A User's Guide to TLA

Stephan Merz
Institut für Informatik, Universität München, Germany
merz@informatik.uni-muenchen.de

1 The Context of TLA

Temporal logics have found their way into computer science in the second half of the 70's [29] and have since found wide-spread acceptance as a formal language for describing reactive systems. A large body of literature [26, 11, 15, 30, 7] discusses formalisms and (manual as well as machine-supported) proof techniques for the modelling and analysis of systems.

From a logical point of view, a system is represented as a semantical *model*, and system properties are expressed as *formulas* in an appropriate logic, which can be interpreted over the semantical models. The fundamental semantic structures used to represent reactive systems are *transition systems*. More concretely, transition systems are usually described by modelling languages such as process algebras (e.g., CCS or CSP), protocol specification languages (e.g., Lotos or SDL), notations for state machines (e.g., StateCharts or I/O automata), synchronous languages (e.g., Esterel or Estelle) or (idealized) programming languages of the kind frequently found in textbooks. These languages differ mainly in the programming idioms they naturally support; they are sometimes deliberately restricted in order to facilitate automatic analysis.

Logical languages that are well-suited to express properties of transition systems or of the runs they generate include various flavors of temporal logic as well as related formalisms such as multimodal ("Hennessy-Milner") logic, the μ-calculus, or appropriate languages of first or second order predicate logic.

Lamport's Temporal Logic of Actions (TLA) provides the means for the description of transition systems (i.e., states and state transitions) and the formulation of properties in a single logical formalism, equipped with a formal semantics and proof rules for reasoning about systems. The principal advantage of a unified language for both systems and properties is that one can not only talk about a single system and its properties, but that one can relate several systems. This is necessary when comparing descriptions of a system at different levels of abstraction and, e.g., proving that a lower-level model *implements* a higher-level model. In particular, one needs principles that support large systems to be implemented by a collection of components, which are then developed and analysed separately. Goals of TLA are

- to provide a minimal set of orthogonal concepts that allow systems to be described and analysed,
- to represent structural concepts such as composition or implementation in an appropriate logic, and
- to reduce the reasoning about systems to ordinary mathematical reasoning, as far as possible.

The purpose of these notes is to provide a tutorial introduction to the spirit and the practical use of TLA, without striving for completeness. Lamport's original description of TLA [22] is the definitive reference paper that includes much additional material. There are many other papers that give a more formal definition of the logic [5], discuss mechanical verification in TLA [10, 16, 27], refinement and composition [6, 3], real-time and hybrid systems [2, 21], and case studies [4, 18]. Lamport has also introduced TLA$^+$, a complete specification language based on the logic TLA that includes concrete syntax for definitions, a module structure, and provides a built-in set theory for the static aspects of system descriptions. Most papers about TLA and its use for system specifications are accessible via the TLA home page on the World Wide Web [19].

2 Specification of a Component: A Lossy Queue

In this tutorial, we will introduce TLA step by step at the hand of small examples. Suppose we want to specify a simple queue, as illustrated in figure 1.

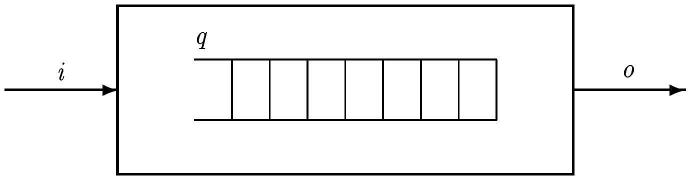

Figure 1: A lossy queue.

The queue accepts inputs and produces outputs at the interface "wires" i and o; these are the only state components visible to an outside observer (or user) of the queue. Internally, the queue stores the values as an (unbounded) sequence q. For the time being, we do not impose an acknowledgement protocol, and therefore inputs may be enqueued several times, or they may be lost.

In TLA, a component specification describes a state machine and consists of an initial condition, the next-state relation, and fairness requirements, which ensure that certain actions actually occur. Our TLA specification of the lossy queue appears as formula $LQSpec$ in figure 2; we now explain it in more detail.

$$
\begin{array}{lcl}
LQInit & \triangleq & q = \langle\,\rangle \wedge i = o \\
LQEnq & \triangleq & q' = q \circ \langle i' \rangle \wedge o' = o \\
LQDeq & \triangleq & q \neq \langle\,\rangle \wedge o' = head(q) \wedge q' = tail(q) \wedge i' = i \\
LQNext & \triangleq & LQEnq \vee LQDeq \\
LQLive & \triangleq & \mathrm{WF}_{q,o}(LQDeq) \\
LQISpec & \triangleq & LQInit \wedge \Box[LQNext]_{q,o} \wedge LQLive \\
LQSpec & \triangleq & \exists\!\!\exists\, q : LQISpec
\end{array}
$$

Figure 2: A lossy queue.

Formula $LQISpec$ represents the "inner specification" of the lossy queue, with the internal sequence q visible. It contains the flexible variables i, o, and q that represent the state components of the queue shown in figure 1. In temporal logics such as TLA, the dynamics of a system is modelled by different valuations of flexible variables at different states. Formula $LQISpec$ also contains standard function symbols for sequences: $\langle\,\rangle$ denotes the empty sequence, $\langle x \rangle$ stands for the one-element sequence that contains just x, the function $\circ$ (used with infix notation) concatenates sequences, and $head$ and $tail$ yield the first element of a sequence and the sequence obtained by removing the first element.

The formula $LQInit$ describes the initial state of the component. It asserts that (the values assigned to) i and o are equal and that (the value of) q is the empty sequence.

Formulas such as $LQInit$ are called *state predicates* in TLA because they describe single states. Syntactically, state predicates are ordinary (first-order) formulas that may contain flexible variables.

The formulas $LQEnq$, $LQDeq$, and $LQNext$ are *transition predicates*, or *actions*, and describe pairs of states. Syntactically, actions are like formulas in predicate logics, but they may contain primed occurrences of variables such as q' as well as ordinary, unprimed occurrences[1]. The idea is that a primed occurrence of a variable x refers to the value of x in the second state of a pair, whereas an unprimed occurrence refers to the value of x in the first state. So, $LQEnq$ describes an "enqueue" action where the new value of the input wire is appended to the queue. Similarly, $LQDeq$ models a "dequeue" action: it requires that the queue is nonempty at the state before the action occurs, and that the new values of the output wire and the queue are given by the head and the tail of the old queue. Observe that the $LQEnq$ and $LQDeq$ actions assert that the "unaffected" interface wire does not change. This is typical of interleaving specifications where the environment and the component never perform an action at the same time. Finally, the action $LQNext$, which represents the next-state relation of the lossy queue, is simply the disjunction of $LQEnq$ and $LQDeq$: it is true of a pair of states if one of $LQEnq$ or $LQDeq$ is.

We now turn to the temporal formula $LQISpec$. Ignoring the fairness requirement $LQLive$ for the moment, it asserts $LQInit \wedge \Box[LQNext]_{q,o}$. This formula holds of an infinite sequence $\sigma = s_0 s_1 \ldots$ of states (a *behavior*) if both $LQInit$ and $\Box[LQNext]_{q,o}$ hold of σ. A state predicate holds of σ iff (if and only if) it holds of the first state of σ, so the first conjunct requires $LQInit$ to hold initially, as desired. The second conjunct asserts that for every step $\langle s_i, s_{i+1} \rangle$ in σ, either both q and o are unchanged or the action $LQNext$ holds of the step. In general, when N is an action and v is a tuple of variables, then $\Box[N]_v$ is a temporal formula that asserts that at every step, either all variables in v keep their values, or N holds.

