TLA+

A Report written for the Seminar Advanced Programming Tools at HPI

Daniel Stachnik

WS22/23

Motivation

> *For the human makers of things, the incompletenesses and inconsistencies of our ideas, become clear only during implementation.*
>
> — Fred Brooks, *The Mythical Man-Month*

In his influential book, Fred Brooks shared insights gained during his career as a software engineer. He found that thoughts and designs of software engineers are vague and error-prone until they are implemented [3]. This is because only during implementation do the intricate details of the problem become apparent. However, looking at how software is developed today, there is little evidence for or against any particular software design decision or architecture [27]. Quality standards and best-practices are based on experience rather than a clear scientific model. This resembles the way Romans built streets, and how most engineering was before it was formalized [27]. Notably, this differs software engineering from other engineering disciplines, for example, civil engineering.

Conceptually, in civil and in software engineering, we can broadly distinguish two phases, the design phase, defined as the conceptualization and framing of a system [6], and the implementation phase.[1]

Looking at the design phase in particular, we see that software engineering seems to lack rigorous design. In comparison, almost any other engineering discipline employs some form of computer aided design (CAD) software. In the case of civil engineering, this allows designing constructions and simulate their real live properties before even the first brick is laid.

What is the CAD for software engineering? What tool can we use to check our ideas before writing the first line of code?

For many use cases, TLA+ can be this tool. TLA+ is a lightweight formal method, the formal method part meaning that it's a method using mathematical notation to define program properties, and lightweight meaning that it's abstracted away from the actual implementation, thus being easier to specify and learn [30].

Overview of TLA+

TLA+, an acronym for Temporal Logic of Actions Plus, is a specification language that facilitates the design of programs and is particulary well suited for designing concurrent programs [15]. Unlike a programming language, a written TLA+ specification does not produce a runnable program that satisfies requirements [29]. Leslie Lamport, a Turing Award winner for his contributions to distributed computing, developed TLA+ to simplify concurrent program design [15].

[1] In case of an agile process, those phases repeat arbitrarily. [25]

TLA+ specifications can be analyzed with two tools: the TLAPS Proof System, which conducts mathematical correctness proofs, and the TLC model checker, which exhaustively checks all states of a given input variable set to demonstrate correctness for specific inputs [29]. The TLAPS Proof System facilitates writing proofs for the entire input space, such as all natural numbers, while the TLC model checker provides confidence in the correctness of specific inputs, such as natural numbers from 1 to 10 (in TLA+ shorthand: $1..10$).

Lamport believes that the model checker is especially valuable to software engineers because it requires less assistence from the user, yet still provides confidence in correctness [17]. For an example of TLAPS in action, see [20].

The TLA+ Language

This report covers the basics of TLA+ syntax. If you're interested in learning TLA+, many online tutorials are available, including:

- Leslie Lamport's video course for learning TLA+ [2]
- Leslie Lamport's PlusCal tutorial [3]
- Hillel Wayne's TLA+/PlusCal tutorial [4]

Leslie Lamport found that all algorithms he encountered could be modeled as state machines [13]. Thus, he based the core conceptual model of TLA+ on state machines. Following mathematical notation, deterministic finite state machines are typically defined as a quintuple (S, Σ, f, s, A) with S being the set of states, Σ the input alphabet, $f : S \times \Sigma \to S$ the state transition function, s the start state, and A the set of accepting states.[24]. This corresponds to the way state machines are taught in computer science classes [5] However, this representation quickly becomes cumbersome. Consider a simple increment operation like `x + 1`: we would need to specify every state before the increment and every state after the increment. f would end up as a mapping of 1 to 2, 2 to 3, 3 to 4, and so on. Besides leaving implicit what f does, we would also need to specify the mapping for each input. Instead, in TLA+, the transition function (commonly called $Next$ state function) specifies implicitly what the condition at the current state **and** the next state must be. For example:

```
1 Next == x \in 1..10 /\ x' = x + 1
```

Here, in the current state, x must be in range $1..10$, and the next state (`x'`) will be `x + 1` for all `x`.

[2] https://lamport.azurewebsites.net/video/videos.html.
[3] https://lamport.azurewebsites.net/tla/tutorial/home.html.
[4] https://www.learntla.com.
[5] As was confirmed when asking students participating in the seminar how they were taught state machines.

