The Future of TLA$^+$

Leslie Lamport, Stephan Merz, and Chris Newcombe

9 July 2024

Preface

The future of TLA$^+$ is now in the hands of the TLA$^+$ Foundation. This note is intended as a guide for how the Foundation should consider proposals to change the language. It is about TLA$^+$ itself, but an appendix discusses PlusCal and possible other languages that are translated to TLA$^+$.

The superscript 1 at the end of this sentence is a link to an end note; click on it now.[1]

1 Introduction

The TLA$^+$ language has been unchanged since 2006. Changes to it have a cost, so they require justification. In this note, we discuss the cost and possible justification of changes to the various parts of the language. We begin with some general comments on changing TLA$^+$.

The most common changes requested by users have been motivated by wanting TLA$^+$ to be more like a programming language. Any proposed change with that as its *only* justification should be immediately rejected. The reason for using TLA$^+$ is that it isn't a programming language; it's mathematics. That's what makes it better for its purpose than a programming language. Perhaps the most difficult thing for users to learn is how to use TLA$^+$ to think more productively by thinking mathematically. Making TLA$^+$ more like a programming language is a hindrance, not a help.

There are three possible costs of a change.

- Making existing specifications illegal.
- Complicating the language. Simplicity was a major goal of the TLA logic that underlies TLA$^+$. Simplicity is a major goal of TLA$^+$. Any new feature complicates the language, making the language more intimidating even for users who don't use that feature.

- Requiring changes to the tools. This is currently a major concern because there are few people contributing to the maintenance of the tools. All changes require changing the parser.

One kind of change that doesn't complicate the language or require changing tools is removing an existing feature from the language. The only tool that requires changing is the parser, and those changes should be minimal. Removing a feature simplifies the language, which is a very good justification. However, it can make existing specifications illegal, so it requires more justification than simply simplification unless the feature is almost never used.

Programming languages often add features to save a few keystrokes. (For example, the ++ construct of C and its successors.) This may be a good idea when programs can contain many thousands of lines of code. It is not a good idea for TLA$^+$, where specifications are seldom more than about 2000 lines long. We believe that to justify the cost in complexity and in changing the tools, a feature added just to save space should reduce by a significant amount the average size of a significant class of specifications.

There are two kinds of changes that might be proposed: Changes that affect the underlying formal semantics of TLA$^+$ formulas, and changes to how those formulas are written. They are considered separately.

2 The Underlying Formalism

2.1 TLA

TLA$^+$ is based on TLA, which is a temporal logic where the meaning of a formula is a predicate on behaviors, a behavior is a sequence of states, and a state is an assignment of values to variables. TLA evolved from the Floyd-Hoare approach to reasoning about correctness of programs, and is the only temporal logic that provides a practical way to describe both concurrent programs and their properties. Any change to this basis would produce a language that should not be called TLA$^+$ and would not concern the TLA$^+$ Foundation.

2.2 Ordinary Math

TLA assumes a basis consisting of ordinary math, which does not contain any modal logic such as TLA.[2] TLA$^+$ formalizes this math in terms of what logicians call ZFC. ZFC was developed about 100 years ago, and until at

least the 1970s was considered by most mathematicians who thought about such things to be the standard formalism of ordinary math.[3]

Given the ubiquity of types in computer science, adding them to TLA$^+$ may be proposed. Even the simple decidable type systems of programming languages add a great deal of complexity, without providing significant benefit to TLA$^+$. The errors they catch are almost always quickly found by model checking. While such a type system would make tools easier to build, Apalache shows how the necessary type declarations can be given to a tool without changing the language. The more expressive type systems that have been introduced to formalize mathematical proofs, such as Coq, are much more complicated. They would put TLA$^+$ beyond the ability of so many potential users that no proposal to add them should be taken seriously.

2.3 Proofs

Hierarchically structured proofs are necessary for keeping track of the details in correctness proofs. The only questionable semantic issue in the structuring is the decision to make ASSUME F PROVE G mean $\vdash F \Rightarrow G$ rather than $(\vdash F) \Rightarrow (\vdash G)$. A lot of thought went into that decision, and we doubt if anyone will propose changing it.

There are two kinds of semantic decisions embedded in what must be given to the backend provers. The first is the decision that the prover will not use a known fact or expand a definition unless it is explicitly instructed to do so. This may turn out to be unnecessary if provers become smarter, which should be possible with the use of AI.

The second kind of decision was how to provide instructions to the backend provers. There are now only three simple ways to tell backend provers how to prove something that are independent of the prover: HAVE, TAKE, and WITNESS. We don't know if these statements are useful enough and used enough to be worth keeping. Another way to instruct backend provers could be with a BY clause containing a formula F WITH $x \leftarrow A, y \leftarrow B$, where F has the form $\forall\, x \ldots, y \ldots : G$, that would denote the formula obtained by performing the indicated substitutions for x and y in G. Thus far, this hasn't been found to be necessary. Other instructions to the backend provers, including which provers to use, are provided in an *ad hoc* manner through the TLAPS module. A new prover might call these decisions into question.

