The Specification Language TLA$^+$

Stephan Merz

INRIA Lorraine, LORIA, 615 rue du Jardin Botanique, F-54602 Villers-lès-Nancy, France, Stephan.Merz@loria.fr

1 Introduction

The specification language TLA$^+$ was designed by Lamport for formally describing and reasoning about distributed algorithms. It is described in Lamport's book Specifying Systems [29], which also gives good advice on how to make the best use of TLA$^+$ and its supporting tools. Systems are specified in TLA$^+$ as formulas of the Temporal Logic of Actions, TLA, a variant of linear-time temporal logic also introduced by Lamport [27]. The underlying data structures are specified in (a variant of) Zermelo–Fränkel set theory, the language accepted by most mathematicians as the standard basis for formalizing mathematics. This choice is motivated by a desire for conciseness, clarity, and formality that befits a language of formal specification where executability or efficiency are not of major concern. TLA$^+$ specifications are organized in modules that can be reused independently.

In a quest for minimality and orthogonality of concepts, TLA$^+$ does not formally distinguish between specifications and properties: both are written as logical formulas, and concepts such as refinement, composition of systems, and hiding of the internal state are expressed using logical connectives of implication, conjunction, and quantification. Despite its expressiveness, TLA$^+$ is supported by tools such as model checkers and theorem provers to aid a designer in carrying out formal developments.

This chapter attempts to formally define the core concepts of TLA and TLA$^+$ and to describe the motivation behind some choices, in particular with respect to competing formalisms. Before doing so, an introductory overview of system specification in TLA$^+$ is given using the example of a resource allocator. Lamport's book remains the definitive reference for the language itself and on the methodology for using TLA$^+$. In particular, the module language of TLA$^+$ is only introduced by example, and the rich standard mathematical library is only sketched.

The outline of this chapter is as follows. Sect. 2 introduces TLA$^+$ by means of a first specification of the resource allocator and illustrates the use

of the TLC model checker. The logic of TLA is formally defined in Sect. 3, followed by an overview of the TLA$^+$ proof rules for system verification in Sect. 4. Section 5 describes the version of set theory that underlies TLA$^+$, including some of the constructions most frequently used for specifying data. The resource allocator example is taken up again in Sect. 6, where an improved high-level specification is given and a step towards a distributed refinement is taken. Finally, Sect. 7 contains some concluding remarks.

2 Example: A Simple Resource Allocator

We introduce TLA$^+$ informally, by means of an example that will also serve as a running example for this chapter. After stating the requirements informally, we present a first system specification, and describe the use of the TLA$^+$ model checker TLC to analyse its correctness.

2.1 Informal Requirements

The purpose of the resource allocator is to manage a (finite) set of resources that are shared among a number of client processes. The allocation of resources is subject to the following constraints.

1. A client that currently does not hold any resources and that has no pending requests may issue a request for a set of resources.

   Rationale: We require that no client should be allowed to "extend" a pending request, possibly after the allocator has granted some resources. A single client process might concurrently issue two separate requests for resources by appearing under different identities, and therefore the set of "clients" should really be understood as identifiers for requests, but we shall not make this distinction here.

2. The allocator may grant access to a set of available (i.e., not currently allocated) resources to a client.

   Rationale: Resources can be allocated in batches, so an allocation need not satisfy the entire request of the client: the client may be able to begin working with a subset of the resources that it requested.

3. A client may release some resources that it holds.

   Rationale: Similarly to allocation, clients may return just a subset of the resources they currently hold, freeing them for allocation to a different process.

4. Clients are required to eventually free the resources they hold once their entire request has been satisfied.

The system should be designed such that it ensures the following two properties.

- *Safety:* no resource is simultaneously allocated to two different clients.
- *Liveness:* every request issued by some client is eventually satisfied.

2.2 A First TLA$^+$ Specification

A first TLA$^+$ specification of the resource allocator appears in Fig. 1 on the following page. Shortcomings of this model will be discussed in Sect. 6, where a revised specification will appear.

TLA$^+$ specifications are organised in modules that contain declarations (of parameters), definitions (of operators), and assertions (of assumptions and theorems). Horizontal lines separate different sections of the module *SimpleAllocator*; these aid readability, but have no semantic content. TLA$^+$ requires that an identifier must be declared or defined before it is used, and that it cannot be reused, even as a bound variable, in its scope of validity.

The first section declares that thr module *SimpleAllocator* is based on the module *FiniteSet*, which is part of the TLA$^+$ standard library (discussed in Sect. 5). Next, the constant and variable parameters are declared. The constants *Clients* and *Resources* represent the sets of client processes and of resources managed by the resource allocator. Constant parameters represent entities whose values are fixed during system execution, although they are not defined in the module, because they may change from one system instance to the next. Observe that there are no type declarations: TLA$^+$ is based on Zermelo–Fränkel (ZF) set theory — so all values are sets. The set *Resources* is assumed to be finite – the operator *IsFiniteSet* is defined in the module *FiniteSet*. The variable parameters *unsat* and *alloc* represent the current state of the allocator by recording the outstanding requests of the client processes, and the set of resources allocated to the clients. In general, variable parameters represent entities whose values change during system execution; in this sense, they correspond to program variables.

The second section contains the definition of the operators *TypeInvariant* and *available*. In general, definitions in TLA$^+$ take the form

$$Op(arg_1, \ldots, arg_n) \triangleq exp.$$

In TLA$^+$, multiline conjunctions and disjunctions are written as lists "bulleted" with the connective, and indentation indicates the hierarchy of nested conjunctions and disjunctions [26]. The formula *TypeInvariant* states the intended "types" of the state variables *unsat* and *alloc*, which are functions that associate a set of (requested or received) resources with each client.[1] Observe, again, that the variables are not constrained to these types: *TypeInvariant* just declares a formula, and a theorem towards the end of the module asserts that the allocator specification respects the typing invariant. This theorem will have to be proven by considering the possible transitions of the system.

[1] In TLA$^+$, the power set of a set S is written as SUBSET S.

MODULE *SimpleAllocator*

EXTENDS *FiniteSet*
CONSTANTS *Clients, Resources*
ASSUME *IsFiniteSet(Resources)*
VARIABLES
unsat, *alloc*

unsat[*c*] denotes the outstanding requests of client *c*
alloc[*c*] denotes the resources allocated to client *c*

$$TypeInvariant \triangleq$$
$$\land\ unsat \in [Clients \to \text{SUBSET } Resources]$$
$$\land\ alloc \in [Clients \to \text{SUBSET } Resources]$$
$$available \triangleq \text{ set of resources free for allocation}$$
$$Resources \setminus (\text{UNION}\{alloc[c] : c \in Clients\})$$

$$Init \triangleq \text{ initially, no resources have been requested or allocated}$$
$$\land\ unsat = [c \in Clients \mapsto \{\}]$$
$$\land\ alloc = [c \in Clients \mapsto \{\}]$$
$$Request(c, S) \triangleq \text{ Client } c \text{ requests set } S \text{ of resources}$$
$$\land\ S \neq \{\} \land unsat[c] = \{\} \land alloc[c] = \{\}$$
$$\land\ unsat' = [unsat \text{EXCEPT}![c] = S]$$
$$\land\ \text{UNCHANGED } alloc$$
$$Allocate(c, S) \triangleq \text{ Set } S \text{ of available resources are allocated to client } c$$
$$\land\ S \neq \{\} \land S \subseteq available \cap unsat[c]$$
$$\land\ alloc' = [alloc \text{EXCEPT}![c] = @ \cup S]$$
$$\land\ unsat' = [unsat \text{EXCEPT}![c] = @ \setminus S]$$
$$Return(c, S) \triangleq \text{ Client } c \text{ returns a set of resources that it holds.}$$
$$\land\ S \neq \{\} \land S \subseteq alloc[c]$$
$$\land\ alloc' = [alloc \text{EXCEPT}![c] = @ \setminus S]$$
$$\land\ \text{UNCHANGED } unsat$$
$$Next \triangleq \text{ The system's next-state relation}$$
$$\exists c \in Clients, S \in \text{SUBSET } Resources :$$
$$Request(c, S) \lor Allocate(c, S) \lor Return(c, S)$$
$$vars \triangleq \langle unsat, alloc \rangle$$

$$SimpleAllocator \triangleq \text{ The complete high-level specification}$$
$$\land\ Init \land \Box[Next]_{vars}$$
$$\land\ \forall c \in Clients : \text{WF}_{vars}(Return(c, alloc[c]))$$
$$\land\ \forall c \in Clients : \text{SF}_{vars}(\exists S \in \text{SUBSET } Resources : Allocate(c, S))$$

$$Safety \triangleq \forall\, c_1, c_2 \in Clients : c_1 \neq c_2 \Rightarrow alloc[c_1] \cap alloc[c_2] = \{\}$$
$$Liveness \triangleq \forall\, c \in Clients, r \in Resources : r \in unsat[c] \rightsquigarrow r \in alloc[c]$$

THEOREM *SimpleAllocator* $\Rightarrow \Box$ *TypeInvariant*
THEOREM *SimpleAllocator* $\Rightarrow \Box$ *Safety*
THEOREM *SimpleAllocator* $\Rightarrow$ *Liveness*

Fig. 1. A simple resource allocator.

The set *available* is defined to contain those resources that are currently not allocated to any client.

The third section contains a list of definitions, which constitute the main body of the allocator specification. The state predicate *Init* represents the initial condition of the specification: no client has requested or received any resources. The action formulas $Request(c, S)$, $Allocate(c, S)$, and $Return(c, S)$ model a client c requesting, receiving, or returning a set S of resources. In these formulas, unprimed occurrences of state variables (e.g., *unsat*) denote their values in the state before the transition, whereas primed occurrences (e.g., *unsat*′) denote their values in the successor state, and UNCHANGEDt is just a shorthand for $t' = t$. Also, function application is written using square brackets, so $unsat[c]$ denotes the set of resources requested by client c. The EXCEPT construct models a function update; more precisely, when t denotes a value in the domain of the function f, the expression $[f \text{ EXCEPT } ![t] = e]$ denotes the function g that agrees with f except that $g[t]$ equals e. In the right-hand side e of such an update, @ denotes the previous value $f[t]$ of the function at the argument position being updated. For example, the formule $Allocate(c, S)$ requires that S be a nonempty subset of the available resources that are part of the request of client c, allocates those resources to c, and removes them from the set of outstanding requests of c.

The action formula *Next* is defined as the disjunction of the request, allocate, and return actions, for some client and some set of resources; it defines the next-state relation of the resource allocator. Again, there is nothing special about the names *Init* and *Next*, they are just conventional for denoting the initial condition and the next-state relation.

The overall specification of the resource allocator is given by the temporal formula *SimpleAllocator*. This is defined as a conjunction of the form

$$I \wedge \Box[N]_v \wedge L$$

where I is the initial condition (a state predicate), N is the next-state relation, and L is a conjunction of fairness properties, each concerning a disjunct of the next-state relation. While not mandatory, this is the standard form of system specification in TLA$^+$, and it corresponds to the definition of a transition system (or state machine) with fairness constraints. More precisely, the formula $\Box[N]_v$ specifies that every transition either satisfies the action formula N or leaves the expression v unchanged. In particular, this formula admits "stuttering transitions" that do not affect the variables of interest. Stuttering invariance is a key concept of TLA that simplifies the representation of refinement, as well as compositional reasoning; we shall explore temporal formulas and stuttering invariance in more detail in Sect. 3.4.

The initial condition and the next-state relation specify how the system *may* behave. Fairness conditions complement this by asserting what actions *must* occur (eventually). The weak fairness condition for the return action states that clients should eventually return the resources they hold. The strong fairness condition for resource allocation stipulates that for each client c, if it

is possible to allocate some resources to c infinitely often, then the allocator should eventually give some resources to c.

The following section of the specification defines the two main correctness properties *Safety* and *Liveness*. The formula *Safety* asserts a safety property [10] of the model by stating that no resource is ever allocated to two distinct clients. Formula *Liveness* represents a liveness property that asserts that whenever some client c requests some resource r, that resource will eventually be allocated to c.[2] Observe that there is no formal distinction in TLA$^+$ between a system specification and a property: both are expressed as formulas of temporal logic. Asserting that a specification S has a property F amounts to claiming validity of the implication $S \Rightarrow F$. Similarly, refinement between specifications is expressed by (validity of) implication, and a single set of proof rules is used to verify properties and refinement; we shall explore deductive verification in Sect. 4.

Finally, the module *SimpleAllocator* asserts three theorems stating that the specification satisfies the typing invariant as well as the safety and liveness properties defined above. A formal proof language for TLA$^+$, based on a hierarchical proof format [28], is currently being designed.

2.3 Model-Checking the Specification

Whereas programs can be compiled and executed, TLA$^+$ models can be validated and verified. In this way, confidence is gained that a model faithfully reflects the intended system, and that it can serve as a basis for more detailed designs and, ultimately, for implementations. Tools can assist the designer in carrying out these analyses. In particular, simulation lets a user explore some traces, possibly leading to the detection of deadlocks or other unanticipated behavior. Deductive tools such as model checkers and theorem provers assist in the formal verification of properties. The TLA$^+$ model checker TLC is a powerful and eminently useful tool for verification and validation, and we shall now illustrate its use for the resource allocator model of Fig. 1.