Observe that the formula $LQInit \wedge \Box[LQNext]_{q,o}$ holds of a behavior where $LQInit$ holds initially and neither q nor o ever change. Similarly, the formula is satisfied if the queue performs enqueues, but never dequeues a value. For the sake of presentation, suppose that we want to require that the queue should eventually dequeue any value it has stored. We can do so in TLA by asserting weak fairness of the $LQDeq$ action. Weak fairness of an action A means that A will eventually happen whenever it is persistently enabled. Similarly, strong fairness for A asserts that A will eventually happen whenever it is recurrently enabled. In the context of the queue example, the two conditions are equivalent for $LQDeq$ because only an occurrence of $LQDeq$ can disable the action. Strong fairness conditions are, for example, usually associated with semaphores. In TLA, weak and strong fairness requirements for an action A are expressed by formulas $\mathrm{WF}_v(A)$ and $\mathrm{SF}_v(A)$, which are actually abbreviations:

$$\begin{aligned} \mathrm{WF}_v(A) &\triangleq \Diamond\Box\textsc{Enabled}\,\langle A \rangle_v \Rightarrow \Box\Diamond\langle A \rangle_v \\ \mathrm{SF}_v(A) &\triangleq \Box\Diamond\textsc{Enabled}\,\langle A \rangle_v \Rightarrow \Box\Diamond\langle A \rangle_v \end{aligned}$$

The formula $\Diamond\langle A \rangle_v$, which abbreviates $\neg\Box[\neg A]_v$, asserts that some future step in the behavior satisfies $A \wedge (v' \neq v)$. The state predicate $\textsc{Enabled}\,\langle A \rangle_v$ is true in a state s iff there exists some state t such that the pair $\langle s, t \rangle$ satisfies $A \wedge (v' \neq v)$. Syntactically, $\textsc{Enabled}\,\langle A \rangle_v$ can be defined as[2]

$$\textsc{Enabled}\,\langle A \rangle_v \triangleq \exists\, v_1', \ldots, v_n' : A \wedge (v' \neq v)$$

where $v_1', \ldots, v_n'$ are all variables that have primed occurrences in $A \wedge (v' \neq v)$.

Finally, the specification $LQSpec$ of the lossy queue is obtained from $LQISpec$ by quantifying over q. The formula $\boldsymbol{\exists}\, x : F$ holds of a behavior σ if there exists some

[1] In particular, every state predicate can also be regarded as an action.

[2] Observe that the definition of $\textsc{Enabled}\,\langle A \rangle_v$ actually gives a state predicate because it does not contain any free occurrences of primed flexible variables.

behavior τ that satisfies F and differs from σ at most by the values assigned to x at each state. (Strictly speaking, this condition is only sufficient for $\boldsymbol{\exists}\, x \; : \; F$ to hold of σ. The precise definition [22, 5] is somewhat subtle, in order to make TLA formulas invariant under stuttering.) The point is that one chooses not just a single different value for x, but an entire sequence of values. We indicate this by using a bold existential quantifier $\boldsymbol{\exists}$ for the quantification over flexible variables at the level of temporal formulas, in contrast to the ordinary first-order quantifier $\exists$.

The most important observation about TLA specifications, in contrast to many other TL-based formalisms, is they are very close to the description of a state machine. Writing a TLA specification is therefore often quite similar to writing a corresponding piece of code, or to writing a specification in a formalism based on automata such as I/O automata [24]. This similarity can, however, be deceptive: TLA formulas have a precise meaning, properties of the system are expressed in the same language, and one can manipulate formulas with logical rules; there is no concern about specifications being "executable".

3 Specification Styles

Specification $LQSpec$ is not very useful: it lacks a communication protocol between the queue and its environment, and therefore values may be lost. In the following section, we will discuss alternative queue specifications and thereby show how different communication paradigms can be expressed in TLA. In either case, we assume for simplicity that the sender never wishes to enqueue the same value twice; in practice, this could be achieved by adding a bit to the input line that flips with every new value transmitted.

3.1 Synchronous Communication, Interleaving

With synchronous communication, the receiver observes every change of state of the communication channel, and it may immediately record the value transmitted in some internal state component. (In an interleaving model, the input and output wires of the queue should, however, not change at the same time.) Figure 3 contains the specification of a queue with synchronous input.

$$
\begin{array}{lcl}
SIQInit & \triangleq & q = \langle\,\rangle \wedge i = o \\
SIQEnq & \triangleq & i' \neq i \wedge q' = q \circ \langle i' \rangle \wedge o' = o \\
SIQDeq & \triangleq & q \neq \langle\,\rangle \wedge o' = head(q) \wedge q' = tail(q) \wedge i' = i \\
SIQNext & \triangleq & SIQEnq \vee SIQDeq \\
SIQLive & \triangleq & \mathrm{WF}_{q,i,o}(SIQDeq) \\
SIQISpec & \triangleq & SIQInit \wedge \Box[SIQNext]_{q,i,o} \wedge SIQLive \\
SIQSpec & \triangleq & \boldsymbol{\exists}\, q \; : \; SIQISpec
\end{array}
$$

Figure 3: A queue with synchronous input.

The form of the specification is very similar to that of the lossy queue. However, the input wire i now appears as a subscript in the next-state relation of the synchronous queue. This ensures that the queue "reacts" appropriately (namely, with an $SIQEnq$ action) to any change of i. We have also added the conjunct $i' \neq i$ to $SIQEnq$, which is now supposed to happen if and only if the value of the input wire changes.

A TLA formula F characterizes the set of its *models* $[\![F]\!]$, i.e. the behaviors of which it holds. In formalisms based on linear time, a specification Φ implements another specification Ψ iff every behavior allowed by Φ is also allowed by Ψ, that is, iff $[\![\Phi]\!] \subseteq [\![\Psi]\!]$ holds or, in logical terms, iff the implication $\Phi \Rightarrow \Psi$ is valid. Obviously, the formulas defining the initial condition and the actions of the synchronous queue imply the corresponding formulas for the lossy queue; this is also true for the fairness condition. Therefore, we may trivially conclude that the synchronous queue is an implementation of the lossy queue: the latter specification is rather loose and allows more behaviors.

3.2 Synchronous Communication, Noninterleaving

Formula $SIQSpec$ allows that an environment step and a component step occur at the same time, provided the component step changes only internal state components. If we relax this requirement, we obtain a specification in noninterleaving style, where environment and component are allowed to move simultaneously. There has been many a heated debate over interleaving vs. noninterleaving models of reactive systems, and we do not wish to enter this debate. Let us just remark that experience indicates that noninterleaving specifications are often harder to write and to reason about. On the other hand, they compose better than interleaving specifications, where the interleaving assumption often has to be made explicit at the system level. In fact, one way to write noninterleaving specifications is to think of the component as being composed of subprocesses, and to write separate next-state relations per subprocess.