Both state machine definitions require a starting state. In TLA+, this is commonly defined as an $Init$ state function, which could be denoted as

```
Init == x \in 1..10
```

Note that in TLA+ variables can belong to a set. The TLC model checker will check all of those starting conditions separately.

While a concise way of defining state machines is essential, it's not sufficient to make TLA+ useful. As we are interested in the correctness of our specification, we want to define assertions and invariants that ensure properties of the specification. For example, we might want the system to ensure that all numbers in a set are smaller than another given number. Again adhering to mathematical notation, we could define the invariant $\forall p \in P : p < n$ which would check that $p < n$ for all elements of P. These properties are commonly referred to as safety properties [29] [15], because they state "what mustn't fail" [15].

On the other hand, we have liveness properties that ensure that "good things happen eventually" [15]. We might want to specify that each process of an operating system must be serviced eventually, e.g. there cannot be any starvation in scheduling. This would be hard to specify using traditional logic alone. The temporal logic eventually-operator <> can help specify this constraint in TLA+ using the following statement:

```
<>(\A p \in P : p < n)
```

This operator denotes that the condition in parentheses will eventually be true. In practice, liveness properties cover only a minor part of TLA+ specs [29] [17], which is why will not dive deeper into temporal logic. However, the tutorials mentioned above each have chapters on it.

PlusCal

Depending on the systems you have designed, you may find TLA+ too abstract for your needs. Software engineers are used to programming language syntax when specifying algorithms. To address this, Leslie Lamport introduced PlusCal, a DSL based on TLA+ [12]. PlusCal is defined within an ordinary TLA+ script and must be transpiled to TLA+ code before model checking it. A simple example of the Euclid's algorithm, as given in [14], could be written as:

```
(*--fair algorithm Euclids {
    variables x = M, y = N;
    {
        while (x # y) {
            if (x < y) {
                y := y - x;
            } else {
```

```
8                   x := x - y;
9               }
10          };
11          assert x = GCD(M,N) /\ y = GCD(M,N);
12      }
13  }*)
```

Note that the initial values of the variables `x` and `y` are set to `M` and `N`, respectively. Those denote constants which can be specified with a concrete instance using the model checker. The definition of `GCD` is omitted for brevity. `GCD` returns the greatest common denominator. The PlusCal snippet should be much easier to understand for people with prior programming experience.

Overview of the TLC Model Checker

This chapter is based on the description of the TLC model checker by Yu, Manolios and Lamport [31], if not otherwise noted.

The TLC model checker is a tool used for verifying TLA+ specifications. It works by exhaustively exploring the state space of the specification on some user-defined input variables, meaning that it systematically searches all possible states of the system. Unlike other tools presented during the seminar that use symbolic representation, TLC uses explicit state representation, which is better suited for asynchronous runs and more expressive. This is supported by [21], who found that model checkers using symbolic solvers couldn't analyze real-life industrial problems as much as explicit state checking.

To begin the verification process, TLC starts with the start states of the system. It then runs the transition functions, checks the assertions and invariants, and adds the resulting states to a queue in a breadth-first manner. The next state is taken from the queue and again applied to all transitions functions. This process continues until there are no more new states to be explored. If at any point, an assertion or invariants fails, the programs halts and returns the failing state transitions leading to the exception.

Who uses TLA+ and what can we learn from that

AWS

Amazon Web Services (AWS) offer a range of fault-tolerant distributed systems [22]. While the critical systems of AWS are highly tested using conventional automated tests, Newcombe et al. explained that various AWS teams found tests alone "inadequate as a method for finding subtle errors in design, as the number of reachable states of the code is astronomical" [22]. Even though they felt intimiated by formal method's reputation as being hard-to-use and an unproductive tool for software development,

they felt the status quo was unacceptable. Hence, they investigated formal methods and decided to try TLA+ and the TLC model checker.

After testing TLA+ on several projects, they concluded that it provided higher confidence in correctness, particularly for concurrent systems [22]. They found the syntax simple yet highly expressive, and that the bugs found using TLA+ wouldn't be found using other techniques like testing. Crucially, they found TLA+ to be a feasible tool for use in mainstream software development.

[22] and [21] further elaborate that TLA+ is seen as a "what-if" tool for system design. It increased the iteration speed of system designs and facilated big and safe refactorings. In addition, TLA+ scripts provided precise documentation for the whole team, thanks to its emphasis on mathematical notation.

