Using The Temporal Logic of Actions: A Tutorial on TLA Verification

Peter B. Ladkin
Universität Bielefeld, Germany
ladkin@rvs.uni-bielefeld.de

June 1997

1 Contents

The purpose of these notes is to tell a story: to introduce the main ideas in verification using TLA, and some background to those ideas. Concepts are introduced dynamically (or ignored where possible!) by working through some examples. What doesn't arise in the examples is generally not mentioned. For those wishing more in the way of reference manuals to and explanations of TLA, TLA+, and their use, they abound in [Lama].

On to the story. In this tutorial, I briefly :-)

- talk about change
- introduce a linguistic convention for describing some sorts of change
- note that computer systems essentially involve change
- explain why verification is needed
- explain what a TLA specification is: a state machine
- explain what a TLA verification is: a proof that one state machine simulates another

- ..that proceeds by showing a logical implication
- define the logic of TLA; but mostly...
- ....work through an example

The example is taken directly from [Lad96a], in which the entire verification is laid out in hierarchical form. In these notes, the verification is broken up into levels, to illustrate how a proof may be intellectually decomposed. Once one understands the concept of proof level and how it looks, the textual form in [Lad96a] should be easier to survey than the form presented here.

I shall only introduce those concepts which seem to be needed for doing the proof. That means there are fundamental concepts of TLA which are omitted here – I make no pretence to completeness. I also wave my hands about TLA+ syntax, and about some semantic issues, in places where I think 'getting a feel' is more helpful than precision (um, did I really say that? Precision is, of course, of utmost importance in specification and verification!). *The* definitive source for further information is [Lama], which accumulates and points to all work on TLA.

Some may carp at my use of an outmoded syntax for TLA+ (namely, **include** -ing and **import** -ing of modules) and some extra keywords. I find that the explanation of **import** and **include** that I give here is straightforward to understand, even if these concepts themselves are not optimal; and I find the extra keywords helpful. All is explained in Section 5.1. Besides, this has to do with TLA+, the specification language that is an extension of TLA, and not TLA itself. The syntax of TLA itself has not changed. The 'latest' TLA+ syntax is available on-line at [Lama] and I don't consider it hugely different from what I use here.

2 Change

The ideas of *stasis* and *change* have been at the center of philosophical inquiry for millenia. The discussion of *being and becoming* [Res95] has Heraclitus claiming that *being* is illusory; there is only *becoming*, and the Eleatics believing the opposite, along with McTaggart [McT27] and Bradley [Bra97]. Let's summarise 3,000 years of discussion by saying there's been arbitrage. McTaggart's (in)famous argument for the unreality of time (in

[PM93]) presages the modern syntax (the 'A-series') and semantics (the 'B-series') of temporal logic.

Take a snapshot. The relations between objects that you observe in the snapshot are said to be '*state predicates*'. Take another. You may observe that some state predicates are no longer true: some relations in the new snapshot contradict the relations in the first. We can point out the differences. Let us call them 'changes'. To describe them, we may conveniently refer to the first and second snapshots. This is the B-series way of conceiving of the passage of time.

We may know as we're taking the first snapshot some features that will remain the same in our second, and some that will be different. We say: *in the future*, this will remain, this will change. When we take the second, we notice some more features: *in the past*, these were different. This is the A-series way, 'looking' from a continously-changing present into the future and the past.

Changes are noted by comparing two snapshots, identifying objects across the two snapshots as being the 'same object' or a different one. Sometimes, we may wish to remark on a change by trying to find a state predicate on which we believe that change is consequent: thus change of position is assimilated to a state property *velocity*. Velocity is in turn correlated with *force*, so that a constant force results in constant velocity (Aristotle). It takes a Galileo to point out that this ain't true, and a Newton to iterate the process and relate constant force instead to constant change-of-velocity.

All we can see on our snapshot is position, but apparently there are more things in heaven and earth, Horatio, than we can see on our snapshots. Velocity and acceleration are state properties nowadays.

2.1 Some Notation for Change

Suppose we count the oranges in the first snapshot. There are three. Let's introduce a name for the result of the counting operation: $NumberOfOranges$.

$$NumberOfOranges = 3$$

And then in the second snapshot, we perform the operation and find there are four.

$$NumberOfOranges = 4$$

That is the only difference we can see. But the two sentences are contradictory: $NumberOfOranges$ cannot have both the value 3 and the value 4 at the same time, according to our normal logic and logical notation. We have to distinguish the reference of the name *NumberOfOranges* in the first snapshot from that in the second. But it's not *completely* different: we're counting in some sense *the same thing*. How can we make sense of this?

We also want to compare them; to say: *the NumberOfOranges has changed*, or *the NumberOfOranges has increased*, How may we say: *everything else has stayed the same*? (But of course, it hasn't. We just can't see the positions of the molecules to note the differences. We really mean: everything else *we can notice* has stayed the same.) So how can we note the difference between our two quantities and yet note that they are values for somehow the same feature?

Let me perform the comparison:

$$NumberOfOranges = 3 \wedge NumberOfOranges' = 4$$

uses the meta-operator *prime* to talk about *the-next-time-this-changes*. We can also say that the *NumberOfApples* has not changed:

$$NumberOfApples' = NumberOfApples$$

Pace McTaggart (who thought such a way of speaking essential to the idea of time, but that it wasn't possible ultimately to make sense of this way of speaking), this is just syntax and it's a conservative extension of FOL – I have, if you like, simply introduced a way of making new variables from old.

According to this view, then, remarking 'change' involves remarking a difference to the 'same' object or 'same' relation on two snapshots. We must be able to identify objects and relations across snapshots, but if we can do that, it suffices. Accordingly, let's restrict ourselves to writing *prime* only once: $NumberOfApples''$ is not well-formed.

2.2 Change Tokens and Types

Now I can classify occurences of change (which we can call change *tokens*) into *types*: A pair of snapshots shows a *change in the NumberOfOranges* if and only if

$$NumberOfOranges' \neq NumberOfOranges$$

A pair of snapshots shows a *doubling in the NumberOfOranges* if and only if

$$NumberOfOranges' = 2 \times NumberOfOranges$$

And any *doubling in the NumberOfOranges* is also a *change in the NumberOfOranges*, because any pair of snapshots showing a doubling in the *NumberOfOranges* also shows a change: substitute

$$x \leftarrow NumberOfOranges', y \leftarrow NumberOfOranges$$

in

$$\vdash_{PA} (x = 2 \times y \;\Rightarrow x \neq y)$$

The 'classical' thing to do would be thus to identify the type *change in the NumberOfOranges* with the set of pairs of snapshots (*tokens*) that exemplify it. Then the implication allows change-type-inclusion to be 'reduced' to set-inclusion.

3 A Natural Deduction System

So what's the logic of all this? Let's formulate it in a *natural deduction* system. This is convenient, because the hierarchical proof method we shall use [Lam93, Lam95] is a form of (inverted) natural deduction. I'm going to assume the usual notions of logical syntax. In natural deduction, there are no axioms, but there are a *lot* of rules of inference: traditionally, one *intro*duction and one *elim*ination rule for each logical constant (sentential connectives, quantifiers, modal operators).

3.1 Assumptions in Natural Deduction

How *assumptions* and *assumption sets* function is essential for defining any natural-deduction system, because in a natural-deduction system, there need be no axioms. However, these notions will not play much of an explicit role in the example. Defining these notions in the context of the hierarchical proof method (which is an inverted form of standard natural deduction) is much easier than in more traditional natural deduction systems. I include an explanation here for completeness, but in fact it may be skipped without much pain by those wanting to go straight to the TLA verification example.

Instead of using axioms in natural deduction, one makes *assumptions*, which may be arbitrary formulas. Assumptions are accumulated in *assumption sets* as one proceeds with a proof, and 'carried along with' the proof. That is, every occurrence of every formula in a natural-deduction proof has an assumption set associated with it, and the definition of the assumption set of an occurrence of a formula is given recursively by the proof rule used to derive the formula. The assumption set starts out as $\emptyset$. Then:

- Make a new assumption: increment the assumption set with it.

The inference rules indicate how to modify the assumption set on which the conclusion is based, given the assumption set of each premise, by using set operations and by *discharging* certain assumptions (discharge being represented by square brackets).

- 'Discharge' an assumption: decrement the assumption set by it.

A *theorem* is the conclusion of a chain of inferences, such that all the assumptions made in the course of the proof have been discharged by the time this conclusion is reached: its assumption set is $\emptyset$. In rules, we write assumptions A that are discharged by application of the rule in square parentheses, thus: $[A]$.

For example, the rule ($\Rightarrow$-intro):

$$\frac{\begin{array}{c}[A]\\ B\end{array}}{A \Rightarrow B}$$

says that if the set of assumptions on which the premise B is based is Λ, then the set of assumptions on which the conclusion is based is

$$\Delta = \Lambda \backslash \{A\}$$

Another example: ($\vee$-elim)

$$\frac{A \vee B \qquad \begin{array}{c}[A]\\ C\end{array} \qquad \begin{array}{c}[B]\\ C\end{array}}{C}$$

says that if the set of assumptions on which the first premise is based is Λ, set of assumptions of the second premise is Γ and of the third, Θ, then the set of assumptions on which the conclusion is based is

$$\Delta = \Lambda \cup (\Gamma \backslash \{A\}) \cup (\Theta \backslash \{B\})$$

See [Pra65] for a precise definition and explanation of assumptions and discharging.

In the hierarchical proof method, assumption sets are easier to define. One makes assumptions at particular proof steps. The *scope* of an assumption is the set of all substeps that appear under it. The set of all assumptions of a proof step is the set of assumptions within whose scope that step lies.

3.2 Classical Rules

Let's start with a Prawitz-type natural deduction system [Pra65] with the classical rules, presented in the traditional manner: proofs 'move' downwards towards their desired conclusion.

3.2.1 Propositional Rules

(&-intro) $\dfrac{A \quad B}{A \wedge B}$ (&-elim) $\dfrac{A \wedge B}{A} \quad \dfrac{A \wedge B}{B}$

($\vee$-intro) $\dfrac{A}{A \vee B} \quad \dfrac{A}{A \vee B}$

($\vee$-elim) $\dfrac{A \vee B \qquad \begin{matrix}[A] \\ C\end{matrix} \qquad \begin{matrix}[B] \\ C\end{matrix}}{C}$

Note that these rules strictly speaking build proof *trees*, not simple sequences of formulas, since proof branches converge in certain rules. Branches converge as we move 'downwards' in the proof towards the desired conclusion. The root of the tree is the desired theorem, and the leaves are the initial assumptions.

Define $(\neg A)$ to be the formula $(A \Rightarrow \perp)$

($\Rightarrow$-intro) $\dfrac{\begin{matrix}[A] \\ B\end{matrix}}{A \Rightarrow B}$ ($\Rightarrow$-elim) $\dfrac{A \quad A \Rightarrow B}{B}$

($\perp_{\mathsf{I}}$) $\dfrac{\perp}{A}$ ($\perp_{\mathsf{C}}$) $\dfrac{\begin{matrix}[\neg A] \\ \perp\end{matrix}}{A}$

A restriction on both $\perp$ rules is that A is different from $\perp$. A restriction on $\perp_{\mathsf{C}}$ is that A is not of the form $B \Rightarrow \perp$ (but this is only for convenience:

Exercise:- Show that if A has the form $\neg B$ in $\perp_{\mathsf{C}}$ that there's another way of deriving the conclusion). Note that $\perp_{\mathsf{C}}$ subsumes $\perp_{\mathsf{I}}$.

The use of $\perp$ is logically important: if $\perp_{\mathsf{C}}$ is used, the theorems are those of classical propositional logic; if $\perp_{\mathsf{I}}$ is used, the theorems are those of intuitionistic propositional logic. The *formal* difference between the two logics is thus reduced just to that of how negation is treated.

3.2.2 Quantifiers

Now let's include the quantifiers:

$$(\forall-\text{intro}) \quad \frac{A}{\forall x\ :\ A[a/x]} \qquad (\forall-\text{elim}) \quad \frac{\forall x\ :\ A}{A[x/t]}$$

in which a does not occur in the assumption set of premise A, nor within the scope of a quantifier binding x in A; resp. no occurence of x in A occurs within the scope of a quantifier binding a variable of t (i.e. the standard logic stuff usually embodied in the definition of <*term*> *is free for* <*variable*> *in* <*formula*>).

The rules for the existential quantifier are

$$(\exists-\text{intro}) \quad \frac{A[x/t]}{\exists x\ :\ A}$$

$$(\exists-\text{elim}) \quad \frac{\exists x\ :\ A \qquad \begin{matrix}[A[a/x]\,] \\ B\end{matrix}}{B}$$

with appropriate restrictions here too.

Now the modalities. First I define the notion of *essentially modal* formula:

- $\Box A$ is essentially modal;
- $\perp$ is essentially modal;
- the essentially modal formulas are closed under $\wedge$, $\vee$, $\exists$

The rules are

$$(\Box-\text{intro}) \quad \frac{A}{\Box A} \qquad (\Box-\text{elim}) \quad \frac{\Box A}{A}$$

in which the assumption set of the premise of ($\Box-$intro) must include only essentially-modal formulas.

This set of rules defines a natural deduction system whose theorems are those of S4 [Pra65].

3.2.3 Cheating

Actually, we don't use many of these rules in a TLA proof. When we come across a piece of purely propositional or predicate-logical reasoning, we normally *cheat*, which is defined in Section 3.5 as consisting of asserting its validity (if we do so wrongly, then we *lie* instead of cheating, as defined in the same section). The hard part of verifications is normally not the pure logic involved, but rather the mathematics that occurs. So in practice, we cheat on propositional and predicate logic, even in rigorous proofs. This has disadvantages of course, such as occasional lying, but that seems to be reasonably controllable, and tends to disappear with automated help. But now for rules which *do* appear in rigorous TLA proofs.

3.3 Temporal Rules

What proof rules do we need for the temporal part of the logic? First, *Temporal Logic rules* sufficient for '*simple linear-time temporal logic*' (STL), as some computer scientists call it. It's S4.3 plus a bit. While we could extend the Prawitz rules with axioms for S4.3 (axioms are inference rules without premises), we shall also need to be able to understand the existing TLA literature, including proofs, which is formulated somewhat differently by Lamport. Accordingly, let's look briefly at the Lamport axioms for STL. They are introduced by Lamport [Lam94b] as follows (with STL1 cleaned up a bit):

STL1: $\dfrac{F}{\Box F}$
in which $assumptions(F) = \emptyset$

STL2: $\vdash \Box F \Rightarrow F$

STL3: $\vdash \Box\Box F \equiv \Box F$

STL4: $\dfrac{F \Rightarrow G}{\Box F \Rightarrow \Box G}$

STL5: $\vdash \Box(F \wedge G) \equiv (\Box F) \wedge (\Box G)$

STL6: $\vdash (\Diamond\Box F) \wedge (\Diamond\Box G) \equiv \Diamond\Box(F \wedge G)$

STL7: $\vdash \Box\Diamond\Box F \equiv \Diamond\Box F$

Lamport does not use assumptions: the assumption set of each premise is $\emptyset$. Thus it should be clear that many of these rules are derived rules of the Prawitz system. Some hints:

- STL1: follows immediately from $\Box-$intro
- STL2: assume $\Box F$, apply $\Box-$elim then $\Rightarrow -$intro
- STL3: assume $\Box F$, apply $\Box-$intro then $\Rightarrow -$intro; assume $\Box\Box F$; apply $\Box-$elim then $\Rightarrow -$intro; finally apply $\wedge-$intro and definition of $\equiv$
- STL4: assume $\Box F$, apply $\Box-$elim, $\Rightarrow -$elim and $\Box-$intro to get $\Box G$, finally $\Rightarrow -$intro to get $\Box F \Rightarrow \Box G$
- STL5.1: assume $\Box(F \wedge G)$, apply $\Box-$elim then $\wedge-$elim twice then $\Box-$intro twice, followed by $\wedge-$intro to get $(\Box F \wedge \Box G)$
- STL5.2: assume $\Box F \wedge \Box G)$, apply $\wedge-$elim twice then $\Box-$elim twice then $\wedge-$intro, followed by $\Box-$intro to get $\Box(F \wedge G)$

Bryan Olivier has pointed out that STL7 is derivable from classical propositional logic plus STL1-STL6 [Lam96], from which it follows that we only need to add STL6 to the Prawitz rules to obtain STL. Nevertheless, STL7 is useful when we wish to quote it as a derived rule in a proof.

