Leslie Lamport

Chapter on TLA$^+$

from

Software Specification Methods

An Overview Using a Case Study

Henri Habrias and Marc Frappier, editors

Hermes, April 2006

Contents

Chapter 7

TLA$^+$

Leslie Lamport

7.1 Overview of TLA$^+$

TLA$^+$ is a formal specification language based on set theory, first-order logic, and the Temporal Logic of Actions (TLA) [6,4]. In spirit, TLA$^+$ is close to Z. In fact, some aspects of TLA$^+$ were inspired by Z. I will therefore assume that the reader has read the chapter on Z, and I will explain the TLA$^+$ specification largely in terms of how it differs from the Z specification.

For reasons explained below, Case 1 is problematic. I will therefore first present a complete specification for Case 2 and only afterwards discuss Case 1.

A complete description of TLA$^+$ and its tools can be found in [7]. Here I begin with a brief description of TLA and then describe the major differences between TLA$^+$ and Z.

7.1.1 TLA

A TLA specification is a temporal formula, often named *Spec*. The meaning of a temporal formula is a predicate on behaviors. A behavior represents a conceivable execution of a system. The behaviors satisfying *Spec* are the ones that represent correct behaviors of the system. More precisely, a behavior represents a conceivable history of a universe that may contain the system. A behavior satisfying specification *Spec* represents a history of the universe in which the system behaves correctly. To make this precise, we need some terminology.

A *state* is an assignment of values to variables. A *step* is a pair of states. A *behavior* is an infinite sequence of states; the *steps of* a behavior are its successive pairs of states. A *state predicate* is a formula whose meaning is a predicate (Boolean-valued function) on states. An *action* is a formula whose meaning is a predicate on steps. We often conflate a formula and its meaning. For example, if A is an action, then an *A-step* is defined to be a step that satisfies A. (Formally, the step satisfies the meaning of A, not the formula A.)

In TLA, actions are written as formulas containing primed and unprimed variables. Unprimed variables refer to the variables' values in the first state of the step; primed variables refer to their values in the second state. State predicates are actions with no primed variables.

Like most industrial specifications I have seen, the invoice system has the simplest possible nontrivial TLA specification—namely, it is a temporal formula *Spec* defined by

$$Spec \triangleq Init \wedge \Box[Next]_{\langle v_1,\ldots,v_n\rangle}$$

where *Init* is a state predicate, *Next* an action, and the v_i are the specification's variables. Formula *Spec* is true of a behavior σ iff *Init* is true of the first state of σ and every step (successive pair of states) of σ is either a *Next* step (one that satisfies *Next*) or a "stuttering step" that leaves all the variables v_i unchanged. Nothing happens in a stuttering step, so it is impossible to observe that such a step has occurred. Hence, a specification should not be able to forbid stuttering steps. Allowing them permits implementation/refinement to be simple implication [5], and it permits composition to be conjunction [1]. However, since the specification exercise includes neither refinement nor composition, stuttering steps are irrelevant and can be ignored—except when a behavior ends in an infinite sequence of such steps. A behavior that ends this way represents an execution that terminates. Formula *Spec* allows terminating executions. Forbidding termination requires conjoining a liveness property [2] to the definition of *Spec*. Since there is no liveness requirement for the invoice system, I will ignore liveness.

A TLA specification consists of the definition of the formula *Spec*—that is, the one-line definition given above preceded by the definitions of *Init* and *Next*. These are ordinary mathematical formulas, involving no temporal logic. The $\Box$ in the line above is the only temporal-logic operator in the entire specification. (If we were specifying liveness properties, temporal operators would appear in the definitions of those properties as well.)

7.1.2 TLA$^+$ versus Z

The invoice system example reveals the following differences between the usual way of writing specifications in TLA$^+$ and Z. (There is another style of Z specification, not used in this book, in which sequences of states are described explicitly with ordinary mathematics.)

