Computing and Informatics, Vol. 20, 2001, 1–27, V 2003-May-9

ON THE LOGIC OF TLA$^+$

Stephan MERZ

INRIA Lorraine, LORIA, Nancy
e-mail: `Stephan.Merz@loria.fr`

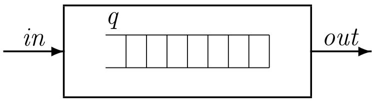
Abstract. TLA$^+$ is a language intended for the high-level specification of reactive, distributed, and in particular asynchronous systems. Combining the linear-time temporal logic TLA and classical set-theory, it provides an expressive specification formalism and supports assertional verification.

1 A TASTE OF TLA$^+$

The specification language TLA$^+$ has been introduced by Leslie Lamport [20] for the description of reactive and distributed, especially asynchronous systems. In this paper, I describe the semantical base of TLA$^+$, which combines the linear-time temporal logic TLA and Zermelo-Fränkel set theory. My intention is not to define a new or extended formalism nor to explain the use of TLA$^+$ in practice. Lamport's original work covers much more material than this paper. In particular, his recent book [25] presents a tutorial introduction to writing specifications in TLA$^+$, formally defines the language of TLA$^+$, and describes the tools that support it. In contrast, this presentation of TLA$^+$ emphasizes the mathematical machinery underlying TLA$^+$, explaining Lamport's choices from a logical perspective. It is my hope that it will find some use for purposes such as comparing specification formalisms or for constructing new tools to support system development in TLA$^+$.

Before we begin exploring the semantics of TLA$^+$, let us have a look at a simple example that introduces the typical structure of a TLA$^+$ specification. The TLA$^+$ module *SyncQueueInternal*, shown in figure 1(b), describes an unbounded FIFO queue, which is illustrated in figure 1(a). The external interface consists of an input channel *in* and an output channel *out*. Internally, the FIFO maintains a queue q of values that have been received via *in* but have not yet been sent via *out*.

The module consists of three sections, separated by horizontal bars for better readability, that contain declarations, definitions, and assertions. This structure of a module is conventional, but not mandatory: formally, a module is simply a list of



(a) Pictorial representation.

——— MODULE *SyncQueueInternal* ———

EXTENDS *Sequences*
CONSTANT *Message*
VARIABLES *in, out, q*

$$
\begin{array}{lcl}
\mathit{NoMsg} & \triangleq & \text{CHOOSE } x : x \notin \mathit{Message} \\
\mathit{Init} & \triangleq & q = \langle\rangle \land \mathit{in} = \mathit{NoMsg} \land \mathit{out} = \mathit{NoMsg} \\
\mathit{Enq}(m) & \triangleq & \land\ \mathit{in} \neq m \\
 & & \land\ \mathit{in}' = m \land q' = \mathit{Append}(q, m) \\
 & & \land\ \mathit{out}' = \mathit{out} \\
\mathit{Deq} & \triangleq & \land\ q \neq \langle\rangle \\
 & & \land\ \mathit{out}' = \mathit{Head}(q) \land q' = \mathit{Tail}(q) \\
 & & \land\ \mathit{in}' = \mathit{in} \\
\mathit{Next} & \triangleq & (\exists\, m \in \mathit{Message} : \mathit{Enq}(m)) \lor \mathit{Deq} \\
\mathit{vars} & \triangleq & \langle \mathit{in}, \mathit{out}, q \rangle \\
\mathit{FifoI} & \triangleq & \mathit{Init} \land \Box[\mathit{Next}]_{\mathit{vars}} \land \mathrm{WF}_{\mathit{vars}}(\mathit{Deq})
\end{array}
$$

THEOREM

$$
\begin{array}{lcl}
\mathit{FifoI} & \Rightarrow & \land\ \Box(q \in \mathit{Seq}(\mathit{Message})) \\
 & & \land\ \Box[\mathit{Deq} \Rightarrow \mathit{out}' \neq \mathit{out}]_{\mathit{vars}} \\
 & & \land\ \forall\, m \in \mathit{Message} : \mathit{in} = m \leadsto \mathit{out} = m
\end{array}
$$

(b) TLA+ specification with the internal behavior exposed.

——— MODULE *SyncQueue* ———

CONSTANT *Message*
VARIABLES *in, out*

$$\mathit{Internal}(q) \triangleq \text{INSTANCE } \mathit{SyncQueueInternal}$$

$$\mathit{Fifo} \triangleq \boldsymbol{\exists}\, q : \mathit{Internal}(q)!\mathit{FifoI}$$

(c) TLA+ interface specification.

Fig. 1. A FIFO queue with synchronous communication.

statements. Any identifier must have been declared or defined exactly once (possibly in an imported module) before it is used.

The first section declares *SyncQueueInternal* to be based on the standard TLA$^+$ module *Sequences*, which defines finite sequences and associated operations. Next, we find a declaration of the module parameters. The constant parameter *Message* intuitively represents the set of messages that are to be sent via the FIFO queue. The variable parameters *in*, *out*, and q represent the current state of the queue as shown in figure 1(a); their values will change as messages are received and forwarded.

The second section contains a list of definitions, which constitute the main body of the specification. The constant *NoMsg* is defined to equal some value that is not an element of the set *Message* (section 4 explains why this definition is sensible). The state predicate *Init* identifies legal initial states of the specification: the value of q should be the empty sequence $\langle\rangle$, while both *in* and *out* should equal the value *NoMsg*. For any value m, the formula $Enq(m)$ characterizes state transitions that correspond to an "enqueue" action[1]: we require m to be different from the current value of *in* so that the queue can recognize that the input channel has changed. (This condition is not essential, but is introduced mainly for expository purposes. An implementation could for example instantiate the parameter *Message* by a set of pairs consisting of the underlying data and an extra bit, which serves to distinguish two successive enqueue actions for the same data.) The value of the variable *in* at the state following the transition, denoted by in', will be m, and the new value of q is obtained by appending m at the end of whatever value q contains before the transition. Finally, we stipulate that the output channel *out* should not change during an enqueue action.The definition of the dequeue action *Deq* is similar. The action *Next* is defined as the disjunction of all enqueue actions $Enq(m)$, for m in *Message*, and of the dequeue action *Deq*.

The main definition of module *SyncQueueInternal* is that of the temporal formula *FifoI*, representing the "internal" specification of the FIFO queue. It is written as a conjunction: the first conjunct *Init* asserts that the first state of any behavior satisfying *FifoI* must respect the initial condition. The second conjunct specifies the next-state relation of the queue. More precisely, it asserts that every transition allowed by *FifoI* must either respect the formula *Next* or leave the expression *vars* unchanged; the latter is defined as the tuple $\langle in, out, q\rangle$ containing the state variables of interest. Because the value of a tuple is unchanged if and only if all its components are unchanged, this formula admits "stuttering steps" that do not affect the variables of interest. In a larger system that contains the FIFO queue as a component, such steps may represent actions of different components. The final conjunct of formula *FifoI* asserts a condition of weak fairness concerning the action *Deq*. It rules out behaviors where from some state onward, the *Deq* action is always enabled, but never occurs. Section 2 explores in more detail the temporal

[1] In this formula and throughout the paper, we use a standard TLA$^+$ notation that displays multi-line conjunctions and disjunctions as a list "bulleted" by the connective. This layout makes long formulas easier to read and reduces the number of parentheses.

logic TLA that underlies TLA$^+$, and discusses the fundamental concept of stuttering invariance.

The third section of module *SyncQueueInternal* asserts a theorem: it claims that the formula shown follows from the definitions. (In general, a module may state assumptions about constants, and theorems should then follow from the definitions and the stated or imported assumptions.) In a loose reading, the assertion of a theorem in a module can be regarded merely as a comment that highlights the specifier's intuitions. Formally, however, a theorem represents a proof obligation that must be discharged for the module to be correct, and we will turn to proof rules for verification in section 3. The theorem asserted of module *SyncQueueInternal* states that every behavior that satisfies formula *FifoI* has the following properties:

- at every state, the value of the variable q is a finite sequence whose elements are contained in *Message*,
- every *Deq* step changes the value of the output channel *out*, and
- every message that appears on the input channel will eventually be output.

The current version of TLA$^+$ described in [25] does not contain a language for writing proofs, although Lamport advocates a hierarchical proof notation [24].

Like HOL [15] and other logical specification languages, TLA$^+$ is declarative: the names of formulas such as *Init*, *Next* or *FifoI* are formally irrelevant, although it is good practice to make them meaningful. The meaning of a formula can always be uniquely and compositionally determined from the meaning of its subformulas, and how to do this is the main subject of the present paper. As in any logic, there are many logically equivalent ways to express a specification. For example, we could have replaced the definitions of *Enq* and *Next* by

$$\begin{array}{rcl} \mathit{Enq} & \triangleq & \wedge\ \mathit{in}' \in \mathit{Message} \wedge \mathit{in}' \neq \mathit{in} \\ & & \wedge\ q' = \mathit{Append}(q, \mathit{in}') \wedge \mathit{out}' = \mathit{out} \\ \mathit{Next} & \triangleq & \mathit{Enq} \vee \mathit{Deq} \end{array}$$

without changing the meaning of formula *FifoI*.

