Formal But Lively Buffers in TLA$^+$

Peter B. Ladkin
Universität Bielefeld, Technische Fakultät
Postfach 10 01 31, D-33501 Bielefeld
ladkin@techfak.uni-bielefeld.de

Version of January 6, 1996

Abstract

I perform some rigorous verifications in TLA, by using simple examples which nevertheless illustrate TLA techniques, in particular liveness proofs. Since the method of invariants for safety proofs is well understood, our example needs only the trivial invariant, which is simply omitted. We specify in TLA a buffer implemented as an array, a double buffer implemented as two arrays in series, and an abstract buffer which uses a sequence. We prove, formally and rigorously, that the two 'implementations' implement the abstract buffer. The non-trivial part is the proof of liveness.

1 The Problem

When she was told Calvin Coolidge had died, Dorothy Parker asked, "How could they tell?". If your system dies, how could *you* tell? The chances are slim if you don't specify that it shall live. Nonetheless, many specification methods, including most process algebraic approaches, omit a liveness requirement. Liveness can be hard to prove, which is maybe one reason why it isn't stated. And engineers writing specifications don't always understand its importance. We illustrate how to perform rigorous verifications in TLA, emphasising the proofs of liveness. We specify some buffers in TLA$^+$. Our examples are not mathematically very interesting, because that is not our point. We chose them small, in order better to illustrate the proof techniques.

TLA$^+$ is a specification language based on Leslie Lamport's TLA, the Temporal Logic of Actions [Lam94b]. TLA$^+$ contains structuring and naming conventions (such as modules, with `import` and `include`), and some shorthand for necessary mathematics (e.g., a notation for values of functions). TLA$^+$ is to TLA rather as LaTeX is to TeX.

We define an abstract bounded buffer of length N, which contains a sequence of elements *push*ed on it, and which elements may be *pop*ped off. The sequence may contain at most N elements at one time. We don't require anything to be pushed on the buffer, but if there is something there, we want it eventually to be popped off (that's liveness). And if it's full, then it must be *pop*ped before anything more may be *push*ed on. We then define an implementation of a buffer as an array of fixed length, which may have elements pushed in the last place, and popped off the first place, and which may shuttle them forwards to get them ready for popping. Again, we don't require anything to be pushed. But if there are elements in the buffer, and if the shuttle doesn't happen sometime when it can, they can't be popped because they don't get to the front. So, to show that the concrete buffer implements the abstract buffer requires that the shuttle action is lively. A further wrinkle is that the shuttle action is 'invisible' to the abstract buffer. It can't 'see' a shuttle because shuttling alters only the absolute positions of elements in the array. The abstract buffer can only 'see' relative position.

The Fairchild F3341 chip implemented a 'bubble-through' FIFO buffer in exactly this way. It was 64 bits long by 4 bits wide. Two were used in parallel to give an 8-bit-wide transfer path. It was used by original equipment manufacturers to buffer disk transfers, notably Digital Equipment Corporation in their RP03 drives.

We also specify the concrete buffer as two smaller concrete buffers of half the size rigged in series. We want to prove that this is also a buffer. We provide a completely formal, syntactic proof that the buffers-in-series implement a larger concrete buffer, and then that the concrete buffer implements the abstract buffer specification. Since implementation is just logical implication in TLA, and implication is transitive, this shows that the double-buffer is indeed an implementation of a buffer. The specifications were written using the `tla.sty` style file of Lamport.

The hard part of the proof is the proof of liveness. Our proof is rigorously syntactically formal. That is to say, every deduction in the proof is a syntactically rigorous application of one of the TLA inference rules (with the exception of some deductions from the theory of sequences and arithmetic, and the Bridge Rule, a derivable TLA rule defined in the proof on Page 46 which is needed to tie together the TLA rules WF2 and Lattice in the liveness proof). This level of rigor is achieved by automated theorem provers, but rarely by 'hand proofs', which often employ looser standards. Rigorous proofs are harder, as can be seen by comparing our proof sketch in Section 6 with the formal TLA proof in Section 9. The proof was constructed using the `pf.sty` style file of Lamport.

Both [Mil89] and [Hoa85] treat the example of buffers, and composing larger buffers from smaller, as we do. Buffers are specified in [Mil89, pp72-73] as constructed from single-place buffers and proved correct with respect to their properties. Two buffers (or other processes with a single input and a single output channel) are constructed in [Hoa85, Section 4.4 Pipes, pp151-161]. What are the differences between those approaches and ours? Both of these approaches use process algebra in order to check mathematically that the buffer specifications/constructions are correct. However, firstly, the reasoning used for the mathematical arguments is the usual informal mathematical reasoning, and, secondly, liveness properties of the constructions may not be stated. Any liveness properties in these process algebras follow only from general liveness properties of the specification style and may not be tailored to individual problems.

When a part of the reasoning is not formal, the possibility remains that errors may be made. The purpose of purely formal reasoning is to minimise the possibility of errors in a field of computer science in which errors are both rife and notorious, by the nature of the subject. When the entire reasoning needed for the proofs in CCS and CSP is formalised, a part of it must be in formal logic. One would be in the position of having part of the reasoning in a formal logic, and part in the axiomatic theory of the process algebra.

Liveness properties are not explicit in process algebra formalisations such as these. Thus, what liveness properties there are follow from general principles of the process algebra, and cannot be tailored to the wishes of the specifier, as ours are. A specifier who requires specific liveness properties may then be constrained to use a specification and verification method in which they may be stated explicitly, such as TLA. Liveness properties may of course be stated in the temporal logic version of CCS, the Modal Mu-Calculus [Sti89].

Writing proofs is hard - rather like programming. However, the technology of programming is ahead of the technology of proving. This paper aims to contribute to the technology of proving.

2 Conclusions

In a case study, it's appropriate to put the conclusions at the front. The general lessons learned from this exercise are:

- Writing literate proofs is as important, for similar reasons, as writing literate programs, and requires non-trivial skills;
- Writing proofs is aided by derived proof rules such as Theorem 1;
- Writing syntactically correct proofs (ultimately, the only appropriate sort of proofs) by hand can be aided by good organisation, such as suggested by Lamport, but resembles trying to write programs without use of a compiler.
- A well-designed proof checker would save work and time, playing a role rather like a compiler when developing programs. But proof-formatting aids are as helpful as a proof checker. Examples would include a structured editor, and tools allowing fragments of a proof to be viewed according to the proof structure.
- The hierarchical structure recommended by Lamport for the most part works well, but there are occasions on which proofs of substeps may be used also in a different part of the proof, and there are as yet limited means within the hierarchical proof support tools to do so. Rigid hierarchical structure often entails duplication of proof fragments, which is unnecessary and undesirable for 'literacy'.

Issues specific to TLA are:

- A certain amount of pure temporal logic (TL) manipulation is needed for the liveness proofs;
- A library of derived TL rules is needed for effective use of TLA. Deriving TLA temporal logic rules needed even for such simple proofs as this is not easy, unless one happens to be good at temporal logic. The most significant derived rule, needed for Weak Fairness proofs, is the Bridge Rule

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

The Bridge Rule is derived in the formal proof on Page 46 from the STL rules. It is needed to tie together the conclusion of the Lattice Rule and the fourth hypothesis of WF2 in the liveness proof. It is also used in the proof that the Lazy Caching algorithm implements the Complete Cache algorithm, in [Lad96]. For background to the Lazy Caching algorithm and its proof of correctness, see [LLOR94]. In order to state derived rules, one needs to use *propositional variables*, logical symbols that may be substituted by propositions. These are not TLA variables. However, it seems appropriate to declare them when used. We thus introduce an extra-TLA meta-category PROPOSITIONAL VARIABLE to declare these variables.

- A restriction must be added to the application of the Lattice Rule in order that it shall be sound.

- An inference rule is not literally a statement of TLA. TLA proof steps are, however, supposed to be true TLA statements. Establishing the validity of a derived inference rule is a TLA meta-operation, and is not strictly TLA syntax. For convenience in writing proofs which need some temporal logic manipulation, it is desirable to allow the statement of inference rules as legitimate proof steps, even though TLA strictly defined does not allow it. The proof of such a rule, i.e., the substeps, consist in establishing its validity as a derived rule. Normally, that will be via a derivation. I've used such an operation to introduce and prove the Bridge Rule, as well as in other places. It obviates the need to claim a theorem outside of the proof proper, and then refer to it inside the proof. Other useful derived rules, which are used in the proof of the Bridge Rule and have frequent application in any temporal logic calculations, are STL7, STL2$^\diamond$, STL4$^\diamond$ and STL6$^\diamond$. See the formal proof of the Bridge Rule for statements of these rules.

- Propositional logic does not follow from the TLA proof rules as given in [Lam94b].[1] This is easy to fix and should be fixed by a user, either by including Modus Ponens and a set of Hilbert-style axioms for

[1] This may be seen as follows. Rules STL1 and STL4 are the only rules with non-trivial hypotheses. (The other rules with trivial hypotheses are usually called *axioms* in a Hilbert-style formulation of logic.) Both these rules increase the number of occurrences of the operator $\Box$ in the consequent over the number in the antecedent, as do all the STL rules (this follows trivially for the others). However, any propositional rule (or axiom) has the same number of occurrences of $\Box$ in hypothesis and consequent, namely, none. Thus no application of STL1 and STL4 can yield, for example, Modus Ponens, contrary

propositional logic, or by including a set of Gentzen-style 'introduction' and 'elimination' rules. Predicate logic is included in TLA, as shown in Rules E1 and F1 of [Lam94b, Figure 9].

3 Why Do This?

Why is this logic of practical use? Here are a few questions asked about an earlier version of this paper, and some responses. Many of these points have been made before, but it seems they are not yet clearly understood within the community. Similar issues are discussed in [Lam89].

Is this work of any practical importance? You're talking about proving the implementation, but still the implementation is just a formal specification. How is it related to a program?

The formal specification of the concrete buffer is a simplified specification of the design of a chip. (We have abstracted the four-bit-wide data path into a single data element. In a verification of this design, this abstraction would have to be decomposed into its component parts, but one would not expect the proof to differ significantly – just to gain a few new theorems.) A formal specification may be a precise description of an algorithm or a program or an architecture or whatever. You can't tell from just looking at the symbols what a specification describes. It could be at a very low level of detail, describing the results of executing each individual step of an imperative program, or it could be thirty lines describing a property of a system implemented by many megalines of imperative code. A precise description of any implementation can always be considered to be a specification - a better one or a worse one. Even Fortran – but you might not be able easily to write down the exact semantics of a Fortran program, because it depends on the inner workings of the compiler (often a better approach is to describe the compiled code instead, using the semantics of the assembler [Yu93]). But if things are so tricky that you don't know what's going on or can't say so succinctly, how on earth can you expect that your code does what you think it does? In safety-critical applications, for example, it's crucial to know what the code does and how, and while proofs are merely hard, truly exhaustive testing is all but impossible.

to what Lamport says in [Lam94b, Section 5.6], that STL1 'incorporates all the rules of ordinary logic, such as *modus ponens*'. A similar extension to the argument shows that Modus Ponens does not follow from the Lattice Rule in combination with STL1–STL6.

To go about proving code correct is a task which can be structured into levels (e.g., [Par72, Par74, Wen78]). One separates the business of proving the architecture correct (i.e., that the machine implements the semantics of its machine code) from the business of proving the assembler correct (i.e., that the semantics of the assembler instructions are correct with respect to the machine code and its semantics), from the business of proving the compiler correct (i.e., that the sequence of instructions implementing each high-level statement implements its formal semantics) from the business of proving that the program is correct (one assumes that the instructions implement their formal semantics, and proves that the sequence of instructions implements the algorithm). You may ask 'where do the formal specs stop?' At the low end, at some level of granularity of the architecture, and at the high end, at the requirements specification. At which low level they terminate depends on the physics of the chip. But these questions (what does a proof apply to, how can you decompose proofs, what features can you show correct and how?) concern the metamathematics of a proof, which is not the topic of this paper. This paper is about the technical details of a particular proof, just one of many that would be involved in verification of code on a particular machine. In particular, proofs of liveness are important, because liveness is important, and further because engineers seem unused to specifying, or indeed thinking about, liveness.

What is important about liveness?

Liveness is important for systems with many concurrently-running processes, because, first, any system property may be regarded as a combination of safety + liveness [AS87] and, second, liveness is not guaranteed in a concurrent system. (Most specification methods understand and promote proofs mainly of *safety* properties.) For serial algorithms or code, one often assumes that at any stage there is nothing to prevent the next program action from taking place (PC users may smile at this point). It is generally more pressing to show that the program calculations yield the desired results. However, in concurrent systems, there are other processes in the environment whose actions may interfere with your own. These interferences may be encouraged (communication) or deplored (competition for other system resources). The trick is to design algorithms that succeed no matter what other processes do. And the property that *they succeed in doing something no matter what else happens* is assured by a proof of liveness.

Algorithms in TLA are described more carefully and more generally than in an imperative program. They are construed as a series of actions, defined by assertions concerning the pre- and post-values of program variables. For example, in Pascal one may write `x := 1` to generate an action which sets the value of `x` equal to `1`. In TLA, one may define an action $\texttt{unity} \triangleq \texttt{x}' = \texttt{1}$ which is a statement of logic expressing the postcondition that the value of x after the action, denoted $\texttt{x}'$, is equal to `1`. In a given state, many actions may be enabled, and similarly many actions may take place at once (precisely when the preceding and succeeding states satisfy the definitions of more than one action). In an imperative program, usually one action is defined to follow another, and which one may be known at compile time (sequential composition) or only at run-time (many conditionals). TLA can specify such algorithms, but also more general ones that exist in a concurrent environment, in which the ordering of statements from different processes in an execution is much less predictable, and therefore in which ensuring liveness becomes important. Buffers, even though they may be essentially sequential objects, operate mostly in a environment with other processes around. It's appropriate that they should be specified accordingly and liveness must be proven.

Where is the particular example you chose relevant, and are there any 'real' implementations around that directly relate to your specification of an implementation?

Every buffer implemented as an array in this way is a real implementation. A buffer implemented by a linked list or a ring buffer is different from this, and this proof shows nothing about such buffers. But one should be able to prove that they have the properties of a buffer also.

I prove that the properties of a buffer are true of an array with the operations I describe. That shows that a particular algorithm implements a buffer, if by buffer you mean something that satisfies the specification *AbstractBuffer*. The practical application is simply that if you implement a buffer by an array in this way, and want to know for sure that you have made no mathematical mistakes in your algorithm design, this proof shows that. And similarly if you join two such buffer implementations in series. But the subject matter of this paper is not really buffers. It's proofs and how to do them, in detail.

What's the point of doing such a detailed and intricate proof?

To distinguish between correct and incorrect proofs. Every correct proof admits such detail, and no incorrect proof does. If the detail works out, the proof is correct. Conversely, if the proof is incorrect, the detail will not work. Hence the point of the intricacy is to ensure that the proof is correct. A formal logic is defined syntactically by inference rules. Proofs of algorithm properties do not necessarily require elegant new theorems of the sort that excite mathematicians – after all, the mathematical insight went into devising the algorithm, and if you don't already have some idea of why the algorithm works, you're unlikely to be able to construct a formal proof of it. Rather, a formal proof requires a careful checking of details, and some organisation. If the details are not checked carefully, there may be gaps or falsehoods. It's easy to omit proof obligations which are only discovered in the course of performing the formal proof. I know of no scientist working in verification who has not had such an experience. 'Normal' mathematical standards do not seem to suffice – Lamport notes that many concurrent 'algorithms' which were incorrect have been published along with their 'proofs' [LLOR94].

There is also the problem of scale. Methods of organising large proofs to ensure that all the bits fit together requires some kind of reliable decomposition. Syntactic proof methods lend themselves to decompositions of this sort – if all the bits are locally correct, there is nothing more to do to make the proof globally correct. There is less of a problem of 'interpretation' or 'gaps' than with proofs that may not be completely formal and which rely on an interpretive or implicit semantic component.

Finally, syntactic proofs are the only type of proof universally recognised by logicians as unequivocally demonstrating the relation of logical consequence between assumptions and conclusions. Of course, the proof rules have to be justified semantically, but as for a programming language, this is done once and thoroughly, and then all syntactically correct proofs using these rules are known to be valid according to the intended meaning of the formulae.

How can I tell the difference between an 'important' or 'creative' proof step, and one that is just routine?

That's not so easy. There are some simple heuristics, such as that a step which has few substeps when expanded fully in logic is likely to be easier to perform than one which has many. However, it's not so obvious how to distinguish a routine proof with many steps from a 'creative' proof with many steps. One can attempt to locate the 'creativity' in a proof by finding metalogical theorems such as Theorem 1, which factor out logically routine steps and focus on the 'proof obligations', the logically variable steps. The obligations are necessary and sufficient conditions for a proof of a certain form to exist. This kind of theorem seems to be the closest we can come to defining 'creativity' in a proof at the moment.

4 A Short Introduction to TLA

TLA is an assertional method designed for the specification and verification of concurrent algorithms. It is based on the by-now standard view of a system execution as an infinite progressive sequence of states, each state being defined by the collection of values of variables that are true in that state. As the system executes, the states change. A specification (or a program) constrains the states to change in certain specific ways. For example, executing an assignment $x \leftarrow 1$ causes the state to change so that in the new state the value of the variable x is 1. The state may also change in other ways. For example, suppose microcode variables are included in the state. Many of these variables will also change during the assignment, as will the register in which this assignment is carried out. But the specification may not need to comment on these changes – they can be 'hidden'.

An *action* in TLA is semantically a relation between states. Let $\mathcal{A}$ be an action. A pair of states $\langle a, b\rangle \in \mathcal{A}$ just in case intuitively state b is what you get by executing the action in state a. We specify action $\mathcal{A}$ by giving a syntactic definition of the relation. For example, an action such as $x \leftarrow 1$ can be defined by the logical relation $\textit{x-in-the-new-state} = 1$. Notice that according to this definition, any action which ensures x in the new state is equal to 1 counts as a $x \leftarrow 1$ action. That's appropriate and useful in concurrency – a state change may incorporate many actions defined in a description (may *satisfy* many action predicates). What's important is that the relation between the new state and the old is precisely defined. If certain other variables may not change during the execution of an action, we have explicitly to state that in the action definition.