Second, we add the **Basic TLA Rules** which specifically axiomatise '*prime*':

TLA1. $\dfrac{P \wedge (f' = f) \;\Rightarrow\; P'}{\Box P \;\equiv\; P \wedge \Box[P \Rightarrow P']_f}$ TLA2. $\dfrac{P \wedge [\mathcal{A}]_f \;\Rightarrow\; Q \wedge [\mathcal{B}]_g}{\Box P \wedge \Box[\mathcal{A}]_f \;\Rightarrow\; \Box Q \wedge \Box[\mathcal{B}]_g}$

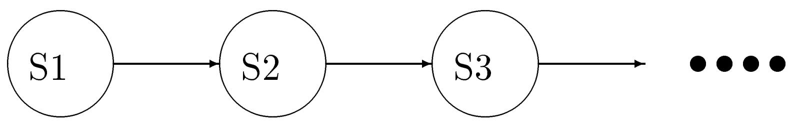
where the notation

$$[\mathcal{A}]_f \;\triangleq\; \mathcal{A} \vee (f' = f)$$

These rules were formulated and used by Lamport specifically for TLA, hence the name [Lam94b].

3.4 Behaviors and the Rules They Engender

At this point, we need to make a brief nod towards the semantics. We use more-or-less the standard '*simple temporal logic*' semantics used in computer science [MP92, Gol92]. We have 'snapshots', i.e. classical model-theoretic worlds, and models of the modal logic will be linear chains of these snapshots, according to classical Kripke semantics with the S4.3 correspondence. We actually want our models, called '*behaviors*' in computer-science parlance, to be $\omega-$sequences of snapshots. That is, chains ordered as the natural numbers are ordered:



Hence we need to add an induction rule:

INV1: $$\frac{I \wedge [\mathcal{N}]_f \Rightarrow I'}{I \wedge \Box[\mathcal{N}]_f \Rightarrow \Box I}$$

in which $\mathcal{N}$ is an action formula, thus defining a relation between a given state and the next. We suppose also that f is the state function which is a sequence whose elements are the variables of $\mathcal{N}$ (TLA includes ZF, so sequences can be formed). The rule says intuitively: if it is provable that I is preserved under state-relation $\mathcal{N}$, then it follows if $\mathcal{N}$ always holds and I holds, I will always hold. We also add the technical rule:

INV2: $\vdash \Box I \;\Rightarrow\; (\Box[\mathcal{N}]_f \;\equiv\; \Box[\mathcal{N} \wedge I \wedge I']_f)$

These two rules are known as the 'Invariance Rules'. We hope that it is clear that these rules are sound. They suffice for many purposes, for example to prove *safety properties* of systems.

The Ontology. What's a system? Let's say: a state machine. What's a state machine? A state machine can be identified with a set of behaviors. To some, that doesn't yet clearly say what it *is*, but I won't define it any other way. (It's an extensional definition, which precludes there being a 'requirements problem': how to ensure that no unwanted system behavior is allowed by the system requirements specification. TLA's extensional conception simply defines this problem away: the system *is* the set of behaviors.) To define a state machine, then, one must specify a set of behaviors, which we shall do in *TLA specifications*.

We have change, and this change is remarked through a syntax which allows names for quantities which may have a different value in each state. We assume that there is a collection of variables which take values in each state of a behavior. So these variables are interpreted by functions whose domain is the set of states in a behavior - so-called *state functions*. Such variables are called *flexible variables*, and are to be distinguished from normal mathematical variables, *rigid variables*, which take one value and one only per 'model' - that is, per behavior. Rigid variables take thus a constant value over a whole behavior, and so as far as the behavior is concerned, they act as a constant. What sorts of values can any variable take? Sets. TLA is based on Zermelo-Fraenkel set theory (ZF), the 'standard' set theory of mathematics. Thus all data structures are sets. This is no restriction - all of math, including the math of data structures, can be formulated in set theory, more-or-less.

Back to Syntax Briefly. Because there are two sorts of variables, there must be two sorts of quantifiers: $\exists$ for rigid variables and $\boldsymbol{\exists}$ for flexible variables. Both quantifiers satisfy similar intro and elim rules.

The Definition of Truth is 'standard' (for applications of temporal logic in computer science): all formulas are interpreted in the *first state* of a behavior. This distinguishes somewhat this semantics from a raw Kripke semantics for a temporal logic. So any purely first-order formulas, not involving temporal operators or *prime*, are true in a behavior if and only if they are true in the first state of the behavior. Temporal formulas $\Box A$ are true if and only if they are true in the first state; which is the case if and only if A is true in *all* states. The semantics of $\Diamond$ follows from the syntactic equivalence $\Diamond A \quad \triangleq \quad \neg\Box\neg A$: $\Diamond A$ is true if and only if it is true in the first state; which is the case if and only if A is true in *some* state (including possibly the first). Formulas including *prime* are evaluated on pairs of states: the values of the unprimed variables are taken from the first state, and the values of the primed variables are taken from the *second* state.

This explanation is of course crude, but I'm hoping it will suffice for our purposes. Those who wish for a precise account of the syntax and semantics of TLA may find one in [Lam94b].

3.5 Hierarchical Proofs

Before we consider proofs in detail, let's revise the idea of how natural deduction proofs proceed from bottom-up to top-down [Lam93, Lam95]. Consider an (informal) procedure with polyadic output:

proof-procedure(T**)**:
To prove T, enumerate a number of assertions, say $A_1, \ldots, A_k$ from which T

1. $A \wedge (P \wedge Q \wedge R)$

PROOF:

1.1. A

PROOF: I guess A just is true. □

1.2. $P \wedge Q \wedge R$

PROOF:

1.2.1. $P \wedge Q$

PROOF:

1.2.1.1. P

PROOF: I guess P just is true. □

1.2.1.2. Q

PROOF: I guess Q just is true. □

1.2.1.3. Q.E.D.

PROOF: $\wedge-$intro from 1.2.1.1 and 1.2.1.2. □

1.2.2. R

PROOF: I guess R just is true. □

1.2.3. Q.E.D.

PROOF: $\wedge-$intro from 1.2.1 and 1.2.2. □

1.3. Q.E.D.

PROOF: $\wedge-$intro from 1.1 and 1.2. □

Figure 1: A simple hierarchical proof

logically follows by means of the rules. Then perform *proof-procedure(A_1);* ... *proof-procedure(A_k)*

To prove theorem A , invoke *proof-procedure(A)*

This describes a recursive procedure for writing natural-deduction style proofs, and its advantages have been explained in [Lam93, Lam95].

proof-procedure(T) has output: where do we write the output? Say, in infix order: right where we invoke *proof-procedure(T)*. For good measure, we enumerate the nodes according to a path-numbering scheme: suppose T has number m. Then A_1 is assigned number $m.1$, ..., A_k is assigned number $m.k$. Consider a trivial example in Figure 1. Instead of using the full path numbering, it is often convenient to use just an indicator of the level of embedding, as in Figure 2. We can rewrite the natural-deduction proof-rules in this style as in Figure 3.

This may at first look tedious, but is in fact a very helpful way to keep

$\langle 1 \rangle 1.$ $A \wedge (P \wedge Q \wedge R)$
PROOF:
$\langle 2 \rangle 1.$ A
PROOF: I guess A just is true. $\square$
$\langle 2 \rangle 2.$ $P \wedge Q \wedge R$
PROOF:
$\langle 3 \rangle 1.$ $P \wedge Q$
PROOF:
$\langle 4 \rangle 1.$ P
PROOF: I guess P just is true. $\square$
$\langle 4 \rangle 2.$ Q
PROOF: I guess Q just is true. $\square$
$\langle 4 \rangle 3.$ Q.E.D.
PROOF: $\wedge-$intro from $\langle 4 \rangle 1$ and $\langle 4 \rangle 2$. $\square$
$\langle 3 \rangle 2.$ R
PROOF: I guess R just is true. $\square$
$\langle 3 \rangle 3.$ Q.E.D.
PROOF: $\wedge-$intro from $\langle 3 \rangle 1$ and $\langle 3 \rangle 2$. $\square$
$\langle 2 \rangle 3.$ Q.E.D.
PROOF: $\wedge-$intro from $\langle 2 \rangle 1$ and $\langle 2 \rangle 2$. $\square$

Figure 2: A simple hierarchical proof with relative labelling

(&-intro):

$\langle 111 \rangle 1.$ A $\wedge$B

PROOF:

$\langle 112 \rangle 1.$ A

$\langle 112 \rangle 2.$ B

$\langle 112 \rangle 3.$ Q.E.D.

PROOF:Follows from $\langle 112 \rangle 1$ and $\langle 112 \rangle 2$ by $\wedge-$intro. $\square$

(&-elim):

$\langle 111 \rangle 1.$ A

PROOF:

$\langle 112 \rangle 1.$ A $\wedge$B

$\langle 112 \rangle 2.$ Q.E.D.

PROOF:Follows from $\langle 112 \rangle 1$ by $\wedge-$elim. $\square$

($\vee-$elim):

$\langle 111 \rangle 1.$ C

PROOF:

$\langle 112 \rangle 1.$ $A \vee B$

ASSUME A

$\langle 112 \rangle 2.$ C

ASSUME B

$\langle 112 \rangle 3.$ C

$\langle 112 \rangle 4.$ Q.E.D.

PROOF:Follows from $\langle 112 \rangle 1$, $\langle 112 \rangle 2$ and $\langle 112 \rangle 3$ by $\wedge-$elim. $\square$

The scope of the ASSUME is the immediately following step. The *assumption set* of a statement is then the set of all ASSUMEd formulas in whose scope it is.

Figure 3: Proof rules in hierarchical style, illustrating assumptions

practical proofs going: one cannot lie, but one can cheat and steal:

lie	$\triangleq$	state a non-logically-true assertion as a proof step
cheat	$\triangleq$	accept the current statement as true without further proof
steal	$\triangleq$	use an automated decision procedure to prove the current statement

The notation $\triangleq$ means '*is defined to be*'.

Hierarchical proofs may be terminated at a *cheat* step or a *steal* step. This is not really doable with the bottom-up approach.

4 TLA, State Machines, and Why Verify?

The *Temporal Logic of Actions* (TLA) was developed in the late 1980's - early 1990's by Leslie Lamport at DEC SRC in Palo Alto, CA to aid in the verification of concurrent computer systems and algorithms. It was also intended to facilitate hand proofs. TLA is a 'standard' linear-time temporal logic based on first-order Zermelo-Fraenkel set theory (ZF) with two sorts of variables (and hence two sorts of quantifiers) and a special operator "*prime*".

4.1 State Machines

A TLA *specification* defines a '*state machine*'. A TLA *verification* consists in taking two TLA specifications, one that describes a more detailed machine (the 'implementation') and one that describes a simpler machine (the 'specification') and a rigorous proof that the implementation-machine *simulates* the specification-machine. This proof is accomplished in TLA by showing, modulo some functional correspondence between the variables, that a canonical specification of the implementation-machine logically implies a canonical specification of the specification-machine: more accurately, since the Deduction Theorem doesn't hold in temporal logic, that

$$\vdash_{TLA} \mathit{Impl.spec} \Rightarrow f(\mathit{Spec.spec})$$

where f is a syntactic substitution function that replaces the free variables in *Spec.spec* with terms in the free variables of *Impl.spec* (state functions). The function f is called a *Refinement Mapping* [AL91]. I shall overload the word '*specification*', which I shall use for both TLA specification and for specification-machine.

So where do all these state machines come from? Consider a Central Processing Unit, a CPU. A CPU is a piece of hardware that **executes** an *instruction*. An instruction operates on *data*. Data may reside locally in *registers*, or in a *cache* elsewhere in the chip, or further away in *memory*, or even further away on *disk*. We have here certain data structures, and certain *actions* specified by instructions.

Actions:
It *adds*, *multiplies* (unless it's a Pentium :-)
puts a value in memory,
gets a value from memory.

Now consider *yourself*, the user. You
save **files**,
send an **email message** to a colleague
        on another machine
compile a **program**
        while reading your mail,
call the **Web pages**
        from a machine in California (say, to learn about *TLA*)

There are again two parts to these descriptions: an *action* and a *data structure*. Some of the actions are general operations on data structures common to all systems. Some are actions and structures specific to what you're doing at the moment. Whether a given function is 'system' or 'application' really depends on convention, that is, how the structure of the system was conceived by its designers - although there is a lot of agreement. Think of a window system, for example. Is it a 'system' function (Windows NT) or 'application software' (Unix, MS-DOS)? (Logically, who cares?)

How is all this done? The fundamental concept is that of a machine. There are *virtual machines*, such as in the system and applications. There are real machines, such as the processor with limited capabilities. An operating system such as Unix gives *interprocess* and *intermachine* communication capabilities. That's a more sophisticated virtual machine. The window system, which organises the user interface, lies 'above' that. One can view the various virtual machines as arranged in a *hierarchy* of levels of increasing functional sophistication, in which a virtual machine at Level n provides functions which are used as 'primitives' by Level $(n + 1)$, and which itself uses the functions provided by Level $(n - 1)$ as primitive.

Interestingly, such ideas have been used in the engineering domain of Man-Machine Interaction [VR92]. It turns out that it is cognitively easier for an operator to understand how a system works if it is presented to himher as an *abstraction hierarchy*, a hierarchical description such as we have just considered.

We should not be surprised that hierarchies of machines are fruitful for design, verification and construction of systems. TLA verification is based on the possibility of writing a system in a hierarchical description, as are other verification systems such as SRI's EHDM [RvHO91] and PVS [Rus].

At the bottom of the hierarchy, the processor '*knows*' how to:
`get`, `add`, `multiply`, `store`
bytes from *memory* and *registers*.

Data structures: bytes.

Near the top of the hierarchy, the networked machine '*knows*':
`verify` password, `store` files
(on another machine),
`call procedure` (from another machine),
`start` applications (editor, mailserver),
`save state` of application,
`swap` one running program for another,
`kill` application.

Data structures are: files, memory locations, byte streams, messages, control blocks, Ethernet addresses, address tables, ports, Internet addresses, passwords, login names,.......

Not only verification procedures, but many structures in computer engineering are based on this idea of a hierarchy of virtual machines. The ISO Open Systems Inteconnection standard for inter-machine communication is based on seven layers of protocols. The TCP/IP packet-switching protocols on which the Internet is based have fewer (four, at last count :-). The *PSOS* (Provably Secure Operating System) [Neu] and the SIFT OS for digital flight control [SRIa] (both SRI, 1970's-80's) were based on hierarchical design so that they could be *verified* (proved correct) by SRI's *EHDM* [RvHO91] (Extended Hierarchical Development Method) verification system. Hierarchical decomposition is still the most fruitful way of proving algorithms and designs correct, and systems such as PVS (from SRI) [Rus] and TLA depend on it.

The lower-level Virtual Machine **implements** the higher-level Virtual Machine, by simulation:

The LLVM *simulates* the HLVM

by

- defining higher-level data structures ‘in terms of’ lower-level data structures:
  words in terms of *bytes*,
  arrays in terms of *words*
  sets in terms of *arrays*
- defining higher-level actions ‘in terms of’ lower-level actions:
  send-message in terms of a *C program*

A TLA program specification

- describes the possible actions in Formal Logic
- describes properties of the machine:
  - Initial (Starting) Condition
  - Safety (the only possible actions are those described)
  - Liveness (if certain actions *can* happen, eventually they *will* happen)

Program specification and verification in TLA proceeds by

- specifying both the HLVM and LLVM,
- then proving using the TLA rules that

$$\text{LLVM.Spec} \Rightarrow f(\text{HLVM.Spec})$$

in which f is a syntactic transformation, replacing the variables of HLVM.Spec by terms in the free variables of LLVM.Spec (the *refinement mapping*). The TLA proof system is supported by

- a specification-writing LaTeX style file
- a proof-writing LaTeX style file
- ...but not yet so far by automated provers. A first attempt was [Eng95]. Stefan Merz in Munich also has an implementation in Isabelle [Mer97], and Sara Kalvala at Warwick is currently implementing TLA in Isabelle [Kal]. So at the moment, stealing (Section 3.5) is harder than it will be....

