# Metadata

on distributed systems broadly defined.

Wednesday, July 4, 2018

## If You’re Not Writing a Program, Don't Use a Programming Language

This article by Leslie Lamport has appeared recently at the Distributed Computing & Education Column.

This article focuses more on the distinction between the TLA+ modeling language versus programming languages. Earlier I had written a blog post about more general benefits of modeling and model-checking.

This is not a technical article so instead of trying to summarize/review the main points, I just include some choice quotes from the article below. The most important takeout message is at the end, which I emphasize with the bold font.

*Algorithms are not programs, and they can be expressed in a simpler and more expressive language. That language is the one used by almost every branch of science and engineering to precisely describe and reason about the objects they study: the language of mathematics.*

*Programming is too often taken to mean coding, and the algorithm is almost always developed along with the code. To understand why this is bad, imagine trying to discover Euclid's algorithm by thinking in terms of code rather than in terms of mathematics.*

> *The [TLA+] abstraction helped a lot in coming to a much cleaner architecture (we witnessed first-hand the brainwashing done by years of C programming). One of the results was that the code size is about 10× less than in [the previous version].*
>
> *(Eric Verhulst, Raymond T. Boute, José Miguel Sampaio Faria, Bernard H. C. Sputh, and Vitaliy Mezhuyev. Formal Development of a Network-Centric RTOS. Springer, New York, 2011.)*

*The common obsession with languages might lead readers to think this result was due to some magical features of TLA+. It wasn’t. It was due to TLA+ letting users think mathematically.*

*A correctness property of an algorithm is often best expressed as a higher-level algorithm. Proving correctness then means proving that the original algorithm refines the higher-level one. This usually involves both data refinement and step refinement.*

*I expect that this kind of refinement sounds like magic to most readers, who won't believe that it can work in practice. I will simply report that among the refinement proofs I have written is a machine-checked correctness proof of the consensus algorithm at the heart of a subtle fault-tolerant distributed algorithm by Castro and Liskov that uses 3F + 1 processes, up to F of which may be malicious (Byzantine). The proof shows that the Castro-Liskov consensus algorithm refines a version of the 2F + 1 process Paxos consensus algorithm that tolerates F benignly faulty processes. Steps of malicious processes, as well as many steps taken by the good processes to prevent malicious ones from causing an incorrect execution of Paxos, refine stuttering steps of the Paxos algorithm. I found that viewing the Castro-Liskov algorithm as a refinement of Paxos was the best way to understand it.*

*Today, programming is generally equated with coding. It's hard to convince students who want to write code that they should learn to think mathematically, above the code level,*

**About Me**

My photo **Murat**

I am a computer science and engineering professor at SUNY Buffalo. (Currently on sabbatical at Microsoft Azure Cosmos DB.) I work on distributed systems, distributed consensus, and cloud computing. You can follow me on Twitter.

View my complete profile

*about what they're doing. Perhaps the following observation will give them pause.* ***It's quite likely that during their lifetime, machine learning will completely change the nature of programming. The programming languages they are now using will seem as quaint as Cobol, and the coding skills they are learning will be of little use. But mathematics will remain the queen of science, and the ability to think mathematically will always be useful.***

**Related posts on TLA+**

Hillel maintains a nice tutorial for TLA+. He has an upcoming book on TLA+.

I have written up a lot on TLA+. Here are two gentle introduction posts I had written in 2014 and 2015. I gave many examples of modeling with TLA+, as I think lack of examples is one of the biggest obstacles before TLA+ achieving wider adoption. Some include modeling of 2-phase commit transactions, synchronized round consensus, chain replication, hygenic dining philosophers, and Paxos and Flexible Paxos. I also use TLA+ in my research papers to provide an unambiguous pseudocode for algorithms, as well as a way to model check their correctness.

This summer, I will be joining Microsoft Azure Cosmos DB team for my sabbatical for 6 months. I will get a chance to use TLA+ in practice in the context of a very large-scale globally distributed, multi-model database service. So you can expect many more TLA+ posts in this space soon.

**MAD questions**

1. Instead of the succinctness distinction between a programming language and a specification language, maybe the difference in benefits comes more from the process. You should approach programming in a two phase manner: think before you code. Before you start writing any pretensibly executable code, think first and write a design/algorithm/model first using that programming language as pseudocode.