In the case of the queue, we associate a separate process with each state component i, q, and o. The input process is affected by enqueue actions, the output process by dequeue actions, and the queue process, by both. The subprocesses are related by the transition invariant that the concatenation of the old queue value and the value enqueued (if any) should equal the value dequeued (if any) prepended to the new value of the queue. Figure 4 shows a noninterleaving queue specification based on this idea.

$$
\begin{array}{lcl}
SNQInit & \triangleq & q = \langle\,\rangle \wedge i = o \\
di & \triangleq & \textbf{if } i' = i \textbf{ then } \langle\,\rangle \textbf{ else } \langle\, i' \,\rangle \\
do & \triangleq & \textbf{if } o' = o \textbf{ then } \langle\,\rangle \textbf{ else } \langle\, o' \,\rangle \\
SNQEnq & \triangleq & i' \neq i \wedge q \circ di = do \circ q' \\
SNQDeq & \triangleq & q \neq \langle\,\rangle \wedge o' = head(q) \wedge q \circ di = do \circ q' \\
SNQLive & \triangleq & \mathrm{WF}_o(SNQDeq) \\
SNQISpec & \triangleq & SNQInit \wedge \Box[SNQEnq]_i \wedge \Box[SNQEnq \vee SNQDeq]_q \\
 & & \qquad \wedge \Box[SNQDeq]_o \wedge SNQLive \\
SNQSpec & \triangleq & \exists\, q \,:\, SNQISpec
\end{array}
$$

Figure 4: A noninterleaving synchronous queue.

The specification of the noninterleaving queue contains a separate conjunct of the form $\Box[N]_v$ for each of the (conceptual) subprocesses. The fairness condition requires only o to change value; this emphasizes that dequeue actions are accounted for by the output process. The specification allows simultaneous enqueue and dequeue actions provided the queue is nonempty; a value received cannot be output at the same step. However, the specification does not *require* simultaneous actions. In fact, it is not hard to see that the implication $SIQSpec \Rightarrow SNQSpec$ is valid: the interleaving queue is an implementation of the noninterleaving queue.

3.3 Asynchronous Communication

As a last variant of the queue, we model asynchronous communication between the environment and the queue via an explicit handshake protocol: the interface now includes an extra signalling “wire” sig. The environment may send a new value when sig is 0 and then sets sig to 1; the queue resets sig to 0 when it enqueues the value and thereby acknowledges its receipt. The TLA specification is shown in figure 5.

$$
\begin{array}{lcl}
AQInit & \triangleq & q = \langle\, \rangle \land i = o \land sig = 0 \\
AQEnv & \triangleq & sig = 0 \land sig' = 1 \land \text{UNCHANGED } \langle q, o \rangle \\
AQEnq & \triangleq & sig = 1 \land sig' = 0 \land q' = q \circ \langle i' \rangle \land o' = o \\
AQDeq & \triangleq & q \neq \langle\, \rangle \land o' = head(q) \land q' = tail(q) \land \text{UNCHANGED } \langle sig, i \rangle \\
AQNext & \triangleq & AQEnv \lor AQEnq \lor AQDeq \\
AQLive & \triangleq & \mathrm{WF}_{q,i,o,sig}(AQEnq) \land \mathrm{WF}_{q,i,o,sig}(AQDeq) \\
AQISpec & \triangleq & AQInit \land \Box[AQNext]_{q,i,o,sig} \land AQLive \\
AQSpec & \triangleq & \boldsymbol{\exists}\, q : AQISpec
\end{array}
$$

Figure 5: A queue with asynchronous input.

The initial condition contains a conjunct that asserts that sig starts with value 0, so that the environment starts the handshake protocol. More importantly, the specification includes an extra action $AQEnv$, which represents the changes to state components attributed to the environment. (If $v = \langle v_1, \ldots, v_n \rangle$ is a tuple of variables, UNCHANGED v abbreviates the transition formula $v_1' = v_1 \land \ldots \land v_n' = v_n$.) $AQEnv$ allows the environment to send a value on the input wire when sig is 0; of course, the variables q and o, which are controlled by the queue, should remain unchanged. The $AQEnq$ action, which represents the receipt of a value by the queue, is now only enabled if sig is 1. Observe that $AQEnq$ is an action attributed to the component (unlike $LQEnq$, $SIQEnq$ or $SNQEnq$), and therefore we require a fairness condition to ensure that the queue eventually reacts to an input from the environment. We do not require fairness of $AQEnv$, because the environment may choose to become quiescent.

By a similar argument as for the specification $SIQSpec$ of the synchronous queue, the asynchronous queue implements the lossy queue, as the implication $AQSpec \Rightarrow LQSpec$ is valid.

4 The Logic TLA

We have so far introduced TLA informally, at the hand of example specifications. Let us now more formally define its syntax and semantics. Assume given a language $\mathcal{L}_P$ of classical first-order logic that defines function and predicate symbols, as well as a (countably) infinite set $\mathcal{V}$ of variables, partitioned into infinite and disjoint subsets $\mathcal{V}_r$ and $\mathcal{V}_f$ of *rigid* and *flexible* variables. We let $\mathcal{V}_f' = \{x' : x \in \mathcal{V}_f\}$ be the set of primed flexible variables, and call $\mathcal{V}_e = \mathcal{V} \cup \mathcal{V}_f'$ the (extended) set of rigid as well as primed and unprimed flexible variables.

TLA formulas are defined in two tiers: at the nontemporal or transition level and at the temporal level.

4.1 Transition formulas

A *transition function* is a first-order expression built from the predicate and function symbols of $\mathcal{L}_P$ and variables from $\mathcal{V}_e$. A *transition predicate* (or *action*) is a boolean-valued transition function. For example, A defined as $\exists\, q' : q' = q \circ \langle i \rangle$ is an action (for a suitable first-order language). The definitions of free and bound occurrences of variables (from $\mathcal{V}_e$) are the usual ones from first-order logic. For example, the above action contains free occurrences of q and i and bound occurrences of q'. A *state function* (*state predicate*) is a transition function (transition predicate) that contains no free occurrences of variables from $\mathcal{V}_f'$. So, the example formula A shown above is in fact a state predicate. If v is a state function, then v' is an abbreviation for the transition function that results from v by replacing all free occurrences of variables $x \in \mathcal{V}_f$ by their primed counterparts $x' \in \mathcal{V}_f'$, renaming bound variables as necessary to avoid variable capture. For example, A' is $\boldsymbol{\exists}\, r : r = q' \circ \langle i' \rangle$.

The semantics of transition functions is defined in terms of a first-order interpretation for the underlying language $\mathcal{L}_P$. The interpretation defines a universe of values as well as a function and relation (of suitable arity) for each function and predicate symbol in $\mathcal{L}_P$. A *state* is a mapping of flexible variables $x \in \mathcal{V}_f$ to values of the universe.

Transition functions are evaluated over a pair $\langle s, t \rangle$ of states, plus a valuation ξ of the rigid variables. Because we have defined transition functions over an extended set $\mathcal{V}_e$ of variables, it is convenient to combine all three valuations into a single valuation α for $\mathcal{V}_e$ such that $\alpha(x) = \xi(x)$ if x is a rigid variable, $\alpha(x) = s(x)$ if x is an unprimed flexible variable, and $\alpha(x') = t(x')$ if x' is a primed flexible variable. The semantics of a transition function f, written $s[\![f]\!]^{\xi} t$, is then the ordinary first-order semantics of f, evaluated over the extended valuation α. If f is a state function, the second state of the pair is actually irrelevant, and we write simply $s[\![f]\!]^{\xi}$.