TLA refers to *x-in-the-new-state* syntactically by use of a special operator. In defining actions, one wants to talk about *x-in-the-old-state* and *x-in-the-new-state*, and to specify *mathematical* relations between these values. Values of variables in the old state are referred to by just using the variable name, say x. Values of the variable x in the new state are referred to as x', as in the language Z [Spi89]. A logical definition of the action will be a mathematical expression involving x and x'. The assignment action could be specified by the equation $x' = 1$. Notice that 'normal' mathematical equality is used in this definition; in fact, normal mathematical assertions and inferences can be made without tears. TLA uses normal mathematics for talking about everything. It includes Zermelo-Fraenkel set theory (just as in mathematics), so Riemann integrals and suchlike can be defined and used in a TLA specification just as you would use them in normal mathematics [Lam93b].

Variables such as x above may change value from state to state, and are called *flexible variables*. In contrast, there are parameters of formulas (for example, the a in the action $push(a)$ which we use) which have a constant but unspecified value, not dependent on state. They are called *rigid variables*. Quantification over rigid and over flexible variables is different. If x is flexible, the quantifier $\boldsymbol{\exists}\, x$ means that *there is a value of* x *in every state* such that [...]. In different states, the properties asserted of x may or may not hold (for example, the value of x may be related to the sequence of previous values of x). In contrast, quantification over rigid variables is just normal first-order quantification. The difference is not so important for the buffer example, since only rigid variables are quantified over.

TLA is a temporal logic. How much temporal logic is actually used in a TLA proof that an implementation meets a specification? In the example in this paper, quite a lot. But the amount of pure temporal logic used seems to vary much less than linearly with the size of the specification. Hence not much of a real example consists of temporal reasoning – maybe 2–4%, as may be seen from the lengthy proof in [LLOR95]. Most of the reasoning is straight mathematics and propositional and predicate logic. Theorem 1 shows that, of the five proof obligations incurred for an implementation proof, four are classical logic and just one is temporal logic - and that one is the liveness proof. TLA$^+$ includes the usual module constructs, as well as useful shorthand notation for functions and finite sets, and such goodies.

An uncommon logical feature of TLA is the CHOOSE operator, Hilbert's ϵ operator. I use it in the definition of the length of a sequence. CHOOSE binds a variable and a predicate, and the result is a term. CHOOSE $x\,:\,P(x)$ denotes an object that satisfies P, if there is one, although it is not specified which if there is more than one, and has an unspecified value if there is no such object. It is extensional, in that it selects the same object satisfying equivalent predicates. It's a primitive of TLA$^+$. CHOOSE satisfies axioms such as the following:

$$\vdash (\exists\, x\,:\,P(x)) \Rightarrow P(\textsc{choose}\ x\,:\,P(x))$$

provided that no instance of CHOOSE $x\,:\,P(x)$ appears within the scope of a construct (a quantifier or another CHOOSE) that binds a free variable y

in $\text{CHOOSE } x : P(x)$ (this condition may be fulfilled by a suitable change of bound variable at the appropriate place); and

$$\vdash (\forall x : (A(x) \equiv B(x))) \Rightarrow (\text{CHOOSE } x : A(x)) = (\text{CHOOSE } x : B(x))$$

Proving implementations correct in TLA is done by writing both the high-level specification and a description of the implementation in TLA, and proving that the implementation description logically implies the specification. Implementation is implication. The proof is carried out by using the assertional method [Gri81] as adapted to concurrent programs by Ashcroft [Ash75]. Section 8 gives a general TLA proof scheme for showing that one description implements another. A proof using this schema may be many pages long, for examples see [LLOR95].

A TLA specification has the form $Init \wedge \Box[\mathcal{N}]_{vars} \wedge F_{vars}(\mathcal{M})$. *Init* is a description of the initial state in which the system starts. $\Box[\mathcal{N}]_{vars}$ asserts the *safety* properties of the system. A safety property asserts that nothing untoward happens during execution. The standard type of safety property asserts that every action taken by the system in an execution is a legitimate action defined by the specification. $\Box$ is the *always* operator, saying that the following assertion is true in every state. $\mathcal{N}$ is the disjunction of all the action definitions of the system, i.e. it asserts that every step is a step which satisfies one or other of the action predicates of the specification. $[\mathcal{N}]_{vars}$ is defined to mean $\mathcal{N} \vee vars' = vars$, i.e. that either $\mathcal{N}$ holds or the variables of interest do not change value during the step (in which case the step is a so-called *stuttering* step). (When considering liveness, we shall also need the notation $\langle\mathcal{N}\rangle_{vars}$ which is defined to mean $\mathcal{N} \wedge vars' \neq vars$.) The entire formula $\Box[\mathcal{N}]_{vars}$ thus says that every step is either a legitimate action, or the values of the variables of interest don't change.

But one way for a system to satisfy the safety formula is for it never to progress beyond the initial state, i.e. that $vars' = vars$ is true at every state change, for ever. In order for the system to do any useful work, it must actually take some legitimate step at some point. The various ways of saying this are known as *liveness* properties. The forms of liveness property used in TLA specifications is a conjunction of so-called *Weak Fairness* requirements on actions and *Strong Fairness* requirements, specified in the formula above by the conjunct $F_{vars}(\mathcal{M})$. In this paper, we only consider weak fairness requirements, denoted $WF_{vars}(\mathcal{M})$. To assert weak fairness of an action $\mathcal{A}$ means to say that if $\mathcal{A}$ is enabled continuously, it must eventually happen in some state change. For an action to be *enabled* in a state s simply means that there is some state t such that the pair of states $\langle s, t \rangle$ satisfies the action predicate (it doesn't say that t *must be* the next state in the execution, just that it *could* be). More formally, $\mathit{Enabled}(\mathcal{A})$ is the formula obtained from $\mathcal{A}$ by replacing the primed variables by new variables and existentially quantifying them all. For example, $\mathit{Enabled}(\mathit{unity}) \quad \triangleq \quad \exists\, c \,:\, c = 1$. To perform a proof, it's not necessary to know how weak fairness is expressed in temporal logic, since there's a TLA proof rule geared specially to the proof of Weak Fairness properties. But this proof rule ensures that liveness properties of the specification must be proved from liveness properties asserted of the implementation – they don't come from nowhere! Sufficiently much liveness must be included in the implementation description to enable the liveness properties of the specification to be proved. (Connoisseurs who wish to consider other liveness properties than weak or strong fairness may write them out directly in temporal logic and reason with them using the PTL rules. TLA only includes specific inference rules for weak and strong fairness.)

One final TLA operator that appears in the proof rules is the '*leadsto*' operator, $\leadsto$. To say $P \leadsto Q$ is to say that it is always the case that if P then eventually Q, expressed in temporal logic by $\Box(P \Rightarrow \Diamond Q)$. We shall use the $\leadsto$ operator and its definition in the proof.

This brief description of TLA leads to the specification and the proof of implementation.

5 Constructing the Specifications

We use three specifications for this example. *AbstractBuffer* (Figure 5) specifies the actions, safety and liveness properties which define a bounded buffer of size N, as in Figure 1. The buffer is represented by variable *buffer*, and has *push* and *pop* actions respectively to put elements in the buffer (provided that there aren't N elements there already), and to take elements out of the buffer (provided there is something there). The *buffer* itself is a sequence. A state of the buffer is given by the values of variables, that is, by the value of *buffer*. The final portion of the module contains the assertion of properties of the buffer: that it starts off empty, that every state change of the buffer is caused by a push or pop action, and that always if there's something in the buffer, eventually a pop must happen (recall that we don't require that anything is ever put in the buffer).

The specification *ConcreteBuffer* (Figure 6) implements this in a very

$$
\begin{array}{ccc}
\mathit{buffer} & & \mathit{buffer}' \\
< b, x, y, d > & \xrightarrow{\quad pop \quad} & < x, y, d > \\[2em]
< b, x, y, d > & \xrightarrow{\quad push(a) \quad} & < b, x, y, d, a >
\end{array}
$$

Figure 1: An Abstract Buffer with Operations

$$
\begin{array}{ccc}
\mathit{Buffer} & & \mathit{Buffer}' \\
\begin{array}{|c|c|c|c|c|c|c|}\hline \perp & b & x & y & \perp & d & \perp \\ \hline\end{array} & \xrightarrow{\quad push(a) \quad} & \begin{array}{|c|c|c|c|c|c|c|}\hline \perp & b & x & y & \perp & d & a \\ \hline\end{array} \\[2em]
\begin{array}{|c|c|c|c|c|c|c|}\hline \perp & b & x & y & \perp & d & a \\ \hline\end{array} & \xrightarrow{\quad move(6) \quad} & \begin{array}{|c|c|c|c|c|c|c|}\hline \perp & b & x & y & d & \perp & a \\ \hline\end{array} \\[2em]
\begin{array}{|c|c|c|c|c|c|c|}\hline \perp & b & x & y & d & \perp & a \\ \hline\end{array} & \xrightarrow{\quad move(2) \quad} & \begin{array}{|c|c|c|c|c|c|c|}\hline b & \perp & x & y & d & \perp & a \\ \hline\end{array} \\[2em]
\begin{array}{|c|c|c|c|c|c|c|}\hline b & \perp & x & y & d & \perp & a \\ \hline\end{array} & \xrightarrow{\quad pop \quad} & \begin{array}{|c|c|c|c|c|c|c|}\hline \perp & \perp & x & y & d & \perp & a \\ \hline\end{array}
\end{array}
$$

Figure 2: A Concrete Buffer with Operations

particular way. The *Buffer* is an array of fixed length N, as in Figure 2, where $N = 7$. Data values are inserted at the end of the array by *push*, provided this terminal position is void, and are removed from the first position of the array by *pop*, providing this position is non-void. Thus *push*, but not *pop*, is parametrised by the data value that is pushed. Since this value does not change from state to state (although the element is moved around, it doesn't spontaneously change), it is a rigid variable so we may employ a first-order existential quantifier. $\exists\, a : push(a)$ asserts that some element has been pushed. To move elements from the rear of the *Buffer* to the front, there is an action *move* which moves an element into the position in front, if this position is void. *move* is itself parametrised by the starting position of the element moved. This parameter is another rigid variable. Existential quantifiers over these variables are the only quantifiers that appear in the specification. The specification is a logical formula asserting that the array starts out filled with $\perp$ elements (which are not $Data$, by the assertion), that every state change of the Buffer is caused by a push, a pop or a move, and finally that if something can be popped, it eventually will be, and that if something can be moved, it eventually will be (recall our discussion in Section 1 why this is necessary). We need to show that the concrete buffer defined as an array implements the abstract buffer. A different way to implement the concrete buffer would be as a linked list.

AbstractBuffer and *ConcreteBuffer*, as well as *Sequences*, import the *Naturals* module, which is not given here. *Naturals* includes definitions of the set *Nat* of natural numbers along with the standard arithmetic operations. It also defines such constant symbols as 1, 2. Since there are many ways one could wish to write such a module, we don't commit ourselves to one. The use we make of it is slight and any reasonable definition will suffice.

Finally, we write a double-buffer specification *DoubleBuffer* (Figure 7), which implements a buffer of size $2N$ by arranging two concrete buffers of size N in series, as in Figure 3 in which $N = 3$. This specification has similar actions to the concrete buffer, with identical enabling conditions, but one *move* action consists in shuttling an element from the front of the rear buffer to the rear of the front buffer. This is accomplished by an action which is simultaneously a *pop* of the rear buffer and a *push* of the same popped element onto the front buffer.

An important feature of the *DoubleBuffer* specification is that the *same* element that is popped from the rear buffer must appear on the front buffer. Consequently, $move(N+1)$ is defined as $B1.pop \wedge B2.push(B1[1])$.

The goal is to show that this double buffer implements an abstract buffer

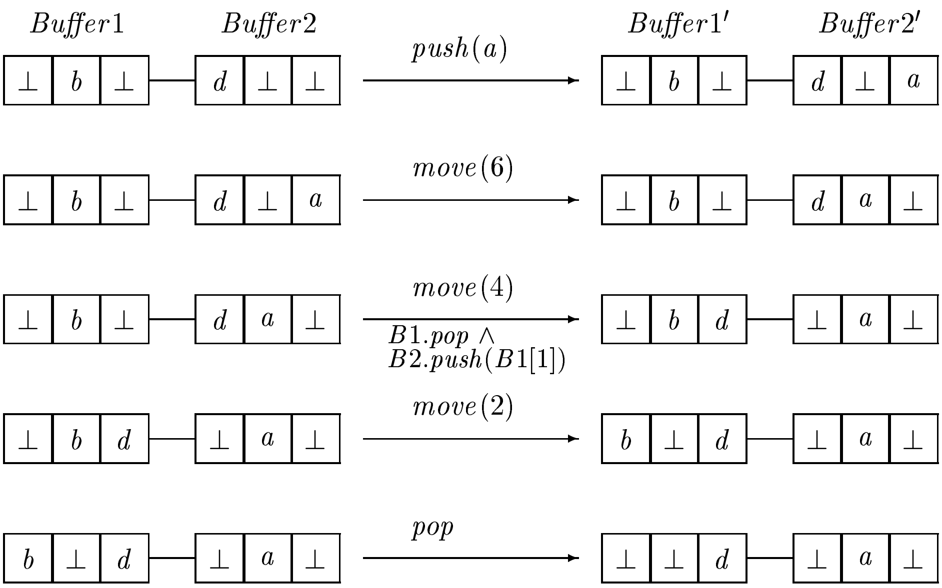

Figure 3: A Double Buffer with Operations

of size $2N$. First, we prove that the double buffer actually implements a single array buffer with the same total number of places. We then prove that a concrete array buffer implements the abstract buffer specification. These two assertions are contained in Module *Theorems* (Figure 9).

TLA is a system for writing and verifying specifications and implementations of concurrent algorithms. Is there any explicit concurrency in this specification? Just a little, in module *DoubleBuffer*. The specification *DoubleBuffer* specifies a serial 'composition' of two buffers, of which the actions are not generally constrained to synchronise. The only exception to this is the *move(N+1)*, which is defined to be simultaneously a *pop* of *Buffer2* and a *push* of the same element on *Buffer1*. TLA specifications in general don't enforce concurrency like this, but don't rule it out either, except by means of whatever conditions are explicitly written in the action definitions.

6 Proving the Implementation

The proof that the system specified in *DoubleBuffer* implements the system specified in *AbstractBuffer*, with the parameter N of *AbstractBuffer* instantiated as $2.N$, is straightforward. Implementation in TLA is implication, and we can use the transitivity of implication. We show that the double buffer implements a single concrete buffer, and then we show that a concrete buffer implements an abstract buffer. (In the Module *Theorems*, where this is stated, we retain the parameter N in this second assertion for convenience. Literally, the parameter should be $2N$, but since the parameter is a parameter of the proof, a proof for N is isomorphic to a proof for $2N$. We might as well save ink and say 'N'.)

Implementations of specifications, that is, implications of TLA specification formulas, are proved according to a template given in Section 8, available also as [LL94]. Call the description of the implementation the *implementiens*, and the description that is (asserted to be) implemented the *implementiendum*. The safety properties are proved by step simulation: a step of the implementiens is either an action step or a stuttering step of the implementiendum. The liveness properties of the implementiendum are proved from the liveness properties of the implementiens. In order to show the implication, it may be necessary to add history variables (state functions which record part of the history of a computation), or so-called prophecy variables (variables that predict a future action) to a specification. One may always add such variables, and the specification obtained by adding history

Temporal Logic Rules

STL1. $\dfrac{F \text{ a tautology}}{\Box F}$ STL4. $\dfrac{F \Rightarrow G}{\Box F \;\Rightarrow\; \Box G}$

STL2. $\vdash \Box F \;\Rightarrow\; F$ STL5. $\vdash \Box(F \wedge G) \;\equiv\; (\Box F) \wedge (\Box G)$

STL3. $\vdash \Box\Box F \;\equiv\; \Box F$ STL6. $\vdash (\Diamond\Box F) \;\wedge\; (\Diamond\Box G) \;\equiv\; \Diamond\Box(F \wedge G)$

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

$$\frac{F \wedge (\mathsf{c} \in \mathsf{S}) \;\Rightarrow\; (H_{\mathsf{c}} \leadsto (G \;\vee\; \exists \mathsf{d} \in \mathsf{S} : (\mathsf{c} \succ \mathsf{d}) \wedge H_{\mathsf{d}}))}{F \;\Rightarrow\; ((\exists \mathsf{c} \in \mathsf{S} : H_{\mathsf{c}}) \leadsto G)}$$

Basic TLA Rules

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

Invariance and Weak Fairness Rules

INV1. $\dfrac{I \wedge [\mathcal{N}]_f \Rightarrow I'}{I \wedge \Box[\mathcal{N}]_f \Rightarrow \Box I}$ INV2. $\vdash \Box I \;\Rightarrow\; (\Box[\mathcal{N}]_f \;\equiv\; \Box[\mathcal{N} \wedge \mathcal{I} \wedge \mathcal{I}']_f)$

WF1.

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

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

Figure 4: Axioms and proof rules used of TLA.

variables is logically equivalent to the original specification when the history variables are hidden by quantification. Prophecy variables were defined in [AL91]. A prophecy variable, *Descending*, a so-called stuttering variable, is used in the proof that the concrete buffer implements the abstract buffer. It's a state function defined in the scope of the proof. A stuttering variable is a variable on a well-founded order (often, as in this case, the natural numbers) which is decreased by implementation steps which are stuttering steps of the specification. When this variable reaches a minimum value, then some implementation step occurs which is a non-stuttering step of the specification. Its value at the outset predicts how many stuttering steps may occur before a non-stuttering step is enabled. In this case, how many *CB.move*s may occur before a *CB.pop* is enabled.

Firstly, we show that *DoubleBuffer* is a *ConcreteBuffer* with size $2.N$ – the first implication in Module *Theorems*. The step simulation is straightforward. There are no stuttering steps. The *push* action is the *push* of the rear buffer, the *pop* is the *pop* of the forward buffer, and the *move* action is the *move* action of whichever buffer holds the two slots – in the case it's a move from the first slot of the rear buffer to the last slot of the forward buffer, the *move* action is simultaneously a *pop* of the rear and a *push* of the forward with the same element (whatever that element may be).

