Lazy Caching in TLA

Peter Ladkin Leslie Lamport Bryan Olivier
Denis Roegel

Tue 13 Apr 1999 [12:12]

Abstract

We address the problem, proposed by Gerth, of verifying that a simplified version of the lazy caching algorithm of Afek, Brown, and Merritt is sequentially consistent. We specify the algorithm and sequential consistency in TLA$^+$, a formal specification language based on TLA (the Temporal Logic of Actions). We then describe how to construct and check a formal TLA correctness proof.

Contents

1 Introduction

Assertional verification of concurrent algorithms began in 1975 with Ashcroft's seminal paper [4]. By the late 1980's, assertional methods had been developed for specifying concurrent systems and proving that a lower-level specification implements a higher-level one. Our goal is to transform assertional specification and verification from a scientific theory into an engineering discipline.

Engineering is the practical application of scientific principles. An engineering discipline comprises a well-defined collection of intellectual tools that can be applied to a class of problems. The intellectual tools of our approach are the logic TLA (the Temporal Logic of Actions) [14], the specification language TLA$^+$ [12], and a hierarchical proof style for writing rigorous proofs [15]. The class of problems we consider are the specification and verification of safety and liveness properties of concurrent systems.

We demonstrate our approach on a problem suggested by Gerth [8]: verifying that a simplified version of the lazy caching algorithm of Afek, Brown, and Merritt [3] is sequentially consistent [16]. Although our exposition is self-contained, it is about specification and verification, not about lazy caching. We formally specify the algorithm, but a formal specification is no substitute for an intuitive explanation. Readers looking for such an explanation are referred to [3].

Published "proofs" of incorrect concurrent algorithms have taught us the need for rigor. We achieve rigor by using formal mathematics. TLA is a formal logic, with precise proof rules; and TLA$^+$ has a formal semantics. Each step in a hierarchical proof is a mathematical formula; English appears only in the proofs of the lowest-level steps. Greater reliability is obtained by carrying out the proofs to lower levels of detail. Ultimately, one reaches a point where prose can be eliminated and the proof checked by computer [7]. However, the function of proofs in engineering is not to attain absolute certainty, but to achieve a reasonable degree of confidence with a reasonable amount of effort. We believe that, at the moment, for many large applications, the most cost-effective approach stops short of mechanical verification.

We believe that such proofs are inevitably long and boring. They are long because many details must be checked to ensure correctness. They are boring because even the most interesting proof becomes boring when carried out to the level of detail needed to avoid errors. Conventional mathematical proofs try to be interesting and to avoid boring details; as a result, a significant fraction of the theorems published in mathematical journals are wrong [6, 15]. Discipline can be used to eliminate careless mistakes when checking a long series of trivial steps; discipline cannot help detect subtle errors in interesting steps.

In assertional reasoning, insight is typically required to find an invariant and construct a refinement mapping [1]. The proof itself is a tedious matter of checking the details. For the lazy caching algorithm, we give the invariant and refinement mapping, and we describe the high-level structure of the proof. A complete proof would be much too long and boring to include here. Moreover, the lower-level parts we have done are too long to be read conveniently as a conventional paper document. We hope to develop a tool for managing and displaying structured proofs in hypertext.

To be useful, an engineering discipline should be applicable to a reasonably broad class of problems. There would be little point developing a complete specification and proof method just for caching algorithms. TLA and TLA^+ have been applied to a number of diverse domains, including hybrid systems [12] and distributed fault-tolerant algorithms [17]. Nothing new has been introduced for the lazy caching example.[1] Some formalisms might be better suited to reasoning about caching algorithms. However, we are not interested in finding the simplest or most elegant possible proof. This kind of short, subtle algorithm can sometimes be verified by a clever trick that does not generalize to other applications. We have no objection to using cleverness to simplify a proof, we just do not want to depend upon it.

The TLA^+ specification of the lazy caching algorithm takes about 70 lines. The second author has recently participated in two projects to verify cache coherence protocols of real multiprocessor computers.[2] Each of their specifications is about 1800 lines of TLA^+. Problems of this size are addressed with rigorous discipline, not clever tricks. We therefore obtain our proof of the lazy caching algorithm by a straightforward, rigorous application of our method, just as we would for a larger, industrial example. This rigor is overkill for so simple an algorithm, but it is essential for handling real systems. The most novel part of the proof is the specification of sequential consistency, and it is a direct application of an idea introduced in [11] for specifying serializability. As with any engineering discipline, it takes practice to learn to write formal specifications and proofs with TLA and TLA^+. But writing a specification and proof of, for example, a Byzantine

[1]While not new, the TLA formulation of the rules for introducing auxiliary variables appear in print here for the first time.