- A TLA$^+$ specification is a single temporal-logic formula. In Z, there is no single formula or object that mathematically constitutes *the* specification.
- One can assert in TLA$^+$ that a specification satisfies a property; Z has no mechanism for making such an assertion.
- Unlike Z, TLA$^+$ is untyped. Type correctness of a TLA$^+$ specification *Spec* is an invariance property asserting that, in every state reached during every possible execution satisfying *Spec*, each state variable is an element of an appropriate set (its type). One finds type errors by checking that invariance property. In principle, being untyped makes TLA$^+$ significantly more expressive than Z. In practice, the inexpressiveness of Z's type system is at

worst a minor nuisance for writing the specifications that typically arise in industry. There are advantages to a typed language, but I have found them not to be worth the extra complexity that types introduce. (Type checking is discussed in Section 7.4.) However, eliminating types eliminates type declarations that can contain information helpful to the reader; such information needs to be included in comments.

- In Z, schemas are distinct from formulas and have their own logic. In TLA$^+$, there are only formulas. What would be a schema in a Z specification usually becomes the definition of a formula in the corresponding TLA$^+$ specification.
- While both TLA$^+$ and Z use sets and functions, they have different built-in operators for describing them. For example, TLA$^+$ has constructs for manipulating records that Z lacks; Z has a panoply of operators for describing sets of relations that TLA$^+$ lacks. While TLA$^+$ can easily define Z's mathematical operators, the Z syntax for them can be more convenient. Syntactic differences lead to stylistic differences in the specifications. A TLA$^+$ specification might use records where a Z specification uses tuples or a schema, and it might use total functions where a Z specification uses partial functions.
- A TLA$^+$ specification can distinguish between the system's interface, which must be implemented, and its internal state, which serves only to specify the behavior of the interface. This distinction can be made only informally in Z.

The following additional differences between ordinary TLA$^+$ and Z specifications are not revealed by this simple example.

- TLA$^+$ can be used to specify both safety and liveness properties [2]. Z lacks anything corresponding to the TLA$^+$ operators for expressing liveness.
- TLA provides a simple mathematical definition of what it means for one specification to implement another. Implementation is implication. A specification $S1$ implements a specification $S2$ iff $S1 \Rightarrow S2$ is a valid formula. (There is no formal difference between a property and a specification; satisfying a property and implementing a specification are synonymous.)

For an engineer, the most significant difference between Z and TLA$^+$ is probably the set of tools they provide for checking a specification. The tools currently available for checking TLA$^+$ specifications are the SANY syntactic analyzer and the TLC model checker, which is described in Section 7.4 below.

7.2 A Specification of Case 2

There is no such thing as *the* specification of a system. A specification is an abstraction that describes some aspects of the system and ignores others. It is like a map. One wants a different map of Texas for driving from Amarillo to Houston than for finding new deposits of helium. So the first question one should ask is:

Question 1: What is the purpose of the specification?

Answer: This question does not seem to have an answer. The invoice example is artificial because it does not indicate what the specification is to be used for. In my experience, engineers are most interested in specifications as a way of finding errors early in the design process. For that purpose, one writes a specification of a high-level design and checks that it satisfies certain properties. The description of the invoice system gives no nontrivial properties to be checked. Since I am just copying the Z specification, I do not have to answer this question. I will accept whatever answer is implicit in the Z specification.

The first question engineers who sit down to write a specification usually ask is:

Question 2: How do we begin?

Answer: Knowing how to begin is probably the hardest part of writing a specification. My best answer to this question is that one begins by informally writing a single correct behavior of the system. It can be written either as a sequence of states or a sequence of events. Doing this determines the grain of atomicity of the specification. For the invoicing system, it answers questions such as, is the placing of a new order represented as two events—the user places the order and the system replies—or as a single event?

Since I am mimicking the Z specification, knowing where to begin is not a problem. The Z specification tells us that placing an order is represented as a single event. Thus, the specification cannot describe a scenario in which one user places an order and, before the system responds, a second user places another order. If there are multiple users, which is not ruled out by the system description, such a scenario cannot be avoided. Whether abstracting away this real possibility ignores an irrelevant complication or hides potential problems depends on the purpose of the specification.

The next questions one asks are about the same for a TLA$^+$ specification as for a Z specification. I will therefore jump directly to an explanation of the complete specification, which appears in Figures 7.1 and 7.2. (Since the specification is explained in the text, I have omitted the explanatory comments that should appear in every specification.) If you have read the Z specification, then you already know pretty much what the TLA$^+$ specification says. I will therefore just explain the TLA$^+$ notation and the differences between the two specifications.

