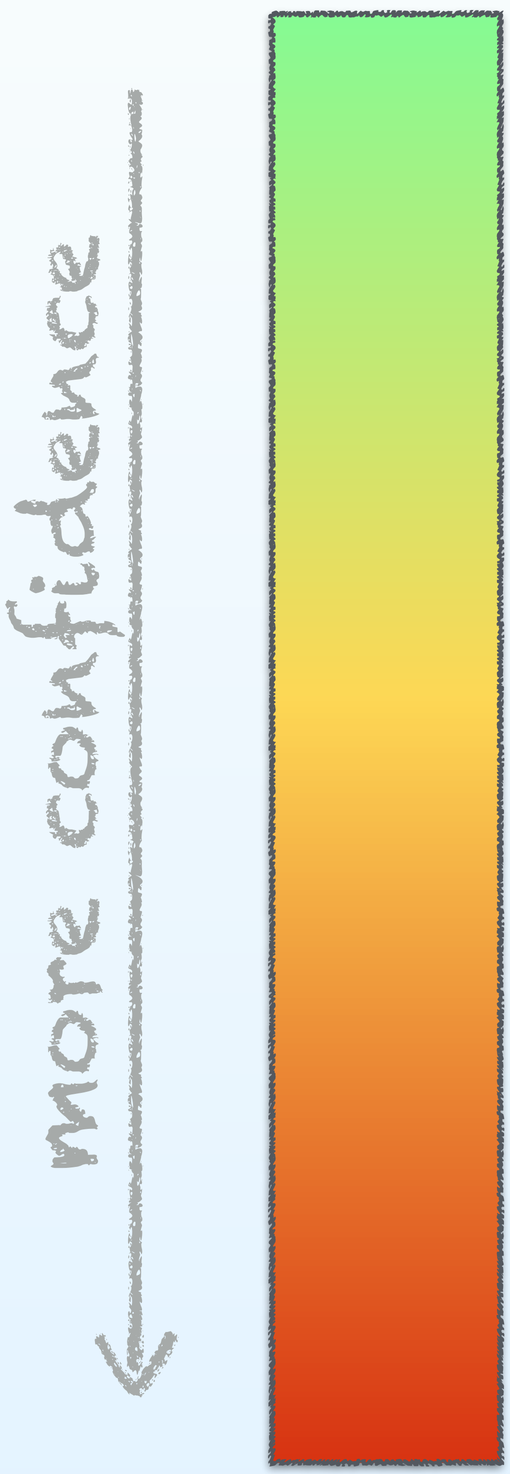
Model Checking for the Working Man (m/f)

@rix0rrr

Build the right software

- Understand the customer
- Iterative development
- Hire good programmers
- …

Build the software right



User Testing

Type Checking

Static analysis/linting

Unit Tests

TDD

Property-based Testing

Programming by Contract

Correctness by construction/Progr. by calculation

Model checking

Programming with dependent types


Stay a while
and listen

Model Checking

Specification + Model

```
x := 0;
y := 1;
while x < N do
    x := x + 1;
    y := y * 2
end
```