TLA$^+$ does not hide the complexity of a system by using built-in data types; as we will see in section 4, every value is just a set. Similarly, it does not presuppose any fixed system model such as shared-variable or message-passing concurrency, synchronous or asynchronous communication, etc. Its expressiveness comes from the unfettered use of set theory and the mechanism of definitional extension. For our example, we have chosen the internal variable q to change at the same time as the interface variables *in* and *out*, representing a synchronous style of communication. A specification of a FIFO queue using asynchronous communication channels is presented in Lamport's book [25, ch. 4].

The specifications of module *SyncQueueInternal* describe the behavior of the FIFO queue in terms of the three variables *in*, *out*, and q. One important principle in writing specifications is that of *information hiding*, which requires a component specification not to reveal the internal structure (or "implementation details") of the component. In our example, the variable q is such an implementation detail: as illustrated by the box in figure 1(a), only the behavior of the externally visible variables *in* and *out* should be constrained by the queue specification. Module *SyncQueue*, shown in figure 1(c), contains an interface specification of the FIFO queue based on the previous specification. In fact, it declares *in* and *out* as its only variable parameters. The following line instantiates the previously discussed module *SyncQueueInternal*: any operator *Op* defined in that module can be referenced as $Internal(q)!Op$ in module *SyncQueue*. The general form of instantiation in TLA$^+$ allows for substitution of expressions for module parameters; any remaining parameters are implicitly instantiated with the identifier of the same name valid at the point of instantiation; it is an error if that identifier has not been declared or defined. In our case, the parameters *Message*, *in*, and *out* of module *SyncQueueInternal* are instantiated by the corresponding parameters of module *SyncQueue*, whereas parameter q is instantiated by the local parameter of the operator *Internal*.

Module *SyncQueue* then defines the formula *Fifo*, representing the interface specification of the FIFO queue, as the formula obtained from $Internal(q)!FifoI$ by existential quantification over q. This formula is satisfied by every behavior where *in* and *out* take the values as described by the internal specification, but where q may take arbitrary values. (The precise semantics is defined in section 2.4.) In this respect, existential quantification represents hiding of internal state components, and formula *Fifo* specifies the interface of the FIFO queue.

2 TLA: THE TEMPORAL LOGIC OF ACTIONS

TLA$^+$ combines TLA, the Temporal Logic of Actions [23], and mathematical set theory. We now present the semantics of TLA, while sections 3 and 4 explore the verification of temporal formulas and the specification of data structures in set theory. Again, we emphasize that this exposition is aimed at a precise definition of TLA as a logical language; it does not attempt to explain the use of TLA for system specification.

2.1 Rationale

The logic of time has its origins in philosophy and linguistics, where it was intended to formalize temporal references in natural language [19, 32]. Around 1975, Pnueli [31] and others recognized that such logics could be useful as a basis for the semantics of computer programs. In particular, traditional formalisms based on pre- and post-conditions were found to be ill suited for the description of reactive systems that are continuously interacting with their environment and are not necessarily intended to terminate. Temporal logic, as it came to be called in computer science, offered an elegant framework to describe safety and liveness properties [9, 22] of reactive systems. Different dialects of temporal logic can be distinguished according to the properties assumed of the underlying model of time (e.g., discrete or dense) and to the connectives that can be used to refer to different moments in time (e.g., future vs. past references). For computer science applications, the fundamental distinction has been between linear-time and branching-time logics. In the linear-time view, a system is identified with the set of its executions, modeled as infinite sequences of states, whereas the branching-time view also considers the branching structure of a system. Linear-time temporal logics, including TLA, suffice to formulate correctness properties that hold of all the runs of a system, whereas branching-time temporal logics can also express possibility properties such as the existence of a path, from every reachable state, to a "reset" state. The discussion of the relative merits and deficiencies of these two kinds of temporal logics is beyond the scope of this paper, but see, e.g., [36] for a recent contribution to this subject, with many references to earlier papers.

Despite initial enthusiasm about the elegance of temporal logics, attempts to actually write complete system specifications in terms of their temporal logic properties revealed that not even a component as simple as a FIFO queue could be unambiguously specified [33]. This observation has led many researchers to specify reactive systems via some form of state machines while retaining temporal logic as a high-level language to describe the properties expected of such systems. Finite-state models are now routinely verified using model checking techniques [12], which have matured into debugging tools for industrially relevant developments.

Another weakness of standard temporal logic manifests itself when one attempts to compare two specifications of the same system, written at different levels of abstraction. Specifically, atomic system actions are usually described via a "next-state" operator, but the "grain of atomicity" typically changes during refinement, complicating comparisons between specifications. For example, we might want to refine the specification of the FIFO queue of figure 1(b) such that the operation of appending an element to a queue is described as a sequence of more elementary assignments.

TLA addresses these problems in the following ways: "internal" specifications are written by defining their initial conditions and next-state relation, resembling the description of a state machine, and are augmented by liveness and fairness conditions. Abstractness in the sense of information hiding is ensured by quantification over state variables. The refinement problem is solved by systematically allowing for stuttering steps that do not change the values of the state variables of interest; an implementation is allowed to refine such high-level stuttering into lower-level state changes. Similar ideas can be found in Back's refinement calculus [10] and in more recent versions of Abrial's B method [8]. These formalisms require side conditions to prevent infinite stuttering, expressed in terms of well-founded orderings. Temporal logic can state such requirements more abstractly in terms of high-level fairness conditions that must be preserved by a refinement.

Based on these concepts, TLA provides a unified logical language to express system specifications and their properties. A single set of logical rules is used for system verification and for proving refinement.

2.2 Transition formulas

The language of TLA is two-tiered: the base tier contains formulas that describe states and state transitions, whereas the top tier consists of temporal formulas that are evaluated over infinite sequences of states. In this section, we define the syntax and semantics of transition formulas, whereas the following sections will consider temporal formulas. Because transition formulas are just ordinary (untyped, first-order) predicate logic, this section can be quite brief.

Assume a given signature of first-order predicate logic, consisting of:

- at most denumerable sets $\mathcal{L}_F$ and $\mathcal{L}_P$ of function and predicate symbols, each symbol equipped with its arity, and
- a denumerable set $\mathcal{V}$ of variables, partitioned into denumerable sets $\mathcal{V}_F$ and $\mathcal{V}_R$ of flexible and rigid variables.

These sets should be disjoint from one another; moreover, no variable in $\mathcal{V}$ should be of the form v'. By $\mathcal{V}_{F'}$, we denote the set $\{v' \mid v \in \mathcal{V}_F\}$ of primed flexible variables, and by $\mathcal{V}_E$, the union $\mathcal{V} \cup \mathcal{V}_{F'}$ of primed and unprimed variable symbols.

Transition functions and *transition predicates* (also called *actions*) are terms and formulas built from the symbols in $\mathcal{L}_F$ and $\mathcal{L}_P$, and from the variables in $\mathcal{V}_E$. For example, if f is a ternary function symbol, p is a unary predicate symbol, $x \in \mathcal{V}_R$, and $v \in \mathcal{V}_F$, then the term e defined as $f(v, x, v')$ is a transition function, and the formula C defined as $\exists\, v' : p(f(v, x, v')) \wedge \neg(v' = x)$ is an action. We omit a formal inductive definition of the syntax of transition functions and formulas. Collectively, we use the term *transition formula* to refer to transition functions and predicates.

The semantics of transition formulas is also unsurprising. It is based on a first-order interpretation, which defines a universe of values and interprets each symbol in $\mathcal{L}_F$ by a function and each symbol in $\mathcal{L}_P$ by a relation of appropriate arities. In preparation for the semantics of temporal formulas, we distinguish between the valuations of flexible and rigid variables. A *state* is a mapping of the flexible variables in $\mathcal{V}_F$ to values of the universe. Given two states s and t and a valuation ξ of the rigid variables in $\mathcal{V}_R$, we can define the valuation $\alpha_{s,t,\xi}$ of the variables in $\mathcal{V}_E$ as the mapping such that $\alpha_{s,t,\xi}(x) = \xi(x)$ for $x \in \mathcal{V}_R$, $\alpha_{s,t,\xi}(v) = s(v)$ for $v \in \mathcal{V}_F$, and $\alpha_{s,t,\xi}(v') = t(v)$ for $v' \in \mathcal{V}_{F'}$. The semantics of a transition function or transition formula E, written $[\![E]\!]^{\xi}_{s,t}$, is then simply the standard predicate logic semantics of E with respect to the extended valuation $\alpha_{s,t,\xi}$.

We say that a transition predicate A is *valid* for the interpretation iff $[\![A]\!]^{\xi}_{s,t}$ is true for all states s, t and all valuations ξ. It is *satisfiable* iff $[\![A]\!]^{\xi}_{s,t}$ is true for some s, t, and ξ. Similarly, A is valid (satisfiable) for a class $\mathcal{C}$ of interpretations iff it is valid for all (satisfiable for some) interpretations in $\mathcal{C}$.