In order to show that the *ConcreteBuffer* implements an *AbstractBuffer*, we need to produce a state function in *ConcreteBuffer* that satisfies the conditions of being the *AbstractBuffer.buffer*. This association of a state function with each existentially quantified variable of *AbstractBuffer* in the theorem is called the *refinement mapping*. It's easy to see what this mapping should be. In *AbstractBuffer*, the *buffer* is regarded as a sequence of whatever has been put in, up to length N. In *ConcreteBuffer*, there are always N slots in *Buffer*, some of which are filled and some not. If we ignore the slots that aren't filled, the sequence of elements is the sequence needed for the *AbstractBuffer.buffer*. In Module *Sequences*, a standard TLA$^+$ module used in other specifications [LLOR94], the function *SelectSeq* forms a sequence from an input sequence by selecting those elements of the input sequence which satisfy a particular predicate. (The alert may notice that *Sequences* imports a module *Naturals*, which defines certain properties of the natural numbers. We don't bother to include the module as a figure.) We choose the predicate saying that a value is non-void, and define the state function consisting of the sequence of non-void elements of the *ConcreteBuffer.Buffer*. This state function implements *AbstractBuffer.buffer*, and is the sole state-function/variable pair in the refinement mapping.

Step simulation is straightforward. The action *ConcreteBuffer.pop* simulates the action *AbstractBuffer.pop*, and similarly for *ConcreteBuffer.push* and *AbstractBuffer.push*. The only conditions that are not trivial are those on *Len(buffer)*. However, it is easy to prove that *ConcreteBuffer.push* adds a non-void element to *ConcreteBuffer.Buffer*, without removing any non-void elements that are there already and thus increases the length of the state function *SelectSeq(Buffer,NonVoid)* simulating *AbstractBuffer.buffer* by 1. Similarly, it is easy to see that *ConcreteBuffer.pop* decreases the number of non-void elements of *ConcreteBuffer.Buffer* by 1, without adding any. It is straightforward to show that the action *ConcreteBuffer.move* is a stuttering step of *AbstractBuffer*. This completes the demonstration of safety properties.

Liveness properties are more complex. The *AbstractBuffer* satisfies the liveness property that if a *pop* may be done, eventually it will be done. Since the *pop* of the *AbstractBuffer* is simulated by the *pop* of the *ConcreteBuffer*, *ConcreteBuffer.pop* needs to satisfy the same liveness property. However, the enabling conditions for the *AbstractBuffer.pop* and the *ConcreteBuffer.pop* are not the same! *AbstractBuffer.pop* is enabled when there is some non-void element in *AbstractBuffer.buffer*. However, *ConcreteBuffer.pop* has the *prima facie* stronger requirement that there must be a non-void element in the *first place* of its *Buffer*. But if there is not, and there is a non-void element somewhere in the *Buffer*, some *move* is enabled. To see why, note that there must be some first k such that $Buffer[k]$ is non-void. Thus $Buffer[k-1]$ must be void by hypothesis (since k is at least 2 by hypothesis, $Buffer[k-1]$ is a well-defined *Buffer* place) and hence $move(k)$ is enabled. Other moves may also be enabled, but this first element is important since it's the one which will eventually be popped off the front by a *ConcreteBuffer.pop* operation, which simulates the *AbstractBuffer.pop* that we are trying to show will occur. Any enabled concrete *move* remains enabled if a concrete *push* occurs, since only $Buffer[N]$ is affected by a *push*, only $move(N)$ has any precondition concerning $Buffer[N]$, and $move(N)$ is not enabled if *push* is enabled. Similar reasoning shows that when $move(k)$ occurs, any enabled $move(j)$ for $j \neq k$ remains enabled (note that $move(k+1)$ could not have been enabled if $move(k)$ was enabled, but might be enabled afterwards if $Buffer[k+1]$ were non empty). Thus, for k the first number such that $Buffer[k]$ is non-void, $move(k)$ remains enabled through other *move*, or *push*, operations. Weak liveness guarantees it will eventually be executed, whence $Buffer[k-1]$ becomes the first non-void *Buffer* place. The same reasoning applies now to $Buffer[k-1]$ and $move(k-1)$. Intuitively, the first non-void element will eventually make it to *Buffer*[1], at which point *ConcreteBuffer.pop* becomes enabled. *ConcreteBuffer.pop* remains not enabled until *Buffer*[1] is non-void, at which point it becomes enabled. It remains enabled and thus must eventually be executed, according to its weak fairness requirement. This step simulates *AbstractBuffer.pop*. Thus is the weak-fairness requirement on *AbstractBuffer.pop* satisfied. Notice that one requires not only the liveness of *ConcreteBuffer.pop* to show that *AbstractBuffer.pop* is live, but also the liveness of *ConcreteBuffer.move*. If *ConcreteBuffer.move* is not required to be live, the *ConcreteBuffer* could sit there with elements non-void, and with *ConcreteBuffer.push* and *ConcreteBuffer.pop* both disabled, waiting for an enabled *move* that will never happen. The liveness of *AbstractBuffer.pop* could not be proven in this case, since *AbstractBuffer.pop* is enabled, but will not be executed. Thus, the liveness of *ConcreteBuffer.move* is crucial for the proof of liveness of *AbstractBuffer.pop*.

We have informally indicated the refinement mapping, and the step simulation, and how the step simulation and the liveness requirements are proved. This suffices to construct a formal proof in TLA of the implementation according to the template given in Section 8.

There are two minor counters to our claim that the proof we give is syntacically formal. One step of temporal logic is required which we were unable to derive strictly from STL1–STL6 and the Lattice Rule, the TLA temporal logic rules. Further, we were unable to modify our use of WF2, the Lattice Rule and WF1 in order to avoid this calculation in temporal logic. Problems of this sort may easily be avoided by adjoining rules of propositional logic, as we noted in Section 2. A simple derivation may be given using Modus Ponens and tautologies, and we include one due to Lamport.

The second counter is that we require a small bit of simple arithmetic in order to perform the calculations with *Descending* for the stuttering steps (*Descending* is a so-called *stuttering variable*). We have described no *Naturals* module and therefore we assume that the arithmetic is adequate to derive the trivial arithmetic results we need. The steps which need such arithmetic are marked in the proof, and the reader can easily see by inspection that the arithmetic is correct - it involves merely noting such facts as that (a) $+$ is associative; (b) $(i - 2^{N-(k-1)} + 2^{N-k}) < i$ for $k \in 2..N$; and (c) $2^{N-N} = 1$ and $(i-1) < i$. Assertion (a) should be included in any arithmetic module, since it's a basic axiom for addition, and (b) and (c) require only easy calculations with exponents and $<$. We assume the reader is sufficiently assured of the truth of these two assertions not to need a formal proof of either.

7 The Proof Style

We use the hierarchical proof style advocated by (and supported by the macros of) Lamport [Lam93a]. Each step is justified by a sequence of lower-level steps, such that it is a rigorous logical consequence of the lower-level steps. The inference rules which may be used to derive a step from a sequence of lower-level steps are given in Figure 4 (the use of propositional logic, predicate logic, and some elementary propositional temporal logic, is understood). Each lower-level step in turn must be justified in the same manner. The proof stops descending when propositional logic, or obvious theorems concerning data structures such as the natural numbers, or properties of sequences, or any known mathematical theorems, are reached. In practice, one halts a descent in a proof when one is certain how to complete the proof but too lazy to do so.

The point about the style is that a proof in TLA is logically rigorous, and entirely syntactic. That means that each step in the proof is derived from other, named, steps using only the permitted syntactic rules of inference. Furthermore, no new principles need to be used in order to prove a new example correct. TLA is a logic, with general rules, and has remained stable for a number of years. Every proof of correctness in TLA uses the same old TLA rules, albeit in a new way. The proof style supports this rigorous syntactic derivation.

The proof steps are numbered in the following way: step $\langle n \rangle.m$ has as substeps steps $\langle n+1 \rangle.1$ through $\langle n+1 \rangle.k$, with the last step being a *QED* step. The proof of the *QED* step is usually a commentary indicating how the steps $\langle n+1 \rangle.1$ through $\langle n+1 \rangle.k$ yield the proof of step $\langle n \rangle.m$.

Formulas are also written in a special form. Conjuncts and disjuncts may be lengthy, and so rather than being written in a line, they are written as tableaux in which each line is a separate con(dis)junct, preceded by the con(dis)junction sign. Thus nested con(dis)juncts are indented, and consequently much easier to read or to manipulate. The notation is explained in [Lam94a].

8 The Proof Scheme for Implementation

In order to introduce the proof style, and to give a taste of a TLA proof, we present a general proof scheme to be used when showing that one specification $(\exists\, y \,:\, \mathit{Init}_1 \wedge \Box[N_1]_v \wedge L_1)$ implements another $(\exists\, x \,:\, \mathit{Init}_2 \wedge \Box[N_2]_w \wedge L_2)$. This schema illustrates the hierarchical proof style recommended for use with TLA [Lam93a], and briefly explained in Section 7 above.

To Prove:

$$(\exists\, y \,:\, \mathit{Init}_1 \wedge \Box[N_1]_v \wedge L_1) \Rightarrow (\exists\, x \,:\, \mathit{Init}_2 \wedge \Box[N_2]_w \wedge L_2)$$

Most of the structure of an implementation proof is just logic, that is, propositional logic and application of TLA Proof Rules, as may be seen below. The important parts to show for each individual problem instance are indicated by '********'. All of these are simply mathematics and not temporal logic, with the important exception of the liveness proof. Showing liveness can involve some non-trivial temporal logic, and uses rules WF1, Lattice, and WF2.

First, the definition of the refinement mapping.

LET: $\overline{\mathit{Init}_2} \triangleq \mathit{Init}_2 \quad \mathit{with} \quad x \leftarrow \overline{x}$
$\overline{w} \triangleq w \quad \mathit{with} \quad x \leftarrow \overline{x}$
$\overline{L_2} \triangleq L_2 \quad \mathit{with} \quad x \leftarrow \overline{x}$

Now the proof that the initial condition of the implementation implies the initial condition of the specification.

$\langle 1 \rangle 1.$ $\mathit{Init}_1 \Rightarrow \overline{\mathit{Init}_2}$
PROOF: ******This must be shown for each problem instance.******* $\Box$

Next, the proof that the initial condition and safety property of the implementation implies that there is an invariant.

$\langle 1 \rangle 2.$ $\mathit{Init}_1 \wedge \Box[N_1]_v \Rightarrow \Box \mathit{Inv}$
$\langle 2 \rangle 1.$ $\mathit{Init}_1 \Rightarrow \mathit{Inv}$
PROOF: ******This must be shown for each problem instance.******* $\Box$
$\langle 2 \rangle 2.$ $\mathit{Inv} \wedge \Box[N_1]_v \Rightarrow \Box \mathit{Inv}$
PROOF:
$\langle 3 \rangle 1.$ $\mathit{Inv} \wedge [N_1]_v \Rightarrow \mathit{Inv}'$
PROOF: ******This must be shown for each problem instance*******. $\Box$

⟨3⟩2. Q.E.D.

PROOF: Immediate from ⟨3⟩1 by Rule INV1. □

⟨2⟩3. Q.E.D.

PROOF: Immediate from ⟨2⟩1 and ⟨2⟩2 by propositional logic. □

Using the invariant and the safety property of the implementation, one derives the safety property of the specification. Some manipulation using the invariant is needed in the substeps.

⟨1⟩3. $\mathit{Init}_1 \wedge \Box[N_1]_v \Rightarrow \Box[\overline{N_2}]_{\overline{w}}$

PROOF:

⟨2⟩1. $\Box \mathit{Inv} \wedge \Box[N_1]_v \Rightarrow \Box[\overline{N_2}]_{\overline{w}}$

PROOF:

⟨3⟩1. $\Box \mathit{Inv} \wedge \Box[N_1]_v \Rightarrow \Box[N_1 \wedge \mathit{Inv} \wedge \mathit{Inv}']_v$

PROOF: Immediate from INV2 using STL5. □

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

PROOF:

⟨4⟩1. $[N_1 \wedge \mathit{Inv} \wedge \mathit{Inv}']_v \Rightarrow [\overline{N_2}]_{\overline{w}}$

PROOF: ******This must be shown*********. □

⟨4⟩2. Q.E.D.

PROOF: Immediate from ⟨4⟩1 by STL4. □

⟨3⟩3. Q.E.D.

PROOF: Immediate from ⟨3⟩1 and ⟨3⟩2 by propositional logic. □

⟨2⟩2. Q.E.D.

PROOF: Immediate from ⟨1⟩2 and ⟨2⟩1 by propositional logic. □

The proof that the liveness property of the specification is implied by the implementation has no general form. The TLA Proof Rules WF1, Lattice, and WF2 are used, and it is necessary to instantiate the hypotheses of these rules by appropriate formula taken from the problem instance. This argument may involve some non-trivial temporal logic, but is the only part of a TLA implementation proof to do so.

⟨1⟩4. $\Box \mathit{Inv} \wedge \Box[N_1]_v \wedge L_1 \Rightarrow \overline{L_2}$

PROOF: *******Uses Rules WF1, WF2 and the Lattice Rule.*********

⟨1⟩5. $\mathit{Init}_1 \wedge \Box[N_1]_v \wedge L_1 \Rightarrow \overline{\mathit{Init}_2} \wedge \Box[\overline{N_2}]_{\overline{w}} \wedge \overline{L_2}$

PROOF: Immediate from ⟨1⟩1, ⟨1⟩3 and ⟨1⟩4 by propositional logic. □

⟨1⟩6. Q.E.D.

PROOF: Theorem follows from ⟨1⟩5 by quantifying over the free variables. □

One may observe directly from the proof scheme above that to prove the theorem for any given problem instance, it suffices to show just the following five steps, only the last of which may involve temporal logic:

module *AbstractBuffer*

import

Naturals, *Sequences*

parameters

buffer : VARIABLE

Data, N : CONSTANT

predicates

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

actions

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

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

temporal

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

Figure 5: Module *AbstractBuffer*

$\langle 1 \rangle 1.\ Init_1 \Rightarrow \overline{Init_2}$
$\langle 1 \rangle 2.\ Init_1 \Rightarrow Inv$
$\langle 1 \rangle 3.\ Inv \land [N_1]_v \Rightarrow Inv'$
$\langle 1 \rangle 4.\ [N_1 \land Inv \land Inv']_v \Rightarrow [\overline{N_2}]_{\overline{w}}$
$\langle 1 \rangle 5.\ \Box Inv \land \Box[N_1]_v \land L_1 \Rightarrow \overline{L_2}$

The following theorem summarises this information.

Theorem 1 $\vdash_{TLA} (\exists\, y : Init_1 \land \Box[N_1]_v \land L_1) \Rightarrow (\exists\, x : Init_2 \land \Box[N_2]_w \land L_2)$ if there is a formula Inv and refinement mapping $\overline{x}$ such that $\vdash_{PredicateLogic} (Init_1 \Rightarrow \overline{Init_2}) \land (Init_1 \Rightarrow Inv) \land (Inv \land [N_1]_v \Rightarrow Inv')$ $\land\ ([N_1 \land Inv \land Inv']_v \Rightarrow [\overline{N_2}]_{\overline{w}})$ and $\vdash_{TLA} (\Box Inv \land \Box[N_1]_v \land L_1 \Rightarrow \overline{L_2})$

Theorem 1 will be used to simplify syntactically the proof of liveness of the abstract buffer from the liveness assumptions of the concrete buffer.

module *ConcreteBuffer*

parameters

$Buffer$: VARIABLE

$\perp, Data, N$: CONSTANT

import

$Naturals, Sequences$

assumption

$N \in Nat$

definition

$\perp \;\triangleq\; \text{CHOOSE } x : x \notin Data$

predicates

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

actions

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

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

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

temporal

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

Figure 6: Module *ConcreteBuffer*

module *DoubleBuffer*

parameters

$Buffer1, Buffer2 \; : \; \text{VARIABLE}$

$Data, N \; : \; \text{CONSTANT}$

import *Naturals, Sequences*

include *ConcreteBuffer* **as** *B1* **with** *Buffer* $\leftarrow$ *Buffer1, N* $\leftarrow$ *N*

include *ConcreteBuffer* **as** *B2* **with** *Buffer* $\leftarrow$ *Buffer2, N* $\leftarrow$ *N*

predicates

$$Init \triangleq \begin{array}{l} \land \forall n \in 1 \,..\, N \; : \; Buffer1[n] = \bot \\ \land \forall n \in 1 \,..\, N \; : \; Buffer2[n] = \bot \end{array}$$

actions

$$push(a) \triangleq B2.push(a) \land \text{UNCHANGED } Buffer1$$

$$pop \triangleq B1.pop \land \text{UNCHANGED } Buffer2$$

$$move(k) \triangleq \begin{array}{l} \land \; k \in 2..2N \\ \land \; k \in 2 \,..\, N \\ \qquad \Rightarrow \\ \qquad \land \; B1.move(k) \\ \qquad \land \; \forall i \in (2 \,..\, N) \setminus \{k, k-1\} \; : \; \text{UNCHANGED } Buffer1[i] \\ \qquad \land \; \text{UNCHANGED } Buffer2 \\ \land \; k \in N+2 \,..\, 2N \quad \Rightarrow \\ \qquad \land \; B2.move(k-N) \\ \qquad \land \; \forall i \in (2 \,..\, N) \setminus \{k-N, k-N-1\} \; : \; \text{UNCHANGED } Buffer2[i] \\ \qquad \land \; \text{UNCHANGED } Buffer1 \\ \land \; k = N+1 \\ \qquad \Rightarrow \\ \qquad B2.pop \land B1.push(Buffer2[1]) \end{array}$$

temporal

$$Spec \triangleq \begin{array}{l} \land \; Init \\ \land \; \Box[pop \lor \exists\, b \; : \; push(b) \lor \exists\, k \; : \; move(k)]_{Buffer1, Buffer2} \\ \land \; WF_{Buffer1, Buffer2}(pop) \\ \land \; WF_{Buffer1, Buffer2}(\exists\, k \; : \; move(k)) \end{array}$$

Figure 7: Module *DoubleBuffer*

module *Sequences*

import *Naturals*

$$
\begin{array}{lll}
m \,..\, n & \triangleq & \{i \in Nat : (m \leq i) \wedge (i \leq n)\} \\
Len(s) & \triangleq & \text{CHOOSE } n : (n \in Nat) \wedge ((\text{DOMAIN } s) = (1 \,..\, n)) \\
Head(s) & \triangleq & s[1] \\
Tail(s) & \triangleq & [i \in 1 \,..(Len(s) - 1) \mapsto s[i+1]] \\
s \circ t & \triangleq & [i \in 1 \,..(Len(s) + Len(t)) \mapsto \textbf{if } i \leq Len(s) \textbf{ then } s[i] \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else } t[i - Len(s)]] \\
Seq(S) & \triangleq & \text{UNION } \{[(1 \,..\, n) \rightarrow S] : n \in Nat\}
\end{array}
$$

$$
\begin{array}{l}
SelectSeq(s, test(\_)) \triangleq \textbf{let } F[t : Seq(\{s[i] : i \in (1 \,..\, Len(s))\})] \triangleq \\
\qquad \textbf{if } t = \langle\,\rangle \textbf{ then } \langle\,\rangle \\
\qquad\quad \textbf{else if } test(Head(t)) \\
\qquad\qquad \textbf{then } \langle Head(t) \rangle \circ F[Tail(t)] \\
\qquad\qquad \textbf{else } F[Tail(t)] \\
\quad \textbf{in } F[s] \\
SubSeq(s, m, n) \triangleq [i \in (1 \,..(1 + n - m)) \mapsto s[i + m - 1]]
\end{array}
$$

Figure 8: Module *Sequences*

module *Theorems*

include *Doublebuffer* **as** *DB(N)*
include *ConcreteBuffer* **as** *CB(N)*
include *AbstractBuffer* **as** *AB(N)*

theorems

$\exists\, Buffer1, Buffer2 : DB(N).Spec \Rightarrow \exists\, Buffer : CB(2N).Spec$

$\exists\, Buffer : CB(N).Spec \Rightarrow \exists\, buffer : AB(N).Spec$

Figure 9: Module *Theorems*

9 The Proof

We must assume that TLA contains either Modus Ponens plus a set of Hilbert-style axioms, or a set of Gentzen-style 'introduction' and 'elimination' rules, for propositional logic, as suggested in Section 2. We assume propositional and predicate logic as given - neither the purpose of TLA nor of this paper is to explain to the reader how to use them in a proof.