4.1.1 A Typical Problem with Concurrent Programs

Interacting state machines have their own sets of problems which arise through the interactions. For some reason not very well understood, it seems to be cognitively hard to write concurrent programs and to understand how such programs really operate. Even mathematicians and computer scientists have been caught out by writing 'proofs' of correctness of concurrent algorithms which weren't. Rigor is not only desirable, but seems to be required.

Here's an example of a concurrency problem. Suppose *Program 1* and *Program 2* read a common variable *turn*, which can also be written by *Program 3*. We say that *Program 1*, *Program 2* and *Program 3 share* the variable *turn*. For example, *Program 3* may be the operating system, and *turn* is a variable controlling access to the printer. Just one process at a time may send something to the printer, and the processes can tell whose turn it is by reading the variable *turn*. Suppose *turn* has 3 bits; that the initial value of *turn* is 000; that value 001 corresponds to *Process 1*; and that value 101 corresponds to *Process 2*. All three processes access *turn* from right to left.

Suppose now that the processes access *turn* concurrently, reading its bits one at a time, in the following order:

P3 writes bit3
P2 reads bit3
P1 reads bit3
P3 writes bit2
P2 reads bit2
P1 reads bit2
P1 reads bit
P3 writes bit1
P2 reads bit1

This looks as in Figure 4. P1 reads 001, P2 reads 101, and as a result, both 'think' they have exclusive access to the printer. The result is that both processes send their files to the printer at the same time. An *artistic* outcome, maybe, but not what is wanted. Imagine if this sort of thing were to happen in an airplane flight control system!

4.1.2 Testing Won't Do It

Maybe occurrences of such problems may be found by thorough testing of such systems? Such high-integrity systems as those found in commercial

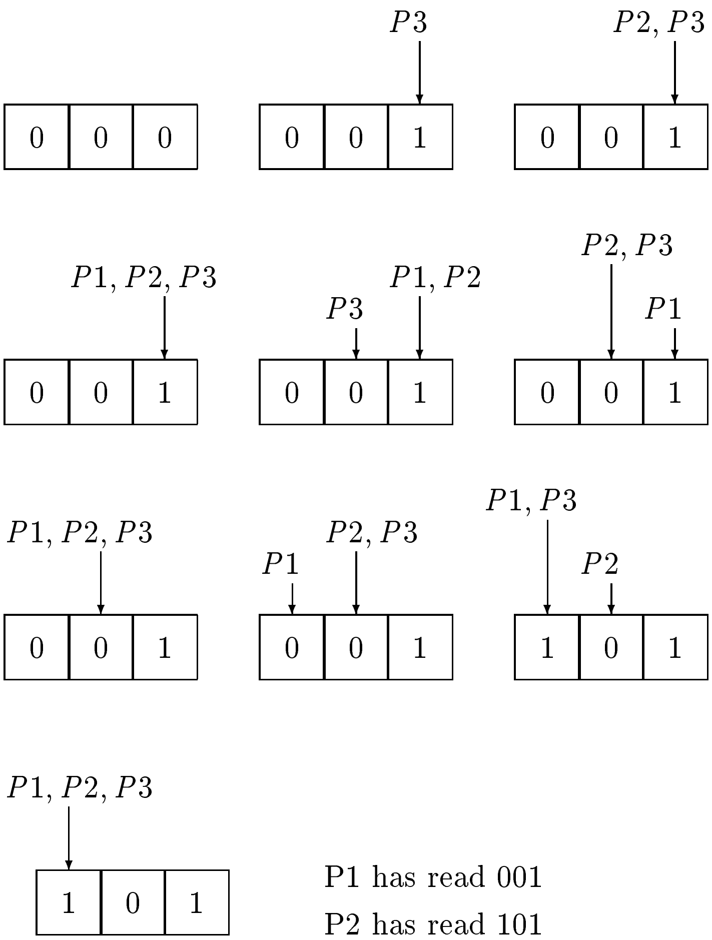


Figure 4: A Concurrent Readers-Writer Mess

aircraft flight control have to be certified to a reliability of 1 in 10^{-9}. That is, one expected failure every 10^9 cycles (trips) [Req92]. There are roughly 500-1,000 aircraft of each common type, except the B737 of which there are about 2,000 as of this writing. Suppose that each makes 10 trips a day (longer-range aircraft that fly intercontinental, such as jumbos, usually make 2 or fewer). That's one failure per 50,000 days for a B737 - one expected failure every 75 years in the whole fleet! That's way longer than the expected life of the airplane type. You can't test for this level of reliability. See [BF93] for frequentist arguments to this conclusion, [LS93] for Bayesian arguments.

By the way, it's worth noting in this context that 'bugs' in computer software or hardware count as *design errors* in traditional engineering disciplines such as aerospace engineering. The term '*failure*' means something breaks :-) It is thus often said that software itself doesn't 'fail' - but that failures may be due to design errors such as occur in software. The use of this vocabulary is a little confusing and it's wise to approach it with care, from either side.

The Bayesian argument is: there is no way to obtain a lower *posterior* probability (expectation of failure) than the *prior* probability (expectation of failure) through testing, at this very high level of required reliability. The only way known to obtain a very high prior is through formal verification - and then this high prior is 1! It follows that one must employ formal verification to ensure a high prior level of reliability. Engineers do not always accept this conclusion but I know of no way around the argument.

5 An Example Specification

Now to the example. I describe and then specify an abstract buffer with two operations:

`push(something)` and `pop`

Figure 5 shows the state of the buffer before each operation and the state after it. Figure 6 shows a concrete 'implementation' of the abstract buffer. A fixed-length data structure, an array, has been chose. Because some of the array elements will not necessarily contain data, we chose a special null-element, $\perp$, to indicate '`nothing here`'. The intended correspondence, 'implementation' to 'specification', is shown in Figure 7 for the data structures, and Figure 8 for the operations.

$$\text{buffer} \qquad\qquad\qquad\qquad\qquad\qquad \text{buffer}'$$

$$< b,x,y,d > \xrightarrow{\quad\text{pop}\quad} < x,y,d >$$

$$< b,x,y,d > \xrightarrow{\quad\text{push(a)}\quad} < b,x,y,d,a >$$

Figure 5: An Abstract Buffer with Operations

How does this fit together? The concrete buffer *simulates* the abstract buffer, and we shall prove that. Simulation means that

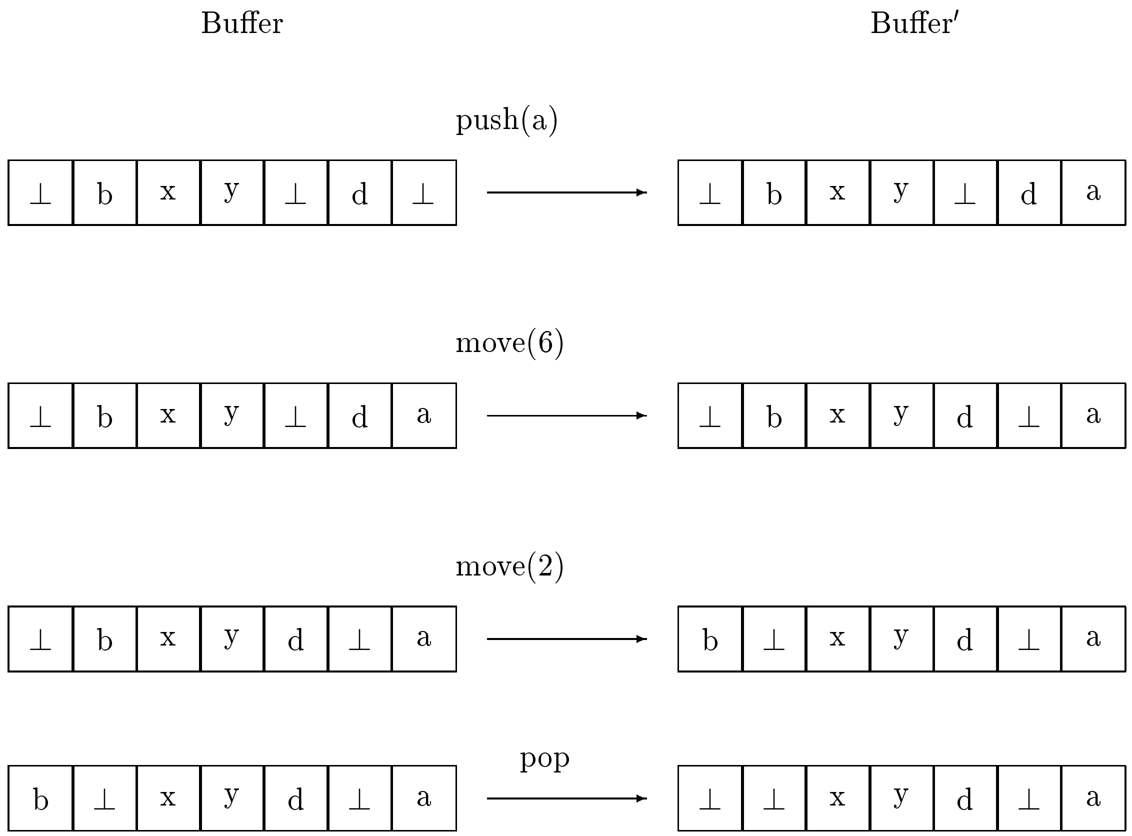
- they start in ‘equivalent’ states
- every action of the concrete buffer corresponds either to an action or to a non-action of the abstract buffer
- when the concrete buffer is sufficiently ‘live’, then the abstract buffer actually does some desired action

This method of *state machine simulation* is common to many verification methods, for example

- TLA of Lamport
- the Input/Output machines of Tuttle, Lynch, Vaandrager (e.g. [Vaa])
- the method of Lam and Shankar (e.g. [LS84]) which is also TL-based

An alternative is to have actions only—then the operation of the system is an abstract machine simulation, but not a *state* machine simulation, since one doesn’t have *state*. This is the set-up in *process algebra*. But one ends up with state anyway - most process algebras have a way of defining state.

How does one specify the actions? We write one from the abstract buffer module in Figure 9. It uses the ‘bulleted list’ notation for conjunctions and



$\perp$ is the symbol for '`nothing here`'

Figure 6: A Concrete Buffer, an Array, with Operations

ConcreteBuffer								AbstractBuffer
$\perp$	b	x	y	$\perp$	d	$\perp$	$\longrightarrow$	$<$ b,x,y,d $>$
$\perp$	b	x	y	$\perp$	d	a	$\longrightarrow$	$<$ b,x,y,d,a $>$
$\perp$	b	x	y	d	$\perp$	a	$\longrightarrow$	$<$ b,x,y,d,a $>$
b	$\perp$	x	y	d	$\perp$	a	$\longrightarrow$	$<$ b,x,y,d,a $>$

Figure 7: Correspondence of the Data Structures

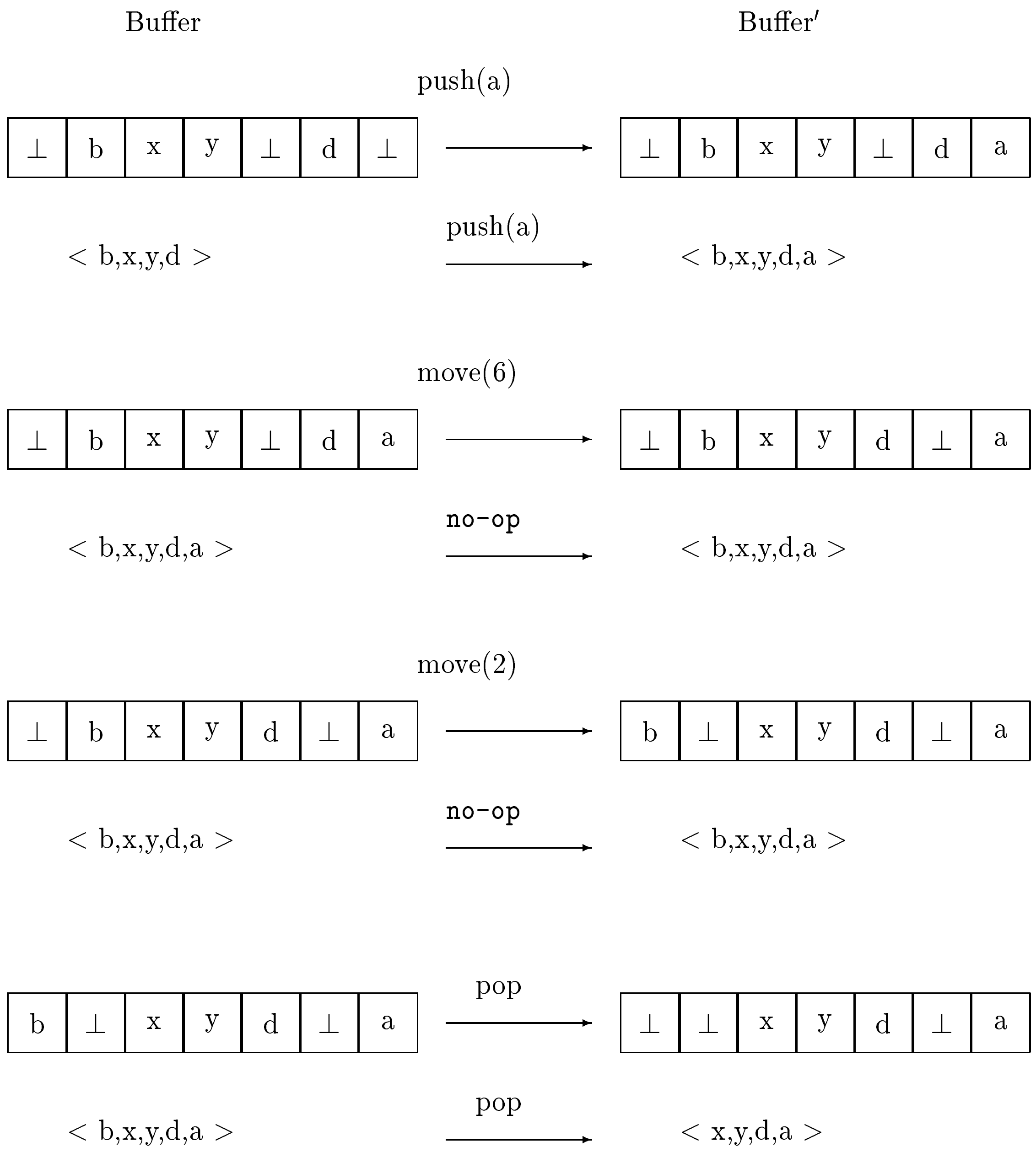


Figure 8: Correspondence of the Operations

module *AbstractBuffer*

...

actions

$$push(a) \triangleq \begin{array}{l} \land\ a \in Data \\ \land\ Len(buffer) < N \\ \land\ buffer' = buffer \circ \langle a \rangle \end{array}$$

...

Recall the notation $\triangleq$ means '*is defined to be*'.

Figure 9: An Action from the Abstract Buffer Module

disjunctions advocated in [Lam94a]. This notation enhances readability, as can be seen by comparing the readability of the definition in Figure 9 with that in Figure 10. If that doesn't persuade you, try Figure 11.

5.1 Some TLA+ Syntax

Now we shall write the two specifications. There are some notions of TLA+ syntax which need to be clarified.

TLA+ is based on modules, which handle the scoping of names (variables, operators, definitions). Flexible and rigid variables must be declared before any use: thus we normally write the declarations first, under the keyword **parameters**. (Flexible variables have type VARIABLE and rigid variables type CONSTANT. 'Data types' aren't declared since everything's a set.)