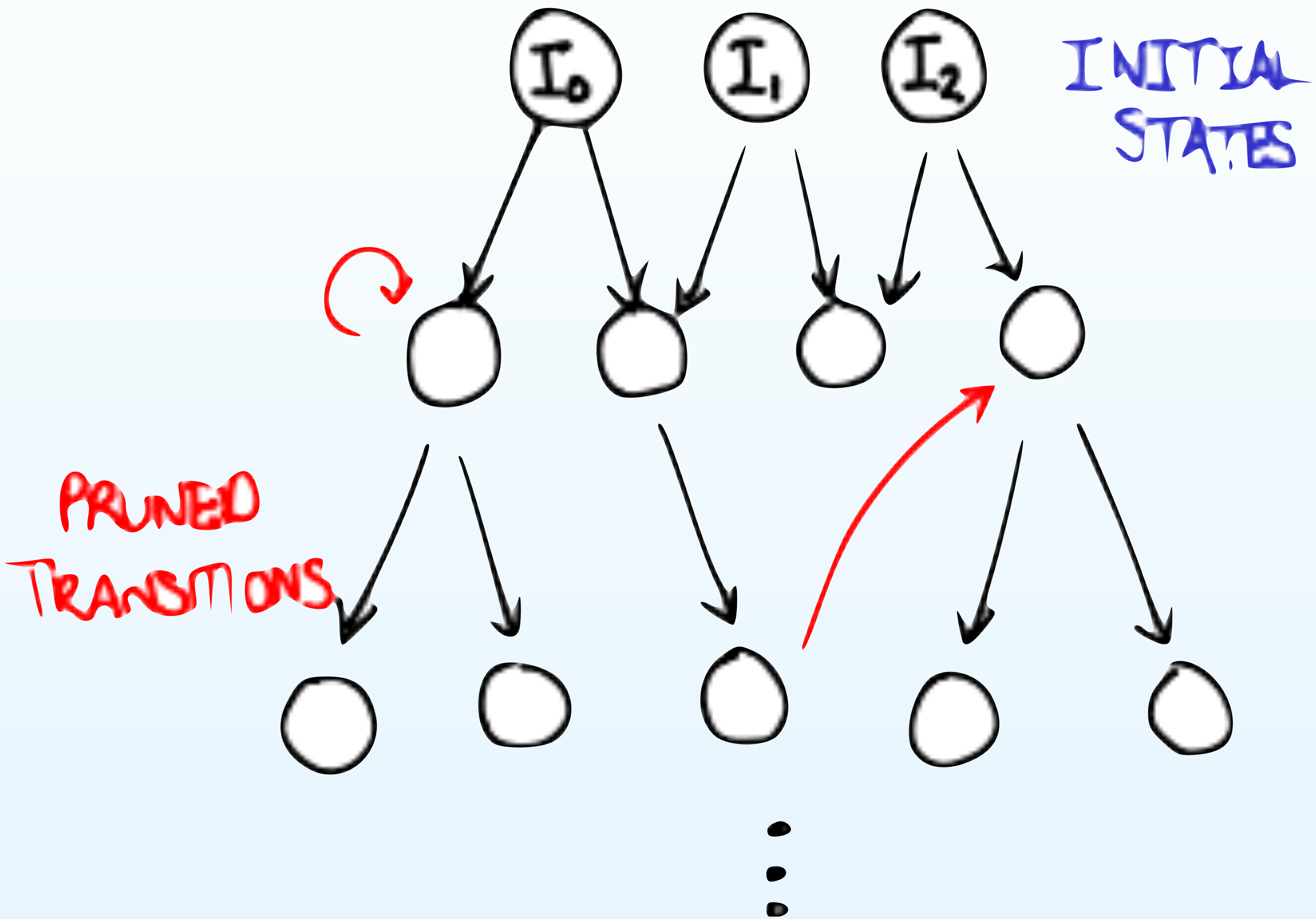
$$N = 10$$

$$\Box\ y = 2^x$$

$$\Diamond\ x = 256$$

Model Checker

OK or **Counterexample**



INITIAL STATES
I_0
I_1
I_2
PRUNED TRANSITIONS

TLA+

formal-methods-amazon-2014-11.pdf (page 1 of 12)