I use a conventional wording for indicating the completion of a proof as follows:

- the word *follows* indicates that some propositional or predicate logic manipulation is required;
- *follows immediately* indicates that trivial propositional or predicate logic is required, such as forward-chaining of conditionals, rearrangement of hypotheses, or substution and quantification of a variable;
- *immediate* indicates that a direct textual substitution is required.

These 'rules' are not necessarily applied rigorously.

The first theorem proves that the *DoubleBuffer* implements a *Buffer* of twice the size. This is a straightforward proof following the schema in Section 8. It is also rigorously logically correct in TLA without recourse to metatheorems concerning the logic. Each step of the *DoubleBuffer* is a step of the *Buffer*, so there are no stuttering steps. Furthermore, since liveness of all the *move* operations of the *DoubleBuffer* is specified, including the *move* that is a *pop* of the rear buffer and simultaneously a *push* on the from buffer, this guarantees that elements will be shuttled from the front of the rear buffer to the rear of the front buffer, even though liveness is not specified for *push* for any individual buffer.

$\langle 1 \rangle 1.\ \exists\, \mathit{Buffer}1, \mathit{Buffer}2 \,:\, \mathit{DB}(N).\mathit{Spec} \Rightarrow \exists\, \mathit{Buffer} \,:\, \mathit{CB}(2N).\mathit{Spec}$

LET: $\overline{\mathit{Buffer}} \;\triangleq\; \mathit{Buffer}1 \circ \mathit{Buffer}2$

The first step is showing that the initial conditions of the specification are satisfied by the initial conditions of the implementation.

$\langle 2 \rangle 1.\ \mathit{DB.Init} \Rightarrow \overline{\mathit{CB}(2N).\mathit{Init}}$

PROOF: $\mathit{DB.Init} \;\triangleq\; \land\, \forall\, n \in 1\,..\,N \,:\, \mathit{Buffer}1[n] = \bot$
$\qquad\qquad\qquad\qquad\quad \land\, \forall\, n \in 1\,..\,N \,:\, \mathit{Buffer}2[n] = \bot$

$\overline{\mathit{CB.Init}} \;\triangleq\; \forall\, n \in 1\,..\,2N \,:\, \overline{\mathit{Buffer}}[n] = \bot$

The proof follows immediately from the definitions of $\mathit{Buffer}1 \circ \mathit{Buffer}2$ and $\overline{\mathit{Buffer}}$. □

The invariant is that every element in all of the buffer arrays is either an element of the set *Data* or is $\perp$. We now prove the invariant. It's messy to write but trivial to prove. I believe it's marginally easier to survey this proof when the invariant is written explicitly, and the formulae are not so complicated as to force explicit definition at this stage.

LET: k : CONSTANT

$\langle 2\rangle 2.$ $DB.Init \wedge \Box[\exists\, a \,:\, DB.push(a) \vee DB.pop \vee \exists\, k \,:\, DB.move(k)]_{Buffer1, Buffer2}$
$\Rightarrow$
$\Box(k \in 1\,..\,N \Rightarrow \wedge\ (Buffer1[k] \in Data \vee Buffer1[k] = \perp)$
$\qquad\qquad\qquad\quad \wedge\ (Buffer2[k] \in Data \vee Buffer2[k] = \perp))$

PROOF:

$\langle 3\rangle 1.$ $DB.Init \Rightarrow$
$(k \in 1\,..\,N \Rightarrow \wedge\ (Buffer1[k] \in Data \vee Buffer1[k] = \perp)$
$\qquad\qquad\qquad \wedge\ (Buffer2[k] \in Data \vee Buffer2[k] = \perp))$

PROOF:

$\langle 4\rangle 1.$ $DB.Init \Rightarrow (k \in 1\,..\,N \Rightarrow (Buffer1[k] = \perp \wedge Buffer2[k] = \perp))$

PROOF: Follows immediately from the definition of $DB.Init$. $\Box$

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

PROOF: Follows immediately by propositional logic from $\langle 4\rangle 1$. $\Box$

$\langle 3\rangle 2.$ $\wedge\ (k \in 1\,..\,N \Rightarrow \wedge\ (Buffer1[k] \in Data \vee Buffer1[k] = \perp)$
$\qquad\qquad\qquad \wedge\ (Buffer2[k] \in Data \vee Buffer2[k] = \perp))$
$\wedge\ \Box[\exists\, a \,:\, DB.push(a) \vee DB.pop \vee \exists\, k \,:\, DB.move(k)]_{Buffer1, Buffer2}$
$\Rightarrow$
$\Box(k \in 1\,..\,N \Rightarrow \wedge\ (Buffer1[k] \in Data \vee Buffer1[k] = \perp)$
$\qquad\qquad\qquad \wedge\ (Buffer2[k] \in Data \vee Buffer2[k] = \perp))$

PROOF:

$\langle 4\rangle 1.$ $\wedge\ (k \in 1\,..\,N \Rightarrow \wedge\ (Buffer1[k] \in Data \vee Buffer1[k] = \perp)$
$\qquad\qquad\qquad \wedge\ (Buffer2[k] \in Data \vee Buffer2[k] = \perp))$
$\wedge\ [\exists\, a \,:\, DB.push(a) \vee DB.pop \vee \exists\, k \,:\, DB.move(k)]_{Buffer1, Buffer2}$
$\Rightarrow$
$(k \in 1\,..\,N \Rightarrow \wedge\ (Buffer1'[k] \in Data \vee Buffer1'[k] = \perp)$
$\qquad\qquad\qquad \wedge\ (Buffer2'[k] \in Data \vee Buffer2'[k] = \perp))$

PROOF:

$\langle 5\rangle 1.$ $\wedge\ (k \in 1\,..\,N \Rightarrow \wedge\ (Buffer1[k] \in Data \vee Buffer1[k] = \perp)$
$\qquad\qquad\qquad \wedge\ (Buffer2[k] \in Data \vee Buffer2[k] = \perp))$
$\wedge\ \exists\, a \,:\, DB.push(a)$
$\Rightarrow$
$(k \in 1\,..\,N \Rightarrow \wedge\ (Buffer1'[k] \in Data \vee Buffer1'[k] = \perp)$
$\qquad\qquad\qquad \wedge\ (Buffer2'[k] \in Data \vee Buffer2'[k] = \perp))$

PROOF:

LET: a : CONSTANT

$\langle 6\rangle 1.$ $\wedge\ (k \in 1\,..\,N \Rightarrow \wedge\ (Buffer1[k] \in Data \vee Buffer1[k] = \perp)$
$\qquad\qquad\qquad \wedge\ (Buffer2[k] \in Data \vee Buffer2[k] = \perp))$
$\wedge\ DB.push(a)$
$\Rightarrow$

$$(k \in 1 \,..\, N \Rightarrow \wedge\ (Buffer1'[k] \in Data \vee Buffer1'[k] = \perp) \\ \wedge\ (Buffer2'[k] \in Data \vee Buffer2'[k] = \perp))$$

PROOF:

$\langle 7 \rangle 1.$ $k \in 1 \,..(N-1) \wedge DB.push(a) \Rightarrow$

$Buffer1'[k] = Buffer1[k] \wedge Buffer2'[k] = Buffer2[k]$

PROOF: Follows immediately from the definition of $DB.push(a)$. □

$\langle 7 \rangle 2.$ $DB.push(a) \Rightarrow Buffer1'[N] = Buffer1[N]$

PROOF: Follows immediately from the definition of $DB.push(a)$. □

$\langle 7 \rangle 3.$ $DB.push(a) \Rightarrow Buffer2'[N] \in Data$

PROOF: Follows immediately from the definition of $DB.push(a)$. □

$\langle 7 \rangle 4.$ Q.E.D.

PROOF: Follows by cases and elementary equational reasoning from $\langle 7 \rangle 1$, $\langle 7 \rangle 2$ and $\langle 7 \rangle 3$. □

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

PROOF: Follows immediately by existential quantification over a from $\langle 6 \rangle 1$. □

$\langle 5 \rangle 2.$ $\wedge\ (k \in 1 \,..\, N \Rightarrow \wedge\ (Buffer1[k] \in Data \vee Buffer1[k] = \perp)$
$\wedge\ (Buffer2[k] \in Data \vee Buffer2[k] = \perp))$
$\wedge\ DB.pop$
$\Rightarrow$
$(k \in 1 \,..\, N \Rightarrow \wedge\ (Buffer1'[k] \in Data \vee Buffer1'[k] = \perp)$
$\wedge\ (Buffer2'[k] \in Data \vee Buffer2'[k] = \perp))$

PROOF: Follows from the definition of $DB.pop$ using similar reasoning to that used for $DB.push(a)$. □

$\langle 5 \rangle 3.$ $\wedge\ (k \in 1 \,..\, N \Rightarrow \wedge\ (Buffer1[k] \in Data \vee Buffer1[k] = \perp)$
$\wedge\ (Buffer2[k] \in Data \vee Buffer2[k] = \perp))$
$\wedge\ \exists j : DB.move(j)$
$\Rightarrow$
$(k \in 1 \,..\, N \Rightarrow \wedge\ (Buffer1'[k] \in Data \vee Buffer1'[k] = \perp)$
$\wedge\ (Buffer2'[k] \in Data \vee Buffer2'[k] = \perp))$

LET: j : CONSTANT

$\langle 6 \rangle 1.$ $\wedge\ (k \in 1 \,..\, N \Rightarrow \wedge\ (Buffer1[k] \in Data \vee Buffer1[k] = \perp)$
$\wedge\ (Buffer2[k] \in Data \vee Buffer2[k] = \perp))$
$\wedge\ DB.move(j)$
$\Rightarrow$
$(k \in 1 \,..\, N \Rightarrow \wedge\ (Buffer1'[k] \in Data \vee Buffer1'[k] = \perp)$
$\wedge\ (Buffer2'[k] \in Data \vee Buffer2'[k] = \perp))$

PROOF: Follows similarly to the case for $DB.push(a)$ by division into cases $j = k, j = (k-1), j \notin \{k, k-1\}$ and using propositional logic on the definition of $DB.move$. □

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

PROOF: Follows immediately from $\langle 6 \rangle 1$ by existential quantification over j in $DB.move(j)$. □

$\langle 5\rangle 4.\ \wedge (k \in 1\,..\,N \Rightarrow \wedge (Buffer1[k] \in Data \vee Buffer1[k] = \bot)$
$\qquad\qquad\qquad\qquad\quad \wedge (Buffer2[k] \in Data \vee Buffer2[k] = \bot))$
$\quad \wedge\ Buffer1' = Buffer1$
$\quad \wedge\ Buffer2' = Buffer2$
$\quad \Rightarrow$
$\qquad (k \in 1\,..\,N \Rightarrow \wedge (Buffer1'[k] \in Data \vee Buffer1'[k] = \bot)$
$\qquad\qquad\qquad\qquad \wedge (Buffer2'[k] \in Data \vee Buffer2'[k] = \bot))$

PROOF: Follows immediately by substitution. □

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

PROOF: Follows immediately by propositional logic from $\langle 5\rangle 1$, $\langle 5\rangle 2$, $\langle 5\rangle 3$ and $\langle 5\rangle 4$. □

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

PROOF: Immediate from $\langle 4\rangle 1$ by Rule INV1. □

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

PROOF: Follows immediately from $\langle 3\rangle 1$ and $\langle 3\rangle 2$ by propositional logic. □

The safety property of the specification follows from the safety property of the implementation using the invariant. In this proof, we explicitly define *Inv* to be the invariant, which makes the layout of the proof more manageable. Safety follows by step simulation. There are no stuttering steps in this case – every step of the double buffer is a corresponding step of the double-size single buffer.

$\langle 2\rangle 3.\ \wedge DB.Init$
$\quad \wedge\ \Box[\vee\ \exists\, a : DB.push(a)$
$\qquad\quad \vee\ DB.pop$
$\qquad\quad \vee\ \exists\, k : DB.move(k)]_{Buffer1, Buffer2}$
$\quad \Rightarrow$
$\quad \Box[\vee\ \overline{\exists\, a : CB(2N).push(a)}$
$\qquad \vee\ \overline{CB(2N).pop}$
$\qquad \vee\ \overline{\exists\, k : CB(2N).move(k)}]_{\overline{Buffer}}$

PROOF:

LET: k : CONSTANT

$\quad Inv \triangleq (k \in 1\,..\,N \Rightarrow \wedge (Buffer1[k] \in Data \vee Buffer1[k] = \bot)$
$\qquad\qquad\qquad\qquad\qquad \wedge (Buffer2[k] \in Data \vee Buffer2[k] = \bot))$

$\langle 3\rangle 1.\ \wedge\ \Box Inv$
$\quad \wedge\ \Box[\vee\ \exists\, a : DB.push(a)$
$\qquad\quad \vee\ DB.pop$
$\qquad\quad \vee\ \exists\, k : DB.move(k)]_{Buffer1, Buffer2}$
$\quad \Rightarrow$
$\quad \Box[\vee\ \overline{\exists\, a : CB(2N).push(a)}$
$\qquad \vee\ \overline{CB(2N).pop}$
$\qquad \vee\ \overline{\exists\, k : CB(2N).move(k)}]_{\overline{Buffer}}$

PROOF:

$\langle 4\rangle 1.$ $\land \Box Inv$
$\land \Box[\lor \exists a : DB.push(a)$
$\lor DB.pop$
$\lor \exists k : DB.move(k)]_{Buffer1, Buffer2}$
$\Rightarrow$
$\Box[\land \lor \exists a : DB.push(a)$
$\lor DB.pop$
$\lor \exists k : DB.move(k)$
$\land Inv$
$\land Inv']_{Buffer1, Buffer2}$

PROOF: Immediate from INV2 using propositional logic. □

$\langle 4\rangle 2.$ $\Box[\land \lor \exists a : DB.push(a)$
$\lor DB.pop$
$\lor \exists k : DB.move(k)$
$\land Inv$
$\land Inv']_{Buffer1, Buffer2}$
$\Rightarrow$
$\Box[\lor \overline{\exists a : CB(2N).push(a)}$
$\lor \overline{CB(2N).pop}$
$\lor \overline{\exists k : CB(2N).move(k)}]_{\overline{Buffer}}$

PROOF:

$\langle 5\rangle 1.$ $[\land \lor \exists a : DB.push(a)$
$\lor DB.pop$
$\lor \exists k : DB.move(k))$
$\land Inv$
$\land Inv']_{Buffer1, Buffer2}$
$\Rightarrow$
$[\lor \overline{\exists a : CB(2N).push(a)}$
$\lor \overline{CB(2N).pop}$
$\lor \overline{\exists k : CB(2N).move(k)}]_{\overline{Buffer}}$

PROOF:

$\langle 6\rangle 1.$ $\exists a : DB.push(a) \land Inv \land Inv'$
$\Rightarrow$
$\overline{\exists a : CB(2N).push(a)}$

PROOF:

LET: a : CONSTANT

$\langle 7\rangle 1.$ $DB.push(a)$
$\Rightarrow$
$\overline{CB(2N).push(a)}$

PROOF: Follows from substitution $\overline{Buffer}[2N] = Buffer2[N]$. □

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

PROOF: Follows immediately by existentially quantifying over a and adjoining Inv and Inv' to the hypothesis. □

$\langle 6\rangle 2.$ $DB.pop \land Inv \land Inv'$

$$\frac{\Rightarrow}{\overline{CB(2N).pop}}$$

PROOF: Follows similarly to $\langle 6\rangle 1$. $\square$

$\langle 6\rangle 3.$ $\exists\, k\ :\ DB.move(k) \wedge Inv \wedge Inv'$
$$\frac{\Rightarrow}{\overline{\exists\, k\ :\ CB(2N).move(k)}}$$

PROOF:

The variable k is already declared. We want the free k in Inv to be instantiated to the variable in $move$ when the quantifier $\exists\, k\ :$ is removed. The reader may convince herself that this is legitimate predicate logic. We note it here to avoid momentary confusion.

$\langle 7\rangle 1.$ $DB.move(k) \wedge Inv \wedge Inv'$
$$\frac{\Rightarrow}{\overline{CB(2N).move(k)}}$$

PROOF: The proof is broken into components, corresponding to the antecedents of the conjuncts of $DB.move$. A full proof would contain a little propositional reasoning but is otherwise similar to $\langle 6\rangle 1$. Some care is needed in checking $move(N+1)$, where the invariant from $\langle 2\rangle 2$ is used.

$\langle 8\rangle 1.$ $\wedge \vee k \in 2..N$
$\quad\quad \vee\, k \in (N+2)..2N$
$\quad \wedge\, DB.move(k) \wedge Inv \wedge Inv'$
$$\frac{\Rightarrow}{\overline{CB(2N).move(k)}}$$

PROOF: Follows similarly to $\langle 6\rangle 1$. $\square$

$\langle 8\rangle 2.$ $DB.move(N+1) \wedge Inv \wedge Inv'$
$$\frac{\Rightarrow}{\overline{CB(2N).move(N+1)}}$$

PROOF: Inv is needed here to ensure that the popped value is actually in $Data$ so that the hypothesis and therefore the result of the push follows.