TLC can compute and explore the state space of finite-state instances of TLA$^+$ models. Besides the model itself, TLC requires a second input file, called the *configuration file*, which defines the finite-state instance of the model to be analysed, and that declares which of the formulas defined in the model represents the system specification and which are the properties to be verified over that finite-state instance.[3] Figure 2 shows a configuration file for analysing the module *SimpleAllocator*. Definitions of the sets *Clients* and *Resources* fix specific instance of the model that TLC should consider. In our case, these sets consist of symbolic constants. The keyword `SPECIFICATION` indicates which formula represents the main system specification, and the

[2] The formula $P \leadsto Q$ asserts that any state that satisfies P will eventually be followed by a state satisfying Q.

[3] TLC ignores any theorems asserted in the module.

```
CONSTANTS
   Clients = {c1,c2,c3}
   Resources = {r1,r2}
SPECIFICATION
   SimpleAllocator
INVARIANTS
   TypeInvariant Safety
PROPERTIES
   Liveness
```

Fig. 2. Sample configuration file for TLC

keywords `INVARIANTS` and `PROPERTIES` define the properties to be verified by TLC. (For a more detailed description of the format and the possible directives that can be used in configuration files, see Lamport's book [29] and the tool documentation [23].)

```
TLC Version 2.0 of January 16, 2006
Model-checking
Parsing file SimpleAllocator.tla
Parsing file /sw/tla/tlasany/StandardModules/FiniteSets.tla
Parsing file /sw/tla/tlasany/StandardModules/Naturals.tla
Parsing file /sw/tla/tlasany/StandardModules/Sequences.tla
Semantic processing of module Naturals
Semantic processing of module Sequences
Semantic processing of module FiniteSets
Semantic processing of module SimpleAllocator
Implied-temporal checking--satisfiability problem has 6 branches.
Finished computing initial states: 1 distinct state generated.
--Checking temporal properties for the complete state space...
Model checking completed. No error has been found.
  Estimates of the probability that TLC did not check
  all reachable states because two distinct states had
  the same fingerprint:
    calculated (optimistic):  2.673642557349254E-14
    based on the actual fingerprints:  6.871173129000332E-15
1633 states generated, 400 distinct states found,
0 states left on queue.
The depth of the complete state graph search is 6.
```

Fig. 3. TLC output

Running TLC on this model produces an output similar to that shown in Fig. 3; some details may vary according to the version and the installation of TLC. First, TLC parses the TLA$^+$ input file and checks it for well-formedness. It then computes the graph of reachable states for the instance of the model

defined by the configuration file, verifying the invariants "on the fly" as it computes the state space. Finally, the temporal properties are verified over the state graph. In our case, TLC reports that it has not found any error. In order to improve efficiency, TLC compares states on the basis of a hash code ("fingerprint") during the computation of the state space, rather than comparing them precisely. In the case of a hash collision, TLC will mistakenly identify two distinct states and may therefore miss part of the state space. TLC attempts to estimate the probability that such a collision occurred during the run, on the basis of the distribution of the fingerprints. TLC also reports the number of states that it generated during its analysis, the number of distinct states, and the depth of the state graph, i.e. the length of the longest cycle. These statistics can be valuable information; for example, if the number of generated states is lower than expected, some actions may have preconditions that never evaluate to true. It is a good idea to use TLC to verify many, properties, even trivial ones, as well as some non-properties. For example, one can attempt to assert the negation of each action guard as an invariant in order to let TLC compute a finite execution that ends in a state where the action can actually be activated. For our example, the TLC run is completed in a few seconds; most of the running time is spent on the verification of the property *Liveness*, which is expanded into six properties, for each combination of clients and resources.

After this initial success, we can try to analyse somewhat larger instances, but this quickly leads to the well-known problem of state space explosion. For example, increasing the number of resources from two to three in our model results in a state graph that contains 8000 distinct states (among 45 697 states generated in all), and the analysis will take a few minutes instead of seconds.

One may observe that the specification and the properties to be verified are invariant with respect to permutations of the sets of clients and resources. Such symmetries are frequent, and TLC implements a technique known as symmetry reduction, which can counteract the effect of state-space explosion. In order to enable symmetry reduction, we simply extend the TLA$^+$ module by the definition of the predicate

$$Symmetry \triangleq Permutations(Clients) \cup Permutations(Resources)$$

(the operator *Permutations* is defined in the standard TLC module, which must therefore be added to the EXTENDS clause) and to indicate

```
SYMMETRY Symmetry
```

in the configuration file. Unfortunately, the implementation of symmetry reduction in TLC is not compatible with checking liveness properties, and in fact, TLC reports a meaningless "counter-example" when symmetry reduction is enabled during the verification of the liveness property for our example. However, when restricted to checking the invariants, symmetry reduction with respect to both parameter sets reduces the number of states explored to

50 (and to 309 for three clients and three resources), and the run times are similarly reduced to fractions of a second for either configuration.

We can use TLC to explore variants of our specification. For example, verification succeeds when the strong fairness condition

$$\forall\, c \in \mathit{Clients} \,:\, \mathrm{SF}_{\mathit{vars}}(\exists\, S \in \text{SUBSET } \mathit{Resources} \,:\, \mathit{Allocate}(c, S))$$

is replaced by the following condition about individual resources:

$$\forall\, c \in \mathit{Clients}, r \in \mathit{Resources} \,:\, \mathrm{SF}_{\mathit{vars}}(\mathit{Allocate}(c, \{r\})).$$

However, the liveness condition is violated when the strong fairness condition is replaced by either of the two following fairness conditions:

$$\forall\, c \in \mathit{Clients} \,:\, \mathrm{WF}_{\mathit{vars}}(\exists\, S \in \text{SUBSET } \mathit{Resources} \,:\, \mathit{Allocate}(c, S))$$
$$\mathrm{SF}_{\mathit{vars}}(\exists\, c \in \mathit{Clients}, S \in \text{SUBSET } \mathit{Resources} \,:\, \mathit{Allocate}(c, S)).$$

It is a good exercise to attempt to understand these alternative fairness hypotheses in detail and to explain the verification results. Fairness conditions and their representation in TLA are formally defined in Sect. 3.3.

3 TLA: The Temporal Logic of Actions

TLA$^+$ combines TLA, the Temporal Logic of Actions [27], and mathematical set theory. This section introduces the logic TLA by defining its syntax and semantics. In these definitions, we aim at formality and rigor; we do not attempt to explain how TLA may be used to specify algorithms or systems. Sections 4 and 5 explore the verification of temporal formulas and the specification of data structures in set theory, respectively.

3.1 Rationale

The logic of time has its origins in philosophy and linguistics, where it was intended to formalize temporal references in natural language [22, 38]. Around 1975, Pnueli [37] 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-conditions and post-conditions were found to be ill-suited for the description of reactive systems that continuously interact 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 for describing safety and liveness properties [10, 25] 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 the connectives available to refer to different moments in time (e.g., future vs. past references). For computer science applications, the most controversial 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, are appropriate for formulating correctness properties that must hold for all the runs of a system. In contrast, 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 logic is beyond the scope of this chapter, but see, for example, [42] for a good summary, with many references to earlier papers.

Despite initial enthusiasm about temporal logic as a language to describe system properties, attempts to actually write complete system specifications as lists of properties expressed in temporal logic revealed that not even a component as simple as a FIFO queue could be unambiguously specified [39]. This observation has led many researchers to propose that reactive systems should be modeled as state machines, while temporal logic should be retained as a high-level language to describe the correctness properties. A major breakthrough came with the insight that temporal logic properties are decidable over finite-state models, and this has led to the development of model-checking techniques [14], which today are routinely applied to the analysis of hardware circuits, communication protocols, and software.

A further weakness of standard temporal logic becomes apparent 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, making comparisons between specifications more difficult. For example, in Sect. 6 we shall develop a specification of the resource allocator of Fig. 1 as a distributed system where the allocator and the clients communicate by asynchronous message passing. Each of the actions will be split into a subaction performed by the allocator, the corresponding subaction performed by the client, and the transmission of a message over the network, and these subactions will be interleaved with other system events. On the face of it, the two specifications are hard to compare because they use different notions of "next state".

TLA has been designed as a formalism where system specifications and their properties are expressed in the same language, and where the refinement relation is reduced to logical implication. The problems mentioned above are addressed in the following ways. TLA is particularly suited to writing state machine specifications, augmented with fairness conditions, as we have seen in the case of the resource allocator. It is often desirable to expose only that part of the state used to specify a state machine which makes up its externally visible interface, and TLA introduces quantification over state variables to hide the internal state, which a refinement is free to implement in a different manner. The problem with incompatible notions of "next state" at different levels of abstraction is solved by systematically allowing for stuttering steps that do not change the values of the (high-level) state variables. Low-level steps

of an implementation that change only new variables are therefore allowed by the high-level specification. Similar ideas can be found in Back's refinement calculus [11] and in Abrial's Event-B method [9] (see also the chapter by Cansell and Méry in this book). Whereas finite stuttering is desirable for a simple representation of refinement, infinite stuttering is usually undesirable, because it corresponds to livelock, and the above formalisms rule it out via proof obligations that are expressed in terms of well-founded orderings. TLA adopts a more abstract and flexible approach because it associates fairness conditions, stated in temporal logic, with specifications, and these must be shown to be preserved by the refinement, typically using a mix of low-level fairness hypotheses and well-founded ordering arguments.

On the basis of 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.

3.2 Transition Formulas

The language of TLA distinguishes between transition formulas, which describe states and state transitions, and temporal formulas, which characterize behaviors (infinite sequences of states). Basically, transition formulas are ordinary formulas of untyped first-order logic, but TLA introduces a number of specific conventions and notations.

Assume a signature of first-order predicate logic,[4] consisting of:

- $\mathcal{L}_F$ and $\mathcal{L}_P$, which are at most denumerable, of function and predicate symbols, each symbol being of given 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 all variables (rigid and flexible, primed or unprimed).

Transition functions and *transition predicates* (also called *actions*) are first-order 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 $f(v, x, v')$ is a transition function, and the formula

$$C \triangleq \exists\, v' : p(f(v, x, v')) \wedge \neg(v' = x)$$

is an action. Collectively, transition functions and predicates are called *transition formulas* in the literature on TLA.

[4] Recall that TLA can be defined over an arbitrary first-order language. The logic of TLA$^+$ is just TLA over a specific set-theoretical language that will be introduced in Sect. 5.

Transition formulas are interpreted according to ordinary first-order logic semantics: an *interpretation* $\mathcal{I}$ defines a universe $|\mathcal{I}|$ 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 $s : \mathcal{V}_F \to |\mathcal{I}|$ of the flexible variables to values. Given two states s and t and a valuation $\xi : \mathcal{V}_R \to |\mathcal{I}|$ of the rigid variables, we define the combined 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 $[\![E]\!]^{\mathcal{I},\xi}_{s,t}$, is then simply the standard predicate logic semantics of E with respect to the extended valuation $\alpha_{s,t,\xi}$. We may omit any of the superscripts and subscripts if there is no danger of confusion.

We say that a transition predicate A is *valid* for the interpretation $\mathcal{I}$ iff $[\![A]\!]^{\mathcal{I},\xi}_{s,t}$ is true for all states s, t and all valuations ξ. It is *satisfiable* iff $[\![A]\!]^{\mathcal{I},\xi}_{s,t}$ is true for some s, t, and ξ.

The notions of free and bound variables in a transition formula are defined as usual, with respect to the variables in $\mathcal{V}_E$, as is the notion of substitution of a transition function a for a variable $v \in \mathcal{V}_E$ in a transition formula E, 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 variables v and x are free in the action C defined above, whereas v' is bound. Observe in particular 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 depends only on a single state, we simply write $[\![P]\!]^{\xi}_{s}$ when P is a state formula. Transition formulas all of whose free variables are rigid variables are called *constant formulas*; their semantics depends only on the valuation ξ.

Beyond these standard concepts from first-order logic, TLA introduces some specific conventions and notations. 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 the action C above is a state formula with v as its single free flexible variable, the formula C' is formed by substituting v' for v. In doing so, the bound variable v' of C has to be renamed, and we obtain the formula $\exists\, y : p(f(v', x, y)) \wedge \neg(y = x)$.

For an action A, the state formula ENABLED A is obtained by existential quantification over all primed flexible variables that have free occurrences in A. Thus, $[\![\text{ENABLED } A]\!]^{\xi}_{s}$ holds if $[\![A]\!]^{\xi}_{s,t}$ holds for some state t; this is a formal counterpart of the intuition that the action A may occur in the state s. For actions A and B, the composite 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 many action-level abbreviations introduced by TLA are defined in terms of implicit quantification and substitution, their interplay can be quite delicate. For example, if P is a state predicate, then ENABLED P is obviously just P, and therefore (ENABLED $P)'$ equals P'. On the other hand, ENABLED (P') is a constant formula – if P does not contain any rigid variables then ENABLED (P') is valid iff P is satisfiable. Similarly, consider the action

$$A \triangleq v \in \mathbb{Z} \land v' \in \mathbb{Z} \land v' < 0$$

in the standard interpretation where $\mathbb{Z}$ denotes the set of integers, 0 denotes the number zero, and $*$ and $<$ denote multiplication and the "less than" relation, respectively. It is easy to see that ENABLED A is equivalent to the state predicate $v \in \mathbb{Z}$, and hence (ENABLED $A)[(n*n)/v, (n'*n')/v']$ simplifies to $(n * n) \in \mathbb{Z}$. However, substituting in the definition of the action yields

$$A[(n * n)/v, (n' * n')/v'] \equiv (n * n) \in \mathbb{Z} \land (n' * n') \in \mathbb{Z} \land (n' * n') < 0,$$

which is equivalent to FALSE, and so ENABLED $(A[(n * n)/v, (n' * n')/v'])$ is again equivalent to FALSE: substitution does not commute with the ENABLED operator. Similar pitfalls exist for action composition $A \cdot B$.