There are two syntactic operators, **import** and **include** , which bring parts of other modules into the current module. Figure 12 shows a module *Example* with an **import** -ed module *foo* and an **include** -ed module *bar*. The imported module declarations and definitions are as if simply copied into the current module. The imported module *definitions only* are copied in; furthermore the definition names (i.e., the word to the left of a ' $\triangleq$ ') are prefixed with '*BB.*', (as stated by the **as** clause), and the variables (flexible and rigid) are substituted for as specified in the **with** clause. The result is as if the module *ExampleII* in Figure 13 had been written instead of module

module *AbstractBuffer Example I*

. . .

actions

$$push(a) \triangleq \begin{array}{l} \land\ a \in Data \\ \land\ Len(buffer) < N \\ \land\ buffer' = buffer \circ \langle a \rangle \end{array}$$

badly-written actions

$push(a) \triangleq$

$a \in Data \land Len(buffer) < N \land buffer' = buffer \circ \langle a \rangle$

Figure 10: An Action, Well- and Badly-Written

module *AbstractBuffer Example II*

. . .

actions

$$push(a|b) \triangleq \begin{array}{l} \land\ a \in Data \\ \land\ Len(buffer) < N \\ \land\ \lor\ buffer' = buffer \circ \langle a \rangle \\ \quad\ \lor\ buffer' = buffer \circ \langle b \rangle \end{array}$$

badly-written actions

$push(a|b) \triangleq$

$a \in Data \land Len(buffer) < N \land$

$(buffer' = buffer \circ \langle a \rangle \lor buffer' = buffer \circ \langle b \rangle)$

Figure 11: An Action, Well-Written and Really-Badly-Written

Example – with one caveat.

The caveat is that that neither **import** nor **include** are transitive. Importing or including *Example* is thus not the same thing as importing or including *ExampleII*! If one **import** -s *Example* in a further module, only *what is explicitly written as **parameters** and **definitions** is copied*, not items appearing in *Example* under **import** or **include** . Similarly if *Example* is **include** -ed: only **definitions** are included, with the specified definition-name alteration and the substitution of variables. So if you want the other things too, you had better explicitly import or include the modules that *Example* imports and includes.

I introduce some keywords that actually aren't in the TLA+ definition (besides **import** and **include**). TLA+ makes no formal distinction between definitions of predicates, actions, and temporal formulas defining the state machine specification. They are all just **definitions**. I find it helpful to distinguish certain kinds of definitions, such as the definitions of state **predicates**, the definitions of **actions**, and the definition of the **temporal** formula that is the specification of the machine; the formula whose behaviors are supposed to be exactly the possible behaviors of the machine. These keywords in the modules I write are thus to be regarded as aliases for the TLA+ keyword **definitions**. See [Lama] for a syntax definition of TLA+.

5.2 Building the Specification Modules

In order to discuss the construction of the modules, I write pieces of module in the LaTeX module-definition language. This creates pseudo-modules, in that they are not complete or syntactically correct TLA+ modules. I indicate such pseudo-modules by using

- funny names
- elision signs, namely " ...".

This should cause little confusion – I think the reader will easily be able to distinguish pseudo-modules from genuine modules.

There is precisely one flexible variable in each buffer-machine. The abstract one is called *buffer* and the concrete one *Buffer*. These variables must be declared in the TLA+ modules.

First, we define the starting states of the variables, as in Figure 14. The *push* actions are compared in Figure 15, the *pop* actions in Figure 16, and

module *Example*

parameters

x : VARIABLE

a : CONSTANT

imports

foo

includes

bar **as** *BB* **with** $z \leftarrow x$

module *foo*

parameters

y : VARIABLE

b : CONSTANT

definitions

$foodef \triangleq (y' \neq y)$

module *Example*

parameters

z : VARIABLE

c : CONSTANT

definitions

$bardef \triangleq (z' = z)$

Figure 12: Example Modules

module *ExampleII*

parameters

x : VARIABLE
a : CONSTANT
y : VARIABLE
b : CONSTANT

definitions

$foodef \triangleq (y' \neq y)$
$BB.bardef \triangleq (x' = x)$

Figure 13: The Equivalent Example Module

the *move* action, which is intended to correspond to a *no-op* in the abstract buffer, in Figure 17.

We have defined the *initial conditions* and the *actions*. But we don't yet have a *specification*. A specification defines the initial conditions and the actions, **and also** the sentence that says

- the system starts in the initial condition
- if variables change values, it must be because of a defined action (safety)
- it's always true that some desired action eventually happens if it can (liveness)

The entire Abstract Buffer Specification is shown in Figure 18, and the entire Concrete Buffer specification in Figure 19. The temporal logic formula that defines the state machine is called *Spec* in both modules, captioned by the keyword **temporal**. To understand the *Spec*s, it is helpful to recall the notation:

$$[A]_x \text{ means } A \vee (x' = x)$$
$$[A]_{\langle x,y \rangle} \text{ means } A \vee (x' = x \wedge y' = y)$$

Intuitively, $[A]_x$ means *Either A, or x doesn't change value.*

Now we need to state the theorem that is to be proved in the verification: namely, that the implementation machine simulates the specification

module *Bits of AbstractBuffer*

...

predicates

$Init \;\triangleq\; buffer = \langle\,\rangle$

...

module *Bits of ConcreteBuffer*

...

predicates

$Init \;\triangleq\; \forall\, n \in 1..N \;:\; Buffer[n] = \bot$

...

Figure 14: The Start State Definitions, written in Pseudo-Modules

machine. The two sentences defining these machines are compared in the pseudo-module in Figure 20.

To verify the buffer implementation, we may think we want to show that

$$\mathit{Conc\text{-}Buffer\text{-}Spec} \Rightarrow \mathit{Abs\text{-}Buffer\text{-}Spec}$$

But this formula is not provable! *Conc-Buffer-Spec* has a variable *Buffer* which doesn't occur in *Abs-Buffer-Spec*, and *Abs-Buffer-Spec* has a variable *buffer* which doesn't occur in *Conc-Buffer-Spec*. We have forgotten the *refinement mapping*: the state function of the concrete *Buffer* that tells us how the abstract *buffer* is obtained from the concrete one. The refinement mapping shows a uniform way to convert any value of the concrete *Buffer* into a value of the abstract *buffer*. In order properly to state the theorem which we need to prove to prove the machine simulation, we need to state it itself in a TLA+ module, as we do in Figure 21.

The module in Figure 21 includes **include** statements. Recalling what **include** means, the definitions (predicates, actions, and temporals) from *ConcreteBuffer* are valid definitions in this buffer, using the same names but with a prefix given by the **as** clause. The variables used in the **include**-ed

module *Bits of Buffer Actions*

...

abstract actions

$$push(a) \triangleq \begin{array}{l} \land\ a \in Data \\ \land\ Len(buffer) < N \\ \land\ buffer' = buffer \circ \langle a \rangle \end{array}$$

concrete actions

$$push(a) \triangleq \begin{array}{l} \land\ a \in Data \\ \land\ Buffer[N] = \bot \\ \land\ Buffer'[N] = a \\ \land\ \forall\, i \in 1..(N-1)\ : \\ \qquad \text{UNCHANGED } Buffer[i] \end{array}$$

...

Figure 15: The *Push* Definitions Compared

module *Bits of Buffer Actions*

...

abstract actions

$$pop \triangleq \begin{array}{l} \land\ Len(buffer) > 0 \\ \land\ buffer' = tail(buffer) \end{array}$$

concrete actions

$$pop \triangleq \begin{array}{l} \land\ Buffer[1] \neq \bot \\ \land\ Buffer'[1] = \bot \\ \land\ \forall\, i \in 2..N\ : \\ \qquad \text{UNCHANGED } Buffer[i] \end{array}$$

...

Figure 16: The *Pop* Definitions Compared

module *Bits of Buffer Actions*

...

abstract actions

$no - op \triangleq ???$

concrete actions

$$
\begin{aligned}
move(k) \triangleq\ & \land\ k \in 2..N \\
& \land\ Buffer[k] \neq \bot \\
& \land\ Buffer[k-1] = \bot \\
& \land\ Buffer'[k] = \bot \\
& \land\ Buffer'[k-1] = Buffer[k] \\
& \land\ \forall\, i \in 1..(k-2) : \\
& \qquad \text{UNCHANGED } Buffer[i] \\
& \land\ \forall\, i \in (k+1)..N : \\
& \qquad \text{UNCHANGED } Buffer[i]
\end{aligned}
$$

...

Figure 17: The *Move* Definitions Compared

module *AbstractBuffer*

imports

Sequences, *Naturals*

parameters

buffer : VARIABLE

Data, *N* : CONSTANT

predicates

$Init \triangleq buffer = \langle\, \rangle$

actions

$$
\begin{aligned}
push(a) \triangleq\ & \land\ a \in Data \\
& \land\ Len(buffer) < N \\
& \land\ buffer' = buffer \circ \langle\, a \,\rangle
\end{aligned}
$$

$$
\begin{aligned}
pop \triangleq\ & \land\ Len(buffer) > 0 \\
& \land\ buffer' = tail(buffer)
\end{aligned}
$$

temporal

$$
\begin{aligned}
Spec \triangleq\ & \land\ Init \\
& \land\ \Box[pop \lor \exists\, b : push(b)]_{buffer} \\
& \land\ WF_{buffer}(pop)
\end{aligned}
$$

Figure 18: The Abstract Buffer Specification

module *ConcreteBuffer*

imports

Naturals

parameters

Buffer : VARIABLE

Data, N : CONSTANT

assertions

$\bot \notin \mathit{Data}$

predicates

$\mathit{Init} \triangleq \land \forall n \in 1..N : \mathit{Buffer}[n] = \bot$

actions

$$
\begin{aligned}
\mathit{push}(a) \triangleq\ & \land a \in \mathit{Data} \\
& \land \mathit{Buffer}[N] = \bot \\
& \land \mathit{Buffer}'[N] = a \\
& \land \forall i \in 1..(N-1) : \\
& \qquad \text{UNCHANGED } \mathit{Buffer}[i]
\end{aligned}
$$

$$
\begin{aligned}
\mathit{pop} \triangleq\ & \land \mathit{Buffer}[1] \neq \bot \\
& \land \mathit{Buffer}'[1] = \bot \\
& \land \forall i \in 2..N : \\
& \qquad \text{UNCHANGED } \mathit{Buffer}[i]
\end{aligned}
$$

$$
\begin{aligned}
\mathit{move}(k) \triangleq\ & \land k \in 2..N \\
& \land \mathit{Buffer}[k] \neq \bot \\
& \land \mathit{Buffer}[k-1] = \bot \\
& \land \mathit{Buffer}'[k] = \bot \\
& \land \mathit{Buffer}'[k-1] = \mathit{Buffer}[k] \\
& \land \forall i \in 1..(k-2) : \\
& \qquad \text{UNCHANGED } \mathit{Buffer}[i] \\
& \land \forall i \in (k+1)..N : \\
& \qquad \text{UNCHANGED } \mathit{Buffer}[i]
\end{aligned}
$$

temporal

$$
\begin{aligned}
\mathit{Spec} \triangleq\ & \land \mathit{Init} \\
& \land \Box[\lor \mathit{pop} \\
& \qquad \lor \exists b : \mathit{push}(b) \\
& \qquad \lor \exists k : \mathit{move}(k)]_{\mathit{Buffer}} \\
& \land \mathit{WF}_{\mathit{Buffer}}(\mathit{pop}) \\
& \land \mathit{WF}_{\mathit{Buffer}}(\exists k : \mathit{move}(k))
\end{aligned}
$$

Figure 19: The Concrete Buffer Specification

module *Specification Comparison*

...

temporal

$$
\begin{aligned}
&Conc - Buffer - Spec \ \triangleq \\
&\qquad \land\ Init \\
&\qquad \land\ \Box[\lor\ pop \\
&\qquad\qquad \lor\ \exists\, b\ :\ push(b) \\
&\qquad\qquad \lor\ \exists\, k\ :\ move(k)]_{Buffer} \\
&\qquad \land\ WF_{Buffer}(pop) \\
&\qquad \land\ WF_{Buffer}(\exists\, k\ :\ move(k))
\end{aligned}
$$

$$
\begin{aligned}
&Abs - Buffer - Spec \ \triangleq \\
&\qquad \land\ Init \\
&\qquad \land\ \Box[\lor\ pop \\
&\qquad\qquad \lor\ \exists\, b\ :\ push(b)]_{buffer} \\
&\qquad \land\ WF_{buffer}(pop)
\end{aligned}
$$

Figure 20: Comparison of the Two *Spec* Statements

module *Theorems*

parameters
$Buffer, buffer$: VARIABLE
$Data, N$: CONSTANT

includes
$ConcreteBuffer$ **as** $CB(N)$
$AbstractBuffer$ **as** $AB(N)$

theorems
$CB(N).Spec \Rightarrow AB(N).Spec[buffer/f(Buffer)]$

$f(Buffer)$ is the *Refinement Mapping* that we must define.

Figure 21: The Verification Theorem

definitions must be explicitly defined (or **import**-ed) in the current (**include**-ing) module, before they are used, that is, before the **include** statements. Hence the flexible variables *Buffer* and *buffer* must be declared, as must the rigid variables *Data* and *N*. The action definition names become, e.g. $CB.move$ and $AB.pop$, and of course the specification formulae $AB.Spec$ and $CB.Spec$.

By convention, we let

$$\overline{buffer} \triangleq f(Buffer)$$

(this is the *Refinement Mapping*) and by extension $\overline{AB(N).Spec}$ be $AB(N).Spec$, with every occurrence of *buffer* replaced by $\overline{buffer}$.

6 The Proof

We must prove
$\langle 1 \rangle 1.$ $CB(N).Spec \Rightarrow \overline{AB(N).Spec}$

6.1 Partitioning the Proof

We are proceeding top-down, using the hierarchical method and the LaTeX support style files tla.sty and pf.sty, which includes the proof-step numbering scheme. Each specification is a conjunction of three clauses

$$Init \wedge \Box[action \vee action \vee ...]_{variables} \wedge Liveness$$

so it seems most reasonable to prove each conjunct separately:

$\langle 2\rangle 1.\ \ CB(N).Spec \Rightarrow \overline{AB(N).Init}$
$\langle 2\rangle 2.\ \ CB(N).Spec \Rightarrow \Box[\vee\ \overline{AB(N).pop}$
$\qquad\qquad\qquad\qquad\qquad \vee\ \overline{\exists\, a\ :\ AB(N)push(a)}]_{\overline{buffer}}$
$\langle 2\rangle 3.\ \ CB(N).Spec \Rightarrow \overline{WF_{buffer}(AB(N).pop)}$
$\langle 2\rangle 4.$ Q.E.D.
PROOF: Follows by propositional logic from the conjunction of steps $\langle 2\rangle 1$, $\langle 2\rangle 2$ and $\langle 2\rangle 3$. □

The first step of this, along with its substep, is

$\langle 2\rangle 1.\ \ CB(N).Spec \Rightarrow \overline{AB(N).Init}$
$\langle 3\rangle 1.\ \ CB(N).Init \Rightarrow \overline{AB(N).Init}$
$\langle 3\rangle 2.$ Q.E.D.
PROOF: Follows directly by propositional logic. □

The second step, along with its substep, is

$\langle 2\rangle 2.\ \ CB(N).Spec \Rightarrow \Box[\vee\ \overline{AB(N).pop}$
$\qquad\qquad\qquad\qquad\qquad \vee\ \overline{\exists\, a\ :\ AB(N)push(a)}]_{\overline{buffer}}$
PROOF:
$\langle 3\rangle 1.\ \ \wedge\ CB(N).Init$
$\qquad \wedge\ \Box[\vee\ CB(N).pop$
$\qquad\qquad \vee\ \exists\, a\ :\ CB(N).push(a)$
$\qquad\qquad \vee\ \exists\, k\ :\ CB(N).move(k)]_{Buffer}$
$\qquad \Rightarrow \Box[\vee\ \overline{AB(N).pop}$
$\qquad\qquad \vee\ \overline{\exists\, a\ :\ AB(N)push(a)}]_{\overline{buffer}}$
$\langle 3\rangle 2.$ Q.E.D.
PROOF: Follows directly by propositional logic. □

The third step,

$\langle 2\rangle 3.$ $CB(N).Spec \Rightarrow \overline{WF_{buffer}(AB(N).pop)}$

had better wait until we understand a little more about liveness.