4.2 Temporal formulas

The temporal formulas of TLA are inductively defined as follows:

- If P is a state predicate then P is a temporal formula.
- If A is an action and v is a state function then $\Box[A]_v$ is a temporal formula.
- If F and G are temporal formulas then so are $\neg F$ and $F \wedge G$.
- If F is a temporal formula then so is $\Box F$.
- If F is a temporal formula and $x \in \mathcal{V}_r$ is a rigid variable then $\exists\, x : F$ is a temporal formula.
- If F is a temporal formula and $x \in \mathcal{V}_f$ is a flexible variable then $\boldsymbol{\exists}\, x : F$ is a temporal formula.

The important observation to make is that $\Box A$ is well-formed for an action A only if A is a state predicate. (The operator $\Box[A]_v$ counts as different from the ordinary box operator.) This restriction is designed to ensure the invariance of temporal formulas under stuttering, as explained below.

The set of free variables contained in a temporal formula is defined as usual, except that no difference is made between primed and unprimed occurrences of flexible variables. For example, the set of free variables of temporal formula G defined as $\Box[q' = q \circ \langle i' \rangle]_{q,i,o}$ is $\{q, i, o\}$. Similarly, substitution of a state function t for a flexible variable x in a formula F, written $F[t/x]$, replaces all free occurrences of x by t and all free occurrences of x' by t'. So, $G[q_1 \circ q_2/q]$ is the formula

$$\Box[(q_1 \circ q_2)' = (q_1 \circ q_2) \circ \langle i' \rangle]_{q_1 \circ q_2, i, o}$$

The substitution of a transition function for a (primed or unprimed) flexible variable is not defined.

A *behavior* is an infinite sequence of states. If $\sigma = s_0 s_1 \ldots$ is a behavior then $\sigma|_n$ is its suffix $s_n s_{n+1} \ldots$. The finite sequence that consists of the states $s_1, \ldots, s_n$ is written as $\langle s_1, \ldots, s_n \rangle$, and the concatenation of a finite sequence of states ρ and a behavior σ is denoted as $\rho \circ \sigma$.

Stuttering equivalence is the smallest equivalence relation on behaviors such that the behaviors $\rho \circ \langle s \rangle \circ \sigma$ and $\rho \circ \langle s, s \rangle \circ \sigma$ are equivalent. Two behaviors $\sigma = s_0 s_1 \ldots$ and

$\tau = t_0 t_1 \ldots$ are *equal up to* x (for a flexible variable x) if $s_i(y) = t_i(y)$ for every flexible variable y except possibly x. They are *similar up to* x, written $\sigma \simeq_x \tau$, iff there exist behaviors σ' and τ' such that:

- σ and σ' are stuttering equivalent,
- σ' and τ' are equal up to x, and
- τ' and τ are stuttering equivalent.

Given a behavior $\sigma = s_0 s_1 \ldots$ and a valuation ξ for the rigid variables, the semantics $\sigma[\![F]\!]^\xi$ of temporal formulas is inductively defined as follows:

- $\sigma[\![P]\!]^\xi$ is $s_0[\![P]\!]^\xi$, for a state predicate P,
- $\sigma[\![\Box[A]_v]\!]^\xi$ is true iff $s_n[\![A \vee v' = v]\!]^\xi s_{n+1}$ is true for all $n \geq 0$,
- $\sigma[\![\neg F]\!]^\xi$ is true iff $\sigma[\![F]\!]^\xi$ is false,
- $\sigma[\![F \wedge G]\!]^\xi$ is true iff both $\sigma[\![F]\!]^\xi$ and $\sigma[\![G]\!]^\xi$ are,
- $\sigma[\![\Box F]\!]^\xi$ is true iff $\sigma|_n[\![F]\!]^\xi$ is true for all $n \geq 0$,
- $\sigma[\![\exists\, x : F]\!]^\xi$ is true iff $\sigma[\![F]\!]^\zeta$ is true for some valuation ζ such that $\zeta(y) = \xi(y)$ for all rigid variables y except possibly x, and
- $\sigma[\![\boldsymbol{\exists}\, x : F]\!]^\xi$ is true iff $\tau[\![F]\!]^\xi$ is true for some behavior τ such that $\tau \simeq_x \sigma$.

Except for the definition of flexible quantification, which explicitly allows for the insertion or removal of stuttering steps, these semantic definitions are the usual ones for linear-time temporal logics. The restrictions on the syntax of temporal formulas and the complicated definition of the semantics of $\boldsymbol{\exists}\, x : F$ help to ensure that TLA formulas are invariant under stuttering, as asserted in the following proposition, which is proved in [5].

Proposition 1 *If F is a temporal formula, ξ is a valuation of rigid variables, and σ and τ are two stuttering equivalent behaviors then $\sigma[\![F]\!]^\xi$ is true iff $\tau[\![F]\!]^\xi$ is true.*

5 Properties and Refinements

Sections 2 and 3 introduced the way one *describes* systems as TLA formulas, and we have also indicated that some of the example specifications were refinements of others. We will now turn to proof rules that can be used to prove that a specification satisfies some property or that one specification refines another one. Again, we do not attempt to give a complete set of rules; our purpose is to illustrate the TLA approach to formal reasoning about specifications.

5.1 Invariants

The first step in any deduction-based approach to system verification is to establish useful (state) invariants, which provide information about the set of reachable states. In TLA, an invariant has the form $\Box I$, where I is a state predicate, and the formula $\mathit{Spec} \Rightarrow \Box I$ is valid iff the specification represented by *Spec* satisfies invariant I. For example, the synchronous queue has the invariant that any two consecutive elements of the sequence formed by prepending the current value of the output wire to the internal queue are different, and that the last element of that sequence equals the current value of the input wire.

This property is expressed as $\Box P$, where P is the state predicate[3]

$$
\begin{array}{l}
\wedge\ (\langle\, o\, \rangle \circ q)[\mathit{len}(\langle\, o\, \rangle \circ q)] = i \\
\wedge\ \forall\, k\ :\ k \in I\!N \wedge 1 \leq k \wedge k < \mathit{len}(\langle\, o\, \rangle \circ q) \Rightarrow (\langle\, o\, \rangle \circ q)[k] \neq (\langle\, o\, \rangle \circ q)[k+1]
\end{array}
$$

P is written in a first-order language where $\mathit{len}(s)$ denotes the length of a finite sequence such that $\mathit{len}(\langle\, s_1, \ldots, s_n \,\rangle) = n$, and where $s[k]$ denotes the k-th element of s, for $1 \leq k \leq \mathit{len}(s)$; in particular, $\mathit{head}(s)$ equals $s[1]$ for a nonempty sequence s.

The basic TLA rule for proving an invariant is

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

where I is a state predicate, N is an action, and v is a state function. It asserts that whenever the premise $I \wedge (N \vee v' = v) \Rightarrow I'$ is a valid transition formula (over a given class of first-order interpretations), then the conclusion $I \wedge \Box[N]_v \Rightarrow \Box I$ is a valid temporal formula (over all temporal structures based on these interpretations). The strategy to reduce temporal assertions to transition-level proof obligations is central to the TLA approach, as it enables the use of established proof rules and technology for the greater part of system verification. In fact, some 90% or more of the overall verification effort will be spent on transition-level reasoning, for typical, medium-size examples.

Let us illustrate the rule (INV1) by proving that the synchronous queue described by formula $\mathit{SIQISpec}$ satisfies the above invariant P. We present the proof in a style suggested by Lamport [20] that emphasizes the hierarchical structure of a proof. Such a proof is best read in top-down manner: the lower-level steps should be read only if the higher levels have been checked. The reader is invited to verify the details of the transition-level proofs, which we omit here, assuming the necessary "data axioms" about finite sequences and to work out a similar invariant for the asynchronous queue.

