pressron

TLA$^+$ in Practice and Theory Part 1: The Principles of TLA$^+$

25 May 2017

This is the first installment in a four-part blog post series about TLA$^+$. A new post will be published every Thursday.

Introduction

TLA$^+$ is a formal specification and verification language that helps engineers design, specify, reason about and verify complex, real-life algorithms and software or hardware systems. TLA$^+$ has been successfully used by Intel, Compaq and Microsoft in the design of hardware systems, and has started seeing recent use in large software systems, at Microsoft, Oracle, and most famously at Amazon, where engineers use TLA$^+$ to specify and verify many AWS services.

I will get to explaining what a formal specification and verification language is and how it helps create better, more robust algorithms and digital systems shortly, but first I'd like to clarify the focus of this four-part blog post series on TLA$^+$. I called this series, *TLA$^+$ in Practice and Theory*, but it will be almost all theory, where "theory" means two things — mathematical theory and design theory. While I will cover virtually all of TLA$^+$, this is not a tutorial although it may serve to complement one. I will attempt to clarify some of the concepts that can be hard to grasp when learning TLA$^+$ — which, while very small and simple, does contain some ideas that may take a while to fully understand — but this material is by no means essential to learning TLA$^+$ or for writing good, useful specifications, nor will reading these posts teach you how to write good specifications; so this series is neither necessary nor sufficient for putting TLA$^+$ to good use, but I think it is both necessary and sufficient for *understanding* it.

I hope to make you understand what TLA$^+$ is for, why it is designed in this way, and how its design and mathematical theory compare to other approaches to formal methods. Most importantly, I hope it will convey how powerful a tool is (almost) ordinary mathematics for reasoning about the systems we design and build. This series is addressed to those who are interested in formal methods or programming languages (even though TLA$^+$ is not a programming language at all) and may find TLA$^+$'s theoretical approach and pragmatic design interesting, and to those who already know TLA$^+$ but wish to dig deeper into its theory.

There is no shortage of good beginner tutorials on TLA$^+$. ***The TLA$^+$ Hyperbook*** is a complete, thorough, hands-on tutorial written by Leslie Lamport, TLA$^+$'s inventor, and is probably the best way to get started with TLA$^+$ if you want to start applying it in practice. You should expect to work through the hyperbook and become productive enough to specify and verify real-world systems in about two weeks. ***Specifying Systems***, also by Lamport, is an older book, less hands-on but more in-depth and with a greater emphasis on theory. Both books are freely available for download. Very recently Lamport made an online **TLA$^+$ video course**, that at the time of this writing is still being produced. Finally, **this short tutorial**, with a heavy focus on PlusCal (an alternative, pseudocode-like syntax for TLA$^+$), is for those who are in a rush to start using TLA$^+$ within hours, and who are not interested in theory. Additional complementary learning material is the **Dr. TLA$^+$** series of lectures, which covers various algorithms and their specification in TLA$^+$, and a collection of TLA$^+$ examples that you can find in **this GitHub repository**.

There is no shortage of papers about the theory of TLA$^+$, either. There are nice collections of academic papers on TLA$^+$ by both **Leslie Lamport** and **Stephan Merz**, and there are others. However, those are technical, aimed at researchers and generally assume significant prior knowledge[1]. This series is intended to be a gentler introduction to the theory of TLA$^+$ for curious practitioners (software and hardware developers) as well as academics who are more familiar with other approaches to software specification and verification.

The next three installments in the series will be a deep dive into the language itself and the theory behind it, but the choices made in the design of the language can only be understood in context. This post will provide this necessary context in the form of a general introduction to the motivation, history and design principles of TLA$^+$, as well as a comparison of those with other approaches.

TLA$^+$ in Practice

Any treatment of the theory of TLA$^+$ must begin with an overview of its practice, as the theoretical choices made in the design of TLA$^+$ were motivated, first and foremost, by the necessities of practice.

It was a technical report called ***Use of Formal Methods at Amazon Web Services*** (*Communications of the ACM*, April 2015)[2], detailing the experience of Amazon engineers using TLA$^+$ in the design of AWS that first drew my attention to TLA$^+$[3] as a practical tool for designing and verifying complex software. Here are some excerpts from that report that will serve as an overview for the use of TLA$^+$ in industry:

> *High complexity increases the probability of human error in design, code, and operations. Errors in the core of the system could cause loss or corruption of data, or violate other interface contracts on which our customers depend. So, before launching such a service, we need to reach extremely high confidence that the core of the system is correct. We have found that the standard verification techniques in industry are necessary but not sufficient. We use deep design reviews, code reviews, static code analysis, stress testing, fault-injection testing, and many other techniques, but we still find that subtle bugs can hide in complex concurrent fault-tolerant systems.*
>
> *… [H]uman fallibility means that some of the more subtle, dangerous bugs turn out to be errors in design; the code faithfully implements the intended design, but the design fails to correctly handle a particular 'rare'*

scenario. We have found that testing the code is inadequate as a method to find subtle errors in design.