For an action A and a state function t, one writes $[A]_t$ (pronounced "square A sub t") for $A \lor t' = t$, and, dually, $\langle A \rangle_t$ ("angle A sub t") for $A \land \neg(t' = t)$. Therefore, $[A]_t$ is true of any state transition that satisfies A, but in addition it permits stuttering steps that leave (at least) the value of t unchanged. Similarly, $\langle A \rangle_t$ demands not only that A be true but also that the value of t changes during the transition. As we shall see below, these constructs are used to encapsulate action formulas in temporal formulas.

3.3 Temporal Formulas

Syntax

We now define the temporal layer of TLA, again with the aim of giving precise definitions of syntax and semantics. The inductive definition of temporal formulas (or just "formulas") is given as follows:

- Every state formula is a formula.
- Boolean combinations (the connectives $\neg$, $\land$, $\lor$, $\Rightarrow$, 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 $\Box[A]_t$ is a formula (pronounced "always square A sub t").
- If F is a formula and $x \in \mathcal{V}_R$ is a rigid variable, then $\exists\, x : F$ is a formula.
- If F is a formula and $v \in \mathcal{V}_F$ is a flexible variable, then $\mathbf{\exists}\, v : F$ is a formula.

In particular, an action A by itself is not a temporal formula, not even in the form $[A]_t$. Action formulas occur within temporal formulas only in subformulas $\Box[A]_t$. We assume quantifiers to have lower syntactic precedence than the other connectives, and so their scope extends as far to the right as possible.

At the level of temporal formulas, if $v \in \mathcal{V}_F$ is a flexible variable, then we consider unprimed occurrences v as well as primed occurrences v' to be occurrences of v, and the quantifier $\exists$ binds both kinds of occurrence. More formally, the set of free variables of a temporal formula is a subset of $\mathcal{V}_F \cup \mathcal{V}_R$. The free occurrencesof (rigid or flexible) variables in a state formula P, considered as a temporal formula, are precisely the free occurrences in P, considered as a transition formula. However, a 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 if v occurs in t. Similarly, the 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, after bound variables have been renamed 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

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

results in the formula, up to renaming of the bound variable,

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

Because state formulas do not contain free occurrences of primed flexible variables, the definitions of free and bound occurrences and of substitutions introduced for transition formulas and for temporal formulas agree on state formulas, and this observation justifies the use of the same notation at both levels of formula. Substitutions of terms for primed variables or of proper transition functions for variables are not defined at the temporal level of TLA.

Semantics

Given an interpretation $\mathcal{I}$, temporal formulas are evaluated with respect to an ω-sequence $\sigma = s_0 s_1 \ldots$ of states $s_i : \mathcal{V}_F \to |\mathcal{I}|$ (in the TLA literature, σ is usually called a *behavior*), and with respect to a valuation $\xi : \mathcal{V}_R \to |\mathcal{I}|$ of the rigid variables. For a behavior $\sigma = s_0 s_1 \ldots$, we write σ_i to refer to state s_i, and we write $\sigma|_i$ to denote the suffix $s_i s_{i+1} \ldots$ of σ. The following inductive definition assigns a truth value $[\![F]\!]^{\mathcal{I},\xi}_{\sigma} \in \{\mathbf{t}, \mathbf{f}\}$ to temporal formulas; the semantics of the quantifier $\exists$ over flexible variables is deferred to Sect. 3.4.

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

- $[\![\Box[A]_t]\!]^{\mathcal{I},\xi}_\sigma = \mathbf{t}$ iff for all $i \in \mathbb{N}$, either $[\![t]\!]^{\mathcal{I},\xi}_{\sigma_i} = [\![t]\!]^{\mathcal{I},\xi}_{\sigma_{i+1}}$ or $[\![A]\!]^{\mathcal{I},\xi}_{\sigma_i,\sigma_{i+1}} = \mathbf{t}$ holds: the formula $\Box[A]_t$ holds iff every state transition in σ that modifies the value of t satisfies A.
- $[\![\exists\, x : F]\!]^{\mathcal{I},\xi}_\sigma = \mathbf{t}$ iff $[\![F]\!]^{\mathcal{I},\eta}_\sigma = \mathbf{t}$ for some valuation $\eta : \mathcal{V}_R \to |\mathcal{I}|$ such that $\eta(y) = \xi(y)$ for all $y \in \mathcal{V}_R \setminus \{x\}$: this is standard first-order quantification over (rigid) variables.

Validity and satisfiability of temporal formulas are defined as expected. We write $\models_{\mathcal{I}} F$ (or simply $\models F$ when $\mathcal{I}$ is understood) to denote that F is valid for (all behaviors based on) the interpretation $\mathcal{I}$.

Derived Temporal Formulas

The abbreviations for temporal formulas include the universal quantifiers $\forall$ and $\mathbf{\forall}$ over rigid and flexible variables. The formula $\Diamond F$ ("eventually F"), defined as $\neg\Box\neg F$, asserts that F holds for 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 future transition satisfies A and changes the value of t. We write $F \rightsquigarrow G$ ("F leads to G") for the formula $\Box(F \Rightarrow \Diamond G)$, which asserts that whenever F is true, G will become true eventually. Combinations of the "always" and "eventually" operators express "infinitely often" ($\Box\Diamond$) and "always from some time onward" ($\Diamond\Box$). Observe that a formula F can be both infinitely often true and infinitely often false, and thus $\Diamond\Box F$ is strictly stronger than $\Box\Diamond F$. These combinations are the basis for expressing fairness conditions. In particular, weak and strong fairness for an action $\langle A\rangle_t$ are defined as

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

Informally, fairness conditions assert that an action should eventually occur if it is "often" enabled; they differ in the precise interpretation of "often". Weak fairness $\mathrm{WF}_t(A)$ asserts that the action $\langle A\rangle_t$ must eventually occur if it remains enabled from some point onwards. In other words, the weak fairness condition is violated if, eventually, $\textsc{Enabled}\,\langle A\rangle_t$ remains true without $\langle A\rangle_t$ ever occurring.

The strong fairness condition, expressed by the formula $\mathrm{SF}_t(A)$, requires $\langle A\rangle_t$ to occur infinitely often provided that the action is enabled infinitely often, although it need not remain enabled forever. Therefore, strong fairness is violated if, from some point onward, the action is repeatedly enabled but never occurs. It is a simple exercise in expanding the definitions of temporal

formulas to prove that the various formulations of weak and strong fairness given above are actually equivalent, and that $\mathrm{SF}_t(A)$ implies $\mathrm{WF}_t(A)$.

When one is specifying systems, the choice of appropriate fairness conditions for system actions often requires some experience. Considering again the allocator example of Fig. 1, it would not be enough to require weak fairness for the *Allocate* actions, because several clients may compete for the same resource: allocation of the resource to one client disables allocating the resource to any other client until the first client returns the resource.

3.4 Stuttering Invariance and Quantification

The formulas $\Box[A]_t$ are characteristic of TLA. As we have seen, they allow for "stuttering" transitions that do not change the value of the state function t. In particular, repetitions of states cannot be observed by formulas of this form. Stuttering invariance is important in connection with refinement and composition [25]; see also Sect. 3.5.

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 $v \in V$. For any behavior σ, we define its *V-unstuttered variant* $\natural_V \sigma$ as the behavior obtained by replacing every maximal finite subsequence of V-equivalent states in σ by the first state of that sequence. (If σ ends in an infinite sequence of states all of which are V-equivalent, that sequence is simply copied at the end of $\natural_V \sigma$.)

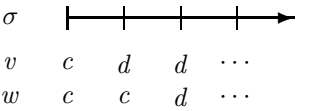
Two behaviors σ and τ are *V-stuttering equivalent*, written $\sigma \approx_V \tau$, if $\natural_V \sigma = \natural_V \tau$. Intuitively, two behaviors σ and τ are V-stuttering equivalent if one can be obtained from the other by inserting and/or deleting finite repetitions of V-equivalent states. In particular, the relation $\approx_{\mathcal{V}_F}$, which we also write as $\approx$, identifies two behaviors that agree up to finite repetitions of identical states.

TLA is insensitive to stuttering equivalence: the following theorem states that TLA is not expressive enough to distinguish between stuttering-equivalent behaviors.

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 $[\![F]\!]^{\mathcal{I},\xi}_{\sigma} = [\![F]\!]^{\mathcal{I},\xi}_{\tau}$.*

For the fragment of TLA formulas without quantification over flexible variables, whose semantics was defined in Sect. 3.3, it is not hard to prove Theorem 1 by induction on the structure of formulas [6, 27]. However, its extension to full TLA requires some care in the definition of quantification over flexible variables: it would be natural to define that $[\![\exists\, v : F]\!]^{\mathcal{I},\xi}_{\sigma} = \mathbf{t}$ iff $[\![F]\!]^{\mathcal{I},\xi}_{\tau} = \mathbf{t}$ for some behavior τ whose states τ_i agree with the corresponding states σ_i for all variables except for v. This definition, however, would not preserve stuttering invariance. As an example, consider the formula F defined below:

$$F \triangleq \land\, v = c \land w = c \\ \land\, \Diamond(w \neq c) \land \Box[v \neq c]_w$$



The formula F asserts that both of the variables v and w initially equal the constant c, that eventually w should be different from c, and that v must be different from c whenever w changes value. In particular, F implies that the value of v must change strictly before any change in the value of w, as illustrated in the picture. Therefore, $\sigma_1(w)$ must equal c.

Now consider the formula $\exists\, v : F$, and assume that τ is a behavior that satisfies $\exists\, v : F$, according to the above definition. It follows that $\tau_0(w)$ and $\tau_1(w)$ must both equal c, but that $\tau_i(w)$ is different from c for some (smallest) $i > 1$. The behavior $\tau|_{i-1}$ cannot satisfy $\exists\, v : F$ because, intuitively, "there is no room" for the internal variable v to change before w changes. However, this is in contradiction to Theorem 1 because τ and $\tau|_{i-1}$ are $\{w\}$-stuttering equivalent, and w is the only free flexible variable of $\exists\, v : F$.

This problem is solved by defining the semantics of $\exists\, v : F$ in such a way that stuttering invariance is ensured. Specifically, the behavior τ may contain extra transitions that modify only the bound variable v. Formally, 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 ($\tau' \approx \tau$).

Being defined as the composition of equivalence relations, $\simeq_v$ is itself an equivalence relation.

Now, we define $[\![\exists\, v : F]\!]^{\mathcal{I},\xi}_{\sigma} = \mathbf{t}$ iff $[\![F]\!]^{\mathcal{I},\xi}_{\tau} = \mathbf{t}$ holds for some behavior $\tau \simeq_v \sigma$. This definition can be understood as "building stuttering invariance into" the semantics of $\exists\, v : F$. It therefore ensures that Theorem 1 holds for all TLA formulas.

3.5 Properties, Refinement, and Composition

We have already seen in the example of the resource allocator that TLA makes no formal distinction between system specifications and their properties: both are represented as temporal formulas. It is conventional to write system specifications in the form

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

where v is a tuple of all of the state variables used to express the specification, the variables x are internal (hidden), *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 individual system actions, and L is a conjunction of formulas $\mathrm{WF}_v(A)$ or $\mathrm{SF}_v(A)$ asserting fairness assumptions about disjuncts of *Next*. However, other forms of system specifications are possible and can occasionally be useful. Asserting that a system (specified by) S satisfies a property F amounts to requiring 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 theorems asserted in module *SimpleAllocator* (Fig. 1) state three properties of the resource allocator.

System Refinement

TLA was designed to support stepwise system development based on a notion of *refinement*. In such an approach, a first, high-level specification formally states the problem at a high level of abstraction. A series of intermediate models then introduce detail, adding algorithmic ideas. The development is finished when a model is obtained that is detailed enough so that an implementation can be read off immediately or even mechanically generated (for example, based on models of shared-variable or message-passing systems). The fundamental requirement for useful notions of refinement is that they must preserve system properties, such that properties established at a higher level of abstraction are guaranteed to hold for later models, including the final implementation. In this way, crucial correctness properties can be proven (or errors can be detected) early on, simplifying their proofs or the correction of the model, and these properties need never be reproven for later refinements.

A lower-level model, expressed by a TLA formula C, preserves all TLA properties 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 is expressed at a lower level of abstraction, it will typically admit transitions that are invisible at the higher level, acting on state variables that do not appear in A. The stuttering invariance of TLA formulas is therefore essential to make validity of implication a reasonable definition of refinement.