[2]These protocols, like those used in all other multiprocessors we know about, are unrelated to lazy caching. The practical applications of the lazy caching algorithm lie outside the realm of conventional multiprocessor cache consistency.

agreement algorithm teaches the skills needed to verify a caching algorithm. One does not need a new proof method for each problem domain.

In Section 2, we introduce TLA and TLA$^+$ by writing a formal specification of Gerth's version of the lazy caching algorithm. Section 3 presents two specifications of sequential consistency—the one we use, and an equivalent one in the spirit of the original definition [16], as adapted by Gerth [8]. In Section 4, we describe the proof that the specification of the lazy caching algorithm implements the specification of sequential consistency.

Since we wrote the proof, our intellectual tools have been augmented by mechanical ones—namely, a parser and model checker for TLA$^+$ are under development. Section 4.5 describes our use of these tools to check the proof.

2 Lazy Caching in TLA$^+$

2.1 TLA and TLA$^+$

TLA is a temporal logic. Temporal logic formulas contain *flexible variables*, which represent quantities that change with time, and *rigid variables*, which represent quantities that do not change with time. Flexible variables are usually just called *variables*; rigid variables are sometimes called *constants*. The meaning $[\![S]\!]$ of a TLA formula S is a boolean function on behaviors, where a behavior is an infinite sequence of states and a state is an assignment of values to all flexible variables.[3] A behavior σ *satisfies* a formula S iff $[\![S]\!](\sigma)$ equals TRUE. A formula is *valid* iff it is satisfied by all behaviors. A specification S is said to *implement* a specification T iff every behavior satisfying S satisfies T, which is true iff the formula $S \Rightarrow T$ is valid. When we write "S implies T", we usually mean that $S \Rightarrow T$ is valid. The syntax and semantics of TLA are described in [14].

TLA$^+$ is a formal language based on TLA and Zermelo-Fraenkel set theory. We will explain its features as they are used. Most of the operators and constructs of TLA$^+$, including all the ones we use, are summarized in Figures 1 and 2. We try to explain our specifications in enough detail that readers unfamiliar with TLA and TLA$^+$ will be able to understand them.

We do not attempt to explain the choices made in TLA$^+$. The rationale for much of its notation is not apparent from this one example. Also, why we *don't* write certain things may be puzzling. There are a number of restrictions in TLA$^+$ that are needed to maintain its simplicity. Surprising

[3] One source of TLA's simplicity is that there is a single state space, instead of a different set of states for each specification.

Logic

$\text{TRUE} \quad \text{FALSE} \quad \land \quad \lor \quad \lnot \quad \Rightarrow \quad \equiv$

$\forall x : p(x) \qquad \exists x : p(x) \qquad \forall x \in S : p(x) \qquad \exists x \in S : p(x)$

$\text{CHOOSE } x : p(x)$ [Equals some x satisfying p, or an arbitrary value if no such x exists]

Sets

$= \quad \neq \quad \in \quad \notin \quad \cup \quad \cap \quad \subseteq \quad \setminus$ [set difference]

$\{e_1, \ldots, e_n\}$	[Set consisting of elements e_i]
$\{x \in S : p(x)\}$	[Set of elements x in S satisfying $p(x)$]
$\{e(x) : x \in S\}$	[Set of elements $e(x)$ such that x in S]
$\text{UNION } S$	[Union of all elements of S]

Functions

$f[e]$	[Function application]
$\text{DOMAIN } f$	[Domain of function f]
$[x \in S \mapsto e(x)]$	[Function f such that $f[x] = e(x)$ for $x \in S$]
$[S \rightarrow T]$	[Set of functions f with $f[x] \in T$ for $x \in S$]
$[f \text{ EXCEPT } ![e_1] = e_2]$	[Function $\widehat{f}$ equal to f except $\widehat{f}[e_1] = e_2$]

Tuples

$e[i]$	[The i^{th} component of tuple e]
$\langle e_1, \ldots, e_n \rangle$	[The n-tuple whose i^{th} component is e_i]
$S_1 \times \ldots \times S_n$	[The set of all n-tuples with i^{th} component in S_i]

