HAL
open science

Formal specification and verification

Stephan Merz

To cite this version:

Stephan Merz. Formal specification and verification. Dahlia Malkhi. Concurrency: the Works of Leslie Lamport, 29, Association for Computing Machinery, pp.103-129, 2019, ACM Books, 10.1145/3335772.3335780 . hal-02387780

HAL Id: hal-02387780

https://inria.hal.science/hal-02387780v1

Submitted on 2 Dec 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Formal specification and verification

Stephan Merz
University of Lorraine, CNRS, Inria, LORIA, Nancy, France

1. Introduction

Beyond his seminal contributions to the theory and the design of concurrent and distributed algorithms, Leslie Lamport has throughout his career worked on methods and formalisms for rigorously establishing the correctness of algorithms. Commenting on his first article about a method for proving the correctness of multi-process programs [32] on the website providing access to his collected writings [47], Lamport recalls that this interest originated in his submitting a flawed mutual-exclusion algorithm in the early 1970s. As a trained mathematician, Lamport is perfectly familiar with mathematical set theory, the standard formal foundation of classical mathematics. His career in industrial research environments and the fact that his main interest has been in algorithms, not formalisms, has certainly contributed to his designing reasoning methods that combine pragmatism and mathematical elegance. The methods that he designed have always been grounded in the semantics of programs and their executions rather than being driven by the syntax in which these programs are expressed, in contrast to many methods advocated by "pure" computer scientists.

The famous "Time/Clocks" paper [33] describes executions of distributed state machines and introduces the happened-before or causality relation, a partial order on the events occurring in runs. The "philosophically correct" way for reasoning about distributed executions would thus appear to be based on a partial ordering between operations. Indeed, Lamport explored this idea and proposed a method based on two relations called *precedes* and *may affect* [34]. This method can notably be applied to algorithms that involve non-atomic operations, such as the Bakery algorithm. However, Lamport felt that the method did not scale well, unlike assertional reasoning about global states of systems as viewed by an idealized external observer, even if no single process can observe them. This style of reasoning considers linearizations of distributed executions, and it generally requires algorithms to be described in terms of their atomic operations (an exception being [37]). The notion of an overall system invariant that is preserved by every operation plays a central role in this approach: such an invariant explains why the algorithm is correct. Assertional proofs have since been demonstrated to be completely rigorous, to be amenable to mechanized checking, and to scale well.

Lamport realized that there are two fundamental classes of correctness properties that arise in the verification of concurrent algorithms, for which he coined the terms *safety* and *liveness* properties [32]. Generalizing, respectively, partial correctness and termination of sequential programs, safety properties assert that "nothing bad ever happens", and liveness properties require that "something good happens eventually." These intuitive concepts were later formalized by Alpern and Schneider [7] who showed that any property of runs can be expressed as the intersection of a safety property and a liveness property. Although proofs of liveness properties generally rely on auxiliary invariants, the basic principles for proving safety and liveness properties are different, and the two are therefore best considered separately.

Lamport advocates describing algorithms in terms of state machines whose operations are atomic. Invariants, and more generally safety properties, are then established by induction: the invariant holds in all possible initial states and is preserved by every operation of the state machine. The proof of liveness properties usually relies on associating a measure with the states of the algorithm and showing that the measure decreases with every step as long as the "good state" has not been reached. This argument is made formal through the use of well-founded orderings, which do not admit infinite decreasing chains. A direct application of that proof principle would require fixing a scheduler that governs the execution of different processes—an undesirable requirement since one wants to establish the correctness of the algorithm for any "reasonable" scheduler. A useful generalization requires that as long as the target state has not been reached, no step of the algorithm increases the measure, "helpful" steps decrease the measure, and some helpful step will eventually be executed. In order to justify the latter (which in itself is a liveness property!), one invokes fairness assumptions [9, 42] that assert that executable operations will not be neglected forever, fixing the precise understanding of a "reasonable" scheduler for the particular application.

Another fundamental concept underlying the rigorous development and proof of concurrent algorithms advocated by Lamport is that of *refinement*. It allows a designer to describe the fundamental correctness properties using a high-level (possibly centralized) state machine and then prove that another state machine whose description is given at a lower level of abstraction faithfully implements the high-level description. For example, a high-level state machine describing a consensus algorithm [51] could have a variable $chosen$ holding a set of values, initialized to the empty set, a $Choose$ operation that assigns $chosen$ to the singleton set $\{v\}$ for some value v among the proposed values, and $Decide$ operations that set the decision values of each process to that chosen value. A lower-level refinement would then describe the actual algorithm in terms of exchanged messages and votes. A technical complication in that approach is that the lower-level state machine will have operations that modify only low-level variables, i.e., variables that do not exist at the higher level of abstraction. These operations cannot be mapped to operations of the high-level state machine. For example, operations that send messages in the putative Consensus algorithm have no meaning in the high-level specifi-

cation. Lamport advocates that formalisms for describing executions of state machines should be insensitive to *stuttering steps* that leave unchanged the state visible to the specification.

The remainder of this chapter will focus on the Temporal Logic of Actions TLA and the specification language TLA$^+$. These are Lamport's contributions to the formal specification and verification of algorithms that have had the greatest impact in the academic community and in industry, and their design was guided by the principles outlined above.

2. The Temporal Logic of Actions

Temporal logic [62] is a branch of modal logic in which the accessibility relation corresponds to temporal succession. During the 1970s, several authors [15, 31] suggested adopting temporal logic as a basis for proving programs correct. Pnueli's influential paper [61] provided the insight that temporal logic is particularly useful for specifying and reasoning about *reactive systems*, which include concurrent and distributed systems. A generalization of Pnueli's logic that is based on the "next" and "until" connectives remains the standard variant of linear-time temporal logic used in computer science to this day.

2.1 The Genesis of TLA

In 1977, Lamport proposed a method for proving the correctness of concurrent programs [32]. It used invariant reasoning (based on Floyd's method [22]) for establishing safety properties and lattices of leads-to properties for proving liveness. The method relied on a fixed progress assumption for each process in order to establish elementary leads-to properties.

Lamport was introduced to temporal logic in the late 1970s during a seminar organized by Susan Owicki at Stanford University. At that time, the distinction between linear-time and branching-time temporal logics was not yet clearly established in computer science. Lamport clarified this difference [35] and showed that the expressive powers of LTL and CTL are incomparable. He quickly realized that temporal logic was a convenient language for expressing and reasoning about fairness and liveness properties. For example, weak and strong fairness of executions with respect to an action α, representing an operation of a process in a concurrent system, can be written as

$$\Box(\Box en(\alpha) \Rightarrow \Diamond exec(\alpha)) \qquad \text{and} \qquad \Box(\Box\Diamond en(\alpha) \Rightarrow \Diamond exec(\alpha)).$$

In these formulas, the predicate $en(\alpha)$ characterizes those states in which α is enabled (may be executed), $exec(\alpha)$ is true when action α has been executed, and $\Box$ and $\Diamond$ are the "always" and "eventually" operators of LTL. Weak fairness requires that an action cannot remain perpetually enabled without eventually being executed. Strong fairness requires that even an action that is infinitely often (but perhaps not perpetually) enabled must eventually be executed. Using such formulas, more general fairness hypotheses than uniform progress of processes considered in [32] can be expressed unambiguously. Moreover, the principles of reasoning about leads-to lattices could be derived from the general proof rules of temporal

logic. A joint paper with Owicki [59] develops these ideas into a full-fledged method for proving the correctness of concurrent programs. In the introduction to this paper, the authors write:

> *While we hope that logicians will find this work interesting, our goal is to define a method that programmers will find useful.*

This motto describes well Lamport's approach to formalisms for specification and verification.

Whereas standard LTL was clearly useful for expressing fairness and liveness properties, Lamport felt that it was not convenient for writing complete specifications of actual systems. His intuition was confirmed when he observed colleagues at SRI struggling with specifying a FIFO queue in Pnueli's temporal logic. (It was later proved that this is actually impossible unless one assumes that the input values presented to the queue were unique.) This observation led his colleagues to introduce a more expressive temporal logic based on intervals [63]. In contrast, Lamport concluded that the fundamental problem was the "property-oriented" style of specifications as a list (conjunction) of properties observed at the interface of a system, such as the inputs and outputs of the queue, but excluding any reference to internal system states. He designed TLA as a logic geared towards specifying and reasoning about state machines, based on a few orthogonal and simple concepts that provide a higher level of abstraction (and elegance!) than the use of a pseudo-programming language, as in the earlier paper with Owicki [59].