Assume that we are given two system specifications *Abs* and *Conc* in the standard form,

$$Abs \triangleq \exists\, x : AInit \wedge \Box[ANext]_v \wedge AL \quad \text{and}$$
$$Conc \triangleq \exists\, y : CInit \wedge \Box[CNext]_w \wedge CL.$$

Proving that *Conc* refines *Abs* amounts to showing the validity of the implication $Conc \Rightarrow Abs$, and, using standard quantifier reasoning, this reduces to proving

$$(CInit \wedge \Box[CNext]_w \wedge CL) \;\Rightarrow\; (\exists\, x : AInit \wedge \Box[ANext]_v \wedge AL).$$

The standard approach to proving the latter implication is to define a state function t in terms of the free variables w (including y) of the left-hand side, and to prove

$$(CInit \wedge \Box[CNext]_w \wedge CL) \;\Rightarrow\; (AInit \wedge \Box[ANext]_v \wedge AL)[t/x].$$

In the computer science literature, the state function t is usually called a *refinement mapping*. Proof rules for refinement will be considered in some more detail in Sect. 4.5. A typical example of system refinement in TLA$^+$ will be given in Sect. 6.3, where a "distributed" model of the resource allocator will be developed that distinguishes between the actions of the allocator and those of the clients.

Composition of systems.

Stuttering invariance is also essential for obtaining a simple representation of the (parallel) composition of components, represented by their specifications. Assume that A and B are specifications of two components that we wish to compose in order to form a larger system. Each of these formulas describes the possible behaviors of the "part of the world" relevant to the respective component, represented by the state variables that have free occurrences in the specification of the component. A system that contains both components (possibly among other constituents) must therefore satisfy both A and B: composition is conjunction. Again, state transitions that correspond to a local action of one of the components are allowed because they are stuttering transitions of the other component. Any synchronisation between the two components is reflected in changes of a common state variable (the interfaces of the components), and these changes must be allowed by both components.

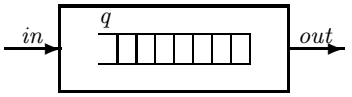
As a test of these ideas, consider the specification of a FIFO queue shown in Fig. 4 which is written in the canonical form of a TLA specification. The queue receives inputs via the channel *in* and sends its outputs via the channel *out*; it stores values that have been received but not yet sent in an internal queue q. Initially, we assume that the channels hold some "null" value and that the internal queue is empty. An enqueue action, described by the action *Enq*, is triggered by the reception of a new message (represented as a change of the input channel *in*); it appends the new input value to the internal queue. A dequeue action, specified by the action *Deq*, is possible whenever the internal queue is non-empty: the value at the head of the queue is sent over the channel *out* and removed from the queue.

We expect that two FIFO queues in a row implement another FIFO queue. Formally, let us assume that the two queues are connected by a channel *mid*, the above principles then lead us to expect that the formula[5]

$$Fifo[mid/out] \wedge Fifo[mid/in] \Rightarrow Fifo$$

will be valid. Unfortunately, this is not true, for the following reason: the formula *Fifo* implies that the *in* and *out* channels never change simultaneously,

[5] TLA$^+$ introduces concrete syntax, based on module instantiation, for writing substitutions such as $Fifo[mid/out]$.



(a) Pictorial representation

MODULE *InternalFIFO*

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

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

(b) Internal specification

MODULE *FIFO*

CONSTANT *Message*
VARIABLES *in, out*
$Internal(q) \triangleq$ INSTANCE *InternalFIFO*

$$Fifo \triangleq \exists\, q : Internal(q)!IFifo$$

(c) Interface specification

Fig. 4. Specification of a FIFO queue

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 to disallow simultaneous changes to its input and output interfaces, a specification style known as "interleaving specifications". In fact, the above argument shows that the composition of two queues specified in interleaving style does not implement an interleaving queue. The choice of an interleaving or a non-interleaving specification style is made by the person who writes the specification; interleaving specifications are usually found to be easier to write and to understand. The problem disappears if we explicitly add an "interleaving" assumption to the composition: the implication

$$
\begin{array}{l}
Fifo[mid/out] \land Fifo[mid/in] \land \Box[in' = in \lor out' = out]_{in,out} \\
\Rightarrow\ Fifo
\end{array}
\tag{1}
$$

is valid, and its proof will be considered in Sect. 4.5. Alternatively, one can write a non-interleaving specificationof a queue that allows input and output actions to occur simultaneously.

3.6 Variations and Extensions

We now 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 that this is 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 might consider $'$ as an operator, resembling the next-time modality of temporal logic. The two styles of presentation result in the same semantics of temporal formulas. The style adopted in this chapter corresponds well to the verification rules of TLA, explored in Sect. 4, where action-level hypotheses are considered as ordinary first-order formulas over an extended set of variables.

Compositional Verification

We have argued in Sect. 3.5 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 a specification is in the form of implications $E \Rightarrow M$ where E describes the assumptions about the environment and M the specification of the component. However, it turns out that often a stronger form of specification 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 $E \stackrel{+}{\triangleright} M$, admits a strong composition rule that can discharge mutual assumptions between components [4, 15]. It can be shown that the formula $E \stackrel{+}{\triangleright} M$ 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 [5, 7].

TLA*

The language of TLA distinguishes between bthe tiers of transition formulas and of temporal formulas; transition formulas must be guarded by "brackets"

to ensure stuttering invariance. Although a separation between the two tiers is natural when one is writing system specifications, it is not a prerequisite for obtaining stuttering invariance. The logic TLA* [36] generalizes TLA in that it distinguishes between *pure* and *impure* temporal formulas. Whereas pure formulas of TLA* contain impure formulas in the same way that temporal formulas of TLA contain transition formulas, impure formulas generalize transition formulas in that they admit 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 additional quantifiers. For example, the TLA* formula above is equivalent to the TLA formula[6]

$$\begin{aligned}\exists\, v \,:\; &\wedge\, \Box((v = c) \equiv \Diamond\langle B\rangle_u)\\ &\wedge\, \Box[A \Rightarrow v' = c]_t\end{aligned}$$

where c is a constant and v is a fresh flexible variable. TLA* thus offers a richer syntax without increasing the expressiveness, allowing high-level requirement specifications to be expressed more directly. (Kaminski [21] has shown that TLA* without quantification over flexible variables is strictly more expressive than the corresponding fragment of TLA.) Besides offering a more natural way to write temporal properties beyond standard system specifications, the propositional fragment of TLA* admits a straightforward complete axiomatization. (No complete axiomatization is known for propositional TLA, although Abadi [1] axiomatized an early version of TLA that was not invariant under stuttering.) 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* formulation 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

TLA can be considered as a fragment of the standard linear-time temporal logic (LTL) [34]. In particular, TLA does not include binary operators such as **until**. The main reason for that omission is the orientation of TLA towards

[6] Strictly, this equivalence is true only for universes that contain at least two distinct values; one-element universes are not very interesting.

writing specifications of state machines, where such operators are not necessary. Moreover, nested occurrences of binary temporal operators can be hard to interpret. Nevertheless, binary temporal operators are definable in TLA using quantification over flexible variables. For example, suppose that P and Q are state predicates whose free variables are in the tuple w of variables, that v is a flexible variable that does not appear in w, and that c is a constant. Then P **until** Q can be defined as the formula

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

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 similar to that shown abive for the translation of TLA* formulas.

4 Deductive System Verification in TLA

Because TLA formulas are used to describe systems as well as their properties, the proof rules for system verification are just logical axioms and rules of TLA. More precisely, a system described by a formula *Spec* has a property *Prop* if and only if every behavior that satisfies *Spec* also satisfies *Prop*, that is, iff the implication $Spec \Rightarrow Prop$ is valid. (To be really precise, the implication should be 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, whenever possible, to a proof of verification conditions expressed in the underlying predicate logic, a strategy that is commonly referred to as *assertional reasoning*. In this section, we present some typical rules and illustrate their use. We are not trying to be exhaustive, more information can be found in Lamport's original paper on TLA [27].

4.1 Invariants

Invariants characterize the set of states that can be reached during system execution; they are the basic form of safety properties and the starting point for any form of system verification. 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 whenever the hypothesis $I \wedge [N]_t \Rightarrow I'$ is valid as a transition formula, the conclusion $I \wedge \Box[N]_t \Rightarrow \Box I$ is a valid temporal formula. The hypothesis 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 the rule (INV1) is easily established by induction on behaviors. Because the hypothesis is a transition formula, it can be proven using ordinary first-order reasoning, including "data" axioms that characterize the intended interpretations.

For example, we can use the invariant rule (INV1) to prove the invariant $\Box(q \in \mathit{Seq}(\mathit{Message}))$ of the FIFO queue that was specified in module *InternalFIFO* in Fig. 4(b) on page 420. We have to prove

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

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

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

$$q \in \mathit{Seq}(\mathit{Message}) \wedge [\mathit{Next}]_{\mathit{vars}} \Rightarrow q' \in \mathit{Seq}(\mathit{Message}). \tag{4}$$

Because the empty sequence is certainly a finite sequence of messages, the proof obligation (3) follows from the definition of *Init* and appropriate data axioms. Similarly, the proof of (4) reduces to proving preservation of the invariant by the *Deq* and *Enq* actions, as well as under stuttering, and these proofs are again straightforward.

The proof rule (INV1) requires that the invariant I is *inductive*: it must be preserved by every possible system action. As with ordinary mathematical induction, it is usually necessary to strengthen the assertion and find an "induction hypothesis" that makes the proof go through. This idea is embodied in the following derived invariant rule

$$\frac{\mathit{Init} \Rightarrow I \qquad I \wedge [\mathit{Next}]_t \Rightarrow I' \qquad I \Rightarrow J}{\mathit{Init} \wedge \Box[\mathit{Next}]_t \Rightarrow \Box J}\,(\mathrm{INV}).$$

In this rule, I is an inductive invariant that implies J. The creative step consists in finding this inductive invariant. Typically, inductive invariants contain interesting "design" information about the model and represent the overall correctness idea. Some formal design methods, such as the B method [8] (see also the chapter by Cansell and Méry in this book) therefore demand that an inductive invariant be documented with the system model.

For example, suppose we wish to prove that any two consecutive elements of the queue are different. This property can be expressed in TLA$^+$ by the state predicate

$$J \triangleq \forall\, i \in 1..\mathit{Len}(q) - 1 : q[i] \neq q[i+1]$$

We have used some TLA$^+$ syntax for sequences in writing the formula J; in particular, a sequence s in TLA$^+$ is represented as a function whose values

can be accessed as $s[1], \ldots, s[Len(s)]$. The sequence formed of the values $e_1, \ldots, e_n$ is written as $\langle e_1, \ldots, e_n \rangle$, and the concatenation of two sequences s and t is written $s \circ t$.

The invariant rule (INV1) is not strong enough to prove that J is an invariant, because J is not necessarily preserved by the *Enq* step: there is no information about how the old value *in* of the input channel relates to the values in the queue. (Try this proof yourself to see why it fails.) The proof succeeds using the rule (INV) and the inductive invariant

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

which asserts that the current value of the input channel can be found either as the last element of the queue or (if the queue is empty) as the current value of the output channel.

4.2 Step Simulation

When one is proving refinement between two TLA specifications, a crucial step is to show that the next-state relation of the lower-level specification, expressed as $\Box[M]_t$, say, simulates the next-state relation $\Box[N]_u$ of the higher-level specification, up to stuttering. The following proof rule is used for this purpose; it relies on a previously proven state invariant I:

$$\frac{I \wedge I' \wedge [M]_t \Rightarrow [N]_u}{\Box I \wedge \Box[M]_t \Rightarrow \Box[N]_u}\,(\text{TLA2}).$$

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

$$\Box I \wedge \Box[M]_t \Rightarrow \Box[M \wedge I \wedge I']_t.$$

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

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

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

For this proof, we make use of the inductive invariant *Inv* introduced in Sect. 4.1 above. By rule (TLA2), we have to prove

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

The proof of (6) reduces to the three cases of a stuttering transition, an *Enq* action, and a *Deq* action. Only the last case is nontrivial. 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 $out' \neq out$.

4.3 Liveness Properties

Liveness properties, intuitively, assert that something good must eventually happen [10, 24]. Because formulas $\Box[N]_t$ are satisfied by a system that always stutters, the proof of liveness properties must ultimately rely on fairness properties of the specification that are assumed. TLA provides rules to deduce elementary liveness properties from the fairness properties assumed for 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:

$$\frac{\begin{array}{c} I \wedge I' \wedge P \wedge [N]_t \Rightarrow P' \vee Q' \\ I \wedge I' \wedge P \wedge \langle N \wedge A\rangle_t \Rightarrow Q' \\ I \wedge P \Rightarrow \textsc{Enabled}\, \langle A\rangle_t \end{array}}{\Box I \wedge \Box[N]_t \wedge \mathrm{WF}_t(A) \Rightarrow (P \rightsquigarrow Q)}\,(\mathrm{WF1})\,.$$

In this rule, I, 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" [33] 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 $\Box I \wedge \Box[N]_t \wedge \mathrm{WF}_t(A)$, and that P holds for state σ_i. We have to show that Q holds for 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 already become true before that happens). Finally, by the second premise, any $\langle A\rangle_t$-successor (which, by assumption, is in fact an $\langle N \wedge A\rangle_t$-successor) of a state satisfying P must satisfy Q, which proves the claim.