Miscellaneous

"$\mathsf{c}_1 \ldots \mathsf{c}_n$"	[A literal string of n characters]
$d_1 \ldots d_n$	[Numbers]
$\text{IF } p \text{ THEN } e_1 \text{ ELSE } e_2$	[Equals e_1 if p true, else e_2]
$\text{LET } x_1 \triangleq e_1 \ldots x_n \triangleq e_n \text{ IN } e$	[Equals e in the context of the definitions]

Nonconstant Operators

p'	[p with variables primed]	$\Box F$	[F is always true]
$[A]_e$	[$A \lor (e' = e)$]	$\Diamond F$	[Eventually: $\lnot \Box \lnot F$]
$\langle A \rangle_e$	[$A \land (e' \neq e)$]	$\text{WF}_e(A)$	[Weak fairness]
$\text{UNCHANGED } e$	[$e' = e$]	$\text{SF}_e(A)$	[Strong fairness]

$\text{ENABLED } A$ [$\exists$ values of primed variables for which action A is true]

$\boldsymbol{\exists} x : F \qquad \boldsymbol{\forall} x : F$ [Temporal quantification.]

Figure 1: TLA$^+$ operators.

ASSUME $N \triangleq A$

Defines N to equal formula A, which can contain only constant parameters, and asserts it as an assumption.

CONSTANT $C_1, \ldots, C_n$

Declares the C_i to be constant parameters (rigid variables).

EXTENDS $M_1, \ldots, M_n$

Imports parameters, assumptions, definitions, and theorems from the modules M_i.

$N \triangleq$ INSTANCE M

Imports definitions from module M with parameters instantiated and with "$N!$" appended to defined names. If N has the form $P(x_1, \ldots, x_n)$, then the x_i become additional formal parameters of each included definition.

MODULE M

Begins a module named M.

THEOREM $N \triangleq T$

Defines N to equal formula T and asserts it to be a theorem that is deducible from the module's definitions and assumptions.

VARIABLE $v_1, \ldots, v_n$

Declares the v_i to be variable parameters (flexible variables).

$N \triangleq E$

Defines N to equal E. If N has the form $P(x_1, \ldots, x_n)$, then this defines P to be an operator with n arguments

$f[x \in S] \triangleq \ldots$

Defines f to be a function with domain S.

A meaningless decoration.

Marks the end of a module.

Figure 2: Syntactic keywords and symbols of TLA$^+$.

Event	Allowed if	Action
$\mathsf{R}_i(d,a)$	$C_i(a) = d \land Out_i = \{\}$ $\land$ no $*$-ed entries in In_i	
$\mathsf{W}_i(d,a)$		$Out_i := append(Out_i, (d,a))$
$\mathsf{MW}_i(d,a)$	$head(Out_i) = (d,a)$	$Mem[a] := d$; $Out_i := tail(Out_i)$; $(\forall k \neq i \;::\; In_k := append(In_k, (d,a)))$; $In_i := append(In_i, (d,a,*))$
$\mathsf{MR}_i(d,a)$	$Mem[a] = d$	$In_i := append(In_i, (d,a))$
$\mathsf{CU}_i(d,a)$	$head(In_i)$ is either (d,a) or $(d,a,*)$	$In_i := tail(In_i)$; $C_i := update(C_i, d, a)$
CI_i		$C_i := restrict(C_i)$

Initially: $\forall a \;\; Mem[a] = 0$
$\land \forall i = 1 \ldots n \;\; C_i \subset Mem \land In_i = \{\} \land Out_i = \{\}$

Fairness: no action other than CI_i can be always enabled but never taken

W—write MW—memory write CU—cache update
R—read MR—memory read CI—cache invalidate

Figure 3: Gerth's version of the lazy caching algorithm, from Figure 4 of [8].

complications can arise from features that appear innocuous—for example, a type system [18]. These complications are usually not apparent in semi-formal expositions such as [5] and [21].

2.2 The Lazy Caching Algorithm

We introduce TLA and TLA$^+$ by first specifying Gerth's version of the lazy caching algorithm. Gerth described the algorithm informally with the state machine of Figure 3.[4] Our specification is a fairly direct translation of this state machine into TLA$^+$. Had the state machine been specified formally, the translation could have been performed by a straightforward algorithm. However, Gerth's state machine would not have had so simple and compact a description if it were written in a general-purpose formal language for specifying state machines.