Lastly, [22] also saw improvements in their system designs because TLA+ required them to state precisely 'what needs to go right'. However, the authors did not state with respect to what the design of the system improved.

ESA

The European Space Agency (ESA) relies on real-time operating systems to control time-critical components. The Rosetta spacecraft used a real-time operating system called *Virtuoso*, which was later developed from scratch using TLA+ and named *OpenComRTOS* [28].[6] This allowed for a direct comparison of developing with and without formal methods.

In general, the authors found TLA+ useful [28]. They stated that TLA+ helped engineers "formulate their thoughts clearly while exposing hidden assumptions". They observed several positive outcomes from using TLA+ for their project. First, communication within the team became clearer. This is similar to the experience of [22], who found TLA+ a precise documentation for inter- and intra-team communication because it reduced ambigious meaning. Second, they explained that using TLA+ did not require extra resources or time. Third, the resulting code size was estimated to be 5-10 times smaller than the predecessor system. However, it is important to note that both the reduction in code size and the same estimated development time could be attributed to the reimplementation of the operating system and the use of a different architecture.

Finally, the authors cautioned that complex designs make full human understanding infeasible. Thus, model checkers can augment human capabilities in testing complex concurrent systems. While rewriting a system can lead to a better system because requirements are better understood, they suggest that designing more rigorously in the first place may be a better alternative to rewriting from scratch.

[6] Technically, both were developed by the contractor company *altreonic* under Eric Verhulst.

Learnings

The previous cases illustrate where TLA+ and other lightweight formal methods might be useful.

The most significant advantage of using TLA+ is the increased confidence in the correctness of the implementation. Both [28] and [22] dealed with concurrent algorithms, which they considered challenging to develop due to the inherent high number of possible state variations. The efficiency of the TLC model checker in reaching high levels of coverage and finding bugs has been noted as a significant benefit by engineers working at Amazon [22] and Intel [1].

Another significant advantage of TLA+ is that it enables the precise specification of high-level overviews of systems. In contrast to traditional methods that use vague descriptions or diagrams, TLA+ requires precisely stating what “needs to go right” [22]. This process can help reveal implicit assumptions and clarify the core relationships of the underlying system [21]. By forcing a discrete state machine design, TLA+ might also lead to cleaner designs that are easier to understand and modify [22] [28].

Lastly, the use of TLA+ can provide a shorter feedback loop for testing design ideas. According to [22], TLA+ has facilitated more extensive design refactorings, which can be validated much earlier using the TLC model checker. This resulted in shorter feedback cycles, allowing engineers to iterate quicker.

Limitations

Theoretical Limitations

Most of the theoretical limitations of TLA+ stem directly from its underlying conceptual models.

The state machine model [13] means that the states and the state transitions of the system design must be discrete. That is why TLA+ is not designed to support probabilistic systems [29]. It’s impossible to design systems using probabilistic state transitions because TLA+ has no support for floating point numbers [29]. Internally, the exact values of states are crucial for the TLC model checker to know when all states are visited. Floating point imprecisions and the enormous state space of floating points would make it challenging to support them.

As Leslie Lamport has a strong mathematical background, he based TLA+ on a few basic mathematical concepts, including basic set theory and predicate logic [15]. Following mathematical convention results in TLA+ being type-less. Lamport argues that types yield benefit in large systems [18]. However, TLA+ specs are meant to be short, so there would be no need for types.[7]

[7] Interestingly, it is still considered a best practice to define type invariants, commonly called `TypeOK` that specify the properties of the types.

The last big conceptual model underyling TLA+ is temporal logic [15]. Temporal logic does not treat time as a monotonically increasing continuous dimension but as a logical ordering of logic [29]. This means that we cannot state “this must happen within 10 ms after x” but only “this must happen after x”. However, the former would be necessary to guarantee real time properties. When developing a real-time operating system, Verhust et al. used traditional means of guaranteeing real-time properties, like mathematically proven time-deterministic Rate Monotonic Scheduling instead [28].

Finally, the implementation gap is an inherent limitation of all lightweight formal methods, including TLA+. A formally written specification is not an implementation, thus TLA+ can only verify the design’s correctness, not the implementation itself. Nevertheless, proponents of TLA+ argue that implementing a correct specification produces only a few bugs, and even if issues arise, they are much easier to fix [22] [28]. However, my personal experience suggests that a significant number of bugs arise from a wrong assumption of the behavior of given APIs, which is not covered by TLA+ or other lightweight formal methods.