But then, wouldn't this have worked for the RTOS project cited above? They could have written the design/algorithm in pseudocode C first, and would have achieved the same benefits. But maybe trying to write the design/algorithm in the programming language bring a lot of baggage with it (brainwashing as the quotation above mentions) that this is impossible. Maybe this is the curse of the Whorfian syndrome Lamport wrote about in another article. "Computer scientists collectively suffer from what I call the Whorfian syndrome -- the confusion of language with reality."

2. Would a succinct programming language nullify (or weaken) the arguments in this article? What is your favorite succinct programming language?

Paul Graham swears by the benefits of Lisp. He quotes ESR there: "Lisp is worth learning for the profound enlightenment experience you will have when you finally get it; that experience will make you a better programmer for the rest of your days, even if you never actually use Lisp itself a lot."

Maybe the benefits come from declarative versus operational thinking? So declarative languages would have an advantage over operational languages.

I am not a programming language guy. And I don't want to start a PL frame war, so maybe I should stop. (This may be the most dangerous MAD question I had written :-)

By Murat at July 04, 2018

Labels: programming, tla

## 4 comments:

**Anonymous said...**

That was a good morning reading: thank you!

I just want to reinforce the bias on the supposition suggested in one of your "mad questions":

**Popular Posts**

Facebook's software architecture

Learning Machine Learning: A beginner's journey

Realtime Data Processing at Facebook

Master your tools

My Distributed Systems Seminar's reading list for Spring 2017

Two-phase commit and beyond

A Comparison of Distributed Machine Learning Platforms

Hybrid Logical Clocks

Paper review. IPFS: Content addressed, versioned, P2P file system

Paxos derived

**Labels**

paper-review misc tla mad-questions Blockchain mldl research-advice research-question teaching book-review big-data writing stream-processing tensorflow my advice Cosmos DB CosmosDB facebook paxos fault-tolerance mlbegin trip-report Azure failures my-paper auditability consistency dataflow seminar scheduling stabilization time bestof indexing mobile presenting programming smartphones wpaxos chaos graph-processing sonification

**Search This Blog**

Search

**Pageviews**

**Follow by Email**

Email address... Submit

> Maybe the benefits come from declarative versus operational thinking? So declarative languages would have an advantage over operational languages.

My opinion is that you are right. I notice that I generally solve more difficult tasks in Haskell easier than in imperative or object-oriented languages, and the declarative approach is what makes the difference to me: I can think about the problem mathematically instead of operationally.

And that concept seems to routinely pop-up in people's heads. For example:
- https://lemire.me/blog/2010/02/12/the-best-software-developers-are-great-at-mathematics/
- https://www.drmaciver.com/2014/07/programming-and-mathematics/

July 5, 2018 at 3:51 AM

**Anonymous said...**

"It's hard to convince students who want to write code that they should learn to think mathematically, above the code level, about what they’re doing."

I think "mathematics," is redundant here. What you're talking about is design, I think, in which mathematics (of course) plays a part. But don't take this as criticism, please. It isn't. It's vital that new programmers realise as soon as possible that there is more to it than just coding. They need to think above the code level, as you say. That is the first major step a learning programmer can/will make. Any encouragement in this direction is useful and helpful.

Good article!

Pattern-chaser

"Who cares, wins"

July 5, 2018 at 7:07 AM

**pron said...**

The succinctness Lamport talks about cannot be achieved in any programming language because programming languages have the requirement that they produce efficient code, while the succinctness is achieved by the use of quantifiers, for which generating efficient code is intractable. In other words, the succinctness stems from embracing nondeterminism, which is at odds with efficient compilation. Some programming languages can achieve similar expressivity by incorporating a sub-language -- for contracts or dependent types -- that allows arbitrary use of quantifiers (and nondeterminism), but this makes the programming language a combination of two languages: a specification language and an implementation language. One could argue that this incorporation of the two languages, which allows for a direct formal relation between the two, offers some advantages, but the opposite could be argued as well (that it results in a combination of a complex programming language and a complex specification language, and that the formal connection between the two is hard to utilize well when truly interesting properties are concerned).

July 5, 2018 at 7:08 PM

**zagubiony said...**

Hello Murat,

I have a question regarding TLA+ and its usage. I've heard and seen good things about TLA+ and am thinking about starting learning it but before I invest my time I would like to know the differences between TLA+ and QuickCheck (property based) tests. It seems that they both of them search through possible input space and check model's invariants. Do you have an opinion on that?

August 10, 2018 at 4:02 AM

Post a Comment

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)

**Two-phase commit and beyond**

In this post, we model and explore the two-phase commit protocol using TLA+. The two-phase commit protocol is practical and is used in man...