1. $\mathit{SIQInit} \Rightarrow P$
   PROOF: Immediately from the definition of $\mathit{SIQInit}$ and data axioms.
2. $P \wedge [\mathit{SIQNext}]_{q,i,o} \Rightarrow P'$
   2.1. $P \wedge \mathit{SIQEnq} \Rightarrow P'$
      PROOF: Definition of SIQEnq and data axioms.
   2.2. $P \wedge \mathit{SIQDeq} \Rightarrow P'$
      PROOF: Definition of SIQDeq and data axioms.
   2.3. $P \wedge \text{UNCHANGED}\ \langle\, q, i, o \,\rangle \Rightarrow P'$
      PROOF: Simple predicate logic.
   2.4. Q.E.D.
      PROOF: From steps 2.1– 2.3 by the definition of $[\mathit{SIQNext}]_{q,i,o}$ and propositional logic.
3. Q.E.D.
   PROOF: Step 2 and rule (INV1) imply $P \wedge \Box[\mathit{SIQNext}]_{q,i,o} \Rightarrow \Box P$. Using step 1 and propositional reasoning, we conclude $\mathit{SIQInit} \wedge \Box[\mathit{SIQNext}]_{q,i,o} \Rightarrow \Box P$, and therefore the assertion.

Rule (INV1) is readily generalized to a rule that handles several transition formulas in the antecedent:

$$
\frac{I \wedge [N_1]_{v_1} \wedge \ldots \wedge [N_k]_{v_k} \Rightarrow I'}{I \wedge \Box[N_1]_{v_1} \wedge \ldots \wedge \Box[N_k]_{v_k} \Rightarrow \Box I} \qquad (\text{INV1}_k)
$$

[3] Here and in the following, we write multi-line conjunctions or disjunctions as lists of formulas prefaced with $\wedge$ or $\vee$, and we use indentation to eliminate parentheses.

5.2 Step Simulation

A central step in the proof that a lower-level specification C implements (implies) a higher-level specification A is to prove a formula of the form $C \Rightarrow \Box[N]_v$, where N is the next-state relation of A. This formula asserts that every step of the lower-level state machine simulates a step of the higher-level one; it can be proved using the rule

$$\frac{I \wedge I' \wedge [M]_u \Rightarrow [N]_v}{\Box I \wedge \Box[M]_u \Rightarrow \Box[N]_v} \qquad \text{(TLA2)}$$

To prove step simulation, one may use an invariant I as well as the next-state relation M of the lower-level specification. On the other hand, it is not usually necessary to use the lower-level fairness conditions, as the liveness part of a specification should not restrict its safety part. (Technically, this condition is referred to as *machine closure*.)

For example, the following proof shows the validity of $\mathit{SIQISpec} \Rightarrow \Box[\mathit{SNQEnq}]_i$, which is required to prove that the synchronous interleaving queue implements the noninterleaving variant. Because this example is so simple, it does not need an extra invariant, so we let I equal TRUE in rule (TLA2).

1. $[\mathit{SIQNext}]_{q,i,o} \Rightarrow [\mathit{SNQEnq}]_i$
  1.1. $\mathit{SIQEnq} \Rightarrow [\mathit{SNQEnq}]_i$
    1.1.1. $\mathit{SIQEnq} \Rightarrow di = \langle i' \rangle$
      PROOF: Definitions of SIQEnq and di.
    1.1.2. $\mathit{SIQEnq} \Rightarrow do = \langle \rangle$
      PROOF: Definitions of SIQEnq and do.
    1.1.3. $\mathit{SIQEnq} \Rightarrow q \circ di = do \circ q'$
      PROOF: Steps 1.1.1 and 1.1.2 and definition of SIQEnq.
    1.1.4. Q.E.D.
      PROOF: Step 1.1.3 and the definitions of SIQEnq and SNQEnq prove $\mathit{SIQEnq} \Rightarrow \mathit{SNQEnq}$ and therefore the assertion.
  1.2. $\mathit{SIQDeq} \Rightarrow [\mathit{SNQEnq}]_i$
    PROOF: The definition of SIQDeq implies $i' = i$ and therefore the assertion, by propositional logic.
  1.3. UNCHANGED $\langle q, i, o \rangle \Rightarrow [\mathit{SNQEnq}]_i$
    PROOF: Simple predicate logic.
  1.4. Q.E.D.
    PROOF: Steps 1.1– 1.3, the definition of $[\mathit{SIQNext}]_{q,i,o}$, and propositional logic.
2. Q.E.D.
  PROOF: Step 1 and rule (TLA2) imply $\Box[\mathit{SIQNext}]_{q,i,o} \Rightarrow \Box[\mathit{SNQEnq}]_i$, and the assertion follows by the definition of $\mathit{SIQISpec}$.

Again, rule (TLA2) has a straightforward generalization for the case where several transition formulas appear in the antecedent:

$$\frac{I \wedge I' \wedge [M_1]_{u_1} \wedge [M_k]_{u_k} \Rightarrow [N]_v}{\Box I \wedge \Box[M_1]_{u_1} \wedge \ldots \wedge \Box[M_k]_{u_k} \Rightarrow \Box[N]_v} \qquad (\text{TLA2}_k)$$

5.3 Simple Temporal Logic

Because TLA is a linear-time temporal logic, its temporal formulas satisfy standard axioms and rules of LTL without the next-time operator [15, 26]. Lamport [22] lists the following rules:

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

From these basic rules, many more theorems can be derived, which can be useful in verification. For example, these include

$$\Box\Diamond\Box F \equiv \Diamond\Box F \qquad \Box(F \Rightarrow G) \Rightarrow (F \leadsto G)$$

$$(F \leadsto G) \wedge (G \leadsto H) \Rightarrow (F \leadsto H) \qquad \frac{\Box F \wedge H \Rightarrow \Box H \qquad \Box F \Rightarrow (G \leadsto H)}{\Box F \wedge \Diamond\Box G \Rightarrow \Diamond\Box H}$$

The rule shown in the lower right-hand column is Ladkin's "Bridge Rule", discussed in [17].

5.4 Liveness from Fairness

The proof of liveness properties relies on the fairness conditions associated with a specification. TLA contains rules to deduce elementary liveness properties directly from fairness conditions; more complex liveness properties can then be obtained using general temporal logic rules such as the method of well-founded orderings, discussed in section 5.5.

The following rule allows to deduce a simple leadsto-formula from a weak fairness condition; there is an analogous rule (SF1) for strong fairness conditions [22].

$$\frac{\begin{array}{l} I \wedge I' \wedge P \wedge [N]_v \Rightarrow P' \vee Q' \\ I \wedge I' \wedge P \wedge \langle N \wedge A \rangle_v \Rightarrow Q' \\ I \wedge P \Rightarrow \textsc{Enabled}\,\langle A \rangle_v \end{array}}{\Box I \wedge \Box[N]_v \wedge \mathrm{WF}_v(A) \Rightarrow (P \leadsto Q)} \qquad \text{(WF1)}$$

The rule asserts that $P \leadsto Q$ holds for a specification with next-state relation N, invariant I, and weak fairness condition for some action A, provided we can prove the following for every reachable state (characterized by I) that satisfies P:

- every successor state (via $[N]_v$) satisfies P or Q,
- every successor state reached via an $\langle A \rangle_v$ action satisfies Q, and
- $\langle A \rangle_v$ is enabled.