… In order to find subtle bugs in a system design, it is necessary to have a precise description of that design. There are at least two major benefits to writing a precise design; the author is forced to think more clearly, which helps eliminate 'plausible hand-waving', and tools can be applied to check for errors in the design, even while it is being written. In contrast, conventional design documents consist of prose, static diagrams, and perhaps pseudo-code in an ad hoc untestable language. Such descriptions are far from precise; they are often ambiguous, or omit critical aspects… At the other end of the spectrum, the final executable code is unambiguous, but contains an overwhelming amount of detail. We needed to be able to capture the essence of a design in a few hundred lines of precise description. As our designs are unavoidably complex, we needed a highly expressive language, far above the level of code, but with precise semantics. That expressivity must cover real-world concurrency and fault-tolerance. And, as we wish to build services quickly, we wanted a language that is simple to learn and apply, avoiding esoteric concepts. We also very much wanted an existing ecosystem of tools. In summary, we were looking for an off-the-shelf method with high return on investment. We found what we were looking for in TLA$^+$.

… In industry, formal methods have a reputation of requiring a huge amount of training and effort to verify a tiny piece of relatively straightforward code, so the return on investment is only justified in safety-critical domains such as medical systems and avionics. Our experience with TLA$^+$ has shown that perception to be quite wrong. … Engineers from entry level to Principal have been able to learn TLA$^+$ from scratch and get useful results in 2 to 3 weeks… without help or training. … Executive management is now proactively encouraging teams to write TLA$^+$ specs for new features and other significant design changes. In annual planning, managers are now allocating engineering time to use TLA$^+$.

Now that we know a little about what TLA$^+$ is used *for*, let me explain what it *is* from a practical point of view. A formal specification language, like a programming language, is a formal system (a language with precise syntactic and semantic rules), but one that focuses on *what* the program should do rather than on *how* it should do it. For example, a specification may say that a subroutine must return its input sorted — that's the *what* — without saying *how*, meaning which algorithm is used to sort. This description should immediately raise the objection that *what* any sub-system does forms the *how* of its super-system, or in programming-speak, the *what* is the *how* of a higher abstraction layer. For example, in specifying an algorithm detailing *how* to find median element in a list, a first step can be sorting the list, even though which algorithm is used for sorting is still irrelevant.

A versatile specification language should therefore be able to describe both the *what* and the *how*, or, better yet, describe at any desired level of detail the operation of any abstraction layer. The realization that *what* and *how* are just a relation between two abstraction levels on some spectrum is given a precise mathematical meaning through something known as an abstraction/refinement relation, which forms the very core of TLA$^+$'s theory.

This also suggests that a versatile specification language could *also* serve as a programming language, as it is able to describe both the *how* and the *what*. But we'll see that the requirements of programming languages may be at odds with what we want from a specification language, so there may be good reasons not to make a specification language serve double-duty.

There are different kinds of specification languages. Their essential differences (at least those felt by the user) are not so much due to a choice of mathematical theories so much as to differences in goal. The first kind is specification languages that are embedded in a mainstream programming language as contracts, either as part of the programming language itself or in annotations. Examples of such specification languages include **JML** for Java, **ACSL** or **VCC** for C, **Spec#** for C#, **Eiffel** contracts, **clojure.spec**, and **SPARK** contracts. That kind of specifications describe the intended behavior of individual program units like functions and classes, and can be used to verify the behavior of the units using tools like automatic **randomized test generation**, **concolic testing**, **model checking**, automated proofs with **SMT solvers** and manual proofs with proof assistants. While such specifications are very useful, they are limited in scope as they cannot easily specify global correctness properties of the program. For example, it's hard to write — let alone verify — a code-level contract that specifies that the program must eventually respond to every user request, that the program will never respond to any request from a user with information belonging to another user, or that the database the program implements is consistent or that it can never lose data.

Another kind of a specification language is one that is specifically designed to, or can with some effort, serve double duty as a programming language. Such languages include general-purpose proof assistants like **Isabelle/HOL** and **Coq**, specification and proof languages more directed at software like **WhyML**, and experimental programming languages based on dependent type theory such as **ATS**, **Agda**, and **Idris**. These languages have a very clear, unique and powerful advantage[4]: they allow what's known as end-to-end verification, namely the ability to specify and verify the behavior of a program from the highest level of global properties down to the machine instructions emitted by the compiler, ensuring that the executable conforms with the specification. This ability, however, comes at a very high cost: those languages are extremely complex, requiring months to learn and years to master. For this reason they are only used by specialists, very rarely in industry, and virtually never without the support of academic experts. To date, no one has been able to verify all interesting properties of large programs in this way. All instances of successful end-to-end verification are of relatively small programs or program components, and usually require an amount of effort that is far beyond the means of all but high-assurance software.

Finally, there are standalone specification languages that don't serve as complete programming languages but may (or may not) allow generation of bits of code in some programming language. Those include **Z**, **VDM**, **B-method**, **Event-B**, **PVS**, **ACL2**, **ASM**, and TLA$^+$. Z is used by Altran UK to create high-level specifications of their software (they also make use of SPARK to specify at the code level), and the B method is used in the railway sector; PVS is used at NASA (and is actually more similar in the theory it employs to tools like Isabelle/HOL and Coq). TLA$^+$ bears a resemblance to Z, B method and ASM, although it also bears some conceptual resemblance — especially in its generality — to ACL2, PVS and Isabelle.