For our running example, we can use the 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

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

We shall use (WF1) to prove

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

where k and x are rigid variables. The following outline of the proof illustrates the application of the rule (WF1); the lower-level steps are all inferred by nontemporal reasoning and are omitted.

1. $at(k, x) \wedge [Next]_{vars} \Rightarrow at(k, x)' \vee out' = x \vee at(k-1, x)'$
   1.1. $at(k, x) \wedge m \in Message \wedge Enq \Rightarrow at(k, x)'$
   1.2. $at(k, x) \wedge Deq \wedge k = 1 \Rightarrow out' = x$
   1.3. $at(k, x) \wedge Deq \wedge k > 1 \Rightarrow at(k-1, x)'$
   1.4. $at(k, x) \wedge vars' = vars \Rightarrow at(k, x)'$
   1.5. Q.E.D.
      follows from steps 1.1–1.4 by the definitions of *Next* and $at(k, x)$.
2. $at(k, x) \wedge \langle Deq \wedge Next \rangle_{vars} \Rightarrow out' = x \vee at(k-1, x)'$
      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, the rule (WF1) cannot be used to prove the stronger property that every input to the queue will eventually be dequeued, expressed by the TLA formula

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

because there is no single "helpful action": the number of *Deq* actions necessary to produce the input element in 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 $(D, \succ)$, i.e. a binary relation such that there does not exist an infinite descending chain $d_1 \succ d_2 \succ \ldots$ of elements $d_i \in D$:

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

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 the rule (LATTICE) in a language of set theory. Also, we have taken the liberty of stating the assumption that $(D, \succ)$ is well-founded as if it were a logical hypothesis. As an illustration of the expressiveness of TLA, we observe in passing that, in principle, this hypothesis could be stated by the temporal formula

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

whose first conjunct expresses the irreflexivity of $\succ$ and whose second conjunct asserts that any sequence of values in D that can change only by decreasing with respect to $\succ$ must eventually become stationary. In system verification, however, well-foundedness is usually considered as a "data axiom" and is outside the scope of temporal reasoning.

Unlike the premises of the rules considered so far, the second hypothesis of the rule (LATTICE) is itself a temporal formula which requires that every occurrence of G, for any value $d \in D$, must be followed either by an occurrence of H or again by some G, for some smaller value e. Because there cannot be

an infinite descending chain of values in D, H must eventually become true. In applications of the rule (LATTICE), this hypothesis must be discharged by another rule for proving liveness, either a fairness rule such as (WF1) or another application of (LATTICE).

If we choose $(\mathbb{N}, >)$, the set of natural numbers with the standard "greater-than" relation, as the well-founded domain, the proof of the liveness property (8) that asserts that the FIFO queue eventually outputs every message it receives can be derived from property (7) and the invariant *Inv* of Sect. 4.1 using the rule (LATTICE).

Lamport [27] lists further (derived) rules for liveness properties, including introduction rules for proving formulas $\mathrm{WF}_t(A)$ and $\mathrm{SF}_t(A)$ that are necessary when proving refinement.

4.4 Simple Temporal Logic

The proof rules considered so far support the derivation of typical correctness properties of systems. In addition, TLA satisfies some standard axioms and rules of linear-time temporal logic that are useful when one is preparing for the application of verification rules. Figure 5 contains the axioms and rules of "simple temporal logic", adapted from [27].

$$\text{(STL1)}\quad \frac{F}{\Box F} \qquad \text{(STL4)}\quad \Box(F \Rightarrow G) \Rightarrow (\Box F \Rightarrow \Box G)$$

$$\text{(STL2)}\quad \Box F \Rightarrow F \qquad \text{(STL5)}\quad \Box(F \wedge G) \equiv (\Box F \wedge \Box G)$$

$$\text{(STL3)}\quad \Box\Box F \equiv \Box F \qquad \text{(STL6)}\quad \Diamond\Box(F \wedge G) \equiv (\Diamond\Box F \wedge \Diamond\Box G)$$

Fig. 5. Rules of simple temporal logic.

It can be shown that this is just a non-standard presentation of the modal logic S4.2 [19], implying that these laws by themselves, characterize a modal accessibility relation for $\Box$ that is reflexive, transitive, and locally convex (confluent). The latter 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.

Many derived laws of temporal logic are useful for system verification. Particularly useful are rules about the "leadsto" operator such as

$$\frac{F \Rightarrow G}{F \rightsquigarrow G}, \qquad \frac{F \rightsquigarrow G \qquad G \rightsquigarrow H}{F \rightsquigarrow H},$$

$$\frac{F \rightsquigarrow H \qquad G \rightsquigarrow H}{(F \vee G) \rightsquigarrow H}, \qquad \frac{F \Rightarrow \Box G \qquad F \rightsquigarrow H}{.F \rightsquigarrow (G \wedge H)}.$$

In principle, such temporal logic rules can be derived from the rules of Fig. 5. In practice, it can be easier to justify them from the semantics of tem-

poral logic. Because the validity of propositional temporal logic is decidable, they can be checked automatically by freely available tools.

4.5 Quantifier Rules

Although we have seen in Sect. 3.4 that the semantics of quantification over flexible variables is non-standard, the familiar proof rules of first-order logic are sound for both types of quantifier:

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

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

In these rules, x is a rigid and v is a flexible variable. The elimination rules $(\exists\,\mathrm{E})$ and $(\mathbf{\exists}\,\mathrm{E})$ require the usual proviso that the bound variable should not be free in the 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 the rule $(\exists\,\mathrm{I})$, we could prove

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

for any state variable v from the premise $\Box(v = v)$, provable by (STL1). However, the 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 $(\mathbf{\exists}\,\mathrm{I})$ and $(\mathbf{\exists}\,\mathrm{E})$ play a fundamental role in proofs of refinement for reactive systems. In this context, the "witness" t is often called a *refinement mapping* [2]. For example, the concatenation of the two low-level queues provides a suitable refinement mapping for proving the validity of the formula (1 on page 420), which claimed that two FIFO queues in a row implement a FIFO queue, assuming interleaving of changes to the input and output channels.

Although the quantifier rules are standard, one should recall from Sect. 3.2 that care has to be taken when substitutions are applied to formulas that contain implicit quantifiers. In particular, the formulas $\mathrm{WF}_t(A)$ and $\mathrm{SF}_t(A)$ contain the subformula $\textsc{Enabled}\,\langle A\rangle_t$, and therefore $\mathrm{WF}_t(A)[e/v]$ is not generally equivalent to the formula $\mathrm{WF}_{t[e/v]}(A[e/v, e'/v'])$. The consequences of this inequivalence for system verification are discussed in more detail in Lamport's original paper on TLA [27].

In general, refinement mappings need not always exist. For example, $(\mathbf{\exists}\,\mathrm{I})$ cannot be used to prove the TLA formula

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

which is valid, except over universes that contain a single element. Formula (10) asserts the existence of a flexible variable whose value changes infinitely

often. (Such a variable can be seen as an "oscillator", which would trigger system transitions.) In fact, an attempt to prove (10) by the rule ($\exists$ I) would require one to exhibit a state function t whose value is certain to change infinitely often in any behavior. Such a state function cannot exist: consider a behavior σ that ends in infinite stuttering, then the value of t never changes over the stuttering part of σ.

One approach to solving this problem, introduced in [2], consists of 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 using the rules ($\exists$ E) and ($\exists$ I) above to prove $G \wedge G_{aux} \Rightarrow \exists\, v : F$. The details are beyond the scope of this introductory overview.

5 Formalized Mathematics: The Added Value of TLA$^+$

The definitions of the syntax and semantics of TLA in Sect. 3 were given with respect to an arbitrary language of predicate logic and its interpretation. TLA$^+$ instantiates this generic definition of TLA with a specific first-order language, namely Zermelo–Fränkel set theory with choice. By adopting a standard interpretation, TLA$^+$ specifications are precise and unambiguous about the "data structures" on which specifications are based. We have seen in the example proofs in Sect. 4, that reasoning about data accounts for most of the steps that need to be proven during system verification. Besides fixing the vocabulary of the logical language and the intended interpretation, TLA$^+$ also introduces 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 chapter, they are just syntactic sugar. We shall therefore concentrate on the set-theoretic foundations, referring the reader to Lamport's book [29] for a detailed presentation of the language of TLA$^+$.

5.1 Elementary Data Structures: Basic Set Theory

Elementary set theory is based on a signature that consists of a single binary predicate symbol $\in$ and no function symbols. TLA$^+$ heavily relies on Hilbert's choice operator. The syntax of transition-level terms and formulas defined in Sect. 3.2 is therefore extended by an additional term formation rule that defines CHOOSE$x : A$ to be a transition function whenever $x \in \mathcal{V}_E$ is a variable and A is an action.[7] The occurrences of x in the term CHOOSE$x : A$ are bound.

[7] Temporal formulas are defined as indicated in Sect. 3.3 on page 413; in particular, CHOOSE is never applied to a temporal formula.

To this first-order language there 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 $\varepsilon(C)$ of C, and mapping the empty collection to an arbitrary value. The interpretation of a term CHOOSE$x : P$ is defined as

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

This definition employs the choice function to return some value satisfying P provided there is some such value in the universe of set theory. Observe that in this semantic clause, the choice function is applied to a collection that need not be a set (i.e., an element of the universe of the interpretation); in set-theoretic terminology, ε applies 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 the choice 'function' satisfies the laws

$$(\exists x : P) \equiv P[(\text{CHOOSE}x : P)/x] \tag{11}$$

$$(\forall x : (P \equiv Q)) \Rightarrow (\text{CHOOSE}x : P) = (\text{CHOOSE}x : Q) \tag{12}$$

TLA$^+$ avoids undefinedness by underspecification [18], and so CHOOSE$x : P$ denotes a value even if no value satisfies P. To ensure that a term involving choice actually denotes the expected value, the existence of some suitable value should be proven. If there is more than one such value, the expression is underspecified, and the user should be prepared to accept any of them. In particular, any properties should hold for all possible values. However, observe that for a given interpretation, choice is deterministic, and that it is not "monotone": no relationship can be established between CHOOSE$x : P$ and CHOOSE$x : Q$ even when $P \Rightarrow Q$ is valid (unless P and Q are actually equivalent). Therefore, whenever a specification *Spec* contains an underspecified application of choice, any refinement *Ref* is constrained to make the same choices in order to prove *Ref* $\Rightarrow$ *Spec*; this situation is fundamentally different from non-determinism, where implementations may narrow the set of allowed values.

In the following, we shall freely use many notational abbreviations of TLA$^+$. For example, $\exists x, y \in S : P$ abbreviates $\exists x : \exists y : x \in S \wedge y \in S \wedge P$. Local declarations are written as LET _ IN _, and IF _ THEN _ ELSE _ is used for conditional expressions.

From membership and choice, one can build up the conventional language of mathematics [32], and this is the foundation for the expressiveness of TLA$^+$.

Figure 6 on the following page lists some of the basic set-theoretic constructs of TLA$^+$; we write

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

to denote set enumeration and assume the additional bound variables in the defining expressions of Fig. 6 to be chosen such that no variable clashes occur. The two comprehension schemes act as binders for the variable x, which must

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

Fig. 6. Basic set-theoretic operators

not have free occurrences in S. The existence of sets defined in terms of choice can be justified from the axioms of Zermelo–Fränkel set theory [41], which provide the deductive counterpart of the semantics underlying TLA$^+$. However, it is well-known that without proper care, set theory is prone to paradoxes. For example, the expression

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

is a well-formed constant formula of TLA$^+$, but the existence of a set S containing precisely those sets which 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 values that are in bijection with the collection of all sets. Therefore, when one is evaluating the above TLA$^+$ expression, the choice function is applied to the empty collection, and the result depends on the underlying interpretation. Perhaps unexpectedly, however, we can infer from (12) that

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

Similarly, a generalized intersection operator dual to the union operator of Fig. 6 does not exist, because generalized intersection over the empty set of sets cannot be sensibly defined.

On the positive side, we have exploited the fact that no set can contain all values in the definition

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

that appears in Fig. 4(b) on page 420. Whatever set is denoted by *Message*, *NoMsg* will denote some value that is not contained in *Message*. If a subsequent refinement wanted to fix a specific "null" message value $\mathit{null} \notin \mathit{Message}$, it could do so by restricting the class of admissible interpretations via an assumption of the form

$$\text{ASSUME}(\text{CHOOSE}x \,:\, x \notin \mathit{Message}) = \mathit{null}$$

Because all properties established for the original specification hold for all possible choices of *NoMsg*, they will continue to hold for this restricted choice.

5.2 More Data Structures

Besides their use in elementary set operations, functions are a convenient way to represent different kinds of data structures. A traditional construction of functions within set theory, followed in Z and B [8,40], is to construct functions as special kinds of relations, which are represented as ordered pairs. TLA$^+$ takes a different approach: it assumes functions to be primitive and assumes tuples to be a particular kind of function. The set of functions whose domain equals S and whose codomain is a subset of T is written as $[S \to T]$, the domain of a function f is denoted by $\text{DOMAIN}\, f$, and the application of a function f to an expression e is written as $f[e]$. The expression $[x \in S \mapsto e]$ denotes a 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. (This expression can be understood as the TLA$^+$ syntax for a lambda expression $\lambda x \in S : e$.) Thus, 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 "function update" is central in Gurevich's ASM notation [12]; see also the chapter by Reisig in this book). 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. Again, this notation generalises to updates of a function at several argument positions; also, the notation @ can be used within the subexpression u to denote the original value of $f[t]$.

