems
- TLA$^+$ has been described as exhaustively-testable pseudocode [Newcombe, 2011]
- Successfully used by Microsoft, Intel, [DEC/Compaq 2003], … for e.g. Paxos, Cosmos DB, [Raft 2016], qspinlock…

Amazon Success Story in Detail

- DynamoDB: scalable high-performance "no SQL" data store with cross datacenter replication and strong consistency guarantees
- First informal proofs and excessive (fault-injecting) testing
- TLC found very subtle bug: shortest error trace 35 steps
- "*Using TLA*$^+$ *in place of traditional proof writing would thus likely have* ***improved time to market****, in addition to achieving* ***greater confidence*** *in the system's correctness.*" [Newcombe et al., 2015]

Now beer?

Everybody statisfied?



Be Suspicious of Success!

```
                    module BlockingQueuePCal
variable store = ⟨⟩ ; k ∈ 1 . . 6 ; waitset = {} ;
          c ∈ subset {"c1", "c2", "c3", "c4"} \ {{}} ;
          p ∈ subset {"p1", "p2", "p3", "p4"} \ {{}} ;

define {
    Inv ≜ waitset ≠ (c ∪ p)

    isEmpty ≜ Len(store) = 0
    isFull ≜ Len(store) = k
 }
macro wait( ) { waitset := waitset ∪ {self} }
    macro notify( ) {
     if ( waitset ≠ {} ) {
          with ( w ∈ waitset ) {
              waitset := waitset \ {w} ;
           }
      }
 }
process ( producer ∈ p ) {
     put :    while ( false ) {       Ouch!!!
                  if ( isFull ) { wait() ; }
                  else { notify() ; store := Append(store, self) ; } ;
          } ;
 }
process ( consumer ∈ c ) {
    take :     while ( true ) {
                  if ( isEmpty ) { wait() ; }
                  else { notify() ; store := Tail(store) ; } ;
          } ;
 }
```

Doing nothing is always safe!

- TLA^+ behavioral properties [Lamport, 1977]
  - *Safety* properties: Something *bad* never happens
  - *Liveness* properties: Something *good* eventually happens

Temporal Logic is really simple... kind of

- TLA^+ has just two temporal operators:
  - $\Diamond P$ *(pronounced Diamond)*: P is true at some point of a behavior
    - $\neg P, \neg P, \neg P, \ldots, \boldsymbol{P}, \neg P, \neg P, \ldots$
  - $\Box P$ (pronounced *Box*): P is always true
    - $P, P, P, P, P, P, \ldots$

Temporal Logic is really simple... kind of

- TLA$^+$ has just two operators:
  - $\Diamond P$ *(pronounced Diamond)*: P is true at some point of a behavior
    - $\neg P, \neg P, \neg P, \ldots, \boldsymbol{P}, \neg P, \neg P, \ldots$
  - $\Box P$ (pronounced *Box*): P is always true
    - $P, P, P, P, P, P, \ldots$

  - $\Diamond\Box P \approxeq \neg P, \neg P, \neg P, \neg P, \boldsymbol{P}, \boldsymbol{P}, \boldsymbol{P}, \boldsymbol{P}, \ldots$
  - $\Box\Diamond P \approxeq \neg P, \neg P, \neg P, \boldsymbol{P}, \neg P, \neg P, \neg P, \boldsymbol{P}, \boldsymbol{P}, \neg P, \neg P, \neg P, \boldsymbol{P}, \ldots$

All $p \cup c$ eventually serviced

module *BlockingQueuePCal*

...

process (*producer* $\in$ *p*) {
    put: **while** (false) {
        if (*isFull*) { *wait*() ; }
        else { *notify*() ; *store* := *Append*(*store*, *self*) ; } ;
    } ;
}

...

$$
Prop \triangleq \begin{array}{l} \land \forall \, con \in c : \Box\Diamond(\langle take(con) \rangle_{vars}) \\ \land \forall \, pro \in p : \Box\Diamond(\langle put(pro) \rangle_{vars}) \end{array}
$$

Fairness

Weak If the action $A \wedge (f' \neq f)$ ever becomes enabled and *remains enabled forever*, then infinitely many $A \wedge (f' \neq f)$ steps occur.
$(\Box\Diamond\neg \mathit{ENABLED}\,\langle A\rangle_e) \vee (\Box\Diamond\langle A\rangle_e)$

Strong If the action $A \wedge (f' \neq f)$ is enabled infinitely often, then infinitely many $A \wedge (f' \neq f)$ steps must occur. If an action ever becomes enabled forever, then it is enabled infinitely often.
$(\Diamond\Box\neg \mathit{ENABLED}\,\langle A\rangle_e) \vee (\Box\Diamond\langle A\rangle_e)$

$SF \implies WF$

Fair processes

module *BlockingQueuePCal*

```
...

fair process ( producer ∈ p ) {
    put: while ( true ) {
            if ( isFull ) { wait() ; }
            else { notify() ; store := Append(store, self) ; } ;
         } ;
}

fair process ( consumer ∈ c ) {
    take: while ( true ) {
            if ( isEmpty ) { wait() ; }
            else { notify() ; store := Tail(store) ; } ;
         } ;
}

...
```

All consumers consume & all producers produce

module *BlockingQueuePCal*

...

$$Prop \triangleq \begin{array}{l} \land \forall con \in c : \Box\Diamond(\langle take(con) \land \neg isEmpty\rangle_{vars}) \\ \land \forall pro \in p : \Box\Diamond(\langle put(pro) \land \neg isFull\rangle_{vars}) \end{array}$$