Previous/Next | 1 Page | Back/Forward | Zoom | Tool Mode

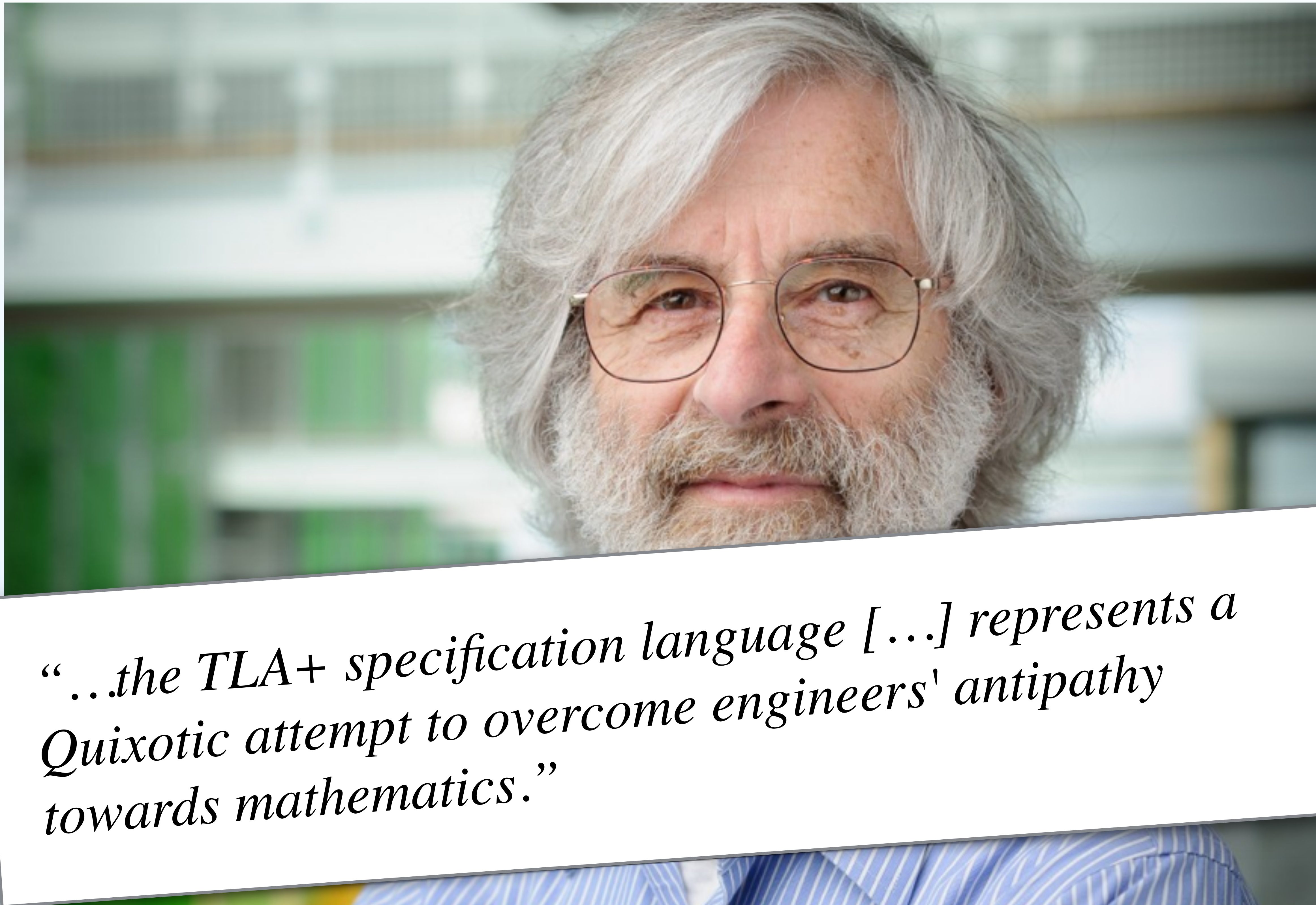
Use of Formal Methods at Amazon Web Services

Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, Michael Deardeuff
Amazon.com

29th September, 2014

Since 2011, engineers at Amazon Web Services (AWS) have been using formal specification and model checking to help solve difficult design problems in critical systems. This paper describes our motivation and experience, what has worked well in our problem domain, and what has not. When discussing personal experiences we refer to authors by their initials.

At AWS we strive to build services that are simple for customers to use. That external simplicity is built on a hidden substrate of complex distributed systems. Such complex internals are required to achieve high availability while running on cost-efficient infrastructure, and also to cope with relentless rapid business growth. As an example of this growth; in 2006 we launched S3, our Simple Storage Service. In the 6 years after launch, S3 grew to store 1 trillion objects [1]. Less than a year later it had grown to 2 trillion objects, and was regularly handling 1.1 million requests per second [2].



"…the TLA+ specification language […] represents a Quixotic attempt to overcome engineers' antipathy towards mathematics."

TLA+ Components

TLA+

Temporal logic language

(for model, maybe spec)