Practical Limitations

In addition to theoretical limitations, there are also some practical considerations to take into account. The primary practical limitation of TLA+ is the steep learning curve. According to estimates by Newcombe et al., engineers need at least one month of regular usage to write TLA+ on their own [21], while Batson et al. estimate that three months of regular usage are needed [1]. Leslie Lamport has compared learning TLA+ to learning any programming language [17].

Formally verifying real programs using state checking is practically impossible due to state explosion [30]. By modeling the system, thus abstracting details away, using TLA+, the number of states can be kept lower. However, state explosion must still be considered and gives a practical upper bound for the complexity of TLA+ specifications (as was noticed, for example, by [22]).

In addition to the learning curve and state explosion, there are two other practical limitations to TLA+. Firstly, the tooling for TLA+ is limited compared to that available for mainstream programming languages. The official way to write TLA+ scripts is using the TLA+ Toolbox [16], which is based on an older version of Eclipse and lacks some features such as code completion and advanced linting. Secondly, there is fragmentation within the TLA+. Besides pure TLA+, there is PlusCal, which is often recommended for software developers getting started with TLA+. However, there are two syntax versions of PlusCal, a “C” and a “P” version. Additionally, any TLA+ specification can be pretty-printed, rendering the mathematical symbols instead of TLA+/PlusCal ASCII representations. This complicates searching for particular symbols or reading TLA+ specifications because one potentially needs to know all of the possible ways to express the same idea.

Alternatives

TLA+ Compared to Testing

The most widespread way of verifying specifications is by writing tests [19]. Both TLA+ and tests provide confidence in the correctness of the system. Tests have the advantage that they work directly on the implementation, so there is no implementation gap. However, testing is practically infeasible for exhaustively checking the state space of non-trivial concurrent programs [22]. In comparison, TLA+ does not work on the implementation, but instead on a model of the implementation. Thus, while tests check the implementation, TLA+ checks the specification of the design. This means that testing requires an implementation as a reference system, thus can only be done while building software, whereas TLA+ can be used earlier in the development cycle to verify designs.

Due to TLA+ and automated tests having different focuses, they should be considered complementary [29].

TLA+ Compared to Other Formal Methods

While TLA+ is a popular formal method, there are many others available for use. One such method is Alloy.[8] Cunha and Macedo compared Alloy with TLA+ [5]. They found that, similar to TLA+, users describe an abstract specification in Alloy. Syntactically, they found writing Alloy to be similar to writing type definitions common in programming languages. This is one of the reasons Alloy is considered easier to use than TLA+ [21] [5]. [21] argues that Alloy is less suited for larger, complex and concurrent problems than TLA+ because TLA+ is more expressive in specifiying and model checking constraints. The focus on structural properties required some work-arounds in the past to model dynamic behaviors, compared to TLA+, which was designed on temporal logic [29]. As of 2020, Alloy introduced temporal logic, too.

Another formal method, Spark, is a formal programming language based on Ada that works directly on the implementation [4]. Spark allows software engineers to specify contracts in the form of preconditions, postconditions, and invariants, which the Spark compiler can check automatically [4]. Since the same program can be used for execution, there is no implementation gap. However, formally verifying code can significantly increase development time. According to [10], the total development time of formally verified programs is approximately 3.3 times longer than traditional development time.

[8] https://alloytools.org.

When should you use TLA+?

By now it should be clear that TLA+ can be a helpful tool. Even though [28] argue that using TLA+ did not increase the total development time, a considereable amount of time must be invested to achieve fluency in TLA+ [21] [1]. TLA+ and other lightweight formal methods provide some confidence in the correctness of the system, usually by means of exhaustive state checking [31] or symbolic solving [11].

Mary Shaw argues that correctness should not be the sole goal of software development, as the degree of oversight and consequences of failure vary between domains [26]. In domains where the consequences of failure are low and there is ample manual oversight, correctness may not be as critical. However, in domains where there is minimal manual oversight or high consequences of failure, a strong focus on correctness is essential. Accordingly, she argues that the use of formal methods should be based on the need of correctness. Just like other means of verification, TLA+, then, is a tradeoff between correctness confidence and effort.