6.2 The Initial Condition

The initial condition substep is trivial to prove:

$\langle 3\rangle 1.$ $CB(N).Init \Rightarrow \overline{AB(N).Init}$
  PROOF: .... is simply trivial math. $\square$

Putting it together, we have for this step

$\langle 2\rangle 1.$ $CB(N).Spec \Rightarrow \overline{AB(N).Init}$
  $\langle 3\rangle 1.$ $CB(N).Init \Rightarrow \overline{AB(N).Init}$
    PROOF: .... is simply trivial math. $\square$
  $\langle 3\rangle 2.$ Q.E.D.
    PROOF: Follows directly by propositional logic. $\square$

This is thus what the first part of the hierarchical proof looks like. The safety condition is more complicated.

6.3 The Safety Condition

First, a trivial logical manipulation. We can get rid of one unused conjunct of the antecedent:

$$\begin{array}{l}\langle 3\rangle 1.\ \wedge\ CB(N).Init \\ \qquad \wedge\ \Box[\vee\ CB(N).pop \\ \qquad\qquad \vee\ \exists\, a\ :\ CB(N).push(a) \\ \qquad\qquad \vee\ \exists\, k\ :\ CB(N).move(k)]_{Buffer} \\ \qquad \Rightarrow \Box[\vee\ \overline{AB(N).pop} \\ \qquad\qquad \vee\ \overline{\exists\, a\ :\ AB(N)push(a)}]_{\overline{buffer}}\end{array}$$

  PROOF:

$$\begin{array}{l}\langle 4\rangle 1.\ \Box[\vee\ CB(N).pop \\ \qquad\qquad \vee\ \exists\, a\ :\ CB(N).push(a) \\ \qquad\qquad \vee\ \exists\, k\ :\ CB(N).move(k)]_{Buffer} \\ \qquad\qquad\quad \Rightarrow \Box[\vee\ \overline{AB(N).pop} \\ \qquad\qquad\qquad\quad \vee\ \overline{\exists\, a\ :\ AB(N)push(a)}]_{\overline{buffer}}\end{array}$$

$\langle 4 \rangle 2.$ Q.E.D.

PROOF:Trivial propositional logic. $\square$

and this substep may in turn be proved using an application of STL4:

$$\begin{aligned}
\langle 4 \rangle 1.\ \ & \Box[\lor\ CB(N).pop \\
& \quad \lor \exists\, a\ :\ CB(N).push(a) \\
& \quad \lor \exists\, k\ :\ CB(N).move(k)]_{Buffer} \\
& \qquad \Rightarrow \Box[\lor\ \overline{AB(N).pop} \\
& \qquad\qquad \lor\ \overline{\exists\, a\ :\ AB(N)push(a)}]_{\overline{buffer}}
\end{aligned}$$

PROOF:

$$\begin{aligned}
\langle 5 \rangle 1.\ \ & (\lor\ CB(N).pop \\
& \quad \lor \exists\, a\ :\ CB(N).push(a) \\
& \quad \lor \exists\, k\ :\ CB(N).move(k) \\
& \quad \lor\ Buffer' = Buffer) \\
& \qquad \Rightarrow (\lor\ \overline{AB(N).pop} \\
& \qquad\qquad \lor\ \overline{\exists\, a\ :\ AB(N)push(a)} \\
& \qquad\qquad \lor\ \overline{buffer}' = \overline{buffer})
\end{aligned}$$

$\langle 5 \rangle 2.$ Q.E.D.

PROOF: directly by STL4:

$$\frac{F \Rightarrow G}{\Box F\ \Rightarrow\ \Box G}$$

Notice that we have now got rid of all the temporal operators, and we're left with a substep that is 'pure math' - that is, first-order set theory at most. Bye, bye temporal logic. So let's do it......

$$\begin{aligned}
\langle 5 \rangle 1.\ \ & (\lor\ CB(N).pop \\
& \quad \lor \exists\, a\ :\ CB(N).push(a) \\
& \quad \lor \exists\, k\ :\ CB(N).move(k) \\
& \quad \lor\ Buffer' = Buffer) \\
& \qquad \Rightarrow (\lor\ \overline{AB(N).pop} \\
& \qquad\qquad \lor\ \overline{\exists\, a\ :\ AB(N)push(a)} \\
& \qquad\qquad \lor\ \overline{buffer}' = \overline{buffer})
\end{aligned}$$

PROOF:

$\langle 6 \rangle 1.$ $CB(N).pop \Rightarrow \overline{AB(N).pop}$

$\langle 6 \rangle 2.$ $\exists\, a\ :\ CB(N).push(a) \Rightarrow \exists\, a\ :\ \overline{AB(N).push(a)}$

$\langle 6 \rangle 3.$ $\exists\, k\ :\ CB(N).move(k) \Rightarrow \overline{buffer}' = \overline{buffer}$

$\langle 6 \rangle 4.$ $Buffer' = Buffer \Rightarrow \overline{buffer}' = \overline{buffer}$

$\langle 6 \rangle 5.$ Q.E.D.

PROOF: Follows by propositional logic from $\langle 6 \rangle 1$, $\langle 6 \rangle 2$, $\langle 6 \rangle 3$, and $\langle 6 \rangle 4$:

$$(A \Rightarrow X) \wedge (B \Rightarrow Y) \wedge (C \Rightarrow Z) \wedge (D \Rightarrow Z)$$
$$\Rightarrow (A \vee B \vee C \vee D \Rightarrow X \vee Y \vee Z) \ \Box$$

We'll now select one of these substeps to prove as an example, namely $\langle 6 \rangle 2$: a concrete *push* is an abstract *push*.

The substep for this assertion consists in getting rid of the rigid-variable quantifiers. It's the same value a that is going to be *push*'d on either buffer, so it's rather easy to get rid of the quantifiers:

$\langle 6 \rangle 2.$ $\exists\, a \,:\, CB.push(a)$
$$\Rightarrow$$
$$\overline{\exists\, a \,:\, AB.push(a)}$$

PROOF:

LET: a : constant

$\langle 7 \rangle 1.$ $CB.push(a)$
$$\Rightarrow$$
$$\overline{AB.push}(a)$$

$\langle 7 \rangle 2.$ Q.E.D.

Follows from $\langle 7 \rangle 1$ by predicate logic. $\Box$

Proving this statement is a matter of unravelling the definitions of the actions. *AB.push* is a conjunct, so again the substeps consist in proving each of the conjuncts separately from the antecedent. The first is trivial:

$\langle 7 \rangle 1.$ $CB.push(a)$
$$\Rightarrow$$
$$\overline{AB.push}(a)$$

PROOF:

$\langle 8 \rangle 1.$ $CB.push(a) \Rightarrow a \in Data$

PROOF: Immediate from the definition of $CB.push$. $\Box$

$\langle 8 \rangle 2.$ $CB.push(a) \Rightarrow Len(\overline{buffer}) < N$

$\langle 8 \rangle 3.$ $CB.push(a) \Rightarrow \overline{buffer}' = \overline{buffer} \circ \langle\, a \,\rangle$

$\langle 8 \rangle 4.$ Q.E.D.

PROOF: Follows immediately from $\langle 8 \rangle 1$, $\langle 8 \rangle 2$ and $\langle 8 \rangle 3$ using propositional logic. $\Box$

Now, at this point, it would be good to know what the refinement mapping actually is! The correspondence has been given intuitively in the pictures and explanation of how the variables correspond, but now it's time for a syntactical definition. Basically, we want to take the non-$\perp$ elements of the array and arrange them in a sequence in the order in which they occur in the array. It turns out in TLA that an array of length N is a finite function with domain $1..N \triangleq \{n \in \mathsf{Nat} : 1 \leq n \leq N\}$, where Nat is the set of natural numbers. A sequence is also defined as a finite function with domain $1..k$ for some $k \in \mathsf{Nat}$. The point here is that *Buffer* and *buffer* don't always have the same length – *buffer* just contains the non-$\perp$ elements of *Buffer*. So we define a function $\mathit{SelectSeq}(\langle \mathit{sequence} \rangle, \langle \mathit{Predicate} \rangle)$, which makes a subsequence of $\langle \mathit{sequence} \rangle$ of exactly those elements that satisfy the $\langle \mathit{Predicate} \rangle$. We shall need some auxiliary math definitions from the module *Sequences*, defined in Figure 22. Definition of this module is by Lamport. Don't let the **let...in** syntax in *SelectSeq* worry you - it's just a way of formally defining an auxiliary notation (with the **let**) and then inserting some actual parameters (in the **in**). *SelectSeq* is defined iteratively as one goes through the sequence: the *head* of the sequence is *test*-ed. If it satisfies the predicate *test*, it's left in and we proceed to the *tail* of the sequence. If not, it's omitted. So in the refinement mapping, we proceed through the sequence *Buffer*, testing each element to see whether it's non-$\perp$, and including it in $\overline{\mathit{buffer}}$ just in case it is.

The refinement mapping may now be defined as

$$\mathit{NonVoid}(k) \triangleq k \neq \perp$$

$$\overline{\mathit{buffer}} \triangleq \mathit{SelectSeq}(\mathit{Buffer}, \mathit{NonVoid})$$

6.4 Digging Down into the Details of Safety

The two remaining clauses of the *push* simulation require some detailed reasoning. Here's the first one, with its substeps. I cheat (remember the definition of cheating!) on the second substep.

$\langle 8 \rangle 2.$ $\mathit{CB.push}(a) \Rightarrow \mathit{Len}(\overline{\mathit{buffer}}) < N$
  PROOF:
  $\langle 9 \rangle 1.$ $\mathit{CB.push}(a) \Rightarrow \mathit{Buffer}[N] = \perp$

module *Sequences*

import *Naturals*

$m..n \triangleq \{i \in Nat : (m \leq i) \land (i \leq n)\}$

$Len(s) \triangleq$

$\quad \text{CHOOSE } n : (n \in Nat) \land ((\text{DOMAIN } s) = (1..n))$

$Head(s) \triangleq s[1]$

$Tail(s) \triangleq [i \in 1..(Len(s) - 1) \mapsto s[i+1]]$

$s \circ t \triangleq$

$\quad [i \in 1..(Len(s) + Len(t)) \mapsto$

$\qquad \textbf{if } i \leq Len(s) \textbf{ then } s[i]$

$\qquad\qquad\qquad \textbf{else } t[i - Len(s)]]$

$Seq(S) \triangleq \text{UNION } \{[(1..n) \rightarrow S] : n \in Nat\}$

$SubSeq(s, m, n) \triangleq$

$\quad [i \in (1..(1 + n - m)) \mapsto s[i + m - 1]]$

$SelectSeq(s, test()) \triangleq$

$\textbf{let } F[\, t : Seq(\{s[i] : i \in (1..Len(s))\})\,] \triangleq$

$\qquad \textbf{if } t = \langle\, \rangle \textbf{ then } \langle\, \rangle$

$\qquad\qquad \textbf{else } \textbf{if } test(Head(t))$

$\qquad\qquad\qquad \textbf{then}$

$\qquad\qquad\qquad\quad \langle Head(t) \rangle \circ$

$\qquad\qquad\qquad\quad F[\, Tail(t)]$

$\qquad\qquad\qquad \textbf{else } F[\, Tail(t)]$

$\textbf{in } F[s]$

Figure 22: The Module *Sequences*

PROOF: Immediate from the definition of $CB.push(a)$. □

⟨9⟩2. $Buffer[N] = \bot \Rightarrow$
$Len(SelectSeq(Buffer, NonVoid)) < N$

PROOF: Follows from the definition of *SelectSeq* and *Len*, along with a certain amount of data structure manipulation, which is omitted. □

⟨9⟩3. Q.E.D.

PROOF: Follows from ⟨9⟩1, ⟨9⟩2 and the definition of $\overline{buffer}$ by propositional logic. □

Now for the third substep of the *push* simulation assertion. This breaks down into three substeps, which are all math.

⟨8⟩3. $CB.push(a) \Rightarrow \overline{buffer}' = \overline{buffer} \circ \langle a \rangle$

PROOF:

⟨9⟩1. $CB.push(a) \Rightarrow$
$SelectSeq(Buffer, NonVoid) =$
$SelectSeq($
$[i \in 1..(N-1) \mapsto Buffer[i]],$
$NonVoid)$

⟨9⟩2. $CB.push(a)$
$\Rightarrow$
$Buffer' =$
$[i \in 1..(N-1) \mapsto Buffer[i]] \circ \langle a \rangle$

⟨9⟩3. $SelectSeq([i \in 1..(N-1) \mapsto Buffer[i]]$
$\circ \langle a \rangle, NonVoid) =$
$SelectSeq([i \in 1..(N-1) \mapsto Buffer[i]], NonVoid)$
$\circ \langle a \rangle$

⟨9⟩4. Q.E.D.

PROOF: Follows immediately from ⟨9⟩1, ⟨9⟩2, ⟨9⟩3 by propositional logic, substitution, and the definition of $\overline{buffer}$. □

We take these one at a time, in order. First, the first:

⟨9⟩1. $CB.push(a) \Rightarrow$
$SelectSeq(Buffer, NonVoid) =$
$SelectSeq($
$[i \in 1..(N-1) \mapsto Buffer[i]],$
$NonVoid)$

PROOF:

$\langle 10\rangle 1.$ $CB.push(a) \Rightarrow Buffer[N] = \bot$

PROOF: Immediate from the definition of $CB.push(a)$. □

$\langle 10\rangle 2.$ $Buffer[N] = \bot$

$$\Rightarrow$$
$$SelectSeq(Buffer, NonVoid) =$$
$$SelectSeq([i \in 1..(N-1) \mapsto Buffer[i]], NonVoid)$$

PROOF: Follows immediately from the definition of *SelectSeq* using manipulations of the data structure. □

$\langle 10\rangle 3.$ Q.E.D.

PROOF: Follows immediately from $\langle 10\rangle 1$ and $\langle 10\rangle 2$ by propositional logic. □

Next, the second and third are really straightforward, if one cheats (definition!) a little:

$\langle 9\rangle 2.$ $CB.push(a)$

$$\Rightarrow$$
$$Buffer' =$$
$$[i \in 1..(N-1) \mapsto Buffer[i]] \circ \langle a \rangle$$

PROOF: Follows from the definition of $CB.push$ and the sequence operations. □

$\langle 9\rangle 3.$ $SelectSeq([i \in 1..(N-1) \mapsto Buffer[i]] \circ \langle a \rangle, NonVoid) =$
$SelectSeq([i \in 1..(N-1) \mapsto Buffer[i]], NonVoid) \circ \langle a \rangle$

PROOF: Follows from the definition of *SelectSeq* and *NonVoid*. □

So where are we? To recap: we're done with the $\langle 9\rangle$'s, which proved $\langle 8\rangle 3$; we'd done $\langle 8\rangle 2$ all in one, and $\langle 8\rangle 1$ was trivial; thus we've proved

$\langle 6\rangle 1.$ $CB.push(a)$

$$\Rightarrow$$
$$\overline{AB.push}(a)$$

which in turn proved

$\langle 5 \rangle 2.\ \exists\, a\ :\ CB.push(a)$
$$\Rightarrow$$
$$\overline{\exists\, a\ :\ AB.push(a)}$$

....which is the second clause in the simulation theorem:

$$\begin{array}{l}
\langle 4 \rangle 1.\ (\lor\ CB(N).pop \\
\quad \lor \exists\, a\ :\ CB(N).push(a) \\
\quad \lor \exists\, k\ :\ CB(N).move(k) \\
\quad \lor\ Buffer' = Buffer) \\
\qquad \Rightarrow (\lor\ \overline{AB(N).pop} \\
\qquad\quad \lor\ \overline{\exists\, a\ :\ AB(N)push(a)} \\
\qquad\quad \lor\ n\overline{buffer}' = \overline{buffer})
\end{array}$$

We leave the other two clauses (for *pop* and *move*) to the reader, and hereby deem this proof completed! All the substeps have been proved. The buffer implementation is safe. The full proof, without cheating, in full hierarchical style, may be seen in [Lad96a].

6.5 The Proof Scheme

The reader will have observed that there is a scheme to this. Now we'll take a look at the general *proof scheme* for TLA verifications. In the proof scheme, some of the steps are *pro forma* applications of inference rules, and some require different proofs for each verification. The steps that are new for each verification are indicated by *******Novel.******* in the proof scheme. The proof scheme helps to control the proof.