2.2 The Logic TLA

Lamport designed TLA around 1990 [38, 40]. TLA formulas are built from *constants*, whose values are fixed throughout an execution, and (state) *variables*. We use x, y, z to denote constants and u, v, w to denote variables. A *state* is a mapping from variables to values. TLA distinguishes three levels of expressions:

- The syntax of state functions and state predicates is that of standard terms and formulas of first-order logic. Concrete examples of state predicates are $v \geq 0$ or $\exists x : x \in u \wedge x \notin v$. Semantically, they are interpreted over individual states.[1]
- Transition functions and transition predicates, also called *actions*, are first-order terms and formulas that may contain both standard (unprimed) variables u, v, w and primed variables u', v', w'. Examples are $u' \in v$ or $\exists x : u' + x = v$. Semantically, transition formulas are interpreted over pairs $\langle s, t \rangle$ of states, with unprimed variables being interpreted in state s and primed variables in t.

[1] We could distinguish a level of constant formulas that do not contain any variables, but we will consider such formulas to be state formulas.

- Temporal formulas are built from state and transition formulas by applying operators of temporal logic according to the rules given below. They are interpreted over *behaviors*, i.e., sequences $\sigma = \langle s_0, s_1, \ldots \rangle$ of states.

Whereas standard LTL builds temporal formulas solely from state formulas, the introduction of transition formulas as a primitive building block is fundamental to specifying state machines. For example, the LTL equivalent to the TLA action $u' \in v$ would be

$$\exists x : x \in v \wedge \circ(u = x)$$

where $\circ$ denotes LTL's next-time operator. Rather than using a pseudo-programming notation as in [59], actions are just first-order formulas over primed and unprimed variables. One reasons about them using ordinary mathematical logic, rather than special principles for reasoning about programs. At the temporal level, any formula can be considered as a system specification or as a property. There is no formal distinction between the two, and reasoning about them relies on the same fundamental principles of temporal logic.

TLA introduces several notations at the levels of state and transition formulas. Given a state formula e, the transition formula e' is obtained by replacing all (free) occurrences of state variables by their primed counterparts. Semantically, e' denotes the value of e at the second state of the pair of states at which e' is evaluated. The action UNCHANGED e is shorthand for $e' = e$. For an action A and a state formula e, the actions $[A]_e$ and $\langle A \rangle_e$ stand for $A \vee e' = e$ and $A \wedge e' \neq e$, respectively. The action formula $[A]_e$ represents closure of A under stuttering (with respect to e), in particular it is true of a pair of states $\langle s, t \rangle$ if A is true or if $s = t$. Dually, $\langle A \rangle_e$ requires A to be true and the step from state s to state t to be observable through a change of e. Finally, the state predicate ENABLED A is obtained by existential quantification over all primed state variables that occur in the action A. For example,[2]

$$\begin{array}{lrcl} \text{if} & A & \triangleq & v > 0 \wedge v' = v - 1 \wedge w' = w \\ \text{then} & \text{ENABLED } A & \triangleq & \exists v', w' : v > 0 \wedge v' = v - 1 \wedge w' = w \end{array}$$

It is easy to see that for this example, ENABLED A is logically equivalent to the predicate $v > 0$. In general, ENABLED A is true at state s if there exists some state t such that A holds for the pair $\langle s, t \rangle$.

Formulas at all three levels are closed under Boolean operators ($\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\equiv$) and first-order quantifiers ($\forall$, $\exists$). The rules for forming temporal formulas are as follows:

- Every state predicate is a temporal formula.
- If A is an action and e is a state formula then $\Box[A]_e$ is a temporal formula.
- If φ is a temporal formula then so is $\Box\varphi$.

[2] The symbol $\triangleq$ denotes "is defined as".

STL1. $\dfrac{\varphi}{\Box\varphi}$

STL2. $\Box\varphi \;\Rightarrow\; \varphi$

STL3. $\Box\Box\varphi \;\equiv\; \Box\varphi$

STL4. $\dfrac{\varphi \;\Rightarrow\; \psi}{\Box\varphi \;\Rightarrow\; \Box\psi}$

STL5. $\Box(\varphi \wedge \psi) \;\equiv\; \Box\varphi \wedge \Box\psi$

STL6. $\Diamond\Box(\varphi \wedge \psi) \;\equiv\; \Diamond\Box\varphi \wedge \Diamond\Box\psi$

TLA1. $\dfrac{P \wedge e' = e \;\Rightarrow\; P'}{\Box P \;\equiv\; P \wedge \Box[P \Rightarrow P']_e}$

TLA2. $\dfrac{P \wedge [A]_e \;\Rightarrow\; Q \wedge [B]_f}{\Box P \wedge \Box[A]_e \;\Rightarrow\; \Box Q \wedge \Box[B]_f}$

Lattice. $\dfrac{(S, \prec) \text{ is a well-founded ordering} \quad \forall x \in S : \varphi(x) \leadsto (\psi \vee \exists y \in S : y \prec x \wedge \varphi(y))}{(\exists x \in S : \varphi(x)) \leadsto \psi}$

Figure 1 Proof rules for simple TLA.

- If φ is a temporal formula, x is a constant, and v is a variable, then $\exists x : \varphi$ and $\mathbf{\exists}\, v : \varphi$ are temporal formulas.

The formulas $\Diamond\varphi$ and $\Diamond\langle A\rangle_e$ are shorthand for $\neg\Box\neg\varphi$ and $\neg\Box[\neg A]_e$, respectively. Also, $\varphi \leadsto \psi$ ("φ leadsto ψ") abbreviates $\Box(\varphi \Rightarrow \Diamond\psi)$. Observe in particular that if A is an action formula, $\Box A$ is in general not well-formed: actions need to be "protected" by square or angle brackets inside temporal formulas.

The operators $\Box$ and $\Diamond$ are the familar "always" and "eventually" operators of LTL: $\Box\varphi$ is true of σ if φ is true of every suffix of σ. The formula $\Box[A]_e$ is true of $\sigma = \langle s_0, s_1, \ldots\rangle$ if, for all $n \in \mathbb{N}$, the action A holds of the state pair $\langle s_n, s_{n+1}\rangle$ or the state formula e evaluates to the same value in s_n and s_{n+1}, and the interpretation of $\Diamond\langle A\rangle_e$ is dual. The syntactic restriction of allowing action formulas to appear only inside brackets ensures that all temporal formulas φ of TLA are insensitive to finite stuttering: if two state sequences σ and τ agree up to insertions or removals of finite repetitions of states then φ is true of σ if and only if φ is true of τ. The formula $\mathbf{\exists}\, v : \varphi$ is true of $\sigma = \langle s_0, s_1, \ldots\rangle$ if φ is true of a sequence $\tau = \langle t_0, t_1, \ldots\rangle$ such that for all n, s_n and t_n agree on the values of all variables except possible v. The formal definition is somewhat more complicated in order to preserve invariance under stuttering [40].

The notion of validity of TLA formulas is standard. In particular, a temporal formula φ is valid if it is true of all behaviors. Lamport also provides a set of proof rules for TLA. In particular, he states that the proof rules reproduced in Figure 1, plus ordinary first-order reasoning, are sufficient (in the sense of relative completeness [8]) for reasoning about algorithms specified in TLA without quantification over state variables. These rules should be read as asserting that if the hypotheses are valid, then so is the conclusion. For example, rule (STL1) is the well-known necessitation rule of modal logic, and it is justified because any suffix of a behavior is again a behavior, of which φ is true by the hypothesis of the rule. Of course, the implication $\varphi \Rightarrow \Box\varphi$ is not valid in general. From the elementary rules of Figure 1,

further useful verification rules can be derived such as the following rule for proving that a state predicate P is an invariant of a system specification

$$\text{INV1.}\quad \frac{P \wedge [A]_e \;\Rightarrow\; P'}{P \wedge \Box[A]_e \;\Rightarrow\; \Box P}$$

This rule, like most other rules for system verification in TLA, establishes a conclusion expressed in temporal logic from non-temporal (action-level) hypotheses. In this way, reasoning at the temporal level is confined to the top levels of a TLA proof and typically represents less than 5% of the proof steps. In particular, reasoning about safety properties does not involve temporal logic.

Lamport's rules are intended for the verification of algorithms. In contrast, providing a proof system for—even propositional—TLA that is complete in the standard sense of formal logic (i.e., that allows all valid formulas to be derived as theorems) is more delicate. In particular, whereas (INV1) is sufficient for proving invariants of systems, the general induction axiom of LTL

$$\Box(\varphi \Rightarrow \circ\varphi) \;\Rightarrow\; (\varphi \Rightarrow \Box\varphi)$$

cannot be expressed in TLA because there is no next-time operator that could be applied to temporal formulas. A generalization of TLA, together with a system of rules that is complete for the propositional fragment of that logic, appears in [55].