Observe that all premises of rule (WF1) are again nontemporal formulas.

To illustrate rule (WF1) at the hand of the queue, we prove that every element stored in the queue will eventually be dequeued or at least move closer to the head of the queue. Formally, let the state predicate $at(k, x)$ be defined as

$$at(k, x) \;\triangleq\; k \in \mathbb{N} \wedge 1 \leq k \leq len(q) \wedge q[k] = x$$

The following proof outline uses rule (WF1) to show

$$SIQISpec \Rightarrow (at(k, x) \leadsto o = x \vee at(k-1, x))$$

where k and x are rigid variables. The lower-level steps, which rely on data axioms, are omitted.

1. $at(k, x) \wedge [SIQNext]_{q,i,o} \Rightarrow at(k, x)' \vee o' = x \vee at(k-1, x)'$
   1.1. $at(k, x) \wedge SIQEnq \Rightarrow at(k, x)'$
   1.2. $at(k, x) \wedge k = 1 \wedge SIQDeq \Rightarrow o' = x$
   1.3. $at(k, x) \wedge k > 1 \wedge SIQDeq \Rightarrow at(k-1, x)'$
   1.4. $at(k, x) \wedge \text{UNCHANGED}\,\langle q, i, o \rangle \Rightarrow at(k, x)'$
   1.5. Q.E.D.
   PROOF: Steps 1.1– 1.4, definition of $[SIQNext]_{q,i,o}$, and propositional logic.
2. $at(k, x) \wedge \langle SIQNext \wedge SIQDeq \rangle_{q,i,o} \Rightarrow o' = x \vee at(k-1, x)'$

PROOF: Follows from Steps 1.2 and 1.3 in the proof of step 1.

3. $at(k, x) \Rightarrow \text{ENABLED}\, \langle \mathit{SIQDeq} \rangle_{q,i,o}$

   PROOF: The definition of $at(k, x)$ implies $q \neq \langle\, \rangle$, which equivales the enabledness condition.

4. Q.E.D.

   PROOF: Steps 1– 3 and rule (WF1).

5.5 Liveness from Well-Founded Orderings

The general method for proving liveness properties in logics of programs is based on the principle of well-founded relations $\prec$. (A binary relation is well-founded iff there is no infinite descending chain $x_0 \succ x_1 \succ x_2 \succ \ldots$.) The principle is embodied in the following axiom:

$$(\forall\, x : H \leadsto (G \vee \exists\, y : y \prec x \wedge H[y/x])) \Rightarrow ((\exists\, x : H) \leadsto G) \qquad \text{(LATTICE)}$$

where x and y are rigid variables such that y does not occur in H and x does not occur in G.

Formula (LATTICE) asserts that whenever any state that satisfies $H(x)$, for some x, is followed by a state that satisfies either G or $H(y)$, for some y such that $y \prec x$, then every state that satisfies $H(z)$ is followed by a state that satisfies G.

Using formula (LATTICE) and the liveness property shown in section 5.4, we can establish that the synchronous queue is live, that is,

$$\mathit{SIQISpec} \Rightarrow (i = x \leadsto o = x)$$

where x is a rigid variable.

1. $\mathit{SIQISpec} \Rightarrow (at(k, x) \leadsto o = x \vee at(k - 1, x))$

   PROOF: Proved in section 5.4.

2. $\mathit{SIQISpec} \Rightarrow ((\exists\, k : at(k, x)) \leadsto o = x)$

   PROOF: Step 1 and axiom (LATTICE).

3. $\mathit{SIQISpec} \Rightarrow \Box(i = x \Rightarrow o = x \vee \exists\, k : at(k, x))$

   3.1. $\mathit{SIQISpec} \Rightarrow \Box(i = (\langle\, o\, \rangle \circ q)[\mathit{len}(\langle\, o\, \rangle \circ q)])$

   PROOF: From the invariant shown in section 5.1.

   3.2. $i = x \wedge i = (\langle\, o\, \rangle \circ q)[\mathit{len}(\langle\, o\, \rangle \circ q)] \wedge q = \langle\, \rangle \Rightarrow o = x)$

   PROOF: Data-level reasoning.

   3.3. $i = x \wedge i = (\langle\, o\, \rangle \circ q)[\mathit{len}(\langle\, o\, \rangle \circ q)] \wedge q \neq \langle\, \rangle \Rightarrow at(\mathit{len}(q), x))$

   PROOF: Data-level reasoning.

   3.4. Q.E.D.

   PROOF: From steps 3.2 and 3.3 by simple temporal logic.

4. Q.E.D.

   PROOF: Steps 2 and 3 imply $\mathit{SIQISpec} \Rightarrow \Box(i = x \leadsto o = x)$, using simple temporal logic.

5.6 Fairness from Liveness

The remaining class of proof rules allows to prove fairness conditions. As with step simulation, this is necessary when one tries to prove that a lower-level specification implements a higher-level one. Strictly speaking, no separate rules are necessary for proving fairness: fairness conditions are abbreviations for TLA formulas, and therefore (LATTICE) and the (STL) rules should in principle suffice. However, the arguments to establish fairness conditions follow a common pattern, and as in the case of (WF1), it is useful to embody that pattern in a derived rule.[4] The following rule (WF2) is useful to show that a weak fairness

[4] The Isabelle/TLA package [27] contains derivations of the (WFi) and (SFi) rules from a primitive axiom system of temporal logic.

condition for the abstract system is implied by a weak fairness condition for the concrete system; there is an analogous rule for strong fairness [22]:

$$\frac{\begin{array}{ll} \langle N \wedge B \rangle_v \Rightarrow \langle C \rangle_w & \\ P \wedge P' \wedge \langle N \wedge A \rangle_v \Rightarrow B & \\ P \wedge \text{Enabled}\, \langle C \rangle_w \Rightarrow \text{Enabled}\, \langle A \rangle_v & \qquad \text{(WF2)} \\ \Box[N \wedge \neg B]_v \wedge \mathrm{WF}_v(A) \wedge \Box F \Rightarrow \Diamond\Box P & \end{array}}{\Box[N]_v \wedge \mathrm{WF}_v(A) \wedge \Box F \Rightarrow \mathrm{WF}_w(C)}$$

That rule is already fairly complex. To understand why it is true, consider any behavior that satisfies $\Box[N]_v \wedge \mathrm{WF}_v(A) \wedge \Box F$ and where $\langle C \rangle_w$ is always enabled from some state s on. We have to show that $\langle C \rangle_w$ is true at some point beyond that state. If the system ever takes a $\langle B \rangle_v$ step (which, by the assumption, must be an $\langle N \wedge B \rangle_v$ step) beyond s, the first premise ensures that this step is a $\langle C \rangle_w$ step, and we are done. So assume further that we have $\Box[N \wedge \neg B]_v$ true at state s. Then, by the fourth premise, there exists a state t past s such that $\Box P$ is true at t (and $\langle C \rangle_w$ is still persistently enabled). So, the third premise ensures that $\langle A \rangle_v$ is persistently enabled from state t on, and by the fairness condition $\mathrm{WF}_v(A)$, it follows that the system will take an $\langle A \rangle_v$ step at some point past t. But this step actually satisfies $P \wedge P' \wedge \langle N \wedge A \rangle_v$ and is therefore a $\langle B \rangle_v$ step, by the second premise, so we reach a contradiction. We illustrate rule (WF2) by proving

$$\mathit{SIQISpec} \Rightarrow \mathrm{WF}_o(\mathit{SNQDeq})$$

Again, we omit the low-level steps that rely on reasoning about finite sequences. The formula P is the invariant for the synchronous queue proven in section 5.1.