Ideology and Aesthetics

Because there is no perfect formalism [5] — one good for every purpose — each is constructed around a set of ideological or aesthetic choices that are important to point, so we can delineate those parts of the debate over a preferred formalism — and, as programmers know, debates over formalism are common — that are a matter of aesthetics.

Leslie Lamport invented TLA$^+$ circa 1994, as a culmination of about two decades of work on *formal methods* — techniques that employ formal reasoning, reasoning based on the precise properties of formal systems, to analyze systems. For Lamport, who has made seminal contributions to the theories of concurrent and distributed algorithms, formal methods have always been first and foremost a practical necessity, a tool without which complex or subtle algorithms could not be designed because a complex or subtle algorithm that is not formally proven correct, is likely wrong. [6] In a **comment** about his first verification paper, written in 1977, Lamport writes of his interest in algorithm verification:

> *This has been a very practical interest. I want to verify the algorithms that I write. A method that I don't think is practical for my everyday use doesn't interest me.*

Lamport cares deeply about how engineers apply and think of formal methods[7], and his theory is driven first and foremost by practicality:

> *We are motivated not by an abstract ideal of elegance, but by the practical problem of reasoning about real algorithms. Rigorous reasoning is the only way to avoid subtle errors… and we want to make reasoning as simple as possible by making the underlying formalism simple*[8].

Practicality is an ideal whose realization depends on carefully deciding what *uses* the formalism serves (and what uses it does not) as well as *who* is its intended audience (and who is not). TLA$^+$ was designed for engineers working in industry and for algorithm designers either in industry or academia; it is *not* designed for mathematicians, logicians or programming language theorists. It addresses the need for a specification and verification of real systems and algorithms. It is *not* intended as a programming language, nor as a tool for exploration of novel logical and mathematical ideas.

Practicality — now with a clear audience and intended use in mind — entails two requirements: **simplicity**, to allow engineers to quickly learn and use the language, and **scalability**, to allow applying the formalism to real-world software or hardware of considerable complexity and size, as well as the nice-to-have **universality**, to allow applying the same formalism to the different kinds of algorithms and systems an engineer may encounter.

But while scalability can be fairly easily tested, and universality is a mathematical property of the formalism, simplicity (as a cognitive measure) often depends on a personal aesthetic point of view. As my goal is to compare and contrast the underpinnings of TLA$^+$'s theory with other approaches, I would like to describe an approach to program analysis which has been popular in programming language theory circles in the last few decades, to which TLA$^+$ stands in stark contrast.

The approach to program analysis taken by programming language theory, which is also connected with the theory of functional programming, is based on two aesthetic choices grounded on a historical view and a scientific mission. The first is the (partial) identification of computation with functions[9];

the second is the attempt to unify **constructive mathematics** — a rethinking of mathematical practices originating in part from a strict philosophical view of mathematics — with programming[10]. That attempt has led to the conjecture that a unification of math and programming could also work the other way around, namely that it is a good idea to mathematically reason about computer programs in the same formal framework designed to program constructive mathematics. This thesis appears explicitly in the introduction of a **paper** about the Lean proof assistant, **a tool designed** "to support both mathematical reasoning and reasoning about complex systems":

> *In practice, there is not a sharp distinction between verifying a piece of mathematics and verifying the correctness of a system: formal verification requires describing hardware and software systems in mathematical terms, at which point establishing claims as to their correctness becomes a form of theorem proving. Conversely, the proof of a mathematical theorem may require a lengthy computation, in which case verifying the truth of the theorem requires verifying that the computation does what it is supposed to do.*

While the introductory words "in practice" serve as a justification, there is no actual evidence to support this thesis, which, while interesting, currently holds only *in theory*. It is a pure aesthetic choice. "Computational mathematics" is indeed very interesting, but as developers rather than logicians, we are more concerned with the opposite problem: how computation is represented mathematically rather than how math is represented computationally.

And, there is good reason to believe that this thesis — that computational math is the right tool for mathematically reasoning about computation — is false. In constructive mathematics, the most common computational object is the constructive, or computable, *function*. Algorithmically speaking, a computable function corresponds to a sequential algorithm. But the systems and algorithms most software engineers build and want to reason about aren't sequential, but interactive or concurrent. The kinds of algorithms that are most interesting and common in constructive mathematics are the least interesting and common in software and vice-versa. This is not to say that one cannot reason about general algorithms in such a formalism, far from it, but reasoning generally about arbitrary algorithms in a functional formalism can be **quite complicated**, and in any event, their treatment is quite *different* from that of sequential algorithms.