By combining choice, sets, and function notation, one obtains an expressive language for defining mathematical structures. For example, the standard TLA$^+$ module introducing natural numbers defines them as an arbitrary set with a constant zero and a successor function satisfying the usual Peano axioms [29, p. 345], and Lamport goes on to define similarly the integers and the real numbers, ensuring that the integers are a subset of the reals. In particular, the arithmetic operators over these sets are identical rather than just overloaded uses of the same symbols.

Recursive functions can be defined in terms of choice, for example

$$\begin{array}{l} \mathit{factorial} \;\triangleq \\ \quad \text{CHOOSE}f \,:\, f = [n \in \mathit{Nat} \mapsto \text{IF}\;\; n = 0 \;\; \text{THEN}\;\; 1 \;\; \text{ELSE}\;\; n * f[n-1]], \end{array}$$

which TLA$^+$, using some syntactic sugar, offers us to write 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, such a definition should be justified by proving the existence of a function that satisfies the recursive equation. Unlike the standard semantics of programming languages, TLA$^+$

does not commit to the least fixed point of a recursively defined function in cases where there are several solutions.

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 $\{j \in Nat : 1 \leq j \wedge j \leq n\}$ (and i is a "fresh" variable). Selection of the ith element of a tuple is therefore just a function application. Strings are defined as tuples of characters, and records are 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 concrete syntax of TLA$^+$ offers us special support for record operations. For example, one writes $acct.balance$ instead of $acct[\text{"balance"}]$.

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

Fig. 7. Finite sequences.

The standard TLA$^+$ module *Sequences* that has already appeared as a library module used for the specification of the FIFO queue in Fig. 4(b) on page 420, represents finite sequences as tuples. The definitions of the standard operations, some of which are shown in Fig. 7, are therefore quite simple. However, this simplicity can sometimes be deceptive. For example, these definitions do not reveal that the *Head* and *Tail* operations are "partial". They should be validated by proving the expected properties, such as

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

6 The Resource Allocator Revisited

Armed with a better understanding of the language TLA$^+$, let us reconsider the resource allocator specification of Sect. 2. We have already verified several properties of the simple allocator specification of Fig. 1 by model checking, and we could use the deduction rules of Sect. 4 to prove these properties in full generality. Does this mean that the specification is satisfactory?

Consider the following scenario: two clients c_1 and c_2 both request resources r_1 and r_2. The allocator grants r_1 to c_1 and r_2 to c_2. From our informal description in Sect. 2.1, it appears that we have reached a deadlock state: neither client can acquire the missing resource as long as the other one does not give up the resource it holds, which it is not required to do. Why then did TLC not report any deadlock, and how could we prove liveness?

Formally, the model contains no deadlock because, according to requirement (3), each client is allowed to give up resource it is holding. The problem with the model is that it actually *requires* clients to eventually give up the resources, even if they have not yet received the full share of resources that they asked for. This requirement is expressed by the seemingly innocous fairness condition

$$\forall\, c \in \mathit{Clients} \,:\, \mathrm{WF}_{\mathit{vars}}(\mathit{Return}(c, \mathit{alloc}[c])),$$

whereas the informal requirement (4) demands only that clients return their resources once their entire request has been satisfied. We should therefore have written

$$\forall c \in \mathit{Clients} \,:\, \mathrm{WF}_{\mathit{vars}}\big(\mathit{unsat}[c] = \{\} \wedge \mathit{Return}(c, \mathit{alloc}[c])\big).$$

Rerunning TLC on the modified specification produces the expected counterexample.

The bigger lesson of this example is that errors can creep into formal specifications just as easily as into programs, and that a model can be inappropriate even if it satisfies all correctness properties. Validation, for example by simulation runs or a review of the model, is extremely important for avoiding this kind of error. We will now revisit the allocator specification and present a corrected model. We will then present a refinement of that model that prepares an implementation as a distributed system.

6.1 A Scheduling Allocator

The specification *SimpleAllocator* (Fig. 1 on page 404) is too simple because the allocator is free to allocate resources in any order. Therefore, it may "paint itself into a corner", requiring cooperation from the clients to recover. We can prevent this from happening by having the allocator fix a schedule according to which access to resources will be granted. Figures 8 on the next page and 9 on page 437 contain a formal TLA$^+$ model based on this idea.

Compared with the specification of the simple allocator in Fig. 1 on page 404, the new specification contains two more state variables, *pool* and *sched*. The variable *sched* contains a sequence of clients, representing the allocation schedule. The variable *pool* contains a set of clients that have requested resources but have not yet been scheduled for allocation. Consequently, the request action merely inserts the client into the pool. The allocation action is restricted to giving out resources to a client only if no client that appears earlier in the schedule is demanding any of them.

```
MODULE SchedulingAllocator
EXTENDS FiniteSet, Sequences, Naturals
CONSTANTS Clients, Resources
ASSUME IsFiniteSet(Resources)
VARIABLES unsat, alloc, pool, sched

TypeInvariant ≜
  ∧ unsat ∈ [Clients → SUBSET Resources]
  ∧ alloc ∈ [Clients → SUBSET Resources]
  ∧ pool ∈ SUBSET Clients ∧ sched ∈ Seq(Clients)
available ≜ Resources \ (UNION{alloc[c] : c ∈ Clients})
PermSeqs(S) ≜ set of permutations of finite set S, represented as sequences
  LET perms[ss ∈ SUBSET S] ≜
        IF ss = {} THEN ⟨⟩
        ELSE LET ps ≜ [x ∈ ss ↦ {Append(sq, x) : sq ∈ perms[ss \ {x}]}]
             IN UNION{ps[x] : x ∈ ss}
  IN perms[S]
Drop(seq, i) ≜ SubSeq(seq, 1, i − 1) ∘ SubSeq(seq, i + 1, Len(seq))

Init ≜
  ∧ unsat = [c ∈ Clients ↦ {}] ∧ alloc = [c ∈ Clients ↦ {}]
  ∧ pool = {} ∧ sched = ⟨⟩
Request(c, S) ≜
  ∧ unsat[c] = {} ∧ alloc[c] = {} ∧ S ≠ {}
  ∧ unsat′ = [unsatEXCEPT![c] = S] ∧ pool′ = pool ∪ {c}
  ∧ UNCHANGED⟨alloc, sched⟩
Allocate(c, S) ≜
  ∧ S ≠ {} ∧ S ⊆ available ∩ unsat[c]
  ∧ ∃ i ∈ DOMAIN sched :
      ∧ sched[i] = c ∧ ∀ j ∈ 1..i − 1 : unsat[sched[j]] ∩ S = {}
      ∧ sched′ = IF S = unsat[c] THEN Drop(sched, i) ELSE sched
  ∧ alloc′ = [allocEXCEPT![c] = @ ∪ S] ∧ unsat′ = [unsatEXCEPT![c] = @ \ S]
  ∧ UNCHANGEDpool
Return(c, S) ≜
  ∧ S ≠ {} ∧ S ⊆ alloc[c]
  ∧ alloc′ = [allocEXCEPT![c] = @ \ S]
  ∧ UNCHANGED⟨unsat, pool, sched⟩
Schedule ≜
  ∧ pool ≠ {} ∧ pool′ = {}
  ∧ ∃ sq ∈ PermSeqs(pool) : sched′ = sched ∘ sq
  ∧ UNCHANGED⟨unsat, alloc⟩
Next ≜
  ∨ ∃c ∈ Clients, S ∈ SUBSET Resources :
      Request(c, S) ∨ Allocate(c, S) ∨ Return(c, S)
  ∨ Schedule
vars ≜ ⟨unsat, alloc, pool, sched⟩
```

Fig. 8. Specification of an allocator with scheduling (part 1 of 2)

$$
\begin{array}{lll}
\mathit{Allocator} & \triangleq & \land\ \mathit{Init} \land \Box[\mathit{Next}]_{\mathit{vars}} \\
& & \land\ \forall c \in \mathit{Clients} : \mathrm{WF}_{\mathit{vars}}(\mathit{unsat}[c] = \{\} \land \mathit{Return}(c, \mathit{alloc}[c])) \\
& & \land\ \forall c \in \mathit{Clients} : \mathrm{WF}_{\mathit{vars}}(\exists S \in \text{SUBSET } \mathit{Resources} : \mathit{Allocate}(c, S)) \\
& & \land\ \mathrm{WF}_{\mathit{vars}}(\mathit{Schedule})
\end{array}
$$

Fig. 9. Specification of an allocator with scheduling (part 2 of 2)

The specification contains a new action *Schedule*, which establishes the allocation schedule. Because this is a high-level specification, we do not commit to any specific scheduling policy: instead we show the protocol to be correct if the processes in the pool are scheduled in an arbitrary order. The auxiliary operator $\mathit{PermSeqs}(S)$ recursively computes the set of permutation sequences of a finite set S. The idea is that $\langle x_1, \ldots, x_n \rangle$ is a permutation of a non-empty finite set S if and only if $\langle x_1, \ldots, x_{n-1} \rangle$ is a permutation of $S \setminus \{x_n\}$. The formal expression in TLA$^+$ makes use of an auxiliary, recursively defined function *perms* that computes the set of permutations $\mathit{perms}[T]$ of any subset $T \subseteq S$, in a style that is similar to the recursive definition of functions over inductive data types in a functional programming language. We could have used a simpler, more declarative definition of the action *Schedule*, such as

$$
\begin{array}{l}
\mathit{Schedule} \triangleq \\
\quad \land\ \mathit{pool} \neq \{\} \land \mathit{pool}' = \{\} \\
\quad \land\ \exists\, \mathit{sq} \in \mathit{Seq}(\mathit{Clients}) : \land\ \{\mathit{sq}[i] : i \in \text{DOMAIN } \mathit{sq}\} = \mathit{pool} \\
\qquad\qquad\qquad\qquad\qquad\ \land\ \forall\, i, j \in 1..\mathit{Len}(\mathit{sq}) : \mathit{sq}[i] = \mathit{sq}[j] \Rightarrow i = j \\
\quad \land\ \text{UNCHANGED}\langle \mathit{unsat}, \mathit{alloc} \rangle.
\end{array}
$$

In this formulation, the schedule is simply required to be any injective sequence (containing no duplicates) formed from the elements of *pool*. The two definitions are logically equivalent. However, this definition would not be acceptable for TLC, because the set $\mathit{Seq}(\mathit{Clients})$ is infinite, even if *Clients* is finite.

Looking at the fairness conditions, observe that the fairness requirement on the return action has been amended as indicated above, so that it agrees with the informal specification. The fairness condition for the allocation action is similar to the one adopted for the simple allocator specification, but with weak fairness substituted for strong fairness. The idea behind this change is that the non-determinism present in the original specification has been resolved by the introduction of the allocation schedule, so that the simpler condition now suffices. (Of course, this intuition will have to be formally verified!) There is an additional weak fairness requirement for the scheduling action, asserting that the allocator should periodically update its schedule when new clients have issued requests.

6.2 Analysis Using Model Checking

We can again ask TLC to verify the safety and liveness properties described in Sect. 2.3. For an instance consisting of three clients and two resources, TLC computes 1690 distinct states and requires about 30 seconds for verification. What sets TLC apart from more conventional model checkers is its ability to evaluate an input language in which models can be expressed at the high level of abstraction that was used in Figs. 8 on page 436 and 9 on the previous page: neither the definition of the operator *PermSeqs* nor the relatively complicated fairness constraints pose a problem. (For better efficiency, we could override the definition of *PermSeqs* by a method written in Java, but this is not a big concern for a list that contains at most three elements.)

Given our experience with the verification of the simple allocator model, one should be suspicious of the quick success obtained with the new model. As Lamport [29, Sect. 14.5.3] writes, it is a good idea to verify as many properties as possible.

Figure 10 contains a lower-level invariant of the scheduling allocator that can be verified using TLC.

$$
\begin{array}{l}
\mathit{UnscheduledClients} \triangleq \quad \text{set of clients that are not in the schedule} \\
\quad \mathit{Clients} \setminus \{\mathit{sched}[i] : i \in \text{DOMAIN } \mathit{sched}\} \\
\mathit{PrioResources}(i) \triangleq \quad \text{bound on resources requested by } i\text{-th client in schedule} \\
\quad \mathit{available} \\
\quad \cup \text{ UNION}\{\mathit{unsat}[\mathit{sched}[j]] \cup \mathit{alloc}[\mathit{sched}[j]] : j \in 1..i-1\} \\
\quad \cup \text{ UNION}\{\mathit{alloc}[c] : c \in \mathit{UnscheduledClients}\} \\
\mathit{AllocatorInvariant} \triangleq \\
\quad \land \forall\, c \in \mathit{pool} : \mathit{unsat}[c] \neq \{\} \ \land \ \mathit{alloc}[c] = \{\} \\
\quad \land \forall\, i \in \text{DOMAIN } \mathit{sched} : \land \mathit{unsat}[\mathit{sched}[i]] \neq \{\} \\
\qquad\qquad\qquad\qquad\quad \land \forall\, j \in 1..i-1 : \mathit{alloc}[\mathit{sched}[i]] \cap \mathit{unsat}[\mathit{sched}[j]] = \{\} \\
\qquad\qquad\qquad\qquad\quad \land \mathit{unsat}[\mathit{sched}[i]] \subseteq \mathit{PrioResources}(i)
\end{array}
$$

Fig. 10. Lower-level invariant of the scheduling allocator

The first conjunct of the formula *AllocatorInvariant* says that all clients in the set *pool* have requested resources, but do not hold any. The second conjunct concerns the clients in the schedule, It is split into three subconjuncts: first, each client in the schedule has some outstanding requests, second, no client may hold a resource that is requested by a prioritized client (appearing earlier in the schedule); and, finally, the set of outstanding requests of a client in the schedule is bounded by the union of the set of currently available resources, the resources requested or held by prioritized clients, and the resources held by clients that do not appear in the schedule. The idea behind this last conjunct is to assert that a client's requests can be satisfied using resources that either are already free or are held by prioritized clients. It follows that prioritized clients can obtain their full set of resources, after which they are required to

eventually release them again. Therefore, the scheduling allocator works correctly even under the worst-case assumption that clients will give up resources only after their complete request has been satisfied.