Finally, the notions of free and bound variables in a transition formula are defined as usual, as is the notion of substitution of a transition function a for a variable v, written $E[a/v]$. We assume that capture of free variables in a substitution is avoided by an implicit renaming of bound variables. For example, the set of free variables of the transition function e shown above is $\{x, v, v'\}$, and v' is a bound variable of the action C. We emphasize that at the level of transition formulas, we consider v and v' to be distinct, unrelated variables.

State formulas are transition formulas that do not contain free primed flexible variables. For example, the action C above is actually a state predicate. Because the semantics of state formulas only depends on a single state, we simply write $[\![P]\!]^\xi_s$ when P is a state formula. *Constant formulas* are even more restrictive in that they may contain free occurrences only of rigid variables; consequently, their semantics depends only on the valuation ξ.

TLA introduces some specific abbreviations at the level of transition formulas. If E is a state formula then E' is the transition formula obtained from E by replacing each free occurrence of a flexible variable v in E with its primed counterpart v' (where bound variables are renamed as necessary). For example, since C is a state formula, we may build the formula C' by substituting v' for v. Since v' is bound in C, this results in the formula $\exists\, y : p(f(v', x, y)) \wedge \neg(y = x)$, up to renaming of the bound variable.

For an action A, the state formula $\textsc{Enabled}\ A$ is obtained by existential quantification over all primed flexible variables that occur free in A. Thus, $[\![\textsc{Enabled}\ A]\!]^\xi_s$ holds if $[\![A]\!]^\xi_{s,t}$ holds for some state t, that is, if action A may occur in state s. For actions A and B, the action $A \cdot B$ is defined as $\exists\, z : A[z/v'] \wedge B[z/v]$ where v is a list of all flexible variables v_i such that v_i occurs free in B or v'_i occurs free in A, and z is a corresponding list of fresh variables. It follows that $[\![A \cdot B]\!]^\xi_{s,t}$ holds iff both $[\![A]\!]^\xi_{s,u}$ and $[\![B]\!]^\xi_{u,t}$ hold for some state u.

Because these abbreviations are defined in terms of quantification and substitution, their interplay can be quite delicate. For example, $\textsc{Enabled}\ P$ is by definition just P for any state predicate P, and therefore $(\textsc{Enabled}\ P)'$ equals P'. On the other hand, $\textsc{Enabled}\ (P')$ is a constant formula—if P does not contain any rigid variables then $\textsc{Enabled}\ (P')$ is valid iff P is satisfiable.

For an action A and a state function t we write $[A]_t$ to denote $A \vee t' = t$, and $\langle A \rangle_t$ for $A \wedge \neg(t' = t)$. Therefore, $[A]_t$ requires A to hold only if t changes value during a transition, whereas the dual formula $\langle A \rangle_t$ strengthens A in requiring that t changes value while A holds true.

2.3 Temporal formulas

We now turn to the temporal tier of TLA. Because it is less familiar that first-order predicate logic and because we wish to give precise definitions, we devote much more space to its presentation. However, the temporal formulas that one actually writes in TLA$^+$ specifications usually follow a standard idiom, and more than 95% of a typical specification consist of definitions at the transition level.

The (temporal) formulas of TLA are inductively defined as follows:

- Every state formula is a formula.

- Boolean combinations (connectives including $\neg$, $\wedge$, $\vee$, $\Rightarrow$, and $\equiv$) of formulas are formulas.
- If F is a formula then so is $\Box F$ ("always F").
- If A is an action and t is a state function then $\Box[A]_t$ ("always square A sub t") is a formula.
- If F is a formula and x is a rigid variable then $\exists\, x : F$ is a formula.
- If F is a formula and v is a flexible variable then $\boldsymbol{\exists}\, v : F$ is a formula.

In particular, an action A by itself is not a temporal formula, not even in the form $[A]_t$. Actions can occur only in subformulas $\Box[A]_t$.

To determine free and bound variables at the temporal level, we do not distinguish between primed and unprimed occurrences of flexible variables, and the quantifier $\boldsymbol{\exists}$ binds both kinds of occurrences. More formally, the set of free variables of a temporal formula is a subset of $\mathcal{V}_F \cup \mathcal{V}_R$. The free occurrences of variables in a state formula P, considered as a temporal formula, are precisely the free occurrences in P, considered as a transition formula. However, variable $v \in \mathcal{V}_F$ has a free occurrence in $\Box[A]_t$ iff either v or v' has a free occurence in A or in t. Similarly, substitution $F[e/v]$ of a state function e for a flexible variable v substitutes both e for v and e' for v' in the action subformulas of F, again after renaming bound variables as necessary. For example, substitution of the state function $h(v)$, where $h \in \mathcal{L}_F$ and $v \in \mathcal{V}_F$, for w in the temporal formula

$$\boldsymbol{\exists}\, v : p(v,w) \wedge \Box[q(v, f(w,v'), w')]_{g(v,w)}$$

results in the formula (up to renaming of the bound variable)

$$\boldsymbol{\exists}\, u : p(u, h(v)) \wedge \Box[q(u, f(h(v),u'), h(v'))]_{g(u,h(v))}$$

Because state formulas do not contain free occurrences of primed flexible variables, the definitions of substitutions for transition formulas and for temporal formulas agree on state formulas. The substitution of a (proper) transition function for a variable is not allowed as it could result in an expression that is not a well-formed TLA formula.

The semantics of temporal formulas is defined in terms of *behaviors*, which are infinite sequences of states, and of valuations of the rigid variables. For a behavior $\sigma = s_0 s_1 \ldots$, we write σ_i to refer to state s_i, and $\sigma|_i$ to denote the suffix $s_i s_{i+1} \ldots$. The following inductive definition assigns a truth value $[\![F]\!]^{\xi}_{\sigma} \in \{\mathbf{t}, \mathbf{f}\}$ to every formula F:

- $[\![P]\!]^{\xi}_{\sigma} = [\![P]\!]^{\xi}_{\sigma_0}$: state formulas are evaluated at the initial state of the behavior.
- The semantics of Boolean operators is the usual one.
- $[\![\Box F]\!]^{\xi}_{\sigma} = \mathbf{t}$ iff $[\![F]\!]^{\xi}_{\sigma|_i} = \mathbf{t}$ for all $i \in \mathbb{N}$: this is the usual clause from linear-time temporal logic.

- $[\![\Box[A]_t]\!]^\xi_\sigma = \mathbf{t}$ iff for all $i \in \mathbb{N}$, $[\![t]\!]^\xi_{\sigma_i} = [\![t]\!]^\xi_{\sigma_{i+1}}$ or $[\![A]\!]^\xi_{\sigma_i,\sigma_{i+1}} = \mathbf{t}$: such a formula holds iff every state transition in σ that does not leave t unchanged satisfies A.
- $[\![\exists\, x : F]\!]^\xi_\sigma = \mathbf{t}$ iff $[\![F]\!]^\eta_\sigma = \mathbf{t}$ for some valuation η such that $\eta(y) = \xi(y)$ for all $y \in \mathcal{V}_R \setminus \{x\}$.
- The semantics of formulas $\boldsymbol{\exists}\, v : F$ will be defined in section 2.4 below.

Abbreviations for temporal formulas include the universal quantifiers $\forall$ and $\boldsymbol{\forall}$ over rigid and flexible variables. The formula $\Diamond F$ ("eventually F"), defined as $\neg\Box\neg F$, asserts that F holds of some suffix of the behavior; similarly, $\Diamond\langle A\rangle_t$ ("eventually angle A sub t") is defined as $\neg\Box[\neg A]_t$ and asserts that some transition satisfies A and changes the value of t. We write $F \leadsto G$ ("F leads to G") for the formula $\Box(F \Rightarrow \Diamond G)$, which asserts that every occurrence of F will eventually be followed by an occurrence of G. Combinations of the "always" and "eventually" operators express "infinitely often" ($\Box\Diamond$) and "almost always" ($\Diamond\Box$). Observe that a formula can be both infinitely often true and infinitely often false, thus "almost always" is strictly stronger than "infinitely often". These combinations are particularly important as the building blocks to formulate fairness conditions. In particular, weak and strong fairness for an action $\langle A\rangle_t$ are defined as

$$\begin{aligned} \mathrm{WF}_t(A) &\equiv (\Box\Diamond\neg\textsc{Enabled}\,\langle A\rangle_t) \vee \Box\Diamond\langle A\rangle_t \\ \mathrm{SF}_t(A) &\equiv (\Diamond\Box\neg\textsc{Enabled}\,\langle A\rangle_t) \vee \Box\Diamond\langle A\rangle_t \end{aligned}$$

Weak fairness stipulates that an action $\langle A\rangle_t$ occurs infinitely often during a behavior if it is almost always enabled; strong fairness even requires that the action must happen infinitely often if it is infinitely often (but not necessarily persistently) enabled.

2.4 Stuttering invariance and quantification

Formulas $\Box[A]_t$ allow for "stuttering": besides state transitions that satisfy A, they also allow any transitions that do not change the state function t. In particular, duplications of states can not be observed by formulas of this form. Stuttering invariance is important in connection with refinement and composition [22].