The aesthetic choice in the design of TLA$^+$ is diametrically opposite: to reason *precisely* — i.e. mathematically, for math is the language of precision — about programs, we shouldn't use a formalism designed to express exotic math in a programming language, but quite the opposite, use *ordinary* math to express and reason about our programs, preferably in a uniform way for all kinds of algorithms; in other words, describe programs in ordinary math. Lamport **writes**:

> *For quite a while, I've been disturbed by the emphasis on language in computer science… I believe that the best way to get better programs is to teach programmers how to think better. Thinking is not the ability to manipulate language; it's the ability to manipulate concepts. Computer science should be about concepts, not languages. But how does one teach concepts without getting distracted by the language in which those concepts are expressed? My answer is to use the same language as every other branch of science and engineering — namely, mathematics.*
>
> *… The obsession with language is a strong obstacle to any attempt at unifying different parts of computer science. When one thinks only in terms of language, linguistic differences obscure fundamental similarities.*

Lamport observes that programming languages are complex, whereas ordinary, classical math is simple. Programming languages need to be mechanically translatable to efficient machine code that interacts with hardware and operating systems, and they are used to build programs millions of lines of code long that then need to be maintained by large and ever-changing teams of engineers over many years. Such requirements place constraints on the design of programming languages that make them necessarily complex, but this complexity is not necessary for *reasoning* about specifications; specifications are orders of magnitude shorter than code, and reasoning doesn't require the generation of an efficient executable. If it is possible to separate programming and reasoning into separate languages, each simple on its own — or, at least, as simple as possible — there may be much to be gained by such an approach.

Lamport writes:

> *The primary goals of a programming language are efficiency of execution and ease of writing large programs. The primary goals of an algorithm language are making algorithms easier to understand and helping to check their correctness. Efficiency matters when executing a program that implements the algorithm. Algorithms are much shorter than programs, typically dozens of lines rather than thousands. An algorithm language doesn't need complicated concepts like objects or sophisticated type systems that were developed for writing large programs.*

The difference between the two approaches — reasoning about programs within the programming language itself or reasoning about programs with ordinary math — can be explained with the following analogy. Consider an electrical engineer designing an analog circuit, and a mechanical engineer designing a contraption made of gears, pulleys and springs. They can either use the standard mathematical approach, using equations describing what the components do and how they interact, or invent an algebra of electronic components or one of mechanical components and reason directly in a language of capacitors and resistors (one capacitor plus one resistor etc.) or that of gears and springs. Lamport's approach is analogous to the former, while the programming language theory approach is analogous to the latter.[11] Of course, to be convenient, the "standard mathematical" approach needs to be flexible enough that concrete domain objects — like resistors — could be modularly defined and composed.

In addition, TLA$^+$ does not identify computation with functions in any way. Instead, it directly represent computation as a discrete **dynamical system**, much as ODEs are used to describe *continuous* natural phenomena. As we'll see in parts 3 and 4 of this series, this means that TLA$^+$ possesses a very strong form of universality. Batch, interactive, sequential, concurrent, parallel, distributed, real-time and hybrid (AKA cyber-physical: systems that interact with the physical world and have both discrete and continuous components) algorithms[12], are all expressed using the same simple formalism. In addition, all program properties that we care about — from correctness to performance — are similarly naturally expressed, and proving them, for all kinds of algorithms, is done using the same proof. While the choice for representing computation in this way is still an aesthetic one, this uniform universality for different kinds of systems and different properties suggests that it is a natural one. Lamport writes:

> *Physicists don't have to revise the theory of differential equations every time they study a new kind of system, and computer scientists shouldn't have to change their formalisms when they encounter a new kind of system.*

Similarly, composition of components in TLA$^+$ is not function composition. We will explore composition in detail in part 4, but just to pique your curiosity, consider that a component imposes certain constraints on the behavior of the system, as it operates following some rules. It also interacts with its environment — be it the user, the network, the operating system or other components — on which it imposes no rules; the environment appears nondeterministic to the component. The composition of multiple components, is then the intersection of the constraints imposed by all components, or, in logic terms, the conjunction of the formulas describing them. Composition in TLA$^+$ is, therefore, simple logical conjunction (“and” or $\land$).

Most importantly, this choice of representation takes nothing from the ability to abstract to computations to any desired level; quite the contrary — it offers very elegant abstraction.

The linguistic approach, however, has some advantages, like the ability to do end-to-end verification. As the unified language can be mechanically compiled to an efficient executable, assuming that the compilation process is itself verified, we can be certain that no mistake has crept into the pipeline from the top-level specification all the way down to the machine instruction. It is tempting to believe that if the semantics of a programming language were well defined, and if the composition rules were simple and consistent, then we could apply formal reasoning directly to program code and easily move up and down the abstraction hierarchy in order to verify correctness of properties at different levels of abstraction. Unfortunately, this is not the case. [13] In practice, end-to-end verification is very expensive, and there is no sign that this will drastically change in the foreseeable future. For the time being end-to-end verification remains reserved for very specialized software, or very small, secure cores of larger systems, and thus forms its own niche. Luckily, very few software systems truly require end-to-end verification, and those that do are either small, or only require such strong guarantees of one or two small internal components.

Giving up on a goal that cannot at the moment be affordably realized nor required by the vast majority of software is a very reasonable concession, provided we can trade it for other advantages[14]; those would be simplicity, universality and scalability, things that can make it applicable for a wide range of software and for a wide range of users. Of course, nothing prevents us from complementing this approach with code-level specification tools, which we can use to verify *local* correctness properties, like those of particular code units, rather than an entire system.

Whether or not your aesthetic choices coincide with Lamport's, I think this radically different ideology from that of programming language theory deserves consideration if it indeed leads to a more pragmatic approach to verification.

