

Model Checking for Industrial Programmers with TLA+

Draft v0.1

Gregory S. Miller
New York, NY
gshanemiller6@gmail.com

ABSTRACT

Systems software development comes with problems in number and complexity that are not easily solved through requirements or testing alone. Beyond organizational challenges coordinating human resources to good effect, software is beset with a large number of cross-interacting states across multiple threads of control. With an ever increasing emphasis on cloud-scale distributed systems, ensuring individual states compose well leading to correct system behavior is non-trivial. Model checking is a middle-way solution to these issues among a portfolio of choices from theorem provers to conventional iteration through best-effort testing at the other extreme.

Model checking comes with three salient features. It is constructive. Given a model coded in a simple language, the model verifier will report counter examples to any correctness claims. Second, verification explicitly constructs and analyses every reachable state over all possible execution orders. Unlike testing it is always complete. Third, model checking gives equal treatment to system states. While the details of a client database connection are always important, a database's ACID properties also depend on replication, leader election, quorums, and write-ahead logging. A model will formally describe how and when these distributed states are considered valid when taken together.

Now while most software developers are vaguely aware model checkers address many of these issues, it has yet remained alluringly locked away in niche safety critical sectors or academia. This paper throws back the curtain to squarely address model checking through Microsoft/Lamport's TLA (Temporal Logic of Actions) system. TLA is not the only choice in this space, but commands some attention ranking in the top-five seeing use at Amazon, Google where distributed software substantially underwrites company success.

This paper is self-contained. No background is assumed, and it's not developed herein through first-principles. The approach here is practical on balance. Engineering competence is 51% right-doing, and 49% right-knowing. Doing each well is permanently caught-up in other, however, an extra nod to operational proficiency is the constant aim. To accomplish these goals conceptual background is first developed. TLA/PlusCal language references are provided. Then general issues of correctness are explained. The remainder of the paper gives an in-depth discussion of an elevator model, and a reliable packet protocol for messaging. Extensive examples are provided throughout linked to Github for easy reference.

CONTENTS

Abstract	1	11.4.3	Summary	22
Contents	2	11.5	Other Errors	22
1 Motivation	3	11.6	Complex Backtraces	22
2 Organization	3	11.7	Search	23
3 Introduction	4	11.8	Verifying Complex Models	23
4 Install	5	12	Elevator Model	24
4.1 TLA (Optional)	5	13	RPC Model	24
4.2 Example Models	5	A	Appendix: TLA Language Reference	24
5 Operational Flow	5	A.1	Model Layout	24
6 System State	5	A.2	Names	24
6.1 State Graph	6	A.3	Comments	24
6.2 Discrete Transition	7	A.4	Types & Common Operators	24
6.3 Verification	7	A.4.1	Boolean	24
6.4 Non-Determinism	7	A.4.2	Strings	24
6.5 Process Speed	8	A.4.3	Integers	25
6.6 Summary	8	A.4.4	Sequences/Tuples	25
7 Executability & Fairness	8	A.4.5	Sets	25
7.1 Introduction	9	A.4.6	Structures/Records	25
7.2 Absolute Fairness	9	A.5	Extends	25
7.3 Weak Fairness	9	A.6	Constants, Assume, Variables	25
7.4 Strong Fairness	11	A.7	Configuration Layout	26
7.5 Summary	11	A.8	\in Operator	27
8 LTL (Linear Temporal Logic)	12	A.9	\E, \A Operators	27
8.1 Examples	12	A.10	[], <> Temporal Operators	27
9 Where Are We At? Going?	13	A.11	=> Implies Operator	27
10 TLA Specification	13	A.12	Definition	27
10.1 The Goal	13	A.13	Flow Control: /\, \V	28
10.2 Getting Started	13	A.13.1	Truth Tables	28
10.3 Correctness Condition	13	A.13.2	Encore	29
10.4 Processes	14	A.13.3	\/, \V in Statements	29
10.5 State Definitions	14	A.13.4	Don't Conflate Executability	29
10.6 Spec	14	A.13.5	IF/CASE: What's the Point?	30
10.7 Fixing Config	15	A.14	RECURSIVE	30
10.8 Testing Model	15	A.15	IF-THEN-ELSE	30
10.9 Specifying Multi-Threaded Models	15	A.16	CASE	30
10.10 Discussion	15	A.17	LET-IN	31
11 Correctness Mechanics	16	A.18	CHOOSE	31
11.1 Assert	16	A.19	ENABLED	31
11.1.1 Part I	16	A.20	Assert	31
11.1.2 Part II	17	A.21	Print	31
11.1.3 Part II: Verify All	17	A.22	Vars Define	31
11.1.4 Part II: Big Picture	17	A.23	UNCHANGED, EXCEPT	31
11.1.5 Part II: Verification Stats	18	A.24	SF, WF	32
11.1.6 Part II: Analyze Backtrace	18	A.25	Assignment v. Equality	32
11.2 Invariants	18	A.26	REPL	32
11.2.1 Example	19	A.27	Formatting & Alignment	32
11.2.2 Summary	19	A.28	References	32
11.3 Deadlock & Termination	19	A.29	Tables	33
11.3.1 TLA Part I	19	B	Appendix: PlusCal Language Reference	34
11.3.2 TLA Part II	20	B.1	Operational Flow	34
11.3.3 PlusCal Part I	20	B.2	Names	34
11.3.4 PlusCal Part II	20	B.3	Model Layout	34
11.4 Properties	21	B.4	Global Variables	35
11.4.1 Always Example	21	B.5	Definitions	35
11.4.2 Eventually Example	21	B.6	Macro	35
		B.7	Procedure	35
		B.8	Process	36