TLA$^+$ specifications are organized into modules. This simple specification consists of a single module named *Invoice*. The module begins with an EXTENDS statement that imports the standard module *Naturals*. This module defines the set *Nat* of natural numbers and the usual arithmetic operators.

The CONSTANT statement declares the constant parameters *OrderId* and *Product* that are the same as in the Z specification. The VARIABLES statement declares the specification's variables. (Unlike a constant, a variable can change its value in the course of a behavior.) The variable *stock* is as in the Z specification. I have replaced the two Z variables *orders* and *orderStatus* by a single

```
MODULE Invoice
EXTENDS Naturals
CONSTANTS OrderId, Product
VARIABLES stock, order, inp, out

ProdOrder ≜ {f ∈ [Product → Nat] : ∃ p ∈ Product : f[p] ≠ 0}

Order ≜ [state : {"pending", "invoiced"}, prods : ProdOrder]
        ∪ [state : {"none"}]

TypeOK ≜ ∧ stock ∈ [Product → Nat]
         ∧ order ∈ [OrderId → Order]

Init ≜ ∧ stock = [x ∈ Product ↦ 0]
       ∧ order = [x ∈ OrderId ↦ [state ↦ "none"]]
       ∧ inp = ⟨""⟩
       ∧ out = ⟨""⟩

InvoiceOrderOp(id) ≜
  ∧ inp' = ⟨"Invoice", id⟩
  ∧ IF order[id].state ≠ "pending"
    THEN ∧ out' = ⟨"order_not_pending"⟩
         ∧ UNCHANGED ⟨stock, order⟩
    ELSE IF ∀ p ∈ Product : order[id].prods[p] ≤ stock[p]
         THEN ∧ out' = ⟨"OK"⟩
              ∧ order' = [order EXCEPT ![id].state = "invoiced"]
              ∧ stock' =
                  [p ∈ Product ↦ stock[p] − order[id].prods[p]]
         ELSE ∧ out' = ⟨"not_enough_stock"⟩
              ∧ UNCHANGED ⟨stock, order⟩

NewOrderOp(pOrder) ≜
  ∧ inp' = ⟨"NewOrder", pOrder⟩
  ∧ ∨ ∃ id ∈ OrderId :
        ∧ order[id].state = "none"
        ∧ out' = ⟨"OK", id⟩
        ∧ order' = [order EXCEPT ![id] = [state ↦ "pending",
                                          prods ↦ pOrder]]
        ∧ UNCHANGED stock
    ∨ ∧ ∀ id ∈ OrderId : order[id].state ≠ "none"
      ∧ out' = ⟨"IdError"⟩
      ∧ UNCHANGED ⟨stock, order⟩
```

Fig. 7.1. The complete specification (beginning).

```
CancelOrderOp(id) ≜
  ∧ inp' = ⟨"CancelOrder", id⟩
  ∧ IF order[id].state = "pending"
      THEN ∧ out' = ⟨"OK"⟩
           ∧ order' = [order EXCEPT ![id] = [state ↦ "none"]]
           ∧ UNCHANGED stock
      ELSE ∧ out' = ⟨"order_not_pending"⟩
           ∧ UNCHANGED ⟨stock, order⟩

EnterStock(pOrder) ≜
  ∧ inp' = ⟨"EnterStock", pOrder⟩
  ∧ out' = ⟨"OK"⟩
  ∧ stock' = [p ∈ Product ↦ stock[p] + pOrder[p]]
  ∧ UNCHANGED order

Next ≜
  ∨ ∃ id ∈ OrderId : InvoiceOrderOp(id) ∨ CancelOrderOp(id)
  ∨ ∃ pOrder ∈ ProdOrder : NewOrderOp(pOrder) ∨ EnterStock(pOrder)

Spec ≜ Init ∧ □[Next]⟨stock, order, inp, out⟩

THEOREM Spec ⇒ □TypeOK
```

Fig. 7.2. The complete specification (end).

record-valued variable *order*, where *order.prods* replaces *orders* and *order.state* replaces *orderStatus*. The variables *inp* and *out* represent the system's input and output. The Z specification assumes that each operation is performed as a single atomic action at the behest of some external agent. In the TLA$^+$ specification, that action sets *inp* to the agent's input and *out* to the system's output. There is no variable *newids* because its value is a simple function of the other variables—namely, it equals *OrderId* minus the set of orders that are pending or invoiced. (Avoiding redundant variables is a stylistic choice; I find that it makes a specification clearer.)