PlusCal

Imperative language

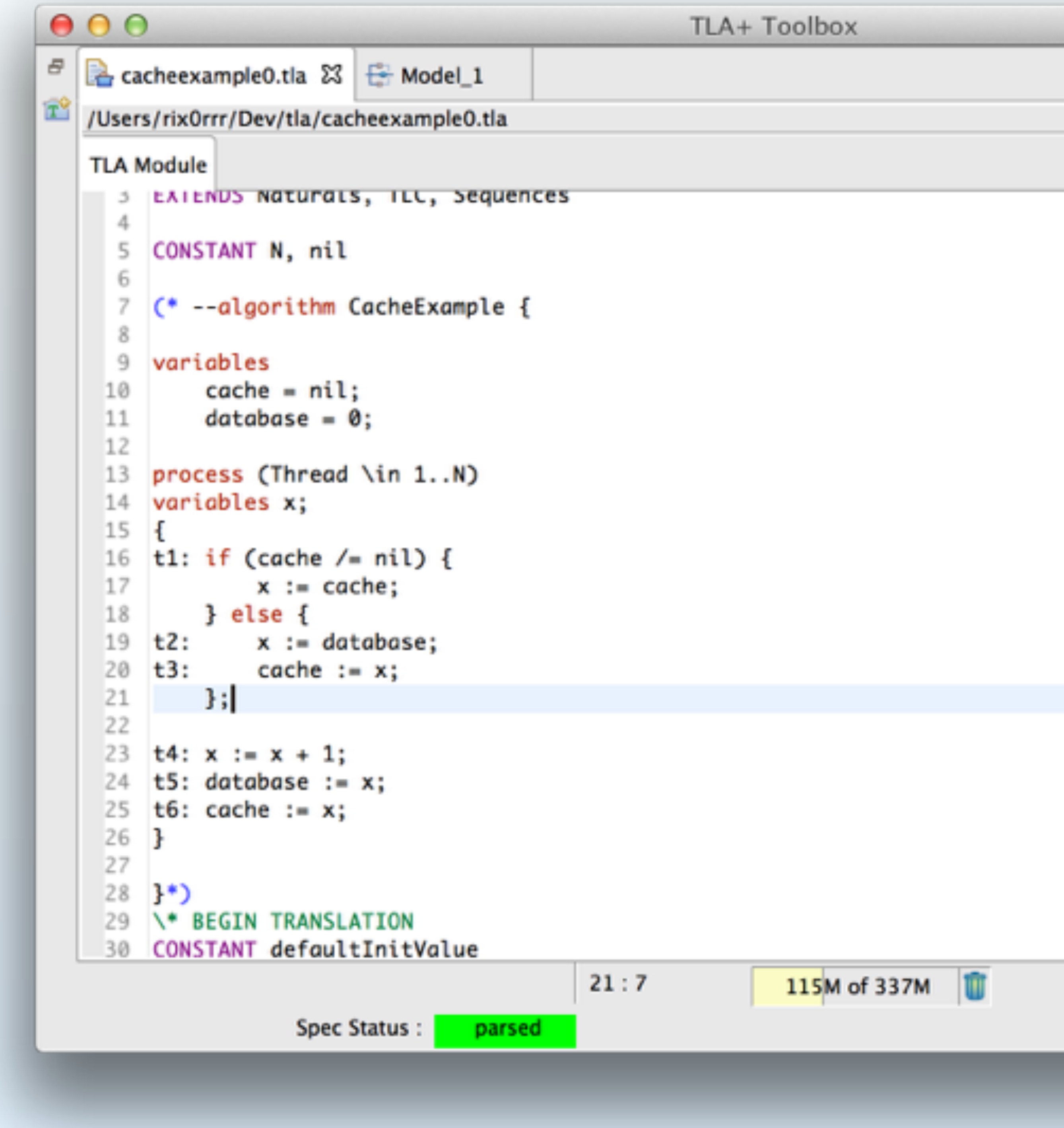
(more natural spec writing)

TLC

Checker tool

Toolbox

Eclipse-based IDE



PlusCal

P-syntax

```
t1:  if (x = 1) then
     begin
         x := x + 1;
         z := y;
     end;
t2:  y := x;
```