2.3 Refinement, Hiding, and Composition

We mentioned above that TLA does not formally distinguish between system specifications and their properties: both are represented as temporal formulas. In practice, specifications of state machines are usually written in the form

$$\mathit{Init} \wedge \Box[\mathit{Next}]_v \wedge L \tag{1}$$

where the state predicate *Init* specifies the possible initial states of the system, the action *Next* its next-state relation, v is the tuple of all state variables used in the specification, and L expresses fairness conditions. Typically, *Next* is a disjunction of actions A_i that describe atomic transitions of the system or of its environment, and L is a conjunction of strong or weak fairness conditions on (some of) the actions A_i.

Refinement of specifications. Perhaps influenced by ideas on program refinement developed by Back and Morgan [11, 57], ultimately inspired by Dijkstra [18], Lamport [36] already observes that "temporal logic supports hierarchical specification and reasoning in a simple, natural way". He also notes that the essential ingredient for this to be possible is the invariance of temporal logic formulas under stuttering, ensured by the absence of a next-time operator. The idea is that a refinement R of a high-level specification S introduces implementation detail, represented by additional state variables. Newly introduced actions that modify

solely these new variables correspond to stuttering steps at the level of S and, by stuttering invariance, cannot invalidate S. Actions that modify the variables present in S must do so in ways that are allowed by (the next-state relation of) S. Whereas R may have fewer behaviors (when projected to the state space of S), the fairness conditions in S must be preserved: if a high-level action A_i for which S states a fairness condition is sufficiently often enabled in a run, R must ensure that some action whose effect corresponds to the occurrence of A_i will eventually occur in that run.

Summing up, R refines S if and only if the implication $R \Rightarrow S$ is valid. Assuming that R and S are specified using formulas of shape (1) (with superscripts R and S), refinement is proved by finding a state predicate I such that all of the following implications hold:

$$\begin{aligned} \mathit{Init}^R &\Rightarrow I \wedge \mathit{Init}^S \\ I \wedge [\mathit{Next}^R]_{v^R} &\Rightarrow I' \wedge [\mathit{Next}^S]_{v^S} \\ \Box I \wedge \Box[\mathit{Next}^R]_{v^R} \wedge L^R &\Rightarrow L^S \end{aligned}$$

In words, I is an invariant of the low-level specification R that is strong enough to prove that every transition according to R's next-state relation is also a possible transition for S, possibly stuttering, and to show that R implies the liveness hypotheses asserted by S. The first two proof obligations establish the safety part of the refinement and do not involve temporal logic; the third one concerns liveness and requires temporal reasoning.

Hiding of internal state. The standard form (1) of specifications is useful for describing a system as a state machine, but it does not distinguish between variables that are visible at the interface and those that represent the internal state of the machine. This distinction is, however, important in the sense that the "contract" between the implementation of a system and its users should only constrain the interface, not the internal state. For example, a high-level specification S_{set} of a key-value store may represent the current content of the store in a variable *store* holding a set of pairs (k, v), but the implementer should be free to choose another suitable data structure, such as a hashtable. Because the operations of the lower-level specification S_{tbl} update a hashtable instead of a set of pairs, the implication $S_{tbl} \Rightarrow S_{set}$ cannot be proved. Indeed, the internal variable *store* (and the set of values that it holds) is not part of the interface of the system, which solely consists of the input and output channels through which the system corresponds with its environment. Lamport realized that this form of *information hiding* corresponds to existential quantification: the actual high-level specification of our system is not S_{set}, but rather $\exists \mathit{store} : S_{set}$, which asserts that the system behaves as if it contained a store represented as a set of key-value pairs. Similarly, the lower-level specification is $\exists \mathit{store} : S_{tbl}$,[3] and in order to establish refinement, we have to prove the implication

$$\big(\exists \mathit{store} : S_{tbl}\big) \;\Rightarrow\; \big(\exists \mathit{store} : S_{set}\big). \tag{2}$$

[3] Of course, it is immaterial if the names of the bound variables in the two specifications are the same or not.

Applying standard quantifier rules, that proof can be reduced to proving the implication

$$S_{tbl} \Rightarrow \big(\exists store : S_{set}\big)$$

where the internal state of the lower-level specification has been exposed. It now suffices to find a suitable state function s that provides a witness term for the internal state of the higher-level specification, i.e., such that the implication

$$S_{tbl} \Rightarrow (S_{set} \text{ WITH } store \leftarrow s) \tag{3}$$

is provable. (The notation used here for substituting the variable *store* by the expression s is not part of TLA.) In our example, a suitable witness s is provided by the contents of the hashtable.

State functions that serve as witness terms for refinement proofs are known as *refinement mappings*. They serve to reconstruct the value of an internal state variable used in the high-level specification from the corresponding lower-level state. Semantically, showing an implication of the form (2) requires exhibiting an infinite sequence of values for the quantified variable on the right-hand side, given a sequence of values for the variable on the left. In contrast, a refinement mapping as in (3) computes values state by state, which is clearly weaker: one cannot refer to previous or future values in the sequence of values satisfying the left-hand specification. Indeed, in general a suitable refinement mapping need not exist even if (2) holds. Abadi and Lamport [1] suggested that for the proof of refinement, the low-level specification may be augmented by *auxiliary variables*. The augmented specification is semantically equivalent to the original one, but the additional variables help in defining a refinement mapping. In particular, Abadi and Lamport defined principles for augmenting specifications by *history* and *prophecy* variables, and they provided sufficient conditions for these principles to be complete for proving refinement. The constructions were presented in a semantic framework independent of TLA. Proof rules for introducing auxiliary variables in TLA appear in [49].

Representing parallel composition. The above discussion has shown that refinement and hiding can be represented in TLA using implication and quantification, and therefore standard principles of logical deduction can be applied to reason about these concepts. Now consider two components, specified by TLA formulas Φ and Ψ, that are intended to operate in parallel. In order for both parallel components to adhere to their specifications, the variables of the first component must evolve as prescribed by Φ, and similarly for the second component. In particular, any steps that change a variable shared by both components must be permitted by both Φ and Ψ. Transitions that exclusively modify variables from one specification appear as stuttering steps to the other one and are trivially allowed by that specification, whereas variables shared between Φ and Ψ synchronize transitions of the two specifications. It follows that the formula $\Phi \wedge \Psi$ characterizes the parallel composition of the two specifications. (We

see in Section 3.3 that for some computational models, it may be useful to adopt a stronger specification $\Phi \wedge \Psi \wedge \Xi$ where Ξ expresses extra synchronization hypotheses embodied in the computational model.)

Expressing composition as conjunction corresponds to the overall philosophy of TLA that structural concepts are expressed by logical operators. Although the conjunction of two specifications Φ and Ψ written in the standard form of (1) is not itself in standard form, it can easily be transformed into standard form by applying the equivalence

$$\Box[A]_u \wedge \Box[B]_v \;\equiv\; \Box\big[[A]_u \wedge [B]_v\big]_{\langle u,v\rangle}$$

Specifying open or closed systems. When specifying a component, one invariably has to describe not just the component itself, but also the environment that the component is going to operate in. A closed-system specification of a component describes the most general environment that is acceptable, together with the component itself. In particular, the specification's next-state relation will be a disjunction of actions that describe the component's transitions and of actions that describe steps of the environment. Although this style works well in practice, it does not yield the most general specification of the overall system. Instead of constraining the environment, we may want to write a specification that allows the environment to behave arbitrarily, but that leaves the behavior of the component unconstrained after a step occurred that is disallowed by the assumptions on the environment. Abadi and Lamport considered ways of writing specifications that separate the environment assumptions E and the component guarantees C. The implication $E \Rightarrow C$ is a natural way for expressing such an assumption-guarantee specification, and this form is explored in [2]. However, the implication holds of behaviors in which first the component violates C, and later the environment violates E. In later work, Abadi and Lamport [3, 4] introduced the stronger operator $\stackrel{+}{\rightarrow}$ such that $E \stackrel{+}{\rightarrow} C$ requires that (the safety part of) C may be violated only if (the safety part of) E was violated strictly earlier.[4] Given two components described through assume-guarantee specifications, one may wish to prove that their composition refines a higher-level specification of the composed system. This is expressed in TLA as a proof obligation of the form

$$(E_1 \stackrel{+}{\rightarrow} C_1) \wedge (E_2 \stackrel{+}{\rightarrow} C_2) \;\Rightarrow\; (E \stackrel{+}{\rightarrow} C)$$

In order to establish the overall system guarantee C from the component guarantees C_1 and C_2, one will need to show that the environment assumptions E_1 and E_2 hold true. Now, the environment of each component consists of the overall environment (assumptions on which are expressed by E) and the other component, so one will want to use both E and C_2 for establishing E_1, and similarly for the other component. Despite the apparent circularity of this reasoning chain, Abadi and Lamport [4] give rules that support this approach in a sound

[4] $\stackrel{+}{\rightarrow}$ can actually be expressed in TLA, but it is useful to consider it as a separate operator.

way, based on a form of computational induction that is embodied in the definition of the $\stackrel{+}{\rightarrow}$ operator.