Following the purely decorative horizontal line come definitions of two constant sets, *ProdOrder* and *Order*. The set *ProdOrder* represents the set of all nonempty bags of products. It would be easy to define a bag as a partial function, the way Z does. (The standard module *Bags* does just that.) However, here it's more convenient to represent a bag of products as a function b whose domain is the set *Product* of all products, where $b[p]$ is the number of copies of p in bag b, for any p in *Product*. The set *Order* is the set of all possible values of $order[id]$ for an *id* in *OrderId*. If *id* is an unused *OrderId*, then $order[id]$ is a record with just a *state* component whose value is the string "none". Otherwise it is a record whose *state* component is either "pending" or "invoiced" and whose *prods* field is an element of *ProdOrder*. (In TLA$^+$, one typically uses a string like "pending"

instead of introducing an unspecified constant *pending* as in the Z spec.) The definitions of *ProdOrder* and *Order* use the following TLA$^+$ notation:

- $\{v \in S : P(v)\}$ is the subset of S containing all elements v satisfying $P(v)$.
- $[S \to T]$ is the set of all functions with domain S and range a subset of T.
- $[l_1 : S_1, \ldots, l_n : S_n]$ is the set of all records r with fields $l_1, \ldots, l_n$ such that $r.l_i \in S_i$ for each i. (A record is a function whose domain is the finite set of strings consisting of the names of its fields.)

The module next defines the type-correctness predicate *TypeOK* and the initial predicate *Init*. A type-correctness predicate asserts that each variable is an element of some set that is usually called its "type". This predicate is not part of the specification, meaning that it is not used in defining *Spec*. A theorem at the end of the module asserts that *TypeOK* is an invariant of *Spec*. It's helpful to state a type invariant early in the specification, because knowing the types of the variables makes the specification easier to read. I haven't bothered to specify the types of *inp* and *out*, since knowing their types isn't important for understanding the specification. The initial predicate *Init* specifies the initial values of the variables. The initial values of *inp* and *out* don't matter and could be left unspecified. However, the TLC model checker requires that all variables be initialized. Since the specification's actions always set *inp* and *out* to equal tuples, we initialize the variables to 1-tuples for uniformity. Predicate *Init* asserts that each variable equals a single value; an initial predicate often asserts that a variable is an element of some set. The definitions of *Init* and *TypeOK* introduce the following TLA$^+$ notation.

- A list bulleted by $\wedge$ or $\vee$ represents the conjunction or disjunction of the items. Indentation is used to eliminate parentheses in nested lists of conjunctions and/or disjunctions. This makes large formulas easier to read. We can also use $\wedge$ and $\vee$ as the customary infix operators.
- $[x \in S \mapsto e(x)]$ is the function f with domain S such that $f[x] = e(x)$ for all x in S.
- $\langle e_1, \ldots, e_n \rangle$ is an n-tuple, for any natural number n.
- $[l_1 \mapsto e_1, \ldots, l_n \mapsto e_n]$ is the record r with fields $l_1, \ldots, l_n$ such that $r.l_i = e_i$ for each i.

The next section of the module defines the next-state action *Next*, which specifies the allowed steps of the system. Jumping to the actual definition of *Next*, we see that it is a disjunction of a collection of actions. (I consider existential quantification to be a form of disjunction.) It defines a *Next* step to be an *InvoiceOrderOp*(*id*) or *CancelOrderOp*(*id*) step for some *id* in *OrderId*, or else a *NewOrderOp*(*pOrder*) or *EnterStock*(*pOrder*) step for some *pOrder* in *ProdOrder*. There is no formal significance to the names of these actions, or to

this particular way of writing *Next* as a disjunction. We could write the definition of *Next* in any number of equivalent ways—for example, by eliminating the definitions of *InvoiceOrderop*, etc. and defining *Next* as one large formula.