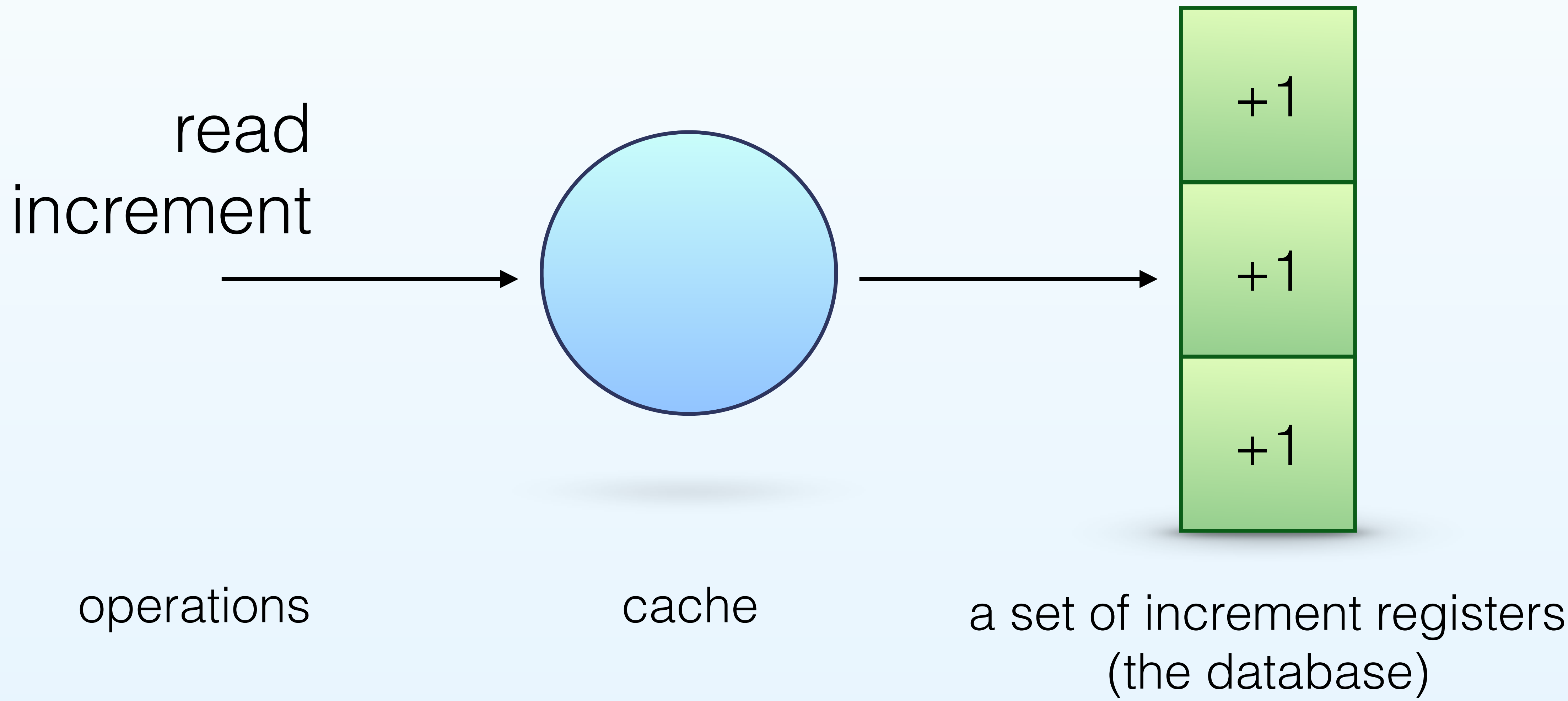
C-syntax

```
t1:  if (x /= 2) {
         x := x + 1;
         z := y;
     };
t2:  y := x;
```

Labels indicate atomic s

All interleavings checked!

The Program


read
increment
+1
+1
+1
operations
cache
a set of increment registers
(the database)

A Python implementation (?)

```python
cache = {}

# Database-backed incrementor
# with cache
def Inc(id):
    x = cache.get(id, None)
    if x is None:
        x = DB_Read(id)
        cache[id] = x

    x = x + 1
    DB_Write(id, x)
    cache[id] = x
```

Translating to PlusCal

```python
cache = {}

# Database-backed incrementor
# with cache
def Inc(id):
    x = cache.get(id, None)
    if x is None:
        x = DB_Read(id)
        cache[id] = x

    x = x + 1
    DB_Write(id, x)
    cache[id] = x
```

Things to consider:

- How to model data? (simplify!)
- What are the atomic steps?
- What concurrency are we modelling?