3 The Parts of the Language

We now discuss possible changes to the TLA$^+$ language that don't affect its semantics. We have divided the language into five parts, discussed in separate subsections. The first are the parts of the language for writing formulas, which form most of a TLA$^+$ specification. Those parts are listed in the first two of the tables on pages 268–278 of *Specifying Systems*, to which the sections refer. The remaining two parts are proofs and commands for structuring specifications into modules.

3.1 Common Operators of Ordinary Math

This part of the language consists of the syntax to express ordinary operators of mathematics. They are the ones listed in Table 1, except excluding strings, the operators for records, and a couple of the operators on functions. They consist of the standard operators of logic and set theory.[4]

Standard mathematical notation has been used except when there was a good reason not to. This is important. Many users will have taken some math course that introduced these operators, and using the same notation for them reinforces the idea that TLA$^+$ is math. This should help users learn to think mathematically.

We believe that the mathematical operators of logic and set theory provided by TLA$^+$ are all the ones in common use, and they are the only ones that are not particular to any field of math. It's unlikely that additional ones are justified or that it's worth changing their names.

Many standard symbols, such as $\in$, have been replaced by the TEX names of the symbols. Some replacement was necessary because they had to be written in ASCII, and the TEX commands are known to mathematicians. But most of the documentation, including *Specifying Systems*, uses the actual symbols. We hope users will use the pretty-printed pdf versions of their specifications, containing the actual mathematical symbols, when explaining them to others.

In hindsight, the decision to encode the mathematics in ASCII was questionable. It was made because at the time, ASCII was the only encoding that seemed likely to be in use for many years to come. Symbols that had no obvious ASCII representation were given their TEX names because TEX and LATEX were, and are likely to be for a long time, the way most mathematicians typeset their papers. Today, Unicode seems like it might have been a feasible alternative, and perhaps some day it will obviate the need for an ASCII representation. However, we don't think that will happen soon.

A Unicode representation that can be automatically converted to the ASCII version is the best alternative for now. If Unicode is implemented, thought should be given to adding some more user-definable symbols to the ones in Table 5 of *Specifying Systems*. That's an addition that would not add complexity to the language and should require no change to a tool except for minimal changes to a parser. Removing or changing the precedence of some of those symbols is another possibility, but would affect existing specifications.

One possible candidate for removal is quantification over tuples, as in $\forall \langle x, y \rangle \in S : F$. We don't know if its use is rare enough to permit it to be removed. One justification for its removal is that the meaning of $\forall \langle x, y \rangle \in S : F$ is not obvious if S contains elements that are not ordered pairs (two-tuples).

Another possible candidate for removal is the notation for binary, octal, and hexadecimal numbers—such as `\022`, which is usually written 22_8 and equals 18. Again, we don't know if its use is rare enough.

3.2 Uncommon Operators of Ordinary Math

Uncommon operators of ordinary math in TLA$^+$ are the ones in Table 2 plus the record operators and a couple of the function operators of Table 1. They are all useful, and we would not expect to find a significantly better way to express any of them.[5]

3.3 TLA operators

The TLA operators are the action operators of Table 3 and the temporal-logic operators of Table 4. There should be no need to change any of them, since the basic semantics of TLA will not change. The only conceivable additional operator that couldn't be defined by a user would be a temporal CHOOSE operator. However, we have found no need for it and don't envisage any.

There are several candidates for elimination: action composition, $\stackrel{+}{\triangleright}$, $\exists$, and $\forall$. Action composition is rarely used, and it's not supported by any of the tools. But it was deemed useful enough by Markus Kuppe recently for him to try adding it to TLC. The operator $\stackrel{+}{\triangleright}$ is needed only to write open system specifications. No one writes such specs and no tool supports $\stackrel{+}{\triangleright}$. It's discussed only in one paper and in *Specifying Systems*. It may be useful some day if people start writing open-system specs, so it may be best to let parsers accept it but not to mention it. The temporal logic quantifiers $\exists$

and $\forall$ are not supported by any current tools, but it might not be difficult to add to TLAPS a backend prover that handles them. They could be used to express hyperproperties, but that might also require another temporal operator. The $\exists$ operator is needed to explain the theory underlying how TLA$^+$ is used.

3.4 Proofs

We see no reason to change the syntax for writing proofs unless there is a change to the semantics as discussed above in Section 2.3. However, the NEW ACTION and NEW TEMPORAL declarations seem to be useless and could probably be removed.