$\langle 9\rangle 1.$ $\wedge\, DB.B1.push(Buffer2[1])$
$\quad \wedge\, DB.B2.pop$
$$\begin{aligned}\equiv\ &\wedge Buffer2[1] \in Data\\ &\wedge \overline{Buffer}[N] = \bot\\ &\wedge \overline{Buffer}'[N] = Buffer2[1]\\ &\wedge \overline{Buffer}[N+1] = Buffer2[1]\\ &\wedge \overline{Buffer}'[N+1] = \bot\\ &\wedge \forall\, i \in 1\,..(N-1)\ :\ \text{UNCHANGED}\ \overline{Buffer[i]}\\ &\wedge \forall\, i \in (N+2)\,..\,2N\ :\ \text{UNCHANGED}\ \overline{Buffer[i]}\end{aligned}$$

PROOF: Follows immediately from the definitions of $\overline{Buffer}$,

$Buffer2$, $DB.B1.push$ and $DB.B2.pop$. □

$\langle 9\rangle 2$. Q.E.D.

PROOF: Follows immediately using the right-hand-side of the equivalence in $\langle 9\rangle 1$, Inv and Inv' from the definition of $DB.move(N+1)$ by substitution. □

$\langle 8\rangle 3$. Q.E.D.

PROOF: Follows immediately from cases $\langle 8\rangle 1$ and $\langle 8\rangle 2$ by propositional logic. □

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

PROOF: Follows immediately from $\langle 7\rangle 1$ by propositional logic and by predicate logic quantification over the parameter of $move$. □

$\langle 6\rangle 4$. $Buffer1' = Buffer1 \wedge Buffer2' = Buffer2$

$$\Rightarrow$$

$$\overline{Buffer}' = \overline{Buffer}$$

PROOF: Immediate from the definition of $Buffer$. □

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

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

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

PROOF: Immediate from $\langle 5\rangle 1$ by STL4. □

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

PROOF: Follows immediately by propositional logic from $\langle 4\rangle 1$ and $\langle 4\rangle 2$. □

$\langle 3\rangle 2$. Q.E.D.

PROOF: Follows immediately by propositional logic from $\langle 2\rangle 2$ and $\langle 3\rangle 1$. □

Liveness for this implementation is straightforward. The liveness for the actions of CB follow directly from the liveness of the corresponding actions of DB. Thus, a simplified version of proof rule WF2 may be used.

$\langle 2\rangle 4$.
$$\begin{aligned}&\wedge\ DB.Init\\ &\wedge\ \Box[\vee\ \exists\, a\ :\ DB.push(a)\\ &\qquad \vee\ DB.pop\\ &\qquad \vee\ \exists\, k\ :\ DB.move(k)]_{Buffer1, Buffer2}\\ &\wedge\ L_1\\ &\quad\Rightarrow\\ &\quad\overline{L_2}\end{aligned}$$

PROOF:

For direct step-simulation, the liveness proof is considerably simplified. The next step is the simplified version of the TLA proof rule WF2 for the case of direct step simulation.

$\langle 3\rangle 1$. $\Box Inv \wedge L_1 \Rightarrow \overline{L_2}$

PROOF:

LET: $\mathcal{B}$: PROPOSITIONAL VARIABLE
$\mathcal{M}$: PROPOSITIONAL VARIABLE
$\mathcal{A}$: PROPOSITIONAL VARIABLE

$\langle 4 \rangle 1.$
$$\frac{\begin{array}{c}\langle \mathcal{B} \rangle_f \;\Rightarrow\; \overline{\langle \mathcal{M} \rangle_g} \\ \mathit{Inv} \wedge \overline{\mathit{Enabled}\ \langle \mathcal{M} \rangle_g} \;\Rightarrow\; \mathit{Enabled}\ \langle \mathcal{B} \rangle_f\end{array}}{\mathrm{WF}_f(\mathcal{A}) \wedge \Box \mathit{Inv} \;\Rightarrow\; \overline{\mathrm{WF}_g(\mathcal{M})}}$$

PROOF: This is an instantiation of proof rule WF2 for direct step simulation. The instantiation is:

$P \;\triangleq\; \mathit{Inv},$
$\mathcal{B} \;\triangleq\; \mathcal{A},$
$\mathcal{N} \;\triangleq\;$ TRUE,
$F \;\triangleq\; \mathit{Inv}.$

The second premise follows by propositional logic since $\mathcal{B} \;\triangleq\; \mathcal{A}$, the antecedent of the third premise is simplified, and the consequent of the fourth premise becomes $\Diamond\Box \mathit{Inv}$, thus the fourth premise follows from the contrapositive $\vdash F \;\Rightarrow\; \Diamond F$ of STL 2 and may be omitted. $\Box$

$\langle 4 \rangle 2.$ $\wedge\ \mathrm{WF}_{\mathit{Buffer1},\mathit{Buffer2}}(\exists\, a\ :\ \mathit{DB.push}(a))$
$\wedge\ \Box \mathit{Inv}$
$\Rightarrow$
$\overline{\mathrm{WF}_{\mathit{Buffer}}(\exists\, a\ :\ \mathit{CB.push}(a))}$

PROOF: This step is an instantiation of the conclusion of $\langle 4 \rangle 1$. Thus the substeps are instantiations of the premises of $\langle 4 \rangle 1$.

$\langle 5 \rangle 1.$ $\langle \exists\, a\ :\ \mathit{DB.push}(a) \rangle_{\mathit{Buffer1},\mathit{Buffer2}}$
$\Rightarrow$
$\langle \overline{\exists\, a\ :\ \mathit{CB.push}(a)} \rangle_{\overline{\mathit{Buffer}}}$

PROOF: This has been shown in the safety step $\langle 2 \rangle 3$, modulo trivial propositional logic. $\Box$

$\langle 5 \rangle 2.$ $\wedge\ \mathit{Inv}$
$\wedge\ \overline{\mathit{Enabled}\langle \exists\, a\ :\ \mathit{CB.push}(a) \rangle_{\mathit{Buffer}}}$
$\Rightarrow$
$\mathit{Enabled}\langle \exists\, a\ :\ \mathit{DB.push}(a) \rangle_{\mathit{Buffer1},\mathit{Buffer2}}$

PROOF:

$\langle 6 \rangle 1.$ $\overline{\mathit{Enabled}\langle \exists\, a\ :\ \mathit{CB.push}(a) \rangle_{\mathit{Buffer}}}$
$\Rightarrow$
$\mathit{Enabled}\langle \exists\, a\ :\ \mathit{DB.push}(a) \rangle_{\mathit{Buffer1},\mathit{Buffer2}}$

PROOF:

$\langle 7 \rangle 1.\ \wedge\ \overline{Enabled \langle \exists\, a\, :\, CB.push(a) \rangle_{Buffer}}$

$$\equiv$$

$$\exists\, c, a\, :\, \wedge\ a \in Data$$
$$\wedge\ \overline{Buffer}[2N] = \bot$$
$$\wedge\ c[2N] = a$$
$$\wedge\ \forall\, i \in 1 \,..\, (2N-1)\, :\, c[i] = \overline{Buffer}[i]$$
$$\wedge\ c \neq \overline{Buffer}$$

$$\equiv$$

$$\exists\, a\, :\, \wedge\ a \in Data$$
$$\wedge\ Buffer2[N] = \bot$$

$\wedge\ Enabled \langle \exists\, a\, :\, DB.push(a) \rangle_{Buffer1, Buffer2}$

$$\equiv$$

$$\exists\, c, d, a\, :\, \wedge\ a \in Data$$
$$\wedge\ Buffer2[N] = \bot$$
$$\wedge\ c[N] = a$$
$$\wedge\ \forall\, i \in 1 \,..\, (N-1)\, :\, c[i] = Buffer2[i]$$
$$\wedge\ \forall\, i \in 1 \,..\, N\, :\, d[i] = Buffer1[i]$$
$$\wedge\ c \neq Buffer1$$
$$\wedge\ d \neq Buffer2$$

$$\equiv$$

$$\exists\, a\, :\, \wedge\ a \in Data$$
$$\wedge\ Buffer2[N] = \bot$$

PROOF: The first equivalences are the definitions of the *Enabled* formulas, which are the definitions of the actions plus the added condition defined by the $\langle\,\rangle_f$ construct, with the primed variables quantified out. The second equivalences are the formulas with the quantifiers passed down through the conjuncts by predicate logic, and then eliminating truths concerning the TLA primitives, namely the data structures, which follow from elementary set theory. □

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

PROOF: Follows immediately from the second equivalences in $\langle 7 \rangle 1$. □

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

PROOF: Follows immediately by adjoining *Inv* to the hypothesis. □

$\langle 5 \rangle 3.$ Q.E.D.

PROOF: Immediate from $\langle 5 \rangle 1$ and $\langle 5 \rangle 2$ using the proof rule in $\langle 4 \rangle 1$. □

$\langle 4 \rangle 3.\ \wedge\ \mathrm{WF}_{Buffer1, Buffer2}(DB.move(k))$

$\wedge\ \Box Inv$

$\Rightarrow$

$\mathrm{WF}_{\overline{Buffer}}(\overline{CB.move(k)})$

PROOF: Follows similarly to $\langle 4 \rangle 2$, but requires some propositional logic with the definition of $DB.move$. □

$\langle 4 \rangle 4.\ \wedge\ \mathrm{WF}_{Buffer1, Buffer2}(DB.pop)$

$\wedge\ \Box Inv$

$\Rightarrow$
$\mathrm{WF}_{\overline{Buffer}}(\overline{CB.pop})$

PROOF: Similar to $\langle 4\rangle 2$. □

$\langle 4\rangle 5$. Q.E.D.

PROOF: Follows immediately by quantification and propositional logic from $\langle 4\rangle 2$, $\langle 4\rangle 3$ and $\langle 4\rangle 4$. □

$\langle 3\rangle 2$. Q.E.D.

PROOF: Follows immediately by propositional logic from $\langle 2\rangle 2$ and $\langle 3\rangle 1$. □

$\langle 2\rangle 5$. $Init_1 \wedge \Box[N_1]_v \wedge L_1$
$\Rightarrow$
$\overline{Init_2} \wedge \Box[\overline{N_2}]_{\overline{w}} \wedge \overline{L_2}$

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

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

PROOF: Theorem follows immediately from $\langle 2\rangle 5$ by quantifying over the free variables. □

The second proof shows that a concrete *Buffer* implements an abstract *buffer*. The refinement mapping is not as trivial as it was in the first proof. The abstract buffer is the selection of non-$\perp$ elements from the concrete buffer. The safety step is straightforward. *CB.move* is shown to be a stuttering step of *AbstractBuffer*. The main part of the proof concerns the proof of liveness, because there are stuttering variables – the one we use is a state function called *Descending*. That means that the Lattice Rule has to be used to obtain the postcondition that holds after the stuttering variable has counted down all the way.

$\langle 1\rangle 2$. $\exists\, Buffer : CB(N).Spec \Rightarrow \exists\, buffer : AB(N).Spec$

LET: $NonVoid(k) \triangleq k \neq \perp$
$\overline{buffer} \triangleq SelectSeq(Buffer, NonVoid)$
$FirstFull \triangleq Buffer[1] \neq \perp$
$NotEmpty \triangleq \exists\, i \in 1\,..\,N : Buffer[i] \neq \perp$
$NotStuffed \triangleq \exists\, i \in 1\,..(N-1) : Buffer[i] = \perp$
$Descending(Buffer) \triangleq \sum_{i=1}^{N}(Buffer[i] = \perp).2^{N-i}$
$MaxDescending \triangleq \sum_{i=1}^{N} 2^{N-i}$

The refinement mapping is given above. We use Theorem 1 to reduce the proof, and we choose the invariant **True**. Two of the clauses of Theorem 1, namely $Init \Rightarrow Inv$ and $Inv \wedge [N_1]_v \Rightarrow Inv'$ are then trivial. We prove $\langle 1\rangle 2$ by proving the other three clauses of Theorem 1.

In the *Safety* part of the proof below, we make use of a temporal-logical truth, that $\Box\exists\, i \in 0\,..\, \mathit{Maxdescending} \,:\, \mathit{Descending}(\mathit{Buffer}) = i$. One option is to use this as an invariant. We choose not to do so, both because we prefer the stronger form, because it allows us to omit two proof steps (as above), and it allows the proof to maintain hierarchical structure without duplication or cross-referencing of proof steps. We explain later in more detail.

Note that $\mathit{FirstFull} \equiv \mathit{NonVoid}(\mathit{Buffer}[1])$ and $\mathit{NotEmpty} \equiv \exists\, i \in 1\,..\,N \,:\, \mathit{NonVoid}(\mathit{Buffer}[i])$. These may figure in proof justifications, but are not given as individual proof steps.

Firstly, the concrete initial state implies the abstract initial state.

$\langle 2\rangle 1.$ $\mathit{CB.Init} \Rightarrow \overline{\mathit{AB.Init}}$
PROOF:
$\langle 3\rangle 1.$ $\mathit{SelectSeq}([i \in 1\,..\,N \mapsto \{\bot\}], \mathit{NonVoid}) = \langle\,\rangle$
PROOF: Immediate from the definitions. $\Box$
$\langle 3\rangle 2.$ Q.E.D.
PROOF: Follows immediately from $\langle 3\rangle 1$ and the definitions. $\Box$

Secondly, the concrete safety property implies the abstract safety property. A *CB.move* operation is a stuttering step for the abstract buffer.

$\langle 2\rangle 2.$ $[\vee\ \exists\, a \,:\, \mathit{CB.push}(a)$
$\vee\ \mathit{CB.pop}$
$\vee\ \exists\, k \,:\, \mathit{CB.move}(k)]_{\mathit{Buffer}}$
$\Rightarrow$
$[\vee\ \overline{\exists\, a \,:\, \mathit{AB.push}(a)}$
$\vee\ \overline{\mathit{AB.pop}}]_{\overline{\mathit{buffer}}}$
PROOF:

First, a concrete *push* is an abstract *push*.

$\langle 3\rangle 1.$ $\exists\, a \,:\, \mathit{CB.push}(a)$
$\Rightarrow$
$\overline{\exists\, a \,:\, \mathit{AB.push}(a)}$
PROOF:
LET: a : CONSTANT
$\langle 4\rangle 1.$ $\mathit{CB.push}(a)$
$\Rightarrow$
$\overline{\mathit{AB.push}(a)}$
PROOF:
$\langle 5\rangle 1.$ $\mathit{CB.push}(a) \Rightarrow a \in \mathit{Data}$
PROOF: Immediate from the definition of *CB.push*. $\Box$
$\langle 5\rangle 2.$ $\mathit{CB.push}(a) \Rightarrow \mathit{Len}(\overline{\mathit{buffer}}) < N$

PROOF:

$\langle 6 \rangle 1.$ $CB.push(a) \Rightarrow Buffer[N] = \bot$

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

$\langle 6 \rangle 2.$ $Buffer[N] = \bot \Rightarrow Len(SelectSeq(Buffer, NonVoid)) < N$

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

$\langle 6 \rangle 3.$ Q.E.D.

PROOF: Follows from $\langle 6 \rangle 1$, $\langle 6 \rangle 2$ and the definition of $\overline{buffer}$ by propositional logic. □

$\langle 5 \rangle 3.$ $CB.push(a) \Rightarrow \overline{buffer}' = \overline{buffer} \circ \langle a \rangle$

PROOF:

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

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

PROOF:

$\langle 7 \rangle 1.$ $CB.push(a) \Rightarrow Buffer[N] = \bot$

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

$\langle 7 \rangle 2.$ $Buffer[N] = \bot$

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

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

$\langle 7 \rangle 3.$ Q.E.D.

PROOF: Follows immediately from $\langle 7 \rangle 1$ and $\langle 7 \rangle 2$ by propositional logic. □

$\langle 6 \rangle 2.$ $CB.push(a)$

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

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

$\langle 6 \rangle 3.$ $SelectSeq([i \in 1 \,..(N-1) \mapsto Buffer[i]] \circ \langle a \rangle, NonVoid) =$
$SelectSeq([i \in 1 \,..(N-1) \mapsto Buffer[i]], NonVoid) \circ \langle a \rangle$

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

$\langle 6 \rangle 4.$ Q.E.D.

PROOF: Follows immediately from $\langle 6 \rangle 1$, $\langle 6 \rangle 2$, $\langle 6 \rangle 3$ by propositional logic, substitution, and the definition of $\overline{buffer}$. □

$\langle 5 \rangle 4.$ Q.E.D.

PROOF: Follows immediately from $\langle 5 \rangle 1$, $\langle 5 \rangle 2$ and $\langle 5 \rangle 3$ using propositional logic. □

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

PROOF: Follows immediately by existentially quantifying over a. □

Second, a concrete *pop* is an abstract *pop*.

$\langle 3\rangle 2.$ $CB.pop$
$$\Rightarrow$$
$$\overline{AB.pop}$$
PROOF: Follows similarly to $\langle 3\rangle 1$. $\square$

Third, a concrete *move* is an abstract stuttering step – it doesn't change any of the values of the abstract variables.

$\langle 3\rangle 3.$ $\exists\, k \,:\, CB.move(k)$
$$\Rightarrow$$
$$(\overline{buffer}' = \overline{buffer})$$
PROOF:
LET: k : CONSTANT
$\langle 4\rangle 1.$ $CB.move(k)$
$$\Rightarrow$$
$$(\overline{buffer}' = \overline{buffer})$$
PROOF:

In the following, we want to write $SelectSeq([i \in 1\,..\,N \mapsto Buffer[i]], NonVoid)$ as the composition of three sequences, consisting respectively of the first $k-2$ elements, the $k-1$'th and k'th, and the tail $N-k$ elements, and similarly with $SelectSeq([idd \,:\, N \mapsto Buffer'[i]], NonVoid)$, to show that they're the same, by showing for the three individual components of the sequence that they're the same. Since sequences always have as domain an initial segment of the natural numbers (namely $1\,..\,Len(S)$ for a sequence S), some rewriting on the indices has to be performed to make this turn out correctly.