While proponents argue that the use of TLA+ can improve the eventual design of the system without significantly impacting development speed [28], this claim lacks measurable evidence. Therefore, better design and faster development speed cannot be considered reasons to use TLA+ for now.

That said, considering all reported sources, use of TLA+ might outweigh the effort of learning and using it when:

1. The consequences of software failure include physical harm or significant financial costs.
2. Assurance is needed that a high-level design is correct.
3. The system is concurrent.

Inventing the Future

> *Going deeper, most software people are just trying to do FAB, and most of the tools are FAB tools — there is very little CAD and even less SIM in "software engineering". To my old eyes, this doesn't look or feel like real engineering process.*
>
> — Alan Kay

Similar to the two phase engineering process described in the introduction, Alan Kay describes the engineering process broadly as the interplay of CAD, SIM, and FAB, where Computer Aided Design (CAD) is verfied by Simulations (SIM), which inform the Fabrication (FAB) that yields the actual outcome [7] [8]. As we have seen, TLA+ and other lightweight formal methods provide some feedback on systems designs. Just like particle simulations give insights about the performance of engines, formal methods

give insights on the correctness of software, by exploring all possible states of an idealized abstraction. Unlike modern CAD, TLA+ does not provide guidance on how to improve the design, and the feedback is mostly limited to safety properties and some liveness properties.

But that need not be like that. We could imagine similar methods that check scalability of designs, for example by observing how the state size changes with respect to input sizes. Or, we could imagine methods that check how fault-tolerant designs are by deliberatly considering faulty state transitions. We could imagine methods that can improve our own designs, for example by changing the underlying state machines to minimize the total state space.

In TLA+, we could easily describe that a collection must be sorted by stating:

$$\forall i_1, i_2 \in DOMAIN(list) : i_1 <= i_2 \implies list[i_1] <= list[i_2]$$

This constraint is completely abstracted from the strategy of how the list is to be sorted. If we could find ways to compose and decompose assertions and invariants into bits that we already know, for example because we keep track of the strategies in a repository, users could be pointed to designs satisfying the constraints. Basically, a list of constraints could yield a design.

Lastly, there's the question of syntax. Leslie Lamport argues that software engineers suffer from what he calls whorfian syndrome; the confusion of language with the concepts represented by it [23]. Lamport advocates for using mathematical notation due to its historic prevelance in science. Undoubtedly, however, TLA+'s syntax is difficult to grasp for many people, especially considering that over 40 % of software developers describe themselves as self-taught [27]. We should aspire to make our tools work for everyone. Squeak Etoys is an example how mathematical concepts can be made much more accessible by making systems explorable and live [9]. A simple way to improve the unfamiliar syntax could be the usage of visual code blocks, as described in [2]. As the state transitions are inherently shallow, i.e., the model checker only needs to know the current state to determine the next state [31], TLA+ lends itself particulary to live programming. We can imagine writing one state transition after the other, while the model checker provides you with all valid states so far and dynamically runs all state transition functions based on those. This way, users can explore possible next states interactively. Is it time for *TLA+ meets Etoys*?

References

[1] Brannon Batson and Leslie Lamport. "High-level specifications: Lessons from industry". In: *Formal Methods for Components and Objects: First International Symposium, FMCO 2002, Leiden, The Netherlands, November 5-8, 2002, Revised Lectures 1*. Springer. 2003, pp. 242–261.

[2] Tom Beckmann et al. “Visual design for a tree-oriented projectional editor”. In: *Companion Proceedings of the 4th International Conference on Art, Science, and Engineering of Programming*. 2020, pp. 113–119.

[3] Frederick P Brooks. “The mythical man-month”. In: *Datamation* 20.12 (1974), pp. 44–52.

[4] Bernard Carré and Jonathan Garnsworthy. “SPARK—an annotated Ada subset for safety-critical programming”. In: *Proceedings of the conference on TRI-ADA'90*. 1990, pp. 392–402.

[5] Alcino Cunha and Nuno Macedo. “Alloy meets TLA: An exploratory study”. In: ().

[6] Peter Freeman and David Hart. “A science of design for software-intensive systems”. In: *Communications of the ACM* 47.8 (2004), pp. 19–21.