1. $\langle \mathit{SIQNext} \wedge (\mathit{SIQDeq} \wedge o' \neq o) \rangle_{q,i,o} \Rightarrow \langle \mathit{SNQDeq} \rangle_o$
2. $P \wedge P' \wedge \langle \mathit{SIQNext} \wedge \mathit{SIQDeq} \rangle_{q,i,o} \Rightarrow \mathit{SIQDeq} \wedge o' \neq o$
   PROOF: The definition of SIQDeq ensures that the queue is nonempty, and therefore we may use the invariant P to conclude that the value of o is different from the head of the queue.
3. $P \wedge \text{Enabled}\, \langle \mathit{SNQDeq} \rangle_o \Rightarrow \text{Enabled}\, \langle \mathit{SIQDeq} \rangle_{i,q,o}$
   PROOF: Both actions are enabled iff $q \neq \langle\,\rangle$.
4. $\Box[\mathit{SIQNext} \wedge \neg \mathit{SIQDeq}]_{q,i,o} \wedge \mathrm{WF}_{q,i,o}(\mathit{SIQDeq}) \wedge \Box P \Rightarrow \Diamond\Box P$
   PROOF: Simple temporal logic.
5. $\Box[\mathit{SIQNext}]_{q,i,o} \wedge \mathrm{WF}_{q,i,o}(\mathit{SIQDeq}) \wedge \Box P \Rightarrow \mathrm{WF}_o(\mathit{SNQDeq})$
   PROOF: Steps 1– 4 and rule (WF2).
6. Q.E.D.
   PROOF: Step 5, the definition of $\mathit{SIQISpec}$, and the theorem $\mathit{SIQISpec} \Rightarrow \Box P$ proved in section 5.1.

5.7 Hiding

It remains to consider rules for the quantifiers $\exists$ and $\boldsymbol{\exists}$ over rigid and flexible variables. The elementary rules are familiar from first-order logic:

$$(\exists\,\mathrm{I}) \quad F[t/x] \Rightarrow \exists\, x : F \qquad\qquad (\boldsymbol{\exists}\,\mathrm{I}) \quad F[t/x] \Rightarrow \boldsymbol{\exists}\, x : F$$

$$(\exists\,\mathrm{E}) \quad \frac{F \Rightarrow G}{(\exists\, x : F) \Rightarrow G} \qquad\qquad (\boldsymbol{\exists}\,\mathrm{E}) \quad \frac{F \Rightarrow G}{(\boldsymbol{\exists}\, x : F) \Rightarrow G}$$

The elimination rules ($\exists$ E) and ($\boldsymbol{\exists}$ E) require the usual side condition that x not be free in G. For the introduction rule ($\exists$ I), the "witness term" t should not contain flexible variables. The witness term t in the introduction rule ($\boldsymbol{\exists}$ I) indicates how to construct the high-level internal variables from the low-level ones, in computer-science parlance, such a construction is often called a "refinement mapping".

We illustrate the use of the quantifier rules by proving that the interleaving variant of the synchronous queue implements the noninterleaving variant, that is, by proving the formula $SIQSpec \Rightarrow SNQSpec$.

1. $SIQISpec \Rightarrow SNQISpec$
   PROOF SKETCH: Parts of this assertion have been proved in sections 5.2 and 5.6; the proof of the remaining parts is similar and omitted.
2. $SIQISpec \Rightarrow \boldsymbol{\exists}\, q : SNQISpec$
   PROOF: By ($\boldsymbol{\exists}$I), we may conclude $SNQISpec \Rightarrow \boldsymbol{\exists}\, q : SNQISpec$, and the assertion therefore follows using step 1.
3. Q.E.D.
   PROOF: Because q is not free in $\boldsymbol{\exists}\, q : SNQISpec$, we may apply rule ($\boldsymbol{\exists}$E) to deduce $(\boldsymbol{\exists}\, q : SIQISpec) \Rightarrow \boldsymbol{\exists}\, q : SNQISpec$, which is the assertion.

Usually, proving that a specification implements another one requires to give a more elaborate refinement mapping rather than just the identity as in our example. For example, the reader is invited to prove that the composition of two noninterleaving synchronous queues in series implements a single noninterleaving synchronous queue. This fact is asserted by the TLA formula

$$(\boldsymbol{\exists}\, mid : SNQSpec[mid/o] \wedge SNQSpec[mid/i]) \Rightarrow SNQSpec$$

In general, it need not even be the case that the internal "high-level" variables can be constructed statewise from the internal "low-level" variables. In logical terms, the rules ($\boldsymbol{\exists}$I) and ($\boldsymbol{\exists}$E) are sound, but incomplete. This incompleteness is not specific to TLA, but arises similarly in many different settings [1, 12, 14, 25]. It has been shown that the incompleteness can be overcome, for a reasonably large class of specifications, by adding so-called *auxiliary variables* such as history and prophecy variables. Introduction rules for history and prophecy variables can be formulated as proof rules in TLA such as the following:

$$\begin{array}{ll} (\boldsymbol{\exists}\mathrm{H}) & (\exists\, h : I) \wedge \Box[\forall\, h \exists\, h' : N]_v \Rightarrow \boldsymbol{\exists}\, h : I \wedge \Box[N]_{v,h} \\ (\boldsymbol{\exists}\mathrm{P}) & \Box(\exists\, p : I) \wedge \Box[\forall\, p' : I' \Rightarrow \exists\, p : I \wedge N]_v \wedge \Box\Diamond(\exists_1 p : I) \\ & \Rightarrow \boldsymbol{\exists}\, p : \Box I \wedge \Box[N]_{v,p} \end{array}$$

In these rules, I is a state predicate and N is an action. The formula $\exists_1 x : F$ asserts that there exists exactly one x satisfying F.

Axiom ($\boldsymbol{\exists}$H) allows to construct a history variable h by "forward induction": by the first conjunct on the left-hand side, we may pick some initial value that satisfies I, and the second conjunct allows to extend some already constructed sequence by a new value h' such that $[N]_{v,h}$ is satisfied.

Prophecy variables are less intuitive and rely on a principle of "backward induction": the first two conjuncts on the left-hand side allow to construct a tree of values that all satisfy I (at the respective state) and are related by $[N]_{v,p}$. The third conjunct ensures that this tree is infinitely often of finite width (in fact, it contains only one node), and therefore the tree contains an infinite path, using König's lemma. The values on this path are a witness for the right-hand side.