3. The Specification Language TLA$^+$

Lamport designed TLA as a variant of linear-time temporal logic that is particularly appropriate for specifying executions of fair state machines. Stuttering invariance is key for representing composition as conjunction of specifications, and refinement as validity of implication. Quantification over state variables adds significant expressive power and is useful notably for distinguishing the state visible at the interface of a component from the internal state used for its implementation.

However, TLA is not a full specification language: it does not fix the interpretation of elementary function and predicate symbols such as $+$ and $\in$. These symbols are provided by an underlying mathematical language based on first-order or higher-order logic. In particular, non-temporal proof obligations that arise during the verification of a system property or during a refinement proof should then be discharged using a (possibly mechanized) proof system associated with that host language.

3.1 Overall Design of TLA$^+$

Starting in the early 1990s and encouraged by successful experiments with TLP [21], a prototype proof system for TLA based on the Larch Prover, Lamport developed the specification language TLA$^+$. The language is described in the book *Specifying Systems* [43]; the Hyperbook and Lamport's video series [46, 48] provide excellent tutorial introductions, whereas the description of TLA$^+$ in [56] focuses on the semantics of the language.

TLA$^+$ is based on a variant of Zermelo-Fraenkel set theory with choice (ZFC) for describing the data manipulated by an algorithm. ZFC is widely accepted by mathematicians as the basis for formalizing mathematical theories. In order to emphasize the expressiveness of ZFC, Lamport shows that a formal definition of the Riemann integral can be given in just 15 lines starting from a standard module defining the real numbers with ordinary arithmetical operators [39]. When writing high-level specifications of algorithms, it is useful to model data in terms of concepts such as sets and functions rather than using low-level data types provided by programming languages and their libraries. In this respect, TLA$^+$ adopts a similar approach as the specification languages Z and (Event-)B [5, 6, 64]. However, the latter languages impose a typing discipline on set theory, whereas TLA$^+$ is untyped. Again, Lamport follows classical mathematical practice and, for example, considers that the set $\{2,4,6,\ldots\}$ of positive even numbers can be viewed as a type just like the set of all integers. He maintains that imposing a decidable type system on a specification language leads to unacceptable restrictions of the expressiveness of that language. Also, embedding partial operations in a typed language often leads to objectionable choices. For example, declaring integer division as a binary operation with integer arguments and result asserts that division by 0 returns an in-

teger, whereas implementations naturally raise an exception.[5] Although TLA$^+$ is untyped and handles partial operators by underspecification [24], this does not preclude tools for TLA$^+$ to reconstruct types when this is convenient for their analyses. An article by Lamport and Paulson [50] contains an interesting discussion of these questions.

3.2 A Glimpse of TLA$^+$

TLA$^+$ specifications are organized in *modules*. A module can extend other modules; semantically, this is equivalent to copying the content of the extended modules (with duplicates removed) into the extending module.

A module may declare constant and variable parameters. Any symbol that occurs in an expression in the module must be either a built-in symbol of TLA$^+$, a parameter in the context in which the expression appears, or a symbol that was previously defined or declared.

Modules may assert properties of constant parameters in the form of assumptions or axioms, both of which express hypotheses of the module.[6] Modules may also state theorems that can be proved using TLAPS, the TLA$^+$ Proof System.

The bulk of the contents of a typical TLA$^+$ module consists of definitions of operators, used to build up more complex expressions. An operator may take zero or more arguments, including operator arguments (whose arity must be specified). For example, the definition

$$\mathit{Symmetric}(R(_,_), S) \triangleq \forall x, y \in S : R(x, y) \equiv R(y, x)$$

introduces an operator characterizing a symmetric binary relation R over a set S. Besides the ordinary operators of first-order set theory, TLA$^+$ also borrows a few constructions from programming languages, such as conditional expressions (including n-ary case distinctions) and local definitions introduced through LET-bindings.

A module containing a system specification will usually define operators corresponding to the initial condition, the next-state relation, the overall specification, and properties to be verified. As a concrete example, Figure 2 contains a TLA$^+$ specification of a FIFO queue. It extends the library module *Sequences*, which defines the set $\mathit{Seq}(S)$ of finite sequences that contain elements of S, as well as operations such as *Head* and *Tail* to access the first element and the remaining elements of a sequence. Module *FIFO* then declares two constants *Data* and *null* that correspond to the data to be stored in the queue and a "null" element representing the absence of data. The module also declares the variables *in*, *out*, and *q* that are used for specifying the state machine describing the behavior of the FIFO queue. Concretely, *in* and *out* represent the channels for data input and output, whereas *q* contains the current contents of the queue.

[5] Some proof assistants such as Isabelle/HOL go even further and define $n \operatorname{div} 0 = 0$, which is unlikely to hold in an implementation and may actually mask errors.

[6] TLC will verify that an assumption evaluates to true for the concrete values substituted for the module parameters, but it will not evaluate axioms.

MODULE *FIFO*

EXTENDS *Sequences*
CONSTANTS *Data*, *null*
ASSUME $null \notin Data$
VARIABLES in, q, out

$$TypeOK \triangleq in \in (Data \cup \{null\}) \land out \in (Data \cup \{null\}) \land q \in Seq(Data)$$

$$Init \triangleq in = null \land out = null \land q = \langle\rangle$$

$$\begin{aligned} Enq \triangleq\ & \land\ in' \in (Data \cup \{null\}) \setminus \{in\} \\ & \land\ q' = \text{IF } in' \in Data \text{ THEN } Append(q, in') \text{ ELSE } q \\ & \land\ out' = out \end{aligned}$$

$$\begin{aligned} Deq \triangleq\ & \land \lor\ q = \langle\rangle \land out' = null \land q' = q \\ & \quad\ \lor\ q \neq \langle\rangle \land out' = Head(q) \land q' = Tail(q) \\ & \land\ in' = in \end{aligned}$$

$$vars \triangleq \langle in, q, out\rangle$$

$$FIFO \triangleq Init \land \Box[Enq \lor Deq]_{vars} \land \mathrm{WF}_{vars}(Deq)$$

THEOREM $FIFOType \triangleq FIFO \Rightarrow \Box TypeOK$

THEOREM $InOut \triangleq FIFO \Rightarrow \Box[in' = in \lor out' = out]_{vars}$

THEOREM $Liveness \triangleq FIFO \Rightarrow \forall x \in Data : (in = x) \leadsto (out = x)$

Figure 2 TLA$^+$ specification of a FIFO queue.

Again, TLA$^+$ is untyped, and consequently one does not declare types for constant or variable parameters. Because TLA$^+$ is based on set theory, there is no need to assert that *Data* is a set. In fact, semantically all values are sets, although it is more useful to think of the elements of set $Data \cup \{null\}$, as well as of numbers or strings, as atomic values.

The second block of the module contains operator definitions.[7] The first operator corresponds to the (intended) type invariant of the specification. The definitions of the operators *Init*, *Enq*, and *Deq* introduce the initial condition and the enqueue and dequeue actions of the queue. The initial predicate simply requires that the input and output channels contain the *null* value and that the queue is empty. The enqueue action models a change of value at the input channel. If a data value is sent over the channel, it is appended to the current contents of the queue. Otherwise (i.e., if a null value appears on the channel), the queue remains unchanged. The output value of the queue remains unchanged during an enqueue operation.

[7] The horizontal bars are decorative and have no semantic meaning.

MODULE *TwoFIFO*

EXTENDS *Sequences*
CONSTANTS $Data, null$
ASSUME $null \notin Data$
VARIABLES $in, q1, mid, q2, out$

$Left \triangleq$ INSTANCE *FIFO* WITH $q \leftarrow q1, out \leftarrow mid$
$Right \triangleq$ INSTANCE *FIFO* WITH $in \leftarrow mid, q \leftarrow q2$
$Conc \triangleq$ INSTANCE *FIFO* WITH $q \leftarrow q2 \circ q1$
$Interleave \triangleq in' = in \vee out' = out$
$TwoFIFO \triangleq Left!FIFO \wedge Right!FIFO \wedge \Box[Interleave]_{in,out}$

THEOREM $Implementation \triangleq TwoFIFO \Rightarrow Conc!FIFO$

Figure 3 Composition of two FIFO queues.