Try it!

```
cache = {}

# Database-backed incrementor
# with cache
def Inc(id):
    x = cache.get(id, None)
    if x is None:
        x = DB_Read(id)
        cache[id] = x

    x = x + 1
    DB_Write(id, x)
    cache[id] = x
```

```
variables
    cache = nil;
    database = 0;

process (Thread \in 1..N)
variables x;
{
t1: if (cache /= nil) {
        x := cache;
    } else {
t2:     x := database;
t3:     cache := x;
    };

t4: x := x + 1;
t5: database := x;
t6: cache := x;
}
```

TLA+ Translation & the “pc” variable

PlusCal

TLA+

```
Init == (* Global variables *)
        /\ cache = nil
        /\ database = 0
        (* Process Thread *)
        /\ x = [self \in 1..N |-> defaultInitValue]
        /\ pc = [self \in ProcSet |-> "t1"]
```

```
t4: x := x + 1;
```

```
t4(self) == /\ pc[self] = "t4"
            /\ x' = [x EXCEPT ![self] = x[self] + 1]
            /\ pc' = [pc EXCEPT ![self] = "t5"]
            /\ UNCHANGED << cache, database >>
```

The Condition?

"All increments should have been applied"

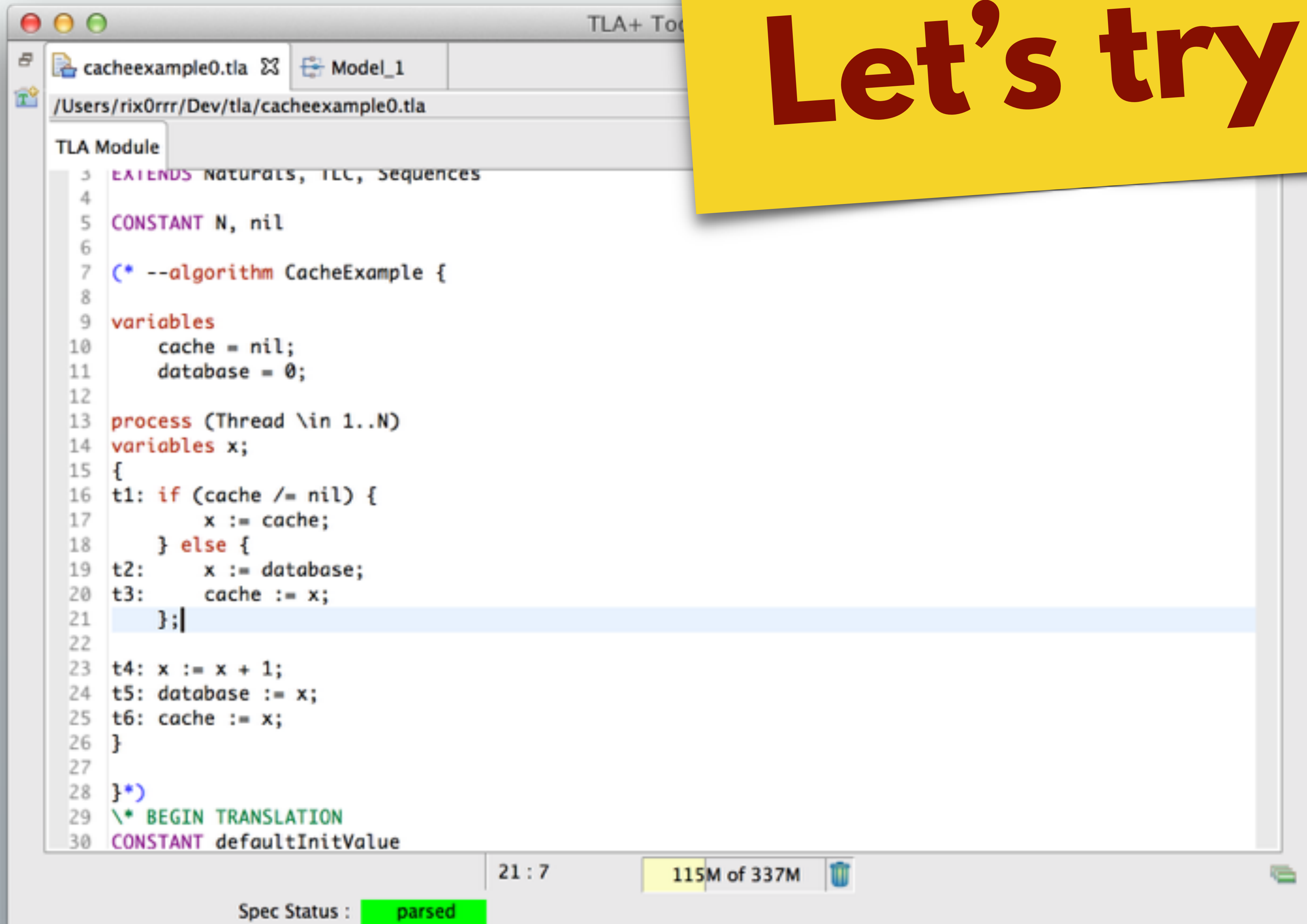
What if I tell you
you can see if a process
has finished by checking
`pc[i] = “Done”` ?