Starvation free

```
──────────────── module BlockingQueuePCal ────────────────
variable store = ⟨⟩ ; k ∈ K ; waitP = ⟨⟩ ; waitC = ⟨⟩ ;

  ...

  macro enqueue(waitset, proc){
        if (proc ∉ SeqToSet(waitset)){
              waitset := Append(waitset, proc);
        };
  }

  fair process (producer ∈ P){
        penq: enqueue(waitP, self);
        pw:    await Head(waitP) = self ;
        put:   if (¬isFull){
                    waitP := Tail(waitP);
                    store  := Append(store, self);
                 };
                 goto penq;
  }

  fair process (consumer ∈ C){
        cenq: enqueue(waitC, self);
        cw:    await Head(waitC) = self ;
        take: if (¬isEmpty){
                    waitC := Tail(waitC);
                    store  := Tail(store)  ;
                 };
                 goto cenq;
  }
──────────────────────────────────────────────────────────
```

What would Doug Lea do?

- java.util.concurrent.ArrayBlockingQueue
  - Two j.u.c.locks.Condition: *notEmpty* and *notFull*
  - Fair j.u.c.l.ReentrantLock:
    - Queue of waiting threads

Collect!

Win Win Win

Reasons to dislike TLA$^+$

- Learning curve
  - Bizarre syntax:
    - pc' = [pc EXCEPT ![self] = "lbl"]
  - Basic pattern repository & standard modules
- Does the implementation correctly implement the specification?
  - Early code generation approaches such as PGo exist [Beschastnikh, 2018]
  - Check code directly with e.g. Java Path Finder [Havelund and Pressburger, 2000]
- "All models are wrong, some are useful" (George Box)
- TLC models have to be finite and ...

TLC & State Space Explosion

Problem of (explicit state) model checking:

Linear increase in size of specification or properties can lead up to *exponential* growth of state space

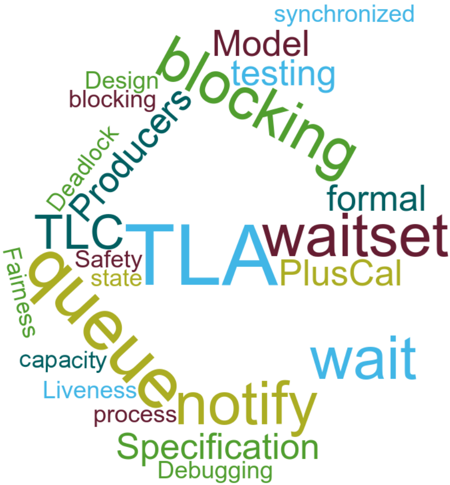
Theorem Proving with TLAPS

It's easier to prove something if it's true

Conclusion

- TLA$^+$ no silver bullet
- TLA$^+$ use of simple math hides idiosyncrasies of programming languages
  - => Focus on the actual problem
  - ...from the design to the implementation phase
- TLA$^+$ scales from simple to complex problems
- Lamport [2017] video course best introduction to TLA$^+$

Q&A


synchronized
Model
blocking
testing
Design
blocking
Producers
Deadlock
formal
TLC
TLA
waitset
Safety
state
PlusCal
Fairness
queue
wait
capacity
Liveness
notify
process
Specification
Debugging

Contact

- slides: `https://bitbucket.org/lemmster/blockingqueue`
- github: `https://github.com/tlaplus/tlaplus`

Bibliography I

Ivan Beschastnikh. PGo is a source to source compiler to compile PlusCal into Go lang, 2018. URL https://github.com/UBC-NSS/pgo. (Accessed 2018-02-19).

Michel Charpentier. An Example of Debugging Java with a Model Checker, 2017. URL http://www.cs.unh.edu/~charpov/teaching-cs745_845-example.html. (Accessed 2018-02-19).

Klaus Havelund and Thomas Pressburger. Model checking JAVA programs using JAVA PathFinder. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4):366–381, March 2000. ISSN 1433-2779, 1433-2787. doi: 10.1007/s100090050043. URL http://link.springer.com/10.1007/s100090050043.

Bibliography II

Rajeev Joshi, Leslie Lamport, John Matthews, Serdar Tasiran, Mark Tuttle, and Yuan Yu. Checking Cache-Coherence Protocols with TLA+. *Formal Methods in System Design*, 22(2):125–131, March 2003. ISSN 0925-9856, 1572-8102. doi: 10.1023/A:1022969405325. URL http://link.springer.com/10.1023/A:1022969405325.

L. Lamport. Proving the Correctness of Multiprocess Programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, March 1977. ISSN 0098-5589. doi: 10.1109/TSE.1977.229904. URL http://ieeexplore.ieee.org/document/1702415/.

Leslie Lamport. The TLA+ Video Course, 2017. URL http://lamport.azurewebsites.net/video/videos.html. (Accessed 2017-04-27).

Chris Newcombe. Debugging Designs using exhaustively testable pseudo-code, 2011. URL http://hpts.ws/papers/2011/sessions_2011/Debugging.pdf.

Bibliography III

Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How Amazon Web Services Uses Formal Methods. *Communications of the ACM*, 58(4): 66–73, March 2015. ISSN 00010782. doi: 10.1145/2699417. URL http://dl.acm.org/citation.cfm?doid=2749359.2699417.

Diego Ongardie. Raft consensus algorithm, 2016. URL https://github.com/ongardie/raft.tla. (Accessed 2018-02-19).