To formalize this notion, for a set V of flexible variables we define two states s and t to be V-equivalent, written $s =_V t$, iff $s(v) = t(v)$ for all $v \in V$. We define V-stuttering equivalence, written $\approx_V$, as the smallest equivalence relation on behaviors that contains $\rho \circ \langle s\rangle \circ \sigma$ and $\rho \circ \langle t, u\rangle \circ \sigma$, for any finite sequence of states ρ, infinite sequence of states σ, and V-equivalent states $s =_V t =_V u$. Intuitively, V-stuttering equivalence allows for duplication and deletion of finite repetitions of V-equivalent states. In particular, the relation $\approx_{\mathcal{V}_F}$, which we also write as $\approx$, identifies two behaviors that differ by duplications or deletions of identical states.

The fundamental theorem asserting that TLA is not expressive enough to distinguish stuttering-equivalent behaviors can now be formally stated as follows:

Theorem 1 (stuttering invariance)**.** Assume that F is a TLA formula whose free flexible variables are among V, that $\sigma \approx_V \tau$ are V-stuttering equivalent behaviors, and that ξ is a valuation. Then $[\![F]\!]^\xi_\sigma = [\![F]\!]^\xi_\tau$.

For TLA formulas without quantification over flexible variables, it is not hard to prove theorem 1 from the semantic clauses of section 2.3 by induction on the structure of formulas [23, 5]. On the other hand, quantification over flexible variables requires some attention: the "obvious" semantic clause for formulas $\exists\, v : F$ would have $[\![\exists\, v : F]\!]^\xi_\sigma = \mathbf{t}$ iff $[\![F]\!]^\xi_\tau = \mathbf{t}$ for some behavior τ whose states τ_i agree with the corresponding states σ_i on all variables except for v. This definition, however, would not preserve stuttering invariance. For example, consider the formula

$$F \;\triangleq\; v = c \wedge w = c \wedge \Diamond(w \neq c) \wedge \Box[v \neq c]_w \qquad\qquad \begin{array}{ccccc} \sigma & & & & \\ v & c & d & d & \cdots \\ w & c & c & d & \cdots \end{array}$$

that requires that both variables v and w initially equal the constant c, that eventually, w is different from c, and that v must be different from c whenever w changes value. Any behavior σ that satisfies F must therefore contain two distinct transitions, the first of which changes v from c to some other value (but preserving the value of w), while the second transition changes w, as indicated in the picture. In particular, $\sigma_1(w)$ must equal c, hence the above definition of quantification implies that $\tau_1(w)$ must equal c, for any behavior τ satisfying the formula $\exists\, v : F$. However, the behavior τ obtained from σ by removing the second state (where v equals d but w equals c) is $\{w\}$-stuttering equivalent to σ. Because w is the only free flexible variable of $\exists\, v : F$, theorem 1 asserts that τ should satisfy $\exists\, v : F$, although $\tau_1(w)$ is different from c.

In other words, the definition of quantification over flexible variables must allow for the removal of transitions that modify only the bound variables. This observation motivates the following semantic clause for quantified formulas: for a flexible variable v, we say that two behaviors σ and τ are *equal up to* v iff σ_i and τ_i agree on all variables in $\mathcal{V}_F \setminus \{v\}$, for all $i \in \mathbb{N}$. We say that σ and τ are *similar up to* v, written $\sigma \simeq_v \tau$ iff there exist behaviors σ' and τ' such that

- σ and σ' are stuttering equivalent ($\sigma \approx \sigma'$),
- σ' and τ' are equal up to v, and
- τ and τ' are again stuttering equivalent ($\tau \approx \tau'$).

Now, we define $[\![\exists\, v : F]\!]^\xi_\sigma = \mathbf{t}$ iff $[\![F]\!]^\xi_\tau = \mathbf{t}$ holds for some behavior τ similar to σ up to v. The definition of $\simeq_v$ explicitly allows for stuttering, and therefore theorem 1 holds true for all TLA formulas.

2.5 Properties, refinement, and composition

As we have seen in the example of the FIFO queue, TLA uses the same formalism of temporal logic to represent system specifications and properties. System specifications are usually written in the form

$$\exists\!\!\!\exists\, x \,:\, \mathit{Init} \wedge \Box[\mathit{Next}]_v \wedge L$$

where v is the list of all relevant state variables, x is the list of internal (hidden) variables, *Init* is a state predicate representing the initial condition, *Next* is an action that describes the next-state relation, usually written as a disjunction of more elementary actions, and L is a conjunction of formulas $\mathrm{WF}_v(A)$ or $\mathrm{SF}_v(A)$ asserting fairness assumptions of individual disjuncts of *Next*. However, other forms of specifications are possible and can occasionally be useful. Asserting that a property F holds of a specification S amounts to saying that every behavior that satisfies S must also satisfy F; in other words, it asserts the validity of the implication $S \Rightarrow F$. For example, the theorem asserted in module *SyncQueueInternal* states three essential properties of the FIFO queue.

Unlike most other temporal logics, TLA is intended to support stepwise system development by refinement of specifications [10]. The basic idea of refinement consists in successively adding implementation detail while preserving the properties established at an abstract level. In a refinement-based approach to system development, one proceeds by writing successive models, each of which introduces some additional detail while preserving the essential properties of the preceding model. Fundamental properties of a system can thus be established at high levels of abstraction, errors can be detected in early phases, and the complexity of formal assurance is spread over the entire development process. A refinement C preserves all TLA properties established of an abstract specification A if and only if for every formula F, if $A \Rightarrow F$ is valid, then so is $C \Rightarrow F$. This condition is in turn equivalent to requiring the validity of $C \Rightarrow A$. Because C will contain extra variables to represent the lower-level detail, and because these variables will change in transitions that have no counterpart at the abstract level, stuttering invariance of TLA formulas is essential to make validity of implication a reasonable definition of refinement.

Stuttering invariance is also essential for composition to be representable as conjunction [16]. In fact, if A and B are TLA specifications of two components, then $A \wedge B$ describes those behaviors that satisfy both components' initial conditions, that allow actions of either process to occur, synchronizing on common variables (which represent interfaces between the components), and that satisfy all relevant liveness properties. In particular, stuttering invariance ensures that each component may perform local actions without interfering with the specification of the other component.

As a test of these ideas, we might want to convince ourselves that two FIFO queues in a row again implement a FIFO queue. Let us assume that the two queues are connected by a channel *mid*, then the above principles seem to imply that the formula

$$\mathit{Fifo}[\mathit{mid}/\mathit{out}] \wedge \mathit{Fifo}[\mathit{mid}/\mathit{in}] \Rightarrow \mathit{Fifo}$$

is valid. Unfortunately, this is not true, for the following reason: formula *Fifo* implies that the *in* and *out* channels never change simultaneously, whereas the conjunction on the left-hand side allows such changes (if the left-hand queue performs an *Enq* action, while the right-hand queue performs a *Deq*). This technical problem can be attributed to a design decision taken in the specification of the FIFO queue that does not allow simultaneous changes to its input and output interfaces, a specification style known as "interleaving specifications". In fact, the argument merely shows that the composition of two non-interleaving queues does not implement a non-interleaving queue. Choosing an interleaving or a non-interleaving style is an artifact of the model that represents the actual system: interleaving specifications are usually easier to write and understand. The problem disappears if we explicitly add a non-interleaving assumption for the composition, as shown by the validity of the implication

$$Fifo[mid/out] \wedge Fifo[mid/in] \wedge \Box[in' = in \vee out' = out]_{in,out} \;\Rightarrow\; Fifo \qquad (1)$$

whose proof will be considered in section 3.5.

2.6 Variations and extensions

We discuss some of the choices that we have made in the presentation of TLA, as well as possible extensions.

Transition formulas and priming. Our presentation of TLA is based on standard first-order logic, to the extent possible. In particular, we have defined transition formulas as formulas of ordinary predicate logic over a large set $\mathcal{V}_E$ of variables where v and v' are unrelated. An alternative presentation would consider $'$ as an operator, resembling the next-time modality of temporal logic. In fact, this appears to be the presentation preferred by Lamport [25]. The semantics of temporal formulas is unaffected by the choice of presentation, and the style adopted in this paper corresponds well to the verification rules of TLA, explored in section 3.

Compositional verification. We have argued in section 2.5 that composition is represented in TLA as conjunction. Because components can rarely be expected to operate correctly in arbitrary environments, their specifications usually include some assumptions about the environment. An *open system specification* is one that does not constrain its environment; it asserts that the component will function correctly provided that the environment behaves as expected. One way to write such specifications is in the form of implications $E \Rightarrow M$ where E describes the environment assumptions and M, the component specification. However, it turns out that often a stronger form of specifications is desirable that requires the component to adhere to its description M for at least as long as the environment has not broken its obligation E. In particular, when systems are built from "open" component specifications, this form, written $E \stackrel{+}{\rhd} M$, allows for a strong composition rule that can discharge mutual assumptions between components [3, 13]. It can be shown that the operator $\stackrel{+}{\rhd}$ is actually definable in TLA, and that the resulting composition rule can be justified in terms of an abstract logic of specifications, supplemented by principles specific to TLA [4, 6].