Symmetrically, a dequeue operation does not modify the input channel. It puts *null* on the output channel and leaves the queue unchanged if the queue is empty and otherwise sends $Head(q)$ on the output channel and removes that element from the queue. Formula *FIFO* represents the overall queue specification. Its next-state relation is the disjunction of the enqueue and dequeue actions. The fairness conjunct requires dequeue actions to happen eventually so that the queue must eventually output the values that it stores.

The third block of the module asserts three theorems. The first theorem states that the type correctness predicate holds throughout any execution. The second theorem states that the values of the input and output channels never change simultaneously, and the third theorem asserts that every data value that appears on the input channel will eventually be output by the queue. We will see in Section 5 how these properties can be verified using the TLA$^+$ tools.

3.3 Composing Modules

Beyond module extension, TLA$^+$ offers *instantiation* as a second way for composing modules. An instance conceptually creates a copy of the original module in which the constant and variable parameters can be instantiated by (constant and state) expressions. This construction is useful for composing specifications. For example, module *TwoFIFO* of Figure 3 declares the composition of two FIFO queues by creating two instances *Left* and *Right* of the *FIFO* module of Figure 2. Instance *Left* uses variables *q1* and *mid* for the internal queue and the output channel; all other parameters are instantiated by the parameters of the same name declared in module *TwoFIFO*. Similarly, instance *Right* uses *mid* and *q2* for *in* and *q*. The conjunction of the two instantiated specifications *Left*!*FIFO* and *Right*!*FIFO* describes the composition

of two FIFO queues that communicate through the shared communication channel *mid*. We would like to assert that the conjunction of these specifications behaves like a FIFO whose internal queue is given by the concatenation of the internal queues of the two components. The instance *Conc* represents this "longer" FIFO queue with input channel *in* and output channel *out*, and we would therefore like to assert the theorem

$$Left!FIFO \wedge Right!FIFO \Rightarrow Conc!FIFO.$$

However, this implication is not valid: the conjunction on the left-hand side allows an enqueue action of the left FIFO queue and a dequeue action of the right FIFO queue to happen simultaneously (observe that both actions leave the shared variable *mid* unchanged). In this case, the values of the channels *in* and *out* change simultaneously, and this is not allowed by specification *Conc*!*FIFO*. Indeed, we wrote our specification according to an interleaving model, where enqueue and dequeue actions do not happen simultaneously. We have to explicitly enforce this interleaving assumption for the composition of the two FIFO queues, as expressed in the specification *TwoFIFO*, in order to obtain the theorem *Implementation* stated at the end of the module.

4. PlusCal: an Algorithm Language

Due to its expressiveness and high level of abstraction, TLA$^+$ is a very powerful language for specifying high-level designs of concurrent algorithms and systems. However, it may feel unfamiliar to programmers, in particular due to the syntax based on mathematical logic and to the absence of explicit control flow in the specification of systems and algorithms. Lamport designed PlusCal [44] as a language for describing algorithms that combines the look and feel of pseudo-code and the precision of TLA$^+$. It uses primitives that are familiar from imperative programming languages for describing the control flow of an algorithm. In contrast, the data manipulated by the algorithm is represented by TLA$^+$ expressions, letting the algorithm designer benefit from the abstraction afforded by set theory without being constrained by concerns of how to concretely implement data structures.

A PlusCal algorithm is embedded as a comment within a TLA$^+$ module and has access to all operators available at that point of the module (whether defined in extended modules or locally). The PlusCal translator converts the algorithm into a TLA$^+$ specification that is inserted into the module. The user then states properties in terms of the TLA$^+$ translation and verifies them just as for any other TLA$^+$ specification, using the tools described in Section 5.

A PlusCal algorithm may declare several process templates for parallel execution, and each template can have a fixed number of instances.[8] Variables can be declared globally, representing shared state (including the communication network of a distributed system), or locally for each process. The control flow of each process is described using standard

[8] PlusCal does not support dynamic spawning of processes.

primitives of imperative languages (sequencing, conditional statements, loops, procedure calls etc.). In addition, the two primitives

either $\{ \quad \ldots \quad \}$ and **with** $(x \in S)$ $\{ \quad \ldots \quad \}$
or $\{ \quad \ldots \quad \}$

are available for modeling non-determinism. The first construct can be used to introduce a fixed number of alternatives, the second one executes a block of code for some value that is chosen non-deterministically from the set S. Synchronization among processes is modeled using the instruction **await** P that blocks until the predicate P becomes true.

An important aspect for the specification of concurrent algorithms is to identify the "grain of atomicity", i.e., which blocks of statements are intended to be executed without interference from other processes. Rather than imposing an arbitrary fixed level of atomicity, PlusCal uses labels to identify yield points at which processes may be interrupted. A group of statements between two labels is assumed to be executed atomically. This allows the designer to choose the degree of atomicity appropriate for the specification and to compare algorithms described at different degrees of atomicity.[9] In order to ensure liveness of PlusCal algorithms, fairness conditions may be attached to labels or to entire processes. These ensure that the group of statements following the label (respectively, the entire process) will eventually execute if it is sufficiently often enabled.

As an example, a PlusCal algorithm modeling a simple producer-consumer system appears in Figure 4. It declares two process templates for the producer and the consumer, each of which is instantiated once for process identities "*p*" and "*c*". The two processes communicate using a shared FIFO queue of bounded capacity *maxCapacity*. Each process has an infinite loop: the producer repeatedly adds new data to the queue, while the consumer retrieves the data from the queue. By declaring a (weak) fairness condition for the consumer process, we ensure that every data item that is present in the queue will eventually be consumed. In this specification of the algorithm, the bodies of the while loops execute atomically; non-atomic execution would be modeled by inserting additional labels. The operations *Len*, *Append*, *Head*, and *Tail* that appear in the presentation of the algorithm are defined in the standard module *Sequences* that is extended by module *ProducerConsumer*.

Invoking the PlusCal translator on module *ProducerConsumer* generates a TLA$^+$ specification corresponding to the algorithm. In particular, the translator generates declarations of TLA$^+$ variables corresponding to the global and local variables of the PlusCal algorithm, and it derives the initial condition from the initializations of the PlusCal variables.[10] The essential step of the translation is to generate a TLA$^+$ action for each group of statements between

[9] Some rules govern where labels must or cannot be placed, essentially to ensure that PlusCal algorithms are easy to translate into TLA$^+$ specifications.

[10] For PlusCal variables that are not initialized, such as *rcvd* in our example, the translator adds a default initialization, which is necessary for model checking using TLC.

```
──────────────── MODULE ProducerConsumer ────────────────
EXTENDS Naturals, Sequences
CONSTANTS Data, maxCapacity
ASSUME maxCapacity ∈ Nat \ {0}
(*
–algorithm ProducerConsumer {
  variable q = ⟨⟩;
  define {
      nonempty ≜ Len(q) > 0
      nonfull  ≜ Len(q) < maxCapacity
  }
  process (Producer = "p") {
  p : while (TRUE) {
        await nonfull;
        with (item ∈ Data) {
           q := Append(q, item)
  }  }  }
  fair process (Consumer = "c")
     variable rcvd; {
  c : while (TRUE) {
        await nonempty;
        rcvd := Head(q);
        q := Tail(q)
  }  }
}
*)
```

Figure 4 A specification of a producer-consumer system in PlusCal.

two consecutive labels. For example, the loop body of the producer process of Figure 4 is represented by the action

$$
\begin{aligned}
Producer \triangleq\ & \land\ nonfull \\
& \land\ \exists item \in Data : q' = Append(q, item) \\
& \land\ rcvd' = rcvd
\end{aligned}
$$

For more complicated algorithms, the translator adds a variable *pc* that represents the current point of control of each process. When a process type has several instances, their local variables are represented using arrays (i.e., TLA$^+$ functions).

Because the translation from PlusCal to TLA$^+$ is fairly direct, the generated TLA$^+$ specification is usually quite readable. This is important because correctness properties of the algorithm are written in TLA$^+$ rather than in PlusCal. For our producer-consumer example, we may want to verify the invariant *BoundedQueue* and the temporal property *Liveness* defined as

$$\begin{array}{l} BoundedQueue \triangleq q \in Seq(Data) \land Len(q) \leq maxCapacity \\ Liveness \triangleq \forall d \in Data : (\exists i \in 1 \,..\, Len(q) : q[i] = d) \leadsto rcvd = d \end{array}$$

that express type correctness and eventual reception of every data item contained in the queue.