Of course, I defined *Next* in this way to mimic the Z specification. There is the following correspondence between the TLA$^+$ actions and the Z operations:

- Action $InvoiceOrderOp(id)$ corresponds to an *InvoiceOrderOp* operation with its input $id?$ equal to id.
- Action $NewOrderOp(pOrder)$ corresponds to any Z operation *NewOrderOp* whose input $order?$ equals $pOrder$. (There may be many such operations—one for each possible output value $id!$.)
- Action $CancelOrderOp(id)$ corresponds to a *CancelOrderOp* operation with input $id?$ equal to id.
- Action $EnterStock(pOrder)$ corresponds to any *EnterStock* operation with input $newstock?$ equal to $pOrder$.

Knowing the meaning of the Z operations, you should be able to understand the definitions of these actions. The only new TLA$^+$ notation used in these definitions is the EXCEPT construct. The instances of this construct that are used here are explained by:

- $[f \text{ EXCEPT } ![i] = e]$ is the function g that is the same as f, except with $g[i] = e$.
- $[r \text{ EXCEPT } !.l = e]$ is the record s that is the same as r, except with $s.l = e$.
- $[f \text{ EXCEPT } ![i].l = e]$ equals $[f \text{ EXCEPT } ![i] = [f[i] \text{ EXCEPT } !.l = e]]$.

Observe that the actions all set the variables *inp* and *out* to tuples—either pairs or one-tuples. It is generally best to have the values of a variable all of a uniform "type". This is why, for example, the $EnterStock(pOrder)$ action sets *out* to the one-tuple $\langle \text{“OK”} \rangle$ rather than simply to the string “OK”.

The expected definition of *Spec* follows the definition of *Next*. Formula *Spec* is the specification of the invoice system. However, it is not a satisfactory specification for several reasons. The first has to do with stuttering. Consider a behavior in which the action $NewOrderOp(\pi)$ is executed twice in a row, with the same product order π, when there is no unused *OrderId*. Both executions set *inp* to $\langle \text{“NewOrder”}, \pi \rangle$, set *out* to $\langle \text{“IdError”} \rangle$, and leave *stock* and *order* unchanged. The second execution leaves all four variables unchanged, so it is a stuttering step. Since stuttering steps are unobservable, the second execution essentially never happens. Execution of the other actions could also produce a stuttering step in case of an error. The specification should distinguish between nothing happening and a second $NewOrderOp(\pi)$ operation being performed. For it to make this distinction, we must ensure that executing an action always changes the value of some variable. An easy way to do this is by adding another component to the tuple *inp* and/or *out* that is changed with every input or output. For

example, we could let *inp* have a Boolean first component that is complemented on each action. In TLA$^+$, the i^{th} element of a tuple t is $t[i]$, so the first conjunct of $NewOrderOp(pOrder)$ could then be written

$$inp' = \langle \neg inp[1], \text{“NewOrder”}, pOrder \rangle$$

We would also have to modify the definition of *Init*, for example to assert that *inp* equals $\langle \text{TRUE} \rangle$.

Modifying the specification in this way highlights its second problem: the encoding of inputs and outputs is rather arbitrary. It would be more elegant simply to say that a $NewOrderOp(pOrder)$ step is performed by providing as input the operation name “NewOrder” and the product order *pOrder*, without specifying how those inputs are encoded in the value of *inp*. It is easy to do this in TLA$^+$; Section 5.1 of [7] shows how. However, such elegance is of little concern to engineers.

The final problem with *Spec* as a specification of the invoice system is that it specifies the possible sequences of values assumed by all four variables *stock*, *order*, *inp*, *out*. A straightforward interpretation of the invoice system's description implies that only *inp* and *out* are directly visible. The values of variables *stock* and *order* can only be inferred from observing the inputs and outputs. An implementation must implement the input and output described by the variables *inp* and *out*, but it is under no obligation to implement *stock* and *order*. A philosophically correct specification would hide those two variables. Such a specification is written informally as

$$\exists\, stock, order \,:\, Spec$$

where $\exists$ is temporal existential quantification [6]. TLA$^+$ does not allow one to write this formula because its meaning is not at all clear. The problem with it has nothing to do with temporal logic; the meaning would be equally unclear with ordinary quantification and a non-temporal formula *Spec*. Logicians seem to be unaware of the problem because they never try to define formally what a definition means. The correct way to hide internal variables in TLA$^+$ is explained in Section 4.3 of [7]. However, engineers are not concerned with philosophical correctness and don't bother hiding internal variables.