$\langle 5\rangle 1.$ $CB.move(k)$
$$\Rightarrow$$
$$\begin{aligned}&\wedge\ SelectSeq([i \in 1\,..(k-2) \mapsto Buffer'[i]], NonVoid) = \\ &\quad\ SelectSeq([i \in 1\,..(k-2) \mapsto Buffer[i]], NonVoid) \\ &\wedge\ SelectSeq([i \in 1\,..(N-k) \mapsto Buffer'[i+k]], NonVoid) = \\ &\quad\ SelectSeq([i \in 1\,..(N-k) \mapsto Buffer[i+k]], NonVoid)\end{aligned}$$
PROOF:
$\langle 6\rangle 1.$ $CB.move(k)$
$$\Rightarrow$$
$$\forall\, i \in 1\,..(k-2) \,:\, Buffer'[i] = Buffer[i]$$
PROOF: Follows immediately from the definition of $CB.move(k)$. $\square$
$\langle 6\rangle 2.$ $CB.move(k)$
$$\Rightarrow$$
$$\forall\, i \in (k+1)\,..\,N \,:\, Buffer'[i] = Buffer[i]$$

PROOF: Follows immediately from the definition of $CB.move(k)$. Note that if $k = N$ then $(k+1)\,..\,N = \emptyset$. □

⟨6⟩3. Q.E.D.

PROOF: Follows by substitution of $Buffer'$ for $Buffer$ from ⟨6⟩1 and ⟨6⟩2, by simple transformation of indices, and by data structure rules which say that two terms representing finite functions with equal values for all their arguments are equal. □

⟨5⟩2. $CB.move(k)$

$$\begin{array}{l} \Rightarrow \\ SelectSeq(\langle Buffer'[k-1], Buffer'[k] \rangle, NonVoid) = \\ SelectSeq(\langle Buffer[k-1], Buffer[k] \rangle, NonVoid) = \\ \langle a \rangle \end{array}$$

PROOF: Follows from the definition of $SelectSeq$ and $CB.move(k)$. □

⟨5⟩3. $k \in 2\,..\,N$

$$\begin{array}{l} \Rightarrow \\ \wedge\ SelectSeq(Buffer, NonVoid) = \\ \quad SelectSeq([i \in 1\,..(k-2) \mapsto Buffer[i]], NonVoid) \circ \\ \quad SelectSeq(\langle Buffer[k-1], Buffer[k] \rangle, NonVoid) \circ \\ \quad SelectSeq([i \in 1\,..\,N-k \mapsto Buffer[i+k]], NonVoid) \\ \wedge\ SelectSeq(Buffer', NonVoid) = \\ \quad SelectSeq([i \in 1\,..(k-2) \mapsto Buffer'[i]], NonVoid) \circ \\ \quad SelectSeq(\langle Buffer'[k-1], Buffer'[k] \rangle, NonVoid) \circ \\ \quad SelectSeq([i \in 1\,..\,N-k \mapsto Buffer'[i+k]], NonVoid) \end{array}$$

PROOF: Follows from the definitions of $\circ$ and $SelectSeq$. Note that if $k = N$ then $(k+1)\,..\,N = \emptyset$, $SelectSeq(\emptyset, P) = \langle\,\rangle$, and $\alpha \circ \langle\,\rangle = \alpha$ for any sequence α, all by definition. □

⟨5⟩4. Q.E.D.

PROOF: Follows from ⟨5⟩1, ⟨5⟩2 and ⟨5⟩3 and the definition of $\overline{buffer}$ by propositional logic and simple equational reasoning. □

⟨4⟩2. Q.E.D.

PROOF: Follows immediately from ⟨4⟩1, by quantification over k. □

⟨3⟩4. $Buffer' = Buffer$

$$\begin{array}{l} \Rightarrow \\ \overline{buffer}' = \overline{buffer} \end{array}$$

PROOF: Immediate from the definition of $\overline{buffer}$. □

⟨3⟩5. Q.E.D.

PROOF: Follows immediately from ⟨3⟩1, ⟨3⟩2, ⟨3⟩3 and ⟨3⟩4 by propositional logic. □

Thirdly, the concrete liveness property implies the abstract liveness property. Liveness of the abstract *pop* operation needs to be shown.
The abstract *pop* is enabled via the refinement mapping under conditions under

which the corresponding concrete operation is not enabled (namely that there's *something* in the buffer somewhere, even though it may not be in Buffer[1]). However, under these conditions, one of the the concrete *move* operations is enabled, as noted in Section 6. The enabling conditions for *move*s, along with weak fairness of each *move*, entail that *move* actions must occur. These actions lead eventually to the satisfaction of the enabling conditions for concrete *pop*, (i.e., there's something in Buffer[1]) which simulates abstract *pop*. So weak fairness of concrete *pop* ensures eventual execution and thus, along with the invariant, entails weak fairness of abstract *pop*.

The proof method is: one picks a state function which acts as a stuttering variable, i.e. it decreases during a stuttering step, and its minimal values correspond to the condition one wants to reach (in this case, the enabling condition for $CB.pop$). One uses WF1 to show that the state function satisfies these requirements. Then, the Lattice Rule allows one to conclude that if the state function has some, any, value, this leads eventually to the goal condition being reached. Some temporal logic is used to convert this assertion into the fourth premise of an application of WF2 (it is straightforward to show the other three premises of WF2).

The structure of the proof is: the high level is an application of WF2; the next lower level consists of its four premises. The fourth of these is the tricky one, justified by a conclusion of the Lattice Rule plus some temporal logic. The conclusion of the Lattice Rule is justified by the premise. The premise is justified by an application of WF1. The three premises of WF1 are shown directly.

In this proof, the premises of TLA rules WF1, WF2 and Lattice are given their exact syntactic form as needed for the rule, and substepping is used to perform simple logical manipulations, rather than combining them into one step. This makes the proof easier to survey.

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

PROOF:

This step is the conclusion of an application of WF2, which is justified by giving the four premises. The fourth premise is the most difficult to prove. The others are straightforward.

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

$\langle \overline{AB.pop} \rangle_{\overline{buffer}}$

PROOF:

$\langle 4 \rangle 1.$ $\land \lor \exists\, a \,:\, CB.push(a)$
$\quad\quad \lor\ CB.pop$
$\quad\quad \lor\ \exists\, k \,:\, CB.move(k))$
$\quad \land\ CB.pop$
$\quad\quad \equiv$
$\quad\quad CB.pop$

PROOF: Follows immediately by propositional logic from the definitions of $CB.push$, $CB.pop$ and $CB.move$. $\square$

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

PROOF: Recall that the statement means $CB.pop \land Buffer' \neq Buffer \Rightarrow \overline{AB.pop} \land \overline{buffer}' \neq \overline{buffer}$. Thus it follows, modulo a little propositional logic, from the analysis needed for step-simulation in $\langle 2 \rangle 2$ (which we didn't give explicitly, but noted it is similar to the *push* simulation). $\square$

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

PROOF: Follows immediately by proposition logic from $\langle 4 \rangle 1$ and $\langle 4 \rangle 2$. $\square$

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

PROOF:

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

PROOF: Follows immediately by propositional logic. $\square$

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

PROOF: Follows immediately from $\langle 4 \rangle 1$ by propositional logic. $\square$

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

PROOF:

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

PROOF:

$\langle 5 \rangle 1.$ $Enabled \langle\, CB.pop \,\rangle_{buffer}$

$\equiv$

$\exists\, d, c \;:\; \wedge\; d \neq \mathit{BufferBuffer}[1] \neq \bot$
$\qquad\qquad \wedge\; c[1] = \bot$
$\qquad\qquad \wedge\; \forall\, i \in 2\,..\,N \;:\; c[i] = \mathit{Buffer}[i]$

$\equiv$

$\mathit{Buffer}[1] \neq \bot$

PROOF: The first equivalence is the definition of *Enabled*. The second follows by moving the quantifiers inside the conjunct by a standard rule of predicate logic, and by observing that the first, third and fourth conjuncts follow by predicate logic from the primitives of TLA. □

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

PROOF: Follows immediately from the definition of *FirstFull* and the second equivalence of $\langle 5\rangle 1$. □

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

PROOF: Follows immediately by propositional logic from $\langle 4\rangle 1$. □

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

PROOF:

The Lattice Rule is used to prove this fourth hypothesis of WF2. First, some simplifications are needed.

$\langle 4\rangle 1$. $\wedge\; \vee\; \exists\, a \;:\; \mathit{CB.push}(a)$
$\qquad\quad \vee\; \mathit{CB.pop}$
$\qquad\quad \vee\; \exists\, k \;:\; \mathit{CB.move}(k)$
$\quad \wedge\; \neg \mathit{CB.pop}$
$\qquad \equiv$
$\qquad \vee\; \exists\, a \;:\; \mathit{CB.push}(a)$
$\qquad \vee\; \exists\, k \;:\; \mathit{CB.move}(k)$

PROOF: Follows immediately by propositional logic. □

$\langle 4\rangle 2$. $\mathit{NotEmpty} \equiv \overline{\mathit{Enabled}\langle \mathit{AB.pop}\rangle_{\mathit{buffer}}}$

PROOF:

$\langle 5\rangle 1$. $\overline{\mathit{Enabled}\langle \mathit{AB.pop}\rangle_{\mathit{buffer}}}$

$\equiv$

$$\begin{array}{l} \exists\, d, c \ : \ \land\ d \neq \mathit{SelectSeq}(\mathit{Buffer}, \mathit{NonVoid}) \\ \qquad\quad\ \land\ \mathit{Len}(\mathit{SelectSeq}(\mathit{Buffer}, \mathit{NonVoid}) > 0 \\ \qquad\quad\ \land\ c = \mathit{tail}(\mathit{Buffer}) \\ \equiv \\ \land\ \mathit{Len}(\mathit{SelectSeq}(\mathit{Buffer}, \mathit{NonVoid}) > 0 \\ \land\ \exists\, c \ : \ c = \mathit{tail}(\mathit{Buffer}) \end{array}$$

PROOF: The first equivalence is the definition of *Enabled*. The second follows by moving the quantifiers inside the conjunct by standard rules of predicate logic, and eliminating the first conjunct, which follows by predicate logic from the TLA requirement that any function symbol with any argument denotes, and predicate logic. □

⟨5⟩2. Q.E.D.

PROOF: Follows from the equivalence in ⟨5⟩1 using definitions of *SelectSeq*, *Len*, and *NonVoid*, by quantifier rules of predicate logic, and by observing that the second conjunct $\exists\, c \ : \ c = \mathit{tail}(\mathit{Buffer})$ is a truth of predicate logic. □

The next step is the simplified version of ⟨3⟩4, to be proved using the Lattice Rule, along with some manipulations in temporal logic.

⟨4⟩3. $$\begin{array}{l} \land\ \Box[\exists\, a \ : \ \mathit{CB.push}(a) \lor \exists\, k \ : \ \mathit{CB.move}(k)]_{\mathit{Buffer}} \\ \land\ \mathit{WF}_{\mathit{Buffer}}(\mathit{CB.pop}) \\ \land\ \mathit{WF}_{\mathit{Buffer}}(\exists\, k \ : \ \mathit{CB.move}(k)) \\ \land\ \Diamond\Box \mathit{NotEmpty} \\ \quad \Rightarrow \\ \quad \Diamond\Box \mathit{FirstFull} \end{array}$$

PROOF:

The substeps are a temporal logic proof rule, which I'll call the *Bridge Rule*, and instances of its two hypotheses. The Bridge Rule enables us to prove the fourth hypothesis of WF2 using the Lattice Rule. The first hypothesis of the instantiated Bridge Rule is a (counterfactual) invariant statement, that the condition intepreting P in WF2 is invariant under the restricted safety condition $\Box[\mathcal{N} \land \neg \mathcal{B}]_f$. The second hypothesis is (a modified form of) the conclusion of the Lattice Rule. The instantiated conclusion of the Bridge rule is step ⟨4⟩3 above.

We state the Bridge Rule using propositional variable X, which will be interpreted so that $\Box X$ is equivalent to

$$\begin{array}{l} \land\ \Box[\exists\, a \ : \ \mathit{CB.push}(a) \lor \exists\, k \ : \ \mathit{CB.move}(k)]_{\mathit{Buffer}} \\ \land\ \mathit{WF}_{\mathit{Buffer}}(\mathit{CB.pop}) \\ \land\ \mathit{WF}_{\mathit{Buffer}}(\exists\, k \ : \ \mathit{CB.move}(k)) \end{array}$$

(Weak Fairness conditions are equivalent to a statement of the form $\Box Y$ by the complementary form of STL6, and conjunctions of statements of the form $\Box Y$ are equivalent to a single statement of that form by STL5.) We will also use propositional variables NE, which will be interpreted as *NotEmpty*, and FF, which will be interpreted as *FirstFull*. However, the rule is valid for any interpretation of the propositional variables X, NE and FF.

In the TL calculation below, we use the TL rules

STL7	$\vdash \Box\Diamond\Box F \equiv \Diamond\Box F$
$\text{STL2}^{\Diamond}$	$\vdash F \Rightarrow \Diamond F$
$\text{STL4}^{\Diamond}$	$\dfrac{F \Rightarrow G}{\Diamond F \Rightarrow \Diamond G}$
$\text{STL6}^{\Diamond}$	$\vdash (\Box\Diamond F) \vee (\Box\Diamond G) \equiv \Box\Diamond(F \vee G)$

STL7 was included in an earlier version of TLA, but Lamport has omitted it since Bryan Olivier noted it followed from STL1-STL6. The rules $\text{STL2}^{\Diamond}$, $\text{STL4}^{\Diamond}$ and $\text{STL6}^{\Diamond}$ are the complements of STL2, STL4 and STL6 for $\Diamond$, and are easily derivable from those rules.

Additionally, there are some propositional logic rules that we use: namely, replacing a subformula by a provably logically equivalent subformula (called *substituting equivalent subformulas* below); and strengthening an antecedent of a conditional (called *strehngthening a hypothesis*).

LET: X, Y, FF, NE : PROPOSITIONAL VARIABLE

$\langle 5 \rangle 1.$ $\dfrac{\Box X \wedge FF \Rightarrow \Box FF \qquad \Box X \Rightarrow \Box(NE \Rightarrow \Diamond FF)}{\Box X \wedge \Diamond\Box NE \Rightarrow \Diamond\Box FF}$

PROOF:

$\langle 6 \rangle 1.$ $\dfrac{Z \wedge FF \Rightarrow \Box FF}{\Box Z \Rightarrow \Box(FF \Rightarrow \Box FF)}$

PROOF: Follows immediately by trivial propositional logic and then STL4. $\Box$

The following rule follows from STL2, STL4 and propositional logic. TL rules such as this may not be that easy to spot or to derive. Lamport suggests that any correct calculation in propositional temporal logic may be used where necessary in a formal TLA proof, which suggests a semantic short-cut using the completeness theorem: one may conclude that the rule is valid by observing that it's valid in all linearly ordered Kripke frames, that is to say, models in which states form a linear sequence. Then one may conclude by completeness that it's derivable, and say in the proof step, as we did for an earlier temporal

logic step, that it's 'derivable from PTL'.
The straightforward proof we give was noted by Lamport. No derivation of this proof rule has been found using only STL1–STL6 + Lattice Rule alone.

$\langle 6 \rangle 2.$ $\dfrac{\begin{array}{l}\Box X \Rightarrow \Box(NE \Rightarrow \Diamond FF) \\ \Box X \Rightarrow \Box(FF \Rightarrow \Box FF)\end{array}}{\Box X \Rightarrow \Box(NE \Rightarrow \Box\Diamond FF)}$

PROOF:

$\langle 7 \rangle 1.$ $\wedge\ \Box(NE \Rightarrow \Diamond FF)$
$\wedge\ \Box(FF \Rightarrow \Box FF)$
$\Rightarrow$
$\Box(NE \Rightarrow \Box\Diamond FF)$

PROOF:

$\langle 8 \rangle 1.$ $\wedge\ \Box(NE \Rightarrow \Diamond FF)$
$\wedge\ \Box(FF \Rightarrow \Box FF)$
$\Rightarrow$
$\wedge\ \Box(NE \Rightarrow \Diamond FF)$
$\wedge\ \Box(FF \Rightarrow \Diamond\Box FF)$

PROOF SKETCH: Uses STL4, STL2$^{\Diamond}$ and propositional logic. □

$\langle 8 \rangle 2.$ $\wedge\ \Box(NE \Rightarrow \Diamond FF)$
$\wedge\ \Box(FF \Rightarrow \Diamond\Box FF)$
$\equiv$
$\wedge\ (NE \leadsto FF)$
$\wedge\ (FF \leadsto \Box FF)$

PROOF: Follows immediately from the definition of $\leadsto$. □

$\langle 8 \rangle 3.$ $\wedge\ (NE \leadsto FF)$
$\wedge\ (FF \leadsto \Box FF)$
$\Rightarrow$
$NE \leadsto \Box FF$

PROOF SKETCH: Follows by transitivity of $\leadsto$. □

$\langle 8 \rangle 4.$ $NE \leadsto \Box FF \equiv \Box(NE \Rightarrow \Diamond\Box FF)$

PROOF: Follows immediately by the definition of $\leadsto$. □

$\langle 8 \rangle 5.$ $\Box(NE \Rightarrow \Diamond\Box FF) \Rightarrow \Box(NE \Rightarrow \Box\Diamond\Box FF)$

PROOF SKETCH: Follows by STL4 and STL7. □

$\langle 8 \rangle 6.$ $\Box(NE \Rightarrow \Box\Diamond\Box FF) \Rightarrow \Box(NE \Rightarrow \Box\Diamond FF)$

PROOF SKETCH: Follows by STL4 and STL2, using substitution of equivalent subformulas. □

$\langle 8 \rangle 7.$ Q.E.D.

PROOF: Immediate from $\langle 8 \rangle 1$, $\langle 8 \rangle 2$, $\langle 8 \rangle 3$, $\langle 8 \rangle 4$, $\langle 8 \rangle 5$, $\langle 8 \rangle 6$ by 'forward chaining'. □

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

PROOF: The conclusion of the rule follows from the premises using $\langle 7 \rangle 1$, Modus Ponens, and propositional logic. □

$\langle 6\rangle 3.$ $\dfrac{\Box X \Rightarrow \Box(NE \Rightarrow \Box Y)}{\Box X \wedge \Box NE \Rightarrow \Box Y}$

PROOF: This rule follows from PTL. $\Box$

$\langle 6\rangle 4.$ $\dfrac{\Box X \wedge \Box NE \Rightarrow \Box Y}{\Box(X \wedge NE) \Rightarrow \Box Y}$

PROOF: Immediate from STL5 and substitution of equivalent subformulas. $\Box$

$\langle 6\rangle 5.$ $\dfrac{\Box(X \wedge NE) \Rightarrow \Box Y)}{\Diamond\Box(X \wedge NE) \Rightarrow \Diamond\Box Y}$

PROOF: Immediate from STL4$^{\Diamond}$. $\Box$

$\langle 6\rangle 6.$ $\dfrac{\Diamond\Box(X \wedge NE) \Rightarrow \Diamond\Box Y)}{\Diamond\Box X \wedge \Diamond\Box NE \Rightarrow \Diamond\Box Y}$

PROOF: STL6 and substituting equivalent subformulas. $\Box$

$\langle 6\rangle 7.$ $\dfrac{\Diamond\Box X \wedge \Diamond\Box NE \Rightarrow \Diamond\Box Y}{\Box X \wedge \Diamond\Box NE \Rightarrow \Diamond\Box Y}$

PROOF: Strengthening a hypothesis, using STL2$^{\Diamond}$. $\Box$

$\langle 6\rangle 8.$ $\dfrac{\Box X \wedge \Diamond\Box NE \Rightarrow \Diamond\Box\Diamond FF}{\Box X \wedge \Diamond\Box NE \Rightarrow \Diamond\Box FF}$

PROOF: STL7, and substituting equivalent subformulas. $\Box$

$\langle 6\rangle 9.$ Q.E.D.