The History of TLA$^+$

Leslie Lamport, best known for his major contributions to the theory of concurrent and distributed algorithms, began his work on formal methods in the late 1970s, from a pragmatic standpoint as we've seen. His work was roughly contemporary to that of Tony Hoare and Robin Milner, but from the very beginning took a different path from their algebraic/linguistic work on process calculi. He made some important contributions to the theory of verifying concurrent/interactive/non-terminating computations, especially to specification via refinement and to the concept of safety and liveness properties (terms he coined). Those mathematical ideas — that would serve as the foundation for TLA$^+$ — were developed years before the formalism itself. This sets TLA$^+$ apart from the specification languages based on programming language theory, where in many cases the formalism came first and the semantics later.[15]

Lamport writes that “Pnueli's introduction of temporal logic in 1977 led to an explosion of attempts to find new logics for specifying and reasoning about concurrent systems. Everyone was looking for the silver-bullet logic that would solve the field's problems… I was not immune to this fever.” But he became disillusioned with its practicality after seeing colleagues “spending days trying to specify a simple FIFO queue — arguing over whether the properties they listed were sufficient. I realized that, despite its aesthetic appeal, writing a specification as a conjunction of temporal properties just didn't work in practice.”

This led to his invention of the **Temporal Logic of Actions** in the late '80s, a logical formalism which, despite its name, tries to minimize as much as possible the use of temporal reasoning (from which it borrows some operators) and instead relies on a concept called *actions*. “TLA gave me, for the first time,” Lamport writes, “a formalism in which it was possible to write completely formal proofs without first having to add an additional layer of formal semantics.”

During a visit to Oxford in 1991, Lamport explored the possibility of adding TLA to the Z specification language, which had been developed there. But "Tony Hoare was at Oxford, and concurrency at Oxford meant CSP. The Z community was interested only in combining Z with CSP — which is about as natural as combining predicate logic with C++. "

So around 1993 Lamport invented TLA$^+$, a complete formalism for specification built around TLA. **This short note** by Lamport is a summary of how he created TLA$^+$ in a process of *erasing* programming constructs from the formal description of algorithms, and distilling algorithms down to their mathematical essence.

While TLA$^+$ was not originally designed with any form of mechanical verification in mind, in 1999, Yuan Yu wrote a model checker for TLA$^+$, called TLC, and in 2008 a team at INRIA built a mechanical proof checker for TLA$^+$, called **TLAPS**. In 2009, Lamport introduced PlusCal, an alternative syntax for TLA$^+$ specifications that looks like pseudocode yet is completely formal, and is automatically translated into readable TLA$^+$. Recently, **research has started** on building a more state-of-the-art model checker for TLA$^+$.

The Feel of TLA$^+$

A TLA$^+$ specification describes a system at some chosen level of detail. It can be no more than a list of some global properties, a high-level description of the algorithm, a code-level description of the algorithm, or even a description of the CPU's digital circuits as they're computing the algorithm — whatever level or levels of abstraction you are interested in.

Lamport writes:

> *I believe that the best language for writing specifications is mathematics. Mathematics is extremely powerful because it has the most powerful abstraction mechanism ever invented — the definition. With programming languages, one needs different language constructs for different classes of system — message passing primitives for communication*

systems, clock primitives for real-time systems, Riemann integrals for hybrid systems. With mathematics, no special-purpose constructs are necessary; we can define what we need.

… Perhaps the greatest advantage of specifying with mathematics is that it allows us to describe systems the way we want to, without being constrained by ad hoc language constructs. Mathematical manipulation of specifications can yield new insight.

TLA$^+$ is not a programming language. It has no built-in notion of IO, no built-in notion of threads or processes, no heap, no stack — actually, no concept of memory at all — and not even subroutines[16]. And yet, any software or hardware system, and *almost* any kind of algorithm, can be written in TLA$^+$ succinctly and elegantly. By reasoning about algorithms or systems rather than programs (in part 3 we'll explore the difference between the two) we gain power and simplicity; in exchange we give up the ability of mechanically translating an algorithm into an efficient executable[17]. Once you've specified your algorithm or system at the level or levels of abstraction that you find most important, you translate the algorithm to your chosen programming language manually [18]. If you are designing an algorithm, the TLA$^+$ specification will closely resemble the code, or, at least, be at the same abstraction level. However, most of the time, engineers use TLA$^+$ to design and reason about complete systems, in which case the specification will be at a much higher abstraction level than the code. Lamport likens this use of TLA$^+$ to that of a **blueprint when designing a house**[19].

A system or an algorithm — at any abstraction level — is expressed in TLA$^+$ as a single logical formula (obviously, if it is non-trivial, we compose it of more manageable pieces) that can be manipulated like any mathematical formula. As scary as that may sound to programmers, the experience is very similar to programming and can be learned by programmers faster than most programming languages. It's math that feels a lot like programming. TLA$^+$ is one of those rare combinations of simplicity, elegance, versatility and power, that, at least in me, evoked an impression similar to the one I had years ago when I learned Scheme. With the exception of its proof language — which is guided by other design goals — it is also rather minimalistic. TLA$^+$ is certainly not perfect, but to me, it feels as elegant and as finely crafted as Scheme or Standard ML.