One set of features added for writing proofs that may bear re-examining are the ones for naming subexpressions of a formula. These features seem to be documented only in Section 6 of *TLA$^+$ Version 2: A Preliminary Guide*. While subexpression names can be used anywhere in a specification, their use outside of proofs is deprecated. Positional naming is complicated, and adding labels clutters a specification. However, the alternatives are decomposing the definitions to give names to the subexpressions, which complicates the specification, or copying the subexpression in the proof. More data on how useful they are in proofs should be obtained before any changes to these features can be proposed.

3.5 Structuring

This part of TLA$^+$ consists of the syntax for beginning and ending a module; the declarations EXTENDS, INSTANCE, CONSTANT(S), and VARIABLE(S); and operator and function definitions. These seem to be sufficient, and there doesn't seem to be any way to significantly improve them. Two constructs could perhaps be eliminated: submodules and the RECURSIVE declaration.

Submodules are used for the following reason. Informally, one can first define a specification *Spec* in terms of a variable x and then write $\exists\, x : \mathit{Spec}$. In TLA$^+$, *Spec* has to be defined in a module $M1$, and what we write naively as $\exists\, x : \mathit{Spec}$ has to be defined in another module $M2$ that instantiates $M1$. It's convenient to make $M1$ a submodule of $M2$ so it can use declarations and definitions from $M1$. Submodules are handled like other modules by a tool, so as long as $\exists$ remains in the language, it seems reasonable to keep submodules.

The RECURSIVE declaration is redundant, since it can be inferred from which definitions each definition depends on. However, we feel it's useful for catching inadvertent circular definitions.

Appendix

PlusCal was intended to be a gateway to TLA$^+$. In particular, it was meant to be used in a course on concurrent programming so that algorithms could first be described in a language familiar to students. The course would begin by describing algorithms in PlusCal, and using TLC would require students to understand the TLA$^+$ translation of those algorithms, which would allow the course to switch to using TLA$^+$.

PlusCal turned out to be useful for people who already knew TLA$^+$ because it was a convenient way to write some specs, especially because it eliminates having to explicitly describe the *pc* variable. There are also users who describe systems only with PlusCal, using just the ordinary math of TLA$^+$ for writing PlusCal expressions. (For example, see the book *Practical TLA$^+$* by Hillel Wayne.) However, users realize TLA$^+$ is there "under the hood", so PlusCal does contribute to the success of TLA$^+$.

To be successful, a gateway must attract people. Potential users of TLA$^+$ will be most attracted to something that looks like a currently fashionable programming language. What is fashionable in mathematics takes decades to change; fashions in programming language change every few years. Being twenty years old, PlusCal cannot be an ideal gateway to TLA$^+$.

We can expect proposals to add features to or modify PlusCal, or to replace it with a more fashionable language. The TLA$^+$ Foundation is concerned only with gateway languages, and one requirement for a gateway is that it be translated to TLA$^+$. But to serve as a gateway, the translation must also be simple. The correspondence between the language's code and its translation should be reasonably obvious.

Since new features complicate the translation, languages that enhance or modify PlusCal should probably be regarded as new versions of the language. The foundation will have to decide on a case-by-case basis whether a proposed language should be considered PlusCal, a variant of PlusCal, a new language, or not a suitable gateway to TLA$^+$ and therefore not relevant to the Foundation.

End Notes

Note 1 You can now return whence you came by clicking on the superscript *back* at the end of this note.[back]

Note 2 However, most mathematicians are probably unaware of CHOOSE, which logicians call Hilbert's epsilon and write ε.[back]

Note 3 The one way the semantics of this aspect of TLA$^+$ differs from the usual semantics of ordinary math is that a function is usually defined to be a set of ordered pairs while TLA$^+$ considers them a primitive defined by a couple of axioms. A major reason for this is that tuples are often used in math, and the most convenient way to formalize how they are used is to define a tuple to be a function f whose i^{th} element is $f[i]$. It is simpler not to have to define a special kind of two-tuple for a function to be a set of.back

Note 4 The two major differences are angle brackets for tuples and square brackets for function application. The decision to make tuples functions means that enclosing them in parentheses would be a nightmare—for example, $(x+1)$ would be a one-tuple that would not equal $x+1$. Square brackets are used for function application because most mathematicians seem to be unaware of the difference between functions and what are called operators in TLA$^+$. The difference is especially important to TLA$^+$ users because a function, but not an operator, can be the value of a variable. Since users would not have learned the difference in math classes, it seems best to make the difference clear by using different notation for function application than for operator application.back

Note 5 No one seems to like the EXCEPT construct. It's not hard to come up with a nicer way to express simple uses of EXCEPT such as $[f \text{ EXCEPT } ![a] = b]$. However, one may also want to write

$$\begin{aligned}[f \text{ EXCEPT } &![(x^2 + 3 * y)/17] \,.\, foo &&= \ @ + 27,\\ &![(x^2 + 3 * y + 1)/17] \,.\, bar &&= \ 42\ ]\end{aligned}$$

and we don't believe there is a significantly better alternative for expressing the general concept.back