The proof scheme is a little more complicated than the buffer example we have just seen. In a general safety proof, we shall need to find an *invariant*. An invariant is an assertion that remains always true throughout every behavior of the machine; that is, an assertion of the form $\Box A$ that can be proved from the safety property and initial condition. This invariant will then be used to establish the safety property of the specification machine. This proof method is known as the *invariant assertion* method, and was first adapted for use in concurrent programs by Ashcroft [Ash75] and extended by Owicki and Gries [OG76] and Lamport, amongst others. Choosing a good invariant is often the hardest part of a safety proof. There is no formula for picking an invariant – it requires practice and experience.

$\langle 1 \rangle 1.$ $(\mathit{Init}_1 \wedge \Box[N_1]_v \wedge L_1)$
$\qquad \Rightarrow (\overline{\mathit{Init}_2} \wedge \Box\overline{[N_2]_w} \wedge \overline{L_2})$

PROOF:

$\langle 2 \rangle 1.$ $\mathit{Init}_1 \Rightarrow \overline{\mathit{Init}_2}$

PROOF: ******Novel.******. $\square$

$\langle 2 \rangle 2.$ $\mathit{Init}_1 \wedge \Box[N_1]_v \Rightarrow \Box\overline{[N_2]_w}$

PROOF: See next slides. $\square$

$\langle 2 \rangle 3.$ $\mathit{Init}_1 \wedge \Box[N_1]_v \wedge L_1 \Rightarrow \overline{L_2}$

PROOF: ******Novel.******. $\square$

$\langle 2 \rangle 4.$ Q.E.D.

PROOF: By propositional logic from $\langle 2 \rangle 1$, $\langle 2 \rangle 2$, $\langle 2 \rangle 3$. $\square$

Note that the novel step $\langle 2 \rangle 1$ is pure math. Let's look at $\langle 2 \rangle 2$: *safety*:

$\langle 2 \rangle 2.$ $\mathit{Init}_1 \wedge \Box[N_1]_v \Rightarrow \Box\overline{[N_2]_w}$

PROOF: First we must find an invariant *Inv*:

$\langle 3 \rangle 1.$ $\mathit{Init}_1 \wedge \Box[N_1]_v \Rightarrow \Box \mathit{Inv}$

$\langle 3 \rangle 2.$ $\Box \mathit{Inv} \wedge \Box[N_1]_v \Rightarrow \Box[\overline{N_2}]_{\overline{w}}$

$\langle 3 \rangle 3.$ Q.E.D.

PROOF: Immediate from $\langle 3 \rangle 1$ and $\langle 3 \rangle 2$ by propositional logic. $\square$

The substeps to prove the invariant from the safety property of the implementation machine are:

$\langle 3 \rangle 1.$ $\mathit{Init}_1 \wedge \Box[N_1]_v \Rightarrow \Box \mathit{Inv}$

$\langle 4 \rangle 1.$ $\mathit{Init}_1 \Rightarrow \mathit{Inv}$

PROOF: ******Novel.****** $\square$

$\langle 4 \rangle 2.$ $\mathit{Inv} \wedge \Box[N_1]_v \Rightarrow \Box \mathit{Inv}$

PROOF:

$\langle 5 \rangle 1.$ $\mathit{Inv} \wedge [N_1]_v \Rightarrow \mathit{Inv}'$

PROOF: ******Novel.****** $\square$

$\langle 5 \rangle 2.$ Q.E.D.

PROOF: Immediate from $\langle 5 \rangle 1$ by Rule INV1. $\square$

$\langle 4 \rangle 3.$ Q.E.D.

PROOF: Immediate from $\langle 4 \rangle 1$ and $\langle 4 \rangle 2$
by propositional logic. $\square$

Now the rest: using the invariant to prove the safety property of the specification machine:

$\langle 3\rangle 2.$ $\Box Inv \wedge \Box[N_1]_v \Rightarrow \Box[\overline{N_2}]_{\overline{w}}$
PROOF:
$\langle 4\rangle 1.$ $\Box Inv \wedge \Box[N_1]_v \Rightarrow \Box[N_1 \wedge Inv \wedge Inv']_v$
PROOF: Immediate from INV2 using STL5. □
$\langle 4\rangle 2.$ $\Box[N_1 \wedge Inv \wedge Inv']_v \Rightarrow \Box[\overline{N_2}]_{\overline{w}}$
PROOF:
$\langle 5\rangle 1.$ $[N_1 \wedge Inv \wedge Inv']_v \Rightarrow [\overline{N_2}]_{\overline{w}}$
PROOF: ******Novel.******. □
$\langle 5\rangle 2.$ Q.E.D.
PROOF: Immediate from $\langle 5\rangle 1$ by STL4. □
$\langle 4\rangle 3.$ Q.E.D.
PROOF: Immediate from $\langle 4\rangle 1$ and $\langle 4\rangle 2$ by propositional logic. □

6.5.1 The Proof Obligations

We can see from this scheme that there are novel parts and pro forma parts. The novel parts, which must be proved anew for each example, are called *proof obligations*. Thus the proof obligations are:

$\langle 2\rangle 2.$ $Init_1 \Rightarrow \overline{Init_2}$
$\langle 2\rangle 3.$ (Safety)
$\langle 3\rangle 1.$ (Inv−holds)
$\langle 4\rangle 1.$ $Init_1 \Rightarrow Inv$
$\langle 4\rangle 2.$ (Inv is persistent)
$\langle 5\rangle 1.$ $Inv \wedge [N_1]_v \Rightarrow Inv'$
$\langle 3\rangle 2.$ (Inv−used)
$\langle 4\rangle 1.$ (technical step)
$\langle 4\rangle 2.$ (persisent simulation)
$\langle 5\rangle 1.$ $[N_1 \wedge Inv \wedge Inv']_v \Rightarrow [\overline{N_2}]_{\overline{w}}$

Notice that all of these proof obligations are statements of TLA without temporal operators. Therefore, they are ZF set theory only. No temporal-logical inference needed.

The step
$\langle 5\rangle 1.$ $Inv \wedge [N_1]_v \Rightarrow Inv'$
may (should) also be broken down into individual actions:

$$N_1 \triangleq A_1 \vee A_2 \vee \ldots \vee A_k$$

$\langle 5 \rangle 1.$ $Inv \wedge [N_1]_v \Rightarrow Inv'$
PROOF:
$\langle 6 \rangle 1.$ $Inv \wedge A_1 \Rightarrow Inv'$
$\langle 6 \rangle 2.$ $Inv \wedge A_2 \Rightarrow Inv'$
...
$\langle 6 \rangle 3.$ $Inv \wedge A_k \Rightarrow Inv'$
$\langle 6 \rangle 4.$ $Inv \wedge v' = v \Rightarrow Inv'$

Since *Inv* only includes variables in v, the last step is trivial.

Similarly for the other obligation:
$\langle 5 \rangle 1.$ $[N_1 \wedge Inv \wedge Inv']_v \Rightarrow [\overline{N_2}]_{\overline{w}}$

But don't forget that maybe the hardest part of all is picking the invariant *Inv*!

7 Liveness

What about the *Liveness* properties? We need to prove

$$\overline{WF_{buffer}(pop)}$$

given

$$WF_{Buffer}(pop) \wedge WF_{Buffer}(\exists\, k \,:\, move(k))$$

and the initial condition and safety property of *ConcreteBuffer*. Hard to do, since we don't yet know what it means :-) Intuitively, $\overline{WF_{buffer}(pop)}$ means

- When *pop can* happen, it must – sometime
- i.e., when *pop* is continually enabled (*can* occur from the current state)
- it must happen sometime (in the present or future)

Let's look at this intuitively with an example.

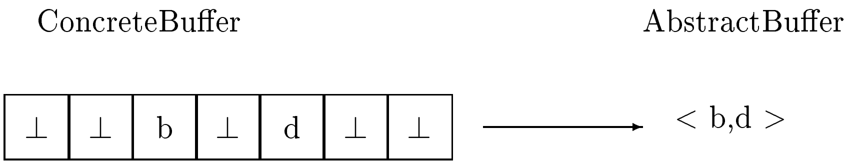


Figure 23: Are the *pop*'s Enabled?

7.1 The Predicate *Enabled*

Consider Figure 23. Is $\overline{AB.pop}$ enabled in this Figure? Is $CB.pop$ enabled?

Intuitively, $AB.pop$ is enabled when *buffer* is nonempty. So *Buffer* is also nonempty, but there is nothing in $\mathit{Buffer}[1]$, so $CB.pop$ is not enabled. So we have the situation that $AB.pop$ may be enabled, even though $CB.pop$ is not. The formal definition of $WF_v(A)$ is

$$\Box\Diamond\neg \mathit{Enabled}\langle A\rangle_v \vee \Box\Diamond\langle A\rangle_v$$

that is, using the definitions of $[A]_v$, and $\langle A\rangle_v \;\triangleq\; (A \wedge v' \neq v)$,

$$\Box\Diamond\neg \mathit{Enabled}(A \wedge v' \neq v) \vee \Box\Diamond(A \wedge v' \neq v)$$

What's the formal definition of *Enabled*? Suppose I have an action such as

$$\begin{array}{lll} \mathit{pop} & \triangleq & \wedge\ \mathit{Len}(\mathit{buffer}) > 0 \\ & & \wedge\ \mathit{buffer}' = \mathit{tail}(\mathit{buffer}) \end{array}$$

It has variable *buffer*, and next value buffer'. If I say *there is* a next value, that means I *can* do *pop*, in principle. That's exactly what *Enabled* says:

$$\begin{array}{l} \mathit{Enabled}(\mathit{pop}) \;\triangleq \\ \qquad \exists\, \mathit{newvalue} \,:\, \wedge\ \mathit{Len}(\mathit{buffer}) > 0 \\ \qquad\qquad\qquad\qquad\;\; \wedge\ \mathit{newvalue} = \mathit{tail}(\mathit{buffer}) \end{array}$$

In general, let action A have flexible variables y, z.

$$\mathit{Enabled}(A) \;\triangleq\; \exists\, \mathit{newy}, \mathit{newz} \,:\, A(y'/\mathit{newy}, z'/\mathit{newz})$$

where

- *newy* and *newz* are new symbols not otherwise occurring in A;

- $A(y'/newy, z'/newz)$ is A, but with $newy$ substituted for every occurrence of y', similarly for $newz$

So,

$$
\begin{array}{l}
Enabled(pop) \triangleq \\
\quad \exists\, newvalue : \land\ Len(buffer) > 0 \\
\quad\quad\quad\quad\quad\quad\ \land\ newvalue = tail(buffer) \\
\quad\quad\quad\quad\quad \Leftrightarrow \\
\quad \land\ Len(buffer) > 0 \\
\quad \land\ \exists\, newvalue : newvalue = tail(buffer)
\end{array}
$$

But $\exists\, newvalue : newvalue = tail(buffer)$ is a theorem of predicate logic with equality. Why? Let $\phi(x) \triangleq (x = tail(buffer))$. Then $\phi(tail(buffer))$ is an instance of an equality axiom. We may now invoke the rule (crudely written) $\dfrac{\phi(t)}{\exists x : \phi(x)}$ to infer $\exists\, newvalue : \phi(newvalue)$. So it follows that

$$
\begin{array}{l}
\land\ Len(buffer) > 0 \\
\land\ \exists\, newvalue : newvalue = tail(buffer)
\end{array}
$$

$$\Leftrightarrow$$

$$
\begin{array}{l}
\land\ Len(buffer) > 0 \\
\land\ \text{TRUE}
\end{array}
$$

$$\Leftrightarrow$$

$$Len(buffer) > 0$$

Similarly,

$$Enabled(CB.pop \land Buffer' \neq Buffer)$$

$$\Leftrightarrow$$

$$
\begin{array}{l}
\exists\, Newbuf : \land\ Buffer[1] \neq \bot \\
\quad\quad\quad\quad\quad \land\ Newbuf[1] = \bot \\
\quad\quad\quad\quad\quad \land\ \forall\, i \in 2..N : Newbuf[i] = Buffer[i]
\end{array}
$$

$$\Leftrightarrow$$

$$
\begin{array}{l}
\land\ Buffer[1] \neq \bot \\
\land\ \exists\, Newbuf : \land\ Newbuf[1] = \bot \\
\quad\quad\quad\quad\quad\quad \land\ \forall\, i \in 2..N : Newbuf[i] = Buffer[i]
\end{array}
$$

The second conjunct is, however, a theorem of ZF:

$$\exists\, \mathit{Newbuf} \;:\; \begin{array}{l} \land\ \mathit{Newbuf}[1] = \bot \\ \land\ \forall\, i \in 2..N \;:\; \mathit{Newbuf}[i] = \mathit{Buffer}[i] \end{array}$$

To show it, define

$$\mathit{TheRealNewBuf} \;\triangleq\; [\mathit{Buffer}\ \text{EXCEPT}\ ![1] = \bot]$$

Then

$$\begin{array}{l} \land\ \mathit{TheRealNewBuf}[1] = \bot \\ \land\ \forall\, i \in 2..N \;:\; \mathit{TheRealNewBuf}[i] = \mathit{Buffer}[i] \end{array}$$

follows immediately from the definition. So it follows finally that

$$\mathit{Enabled}\langle\, \mathit{CB.pop}\,\rangle_{\mathit{Buffer}} \Leftrightarrow (\mathit{Buffer}[1] \neq \bot)$$

As an exercise, try it with $\exists\, k \;:\; \mathit{CB.move}(k)$. It's very similar to $\mathit{CB.pop}$.

7.2 Proof Rules for Liveness

We need to introduce some more rules: the *LATTICE* rule and the *Weak Fairness* rules. (We shall not concern ourselves here with the *Strong Fairness* rules.)

7.2.1 *LATTICE*

LATTICE embodies *structural induction* (SI). SI is a data-structure induction rule. It is used in the theory of programming for showing that recursive programs terminate. It is to be distinguished from the rule TLA1, which describes *semantical induction* over the structure of a behavior. In contrast, *LATTICE* is mathematical rather than temporal-logical. *LATTICE* is the embodiment of structural induction.

LATTICE.
Given $\succ$ a well−founded partial order on a set S
and c a variable not free in F or G

$$\frac{\begin{array}{l} F \land (\mathsf{c} \in \mathsf{S}) \;\Rightarrow \\ \quad (H_{\mathsf{c}} \leadsto (G \lor \exists \mathsf{d} \in \mathsf{S} \;:\; (\mathsf{c} \succ \mathsf{d}) \land H_{\mathsf{d}})) \end{array}}{F \;\Rightarrow\; ((\exists \mathsf{c} \in \mathsf{S} \;:\; H_{\mathsf{c}}) \leadsto G)}$$

Here, H_{c} indicates a formula H which contains the variable c free, and H_{d} denotes $H[\mathsf{c}/\mathsf{d}]$. Assume that no occurrence of c occurs within the scope of a d-quantifier in H! The notation $A \leadsto B$ is defined as $\Box(A \Rightarrow \Diamond B)$. The intuitive meaning of $\leadsto$= leads to is that whenever A happens in the future, at some later time B must happen.

So what's a *well-founded partial order*? First, an *order* is a bianry relation with certain properties: the numbers are *ordered* by $<$, but this order is *total*: namely, all numbers are *comparable.* Sets are ordered by $\subset$, and this order is *partial*: namely, not every pair of sets is comparable. Sequences are ordered by *prefix*, and it's also partial.

An order is **well-founded** if it satisfies the following condition: if you follow a 'descending chain' of elements: $x_1 > x_2 > x_3 >$ ....., it always stops. For example, try it with natural numbers, say $x_1 = 510$. It always stops: the order on natural numbers is well-founded. So is the order on sequences. Try it: say $x_1 = \langle 5, 7, 24, 610, 7, 5, 4, 2, 3, 2 \rangle$.