Verification by Refinement

Beyond these correctness properties, TLC can also establish a formal refinement relationship between the two allocator specifications. The scheduling allocator operates under some additional constraints. Moreover, it introduces the variable *sched*, which did not appear in the specification of the simple allocator, and which is therefore not constrained by that specification. More interestingly, the scheduling policy and the (weaker) liveness assumptions imply that the (original) fairness constraints are effectively met. The scheduling allocator therefore turns out to be a refinement of the simple allocator, implying the correctness properties by transitivity!

We can use TLC to verify this refinement, for small finite instances, using the module *AllocatorRefinement* that appears in Fig. 11.

MODULE *AllocatorRefinement*

EXTENDS *SchedulingAllocator*

Simple $\triangleq$ INSTANCE *SimpleAllocator*

SimpleAllocator $\triangleq$ *Simple*!*SimpleAllocator*

THEOREM *Allocator* $\Rightarrow$ *SimpleAllocator*

Fig. 11. Asserting a Refinement Relationship.

This module extends the module *SchedulingAllocator*, thus importing all declarations and definitions of that module, and defines an instance *Simple* of the module *SimpleAllocator*, whose parameters are (implicitly) instantiated by the entities of the same name inherited from module *SchedulingAllocator*. All operators *Op* defined in the instance are available as *Simple*!*Op*. (It would have been illegal to extend both modules *SchedulingAllocator* and *SimpleAllocator* because they declare constants and variables, as well as define operators, with the same names.) The module then asserts that specification *Allocator* implies the specification *SimpleAllocator*. In order to have this implication checked by TLC, we again defined an instance consisting of three clients and two resources and stipulate

```
SPECIFICATION Allocator
PROPERTIES SimpleAllocator
```

in the configuration file. TLC found the implication to be valid, requiring just 6 seconds.

6.3 Towards a Distributed Implementation

The specification *Allocator* defined in the module *SchedulingAllocator* of Figs. 8 on page 436 and 9 on page 437 describes an overall algorithm (or, rather, a class of algorithms) for resource allocation; analysis by TLC has indicated that this algorithm satisfies the desired correctness properties, even under worst-case assumptions about the clients' behavior. However, the model does not indicate the architecture of the system as a set of independent, communicating processes. Our next goal is therefore to refine that specification into one that is implementable as a distributed system. In particular, we shall assume that the allocator and the clients may run on different processors. Therefore, each process should have direct access only to its local memory, and explicit, asynchronous message passing will be used to communicate with other processes. Instead of a centralized representation of the system state based on the variables *unsat* and *alloc*, we will distinguish between the allocator's view and each client's view of its pending requests and allocated resources. Similarly, the basic actions such as the request for resources will be split into two parts, with different processes being responsible for carrying them out: in the first step, the client issues a request, updates its local state, and sends a corresponding message to the allocator. Subsequently, the allocator receives the message and updates its table of pending requests accordingly.

Figures 12 on the next page and 13 on page 442 contain a TLA$^+$ model based on this idea. This model contains variables *unsat*, *alloc*, and *sched* as before, but these are now considered to be local variables of the allocator. New variables *requests* and *holding* represent the clients' views of pending resource requests and of resources currently held; we interpret $requests[c]$ and $holding[c]$ as being local to the client process c. The communication network is (very abstractly) modeled by the variable *network* which holds the set of messages in transit between the different processes.

Except for the action *Schedule*, which is a private action of the allocator, all of the actions that appeared in the specification *SchedulingAllocator* have been split into two actions as explained above. For example, client c is considered to perform the action $Request(c, S)$ because only its local variables and the state of the communication network are modified by the action. The allocator later receives the request message m and performs the action $RReq(m)$. The fairness conditions of our previous specification are complemented by weak fairness requirements for the actions $RReq(m)$, $RAlloc(m)$, and $RRet(m)$ which are associated with message reception (for all possible messages m); these requirements express the condition that messages will eventually be received and handled.

The observant reader may be somewhat disappointed with the form of the specification of this "distributed" implementation because the formula *Implementation* is again written in the standard form

$$Init \wedge \Box[Next]_v \wedge L$$

MODULE *AllocatorImplementation*

EXTENDS *FiniteSets, Sequences, Naturals*
CONSTANTS *Clients, Resources*
ASSUME *IsFiniteSet(Resources)*
VARIABLES *unsat, alloc, sched, requests, holding, network*
$Sched \triangleq$ INSTANCE *SchedulingAllocator*

$Messages \triangleq$
  $[type : \{\text{“request”}, \text{“allocate”}, \text{“return”}\}, clt : Clients, rsrc : \text{SUBSET } Resources]$
$TypeInvariant \triangleq$
  $\land\ Sched!TypeInvariant$
  $\land\ requests \in [Clients \to \text{SUBSET } Resources]$
  $\land\ holding \in [Clients \to \text{SUBSET } Resources]$
  $\land\ network \in \text{SUBSET } Messages$

$Init \triangleq$
  $\land\ Sched!Init$
  $\land\ requests = [c \in Clients \mapsto \{\}] \land holding = [c \in Clients \mapsto \{\}] \land network = \{\}$
$Request(c, S) \triangleq$ client c requests set S of resources
  $\land\ requests[c] = \{\} \land holding[c] = \{\} \land S \neq \{\}$
  $\land\ requests' = [requests \text{EXCEPT}![c] = S]$
  $\land\ network' = network \cup \{[type \mapsto \text{“request”}, clt \mapsto c, rsrc \mapsto S]\}$
  $\land\ \text{UNCHANGED}\langle unsat, alloc, sched, holding\rangle$
$RReq(m) \triangleq$ allocator handles request message sent by some client
  $\land\ m \in network \land m.type = \text{“request”} \land network' = network \setminus \{m\}$
  $\land\ unsat' = [unsat \text{EXCEPT}![m.clt] = m.rsrc]$
  $\land\ \text{UNCHANGED}\langle alloc, sched, requests, holding\rangle$
$Allocate(c, S) \triangleq$ allocator decides to allocate resources S to client c
  $\land\ Sched!Allocate(c, S)$
  $\land\ network' = network \cup \{[type \mapsto \text{“allocate”}, clt \mapsto c, rsrc \mapsto S]\}$
  $\land\ \text{UNCHANGED}\langle requests, holding\rangle$
$RAlloc(m) \triangleq$ some client receives resource allocation message
  $\land\ m \in network \land m.type = \text{“allocate”} \land network' = network \setminus \{m\}$
  $\land\ holding' = [holding \text{EXCEPT}![m.clt] = @ \cup m.rsrc]$
  $\land\ requests' = [requests \text{EXCEPT}![m.clt] = @ \setminus m.rsrc]$
  $\land\ \text{UNCHANGED}\langle unsat, alloc, sched\rangle$
$Return(c, S) \triangleq$ client c returns resources in S
  $\land\ S \neq \{\} \land S \subseteq holding[c]$
  $\land\ holding' = [holding \text{EXCEPT}![c] = @ \setminus S]$
  $\land\ network' = network \cup \{[type \mapsto \text{“return”}, clt \mapsto c, rsrc \mapsto S]\}$
  $\land\ \text{UNCHANGED}\langle unsat, alloc, sched, requests\rangle$
$RRet(m) \triangleq$ allocator receives returned resources
  $\land\ m \in network \land m.type = \text{“return”} \land network' = network \setminus \{m\}$
  $\land\ alloc' = [alloc \text{EXCEPT}![m.clt] = @ \setminus m.rsrc]$
  $\land\ \text{UNCHANGED}\langle unsat, sched, requests, holding\rangle$
$Schedule \triangleq Sched!Schedule \land \text{UNCHANGED}\langle requests, holding, network\rangle$

Fig. 12. An implementation of the allocator (part 1 of 2)

$$
\begin{array}{l}
\mathit{Next} \triangleq \\
\quad \lor \exists\, c \in \mathit{Clients}, S \in \text{SUBSET } \mathit{Resources} : \\
\qquad\quad \mathit{Request}(c, S) \lor \mathit{Allocate}(c, S) \lor \mathit{Return}(c, S) \\
\quad \lor \exists\, m \in \mathit{network} : \mathit{RReq}(m) \lor \mathit{RAlloc}(m) \lor \mathit{RRet}(m) \\
\quad \lor \mathit{Schedule} \\
\mathit{vars} \triangleq \langle \mathit{unsat}, \mathit{alloc}, \mathit{sched}, \mathit{requests}, \mathit{holding}, \mathit{network} \rangle \\
\hline
\mathit{Liveness} \triangleq \\
\quad \land \forall\, c \in \mathit{Clients} : \mathrm{WF}_{\mathit{vars}}(\mathit{requests}[c] = \{\} \land \mathit{Return}(c, \mathit{holding}[c])) \\
\quad \land \forall\, c \in \mathit{Clients} : \mathrm{WF}_{\mathit{vars}}(\exists\, S \in \text{SUBSET } \mathit{Resources} : \mathit{Allocate}(c, S)) \\
\quad \land \mathrm{WF}_{\mathit{vars}}(\mathit{Schedule}) \\
\quad \land \forall\, m \in \mathit{Messages} : \\
\qquad\quad \mathrm{WF}_{\mathit{vars}}(\mathit{RReq}(m)) \land \mathrm{WF}_{\mathit{vars}}(\mathit{RAlloc}(m)) \land \mathrm{WF}_{\mathit{vars}}(\mathit{RRet}(m)) \\
\mathit{Implementation} \triangleq \mathit{Init} \land \Box[\mathit{Next}]_{\mathit{vars}} \land \mathit{Liveness} \\
\hline
\text{THEOREM } \mathit{Implementation} \Rightarrow \mathit{Sched}\,!\,\mathit{Allocator}
\end{array}
$$

Fig. 13. An implementation of the allocator (part 2 of 2)

that we have seen so often in this chapter. From the discussion of system composition as conjunction in Sect. 3.5, one might have expected to see a conjunction of specifications, one for each process. There are two technical problems with doing so. First, the clients' variables *requests* and *holding* are represented as arrays such that each client accesses only the corresponding array field. The specification of client c should really only specify $\mathit{requests}[c]$ and $\mathit{holding}[c]$, but the composition should ensure type correctness and ensure that the remaining array fields remain unchanged. This is possible, but cumbersome to write down. (Lamport discusses this issue in more detail in [29, Chap. 10].) Second, the current implementation of TLC expects specifications in the standard form and does not handle conjunctions of process specifications.

The module *AllocatorImplementation* claims that the model obtained in this way is a refinement of the scheduling allocator specification, and we can again use TLC to verify this theorem for finite instances. However, TLC quickly produces a counterexample that ends in the states shown in Fig. 14.

In state 7, client `c1` has returned resource `r1` to the allocator. In the transition to state 8, it issues a new request for the same resource, which is handled by the allocator (according to the action *RReq*) in the transition to state 9. This action modifies the variable *unsat* at position `c1` although the value of $\mathit{alloc}[\texttt{c1}]$, is not the empty set – a transition that is not allowed by the scheduling allocator.