The Condition?

```
Ok == (\i \in 1..N : pc[i] = "Done")
      => database = N
```

$$Ok = (\forall i \in 1..N : pc[i] = \text{“}Done\text{”}) \Rightarrow database = N$$

Let’s try it



TLA+ To
cacheexample0.tla
Model_1
/Users/rix0rrr/Dev/tla/cacheexample0.tla
TLA Module
EXTENDS Naturals, TLC, Sequences
CONSTANT N, nil
(* --algorithm CacheExample {
variables
cache = nil;
database = 0;
process (Thread \in 1..N)
variables x;
{
t1: if (cache /= nil) {
x := cache;
} else {
t2: x := database;
t3: cache := x;
};
t4: x := x + 1;
t5: database := x;
t6: cache := x;
}
}*)
* BEGIN TRANSLATION
CONSTANT defaultInitValue
21 : 7
115M of 337M
Spec Status :
parsed

http://rix0rrr.github.io/pluscal-debugger/

The Problem?

	Process 1	**Process 2**
t=1	Read	
t=2		Read
t=3	Increment	
t=4		Increment
t=5	Write	
t=6		Write

The fix?

Conditional Write

```
def Inc(id):
    x = cache.get(id, None)
    if not x:
        x = DB_Read(id)
        cache[id] = x

    y = x + 1
    if DB_ConditionalWrite(id, x, y):
        cache[id] = y
    else:
        Inc(id)
```

Exercise: implement in PlusCal!

```python
def Inc(id):
    x = cache.get(id, None)
    if not x:
        x = DB_Read(id)
        cache[id] = x

    y = x + 1
    if DB_ConditionalWrite(id, x, y):
        cache[id] = y
    else:
        Inc(id)
```

```
variables
    cache = nil;
    database = 0;

process (Thread \in 1..N)
variables x; y; version;
{
t0: if (cache /= nil) {
        x := cache;
    } else {
t1:     x := database;
t2:     cache := x;
    };

t3: y := x + 1;
t4: if (database = x) {
        database := y;
    } else { goto t0; }; (* Retry *)
t5: cache := y;
}
```


TLA+ Toolbox
cacheexample0.tla
Model_1
/Users/rix0rrr/Dev/tla/cacheexample0.tla
TLA Module
CONSTANT N, nil
(* --algorithm
variables
cache =
process (T
variables
{
t1: if (
t2:
t3:
t4: x := x + 1;
t5: database := x;
t6: cache := x;
}
}*)
* BEGIN TRANSLATION
CONSTANT defaultInitValue
21 : 7
115M of 337M
Spec Status :
parsed
Let's try it!!

Processes instead of threads?

How would we model that?