A comparison of the TLA$^+$ and Z specifications may lead one to ask:

Question 3: The Z specification decomposes the operations into conjunctions and disjunctions of simpler operations. Why doesn't the TLA$^+$ specification decompose the action definitions in a similar way.

Answer: It would have been easy to define the actions in terms of simpler ones. However, there is no point doing so for such simple actions. For most systems, the next-state action is naturally written as the disjunction of actions that each describe some single class of system events. Sometimes those actions may be grouped in a natural way, leading to a hierarchical definition of the next-state action as a disjunction of disjunctions. However, I have found that

there is seldom anything to be gained by writing an action as the conjunction of separately-defined actions.

Question 4: But isn't modularity helpful—for example, in re-using specifications?

Answer: Almost everything you have learned about modularity and re-use is irrelevant for specification. Almost every TLA$^+$ specification ever written is no longer than about two thousand lines (excluding comments). I have found that engineers usually want to specify their systems in as much detail as possible. However, they can't understand specifications that are longer than about two thousand lines. If a specification starts becoming too long, an engineer starts over again and writes a less detailed, higher-level specification.

The TLA$^+$ module system permits the same kind of modularity that is provided by Z's schemas (and more). However, I have yet to see an engineer break up a specification into modules. Breaking definitions into simpler definitions within a single module provides all the modularity one needs for a 2000-line specification. This is in large part because TLA$^+$ has a LET... IN construct that permits definitions that are local to a formula, so one can hierarchically structure a single definition.

Re-use of specifications is also a non-issue in practice. The hard part of specifying a system is understanding it and finding a suitable level of abstraction. The effort of writing 2000 lines of formulas is minor. There is little point trying to make a specification reusable. If in the future we want to write a specification similar to that of the invoice system, we can just modify the invoice system's specification.

The specification ends with a theorem asserting type correctness. The temporal formula $\Box \mathit{TypeOK}$ is true of a behavior σ iff every state of σ satisfies the state predicate TypeOK. The formula $\mathit{Spec} \Rightarrow \Box \mathit{TypeOK}$ asserts of a behavior σ that, if σ satisfies Spec, then it satisfies $\Box \mathit{TypeOK}$. The theorem asserts that this formula is true for all behaviors. (Remember that a behavior is any sequence of states, not just one that satisfies some specification.)

7.3 The Problematic Case 1

A TLA$^+$ specification describes a complete system and its environment. Thus far, I have been describing *closed-system* specifications that are satisfied by all behaviors in which both the system and its environment perform correctly. The distinction between the system and the environment is informal, and we must read the comments to discover which actions are to be implemented as part of the system and which are to be performed by the environment.

We can also write *open-system* specifications, also called *rely/guarantee* specifications, that are satisfied by all behaviors in which the system performs correctly as long as the environment does. The system and environment are then

formally distinguished and the specification describes exactly what the system's implementer must implement. As explained in Section 10.7 of [7], transforming a closed-system specification to an open-system one, or vice-versa, is usually trivial. It generally requires changing about three lines of the specification. Closed-system specifications are conceptually a bit simpler; they are the only ones that engineers write—largely because they're the only ones that TLC can check directly.

Our specification of the invoice system is unusual because a single step represents both an operation performed by the environment (providing input) and one performed by the system (changing the system state and producing output). By choosing such a representation, we committed ourselves to a closed-system specification that cannot be transformed into an open-system one.

The problem with Case 1 is that it does not ask for a description of a complete invoice system. Instead, it asks for a description of one operation of such a system. We could transform this into a system-specification exercise by defining a system that performs only the invoicing operation and is used in an environment that performs the other operations of the invoicing system. To write such a specification, we would have to decide how abstractly to represent this "environment". If we represent those other operations at the same level of abstraction as we did in Case 2, then a closed-system specification for Case 1 becomes identical to our closed-system specification for Case 2. All we change are the comments, indicating that all actions are performed by the "environment" except for *InvoiceOrderOp* actions, which are performed jointly by the "system" and the "environment". Had we written an open-system specification for Case 2, we could have obtained the specification for Case 1 by modifying it slightly to attribute all but the invoice system's invoicing actions to the environment.