Computer scientists and programmers love talking about abstractions. One of my favorite things about TLA$^+$ is that it gives a precise mathematical definition to the concept of abstraction, and allows us to reason about it directly: In TLA$^+$, the abstraction/implementation relation is expressed by the simple, familiar logical implication: $X \Rightarrow Y$ is the proposition that X implements Y or, conversely, that Y abstracts X.

Code-level specification language — whether based on contracts or on types — make a clear distinction between algorithms, expressed in the body of subroutines (e.g., a routine implementing the Quicksort), and algorithm properties (e.g., “the subroutine returns a sorted list”), expressed as contracts or types. Even contract systems or types systems that allow full use of the programming language when expressing properties (e.g. dependent types) still make this clear distinction: semantically, properties are distinct from algorithms. TLA$^+$ makes no such distinction. The property “the algorithm sorts”, and the algorithm Quicksort are just two specifications at different levels of detail, different levels of abstraction, and $X \Rightarrow Y$ therefore also means that specification X has the property Y.

As I mentioned above, TLA$^+$ makes it easy to naturally express many different kinds of algorithms: sequential, interactive, concurrent, parallel, etc. There is, however, one glaring omission: while TLA$^+$ allows specifying probabilistic algorithms, the logic lacks the power to reason about them, like specifying that an algorithm yields the right answer with probability 0.75; this shortcoming, however, is probably easy to rectify in a future version of the language. It is easy to specify properties like worst-case time or space complexity, and even properties like “the system will eventually converge to one of three attractors”.

But elegance, universality and power aside, the biggest practical impact you feel when using TLA$^+$ (and the biggest practical difference between TLA$^+$ and some other specification tools) is the availability of a model checker. A model checker can make the difference between a formal method that actually saves you development time, and one that may be prohibitively expensive. But, most importantly, with a proof assistant you can prove that an algorithm is correct only if it actually is, and, if you're specifying a complex algorithm or system, chances are it isn't. A model checker gives you, at the push of a button, a counterexample that shows you exactly where things have gone wrong.

TLA$^+$ can be seen to be made of three parts, a classification that guides the organization of the posts in this series:

At its core is TLA — the temporal logic of actions — which is in some ways analogous to ordinary differential equations. But whereas ODEs are used to describe *continuous* **dynamical systems**, TLA is used to describe arbitrary *discrete* dynamical systems. In addition, TLA accommodates reasoning that is useful when analyzing programs in particular, most importantly assertional reasoning and refinement, the latter is a precise mathematical definition for the concept of abstraction and implementation relationship between algorithms (e.g., a parallel mergesort implements, i.e. refines, general mergesort). While TLA incorporates some **linear temporal logic**, and despite its name, TLA's main design objective is actually to *reduce* the reliance on temporal logic and temporal reasoning as much as possible, because while simpler than many alternatives, it is not quite "ordinary math". We will explore TLA in depth in parts 3 and 4.

Just like ODEs describe the evolution of a continuous system over some **phase space** and the variables take values over some field, so too TLA requires some state space (but does not dictate a particular one) in which the state of the system, namely the values of the variables, is defined at any point in time. The "+" in TLA$^+$ uses a formal set theory based on ZFC to allow TLA variables to take on many kinds of values (atoms like numbers and strings, finite and infinite sequences, sets, records, and functions). We'll go over this "static" part of TLA$^+$ in part 2.

Finally, TLA$^+$ has a module system to allow information hiding and elaborate abstraction/implementation or equivalence relations. We'll go over the module system and composition in part 4.

The TLA$^+$ software package contains an IDE called "**the toolbox**", a $\LaTeX$ pretty-printer, and the TLC model checker, which lets you verify properties of algorithms written in a useful subset of TLA$^+$ on restricted finite-state instances of your system at the push of a button. The proof system TLAPS, **available as a separate download** but fully integrated with the toolbox, allows interactive work with the proof system for the verification of proofs. TLAPS is not a self-contained proof assistant, but rather a front-end for the TLA$^+$ proof language, which uses automated solvers and the proof assistant Isabelle as backends for discharging proof obligations. As this series focuses on the theory, I will not discuss the use of the TLA$^+$ tools, even though they are of the utmost practical importance. The learning material I linked to covers their use.

While the TLA logic is universal (with the aforementioned exception of probabilistic algorithms), TLA$^+$ is a tool for a job, and the job is the formal specification and verification of real-world algorithms and large software systems by practitioners. Like all elegant tools, it can be used to do more, but that doesn't mean it's optimized for tasks it wasn't designed to do. The ergonomics of the language as well as the current tooling means that it is not the best choice in every circumstance. For example, it can be used to mechanically prove general mathematical theorems, but interactive theorem provers designed for that particular task likely do a better job. It can also be used to specify programming languages, but lacking syntactic constructs that make it easy to embed different languages nicely[20], other tools may handle that task better. The limitations of the model checker and the proof system are such that for exploring numerical and statistical algorithms, I would suggest using specialized languages like Matlab, Octave or Julia. And even though the logic itself is universal, that does not mean that it is always the best formalism to derive any kind of insight about a system. Different formalisms may offer different insight.

Some Misconceptions about TLA$^+$

I would like to address a few interesting misconceptions about TLA$^+$ that I found online.