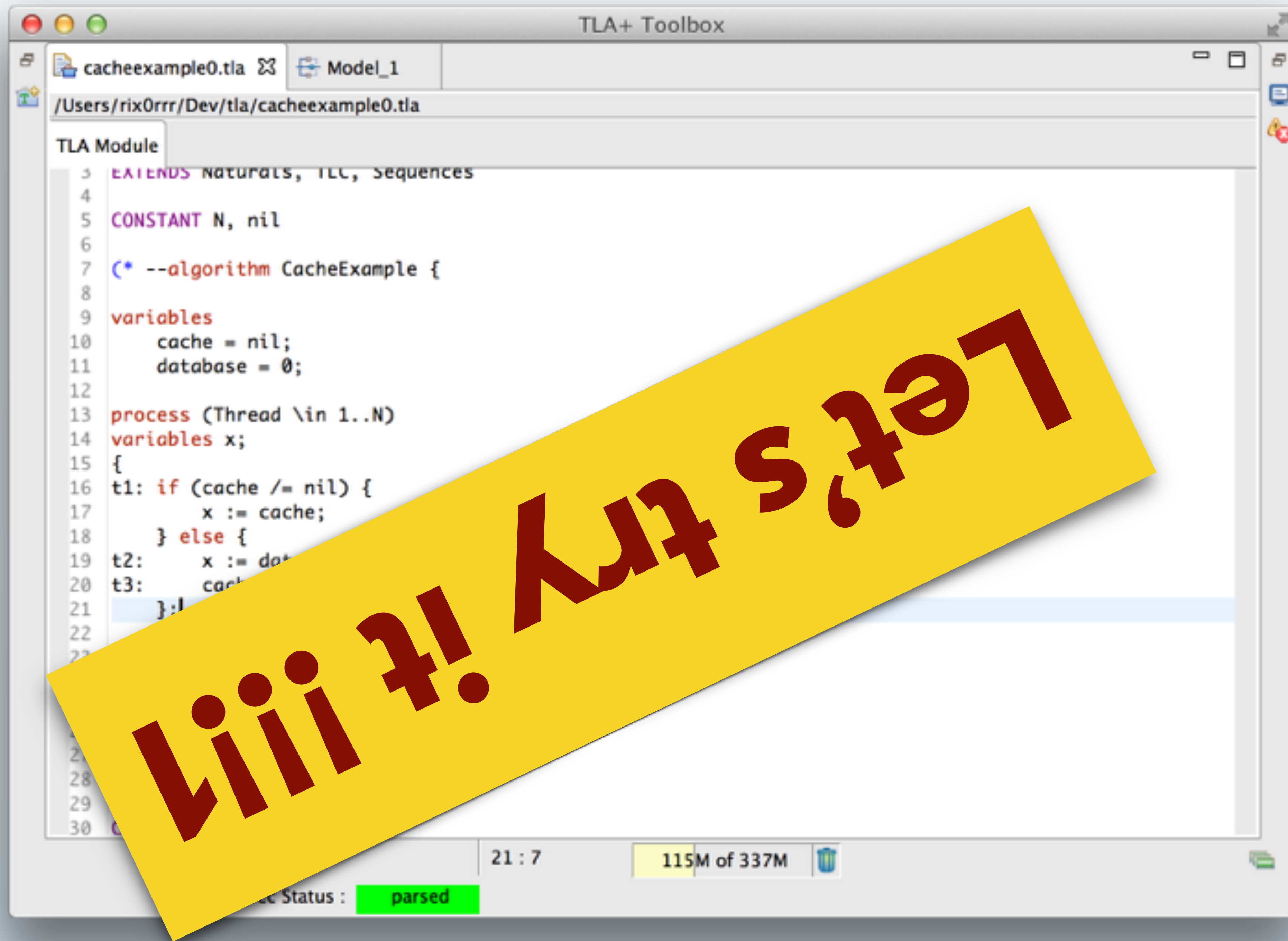
```
variables
    database = 0;
    cache = nil;

fair process (Thread \in 1..N)
variables x; y; version; cache = nil;
{
t0: if (cache /= nil) {
        x := cache;
    } else {
t1:     x := database;
t2:     cache := x;
    };

t3: y := x + 1;
t4: if (database = x) {
        database := y;
    } else { goto t0; }; (* Retry *)
t5: cache := y;
}
```

Termination

$$Termination \equiv \Diamond(\forall i \in N : pc[i] = \text{``}Done\text{''})$$

"at some point in time…"

"all process will be Done"


TLA+ Toolbox
cacheexample0.tla
Model_1
/Users/rix0rrr/Dev/tla/cacheexample0.tla
TLA Module
CONSTANT N, nil
(* --algorithm CacheExample {
variables
cache = nil;
database = 0;
process (Thread \in 1..N)
variables x;
{
t1: if (cache /= nil) {
x := cache;
} else {
t2:
t3:
21 : 7
115M of 337M
Status :
parsed
Let's try it !!!

General Thoughts

Writing the spec

- Where the art is
- Minimize state to keep checking feasible. Skip unnecessary detail.
  - For a general function that can write to a number of registers, only model *one*
  - Don't model actual data *values*, model the *instance* of the data
  - Etc.

Writing the condition

- Write *invariants*
- Write *assertions* inside the code

```
(* Verify that IF the cache isn't hopelessly out of date,
   THEN the cache is consistent with the database *)
macro VERIFY_CacheConsistent(t_version) {
    assert cache_tx_version <= t_version =>
       \A i \in objs: cache_obj_version[i] \in { 0, db_obj_version[i] };
}
```

- Maybe *check for termination*

Common objection

"What if I my code doesn't match my model?"

Thinko-free algorithms

for

very little effort!