While we could do all this, it is quite unnatural to consider the invoicing operation by itself to form a separate system. A more sensible interpretation of Case 1 is that it asks for just the one part of a larger specification that describes the invoicing operation. The definition of *InvoiceOrderOp* in the *Invoice* module provides such a description.

7.4 Validation of the Specification

We check a specification by checking that it satisfies certain desired properties. Most often checked in practice are invariance properties. With TLA$^+$, a property can be an ordinary specification—that is, a formula of the same form $Init \wedge \Box[Next]_{\langle \ldots \rangle}$ as *Spec*. If the specification includes liveness requirements, we can also check that it satisfies liveness properties.

The description of the invoicing system provides no properties that it should satisfy. Indeed, the system is so simple that it would be hard to find properties to check that would increase our confidence that the specification says what we want it to. The only thing we can check is the invariance property $\Box TypeOK$. Checking that *Spec* satisfies $\Box TypeOK$ essentially tells us that the specification

is type correct. Invariance of a type-correctness predicate is a stronger property than is provided by automatic type checking in a typed language. For example, if s is a variable that has some sequence type, then the operation that assigns the tail of s to s will satisfy an automatic type checker. However, it will violate a type-correctness invariant if that operation can ever be executed when s equals the empty sequence.

I developed the logic TLA to provide a simple and elegant way of formalizing the correctness proofs of concurrent algorithms—the kind of proofs I had been writing for about 15 years. TLA$^+$ is well-suited to writing formal proofs; an example of a formal hand proof written in TLA$^+$ appears in [3]. However, very few engineers have the time or the training to write rigorous mathematical proofs. A couple of TLA$^+$ proofs have been checked mechanically by hand-translating them into the logic of a mechanical theorem prover, but there is not yet a mechanical proof checker for TLA$^+$.

Model checking is the most attractive form of verification for engineers, usually yielding by far the greatest confidence for the amount of effort expended. The TLC model checker, written by Yuan Yu, is described in Chapter 14 of [7]. It can check a finite model of a specification obtained by instantiating the constant parameters and, if necessary, specifying constraints to make the set of reachable states finite. One obtains a finite model of the invoice specification as follows, for particular values of α and β:

- Substituting specific finite sets for the parameters *OrderId* and *Product*.
- Replacing the set *ProdOrder* with the set of product orders containing at most α copies of any one item.
- Constraining TLC to examine only states in which $stock[p] \leq \beta$ for all products p.

With *OrderId* and *Product* each containing two elements, $\alpha = 2$, and $\beta = 3$, TLC finds about 70000 reachable states and checks the type invariant in less than 30 seconds on my laptop. In the course of checking invariance, TLC also checks for the absence of deadlock—meaning that the system never reaches a state in which no action is enabled.

For most specifications, the kind of error that would be found by automatic type checking when using a typed language is found by TLC in a few seconds with a very small model.

7.5 Satisfying the Specification

After writing a specification, the next step is to implement it. We would naturally like to check that the implementation satisfies the specification. In TLA, implementation, satisfaction, and refinement all mean the same thing: logical implication. We can check that one TLA$^+$ specification implies another—either by writing a proof or by using TLC.

In principle, it is straightforward to check that a TLA$^+$ model of an implementation satisfies a TLA$^+$ specification. This works quite well in practice for concurrent algorithms. One writes a simple TLA$^+$ specification of what the algorithm is supposed to do, writes a TLA$^+$ description of the algorithm, and shows that the algorithm implements its specification—see [3] for an example. The same idea can work for high-level system designs. One can write a TLA$^+$ specification of what the system is supposed to do, write a TLA$^+$ description of the design, and check that the design satisfies its specification. However, I have found that this is seldom done for real systems—with TLA$^+$ or any other language. Engineers usually specify only the high-level design and check that it satisfies a few properties rather than a complete specification. I hope this changes as engineers gain more experience with specifications.

Ultimately, most systems must be implemented in a programming language or hardware-design language. One would also like to check that this implementation satisfies the TLA$^+$ specification. In principle, this can be done by using a TLA$^+$ representation of the implementation, obtained from a TLA$^+$-based formal semantics of the implementation language. In practice, this kind of verification is economically feasible only for small, extremely critical applications. TLA$^+$ now has no tools to support such low-level verification, so it is probably not an appropriate language for the task.