The first is that TLA$^+$ is only a tool for verifying distributed systems. It is true that these days, TLA$^+$ is mostly known in the context of distributed systems and concurrent algorithms. This has a few reasons: 1) Lamport's algorithmic work is in concurrent and distributed algorithms, and the Balkanized nature of computer science places certain limits on the influence of ideas outside the sphere in which their originators are known. 2) Few other *general* software verification formalisms are able to handle concurrency as elegantly as TLA$^+$, so that is just where it shines in comparison. 3) When engineers write software systems that are too complex or subtle to be obviously correct and could therefore benefit from formal verification, it is usually the case that a concurrent or distributed algorithm is involved. Nevertheless, there is nothing that intrinsically limits TLA$^+$ specifically to concurrent and distributed algorithms. In fact, TLA$^+$ has no special concurrency constructs — like message passing or even processes — at all.

Another was made in a **comment by Carl Hewitt**, inventor of the actor model:

> *TLA$^+$ incorrectly treats state as global, which is scientifically incorrect in addition to being disastrous for engineering.*

The error here is confusing a mathematical notation with the systems it describes. If we use math to describe the position along one dimension of n particles, we might define a vector $\boldsymbol{x}$, with components $x_1 \dots x_n$, each being the position (or state) of a single particle. This notation (which in TLA$^+$ is written as $x[i]$ instead of x_i) says absolutely nothing about the actual physical interaction of the particles. It's just notation, and it treats state as global no more and no less than the mathematical notation of the vector $\boldsymbol{x}$ does. That we have a *notation* for the vector $\boldsymbol{x}$, does not actually mean that each particle is instantaneously aware of the position of all others. TLA⁺ and math in general don't "treat state as global"; they just allow denoting some global notion of state, one that the actual system described does not have.

One may indeed ask why we'd ever want to allow the specification of non-physically realizable interaction at all, and the answer is that working in a formalism that restricts what's *expressible* to what's *physically realizable* would make it more complex and less general[21]. Another reason is that we may wish to define non-physical behavior as a high-level abstractions in order to specify and prove certain properties. For example, if we're specifying an algorithm for distributed transaction, we would very much like to show that our system would behave *as if* distributed transactions were instantaneous. In that case, we would specify our realizable, physically realistic, algorithm, then we'd specify the non-realizable abstract behavior, and then we'd show that our low-level algorithm *implements* the high-level behavior. If we weren't even allowed to describe the abstract behavior, we wouldn't be able to prove that the algorithm has this desired property.

Another (rather technical) misconception is that TLA⁺ is "not higher order", and therefore cannot specify some higher-order algorithms. We will get into the specifics of the formalism in the next three installments and examine its precise "order", which would make the error of this assertion clearly apparent, but at this point I'd like to point out that the very phrasing of the claim is an example of what Lamport calls the "Whorfian syndrome" — after the **Sapir-Whorf hypothesis**, which posits that the language we use shapes our thought — namely, a confusion of language with reality, or of the **signifier with the signified**, or of the formalism and that which it formalizes. The term "higher-order" is a property of the formal *description* of a system, not the system itself.

For example, the proposition "every set of natural numbers contains a minimal element" is higher-order when using a logic whose **structure**, or domain of discourse, is the natural numbers, but is first order in a logic whose structure is the sets of ZFC set theory[22]. Another example is continuous dynamical systems expressed as ODEs. If we'd like to specify a system whose first derivative is itself another dynamical system, we can make use of a higher order ODE like so: $\ddot{x} = -x$. But dynamical systems are usually written as first-order ODEs without any loss of expressivity or convenience, and we would write that same seemingly "higher-order" system as a first-order ODE like so:

$$\begin{cases} \dot{x_1} = x_2 \\ \dot{x_2} = -x_1 \end{cases}$$

When programmers say that a program is higher-order, they mean that it is parameterized by another program. But such an interaction between two programs or algorithms is only higher-order if we choose to model interaction in our language using function composition; there are other ways of expressing such a relationship.

Therefore, what constitutes "higher-order" vs. "first-order" depends on the domain of discourse and/or on the modes of composition. Both are features of the formalism, not of the formalized "reality". We will see that TLA$^+$ expresses behaviors that may perhaps be higher-order in some formalisms as first-order.

In general, Lamport has **this** to say about comparing formalisms:

> *Comparisons between radically different formalisms tend to cause a great deal of confusion. Proponents of formalism A often claim that formalism B is inadequate because concepts that are fundamental to specifications written with A cannot be expressed with B. Such arguments are misleading. The purpose of a formalism is not to express specifications written in another formalism, but to specify some aspects of some class of computer systems. Specifications of the same system written with two different formalisms are likely to be formally incomparable… Arguments that compare formalisms directly, without considering how those formalisms are used to specify actual systems, are useless.*

Finally, **this** was expressed in a discussion of one of Lamport's talks:

> *Writing code in a high-level language can help one think more clearly in exactly the same way that writing natural language or drawing blueprints can… we now have programming languages, not just specification languages, which can be very useful in prototyping high-level specifications, writing executable specifications, and even evolving those into actual programs… With all due respect to Leslie Lamport, who is a great computer scientist and programmer. But he should learn Haskell.*