PROOF: The conclusion of rule $\langle 5\rangle 1$ follows from its hypotheses by forward-chaining the rules $\langle 6\rangle 1$, $\langle 6\rangle 2$, $\langle 6\rangle 3$, $\langle 6\rangle 4$, $\langle 6\rangle 5$, $\langle 6\rangle 6$, $\langle 6\rangle 7$, $\langle 6\rangle 8$, since inference is transitive, using the substitution
$Y \;\triangleq\; \Diamond FF$ where needed. $\Box$

We now prove the instantiated first hypothesis of the Bridge rule.

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

PROOF:

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

PROOF:

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

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. □

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

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

We now prove the instantiated second hypothesis of the Bridge Rule.

$\langle 5 \rangle 3.$ $\wedge\ \Box[\exists\, a : CB.push(a) \vee \exists\, k : CB.move(k)]_{Buffer}$
$\wedge\ WF_{Buffer}(CB.pop)$
$\wedge\ WF_{Buffer}(\exists\, k : CB.move(k))$
$\Rightarrow$
$\Box(NotEmpty \Rightarrow \Diamond FirstFull)$

PROOF:

First we show a temporal logic equivalence, used to change the consequent, then we prove the step with the altered consequent.

$\langle 6 \rangle 1.$ $(NotEmpty \wedge NotStuffed \Rightarrow \Diamond FirstFull)$
$\equiv$
$(NotEmpty \Rightarrow \Diamond FirstFull)$

PROOF:

$\langle 7 \rangle 1.$ $\neg NotStuffed \Rightarrow \Diamond FirstFull$

PROOF:

$\langle 8 \rangle 1.$ $\neg NotStuffed \Rightarrow FirstFull$

PROOF: Follows immediately from the definitions. □

$\langle 8 \rangle 2.$ $FirstFull \Rightarrow \Diamond FirstFull$

PROOF: Immediate from STL2. □

$\langle 8 \rangle 3.$ Q.E.D.

PROOF: Follows immediately from $\langle 8 \rangle 1$ and $\langle 8 \rangle 2$ by transitivity of $\Rightarrow$. □

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

PROOF: Follows by propositional logic from $\langle 7 \rangle 1$. □

$\langle 6 \rangle 2.$ $\wedge\ \Box[\exists\, a : CB.push(a) \vee \exists\, k : CB.move(k)]_{Buffer}$
$\wedge\ WF_{Buffer}(CB.pop)$
$\wedge\ WF_{Buffer}(\exists\, k : CB.move(k))$
$\Rightarrow$

$\Box(\mathit{NotEmpty} \wedge \mathit{NotStuffed} \Rightarrow \Diamond \mathit{FirstFull})$
PROOF:

Some axiomatic arithmetic is required in the next stage. No *Naturals* module has been given, but we assume that such a module would contain enough arithmetic to allow the simple reasoning that follows. This reasoning, from a reasonable set of axioms, is not very hard. We also don't believe that it would be particularly enlightening for our purposes in this paper to develop it here. However, it is important to note that the proof is correct only insofar as the arithmetic reasoning is correct. Arithmetic reasoning will be used where noted also in further steps below.

There is a question whether the following proof step should actually be formulated rather as an invariant. It is true that it *is* an invariant, but it is equally true that it is a temporal-logical necessity provable in TLA. The key issue is that *Descending* is defined as a parametrised sum of products, of which products one of the components is a Boolean value of a state predicate. It follows directly from propositional logic that these state predicates have one Boolean value or the other, hence the value of *Descending* is always defined. This is a stronger statement than claiming this assertion as an invariant of the implementation proof (which is simply to claim that it follows from the *Init* and *Safety* parts of *CB.Spec*).

There are two strong reasons to include the stronger version of the assertion:
(a): proof of a stronger version of a statement, when true, would in principle enable one to prove consequences more easily, and thereby simplify later parts of a proof;
(b): if this were part of the invariant, then the invariant would no longer be TRUE, and it would be necessary to include proofs of the obligations $\mathit{Init} \rightarrow \mathit{Inv}$ and $\mathit{Inv} \wedge [\mathcal{N}_1]_v \rightarrow \mathit{Inv}'$ from Theorem 1. The second of these is very similar to, but slightly weaker than, the claim needed for the hypothesis of the Lattice Rule, below, that $[\mathcal{N}_1]_v$ implies that the value of *Descending* decreases. Thus, the proofs would be similar, and one would either refer to this later step to provide a proof of the former, or, forgoing such a cross-structure reference, duplicate the proofs (with minor modifications) to maintain the hierarchical proof structure. Either solution seems inelegant. The current approach maintains the hierarchical structure while avoiding duplication of proof steps.

$\langle 7 \rangle 1.$ $\Box \exists\, i \in 0 \,..\, \mathit{MaxDescending} : \mathit{Descending}(\mathit{Buffer}) = i$

PROOF:

$\langle 8\rangle 1.\ \exists\, i \in 0 \,..\, MaxDescending \;:\; Descending(Buffer) = i$

PROOF:

$\langle 9\rangle 1.\ Descending(Buffer) \in 0 \,..\, MaxDescending$

PROOF:

$\langle 10\rangle 1.\ \land \sum_{k=1}^{N} (Buffer[k] = \bot).2^{N-k} \leq \sum_{k=1}^{N} 2^{N-k}$
$\qquad \land\ 0 \leq \sum_{k=1}^{N} (Buffer[k] = \bot).2^{N-k}$

PROOF:

LET: k : CONSTANT

$\langle 11\rangle 1.\ k \in 1 \,..\, N$
$\qquad \Rightarrow$
$\qquad \land\ (Buffer[k] = \bot).2^{N-k} \leq 2^{N-k}$
$\qquad \land\ 0 \leq (Buffer[k] = \bot).2^{N-k}$

PROOF:

$\langle 12\rangle 1.\ k \in 1 \,..\, N \quad \Rightarrow$
$\qquad \lor\ (Buffer[k] = \bot).2^{N-k} = 0$
$\qquad \lor\ (Buffer[k] = \bot).2^{N-k} = 2^{N-k}$

PROOF:

$\langle 13\rangle 1.\ k \in 1 \,..\, N \quad \Rightarrow$
$\qquad \lor\ (Buffer[k] = \bot) = 0$
$\qquad \lor\ (Buffer[k] = \bot) = 1$

PROOF: By definition of Boolean values. $\square$

$\langle 13\rangle 2.\ \land\ 0.2^{N-k} = 0$
$\qquad \land\ 1.2^{N-k} = 2^{N-k}$

PROOF: By standard axioms of arithmetic. $\square$

$\langle 13\rangle 3.$ Q.E.D.

PROOF: Follows by substitution and propositional logic from $\langle 13\rangle 1$ and $\langle 13\rangle 2$. $\square$

$\langle 12\rangle 2.\ \land\ 0 \leq 2^{N-k}$
$\qquad \land\ 2^{N-k} \leq 2^{N-k}$

PROOF: By standard axioms of arithmetic. $\square$

$\langle 12\rangle 3.$ Q.E.D.

PROOF: Follows immediately from $\langle 12\rangle 1$ and $\langle 12\rangle 2$ using substitution and propositional logic. $\square$

$\langle 11\rangle 2.\ \forall\, k \in 1 \,..\, N \;:\; a(k) \leq b(k)$
$\qquad \Rightarrow$
$\qquad \sum_{k=1}^{N} a(k) \leq \sum_{k=1}^{N} b(k)$

PROOF: This must follow from properties of $\sum$ defined in the *Naturals* module. $\square$

$\langle 11\rangle 3.$ Q.E.D.

PROOF: Follows immediately by substitution from $\langle 11\rangle 1$ and $\langle 11\rangle 2$. $\square$

$\langle 10\rangle 2.$ Q.E.D.

PROOF: Follows immediately from $\langle 10\rangle 1$ by definitions of '..', *Descending*, and *MaxDescending*. □

$\langle 9\rangle 2$. Q.E.D.

PROOF: Follows immediately from $\langle 9\rangle 1$ by existential generalisation over $Descending(Buffer)$ using the variable i. □

$\langle 8\rangle 2$. Q.E.D.

PROOF: Immediate from $\langle 8\rangle 1$ by STL1. □

The next step is a conclusion of the Lattice Rule, with $P \leadsto Q$ written as $\Box(P \Rightarrow \Diamond Q)$. Its justification is thus the premise of the Lattice Rule.

$$
\begin{array}{l}
\langle 7\rangle 2.\ \wedge\ \Box[\exists\, a : CB.push(a) \vee \exists\, k : CB.move(k)]_{Buffer} \\
\qquad \wedge\ WF_{Buffer}(CB.pop) \\
\qquad \wedge\ WF_{Buffer}(\exists\, k : CB.move(k)) \\
\qquad\quad \Rightarrow \\
\qquad\quad \Box(\wedge\ \exists\, i \in 0\,..\,MaxDescending : Descending(Buffer) = i \\
\qquad\qquad \wedge\ NotEmpty \\
\qquad\qquad \wedge\ NotStuffed \\
\qquad\qquad\quad \Rightarrow \\
\qquad\qquad\quad \Diamond FirstFull)
\end{array}
$$

PROOF:

The substep is the premise of the Lattice Rule, with $P \leadsto Q$ written as $\Box(P \Rightarrow \Diamond Q)$.

LET: i : CONSTANT

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

PROOF: This step is justified by an application of WF1 (and simple propositional logic). The substeps consist of the premises of WF1, with the instantiations

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

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

First, some trivial simplification of the consequent, from

$$\exists j \in 0\,..\,MaxDescending \,:\, j < i \land \Theta$$

to $\exists j < i \land \Theta$, for appropriate Θ.

$$\begin{array}{ll}
\langle 9\rangle 1. & \land\ \exists j < i \,:\, Descending(Buffer) = j \\
 & \quad \equiv \\
 & \quad \exists j \in 0\,..\,MaxDescending \,: \\
 & \qquad \land\ j < i \\
 & \qquad \land\ Descending(Buffer) = j \\
 & \land\ \exists j < i \,:\, Descending(Buffer') = j \\
 & \quad \equiv \\
 & \quad \exists j \in 0\,..\,MaxDescending \,: \\
 & \qquad \land\ j < i \\
 & \qquad \land\ Descending(Buffer') = j
\end{array}$$

Proof: Follows from $\langle 7\rangle 1$ and its substeps by predicate logic. □

The next step is the first premise of WF1.

$$\begin{array}{ll}
\langle 9\rangle 2. & \land\ NotEmpty \\
 & \land\ NotStuffed \\
 & \land\ Descending(Buffer) = i \\
 & \land\ \lor\ \exists\, a \,:\, CB.push(a) \\
 & \quad\ \lor\ \exists\, k \,:\, CB.move(k) \\
 & \quad\ \lor\ Buffer' = Buffer \\
 & \quad \Rightarrow
\end{array}$$

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

PROOF:

$\langle 10\rangle 1.$
$$
\begin{array}{l}
\land NotEmpty \\
\land NotStuffed \\
\land Descending(Buffer) = i \\
\land Buffer' = Buffer \\
\quad \Rightarrow \\
\quad \land NotEmpty' \\
\quad \land NotStuffed' \\
\quad \land Descending(Buffer') = i
\end{array}
$$

PROOF: Immediate by substitution of $Buffer'$ for $Buffer$. □

The following step is also used later.

$\langle 10\rangle 2.$
$$
\begin{array}{l}
\land NotEmpty \\
\land NotStuffed \\
\land Descending(Buffer) = i \\
\land \exists k : CB.move(k) \\
\quad \Rightarrow \\
\quad \lor FirstFull' \\
\quad \lor \exists j < i : \land NotEmpty' \\
\qquad\qquad\qquad \land NotStuffed' \\
\qquad\qquad\qquad \land Descending(Buffer') = j
\end{array}
$$

PROOF:

$\langle 11\rangle 1.$
$$
\begin{array}{l}
\land Descending(Buffer) = i \\
\land \exists k : CB.move(k) \\
\quad \Rightarrow \\
\quad \exists j < i : Descending(Buffer') = j
\end{array}
$$

PROOF:

The following two steps use some more easy arithmetic. Properties of $\sum$ are used which depend for their proofs on the exact definition of arithmetic and $\sum$. Different ways of specifying the properties might lead to different proofs. The proof below is intended as an example of how, and how long, such a proof would be. Any given proof depends on what is defined, and how, in the *Naturals* module, which is not included. One possibility for defined $\sum$ is to use a recursive definition, since $\sum$ is parameterised by N.

LET: k : CONSTANT

$\langle 12\rangle 1.$ $\wedge$ $Descending(Buffer) = i$
$\wedge$ $CB.move(k)$
$\Rightarrow$
$Descending(Buffer') = i + (2^{N-k} - 2^{N-(k-1)})$

PROOF:

$\langle 13\rangle 1.$ $CB.move(k)$
$\Rightarrow$
$Descending(Buffer') =$
$Descending(Buffer) + (2^{N-k} - 2^{N-(k-1)})$

PROOF:

$\langle 14\rangle 1.$ $CB.move(k)$
$\Rightarrow$
$\sum_{m=1}^{N}(Buffer'[m] = \perp).2^{N-m} =$
$\sum_{m=1}^{N}(Buffer[m] = \perp).2^{N-m} + (2^{N-k} - 2^{N-(k-1)})$

PROOF:

$\langle 15\rangle 1.$ $CB.move(k)$
$\Rightarrow$
$\wedge \sum_{m=1}^{k-2}(Buffer'[m] = \perp).2^{N-m} =$
$\sum_{m=1}^{k-2}(Buffer[m] = \perp).2^{N-m}$
$\wedge \sum_{m=(k+1)}^{N}(Buffer'[m] = \perp).2^{N-m} =$
$\sum_{m=(k+1)}^{N}(Buffer[m] = \perp).2^{N-m}$

PROOF:

LET: m : CONSTANT

$\langle 16\rangle 1.$ $CB.move(k)$
$\Rightarrow$
$\vee\ m \in 1\,..(k-2) \quad \Rightarrow$
$\vee\ m \in (k+1)\,..\,N$
$Buffer'[m] = Buffer[m]$

PROOF: Follows immediately from the definition of $CB.move(k)$. □

$\langle 16\rangle 2.$ Q.E.D.

PROOF: Follows from $\langle 16\rangle 1$, given that $\sum_{k=1}^{N}$ is a term, and therefore admits replacement of subterms by provably equal subterms by Leibniz's Law of predicate logic. □

$\langle 15\rangle 2.$ $CB.move(k)$
$\Rightarrow$

$\wedge\ (Buffer'[k] = \bot).2^{N-k} = 2^{N-k}$
$\wedge\ (Buffer'[k-1] = \bot).2^{N-(k-1)} = 0$
$\wedge\ (Buffer[k] = \bot).2^{N-k} = 0$
$\wedge\ (Buffer[k-1] = \bot).2^{N-(k-1)} = 2^{N-(k-1)}$

PROOF:

$\langle 16\rangle 1.$ $CB.move(k)$
$\Rightarrow$
$\wedge\ (Buffer'[k] = \bot) = 1$
$\wedge\ (Buffer'[k-1] = \bot) = 0$
$\wedge\ (Buffer[k] = \bot) = 0$
$\wedge\ (Buffer[k-1] = \bot) = 1$

PROOF: Immediate from the definition of $CB.move(k)$. □

$\langle 16\rangle 2.$
$\wedge\ 1.2^{N-k} = 2^{N-k}$
$\wedge\ 0.2^{N-(k-1)} = 0$
$\wedge\ 0.2^{N-k} = 0$
$\wedge\ 1.2^{N-(k-1)} = 2^{N-(k-1)}$

PROOF: Instantiations of standard axioms for 1 and 0 in arithmetic. □

$\langle 16\rangle 3.$ Q.E.D.

PROOF: Follows immediately by substitution and propositional logic using $\langle 16\rangle 1$ and $\langle 16\rangle 2$. □

The following step follows from various properties of $\sum$, including those which follow from a recursive definition, and properties of '+' and '−'. We state the step and simply assert that it follows. It should be clear that it is semantically correct, although syntactically an atomic step.

$\langle 15\rangle 3.$ $\sum_{m=1}^{N}(\mathit{Buffer}'[m] = \perp).2^{N-m} =$
$\sum_{m=1}^{k-2}(\mathit{Buffer}'[m] = \perp).2^{N-m}+$
$(\mathit{Buffer}'[k-1] = \perp).2^{N-(k-1)}+$
$(\mathit{Buffer}'[k] = \perp).2^{N-k}+$
$\sum_{m=(k+1)}^{N}(\mathit{Buffer}'[m] = \perp).2^{N-m}$
$=$
$\sum_{m=1}^{k-2}(\mathit{Buffer}[m] = \perp).2^{N-m}+$
$0 + 2^{N-k}+$
$\sum_{m=(k+1)}^{N}(\mathit{Buffer}[m] = \perp).2^{N-m}$
$=$
$\sum_{m=1}^{k-2}(\mathit{Buffer}[m] = \perp).2^{N-m}+$
$+2^{N-(k-1)} + 0+$
$\sum_{m=(k+1)}^{N}(\mathit{Buffer}[m] = \perp).2^{N-m}+$
$(2^{N-k} - 2^{N-(k-1)})$
$=$
$\sum_{m=1}^{N}(\mathit{Buffer}[m] = \perp).2^{N-m}+$
$(2^{N-k} - 2^{N-(k-1)})$

PROOF: Follows from the definition of $\sum$, and the associativity and commutativity of $\sum$ and '+', and other standard arithmetic axioms, including those for '$-$', using $\langle 15\rangle 1$ and $\langle 15\rangle 2$. □

$\langle 15\rangle 4.$ Q.E.D.