5. Tool Support

5.1 The model checker TLC

Lamport originally designed TLA$^+$ as a precise and expressive language for specifying algorithms and for (deductively) reasoning about their properties. It was used in the second half of the 1990s by hardware designers at Digital Equipment Corporation, in particular for describing cache coherence protocols of multiprocessors [30]. They wrote rigorous, informal proofs for key invariants maintained by these protocols. Yuan Yu then recognized that it was possible to support this type of reasoning using model checking. Lamport reports that originally he was very skeptical of this idea. Input languages for model checkers such as Spin [28], SMV [54] or Murphi [19] are based on low-level primitives carefully chosen to support efficient verification of finite-state systems, whereas TLA$^+$ uses the full power of ZFC set theory and is intended for modeling systems of arbitrary size. It is not possible in general to systematically enumerate all behaviors that satisfy a given TLA$^+$ formula.

TLC, the TLA$^+$ model checker, accepts a subset of TLA$^+$ specifications written in standard form (1). It is an explicit-state model checker, intended for verifying finite instances of specifications. In addition to the specification, the user has to provide the model checker with a *model* that describes a finite instance by fixing specific values for constant parameters. For the queue specification of Figure 2, one could for example fix parameter values $Data = \{1,2,3\}$ and $null = 0$. TLC then interprets the specification, restricted to this model, by decomposing the next-state relation into disjuncts (bounded existential quantification over finite sets is expanded into an explicit disjunction) and evaluating each disjunct from left to right. The first occurrence of a primed variable v' is expected to be of the form $v' = e$ or $v' \in e$ for an expression e that TLC can evaluate; in the second form, e must evaluate to a finite set. The first form is interpreted as an assignment of (the value denoted by) e to v in the successor state. The second form leads to the generation of one successor state per element of e, with v assigned to that element. Subsequent occurrences of v' are interpreted by the value assigned to v in the successor state in this way. For example, the conjunct

$$in' \in (Data \cup \{null\}) \setminus \{in\}$$

of the action *Enq* of Figure 2 generates one successor state for each element of $Data \cup \{null\}$, except for the current value of *in*. The occurrences of in' in the second conjunct of *Enq* then refer to the value chosen for that successor state. The initial predicate is evaluated in a similar way. TLC will abort with an error message if the initial predicate or some subaction of the next-state relation does not assign a value to some of the variables declared in the module.

When evaluating set-theoretic expressions, TLC will generally enumerate the elements, but it will apply some optimizations. For example, in evaluating the predicate $e \in Nat$ that may occur in a typing invariant, TLC will simply check if (the value denoted by) e is a natural number. TLC disallows unbounded quantification, and it signals an error when it would have to enumerate an infinite set.

Using the strategy outlined above, TLC enumerates all reachable states in a breadth-first manner, and it checks the invariant predicates provided by the user during this state enumeration. When an invariant evaluates to false, the run is aborted and a counter-example is displayed. (Due to breadth-first search, that counter-example will be of minimal length.) Liveness properties are evaluated over the state graph computed during state enumeration, based on the tableau algorithm of [53]. TLC parallelizes state enumeration on multi-core machines and provides a distributed implementation for running in a cluster or cloud environment. States may be stored to disk so that state exploration is not memory bound, and TLC regularly performs checkpoints so that model checking can be resumed in case of a crash. In order to limit the explored state space, the user can impose state constraints. For example, the FIFO queue of Figure 2 generates an unbounded state space even for a fixed finite set *Data* because the length of the queue can grow without bound, and the user can choose not to explore successors of states in which $Len(q)$ exceeds some fixed value. TLC also implements symmetry reduction in order to explore only a quotient of the state space with respect to an equivalence relation. In the queue example, the user may choose to declare *Data* (or more precisely, the set that the parameter *Data* is instantiated with in the concrete model) as a symmetry set because all operations are insensitive to particular values in that set.

Although TLC imposes certain restrictions on the specifications that it can check, most specifications that are written in practice adhere to those restrictions or can easily be rewritten so that they do. (The fact that TLC has been the main analysis tool for TLA$^+$ specifications has also contributed to disciplining users so that they respect those restrictions.) In particular, TLA$^+$ specifications obtained from translating PlusCal algorithms can be checked using TLC. The different properties asserted in the modules of the previous sections can be verified by TLC for concrete instances of parameters, including the theorems of Figures 2 and 3, as well as the properties of the producer-consumer system given at the end of Section 4.[11]

Like most model checkers, TLC is most useful when a counter-example to a putative property is discovered. A positive verdict only means that the checked properties hold for the

[11] The specification *TwoFIFO* of Figure 3 needs to be rewritten in standard form so that TLC can verify it.

particular model that TLC checked, and it requires sound engineering judgement to determine if this gives enough confidence in the correctness of the properties for arbitrary instances of the specification.

5.2 The TLA$^+$ Proof System

TLAPS, the TLA$^+$ Proof System [17], is a proof assistant for checking proofs written in TLA$^+$. For this purpose, TLA$^+$ was extended to include a language for writing hierarchical proofs based on a format that Lamport had proposed earlier [41] for writing rigorous pencil-and-paper proofs. For example, the proof of type correctness for the FIFO queue of Figure 2 can be written as follows.

$$
\begin{array}{l}
\text{THEOREM } \mathit{FIFOType} \triangleq \mathit{FIFO} \Rightarrow \Box \mathit{TypeOK} \\
\langle 1 \rangle 1.\ \mathit{Init} \Rightarrow \mathit{TypeOK} \\
\quad \text{BY DEF } \mathit{Init}, \mathit{TypeOK} \\
\langle 1 \rangle 2.\ \mathit{TypeOK} \wedge [\mathit{Enq} \vee \mathit{Deq}]_{\mathit{vars}} \Rightarrow \mathit{TypeOK}' \\
\quad \text{BY DEF } \mathit{Enq}, \mathit{Deq}, \mathit{vars}, \mathit{TypeOK} \\
\langle 1 \rangle 3.\ \text{QED} \\
\quad \text{BY } \langle 1 \rangle 1,\ \langle 1 \rangle 2,\ \text{PTL DEF } \mathit{FIFO}
\end{array}
$$

Following a standard pattern for invariance proofs (cf. also rule INV1 of Section 2.2), the first two steps of the proof establish that the initial predicate of the *FIFO* specification implies predicate *TypeOK*, and that the predicate is preserved by every step allowed by the next-state relation. The third step concludes the proof of the theorem. The justifications for each step are indicated following the keyword BY. For the first two steps, it suffices to expand the relevant definitions and then apply built-in automatic proof back-ends that mechanize standard mathematical reasoning. The justification of the third step uses the assertions of the two preceding steps and propositional temporal logic; it also expands the definition of $FIFO$ in order to expose its initial and next-state predicates. Because the proof is so simple, we only needed one level of proof: all step names have the form $\langle 1 \rangle n$. The steps of more complicated proofs can be decomposed into lower-level proof steps until TLAPS can prove the leaf steps of the proof automatically.

Figure 5 schematizes the architecture of TLAPS. The central component is the proof manager that interprets the proof language, maintains the context of each proof step (i.e., the visible identifiers, assumptions, and definitions) and computes the corresponding proof obligations. In a non-temporal step (such as the first two steps in the above example), primes are pushed inside complex expressions, and then primed symbols are replaced by fresh identifiers. Similarly, any temporal expressions appearing in the context are abstracted by fresh predicate symbols. Similarly, in a temporal step (such as the QED step above), any first-order formulas are abstracted by propositional variables. This transformation is called coalescing [20]; it is necessary so that back-end provers see the proof obligation either as

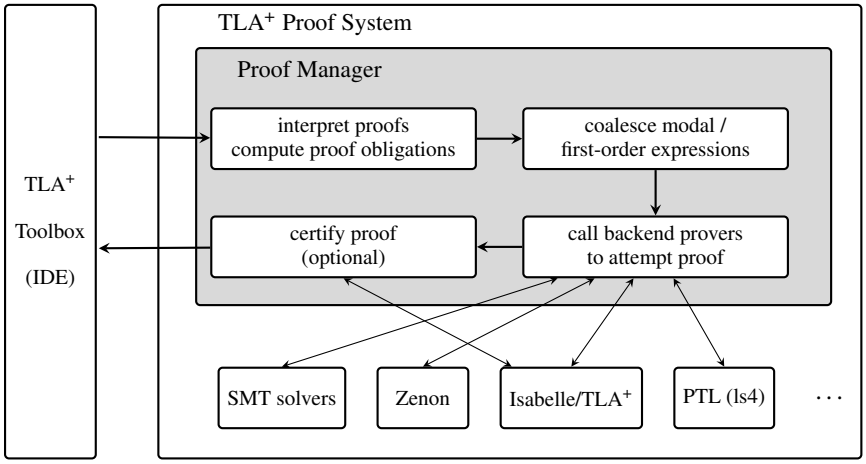

Figure 5 Architecture of the TLA$^+$ Proof System.

a standard formula of mathematical set theory or as a propositional temporal logic formula. The proof manager then calls back-end provers to attempt and prove the proof obligation. Currently, TLAPS supports SMT solvers (via a translation to the SMT-LIB2 language [12]), the tableau prover Zenon [14], and an encoding of TLA$^+$'s mathematical set theory as an object logic in the logical framework Isabelle [60] for proving non-temporal steps. It also comes with a decision procedure for propositional temporal logic [65] for proving temporal proof obligations. The architecture is open for supporting additional back-end provers through suitable translations of proof obligations into their input language. For increased confidence in the correctness of TLAPS proofs, when a back-end prover finds a proof it may return a justification to the proof manager for checking by the trusted kernel of Isabelle. This certification step is optional and currently only available for proofs found by Zenon.