TLA*. TLA defines distinct tiers of transition formulas and temporal formulas, where transition formulas must be guarded by "brackets" to ensure stuttering invariance. Although the separation between the two tiers is natural when writing system specifications, it is not a prerequisite to obtaining stuttering invariance. In [30], I have defined the logic TLA* whose syntax is defined by mutual induction on so-called *pure* and *impure* formulas. The former correspond to the temporal formulas of TLA, whereas the latter, generalizing transition formulas, are formed from Boolean combinations of F and $\circ G$, where F and G are pure formulas and $\circ$ is the next-time modality of temporal logic. For example, the TLA* formula

$$\Box\big[A \Rightarrow \circ\Diamond\langle B\rangle_u\big]_t$$

requires that every $\langle A\rangle_t$ action must eventually be followed by $\langle B\rangle_u$. Assuming appropriate syntactic conventions, TLA* is a generalization of TLA because every TLA formula is also a TLA* formula, with the same semantics. On the other hand, it can be shown that every TLA* formula can be expressed in TLA using some extra quantifiers. For example, the formula above is equivalent to the TLA formula[2]

$$\begin{array}{ll} \boldsymbol{\exists}\, v : & \land\ \Box((v = c) \equiv \Diamond\langle B\rangle_u) \\ & \land\ \Box[A \Rightarrow v' = c]_t \end{array}$$

where c is a constant and v is a fresh flexible variable. Because of its richer syntax, TLA* can be axiomatized in a rather straightforward manner. For example,

$$\Box[F \Rightarrow \circ F]_V \Rightarrow (F \Rightarrow \Box F)$$

where F is a temporal formula and V is a tuple containing all flexible variables with free occurrences in F, is a TLA* representation of the usual induction axiom of temporal logic; this is a TLA formula only if F is in fact a state formula.

Binary temporal operators. Unlike standard linear-time temporal logic [28], TLA does not include binary operators such as **until**, because they are not necessary for writing system specifications, and because they can be confusing, especially when nested. These operators are, however, definable in TLA using quantification over flexible variables. For example, suppose that P and Q are state predicates whose free variables are among w, that v is a flexible variable that does not appear in w,

[2] This definition assumes that the universe contains at least two distinct values; one-element universes are not very interesting.

and that c is a constant. Then P **until** Q can be defined as the formula

$$\begin{aligned}\exists\!\!\exists\, v\ :\ & \wedge\ (v = c) \equiv Q\\ & \wedge\ \Box[(v \neq c \Rightarrow P) \wedge (v' = c \equiv (v = c \vee Q'))]_{\langle v,w\rangle}\\ & \wedge\ \Diamond Q\end{aligned}$$

The idea is to use the auxiliary variable v to remember whether Q has already been true. As long as Q has been false, P is required to hold. For arbitrary TLA formulas F and G, the formula F **until** G can be defined along the same lines, using a technique as shown for the translation of TLA* formulas above.

3 TLA PROOF RULES

As TLA formulas are used to describe systems as well as their properties, deductive system verification can be based on logical axioms and rules of TLA. More precisely, a system described by formula *Spec* has property *Prop* if and only if every behavior that satisfies *Spec* also satisfies *Prop*, that is, iff the implication $\mathit{Spec} \Rightarrow \mathit{Prop}$ is valid over the class of interpretations where the function and predicate symbols have the intended meaning. System verification, in principle, therefore requires reasoning about sets of behaviors. The TLA proof rules are designed to reduce this temporal reasoning, as far as possible, to the proof of verification conditions expressed in the underlying predicate logic, a strategy that is commonly referred to as *assertional reasoning.* In this section, we state some typical rules and illustrate their use; more information can be found elsewhere [23].

3.1 Invariants

Invariants characterize the set of states that can be reached during system execution; they constitute the basic safety properties of interest and are also the starting point for almost any verification attempt. In TLA, an invariant is expressed by a formula of the form $\Box I$ where I is a state formula.

A basic rule for proving invariants is given by

$$\frac{I \wedge [N]_t \Rightarrow I'}{I \wedge \Box[N]_t \Rightarrow \Box I}\ \text{(INV1)}$$

This rule asserts that for every interpretation for which the antecedent $I \wedge [N]_t \Rightarrow I'$ is a valid transition formula, the consequent $I \wedge \Box[N]_t \Rightarrow \Box I$ is a valid temporal formula. The antecedent states that every possible transition (stuttering or not) preserves I; thus, if I holds initially it is guaranteed to hold forever. Formally, the correctness of rule (INV1) is easily established by induction on behaviors. Because the antecedent is a transition formula, its proof relies on standard axioms and proof rules of predicate logic, augmented by "data" axioms that characterize the intended interpretations.

For example, we can use (INV1) to prove the invariant $\Box(q \in Seq(Message))$ of the FIFO queue specified in module *SyncQueueInternal* of figure 1(b). We have to prove

$$FifoI \Rightarrow \Box(q \in Seq(Message)) \tag{2}$$

which, by rule (INV1), the definition of formula *FifoI*, and propositional logic can be reduced to proving

$$Init \quad \Rightarrow \quad q \in Seq(Message) \tag{3}$$

$$q \in Seq(Message) \land [Next]_{vars} \quad \Rightarrow \quad q' \in Seq(Message) \tag{4}$$

Because the empty sequence is certainly a finite sequence of messages, (3) follows from the definition of *Init* and appropriate data axioms. Similarly, the proof of (4) reduces to proving preservation of the invariant under stuttering, *Deq*, and *Enq*(m) actions, for any $m \in Message$, all of which are again straightforward.

3.2 Step simulation

The following rule can be used to prove "action invariants"; it relies on a previously proven state invariant I:

$$\frac{I \land I' \land [M]_t \Rightarrow [N]_u}{\Box I \land \Box[M]_t \Rightarrow \Box[N]_u} \quad \text{(TLA2)}$$

In particular, it follows from (TLA2) that the next-state relation can be strengthened by an invariant:

$$\Box I \land \Box[M]_t \Rightarrow \Box[M \land I \land I']_t$$

Note that the converse of this implication is not valid: the right-hand side holds of any behavior where t never changes, independently of the value of I.

We may use (TLA2) to prove that the FIFO queue never dequeues the same value twice in a row:

$$FifoI \Rightarrow \Box[Deq \Rightarrow out' \neq out]_{vars} \tag{5}$$

This proof requires an invariant that in particular asserts that no consecutive elements of the internal queue are identical:

$$\begin{array}{lll} Inv & \triangleq & \text{LET } oq \triangleq \langle out \rangle \circ q \\ & & \text{IN } \land\ in = oq[Len(oq)] \\ & & \phantom{\text{IN }} \land\ \forall\, i \in 1..Len(oq) - 1 \;:\; oq[i] \neq oq[i+1] \end{array}$$

We have used some TLA$^+$ syntax in formulating *Inv*: the local abbreviation *oq* denotes the sequence obtained by prefixing the current value of the output channel *out* to the internal queue q; also, TLA$^+$ represents a sequence s as a function sucht that its elements can be accessed as $s[1], \ldots, s[Len(s)]$. Formula *Inv* asserts that the current value of the input channel equals the last element of the sequence *oq*, and that no two consecutive elements of *oq* are identical. The proof of $FifoI \Rightarrow \Box Inv$ follows the pattern used in proving invariant (2) above, using rule (INV1).

For the proof of (5), rule (TLA2) requires that we show the validity of

$$Inv \land Inv' \land [Next]_{vars} \Rightarrow [Deq \Rightarrow out' \neq out]_{vars} \tag{6}$$

The proof of (6) reduces to the three cases of a stuttering transition, an $Enq(m)$ action, and a *Deq* action. Only the last case is non-trivial. Its proof relies on the definition of *Deq*, which implies that q is non-empty and that $out' = Head(q)$. In particular, the sequence *oq* contains at least two elements, and therefore *Inv* implies that $oq[1]$, which is just *out*, is different from $oq[2]$, which is $Head(q)$. This suffices to prove $out' \neq out$.

3.3 Liveness properties

Liveness properties, intuitively, assert that something good must eventually happen [9, 21]. TLA provides rules to deduce elementary liveness properties from the fairness properties assumed of a specification; more complex properties can then be inferred with the help of well-founded orderings.

The following rule can be used to prove a leads-to formula from a weak fairness assumption; a similar rule exists for strong fairness.

$$\frac{\begin{array}{c} I \land I' \land P \land [N]_t \Rightarrow P' \lor Q' \\ I \land I' \land P \land \langle N \land A \rangle_t \Rightarrow Q' \\ I \land P \Rightarrow \textsc{Enabled}\, \langle A \rangle_t \end{array}}{\Box I \land \Box[N]_t \land \mathrm{WF}_t(A) \Rightarrow (P \leadsto Q)} \quad \text{(WF1)}$$

In this rule, P and Q are state predicates, I is again an invariant, $[N]_t$ represents the next-state relation, and $\langle A \rangle_t$ is a "helpful action" [27] for which weak fairness is assumed. Again, all three premises of (WF1) are transition formulas. To see why the rule is correct, assume that σ is a behavior satisfying $\Box I \land \Box[N]_t \land \mathrm{WF}_t(A)$, and that P holds of state σ_i. We have to show that Q holds of some σ_j with $j \geq i$. By the first premise, any successor of a state satisfying P has to satisfy P or Q, so P must hold for as long as Q has not been true. The third premise ensures that in all of these states, action $\langle A \rangle_t$ is enabled, and so the assumption of weak fairness ensures that eventually $\langle A \rangle_t$ occurs, unless Q has become true before, in which case we are done. Finally, by the second premise, any $\langle A \rangle_t$-successor (which, by assumption, is in fact an $\langle N \land A \rangle_t$-successor) of a state satisfying P must satisfy Q, which proves the claim.