That misconception is interesting because the commenter's expectation of what *should* be specified is shaped by the particular abilities of his favorite programming language (and I should note that from the level of flexible abstraction or concreteness offered by TLA$^+$, all programming languages seem almost equally constrained in their range of expression). I challenge that commenter to specify the very important and very reasonable program properties — easily, naturally and clearly specified in TLA$^+$ — in his favorite programming language, such as: "every user request would eventually be answered by the server", or "when constructing a response to a request by a user, no information belonging to a different user will be read from the database", or "the transactions appear to have taken place instantaneously (i.e., they're linearizable)", or "the worst case complexity of this sort function is $5n \log n$". As I already mentioned, there are some research programming languages that do allow expressing such properties, but they make their own nontrivial tradeoffs.

Lamport has this to say on the matter:

> *[D]o we need to specify programs in a higher-level language before implementing them in a programming language? People who design PLs would say no, their languages make what the program does so obvious that*

> *no higher-level description is needed. I think the first PL for which this was believed to be true was FORTRAN. It's not true for FORTRAN and I don't think it's true for any existing general-purpose PL.*

Conclusion: On the Merits of Formal Methods

As engineers, we should use tools that help us build software that complies with requirements at the lowest cost. Formal methods and, in fact, all software verification methods — including the many forms of testing — lie on a spectrum of the effort they require and the confidence they provide, and we should pick those that match the requirements of the system we're building. It's too early to tell precisely how much TLA$^+$ helps to develop software in general, and exactly what problems benefit from it most, but I believe — and experience like that of Amazon seems to support that — that a large class of software systems can significantly benefit from a tool like TLA$^+$.

One skepticism towards formal specification is that it demands too much. Not all software requires formal specification and reasoning. There is indeed little use for a formal specification of things that are trivial or unimportant. But those parts of your software that are complex or subtle, and important, will benefit from some careful thinking, and a formal spec helps you think. An objection to formal specification even in cases that could potentially benefit from it was expressed in a comment on **this post**:

> *Even if you have a perfect proof that a program satisfies a specification, how do we verify that a specification is correct? … It's hard to believe that a programmer who have trouble write a correct program in the first place can magically write perfectly correct specification. People with experiences with mathematical logics know how hard and technical it is to encode precisely what we want to prove in a logical language even for relatively simple combinatorial facts. I don't say it's impossible to*

> *write a correct specification, but it is much harder than writing a correct program in the first place.*

Lamport addressed such claims in a 1979 paper, **On the Proof of Correctness of a Calendar Program**, written in a response to another paper that claimed that some software — like calendar programs — is hard to specify correctly. In the paper [23], Lamport addresses this very objection:

> *… We believe that if one really understands what a program should do, then he can specify it precisely in an understandable manner.*

A high-level formalism like TLA$^+$ is designed to allow a precise description of a software (or hardware) system, that allows both its assumptions as well as operation to be stated clearly and concisely enough that the correctness of the specification is far more likely than the correctness of a program. As to the question of the ability of programmers to write such specifications, if you are able to convert informal requirements to a program — which is just a formal specification at a fairly low level — and if you understand your program well enough, then you are also able to specify it formally at a higher level, in a way that shows how and why it works. Actually doing it is a matter of some relatively short training and some practice. The practical successful experience of a company like Amazon with TLA$^+$ shows that it is both useful and easy enough to use.

Other critics may claim that the right programming language, the right programming style (like object-oriented programming or functional programming), or the right discipline (like test-driven development) are all that's required to create good software. In the same 1979 paper, Lamport writes:

> *Advocates of programming methodologies have tended to talk as if their methodologies automatically generate good programs. A programming methodology is no substitute for intelligent reasoning about algorithms and*

> *their complexity, and cannot by itself lead one to a good method of solution. "Structured programming" would not have helped Euclid discover his algorithm.*

The experience of Amazon has shown that using formal methods wisely can *supplement* other methodologies, and reduce the cost of development.

At the opposite end of those who claim formal reasoning is ineffective, there are those with unrealistic expectations. The authors of Spec#, the code-level specification language for C# **report**:

> *[A] conclusion we have drawn from our interaction with developers is that real developers do appreciate contracts… Unfortunately, we have also seen an unreasonable seduction with static checking. When programmers see our demos, they often develop a romantic enthusiasm that does not correspond to verification reality. Post-installation depression can then set in as they encounter difficulties while trying to verify their own programs.*

Reality is **somewhere in the middle**: we can feasibly verify *some* properties of *some* systems, and in general, the more complex or big the program, the more tricky the property, or the more confident we wish to be in the veracity of the verification process, the more work is required. Some reasonable compromise must always be made depending on the software requirements. I believe that TLA$^+$ hits a sweet spot in the compromises it makes, and its versatility in choosing the desired level of detail or abstraction in the specification gives the user freedom to pick a useful point in terms of utility and effort. TLA$^+$ offers simplicity, universality and scalability, which it achieves by making two concessions: not being a programming language, and not trying to be a general tool for studying theory. The former largely sacrifices end-to-end verification, which is neither feasible nor necessary for virtually all ordinary software; the latter sacrifices power which is of no