PROOF: Follows immediately from $\langle 15\rangle 3$ using transitivity of equality. □

$\langle 14\rangle 2.$ Q.E.D.

PROOF: Immediate from $\langle 14\rangle 1$ by substitution of $\mathit{Descending}(\mathit{Buffer}')$ and $\mathit{Descending}(\mathit{Buffer})$ for the $\sum$'s. □

$\langle 13\rangle 2.$ Q.E.D.

PROOF: Immediate from $\langle 13\rangle 1$ using the definition $i \triangleq \mathit{Descending}(\mathit{Buffer})$. □

$\langle 12\rangle 2.$ $i + (2^{N-k} - 2^{N-(k-1)}) < i$

PROOF:

$\langle 13\rangle 1.$ $i + (2^{N-k} - 2^{N-(k-1)}) =$
$i + (2^{N-k} - 2^{N-k+1}) =$
$i + (1.2^{N-k} - 2.2^{N-k}) =$
$i + 2^{N-k}(1-2) =$
$i - 2^{N-k} < i$

PROOF: Follows by axioms of arithmetic for '-', for '1', for exponent, the distributive law of multiplication over '-', the equation $1 - 2 = -1$, and the law $x + (-1).y = x - y$. □

$\langle 13\rangle 2.$ Q.E.D.

PROOF: Follows immediately from $\langle 13\rangle 1$ by transitivity of equality. □

$\langle 12\rangle 3$. Q.E.D.

PROOF: Follows from $\langle 12\rangle 1$ and $\langle 12\rangle 2$ by setting $j \triangleq i + (2^{N-k} - 2^{N-(k-1)})$, using $Descending(Buffer') = j \Rightarrow j \in 0 \,..\, MaxDescending$ from an earlier proof step, then existential quantification over j, followed by existential quantification over k. □

$\langle 11\rangle 2$. $\land\ \exists\, k : CB.move(k)$
$\land\ \neg(NotEmpty' \land NotStuffed')$
$\Rightarrow$
$FirstFull'$

PROOF:

$\langle 12\rangle 1$. $\exists\, k : CB.move(k)$
$\Rightarrow$
$NotEmpty'$

PROOF: Follows immediately by predicate logic from the definitions of $CB.move$ and $NotEmpty$. □

$\langle 12\rangle 2$. $\land\ \exists\, k : CB.move(k)$
$\land\ \neg(NotEmpty' \land NotStuffed')$
$\Rightarrow$
$\neg NotStuffed'$

PROOF: Follows immediately by propositional logic from $\langle 12\rangle 1$. □

$\langle 12\rangle 3$. $\neg NotStuffed'$
$\Rightarrow$
$FirstFull'$

PROOF: Follows immediately from the definitions. □

$\langle 12\rangle 4$. Q.E.D.

PROOF: Follows immediately by propositional logic from $\langle 12\rangle 2$ and $\langle 12\rangle 3$. □

$\langle 11\rangle 3$. Q.E.D.

PROOF: Follows immediately from $\langle 11\rangle 1$ and $\langle 11\rangle 2$ by propositional logic and by moving the quantifier $\exists\, j$. □

$\langle 10\rangle 3$. $\land\ NotEmpty$
$\land\ NotStuffed$
$\land\ Descending(Buffer) = i$
$\land\ \exists\, a : CB.push(a)$
$\Rightarrow$
$\exists\, j < i : \land\ NotEmpty'$
$\land\ NotStuffed'$
$\land\ Descending(Buffer') = j$

PROOF:

$\langle 11\rangle 1.$ $\wedge$ *NotEmpty*
$\wedge$ *NotStuffed*
$\wedge$ $Descending(Buffer) = i$
$\wedge$ $\exists\, a : CB.push(a)$
$\Rightarrow$
$\wedge$ $NotEmpty'$
$\wedge$ $NotStuffed'$
$\wedge$ $\exists\, j < i : Descending(Buffer') = j$

PROOF:

The following step also uses some simple arithmetic.

$\langle 12\rangle 1.$ $\wedge$ $Descending(Buffer) = i$
$\wedge$ $\exists\, a : CB.push(a)$
$\Rightarrow$
$\exists\, j < i : Descending(Buffer') = j$

PROOF:

$\langle 13\rangle 1.$ $\wedge$ $Descending(Buffer) = i$
$\wedge$ $\exists\, a : CB.push(a)$
$\Rightarrow$
$Descending(Buffer') = i - 1$

PROOF: Follows from definition of *Descending* and $CB.move$ using $2^{N-N} = 1$ and other arithmetic axioms. $\square$

$\langle 13\rangle 2.$ Q.E.D.

PROOF: Follows immediately from $\langle 13\rangle 1$ by substituting $j \triangleq i - 1$ and existentially quantifying over j. $\square$

$\langle 12\rangle 2.$ $\wedge$ *NotStuffed*
$\wedge$ *NotEmpty*
$\wedge$ $\exists\, a : CB.push(a)$
$\Rightarrow$ $\wedge$ $NotStuffed'$
$\wedge$ $NotEmpty'$

PROOF: Follows from definition of *NotStuffed* and *NotEmpty* by using predicate logic rules for quantifiers. $\square$

$\langle 12\rangle 3.$ Q.E.D.

PROOF: Follows immediately by propositional logic from $\langle 12\rangle 1$ and $\langle 12\rangle 2$. $\square$

$\langle 11\rangle 2.$ Q.E.D.

PROOF: Follows immediately by moving the quantifier $\exists\, j < i$ outside the conjunction in the consequent. $\square$

$\langle 10\rangle 4$. Q.E.D.

PROOF: Follows immediately by propositional logic from $\langle 10\rangle 1$, $\langle 10\rangle 2$ and $\langle 10\rangle 3$, using $\langle 9\rangle 1$ to substitute the quantifier $\exists j \in 0 \,..\, Maxdescending : j < i \wedge ..$ for the quantifier $\exists j < i : \ldots$ □

End of proof of first premise of WF1 application. The next step is the statement and proof of the second premise of WF1.

$$
\begin{array}{l}
\langle 9\rangle 3.\ \wedge\ NotStuffed \\
\qquad \wedge\ NotEmpty \\
\qquad \wedge\ Descending(Buffer) = i \\
\qquad \wedge\ \langle \wedge \vee \exists\, a : CB.push(a) \\
\qquad\qquad\quad \vee \exists\, k : CB.move(k) \\
\qquad\qquad \wedge \exists\, k : CB.move(k) \rangle_{Buffer} \\
\qquad\quad \Rightarrow \\
\qquad\quad \vee\ FirstFull' \\
\qquad\quad \vee\ \exists j \in 0 \,..\, MaxDescending : \begin{array}[t]{l} \wedge\ j < i \\ \wedge\ NotStuffed' \\ \wedge\ NotEmpty' \\ \wedge\ Descending(Buffer') = j \end{array}
\end{array}
$$

PROOF:

$$
\begin{array}{l}
\langle 10\rangle 1.\ \wedge \vee \exists\, a : CB.push(a) \\
\qquad\qquad \vee \exists\, k : CB.move(k) \\
\qquad\quad \wedge \exists\, k : CB.move(k) \\
\qquad\qquad \equiv \\
\qquad\qquad \exists\, k : CB.move(k)
\end{array}
$$

PROOF: Follows by propositional logic from the definitions of $CB.pop$ and $CB.move$. □

$$
\begin{array}{l}
\langle 10\rangle 2.\ \wedge\ NotStuffed \\
\qquad\quad \wedge\ NotEmpty \\
\qquad\quad \wedge\ Descending(Buffer) = i \\
\qquad\quad \wedge\ \langle \exists\, k : CB.move(k) \rangle_{Buffer} \\
\qquad\qquad \Rightarrow \\
\qquad\qquad \vee\ FirstFull' \\
\qquad\qquad \vee\ \exists j \in 0 \,..\, MaxDescending : \begin{array}[t]{l} \wedge\ j < i \\ \wedge\ NotStuffed' \\ \wedge\ NotEmpty' \\ \wedge\ Descending(Buffer') = j \end{array}
\end{array}
$$

PROOF: Follows by propositional logic, by strengthening the consequent, from the second substep of $\langle 9\rangle 2$ using $\langle 9\rangle 1$. □

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

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

The next step is the statement and proof of the third premise of WF1.

$\langle 9\rangle 4.$ $\wedge$ *NotStuffed*
$\wedge$ *NotEmpty*
$\wedge$ $Descending(Buffer) = i$
$\Rightarrow$
$Enabled\langle \exists\, k \,:\, CB.move(k)\rangle_{Buffer}$

PROOF: Follows by quantifier rules of predicate logic from definition of *NotStuffed* and *NotEmpty*. □

$\langle 9\rangle 5.$ Q.E.D.

PROOF: Immediate from TLA Rule WF1 with hypotheses $\langle 9\rangle 2$, $\langle 9\rangle 3$ and $\langle 9\rangle 4$ under the substitutions

$$P \triangleq \wedge\ NotEmpty \quad \wedge\ NotStuffed \quad \wedge\ Descending(Buffer) = i$$

$$\mathcal{N} \triangleq \vee\ \exists\, a \,:\, CB.push(a) \quad \vee\ \exists\, k \,:\, CB.move(k),$$

$$f \triangleq Buffer,$$

$$Q \triangleq \vee\ FirstFull \quad \vee\ \exists\, j \in 0\,..\,MaxDescending \,:\, \wedge\ j < i \ \wedge\ NotEmpty \ \wedge\ NotStuffed \ \wedge\ Descending(Buffer) = j,$$

$$A \triangleq \exists\, k \,:\, CB.move(k),$$

and the definition $(P \leadsto Q) \equiv \Box(P \Rightarrow \Diamond Q)$. Finally, the statement $i \in 0\,..\,MaxDescending$ is conjoined to the hypotheses. □

$\langle 8\rangle 2.$ Q.E.D.

PROOF: Immediate from $\langle 8\rangle 1$ by the Lattice Rule of TLA with substitutions $\mathsf{c} \triangleq i$, $\mathsf{S} \triangleq 1\,..\,N$, $\succ \triangleq >$,

$$H_{\mathsf{c}} \triangleq \wedge\ NotEmpty \quad \wedge\ NotStuffed \quad \wedge\ Descending(Buffer) = i,$$

$G \triangleq FirstFull$, and

$$F \triangleq \wedge\ \Box[\exists\, a \,:\, CB.push(a) \vee \exists\, k \,:\, CB.move(k)]_{Buffer} \quad \wedge\ WF_{Buffer}(CB.pop) \quad \wedge\ WF_{Buffer}(\exists\, k \,:\, CB.move(k)).$$

$\langle 7\rangle 3.$ $\land\ \Box[\exists\, a : CB.push(a) \lor \exists\, k : CB.move(k)]_{Buffer}$
$\land\ WF_{Buffer}(CB.pop)$
$\land\ WF_{Buffer}(\exists\, k : CB.move(k))$
$\Rightarrow$
$\Box(\exists\, i \in 0\,..\,MaxDescending : Descending(Buffer) = i$
$\Rightarrow$
$(NotEmpty \land NotStuffed)$
$\Rightarrow$
$\Diamond FirstFull))$

Proof: Follows from $\langle 7\rangle 2$ by temporal logic, replacing a formula of the form $A \land B \Rightarrow C$ with one of the form $A \Rightarrow (B \Rightarrow C)$ in the consequent, in which
$A \triangleq \exists\, i \in 0\,..\,MaxDescending : Descending(Buffer) = i$,
$B \triangleq (NotEmpty \land NotStuffed)$, and $C \triangleq \Diamond FirstFull$. □

$\langle 7\rangle 4.$ $\land\ \Box[\exists\, a : CB.push(a) \lor \exists\, k : CB.move(k)]_{Buffer}$
$\land\ WF_{Buffer}(CB.pop)$
$\land\ WF_{Buffer}(\exists\, k : CB.move(k))$
$\land\ \Box(\exists\, i \in 0\,..\,MaxDescending : Descending(Buffer) = i)$
$\Rightarrow$
$\Box((NotEmpty \land NotStuffed)$
$\Rightarrow$
$\Diamond FirstFull)$

Proof: Follows from $\langle 7\rangle 3$ by temporal logic, using the following temporal logic rule:

Let: X, Y, Z : propositional variable

$\langle 8\rangle 1.$ $\dfrac{\Box X \Rightarrow \Box(Z \Rightarrow \Box Y)}{\Box X \land \Box Z \Rightarrow \Box Y}$

Proof: This calculation in temporal logic follows from PTL. □

$\langle 8\rangle 2.$ Q.E.D.

Proof: Follows from $\langle 7\rangle 3$ using $\langle 8\rangle 1$ with
$Z \triangleq (\exists\, i \in 0\,..\,MaxDescending : Descending(Buffer) = i)$,
$Y \triangleq (NotEmpty \land NotStuffed) \Rightarrow \Diamond FirstFull$. □

$\langle 7\rangle 5.$ Q.E.D.

Proof: Follows by propositional logic from $\langle 7\rangle 4$ using $\langle 7\rangle 1$ to eliminate a conjunct in the antecedent. □

$\langle 6\rangle 3.$ Q.E.D.

Proof: Follows immediately by replacing the consequent of $\langle 6\rangle 2$ by an equivalent formula from $\langle 6\rangle 1$. □

This completes the proof of the second hypothesis of the Bridge rule $\langle 5\rangle 1$. We now use the Bridge rule to infer its (instantiated) conclusion, which is the desired simplified version of the fourth premiss of WF2, step $\langle 4\rangle 3$.

$\langle 5\rangle 4$. Q.E.D.

PROOF: Follows from the Bridge rule $\langle 5\rangle 1$ under the instantiation

$$\begin{array}{lcl} \Box X & \triangleq & \wedge\ \Box[\vee\ \exists\, a\ :\ CB.push(a) \qquad , \\ & & \qquad \vee\ \exists\, k\ :\ CB.move(k)]_{Buffer} \\ & & \wedge\ WF_{Buffer}(CB.pop) \\ & & \wedge\ WF_{Buffer}(\exists\, k\ :\ CB.move(k)) \\ NE & \triangleq & NotEmpty, \end{array}$$

$FF \triangleq FirstFull$, with first hypothesis $\langle 5\rangle 2$ and second hypothesis $\langle 5\rangle 3$. □

We now conclude the validity of the fourth premiss of WF2 from its simplified version $\langle 4\rangle 3$.

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

PROOF: Follows immediately from $\langle 4\rangle 3$ using $\langle 4\rangle 1$ and $\langle 4\rangle 2$. □

We now conclude the validity of the conclusion of WF2, which gives the desired liveness property of $AB.pop$ from the specification of CB.

$\langle 3\rangle 5$. 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 $\mathcal{N} \triangleq (\exists\, a\ :\ CB.push(a) \vee CB.pop \vee \exists\, k\ :\ CB.move(k))$, $f = Buffer$, $\mathcal{A} \triangleq \mathcal{B} \triangleq CB.pop$, $\mathcal{M} \triangleq AB.pop$, $g = buffer$, $P \triangleq FirstFull$, $\Box F \triangleq WF_{Buffer}(\exists\, k\ :\ CB.move(k))$. □

Finally, we conclude the main theorem from the steps demonstrating the proof obligations for the initial condition, the safety property, and the liveness property.

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

PROOF: Follows from $\langle 2\rangle 1$, $\langle 2\rangle 2$ and $\langle 2\rangle 3$ by Theorem 1, using $Inv = \mathbf{True}$. □

Acknowledgements

Paul Gibson and Abdellilah Mokkedem suggested this example. I started this work while visiting INRIA Lorraine, where I was hosted by Dominique Méry and Patrick Rambert. Bob Johnston told me about the F3314 chip in which this concrete buffer design was implemented. And Leslie Lamport developed TLA, the support tool `pf.sty` for hierarchical proofs, the macros

`tla.sty` for writing TLA specifications and of course LaTeX , without which we'd *all* be barbarians. Hearty thanks to all.

References

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

[AS87] B. Alpern and F. B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2:117–126, 1987.

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

[GNRR93] R. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, editors. *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[Gri81] D. Gries. *The Science of Programming*. Springer-Verlag, 1981. ISBN 0-387-96480-0/3-540-96480-0/0-387-90641-X/3-540-90641-X.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.

[Lad96] Peter B. Ladkin. Lazy cache implements complete cache. In `http://www.techfak.uni-bielefeld.de/~ladkin/` , January 1996.

[Lam89] Leslie Lamport. A simple approach to specifying concurrent systems. *Communications of the ACM*, 32(1):32–45, January 1989.

[Lam93a] Leslie Lamport. How to write a proof, January 1993. In `http://www.research.digital.com/SRC/tla/`.

[Lam93b] Leslie Lamport. Hybrid systems in TLA$^+$. In *[GNRR93]*, pages 77–102, 1993.

[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.

[LL94] Peter Ladkin and Leslie Lamport. TLA proof schema for implementation, June 1994. Written by Ladkin from notes of Lamport.

[LLOR94] Peter Ladkin, Leslie Lamport, Bryan Olivier, and Denis Roegel. Lazy caching: An assertional view, 1994. In `http://www.research.digital.com/SRC/tla/`. To appear, *Distributed Computing*.

[LLOR95] Peter Ladkin, Leslie Lamport, Bryan Olivier, and Denis Roegel. A formal correctness proof of the lazy caching algorithm, 1995. To appear.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall International, 1989.

[Par72] D.L. Parnas. On the criteria to be used in decomposing systems. *Communications of the ACM*, 15(12), December 1972.

[Par74] D.L. Parnas. On a 'buzzword': Hierarchical structure. In *Information Processing 1974 (Proc. IFIP Congress)*, pages 336–339. 1974.

[Spi89] J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall International, 1989. ISBN 0-13-983768-X.

[Sti89] C. Stirling. Temporal logics for CCS. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models of Concurrency*, volume 354 of *Lecture Notes in Computer Science*, pages 660–672. Springer-Verlag, 1989.

[Wen78] J. Wensley *et al.* SIFT: Design and analysis of a fault-tolerant computer for aircraft control. *Proceedings of the IEEE*, 66(10):1240–1255, October 1978.

[Yu93] Y. Yu. Automated proofs of object code for a widely used microprocessor. Technical Report 114, Digital Equipment Corporation Systems Research Center, Palo Alto, California, October 1993. Revised version of the author's Ph.D. thesis at the University of Texas at Austin.