For our running example, we can use rule (WF1) to prove that every message stored in the queue will eventually move closer to the head of the queue or even to the output channel. Formally, let the state predicate $at(k,x)$ be defined by

$$at(k,x) \;\triangleq\; k \in 1..Len(q) \land q[k] = x$$

We will use (WF1) to prove

$$FifoI \Rightarrow \big(at(k,x) \leadsto (out = x \lor at(k-1,x))\big) \tag{7}$$

where k and x are rigid variables. The following proof outline illustrates the application of rule (WF1), the lower-level steps relying on data axioms are omitted.

1. $at(k,x) \land [Next]_{vars} \Rightarrow at(k,x)' \lor out' = x \lor at(k-1,x)'$
   1.1. $at(k,x) \land m \in Message \land Enq(m) \Rightarrow at(k,x)'$
   1.2. $at(k,x) \land Deq \land k = 1 \Rightarrow out' = x$
   1.3. $at(k,x) \land Deq \land k > 1 \Rightarrow at(k-1,x)'$
   1.4. $at(k,x) \land vars' = vars \Rightarrow at(k,x)'$
   1.5. Q.E.D.
   From steps 1.1–1.4 by the definitions of $Next$ and $at(k,x)$.
2. $at(k,x) \land \langle Deq \land Next\rangle_{vars} \Rightarrow out' = x \lor at(k-1,x)'$
   Follows from steps 1.2 and 1.3 above.
3. $at(k,x) \Rightarrow \text{ENABLED}\ \langle Deq\rangle_{vars}$
   For any k, $at(k,x)$ implies that $q \neq \langle\rangle$ and thus the enabledness condition.

However, rule (WF1) cannot be used to prove the stronger property that every input to the queue will eventually be dequeued,

$$FifoI \Rightarrow \forall\, m \in Message : in = m \leadsto out = m \tag{8}$$

because there is no single "helpful action": the number of Deq actions necessary to produce the input element on the output channel depends on the length of the queue. Intuitively, the argument used to establish property (7) must be iterated. The following rule formalizes this idea as an induction over a well-founded relation $(D,\succ)$: a binary relation such that there does not exist an infinite descending chain $d_1 \succ d_2 \succ \ldots$ of elements $d_i \in D$.

$$\frac{\begin{array}{c}(D,\succ) \text{ is well-founded}\\ F \Rightarrow \forall\, d \in D : \big(G \leadsto (H \lor \exists\, e \in D : d \succ e \land G[e/d])\big)\end{array}}{F \Rightarrow \forall\, d \in D : (G \leadsto H)} \quad \text{(LATTICE)}$$

In this rule, d and e are rigid variables such that d does not occur in H and e does not occur in G. For convenience, we have stated rule (LATTICE) in a language of set theory where, in particular, $\forall\, x \in S : F$ abbreviates the formula $\forall\, x : x \in S \Rightarrow F$.

Unlike the premises of the rules considered so far, the second hypothesis of rule (LATTICE) is itself a temporal formula that requires that every occurrence of G, for any value $d \in D$, be followed either by an occurrence of H, or again by some G, for some smaller value e. Because the first hypothesis ensures that there cannot be an infinite descending chain of values in D, eventually H must become true. In principle, the hypothesis of well-foundedness can itself be expressed in TLA by asserting the validity of the formula

$$\begin{array}{l} \land\ \forall\, d \in D : \neg(d \succ d) \\ \land\ \boldsymbol{\forall}\, v : \Box(v \in D) \land \Box[v \succ v']_v \ \Rightarrow\ \Diamond\Box[\text{FALSE}]_v \end{array}$$

whose first conjunct expresses the irreflexivity of $\succ$ and whose second conjunct asserts that any sequence of values in D that can only change by decreasing with respect to $\succ$ must eventually become stationary. In fact, if this formula is valid over a given interpretation then $\succ$ is interpreted by a well-founded relation. In system verification, well-foundedness is however usually considered as a "data axiom".

Choosing $(Nat, >)$, the set of natural numbers with the standard "greater-than" relation as the well-founded domain, the proof of property (8) follows from property (7) and the invariant *Inv* defined in section 3.2 using rule (LATTICE).

Lamport [23] lists further (derived) rules for liveness properties, including introduction rules for proving formulas $\mathrm{WF}_t(A)$ and $\mathrm{SF}_t(A)$.

3.4 Simple temporal logic

(STL1)	$\dfrac{F}{\Box F}$	(STL4)	$\Box(F \Rightarrow G) \Rightarrow (\Box F \Rightarrow \Box G)$
(STL2)	$\Box F \Rightarrow F$	(STL5)	$\Box(F \land G) \equiv (\Box F \land \Box G)$
(STL3)	$\Box\Box F \equiv \Box F$	(STL6)	$\Diamond\Box(F \land G) \equiv (\Diamond\Box F \land \Diamond\Box G)$

Fig. 2. Simple temporal logic.

The proof rules considered so far support the derivation of typical correctness properties of systems. In addition, TLA satisfies standard axioms and rules of linear-time temporal logic that are useful when preparing the application of verification rules. Figure 2 contains the axioms and rules of "simple temporal logic", adapted from Lamport [23]. It can be shown that this is just a non-standard presentation of the modal logic S4.2 [17], implying that these laws by themselves characterize a modal accessibility relation for $\Box$ that is reflexive, transitive, and locally convex (or confluent). The last condition asserts that for any state s and states t, u that are both accessible from s there is a state v that is accessible from t and u.

3.5 Quantifier rules

The elementary rules for quantification (over rigid or flexible variables) are those familiar from first-order logic:

$$F[c/x] \Rightarrow \exists\, x : F \quad (\exists\,\mathrm{I}) \qquad\qquad \frac{F \Rightarrow G}{(\exists\, x : F) \Rightarrow G} \; (\exists\,\mathrm{E})$$

$$F[t/v] \Rightarrow \boldsymbol{\exists}\, v : F \quad (\boldsymbol{\exists}\,\mathrm{I}) \qquad\qquad \frac{F \Rightarrow G}{(\boldsymbol{\exists}\, v : F) \Rightarrow G} \; (\boldsymbol{\exists}\,\mathrm{E})$$

In these rules, x is a rigid and v is a flexible variable. The elimination rules $(\exists\,\mathrm{E})$ and $(\boldsymbol{\exists}\,\mathrm{E})$ require the usual proviso that the bound variable should not be free in formula G. In the introduction rules, t is a state function, while c is a constant function. Observe that if we allowed an arbitrary state function in rule $(\exists\,\mathrm{I})$, we could prove

$$\exists\, x : \Box(x = v) \tag{9}$$

for any state variable v from the premise $\Box(v = v)$, provable by (STL1). However, formula (9) asserts that v remains constant throughout a behavior, which can obviously not be valid.

Since existential quantification over flexible variables corresponds to hiding of state components, the rules $(\boldsymbol{\exists}\,\mathrm{I})$ and $(\boldsymbol{\exists}\,\mathrm{E})$ play a fundamental role in proofs of refinement for reactive systems. In this context, the "witness" t is often called a *refinement mapping* [1]. For example, the concatenation of the two low-level queues provides a suitable refinement mapping to prove the validity of formula (1), which claimed that two FIFO queues in a row implement a FIFO queue, assuming non-interleaving. Whereas these rules are standard, one should recall from section 2.2 that care has to be taken when substitutions are applied to formulas that are defined in terms of quantification. In particular, the formulas $\mathrm{WF}_t(A)$ and $\mathrm{SF}_t(A)$ contain the subformula $\textsc{Enabled}\,\langle A\rangle_t$, and therefore, e.g., $\mathrm{WF}_t(A)[e/v]$ need not be equivalent to the formula $\mathrm{WF}_{t[e/v]}(A[e/v, e'/v'])$, cf. also [23].

Unfortunately, refinement mappings need not always exist. For example, $(\boldsymbol{\exists}\,\mathrm{I})$ cannot be used to prove the valid TLA formula (excluding one-element universes)

$$\boldsymbol{\exists}\, v : \Box\Diamond\langle \textsc{true}\rangle_v \tag{10}$$

that asserts the existence of a flexible variable whose value changes infinitely often. (Such a variable could serve as a trigger for some computations.) In fact, an attempt to prove (10) by rule $(\boldsymbol{\exists}\,\mathrm{I})$ would require to exhibit a state function t whose value is certain to change infinitely often in any behavior. However, it is easy to show by induction on the syntax of state functions that for any t there exists a behavior such that the value of t remains constant forever.