For most applications, the only feasible way of checking that an implementation satisfies a higher-level specification is by testing. This can be done by translating executions of the implementation into the corresponding higher-level behaviors and using TLC to check that those behaviors satisfy the TLA$^+$ specification. The translation is performed by instrumenting the implementation in some way. There is no tool to help with the instrumentation, but engineers seem to find it easy to do on an *ad hoc* basis.

To my surprise, I have found that engineers are not very interested in this kind of checking. They seem confident in their ability to determine if an execution is correct without checking it against the specification. Instead, they want to use the specification to help generate tests. A promising approach that has been investigated is to use the specification to improve test coverage. Test executions are translated to behaviors that are used as input to TLC. However, instead of just checking that they satisfy the specification, TLC keeps track of which reachable high-level states the behaviors have not reached. This information is used to generate additional tests that drive the implementation into those unreached states [8].

7.6 The Natural Language Description

No complicated formal specification can be understood without a natural language explanation. That explanation normally appears in comments within the module. The TLATEX program described in Chapter 13 of [7] can be used to typeset the commented ASCII specification in a more readable format. Figures 7.1

and 7.2 were generated automatically by TLATEX from the uncommented ASCII specification—the exact specification on which TLC was run. (However, that specification resided in an Emacs buffer while I wrote this chapter, so anything might have happened to it since TLC checked it.)

7.7 Conclusion

The simple invoice specification gives little insight into what TLA$^+$ is like in practice. For example, when first viewing TLA$^+$, a common complaint is the need for UNCHANGED conjuncts in the actions. The invoice specification might make that complaint seem justified, since 13% of its lines are UNCHANGED conjuncts. In a typical specification, the figure is more like 4%. One must use TLA$^+$ to realize that rather than being overhead, those UNCHANGED conjuncts provide useful redundancy. They allow TLC to discover if we have inadvertently neglected to specify the new value of a variable.

Significantly absent from the invoice example is concurrency. Concurrency is not mentioned in the example's description, and it is excluded from the Z specification. TLA$^+$ was designed for specifying and reasoning about concurrent systems. From a practical point of view, the differences between the TLA$^+$ and Z specifications of the invoice system are largely stylistic. The differences would be more significant for a concurrent system—especially if liveness were important. Liveness properties of sequential systems tend to be simple, asserting that an input action must be followed by the corresponding output action. An informal treatment of liveness is usually satisfactory. Liveness properties of concurrent systems can be subtle and can often be made clear only through formal specification.

In principle, a language's inability to express liveness could be a handicap. In practice, it seldom is. Experience has shown that most errors are violations of safety properties. Moreover, the computational complexity of model checking is larger for liveness properties than for safety properties. This means that model checking liveness properties is usually feasible only for small models—ones that may be too small to find subtle errors. The importance of liveness is more philosophical than practical.

References

1. Martín Abadi and Leslie Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, May 1995.
2. Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.
3. Eli Gafni and Leslie Lamport. Disk paxos. *Distributed Computing*, 16(1):1–20, 2003.
4. Leslie Lamport. TLA—temporal logic of actions. A web page, a link to which can be found at URL `http://lamport.org`. The page can also be found by searching the Web for the 21-letter string formed by concatenating `uid` and `lamporttlahomepage`.

5. Leslie Lamport. What good is temporal logic? In R. E. A. Mason, editor, *Information Processing 83: Proceedings of the IFIP 9th World Congress*, pages 657–668, Paris, September 1983. IFIP, North-Holland.
6. Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.
7. Leslie Lamport. *Specifying Systems*. Addison-Wesley, Boston, 2003. Also available on the Web via a link at `http://lamport.org`.
8. Serdar Tasiran, Yuan Yu, Brannon Batson, and Scott Kreider. Using formal specifications to monitor and guide simulation: Verifying the cache coherence engine of the Alpha 21364 microprocessor. In *In Proceedings of the 3rd IEEE Workshop on Microprocessor Test and Verification, Common Challenges and Solutions*. IEEE Computer Society, 2002.