B.9	Labels	36
B.10	Fairness Syntax	36
B.11	Assignment v. Equality	37
B.12	Operator	37
B.13	Assert	37
B.14	Print	37
B.15	Skip	37
B.16	If-Elsif-Else	37
B.17	While	38
B.18	Either	38
B.19	Await	38
B.20	With	38
B.21	Goto	38
B.22	Call	38
B.23	Return	38
B.24	Deferred Typing	38
B.25	Name Mangling and Self	39
B.25.1	Naming Rules	39
B.25.2	Label Naming	39
B.25.3	Self	40
B.25.4	Summary	41
B.26	References	41
	References	41

1 MOTIVATION

It's interesting to watch organizations handle risk. Consider this fictitious story.

Cupertino 1985. On one corner of Main street, the big boss at *Acme Corp* calls an all-hands meeting.

"There's good news, and bad news. The good news is we've landed a major client in a lucrative deal. The bad news is they want our product line, but with numerous additional use-cases that'll change aspects of our architecture. Our sales chief, Bill, knows the use cases and is prepared to help coordinate with development. As with any major effort we are aware the customer will change their mind, refine, or de-emphasize requirements as the work comes into focus. It'll be a mess here and there. We'll need to be agile working closely with business, but I'm confident we can tech this out."

Meanwhile, the CEO at *Tech Corp* calls his own meeting across the street:

"I've got good news, and bad news. The good news is we've landed a big contract. The bad news is the client wants to run our core technology over multi-hosts with fail-over into a second data center. We've identified three viewpoints on system behavior for which formal methods will be used to ensure we've got our act together..." But before the boss can finish his sentence the complaints start rolling in from staff.

"Nobody knows formal methods. And isn't there a risk we'll model the wrong thing? Isn't it a waste of time? It's over engineering. And what about implementation? Just because the team lead shows up in a meeting with a model, how are we supposed to implement it? We could make mistakes, and that can drag on. Formal methods? Are you sure that's smart?"

Indeed formal methods can make software developers more anxious not less in the early stages. Model specification is precise. When combined with systematic state checking the engineering reach is considerable. But that means details, and details are hard to nail down. Unconsciously it's easier to tactically defer arguing perfection is the enemy of good. After all if the system design or requirements are sort-of fuzzy now, then it's OK to write sort-of fuzzy code for now. And the best way to do that is to ask for more JIRA tickets. So long as there's activity putatively trending to a finish line, convergence is practically guaranteed and, it's case-closed, right?

In truth neither company has an uncontested straight line to the net; there are elements of truth on all sides. The goal here is the alleviation of anxiety through choice rooted in knowledge. A model is valuable artifact stating what is correct matched to an explicit algorithm for doing the work. Although models can certainly fly close to implementation detail, they tend to elide selected details working at higher levels of abstraction. Regardless, a verified model always lowers risk. Compared to writing code it's a cheap way to cut through the fog of edge-cases.

Like a general purpose programming language, model checking is amenable for use in many software problems. But it's natural habitat is CSP (Concurrent Sequential Processes). Each thread or module runs vanilla sequential imperative commands with assignment, basic branching, or loops. On the other hand each thread's work is incomplete. It must compose with work held on other machines. Some examples include:

- Reliable packet protocols
- Leader election
- Strong serializability
- State replication
- Databases
- Concurrent data-structures *e.g.* B-tree, key-value stores

Implicit in much of this work is multi-threads either for scaling or reliability reasons, which must cooperate to deliver a singular behavior like *commits are strongly serializable* at some system synchronization point.

2 ORGANIZATION

The introduction §3 motivates model checking within a portfolio of other choices giving its merits, and demerits. Install directions §4 come next. Section §5 reviews what TLA provides, and how it runs at high level.

The primary material starts in §6 *System State*. It provides a conceptual background on what model checking is, and how it substantively differs from imperative sequential programming. Section §7 *Fairness & Executability* considers state from the verifier's perspective when deciding what code can be run. Section §8 introduces *LTL* (Linear Temporal Logic) concepts. Having no equal in industrial programming, it is crucial to validating system states. To complete the theory portion of this paper, *Where Are We At? Going?* §9 summarizes.

A complete TLA language reference guide is provided in Appendix §A. PlusCal is provided in Appendix §B. Both treatments are self-contained. Readers should thoroughly read §A, and have passing familiarity with §B before continuing.