An approach to solving this problem, advocated in [1], consists in adding *auxiliary variables* such as history and prophecy variables. Formally, this approach consists in adding special introduction rules for auxiliary variables. The proof of $G \Rightarrow \exists\, v : F$ is then reduced to first proving a formula of the form $G \Rightarrow \exists\, a : G_{aux}$ using a rule for auxiliary variables, and then use the rules ($\exists$E) and ($\exists$I) above to prove $G \wedge G_{aux} \Rightarrow \exists\, v : F$.

4 FORMALIZED MATHEMATICS: THE ADDED VALUE OF TLA$^+$

The definitions of the syntax and semantics of TLA in section 2 were generic in terms of an underlying language of predicate logic and its interpretation. TLA$^+$ instantiates this generic definition of TLA with a specific first-order theory, namely Zermelo-Fränkel set theory with choice. This adoption of a standard interpretation enables precise and unambiguous specifications of the "data structures" on which specifications are based; we have seen in the example proofs in section 3 that reasoning about the data accounts for most of the steps that need to be proved during system verification. TLA$^+$ also provides facilities for structuring a specification as a hierarchy of modules, for declaring parameters, and most importantly, for defining operators. These facilities are essential for writing actual specifications and must therefore be mastered by any user of TLA$^+$. However, from the foundational point of view adopted in this paper, they are just syntactic sugar. We will therefore concentrate on the set-theoretic foundations, referring the reader to Lamport's book [25] for a detailed presentation of the language of TLA$^+$.

4.1 Elementary data structures: basic set theory

The signature of the predicate logic underlying TLA$^+$ contains a single binary predicate symbol $\in$ and no function symbols.[3] Terms and formulas at the transition level are defined as indicated in section 2.2, with an extra term formation rule that defines CHOOSE $x : A$ to be a transition function, for any action A and variable x.[4] The occurrences of x in the term CHOOSE $x : A$ are bound. To this first-order language corresponds a set-theoretic interpretation: every TLA$^+$ value is a set. Moreover, $\in$ is interpreted as set membership and the interpretation is equipped with an (unspecified) choice function ε mapping every non-empty collection C of values to some element c of C, and mapping the empty collection to an arbitrary value. The interpretation of terms CHOOSE $x : P$ is then defined as

$$[\![\textsc{choose}\ x : P]\!]^{\xi}_{s,t} \quad = \quad \varepsilon(\{d \mid [\![P]\!]_{\alpha_{s,t,\xi[d/x]}} = \mathbf{t}\})$$

[3] Once again, our presentation deviates somewhat from Lamport [25] who prefers to treat all subsequent constructs on an equal footing rather than distinguishing between basic and derived operators.

[4] Temporal formulas are defined as indicated in section 2.3; in particular, CHOOSE is never applied to a temporal formula.

This definition employs the choice function to return some element of the universe satisfying P provided there is some such element. We should remark that the choice function is applied to a collection that need not be an element of the universe: we are here at the "meta level", and in set-theoretic terminology, ε can be applied to classes and not just to sets. Because ε is a function, it produces the same value when applied to equal arguments. It follows that choice satisfies the laws

$$(\exists\, x : P) \;\equiv\; P[(\textsc{choose}\ x : P)/x] \tag{11}$$

$$(\forall\, x : (P \equiv Q)) \;\Rightarrow\; (\textsc{choose}\ x : P) = (\textsc{choose}\ x : Q) \tag{12}$$

TLA$^+$ defines many abbreviated connectives that we will freely use in the following. For example, $\exists\, x, y \in S : P$ abbreviates $\exists\, x : \exists\, y : x \in S \wedge y \in S \wedge P$, and similar notation is used with $\forall$ and CHOOSE . TLA$^+$ also borrows some computer science notation for conditionals IF _ THEN _ ELSE _ and local declarations, written LET _ IN _.

union	$\textsc{union}\ S \triangleq \textsc{choose}\ M : \forall\, x : (x \in M \equiv \exists\, T \in S : x \in T)$
binary union	$S \cup T \triangleq \textsc{union}\ \{S, T\}$
subset	$S \subseteq T \triangleq \forall\, x : (x \in S \Rightarrow x \in T)$
powerset	$\textsc{subset}\ S \triangleq \textsc{choose}\ M : \forall\, x : (x \in M \equiv x \subseteq S)$
comprehension 1	$\{x \in S : P\} \triangleq \textsc{choose}\ M : \forall\, x : (x \in M \equiv x \in S \wedge P)\}$
comprehension 2	$\{t : x \in S\} \triangleq \textsc{choose}\ M : \forall\, y : (y \in M \equiv \exists\, x \in S : y = t)$

Table 1. Basic set-theoretic operators.

From membership and choice, one can build up the conventional language of mathematics [26], and this is the foundation for the expressiveness of TLA$^+$. Table 1 lists some of the basic set-theoretic constructs of TLA$^+$; we write

$$\{e_1, \ldots, e_n\} \triangleq \textsc{choose}\ S : \forall\, x : (x \in S \;\equiv\; x = e_1 \vee \ldots \vee x = e_n)$$

to denote set enumeration and assume the additional bound variables in the defining expressions of table 1 to be chosen such that no variable clashes occur. The two comprehension schemes act as binders for variable x, which must not have free occurrences in S. These constructions make heavy use of the choice operator, and some care has to be taken to justify such constructions in order to avoid paradoxes. For example, the expression

$$\textsc{choose}\ S : \forall\, x : (x \in S \equiv x \notin x)$$

is a well-formed constant formula of TLA$^+$, but the existence of a set S containing precisely those sets that do not contain themselves would lead to the contradiction that $S \in S$ iff $S \notin S$; this is of course Russell's paradox. Intuitively, S is "too big" to be a set. More precisely, the universe of set theory does not contain collections that are in bijection with the collection of all sets. Therefore, the above TLA$^+$ expression denotes some value, but we do not know which one, the precise value depending on the specific choice function supplied by the interpretation. Perhaps unexpectedly, we can however infer from (12) that

$$(\text{CHOOSE } S : \forall x : (x \in S \equiv x \notin x)) = (\text{CHOOSE } x : x \in \{\})$$

because in both expressions, the choice function is applied to the empty collection. Similarly, a generalized intersection operator dual to the union operator of table 1 cannot be sensibly defined, because the intersection of the empty set would have to produce the set of all sets, which we know cannot exist.

The well-definedness of the constructions appearing in table 1 can be formally justified using the axioms of Zermelo-Fränkel set theory [35], which provide the deductive counterpart to the semantics underlying TLA$^+$.

In a similar vein, because *Message* denotes a set and no set can contain all values, the definition

$$\mathit{NoMsg} \;\triangleq\; \text{CHOOSE } x : x \notin \mathit{Message}$$

that appears in figure 1(b) is sensible: *NoMsg* is certain to denote some value that is not contained in (the interpretation of) *Message*.

4.2 More data structures

Some sets can conveniently be interpreted as functions. A traditional approach is to construct functions via products and relations; this is the case in Z and B [7, 34]. TLA$^+$ does not prescribe any concrete construction of functions. The set of functions whose domain equals S and whose codomain is a subset of T is denoted by $[S \rightarrow T]$, the domain of a function f is denoted by $\text{DOMAIN}\, f$, and the application of function f to an expression e is written as $f[e]$. The construct $[x \in S \mapsto e]$ denotes the function with domain S that maps any $x \in S$ to e; again, the variable x must not occur in S and is bound by the function constructor. Therefore, any function f obeys the law

$$f \;=\; [x \in \text{DOMAIN}\, f \mapsto f[x]] \tag{13}$$

and this equation can in fact serve as a characteristic predicate for functional values. TLA$^+$ introduces a notation for overriding a function at a certain argument position (a similar concept underlies Gurevich's ASM notation [11]). Formally,

$$[f \text{ EXCEPT } ![t] = u] \;\triangleq\; [x \in \text{DOMAIN}\, f \mapsto \text{IF } x = t \text{ THEN } u \text{ ELSE } f[x]]$$

where x is a fresh variable.

Combining choice, sets, and function notation, one obtains an expressive language for defining mathematical structures. For example, the standard TLA$^+$ module providing natural numbers defines them as an arbitrary set with constant zero and successor function satisfying the usual Peano axioms [25, p. 345], and Lamport goes on to similarly define the integers and the real numbers, ensuring that the basic arithmetic operations agree rather than having to overload the operation symbols.

Recursive definitions can be introduced in terms of choice, e.g.

$$\mathit{factorial} \triangleq \text{CHOOSE } f : f = [n \in \mathit{Nat} \mapsto \text{IF } n = 0 \text{ THEN } 1 \text{ ELSE } n * f[n-1]]$$

which TLA$^+$, using some syntactic sugar, allows to write even more concisely as

$$\mathit{factorial}[n \in \mathit{Nat}] \triangleq \text{IF } n = 0 \text{ THEN } 1 \text{ ELSE } n * \mathit{factorial}[n-1]$$

Of course, as with any construction based on choice, this definition should be justified by proving the existence of a unique function that satisfies the recursive equation.

