**TLA+ allows you to build models and check them.**

It can be used for low-level concerns (data-structures, algorithms) as well as high-level concerns (real world systems, moving parts, etc.)

We will hopefully do a little of both.

**Bottom Line Up-front**

* I like it
* I’ll probably use it more in the future
* It has some downsides

**What is TLA+ all about?**

Modelling (simplifying) systems

Exposing weaknesses in your designs

Especially useful for concurrent & distributed systems

**PlusCal**

TLA+ also comes with an extra language PlusCal. It feels more imperative to write, so it can be helpful if you want your model to match your code more. It compiles down to TLA+.

We’ll take a quick at it, time depending.

This came up in my hacker news feed today, so I couldn’t resist adding it:

[What is extreme programming and how it makes your code better?](https://medium.com/@omkar_80824/what-is-extreme-programming-and-how-it-makes-your-code-better-49812edafa21)

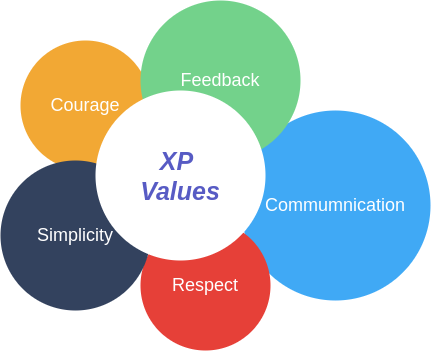
It begins with:

*In early programming days, software projects ran only on the best of the abilities of the people who wrote it. One small mistake and the code wreaks havoc.*

It ends with:

*A bug caught any time before shipping no matter how late in the development process will always cost a million times less than when caught in production.*

What’s in the middle of the article?



It’s a lot longer than that, but it doesn’t explain how to avoid havok or to catch bugs before production (other than “automated tests”).

I’ve never *courage*d a race-condition or *respect*ed a deadlock, but let’s see how the XP folks went...

There is a wiki thread which documents Tom Cargill uploading some buggy code, and asking test-drive folks to find the bug. Two of the founders of Extreme Programming (Don Wells, Ron Jeffries) as well as some others step up to the challenge.

I’ve heavily edited and **paraphrased** it.

<https://wiki.c2.com/?ExtremeProgrammingChallengeFourteen>

<https://wiki.c2.com/?ExtremeProgrammingChallengeFourteenTheBug>

<https://wiki.c2.com/?ExtremeProgrammingChallengeFourteenSplit>

<https://wiki.c2.com/?JavaUnitTestChallengeSolved>

*Tom:* Concurrent programs are hard to test, because you can’t control the scheduler.

*Don:* The testing framework needs to be able to simulate the scheduler in a deterministic way.

*Tom:* Testing frameworks are usually written in the same language as the code, so they lack the power to control their own schedulers.

*Don:* Then design your code for testability. If you can’t control context switches, don’t allow them. Just start writing tests. You’ll be surprised how useful just 20 tests can be.

*Tom:*

public class BoundedBuffer {

private final Object[] buffer = new Object[4];

private int putAt = 0;

private int takeAt = 0;

private int occupied = 0;

synchronized void put(final Object x) throws InterruptedException {

while (occupied == buffer.length) {

wait();

}

notify();

++occupied;

putAt %= buffer.length;

buffer[putAt++] = x;

}

synchronized Object take() throws InterruptedException {

while (occupied == 0) {

wait();

}

notify();

--occupied;

takeAt %= buffer.length;

return buffer[takeAt++];

}

}

Find the bug. Write a test that exposes the bug.

[...Further discussion…]

*Ron:* I haven’t found the bug yet. What is it?

*Tom:* I think we’ll learn more if I don’t say yet.

[...Further discussion…]

*Ron:* I still don't see the defect, and am out of things to test. I've convinced myself that the buffer can't over- or underflow or answer the same element twice. I could be wrong. For sure, I'm stuck.

*Don:* I wish I had more time. Maybe later. There’s obviously a lot of sequential bugs. And we should fix them by doing [...]

*Ron:* I’m not following you here, Don.

*Don:* Correct as usual Ron. Now that the unit tests have been written. I see that there are no bugs. Sometimes unit tests surprise you by telling you that your code actually does work.

*Tom:* Please show me the “obvious sequential errors”. I only put in concurrency errors.

*Ron:* I’m still stuck. By the way, code review isn’t turning up the error either. What does that suggest?

[...Further discussion…]

*Dan:* I agree that concurrent programs are incredibly hard to debug, fortunately, the one posted above is quite simple to fix, even without a test case. Simply change all of your notify() calls to notifyAll().

*Tom Cargill:*

1. The buffer is initially empty, occupied==0.
2. A consumer thread, C1, arrives and waits.
3. A second consumer, C2, arrives and waits.
4. A producer, P1, arrives.
5. P1 fills the buffer, setting occupied=1.
6. P1's notify() notifies C1, say.
7. Before C1 reacquires the lock, a second producer, P2, arrives.
8. P2 acquires the lock ahead of C1, and waits.
9. Now, C2 and P2 are waiting, and C1 is trying to reacquire the lock.
10. C1 reacquires the lock.
11. C1 empties the buffer, setting occupied=0.
12. C1's notify notifies C2 !
13. C2 reacquires the lock.
14. C2 waits again.