Lattice is used for *fairness* proofs as a way of ensuring that *recursion stops* (always provided you've written the recursion properly). Think of G as the *ground clause* and H_c as the recursive call with parameter c

7.2.2 Weak Fairness

The **Weak Fairness Rules** are:

WF1.

$$\frac{\begin{array}{l} P \wedge [\mathcal{N}]_f \;\Rightarrow\; (P' \vee Q') \\ P \wedge \langle \mathcal{N} \wedge \mathcal{A} \rangle_f \;\Rightarrow\; Q' \\ P \;\Rightarrow\; \mathit{Enabled}\ \langle \mathcal{A} \rangle_f \end{array}}{\Box[\mathcal{N}]_f \wedge \mathrm{WF}_f(\mathcal{A}) \;\Rightarrow\; (P \leadsto Q)}$$

WF2.

$$\frac{\begin{array}{l} \langle \mathcal{N} \wedge \mathcal{B} \rangle_f \;\Rightarrow\; \langle \overline{\mathcal{M}} \rangle_{\overline{g}} \\ P \wedge P' \wedge \langle \mathcal{N} \wedge \mathcal{A} \rangle_f \wedge \overline{\mathit{Enabled}\ \langle \mathcal{M} \rangle_g} \;\Rightarrow\; \mathcal{B} \\ P \wedge \overline{\mathit{Enabled}\ \langle \mathcal{M} \rangle_g} \;\Rightarrow\; \mathit{Enabled}\ \langle \mathcal{A} \rangle_f \\ \Box[\mathcal{N} \wedge \neg\mathcal{B}]_f \wedge \mathrm{WF}_f(\mathcal{A}) \wedge \Box F \\ \qquad \wedge\, \Diamond\Box\overline{\mathit{Enabled}\ \langle \mathcal{M} \rangle_g} \;\Rightarrow\; \Diamond\Box P \end{array}}{\Box[\mathcal{N}]_f \wedge \mathrm{WF}_f(\mathcal{A}) \wedge \Box F \;\Rightarrow\; \overline{\mathrm{WF}_g(\mathcal{M})}}$$

where
F, G, H_{c} are TLA formulas, P, Q, I are predicates, $\mathcal{A}$, $\mathcal{B}$, $\mathcal{N}$, $\mathcal{M}$ are actions, f, g are state functions.

While WF1 is fairly straightforward, WF2 embodies a form of proof by contradiction. Let's try to clarify this intuitively, premise by premise. The first premise,

$$\langle \mathcal{N} \wedge \mathcal{B} \rangle_f \;\Rightarrow\; \langle \overline{\mathcal{M}} \rangle_{\overline{g}}$$

says that when an $\mathcal{N}$ action occurs that's also a $\mathcal{B}$ action, an $\mathcal{M}$ action is simulated. The second premise

$$P \wedge P' \wedge \langle \mathcal{N} \wedge \mathcal{A} \rangle_f \wedge \overline{\mathit{Enabled}\ \langle \mathcal{M} \rangle_g} \;\Rightarrow\; \mathcal{B}$$

says that, supposing an $\mathcal{N}$-and-$\mathcal{A}$ action occurs when $\mathcal{M}$ is enabled and P is invariant, then that's also a $\mathcal{B}$ action (so by the first premise, that'll also be an $\mathcal{M}$ action). The third premise,

$$P \wedge \overline{\mathit{Enabled}\ \langle \mathcal{M} \rangle_g} \;\Rightarrow\; \mathit{Enabled}\ \langle \mathcal{A} \rangle_f$$

ensures that the antecedent of Premise 2 is not obviously self-contradictory: if P holds and $\mathcal{M}$ *is* enabled, $\mathcal{A}$ is also enabled. Finally the fourth premise:

$$\Box[\mathcal{N} \wedge \neg\mathcal{B}]_f \wedge \mathrm{WF}_f(\mathcal{A}) \wedge \Box F \wedge \Diamond\Box\overline{\mathit{Enabled}\ \langle \mathcal{M} \rangle_g}$$

$$\Rightarrow\; \Diamond\Box P$$

says that if a $\mathcal{B}$ action *never* occurs, and $\mathcal{A}$ satisfies a weak fairness condition, and eventually $\mathcal{M}$ becomes enabled and stays enabled, then eventually P becomes true and stays true. Recall that by Premise 3, in such a state $\mathcal{A}$ is also enabled; therefore also eventually continuously enabled in this situation.

That condition makes in the future most of the conjuncts in Premise 2 true. So if an $\mathcal{A}$ action ever occurs in this situation, it will be an $\mathcal{M}$ action also. $\mathcal{M}$ will then have occurred after being continuously enabled, which is what $\overline{\mathrm{WF}_g(\mathcal{M})}$ requires. Which is what we want to prove.

But how do we know an $\mathcal{A}$ action will ever occur in this circumstance? By assuming it is weakly fair - since it is continuously enabled, we required it to happen sometime. And its happening will also be an $\mathcal{M}$-simulation that we want to show happens. We are done. We have argued intuitively that

$$\Box[\mathcal{N}]_f \wedge \mathrm{WF}_f(\mathcal{A}) \wedge \Box F \;\Rightarrow\; \overline{\mathrm{WF}_g(\mathcal{M})}$$

is true providing the four premises are temporal-logical truths.

Of course, one should take care to distinguish informal argument – informal justification of this proof rule – from temporal-logical validity. There are some lines of '*intuitive*' reasoning that just aren't valid (for one thing, the Deduction Theorem doesn't hold). Notwithstanding, the reasoning above is a reasonable guide to the validity of WF2.

Why is it a proof by contradiction? Because there's a conjunct in the antecedent of the fourth premise that asserts that $\mathcal{B}$ never happens, but nevertheless we can show an $\mathcal{A}$ must happen under P and $\overline{\mathit{Enabled}\ \langle\mathcal{M}\rangle_g}$, and in these circumstances, by Premise 2 this must be a $\mathcal{B}$ action. So we derive the occurrence of a $\mathcal{B}$ action by assuming it never can happen. If this presumption is wrong (which it must be - since it's contradictory given the other premises) then every behavior contains an unending chain of $\mathcal{B}$ actions and Premise 1 tells us we have an unending chain of $\mathcal{M}$ actions, which is one disjunct of $\overline{\mathrm{WF}_g(\mathcal{M})}$.

According to the fourth premise, if $\mathcal{B}$ never occurs, and $\mathcal{M}$ becomes and remains enabled, then P becomes and remains true. The second premise (it must be a temporal-logical truth as it occurs as an assertion in a TLA proof) must be true in all states, and therefore in none of these future states in which $\mathcal{M}$ remains enabled and P remains true can any $\mathcal{N}$ action that is $\mathcal{A}$ ever occur, since else $\mathcal{B}$ occurs, contrary to supposition. And $\mathcal{A}$ is, in this situation, also possible, since it is enabled by Premise 3.

7.3 The Liveness Proof

Let's look now at the buffers. First, I introduce some auxiliary definitions (as well as a reminder of two that we've seen before):

$$
\begin{array}{ll}
\textsc{Let:} & \mathit{NonVoid}(k) \triangleq k \neq \bot \\
& \overline{\mathit{buffer}} \triangleq \mathit{SelectSeq}(\mathit{Buffer}, \mathit{NonVoid}) \\
& \mathit{FirstFull} \triangleq \mathit{Buffer}[1] \neq \bot \\
& \mathit{NotEmpty} \triangleq \exists\, i \in 1..N : \mathit{Buffer}[i] \neq \bot \\
& \mathit{NotStuffed} \triangleq \exists\, i \in 1..(N-1) : \mathit{Buffer}[i] = \bot \\
& \mathit{Descending}(\mathit{Buffer}) \triangleq \sum_{i=1}^{N} (\mathit{Buffer}[i] = \bot).2^{N-i} \\
& \mathit{MaxDescending} \triangleq \sum_{i=1}^{N} 2^{N-i}
\end{array}
$$

We want to prove (I'll cheat a bit on the numbering by performing a logical simplification without changing the proof step number):

$$
\begin{array}{l}
\langle 2 \rangle 3.\ \wedge \ \Box[\vee \ \exists\, a \ :\ CB.push(a) \\
\qquad\qquad \vee \ CB.pop \\
\qquad\qquad \vee \ \exists\, k \ :\ CB.move(k)]_{Buffer}] \\
\qquad \wedge \ WF_{Buffer}(CB.pop) \\
\qquad \wedge \ WF_{Buffer}(\exists\, k \ :\ CB.move(k)) \\
\qquad \Rightarrow \\
\qquad \overline{WF_{buffer}(AB.pop)}
\end{array}
$$

so we take

$$
\begin{array}{lcl}
\mathcal{N} & \triangleq & (\vee \ \exists\, a \ :\ CB.push(a) \\
 & & \ \vee \ CB.pop \\
 & & \ \vee \ \exists\, k \ :\ CB.move(k)) \\
f & \triangleq & Buffer \\
\mathcal{A} & \triangleq & \mathcal{B} \ \triangleq \ CB.pop \\
\mathcal{M} & \triangleq & AB.pop \\
g & \triangleq & buffer \\
P & \triangleq & FirstFull \\
\Box F & \triangleq & WF_{Buffer}(\exists\, k \ :\ CB.move(k))
\end{array}
$$

and instantiate the conclusion of WF2. The substeps are the premises:

$$
\begin{array}{l}
\langle 2 \rangle 3.\ \wedge \ \Box[\vee \ \exists\, a \ :\ CB.push(a) \\
\qquad\qquad \vee \ CB.pop \\
\qquad\qquad \vee \ \exists\, k \ :\ CB.move(k)]_{Buffer}] \\
\qquad \wedge \ WF_{Buffer}(CB.pop) \\
\qquad \wedge \ WF_{Buffer}(\exists\, k \ :\ CB.move(k)) \\
\qquad \Rightarrow \\
\qquad \overline{WF_{buffer}(AB.pop)}
\end{array}
$$

PROOF:

$$
\begin{array}{l}
\langle 3 \rangle 1.\ \langle \wedge \ \vee \ \exists\, a \ :\ CB.push(a) \\
\qquad\qquad \vee \ CB.pop \\
\qquad\qquad \vee \ \exists\, k \ :\ CB.move(k)) \\
\qquad\quad \wedge \ CB.pop \rangle_{Buffer} \\
\qquad\quad \Rightarrow \\
\qquad\quad \langle \overline{AB.pop} \rangle_{\overline{buffer}}
\end{array}
$$

$\langle 3 \rangle 2.$ $\wedge\ \mathit{FirstFull}$
$\wedge\ \mathit{FirstFull}'$
$\wedge\ \langle\, CB.pop\, \rangle_{Buffer}$
$\wedge\ \overline{\mathit{Enabled}\langle\, AB.pop\, \rangle_{buffer}}$
$\Rightarrow$
$CB.pop$

$\langle 3 \rangle 3.$ $\wedge\ \mathit{FirstFull}$
$\wedge\ \overline{\mathit{Enabled}\langle\, AB.pop\, \rangle_{buffer}}$
$\Rightarrow$
$\mathit{Enabled}\langle\, CB.pop\, \rangle_{Buffer}$

$\langle 3 \rangle 4.$ $\wedge\ \Box[\wedge \vee \exists\, a\ :\ CB.push(a)$
$\vee\ CB.pop$
$\vee\ \exists\, k\ :\ CB.move(k)$
$\wedge\ \neg CB.pop]_{Buffer}$
$\wedge\ WF_{Buffer}(CB.pop)$
$\wedge\ WF_{Buffer}(\exists\, k\ :\ CB.move(k))$
$\wedge\ \Diamond\Box\overline{\mathit{Enabled}\langle\, AB.pop\, \rangle_{buffer}}$
$\Rightarrow$
$\Diamond\Box \mathit{FirstFull}$

$\langle 3 \rangle 5.$ Q.E.D.
PROOF: Follows from $\langle 3 \rangle 1$, $\langle 3 \rangle 2$, $\langle 3 \rangle 3$ and $\langle 3 \rangle 4$ using TLA Proof Rule WF2 with the substitutions given above. □

We take these one by one.

$\langle 3 \rangle 1.$ $\langle \wedge \vee \exists\, a\ :\ CB.push(a)$
$\vee\ CB.pop$
$\vee\ \exists\, k\ :\ CB.move(k)$
$\wedge\ CB.pop\,\rangle_{Buffer}$
$\Rightarrow \langle \overline{AB.pop} \rangle_{\overline{buffer}}$
PROOF:

$\langle 4 \rangle 1.$ $\wedge \vee \exists\, a\ :\ CB.push(a)$
$\vee\ CB.pop$
$\vee\ \exists\, k\ :\ CB.move(k))$
$\wedge\ CB.pop$
$\equiv$

$$CB.pop$$

$$\langle 4 \rangle 2.\ \langle\, CB.pop \,\rangle_{Buffer} \Rightarrow \langle\, \overline{AB.pop} \,\rangle_{\overline{buffer}}$$

Both of these substeps are pretty trivial. We won't go further into them. Now for the next step.

$$\begin{aligned}\langle 3 \rangle 2.\ & \land\ FirstFull \\ & \land\ FirstFull' \\ & \land\ \langle\, CB.pop \,\rangle_{Buffer} \\ & \land\ \overline{Enabled \langle\, AB.pop \,\rangle_{buffer}} \\ & \Rightarrow\ CB.pop\end{aligned}$$

This is also logically trivial - just look at it!

The third step, with its substeps is:

$$\begin{aligned}\langle 3 \rangle 3.\ & \land\ FirstFull \\ & \land\ \overline{Enabled \langle\, AB.pop \,\rangle_{buffer}} \\ & \Rightarrow\ Enabled \langle\, CB.pop \,\rangle_{Buffer}\end{aligned}$$

$$\langle 4 \rangle 1.\ FirstFull \Rightarrow Enabled \langle\, CB.pop \,\rangle_{Buffer}$$

This substep is also trivial. Finally, the rather horrendous fourth premise, with a substep that is a logically-simplified version, substituting the predicate *NotEmpty* for the *Enabled* condition in the antecedent:

$$\begin{aligned}\langle 3 \rangle 4.\ & \land\ \Box[\land \lor \exists\, a\ :\ CB.push(a) \\ & \qquad\quad \lor\ CB.pop \\ & \qquad\quad \lor\ \exists\, k\ :\ CB.move(k) \\ & \qquad \land\ \neg CB.pop]_{Buffer} \\ & \land\ WF_{Buffer}(CB.pop) \\ & \land\ WF_{Buffer}(\exists\, k\ :\ CB.move(k)) \\ & \land\ \Diamond\Box\overline{Enabled \langle\, AB.pop \,\rangle_{buffer}} \\ & \Rightarrow\ \Diamond\Box FirstFull\end{aligned}$$

PROOF:

$$\begin{array}{l}\langle 4\rangle 1.\ \wedge \Box[\vee \exists\, a : CB.push(a) \\ \qquad\qquad \vee \exists\, k : CB.move(k)]_{Buffer} \\ \qquad \wedge\ WF_{Buffer}(CB.pop) \\ \qquad \wedge\ WF_{Buffer}(\exists\, k : CB.move(k)) \\ \qquad \wedge\ \Diamond\Box NotEmpty \\ \qquad \Rightarrow \Diamond\Box FirstFull\end{array}$$

Actually, rather than show this substep, we show this further substep (we'll Bridge the gap below):

$$\begin{array}{l}\langle 5\rangle 2.\ \wedge \Box[\exists\, a : CB.push(a).....]_{Buffer} \\ \qquad \wedge\ WF_{Buffer}(CB.pop) \\ \qquad \wedge\ WF_{Buffer}(\exists\, k : CB.move(k)) \\ \qquad \Rightarrow \Box(NotEmpty \Rightarrow \Diamond FirstFull)\end{array}$$

which because of

$$\begin{array}{l}\langle 6\rangle 1.\ (NotEmpty \wedge NotStuffed \Rightarrow \Diamond FirstFull) \\ \qquad \equiv (NotEmpty \Rightarrow \Diamond FirstFull)\end{array}$$

is equivalent to

$$\begin{array}{l}\langle 6\rangle 2.\ \wedge \Box[\exists\, a : CB.push(a)......]_{Buffer} \\ \qquad \wedge\ WF_{Buffer}(CB.pop) \\ \qquad \wedge\ WF_{Buffer}(\exists\, k : CB.move(k)) \\ \qquad \Rightarrow \Box(\wedge\ NotEmpty \\ \qquad\qquad \wedge\ NotStuffed \\ \qquad\qquad \Rightarrow \Diamond FirstFull)\end{array}$$

which is shown using the Lattice Rule We have somehow to get $\langle 6\rangle 2$ into a form in which we can apply *LATTICE*. First, we fiddle with the conclusion:

$$\begin{array}{l}\langle 7\rangle 1.\ \wedge \Box[\exists\, a : CB.push(a)......... \\ \qquad \wedge\ WF_{Buffer}(CB.pop) \\ \qquad \wedge\ WF_{Buffer}(\exists\, k : CB.move(k)) \\ \qquad \Rightarrow\end{array}$$

$$
\begin{array}{l}
\Box(\land\ \exists\, i \in 0..\mathit{MaxDescending}\ : \\
\qquad\qquad \mathit{Descending}(\mathit{Buffer}) = i \\
\quad\ \land\ \mathit{NotEmpty} \\
\quad\ \land\ \mathit{NotStuffed} \\
\qquad \Rightarrow \\
\qquad \Diamond \mathit{FirstFull})
\end{array}
$$