$\mathit{Seq}(S)$	$\triangleq$	$\text{UNION } \{[1..n] \to S : n \in \mathit{Nat}\}$
$\mathit{Len}(s)$	$\triangleq$	$\text{CHOOSE } n \in \mathit{Nat} : \text{DOMAIN } s = 1..n$
$\mathit{Head}(s)$	$\triangleq$	$s[1]$
$\mathit{Tail}(s)$	$\triangleq$	$[i \in 1..\mathit{Len}(s) - 1 \mapsto s[i+1]]$
$s \circ t$	$\triangleq$	$[i \in 1..\mathit{Len}(s) + \mathit{Len}(t) \mapsto$
		$\text{IF } i \leq \mathit{Len}(s) \text{ THEN } s[i] \text{ ELSE } t[i - \mathit{Len}(s)]]$
$\mathit{Append}(s, e)$	$\triangleq$	$s \circ \langle e \rangle$

Fig. 3. Finite sequences.

Tuples are represented in TLA$^+$ as functions:

$$\langle t_1, \ldots, t_n \rangle \triangleq [i \in 1..n \mapsto \text{IF } i = 1 \text{ THEN } t_1 \ \ldots \ \text{ELSE } t_n]$$

where $1..n$ denotes the set $\{j \in \mathit{Nat} : 1 \leq j \wedge j \leq n\}$ (and i is a "fresh" variable), and selection of the i-th element is just function application. Records are similarly represented as functions whose domains are finite sets of strings. The update operation on functions can thus be applied to tuples and records as well. The standard TLA$^+$ module *Sequences* imported by the specification of the FIFO queue in figure 1(b) represents finite sequences as tuples. The definitions of the standard operations, some of which are shown in figure 3, is therefore quite simple. However, this simplicity can sometimes be deceptive. For example, these definitions do not indicate that the *Head* and *Tail* operations are "partial". They should be validated by proving the expected properties, such as

$$\forall s \in \mathit{Seq}(S) : \mathit{Len}(s) \geq 1 \Rightarrow s = \langle \mathit{Head}(s) \rangle \circ \mathit{Tail}(s)$$

5 CONCLUSIONS

The design of software systems requires a combination of ingenuity and careful engineering. While there is no substitute for intuition, the correctness of a proposed solution can be checked by precise reasoning over a suitable model, and this is the realm of logics and (formalized) mathematics. Any formalism should *aid* the user in the difficult and important activity of writing and analysing formal models. TLA$^+$ builds on the experience of classical mathematics and adds just a thin layer of temporal logic in order to describe executions as sets of traces. A distinctive feature of TLA is its attention to refinement and composition, reflected in the concept of stuttering invariance.

Whereas the expressiveness of TLA$^+$ undoubtedly helps in writing concise, high-level models of systems, one may wonder whether it lends itself as well to the analysis of these models. For example, we have pointed out several times the need to prove conditions of "well-definedness" related to the use of the choice operator. On the other hand, the formal basis of TLA$^+$ is the same one that underlies ordinary mathematics, namely Zermelo-Fränkel set theory, and there are well-known idioms, such as primitive-recursive definitions, that ensure well-definedness. Similarly, there are TLA idioms that control the delicate interplay between temporal operators, e.g. in order to ensure that a specification is machine closed [2].

Reasoning about TLA$^+$ specifications can be supported by proof assistants, and in fact several encodings of TLA in the logical frameworks of different theorem provers have been proposed [14, 18, 29], although no prover has yet been designed to support full TLA$^+$. Perhaps more surprisingly, there has been much recent activity on developing a toolset based on the TLC model checker and simulator to aid in validating and debugging TLA$^+$ models [37], and this has found use in industrial development projects. Obviously, model checking is possible only for a sublanguage of TLA$^+$, but interestingly, most real-world specifications are either written in this sublanguage or can be translated into it using minor transformations. The modeling language of TLC is still much more expressive that that of most other model checkers and therefore enables users to write concise system specifications.

Acknowledgements. I am grateful to Leslie Lamport for providing the subject of this article, for his encouragement of this work, and for detailed comments on a draft version. Parts of this paper have been adapted from an earlier paper on TLA, written with Martín Abadi [5]. Júlia Zappe has also proof-read a draft of this paper.

REFERENCES

[1] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 81(2):253–284, May 1991.

[2] Martín Abadi and Leslie Lamport. An old-fashioned recipe for real time. *ACM Transactions on Programming Languages and Systems*, 16(5):1543–1571, September 1994.

[3] Martín Abadi and Leslie Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, May 1995.

[4] Martín Abadi and Stephan Merz. An abstract account of composition. In Jiří Wiedermann and Petr Hajek, editors, *Mathematical Foundations of Computer Science*, volume 969 of *Lecture Notes in Computer Science*, pages 499–508, Prague, Czech Republic, 1995. Springer-Verlag.

[5] Martín Abadi and Stephan Merz. On TLA as a logic. In Manfred Broy, editor, *Deductive Program Design*, NATO ASI series F, pages 235–272. Springer-Verlag, Berlin, 1996.

[6] Martín Abadi and Gordon Plotkin. A logical view of composition. *Theoretical Computer Science*, 114(1):3–30, June 1993.

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

[8] J.-R. Abrial. Extending B without changing it (for developing distributed systems). In H. Habrias, editor, *1st Conference on the B method*, pages 169–190. IRIN Institut de recherche en informatique de Nantes, 1996.

[9] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.

[10] R. Back and J. von Wright. *Refinement calculus—A systematic introduction*. Springer-Verlag, 1998.

[11] Egon Börger and Robert Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.

[12] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, Cambridge, MA, 1999.

[13] W.-P. de Roever, H. Langmaack, and A. Pnueli, editors. *Compositionality: The Significant Difference*, volume 1536 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[14] Urban Engberg, Peter Gronning, and Leslie Lamport. Mechanical verification of concurrent systems with TLA. In *Fourth Intl. Conf. Computer-Aided Verification (CAV '92)*, volume 663 of *Lecture Notes in Computer Science*, pages 44–55. Springer-Verlag, 1992.

[15] M. C. J. Gordon. Mechanizing programming logics in higher-order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*. Springer-Verlag, 1989.

[16] C. A. R. Hoare. A theory of conjunction and concurrency. In *Parallel Architectures (Parbase '90)*, pages 18–27. IEEE Computer Security Press, 1991.

[17] G. E. Hughes and M. J. Cresswell. *A New Introduction to Modal Logic*. Routledge, London, 1996.

[18] Sara Kalvala. A formulation of TLA in Isabelle. Available at `ftp://ftp.dcs.warwick.ac.uk/people/Sara.Kalvala/tla.dvi`, March 1995.

[19] H. W. Kamp. *Tense Logic and the Theory of Linear Order*. PhD thesis, Univ. of California at Los Angeles, 1968.

[20] Leslie Lamport. The tla home page. `http://www.research.microsoft.com/users/lamport/tla/tla.html`.

[21] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, March 1977.

[22] Leslie Lamport. What good is temporal logic? In R. E. A. Mason, editor, *Information Processing 83: Proceedings of the IFIP 9th World Congress*, pages 657–668, Paris, September 1983. IFIP, North-Holland.

[23] Leslie Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

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

[25] Leslie Lamport. *Specifying Systems*. Addison-Wesley, Boston, Mass., 2002.

[26] A. C. Leisenring. *Mathematical Logic and Hilbert's ε-Symbol*. University Mathematical Series. Macdonald & Co. Ltd., London, U.K., 1969.

[27] Zohar Manna and Amir Pnueli. Verification of concurrent programs: the temporal framework. In R.S. Boyer and J.S. Moore, editors, *The Correctness Problem in Computer Science*, pages 215–273. Academic Press, London, 1982.

[28] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems—Specification*. Springer-Verlag, New York, 1992.

[29] Stephan Merz. Isabelle/TLA. Available on the WWW at `http://isabelle.in.tum.de/library/HOL/TLA`, 1997. Revised 1999.

[30] Stephan Merz. A more complete TLA. In J.M. 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, September 1999. Springer-Verlag.

[31] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on the Foundations of Computer Science*, pages 46–57. IEEE, November 1977.

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

[33] A. P. Sistla, E. M. Clarke, N. Francez, and Y. Gurevich. Can message buffers be characterized in linear temporal logic? *Information and Control*, 63:88–112, 1984.

[34] M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 1992.

[35] Patrick Suppes. *Axiomatic Set Theory*. Dover Publications, 1972.

[36] Moshe Vardi. Branching vs. linear time: Final showdown. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001)*, volume 2031 of *Lecture Notes in Computer Science*, pages 1–22, Genova, Italy, 2001. Springer-Verlag. See `http://www.cs.rice.edu/~vardi/papers/` for more recent versions of this paper.

[37] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking TLA+ specifications. In L. Pierre and T. Kropf, editors, *Correct Hardware Design and Verification Methods (CHARME'99)*, volume 1703 of *Lecture Notes in Computer Science*, pages 54–66, Bad Herrenalb, Germany, 1999. Springer-Verlag.