Now, P2 is waiting, even though the buffer is not full!

The challenge was to **create a test that exposes the bug**.

Before the error was explained, the responses were:

* Write tests
* There’s no error
* I wrote tests showing there’s no error

After the error was explained, the responses were:

* Just change notify() to notifyAll(), you don’t even need a test
* Don’t use notify(), only really smart people understand how to code with it
* Why was everyone so hung up about the non-determinism?

Even after having the bug explained to them, well-known respectable engineers struggled to reproduce it.

Maybe I’ve built XP+testing up as some kind of straw-man by suggesting it should help you produce correct code when it doesn’t. Maybe it’s just about customer management, not correct code.

If that’s the case, and you shouldn’t expect these things of XP+testing, what should you turn to? What steps can you take so that the testers can say “that would have been easy to test” afterwards?

Obviously type systemsimmutabilityHaskell, which is great for single-machine concurrency, but all programming languages still have weaknesses if you’re trying to prove properties across multiple machines.

Let's try something that's not a programming language - TLA+

***basic.tla:***

---- MODULE basic ----

VARIABLES x

Init ==

x = 0

Next ==

x' = x

====

***basic.cfg:***

INIT Init

NEXT Next

When you run the model checker (TLC) it will explore all possible states reachable from the starting conditions.

TLC explores the state space by finding TRUE states.

The following will reach 1 state (the starting state):

Next ==

x’ = x

The following will reach infinitely many states:

Next ==

x’ = x + 1

**Status page explanation**

**Logical operators**

In TLA+ you connect things together using logical operators, meaning you’ll use:

/\ - and

\/ - or

~ - not

Easy to remember because:

/\ looks like the ‘A’ in ‘And’

\/ looks like the ‘V’ in ‘Vel’ (which is Latin for inclusive ‘or’)

There are also predicate logic operators, e.g.

\A x \in xs: isPrime(x)

“for all *x* in *xs*, *x* is prime”

\E x \in xs: isNegative(x)

“there exists some *x* in *xs* which is negative”

**Combining expressions**

We can make the following reach only 10 states by combining two expressions in Next:

Next ==

(x’ = x + 1) /\ (x’ < 10)

This is tricky to read, so it’s more usual to list expressions like so:

Next ==

/\ x’ = x + 1

/\ x’ < 10

*Note: This is the first time we’ve added + or compared < integers, so we’ll need to import the Integer package using EXTENDS.*

**Try it out:**

-------- MODULE hello --------

EXTENDS Integers

VARIABLES x

Init ==

x = 0

Next ==

/\ x' = x + 1

/\ x' < 10

==============================

Run it and you should get:

Success

States

Time Diameter Found Distinct Queue

00:00:00 10 10 10 0

**What is correctness? What is proof?**

It’s subjective and hard to define, and gets philosophical very quickly.

It’s easier to divide-and-conquer *correctness* into *safety* + *liveness*.

(Or sometimes *progress* + *preservation*.)

These are easier to define!

*Safety*: Nothing bad happens

*Liveness*: Something good eventually happens

Safety is easy to model using TLA+ **invariants** (on the next slide).

Liveness is a bit trickier to model. You generally use **properties** (in two slides).

**Invariants**

Let’s add an invariant to say that our system should never experience x=5.

In the *.cfg* file, add:

INVARIANTS

Not5

In the *.tla* file, add:

Not5 ==

x /= 5

Run it, and you should get an Error and a trace of all the states leading up to the bad state.

Invariants are things which must always be true!

**Properties**

Properties let you prove things that are true *sometimes*.

<> cond

Condition *cond* eventually TRUE at least once

<>[] cond

Condition *cond* is TRUE at least once and stays true

[] cond

Condition *cond* is always TRUE. (Just like an invariant)

**Stuttering**

There is a big “elephant in the room”.

When I said “eventually TRUE”, what does that mean? Is it true after 10 seconds? After 15 seconds? After 17.5 seconds? After 18.75 seconds?

Who thinks I’m ‘eventually’ going to raise my hand?

TLA+ will *stutter*, meaning that it will always insert ‘no change’ as a state transition, which can be useful and/or annoying, depending on what you’re trying to prove.

“The system will start working when the power comes on again.” Can you prove the power will come on eventually?

Some network protocols are designed to handle arbitrary delays. ARBITRARY! Not “short” delays!

We’ll probably run into stuttering and have to turn it off at some point.

**That’s the essence of TLA+.**

You tell it:

* How the system should move from state to state
* What should and shouldn’t happen

It will tell you:

* when and how you’re wrong

**Persistent queue demo**

(time depending)

**Split into groups**

**Jugs problem - individual/pair exercise**

**Traffic Lights - Group exercise**

**PlusCal Demo - Dekker’s algorithm**

**River-crossing problem - individual/pair exercise**

**WrapUp**

Good parts:

* Language independent
* Super high-level. Don’t just model a service, model several different services and watch them play together.
* Will often uncover bad states & bad assumptions for you
* Can help you beat “blank page” syndrome.
* Stuttering might help...

Bad parts:

* Language independent
* No type-checking
* Not as “proofy” as I thought it would be - the checker is relatively “brute force”
* Can sometimes feel “implementy” rather than “proofy”.
* Stuttering is a PITA when trying to prove ‘eventually’