Then we use the Lattice Rule to derive this. If this is to be the conclusion of *LATTICE*, the substep must be the premise:

$$
\begin{array}{ll}
\langle 8\rangle 1. & \land\ \Box[\lor\ \exists\, a\ :\ \mathit{CB.push}(a) \\
& \qquad\ \lor\ \exists\, k\ :\ \mathit{CB.move}(k)]_{\mathit{Buffer}} \\
& \land\ \mathit{WF}_{\mathit{Buffer}}(\mathit{CB.pop}) \\
& \land\ \mathit{WF}_{\mathit{Buffer}}(\exists\, k\ :\ \mathit{CB.move}(k)) \\
& \land\ i \in 0..\mathit{MaxDescending} \\
& \quad \Rightarrow \\
& \quad \Box(\land\ \mathit{NotEmpty} \\
& \qquad\ \land\ \mathit{NotStuffed} \\
& \qquad\ \land\ \mathit{Descending}(\mathit{Buffer}) = i) \\
& \qquad\quad \Rightarrow \\
& \qquad\quad \Diamond(\lor\ \mathit{FirstFull} \\
& \qquad\qquad\ \lor\ \exists\, j \in 0..\mathit{MaxDescending}\ : \\
& \qquad\qquad\qquad \land\ j < i \\
& \qquad\qquad\qquad \land\ \mathit{NotEmpty} \\
& \qquad\qquad\qquad \land\ \mathit{NotStuffed} \\
& \qquad\qquad\qquad \land\ \mathit{Descending}(\mathit{Buffer}) = j)
\end{array}
$$

This step itself is justified by an application of WF1 (and simple propositional logic). It is taken as the conclusion of an application of WF1 with the instantiations

$$
\begin{array}{lcl}
\mathcal{N} & \triangleq & \exists\, a\ :\ \mathit{CB.push}(a) \lor \exists\, k\ :\ \mathit{CB.move}(k), \\
\mathcal{A} & \triangleq & \exists\, k\ :\ \mathit{CB.move}(k), \\
P & \triangleq & \land\ \mathit{NotEmpty} \\
& & \land\ \mathit{NotStuffed} \\
& & \land\ \mathit{Descending}(\mathit{Buffer}) = i
\end{array}
$$

and

$$
\begin{array}{lll}
Q & \triangleq & \lor\ FirstFull \\
 & & \lor\ \exists\, j \in 0..MaxDescending\ : \\
 & & \qquad \land\ j < i \\
 & & \qquad \land\ NotEmpty \\
 & & \qquad \land\ NotStuffed \\
 & & \qquad \land\ Descending(Buffer) = j)
\end{array}
$$

The substeps consist of the premises of WF1. I leave this to the reader (See [Lad96a] to cheat!... I mean, to steal!)

7.4 Using The Bridge Rule

The question remaining is to Bridge the gap we left earlier. How do we obtain

$$
\begin{array}{ll}
\langle 4\rangle 4. & \land\ \Box[\lor\ \exists\, a\ :\ CB.push(a) \\
 & \qquad \lor\ \exists\, k\ :\ CB.move(k)]_{Buffer} \\
 & \land\ WF_{Buffer}(CB.pop) \\
 & \land\ WF_{Buffer}(\exists\, k\ :\ CB.move(k)) \\
 & \land\ \Diamond\Box NotEmpty \\
 & \Rightarrow\ \Diamond\Box FirstFull
\end{array}
$$

from

$$
\begin{array}{ll}
\langle 5\rangle 2. & \land\ \Box[\exists\, a\ :\ CB.push(a).....]_{Buffer} \\
 & \land\ WF_{Buffer}(CB.pop) \\
 & \land\ WF_{Buffer}(\exists\, k\ :\ CB.move(k)) \\
 & \Rightarrow\ \Box(NotEmpty \Rightarrow \Diamond FirstFull)
\end{array}
$$

?????

Answer: by the Bridge Rule [Lad96a] (you were presumably waiting for that!).

Let X, Y, Z be propositional variables (that is, symbols standing for any sentences). Then the Bridge Rule is the derived rule

$$
\frac{\begin{array}{c}\Box X \land Y \Rightarrow \Box Y \\ \Box X \Rightarrow \Box(Z \Rightarrow \Diamond Y)\end{array}}{\Box X \land \Diamond\Box Z \Rightarrow \Diamond\Box Y}
$$

To apply the Bridge Rule, we'll use the substitutions

$$\begin{array}{lcl}\Box X & \triangleq & \wedge \Box[\mathcal{N} \wedge \neg CB.pop]_{Buffer} \\ & & \wedge WF_{Buffer}(CB.pop) \\ & & \wedge WF_{Buffer}(\exists\, k \,:\, CB.move(k)) \\ Y & \triangleq & FirstFull \\ Z & \triangleq & NonEmpty \end{array}$$

(I really mean: the conjunction is by STL5 equivalent to a formula of the form $\Box X$).

What does the Bridge Rule mean? Suppose X is an action, which is a disjunction of actions. Then $\Box X$ is a safety property. Let Y be some state predicate. Consider the first premise

$$\Box X \wedge Y \Rightarrow \Box Y$$

If we can prove the first premise, we are showing that Y would be an invariant if $\mathcal{B}$ were never to happen. Well, in our present buffer example, that's right! So it looks as though we can prove this first premise. The second premise of the Bridge Rule is

$$\Box X \Rightarrow \Box(Z \Rightarrow \Diamond Y)$$

which is the same as (is defined to be)

$$\Box X \Rightarrow (Z \leadsto Y)$$

which is the conclusion of the Lattice Rule. The conclusion of the Bridge Rule is

$$\Box X \wedge \Diamond\Box Z \Rightarrow \Diamond\Box Y$$

which is the form of the fourth premise of WF2. Thus the Bridge Rule bridges the logical gap between the Lattice Rule and the fourth WF2 premise.

To summarise:
we prove the first premise

$$\Box X \wedge Y \Rightarrow \Box Y$$

using INV1:

$$\frac{\Box X \wedge Y \Rightarrow Y'}{\Box X \wedge Y \Rightarrow \Box Y}$$

We prove the second premise

$$\Box X \Rightarrow \Box(Z \Rightarrow \Diamond Y)$$

using the Lattice Rule. And we draw the conclusion

$$\Box X \wedge \Diamond\Box Z \Rightarrow \Diamond\Box Y$$

which is the temporal fourth premise of WF2.

So to complete our proof, we need to prove the first Bridge Rule premise:

$\langle 5\rangle 1.$ $\wedge\ \Box[\vee\ \exists\, a\ :\ CB.push(a)$
$\quad\quad\vee\ \exists\, k\ :\ CB.move(k)]_{Buffer}$
$\wedge\ WF_{Buffer}(CB.pop)$
$\wedge\ WF_{Buffer}(\exists\, k\ :\ CB.move(k))$
$\wedge\ FirstFull$
$\Rightarrow$
$\Box FirstFull$

Proof:

$\langle 6\rangle 1.$ $\wedge\ \Box[\vee\ \exists\, a\ :\ CB.push(a)$
$\quad\quad\vee\ \exists\, k\ :\ CB.move(k)]_{Buffer}$
$\wedge\ FirstFull$
$\Rightarrow$
$\Box FirstFull$

$\langle 6\rangle 2.$ Q.E.D.

Proof: Immediate from $\langle 6\rangle 1$ by adding a hypothesis. □

$\langle 6\rangle 1.$ $\wedge\ \Box[\vee\ \exists\, a\ :\ CB.push(a)$
$\quad\quad\vee\ \exists\, k\ :\ CB.move(k)]_{Buffer}$
$\wedge\ FirstFull$
$\Rightarrow$
$\Box FirstFull$

Proof:

$\langle 7\rangle 1.$ $\wedge\ [\vee\ \exists\, a\ :\ CB.push(a)$
$\quad\quad\vee\ \exists\, k\ :\ CB.move(k)]_{Buffer}$
$\wedge\ FirstFull$
$\Rightarrow$
$FirstFull'$

Proof: Follows from the definitions of *push* and *move*. □

$\langle 7\rangle 2.$ Q.E.D.

PROOF: Immediate by TLA Proof Rule INV1 from $\langle 7 \rangle 1$, with the antecedent conjuncts in the other order. $\Box$

That shows the first BR premise and so we're **Done**, not only with the BR premise but with the entire proof.

8 Conclusions

I have tried to motivate some of the features of TLA first of all by considering change, and a vocabulary needed to talk reasonably in a logical language about change. I then considered what kind of proof rules we would need in this language: the 'standard' natural deduction rules as given, for example, by Prawitz, also for modal logic; an extra rule (axiom) for Simple Temporal Logic; two rules for dealing with the *prime* operator; and induction rules (called *Invariant* rules) which arise from the structure of behaviors (they are ω-sequences of states).

I showed how these rules suffice to follow the invariant assertion method for safety proofs. I worked through an example, a buffer implementation, in detail, and discussed a proof scheme and the resulting proof obligations. No proof obligations have temporal operators: ergo pure math (that is, first order set theory) suffices.

We came then to liveness. Liveness is important for TLA, because TLA specifications are *stuttering invariant* (stuttering invariance is one of many important concepts that I deliberately overlooked in this tutorial). Because a safety assertion is of the form $\Box[\mathcal{N}]_v$, a safety assertion conjoined with an initial condition cannot imply that anything ever changes - a possible behavior is that variable v never changes its value from the initial condition, ever. Thus, to ensure that deep down in the forest something stirs, we need to add liveness assertions to every machine specification.

Lamport has shown that the TLA rules as presented here (with the exception of Strong Liveness) suffice to prove any verification in which both machines are specified by *TLA specifications.* A TLA specification has the form

$$Init \wedge \Box[\mathcal{N}]_v \wedge Liveness$$

in which *Init* is a state predicate, $\mathcal{N}$ is an action formula containing only variables occurring in v (v might be a term, rather than a simple variable

– in fact, usually is), and *Liveness* is a Boolean combination of weak and strong fairness conditions.

The point of this tutorial was to do all and only that which is necessary to follow a particular TLA verification – that of the buffer implementation. This has entailed that many fundamental TLA concepts just have not arisen. The style has been discursive - the notes have been constructed from the slides for the tutorial itself. For a full hierarchical proof of the buffer implementation, see [Lad96a]. Another example, with a little more *cheat*ing on the low-level details, is [Lad96b]. And for lots and lots of information about TLA itself, including the definition of TLA+; real-time additions to TLA; use of TLA in specifying and verifying hybrid (part physical) systems; theorems on refinement mappings; difficult verifications; and links to other work around the world, see [Lama].

References

[AL91] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.

[Ash75] Edward A. Ashcroft. Proving assertions about parallel programs. *Journal of Computer and System Sciences*, 10:110–135, February 1975.

[BF93] Ricky W. Butler and George B. Finelli. The infeasibility of quantifying the reliability of life-critical real-time software. *IEEE Transactions on Software Engineering*, 19(1):3–12, January 1993.

[Bra97] F. H. Bradley. *Appearance and Reality.* Macmillan, New York, 2nd edition, 1897.

[Eng95] Urban Engberg. TLP. `http://www.daimi.aau.dk/~urban/tlp/tlp.html`, 1995.

[Gol92] Robert Goldblatt. *Logics of Time and Computation*, volume 7 of *Lecture Notes.* CSLI Press, Stanford, CA, USA, 2nd revised and expanded edition, 1992.

[Kal] Sara Kalvala. A formulation of TLA in Isabelle. Available through [Lama].

[Lad96a] Peter B. Ladkin. Formal but lively buffers in TLA+. Technical Report RVS-RR-96-07, RVS Group, Universität Bielefeld, Technische Fakultät, January 1996. Available through `http://www.rvs.uni-bielefeld.de`.

[Lad96b] Peter B. Ladkin. Lazy Cache implements Complete Cache. Technical Report RVS-RR-96-06, RVS Group, Universität Bielefeld, Technische Fakultät, January 1996. Available through `http://www.rvs.uni-bielefeld.de`.

[Lama] Leslie Lamport. The Temporal Logic of Actions page. `http://www.research.digital.com/SRC/tla`.

[Lamb] Leslie Lamport. Various TLA+ documents. In [Lama].

[Lam93] Leslie Lamport. How to write a proof, January 1993. In `http://www.research.digital.com/SRC/tla/`.

[Lam94a] Leslie Lamport. How to write a long formula. *Formal Aspects of Computing*, 6:580–584, 1994.

[Lam94b] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[Lam95] Leslie Lamport. How to write a proof. *American Mathematical Monthly*, 102(7):600–608, August-September 1995.

[Lam96] Leslie Lamport. personal communication. 1996.

[LS84] S. S. Lam and A. U. Shankar. Protocol verification via projections. *IEEE Transactions on Software Engineering*, 10(4):325–342, July 1984.

[LS93] Bev Littlewood and Lorenzo Strigini. Validation of ultrahigh dependability for software-based systems. *Communications of the ACM*, 36(11):69–80, November 1993.

[McT27] J. M. E. McTaggart. *The Nature of Existence*, chapter 33. Cambridge University Press, Cambridge, 1927. Also Chapter 1 of [PM93].

[Mer97] Stephan Merz. Isabelle/TLA. `http://www4.informatik.tu-muenchen.de/~merz/isabelle/`, 1997.

[MP92] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*, volume 1: Specification. Springer-Verlag, 1992.

[Neu] Peter G. Neumann. Past Security Projects page. `http://www.csl.sri.com/sri-csl-security/past-projects.html`.

[OG76] Susan Owicki and David Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6(4):319–340, 1976.

[PM93] Robin Le Poidevin and Murray MacBeath, editors. *The Philosophy of Time.* Oxford Readings in Philosophy. Oxford University Press, Oxford, 1993.

[Pra65] Dag Prawitz. *Natural Deduction: A Proof-Theoretical Study.* Almqvist and Wiksell, Uppsala, 1965.

[Req92] Requirements and Technical Concepts for Aviation (RTCA). Software considerations in airborne systems and equipment certification. RTCA Paper No. DO-178-B, 7th Draft, July 1992.

[Res95] Nicholas Rescher. Being and becoming. In Jaegwon Kim and Ernest Sosa, editors, *A Companion To Metaphysics*, Blackwell Companions to Philosophy, pages 46–47. Basil Blackwell, Oxford, 1995.

[Rus] John Rushby. Formal Methods and Dependable Systems page. `http://www.csl.sri.com/sri-csl-fm.html`.

[RvHO91] John Rushby, Friedrich von Henke, and Sam Owre. An introduction to formal specification and verification uding EHDM. Technical Report SRI-CSL-91-02, SRI Computer Science Lab, Menlo Park, CA, USA, February 1991. Abstract in [SRIb].

[SRIa] SRI Computer Science Lab. Fault Tolerance at SRI. `http://www.csl.sri.com/ft-history.html`.

[SRIb] SRI Computer Science Lab. Technical Report Abstracts. http://www.csl.sri.com/tr-abstracts.html.

[Vaa] Frits Vaandrager. Publications. http://www.cs.kun.nl/~fvaan/publications.html.

[VR92] Kim J. Vicente and Jens Rasmussen. Ecological interface design: Theoretical foundations. *IEEE Transactions on Systems, Man and Cybernetics*, 22(4):589–606, July/August 1992.