TLAPS is currently restricted to proving safety properties. The planned extension to liveness properties requires support for handling ENABLED predicates and for first-order temporal logic reasoning, for example for mechanizing the Lattice rule of Section 2.2. TLAPS has been used for verifying several distributed algorithms, including variants of Paxos [16, 45] and a version of the Pastry distributed hash table [10].

5.3 The TLA$^+$ Toolbox

Editing and analyzing TLA$^+$ specifications is facilitated by the TLA$^+$ Toolbox, an Eclipse application that provides an IDE (integrated development environment) for TLA$^+$. It provides support for editing TLA$^+$ specifications and proofs, such as looking up operator definitions, properly indenting TLA$^+$ specifications, renumbering proof steps, and hiding subproofs that are irrelevant for the current branch. The Toolbox is integrated with the TLA$^+$ tools, including

SANY, the TLA$^+$ syntactic and semantic analyzer, the TLATEX pretty printer, TLC and TLAPS. In particular, the user interface to TLC provided by the Toolbox greatly simplifies the definition of finite-state models to be verified, the analysis of counter-examples, and the evaluation of TLA$^+$ expressions.

All TLA$^+$ tools are released as open-source software under permissive licences for use in industry or academia.

6. Impact

The concepts that Lamport introduced for the formal specification and verification of algorithms have deeply influenced the research community. The notions of safety, liveness, and fairness are universally recognized for their fundamental importance. The concept of stuttering invariance is valuable in contexts other than those strictly related to refinement and composition; in particular, it plays an important role for partial-order reductions used for model checking distributed systems [23, 66]. The idea of writing system specifications in terms of state machines is widely accepted [6, 25]. The specification language TLA$^+$ is taught at universities around the world, and PlusCal is starting to be used as a vehicle for teaching courses on distributed algorithms.

The historically first significant use of TLA$^+$ in industry was for specifying and verifying cache coherence protocols by a group of hardware engineers that designed Digital Equipment Corporation's Alpha processors [30]. Members of that group subsequently moved to Intel and continued to use TLA$^+$, although little is publicly known about the impact of that work. Work at Microsoft using TLA$^+$ started around 2003 with the specification of the Web Services Atomic Transaction protocol [29]. This experience was considered successful, and engineers at Microsoft continued to use TLA$^+$. Reportedly, use of TLA$^+$ contributed to identifying a serious error in the XBox 360 memory system that would have been difficult to debug using conventional techniques. The Farsite project [13] at Microsoft Research developed a scalable, serverless, and location-transparent distributed file system that could tolerate nodes being unavailable, as well as malicious participants. The designers used TLA$^+$ for specifying the distributed directory service and refined a centralized functional specification into the formal description of a distributed protocol. They report that the main benefit of using formal specification and verification was to understand the invariants that the system must maintain through different levels of refinement. They consider that it would have been far more costly to iterate through several designs at the implementation level where aspects related to the distributed protocol would have been mixed with low-level coding details. In contrast, they found that developing an implementation from the protocol specification was rather straightforward because only sequential code had to be written, without a need for thinking about aspects related to distributed execution. In the later IronFleet project [26, 27], researchers at Microsoft pushed this idea even further. Combining a TLA-style approach to state machine specification and refinement with a Floyd-Hoare style of reasoning about imperative programs

provided by Dafny [52], they obtained a mechanized framework for designing, implementing, and verifying distributed systems from high-level (centralized) specifications to distributed protocols and further to executable code that exhibited competitive performance. Based on an embedding of TLA and its proof rules in Dafny, they could prove not only safety, but even liveness properties in a unified framework. The approach was used to develop a replicated state machine library and a sharded key-value store.

An interesting account of the use of TLA$^+$ in industry was provided by a group around Chris Newcombe working at Amazon Web Services [58]. They reported that not only have TLA$^+$ specifications contributed to finding subtle bugs in high-level designs of distributed protocols, but that the understanding and confidence obtained from formal specification and verification allowed them to make aggressive performance optimizations without sacrificing correctness. Several other companies developing web and cloud services, including the groups working on Azure at Microsoft, actively use TLA$^+$ and TLC for describing and verifying the protocols they design. The TLA$^+$ Google group[12] and regular in-person community meetings provide forums for the members of the TLA$^+$ community to exchange and help each other in case of problems.

TLA$^+$ is intended as a formalism for modeling and verifying high-level designs of algorithms and systems. Doing so does not prevent coding errors from creeping into implementations of verified algorithms: such errors can be caught using techniques of program verification. However, the implementation of a buggy design is virtually guaranteed to contain the design errors, and finding and fixing these issues at the level of executable code is much more difficult and costly than to do so at an early stage of development, using specifications written at the appropriate level of abstraction.

[12] https://groups.google.com/forum/#!forum/tlaplus

Bibliography

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82(2):253–284, 1991.

[2] M. Abadi and L. Lamport. Composing specifications. *ACM Trans. Program. Lang. Syst.*, 15(1):73–132, 1993.

[3] M. Abadi and L. Lamport. Open systems in TLA. In J. H. Anderson, D. Peleg, and E. Borowsky, editors, *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing, Los Angeles, California, USA, August 14-17, 1994*, pages 81–90. ACM, 1994.

[4] M. Abadi and L. Lamport. Conjoining specifications. *ACM Trans. Program. Lang. Syst.*, 17(3):507–534, 1995.

[5] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[6] J.-R. Abrial. *Modeling in Event-B*. Cambridge University Press, 2010.

[7] B. Alpern and F. B. Schneider. Defining liveness. *Inf. Proc. Letters*, 21(4):181–185, Oct. 1985.

[8] K. R. Apt. Ten years of Hoare's logic: A survey – part 1. *ACM Trans. Prog. Lang. and Syst.*, 3(4):431–483, 1981.

[9] P. C. Attie, N. Francez, and O. Grumberg. Fairness and hyperfairness in multi-party interactions. *Distributed Computing*, 6(4):245–254, 1993.

[10] N. Azmy, S. Merz, and C. Weidenbach. A machine-checked correctness proof for Pastry. *Sci. Comput. Program.*, 158:64–80, 2018.

[11] R.-J. Back. On correct refinement of programs. *J. Comp. and System Sci.*, 23(1):49–68, 1981.

[12] C. Barrett and C. Tinelli. Satisfiability modulo theories. In E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, editors, *Handbook of Model Checking*, pages 305–343. Springer, 2018.

[13] W. J. Bolosky, J. R. Douceur, and J. Howell. The Farsite project: a retrospective. *Operating Systems Review*, 41(2):17–26, 2007.

[14] R. Bonichon, D. Delahaye, and D. Doligez. Zenon: An extensible automated theorem prover producing checkable proofs. In N. Dershowitz and A. Voronkov, editors, *14th Intl. Conf. Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 4790 of *Lecture Notes in Computer Science*, pages 151–165, Yerevan, Armenia, 2007. Springer.

[15] R. M. Burstall. Program proving as hand simulation with a little induction. In *Information Processing*, pages 308–312. North-Holland Pub. Co., 1974.

[16] S. Chand, Y. A. Liu, and S. D. Stoller. Formal verification of Multi-Paxos for distributed consensus. In J. S. Fitzgerald, C. L. Heitmeyer, S. Gnesi, and A. Philippou, editors, *21st Intl. Symp. Formal Methods (FM 2016)*, volume 9995 of *Lecture Notes in Computer Science*, pages 119–136, Limassol, Cyprus, 2016.