Intuitively, the problem is due to the asynchronous communication network underlying our model, which makes the allocator receive and handle the request message before it receives the earlier return message. Indeed, it is easy to see that if one allowed the allocator to handle the new request before releasing the old one, it might become confused and deregister `r1` for client `c1` even though the client still held the resource (granted in response to the second

```
STATE 7:
/\ holding = (c1 :> {} @@ c2 :> {} @@ c3 :> {})
/\ alloc = (c1 :> {r1} @@ c2 :> {} @@ c3 :> {})
/\ requests = (c1 :> {} @@ c2 :> {} @@ c3 :> {})
/\ sched = <<  >>
/\ network = {[type |-> "return", clt |-> c1, rsrc |-> {r1}]}
/\ unsat = (c1 :> {} @@ c2 :> {} @@ c3 :> {})

STATE 8:
/\ holding = (c1 :> {} @@ c2 :> {} @@ c3 :> {})
/\ alloc = (c1 :> {r1} @@ c2 :> {} @@ c3 :> {})
/\ requests = (c1 :> {r1} @@ c2 :> {} @@ c3 :> {})
/\ sched = <<  >>
/\ network = { [type |-> "request", clt |-> c1, rsrc |-> {r1}],
  [type |-> "return", clt |-> c1, rsrc |-> {r1}] }
/\ unsat = (c1 :> {} @@ c2 :> {} @@ c3 :> {})

STATE 9:
/\ holding = (c1 :> {} @@ c2 :> {} @@ c3 :> {})
/\ alloc = (c1 :> {r1} @@ c2 :> {} @@ c3 :> {})
/\ requests = (c1 :> {r1} @@ c2 :> {} @@ c3 :> {})
/\ sched = <<  >>
/\ network = {[type |-> "return", clt |-> c1, rsrc |-> {r1}]}
/\ unsat = (c1 :> {r1} @@ c2 :> {} @@ c3 :> {})
```

Fig. 14. Model checking the correctness of the implementation.

request). It depends on the underlying communication network whether such a race condition can occur or not. If messages between any pair of processes are delivered in order, the TLA$^+$ model could represent the communication network as a set of message queues. If communication is truly asynchronous and message order is not guaranteed, one should add the precondition

$$alloc[m.clt] = \{\}$$

to the definition of the action $RReq(m)$ so that a new request will be processed only after the return message corresponding to the previous grant has been received. With this correction, TLC confirms the refinement theorem for our small instance in about 2 minutes.

Finally, we can assert the invariant shown in Fig. 15 to confirm our intuition about how the variables associated with the clients and the allocator relate to each other. The verification of this invariant for the usual small instance of the model with three clients and two resources generates 64 414 states (17 701 of which are distinct) and takes about 12 seconds.

$$
\begin{array}{l}
\mathit{RequestsInTransit}(c) \triangleq \quad \text{requests sent by c but not yet received} \\
\quad \{\mathit{msg}.\mathit{rsrc} : \mathit{msg} \in \{m \in \mathit{network} : m.\mathit{type} = \text{“request”} \wedge m.\mathit{clt} = c\}\} \\
\mathit{AllocsInTransit}(c) \triangleq \quad \text{allocations sent to c but not yet received} \\
\quad \{\mathit{msg}.\mathit{rsrc} : \mathit{msg} \in \{m \in \mathit{network} : m.\mathit{type} = \text{“allocate”} \wedge m.\mathit{clt} = c\}\} \\
\mathit{ReturnsInTransit}(c) \triangleq \quad \text{return messages sent by c but not yet received} \\
\quad \{\mathit{msg}.\mathit{rsrc} : \mathit{msg} \in \{m \in \mathit{network} : m.\mathit{type} = \text{“return”} \wedge m.\mathit{clt} = c\}\} \\
\mathit{Invariant} \triangleq \forall\, c \in \mathit{Clients} : \\
\quad \wedge\ \mathit{Cardinality}(\mathit{RequestsInTransit}(c)) \leq 1 \\
\quad \wedge\ \mathit{requests}[c] = \mathit{unsat}[c] \\
\qquad\qquad\qquad \cup\ \text{UNION}\,\mathit{RequestsInTransit}(c) \\
\qquad\qquad\qquad \cup\ \text{UNION}\,\mathit{AllocsInTransit}(c) \\
\quad \wedge\ \mathit{alloc}[c] = \mathit{holding}[c] \\
\qquad\qquad\qquad \cup\ \text{UNION}\,\mathit{AllocsInTransit}(c) \\
\qquad\qquad\qquad \cup\ \text{UNION}\,\mathit{ReturnsInTransit}(c)
\end{array}
$$

Fig. 15. Relating the allocator and client variables by an invariant

6.4 Some Lessons Learnt

Starting from the informal requirements for the allocator problem presented in Sect. 2.1, it would have been tempting to come up directly with a model similar to the "implementation" presented in Sect. 6.3, or even a more detailed one. However, a low-level specification is at least as likely to contain errors as a program, and the whole purpose of modeling is to clarify and analyse a system at an adequate level of abstraction. The seemingly trivial *SimpleAllocator* specification in Fig. 1 on page 404 helped us discover the need to fix a schedule for resource allocation. It also illustrated the need to validate models: success in model checking (or proving) correctness properties by itself does not guarantee that the model is meaningful. A similar problem would have been more difficult to detect at the level of detail of the final specification, where there are additional problems of synchronisation and message passing to worry about. The specification *SchedulingAllocator* introduced the idea of determining a schedule and thereby fixed the problem in the original specification while remaining at the same high level of abstraction. Finally, the module *AllocatorImplementation* introduced a step towards a possible implementation by attributing the state variables and the actions to separate processes, and by introducing explicit communication.

For each model, TLC was of great help in analysing various properties. Although only small instances can be handled by model checking before running into the state explosion problem, doing so greatly increases one's confidence in the models. Variants of the specifications can be checked without great effort, and various properties (invariants and more general temporal properties) can be verified in a single run. Deductive verification, based on the proof rules of Sect. 4, can then establish system properties in a fully rigorous way. In our own work, we have defined a format of "predicate diagrams" for TLA$^+$ specifications [13]. We have found these diagrams to be helpful in determining

appropriate fairness hypotheses. The format is supported by a tool [17] that uses model checking to identify abstract counter-examples, indicating either too weak an abstraction or missing fairness or ordering annotations.

7 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 in the realm of logics and (formalized) mathematics. The rôle of a formalism is to *help* the user in the difficult and important activity of writing and analysing formal models. TLA$^+$ builds on the experience of classical mathematics and adds a thin layer of temporal logic in order to describe system executions, in particular to express fairness properties. A distinctive feature of TLA is its attention to refinement and composition, reflected in the concept of stuttering invariance. Unlike property-oriented specification languages based on temporal logic, TLA favors the specification of systems as state machines, augmented by fairness conditions and by hiding.

Whereas the expressiveness of TLA$^+$ undoubtedly helps in writing concise, high-level models of systems, it is not so clear a priori that 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. These problems can, to some extent, be mastered by adhering to standard idioms, such as primitive-recursive definitions, that ensure well-definedness. For the specification of reactive systems, TLA adds some proper idioms that control the delicate interplay between temporal operators. For example, restricting fairness conditions to subactions of the next-state relation ensures that a specification is machine closed [3], i.e., that its allowed behavior is entirely described by the initial condition and its next-state relation. Having an expressive specification language is also helpful when new classes of systems arise. For example, Abadi and Lamport [3] have described a format for specifying real-time systems in TLA$^+$, and Lamport [30] describes how discrete real-time systems can be verified using TLC.

The main tool supporting TLA$^+$ is the model checker TLC [43]. It can analyse system specifications in standard form, written in a sublanguage of TLA$^+$ that ensures that the next-state relation can be effectively computed. All the TLA$^+$ specifications that appeared in this chapter fall into this fragment, and indeed the input language of TLC is more expressive than that of most other model checkers. Deductive verification of 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 [16,20,35], although no prover is yet available that fully supports TLA$^+$.

Lamport has recently defined the language $^+$CAL, a high-level algorithmic language for describing concurrent and distributed algorithms. The ex-

pressions of $^+$CAL are those of TLA$^+$, but $^+$CAL provides standard programming constructs such as assignment, sequencing, conditionals, loops, nondeterministic choice, and procedures. The $^+$CAL compiler generates a TLA$^+$ specification from a $^+$CAL program which can then be verified using TLC [31]. A useful complement could be the generation of executable code from a fragment of $^+$CAL for specific execution platforms.

Acknowledgements.

I am indebted to Leslie Lamport for providing the subject of this article, for his encouragement of this work, and for his detailed comments on earlier versions. Parts of this chapter have their roots in an earlier paper on TLA, written with Martín Abadi [6]. I have had the opportunity on several occasions to teach about TLA$^+$, and fruitful discussions with students helped me prepare this chapter.

References

1. M. Abadi. An axiomatization of Lamport's Temporal Logic of Actions. In J.C.M. Baeten and J.W. Klop, editors, *CONCUR '90, Theories of Concurrency: Unification and Extension*, volume 458 of Lecture Notes in Computer Science, pages 57–69. Springer, 1990.
2. M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 81(2):253–284, May 1991.
3. M. Abadi and L. Lamport. An old-fashioned recipe for real time. *ACM Transactions on Programming Languages and Systems*, 16(5):1543–1571, Sept. 1994.
4. M. Abadi and L. Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, May 1995.
5. M. Abadi and S. Merz. An abstract account of composition. In J. Wiedermann and P. Hajek, editors, *Mathematical Foundations of Computer Science*, volume 969 of Lecture Notes in Computer Science, pages 499–508. Springer, 1995.
6. M. Abadi and S. Merz. On TLA as a logic. In M. Broy, editor, *Deductive Program Design*, NATO ASI Series F, pages 235–272. Springer, 1996.
7. M. Abadi and G. Plotkin. A logical view of composition. *Theoretical Computer Science*, 114(1):3–30, June 1993.
8. J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
9. 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.
10. B. Alpern and F.B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, Oct. 1985.
11. R. Back and J. von Wright. *Refinement calculus—A systematic introduction*. Springer, 1998.
12. E.Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer, 2003.

13. D. Cansell, D. Méry and S. Merz. Diagram refinements for the design of reactive systems. *Journal of Universal Computer Science*, 7(2):159–174, 2001.
14. E.M. Clarke, O. Grumberg and D. Peled. *Model Checking*. MIT Press, Cambridge, Mass., 1999.
15. W.-P. de Roever, H. Langmaack, and A. Pnueli, editors. *Compositionality: The Significant Difference*, volume 1536 of *Lecture Notes in Computer Science*. Springer, 1998.
16. U. Engberg, P. Gronning and L. Lamport. Mechanical verification of concurrent systems with TLA. In *Fourth International Conference on Computer-Aided Verification (CAV '92)*, volume 663 of Lecture Notes in Computer Science, pages 44–55. Springer, 1992.
17. L. Fejoz, D. Méry and S. Merz. DIXIT: Visualizing predicate abstractions. In R. Bharadwaj and S. Mukhopadhyay, editors, *Automatic Verification of Infinite-State Systems (AVIS 2005)*, Edinburgh, UK, Apr. 2005, pages 39–48. To appear in ENTCS.
18. D. Gries and F. B. Schneider. Avoiding the undefined by underspecification. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, volume 1000 of Lecture Notes in Computer Science, pages 366–373. Springer., 1995.
19. G. E. Hughes and M. J. Cresswell. *A New Introduction to Modal Logic*. Routledge, 1996.
20. S. Kalvala. A formulation of TLA in Isabelle. Available at `ftp://ftp.dcs.warwick.ac.uk/people/Sara.Kalvala/tla.dvi`, Mar. 1995.
21. M. Kaminski. Invariance under stuttering in a temporal logic of actions. *Theoretical Computer Science*, 2006. To appear.
22. H. W. Kamp. *Tense Logic and the Theory of Linear Order*. PhD thesis, University of California at Los Angeles, 1968.
23. L. Lamport. The TLA home page. `http://www.research.microsoft.com/users/lamport/tla/tla.html`.
24. L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, Mar. 1977.
25. L. Lamport. What good is temporal logic? In R. E. A. Mason, editor, *Information Processing 83: Proceedings of the IFIP 9th World Congress*, Paris, Sept. 1983, pages 657–668, 1983. North-Holland, 1983.
26. L. Lamport. How to write a long formula. *Formal Aspects of Computing*, 6(5):580–584, 1994.
27. L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.
28. L. Lamport. How to write a proof. *American Mathematical Monthly*, 102(7):600–608, 1995.
29. L. Lamport. *Specifying Systems*. Addison-Wesley., 2002.
30. L. Lamport. Real-time model checking is really simple. In D. Borrione and W. J. Paul, editors, *Correct Hardware Design and Verification Methods (CHARME 2005)*, volume 3725 of *Lecture Notes in Computer Science*, pages 162–175. Springer, 2005.
31. L. Lamport. Checking a multithreaded algorithm with =+CAL. In S. Dolev, editor, *20th International Symposium on Distributed Computing (DISC 2006)*, Stockholm, 2006. To appear.
32. A. C. Leisenring. *Mathematical Logic and Hilbert's ε-Symbol*, University Mathematical Series. Macdonald, 1969.

33. Z. Manna and A. Pnueli. Verification of concurrent programs: The temporal framework. In R. Boyer and J. Moore, editors, *The Correctness Problem in Computer Science*, pages 215–273. Academic Press, London, 1982.
34. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer, New York, 1992.
35. S. Merz. Isabelle/TLA. Available at `http://isabelle.in.tum.de/library/HOL/TLA`, 1997. Revised 1999.
36. S. Merz. A more complete TLA. In J. Wing, J. Woodcock, and J. Davies, editors, *FM'99: World Congress on Formal Methods*, volume 1709 of Lecture Notes in Computer Science, pages 1226–1244. Springer, 1999.
37. A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on the Foundations of Computer Science*, pages 46–57. IEEE, 1977.
38. A. N. Prior. *Past, Present and Future*. Clarendon Press, 1967.
39. 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.
40. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 1992.
41. P. Suppes. *Axiomatic Set Theory*. Dover, 1972.
42. M. 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. Springer, 2001. See `http://www.cs.rice.edu/~vardi/papers/` for more recent versions of this paper.
43. Y. Yu, P. Manolios, and L. 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. Springer, 1999.

TLA$^+$ Indexes

Symbol Index

Concept Index