[7] Alan Kay. *How does one get as close as possible to mastering software engineering?* 2022. URL: https://www.quora.com/How-does-one-get-as-close-as-possible-to-mastering-software-engineering/answer/Alan-Kay-11 (visited on 03/05/2023).

[8] Alan Kay. *Is Software Engineering Still an Oxymoron?* 2022. URL: https://www.youtube.com/watch?v=D43PlUr1x_E (visited on 03/05/2023).

[9] Alan Kay. “Squeak Etoys authoring & media”. In: *Viewpoints Research Institute* (2005), pp. 1–7.

[10] Gerwin Klein et al. “Formally verified software in the real world”. In: *Communications of the ACM* 61.10 (2018), pp. 68–77.

[11] Igor Konnov, Jure Kukovec, and Thanh-Hai Tran. “TLA+ model checking made symbolic”. In: *Proceedings of the ACM on Programming Languages* 3.OOPSLA (2019), pp. 1–30.

[12] Leslie Lamport. “A PlusCal User's Manual”. In: (2016).

[13] Leslie Lamport. “Computation and state machines”. In: (2008).

[14] Leslie Lamport. “Euclid writes an algorithm: A fairytale”. In: *International Journal of Software and Informatics 5, 1-2 (2011)* 1 (2011).

[15] Leslie Lamport. “Specifying systems: the TLA+ language and tools for hardware and software engineers”. In: (2002).

[16] Leslie Lamport. *The TLA+ Video Course*. 2021. URL: https://lamport.azurewebsites.net/video/videos.html (visited on 03/05/2023).

[17] Leslie Lamport. *Thinking Above the Code*. 2014. URL: https://www.youtube.com/watch?v=-4Yp3j_jk8Q (visited on 03/05/2023).

[18] Leslie Lamport and Lawrence C Paulson. “Should your specification language be typed”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 21.3 (1999), pp. 502–526.

[19] Jihyun Lee, Sungwon Kang, and Danhyung Lee. “Survey on software testing practices”. In: *IET software* 6.3 (2012), pp. 275–282.

[20] Sebastian Merz and Markus Kuppe. *Let's prove Leftpad*. 2020. URL: https://github.com/hwayne/lets-prove-leftpad/tree/master/tlaps (visited on 01/22/2023).

[21] Chris Newcombe. "Why amazon chose TLA+". In: *Abstract State Machines, Alloy, B, TLA, VDM, and Z: 4th International Conference, ABZ 2014, Toulouse, France, June 2-6, 2014. Proceedings 4*. Springer. 2014, pp. 25–39.

[22] Chris Newcombe et al. "How Amazon web services uses formal methods". In: *Communications of the ACM* 58.4 (2015), pp. 66–73.

[23] Ron Pressler. *The Practice and Theory of TLA+*. 2017. URL: https://www.youtube.com/watch?v=15uy9Ga-14I (visited on 03/05/2023).

[24] Elaine Rich et al. *Automata, computability and complexity: theory and applications*. Pearson Prentice Hall Upper Saddle River, 2008.

[25] Sheetal Sharma, Darothi Sarkar, and Divya Gupta. "Agile processes and methodologies: A conceptual study". In: *International journal on computer science and Engineering* 4.5 (2012), p. 892.

[26] Mary Shaw. "Myths and mythconceptions: What does it mean to be a programming language, anyhow?" In: *Proceedings of the ACM on Programming Languages* 4.HOPL (2022), pp. 1–44.

[27] Mary Shaw. *Progress Toward an Engineering Discipline of Software*. 2015. URL: https://www.youtube.com/watch?v=lLnsi522LS8 (visited on 02/16/2023).

[28] Eric Verhulst et al. *Formal Development of a Network-Centric RTOS: software engineering for reliable embedded systems*. Springer Science & Business Media, 2011.

[29] Hillel Wayne. *Learn TLA+*. 2023. URL: https://www.learntla.com/ (visited on 03/05/2023).

[30] Jim Woodcock et al. "Formal methods: Practice and experience". In: *ACM computing surveys (CSUR)* 41.4 (2009), pp. 1–36.

[31] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. "Model checking TLA+ specifications". In: *Correct Hardware Design and Verification Methods: 10th IFIP WG10. 5 Advanced Research Working Conference, CHARME'99 BadHerrenalb, Germany, September 27–29, 1999 Proceedings 10*. Springer. 1999, pp. 54–66.