[17] D. Cousineau, D. Doligez, L. Lamport, S. Merz, D. Ricketts, and H. Vanzetto. TLA$^+$ proofs. In D. Giannakopoulou and D. Méry, editors, *FM 2012: Formal Methods - 18th International*

Symposium, Paris, France, August 27-31, 2012. Proceedings, volume 7436 of *Lecture Notes in Computer Science*, pages 147–154. Springer, 2012.

[18] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[19] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *IEEE Intl. Conf. Computer Design: VLSI in Computers and Processors*, pages 522–525. IEEE Computer Society, 1992.

[20] D. Doligez, J. Kriener, L. Lamport, T. Libal, and S. Merz. Coalescing: Syntactic abstraction for reasoning in first-order modal logics. In C. Benzmüller and J. Otten, editors, *Automated Reasoning in Quantified Non-Classical Logics, ARQNL@IJCAR 2014, Vienna, Austria, July 23, 2014*, volume 33 of *EPiC Series in Computing*, pages 1–16. EasyChair, 2014.

[21] U. Engberg, P. Grønning, and L. Lamport. Mechanical verification of concurrent systems with TLA. In G. von Bochmann and D. K. Probst, editors, *Computer Aided Verification, Fourth International Workshop, CAV '92, Montreal, Canada, June 29 - July 1, 1992, Proceedings*, volume 663 of *Lecture Notes in Computer Science*, pages 44–55. Springer, 1992.

[22] R. W. Floyd. Assigning meanings to programs. In *Proc. Symposium on Applied Mathematics*, volume 19, pages 19–32. American Mathematical Society, 1967.

[23] P. Godefroid and P. Wolper. A partial approach to model checking. *Inf. Comput.*, 110(2):305–326, 1994.

[24] D. Gries and F. B. Schneider. Avoiding the undefined by underspecification. In J. van Leeuwen, editor, *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*, pages 366–373. Springer, 1995.

[25] Y. Gurevich. Evolving algebras 1993: Lipari guide. In E. Börger, editor, *Specification and validation methods*, pages 9–36. Oxford University Press, 1995.

[26] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. T. V. Setty, and B. Zill. Ironfleet: proving practical distributed systems correct. In E. L. Miller and S. Hand, editors, *Proc. 25th Symp. Operating Systems Principles, SOSP 2015*, pages 1–17, Monterey, CA, 2015. ACM.

[27] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. T. V. Setty, and B. Zill. Ironfleet: proving safety and liveness of practical distributed systems. *Commun. ACM*, 60(7):83–92, 2017.

[28] G. J. Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2003.

[29] J. E. Johnson, D. E. Langworthy, L. Lamport, and F. H. Vogt. Formal specification of a web services protocol. *J. Log. Algebr. Program.*, 70(1):34–52, 2007.

[30] R. Joshi, L. Lamport, J. Matthews, S. Tasiran, M. R. Tuttle, and Y. Yu. Checking cache-coherence protocols with TLA$^+$. *Formal Methods in System Design*, 22(2):125–131, 2003.

[31] F. Kröger. LAR: A logic of algorithmic reasoning. *Acta Informatica*, 8:243–266, 1977.

[32] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3(2):125–143, 1977.

[33] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[34] L. Lamport. A new approach to proving the correctness of multiprocess programs. *ACM Trans. Program. Lang. Syst.*, 1(1):84–97, 1979.

[35] L. Lamport. "Sometime" is sometimes "not never" – on the temporal logic of programs. In P. W. Abrahams, R. J. Lipton, and S. R. Bourne, editors, *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages, Las Vegas, Nevada, USA, January 1980*, pages 174–185. ACM Press, 1980.

[36] L. Lamport. What good is temporal logic? In *IFIP Congress*, pages 657–668, 1983.

[37] L. Lamport. *win* and *sin*: Predicate transformers for concurrency. *ACM Trans. Program. Lang. Syst.*, 12(3):396–428, 1990.

[38] L. Lamport. The Temporal Logic of Actions. Research Report 79, Digital Equipment Corporation Systems Research Center, December 1991.

[39] L. Lamport. Hybrid systems in TLA$^+$. In R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, editors, *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*, pages 77–102. Springer, 1992.

[40] L. Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, 1994.

[41] L. Lamport. How to write a proof. *American Mathematical Monthly*, 102(7):600–608, 1995.

[42] L. Lamport. Fairness and hyperfairness. *Distributed Computing*, 13(4):239–245, 2000.

[43] L. Lamport. *Specifying Systems: The TLA$^+$ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[44] L. Lamport. The PlusCal algorithm language. In M. Leucker and C. Morgan, editors, *Theoretical Aspects of Computing - ICTAC 2009, 6th International Colloquium, Kuala Lumpur, Malaysia, August 16-20, 2009. Proceedings*, volume 5684 of *Lecture Notes in Computer Science*, pages 36–60. Springer, 2009.

[45] L. Lamport. Byzantizing Paxos by refinement. In D. Peleg, editor, *Distributed Computing - 25th International Symposium, DISC 2011, Rome, Italy, September 20-22, 2011. Proceedings*, volume 6950 of *Lecture Notes in Computer Science*, pages 211–224. Springer, 2011.

[46] L. Lamport. The TLA$^+$ hyperbook, 2015.

[47] L. Lamport. My writings. https://lamport.azurewebsites.net/pubs/pubs.html, 2018. Retrieved June 28, 2018.

[48] L. Lamport. The TLA$^+$ video course, 2018.

[49] L. Lamport and S. Merz. Auxiliary variables in TLA$^+$. *CoRR*, abs/1703.05121, 2017.

[50] L. Lamport and L. C. Paulson. Should your specification language be typed. *ACM Trans. Program. Lang. Syst.*, 21(3):502–526, 1999.

[51] L. Lamport, R. E. Shostak, and M. C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[52] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In E. M. Clarke and A. Voronkov, editors, *16th Intl. Conf. Logic for Programming, Artificial Intelligence, and Reasoning*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370, Dakar, Senegal, 2010. Springer.

[53] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In M. S. V. Deusen, Z. Galil, and B. K. Reid, editors, *Proc. Twelfth Ann. ACM Symp. Princ. Prog. Lang. (POPL 1985)*, pages 97–107, New Orleans, LA, U.S.A., 1985. ACM Press.

[54] K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[55] S. Merz. A more complete TLA. In J. Wing, J. Woodcock, and J. Davies, editors, *FM'99: World Congress on Formal Methods*, volume 1709 of *Lecture Notes in Computer Science*, pages 1226–1244, Toulouse, France, 1999. Springer.

[56] S. Merz. The specification language TLA$^+$. In D. Bjørner and M. C. Henson, editors, *Logics of Specification Languages*, Monographs in Theoretical Computer Science, pages 401–451. Springer, Berlin-Heidelberg, 2008.

[57] C. Morgan. *Programming from specifications*. Prentice Hall, 1990.

[58] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff. How Amazon Web Services uses formal methods. *CACM*, 58(4):66–73, 2015.

[59] S. S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Trans. Program. Lang. Syst.*, 4(3):455–495, 1982.

[60] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer Verlag, Berlin, Heidelberg, 1994. See also the Isabelle home page at http://isabelle.in.tum.de/.

[61] A. Pnueli. The temporal logic of programs. In *Proc. 18th Annual Symposium on the Foundations of Computer Science*, pages 46–57. IEEE, 1977.

[62] A. N. Prior. *Past, Present and Future*. Clarendon Press, Oxford, U.K., 1967.

[63] R. L. Schwartz, P. M. Melliar-Smith, and F. H. Vogt. An interval-based temporal logic. In E. M. Clarke and D. Kozen, editors, *Proc. Logics of Programs*, volume 164 of *Lecture Notes in Computer Science*, pages 443–457, Pittsburgh, PA, U.S.A., 1984. Springer.

[64] J. M. Spivey. *The Z Notation: A reference manual*. International Series in Computer Science. Prentice Hall, 2nd edition edition, 1992.

[65] M. Suda and C. Weidenbach. A PLTL-prover based on labelled superposition with partial model guidance. In B. Gramlich, D. Miller, and U. Sattler, editors, *6th Intl. Joint Conf. Automated Reasoning (IJCAR 2012)*, volume 7364 of *LNCS*, pages 537–543, Manchester, UK, 2012. Springer.

[66] A. Valmari. A stubborn attack on state explosion. In *2nd Intl. Wsh. Computer Aided Verification*, volume 531 of *LNCS*, pages 156–165, Rutgers, June 1990. Springer.