TLA adds a third kind of auxiliary variables, which are related to the stuttering-invariant semantics of quantification. In fact, it may be necessary to force the low-level specification to take enough (external) stuttering steps so that the high-level specification may perform its actions. Lamport [23] gives the example of a clock specification S_1 whose hour and minute hands are visible at the interface, but whose internal variables also include a seconds hand. This specification is externally equivalent to a specification S_2 that is written only in terms of the hour and minute hands. However, in order to prove $S_2 \Rightarrow S_1$, we need to ensure that S_2 performs 60 steps without changing its external variables. This can be achieved with the help of a *stuttering variable*, introduced via the axiom

$$(\exists\,\mathrm{S}) \quad \forall\, n \in \mathbb{N} : \exists\, s : \begin{array}[t]{l} \wedge\; s = n \\ \wedge\; \Box[(s > 0 \wedge s' = s - 1 \wedge v' = v) \vee (s = 0 \wedge s' = n)]_{v,s} \\ \wedge\; \mathrm{WF}_{v,s}(s > 0 \wedge s' = s - 1 \wedge v' = v) \end{array}$$

6 Conclusion

In this paper, I have tried to convey some of the spirit that underlies the TLA approach to system specification and verification. Its main tenet is that one should strive to use concepts of formal logic—a framework developed by mathematicians over thousands of years and found useful in formalizing concepts of mathematics and engineering—rather than *ad hoc* notions such as toy programming languages when reasoning about reactive systems. At the hand of simple examples, I have tried to show that TLA supports various styles of specification that are adequate to describe different programming models.

Basically, TLA specifications desribe state machines, augmented by fairness conditions that ensure that actions cannot be delayed forever. These fairness conditions are conveniently expressed in linear-time temporal logic, and therefore one needs some temporal logic laws to reason about TLA specifications. However, TLA does not include some fancier temporal logic operators such as **until** or **atnext**. It can be shown that these operators are definable in TLA, using flexible quantification. But, experience shows that excessive use of such operators makes specifications hard to read and to reason about. Quantification over flexible variables is both more powerful and more elementary.

Structural concepts such as (parallel) composition, refinement, and hiding are expressed as logical concepts, namely as conjunction, implication, and existential quantification, respectively. For this to be possible, invariance of all formulas under stuttering is essential. For example, a lower-level system will usually include extra steps that are not visible at the higher level; this is related to the use of weak bisimulation in process calculi [28]. Similarly, a component of a complex system should allow other components to take steps in which it does not participate.

We have then turned to verification and have studied proof rules for safety and liveness properties. Because TLA adds a relatively thin layer of temporal logic to standard predicate logic, it is possible to quickly reduce most TLA proofs to a set of nontemporal proof obligations. This facilitates the support of TLA verification by existing theorem provers, and in fact there exist a number of embeddings of TLA in interactive proof assistants [9, 13, 27]. Machine assistance is in fact indispensible for any verification task beyond the most elementary toy examples, as the bookkeeping effort can quickly become overwhelming. Fortunately, many proof obligations that arise in system verification are quite trivial and can be solved by automatic tactics of powerful interactive provers.

We have not been able in this tutorial to cover more advanced topics such as real-time and hybrid systems or compositional ("assumption-commitment style") verification. Interested readers will find much more information and new material at the TLA Web page [19].

Acknowledgements

Thanks are due to Martín Abadi and Leslie Lamport for introducing me to TLA and for many hours of discussion on various topics related to TLA. Some of the material presented in this tutorial has been adapted from a more theoretically-minded presentation of TLA [5].

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. Research Report 91, Digital Equipment Corporation, Systems Research Center, 1992. An earlier version, without proofs, appeared in [8, pages 1–27].

[3] Martín Abadi and Leslie Lamport. Conjoining specifications. Research Report 118, Digital Equipment Corporation, Systems Research Center, 1993.

[4] Martín Abadi, Leslie Lamport, and Stephan Merz. A TLA solution to the RPC-Memory specification problem. In Manfred Broy, Stephan Merz, and Katharina Spies, editors, *Formal System Specification: The RPC-Memory Specification Case Study*, volume 1169 of *Lecture Notes in Computer Science*, pages 21–66. Springer-Verlag, Berlin, 1996.

[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] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking. In Manfred Broy, editor, *Deductive Program Design*, volume F-152 of *NATO ASI series*, pages 305–350. Springer-Verlag, Berlin, 1996.

[8] J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, editors. *Real-Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1992. Proceedings of a REX Real-Time Workshop, held in The Netherlands in June, 1991.

[9] Urban Engberg. *Reasoning in the Temporal Logic of Actions*. PhD thesis, Aarhus University, 1994. See also: Preliminary TLP manual. Available at http://www.daimi.aau.dk/~urban/tlp/tlp.html.

[10] Urban Engberg, Peter Gronning, and Leslie Lamport. Mechanical verification of concurrent systems with TLA. In *Computer-Aided Verification*, Lecture Notes in Computer Science, pages 44–55. Springer-Verlag, June 1992. Proceedings of the Fourth International Conference, CAV'92.

[11] Dov Gabbay, Amir Pnueli, S. Shelah, and Jonathan Stavi. On the temporal analysis of fairness. In *Proceedings of the 7th Annual ACM Symposium on Principles of Programming Languages*, pages 163–173. ACM, 1980.

[12] Bengt Jonsson. Simulations between specifications of distributed systems. In Jos C. M. Baeten and Jan Frisco Groote, editors, *CONCUR '91, 2nd International Conference on Concurrency Theory*, volume 527 of *Lecture Notes in Computer Science*, pages 346–360, Amsterdam, The Netherlands, 1991. Springer.

[13] Sara Kalvala. A formulation of TLA in Isabelle. Available at http://www.dcs.warwick.ac.uk/~sk/papers/tla.dvi, March 1995.

[14] N. Klarlund and Fred B. Schneider. Proving nondeterministically specified safety properties using progress measures. *Information and Computation*, 107(1):151–170, November 1993.

[15] Fred Kröger. *Temporal Logic of Programs*, volume 8 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, 1987.

[16] R.P. Kurshan and Leslie Lamport. Verification of a multiplier: 64 bits and beyond. In Costas Courcoubetis, editor, *Computer-Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 166–179. Springer-Verlag, June 1993. Proceedings of the Fifth International Conference, CAV'93.

[17] Peter Ladkin. Formal but lively buffers in TLA+. Research Report RVS-RR-96-07, AG Rechnernetze und verteilte Systeme, Univ. Bielefeld, Bielefeld, 1996.

[18] Peter Ladkin, Leslie Lamport, Bryan Olivier, and Denis Roegel. Lazy caching in TLA. *Distributed Computing*, 1997. to appear.

[19] Leslie Lamport. The TLA Home Page. www.research.digital.com/ SRC/ personal/ Leslie_Lamport/ tla/ tla.html.

[20] Leslie Lamport. How to write a proof. Research Report 94, Digital Equipment Corporation, Systems Research Center, February 1993.

[21] Leslie Lamport. Hybrid systems in TLA$^+$. In Robert L. Grossman, Anil Nerode, Anders P. Ravn, and Hans Rischel, editors, *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*, pages 77–102. Springer-Verlag, 1993.

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

[23] Leslie Lamport. Refinement in state-based formalisms. Technical Note 1996–001, Digital Equipment Corporation, Systems Research Center, Palo Alto, California, December 1996.

[24] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, San Mateo, CA, 1996.

[25] Nancy Lynch and Frits Vaandrager. Forward and backward simulations. Part I: Untimed systems. Technical Report CS-R9313, CWI, March 1993.

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

[27] Stephan Merz. Isabelle/TLA. Available on the WWW at URL http://www4.informatik.tu-muenchen.de/~merz/isabelle/, 1997.

[28] Robin Milner. *A Calculus of Communicating Systems*. Prentice-Hall, London, 1989.

[29] 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.

[30] Fred B. Schneider. *On Concurrent Programming*. Graduate Texts in Computer Science. Springer-Verlag, New York, 1997.