[4]Gerth specified that only $\mathsf{R}_i(d,a)$ and $\mathsf{W}_i(d,a)$ events are externally observable. However, by observing only these events, there is no way to tell that a memory system is using lazy caching and not some other cache coherence algorithm. We therefore also make externally visible the events that change the queues and caches.

MODULE *MemParams*

EXTENDS *Naturals, Sequences*

VARIABLE *ch*
CONSTANT *Data*, *InitData*, *Addr*, *N*

ASSUME $ValAssump \triangleq (InitData \subseteq Data) \land (N \in Nat) \land (N > 0)$

$Proc \triangleq 1 \, . \, . \, N$

Figure 4: Parameters of the cache specifications.

We represent an algorithm by a TLA formula. As in ordinary mathematics, hierarchical structure is obtained by defining complex formulas in terms of simpler ones. Our specification is a sequence of definitions, culminating in the one that describes the lazy caching algorithm.

2.2.1 Some Preliminaries: Module *MemParams*

TLA$^+$ specifications are structured using modules. Parts of the lazy caching specification are placed in the separate module *MemParams* of Figure 4 so they can be easily reused in later specifications.

The module first extends modules *Naturals* and *Sequences*, meaning that it adds the definitions from those modules to the *MemParams* module. (An equivalent specification can be obtained by replacing the EXTENDS statement with the definitions from those two modules.) The standard module *Naturals* defines the set *Nat* of natural numbers, operators on natural numbers such as $+$ and $>$, and the usual representation of natural numbers as Arabic numerals. It also defines the infix operator ". ." so that, if i and j are natural numbers, then $i \, . \, . \, j$ is the set of natural numbers n with $i \leq n \leq j$. The *Sequences* module defines some operations on sequences; this module and the operators it defines are described in Section 2.2.2 below.

Module *MemParams* next declares some parameters. Parameters are the free symbols of a specification. By replacing all defined symbols with their definitions, a TLA$^+$ specification can be reduced to a formula containing only parameters and the operators of Figure 1. The parameter *ch* is a (flexible) variable representing the communication channels between the processors and the memory system. We need such a variable because TLA is based on states rather than events. Gerth's $\mathsf{W}_i(d, a)$ and $\mathsf{R}_i(d, a)$ events are represented in our specification by changes to $ch[i]$, the channel joining process i to the memory. The other parameters, all constants, are: the set *Data* of values that can be stored in a memory location, the subset *InitData* of possible initial values, the set *Addr* of memory addresses, and the number N of processes.

The description of the parameters in the preceding paragraph is an informal comment. The CONSTANT declaration tells us only that N is a constant, not that it is a number. The ASSUME statement asserts, and assigns the name *ValAssump* to, the assumption that *InitData* is a subset of *Data* and N is a positive natural number. (It is unnecessary to assume that *Data* and *InitData* are sets because TLA$^+$ is based on Zermelo-Fraenkel set theory, in which every constant is a set.)

Module *MemParams* ends by defining the constant *Proc*, the set of processor names, to be the set $\{1, \ldots, N\}$ of natural numbers.

2.2.2 Parameters and Mathematical Operators

The specification of the lazy caching algorithm is contained in the *LazyCache* module of Figures 5 and 6. The module first imports four other modules. Importing module *MemParams* imports its definitions, assumptions, and parameter declarations. The other modules contain only operator definitions. We have already discussed the *Naturals* and *Sequences* modules. Module *ChannelInterface* defines the operator *ChanOp* described below; the module is given in Section 3.3.1.

Module *LazyCache* next declares four variable parameters, which correspond to the variables of Gerth's specification. TLA$^+$ does not use subscripted variables, and we prefer to use lower-case names for variables and upper-case names for constants, so we write $c[i]$ instead of C_i, $in[i]$ instead of In_i, etc. Processor i maintains the queues $in[i]$ and $out[i]$ and the cache $c[i]$; variable *mem* represents the main memory. Figure 7 is a picture of the state machine; it describes the flow of data in the algorithm and the meanings of the variables.

When using a formal language, we must specify mathematical operations that are usually taken for granted. TLA$^+$ provides the predefined operators for sets, functions, and tuples shown in Figure 1. Its set notation is standard. As in ordinary mathematics, a function has a domain, which is a set. The set of all functions with domain S and range a subset of T is written $[S \rightarrow T]$. Function application is denoted by square brackets. The TLA$^+$ construct $[x \in S \mapsto e(x)]$ is a "lambda expression" that represents the function f with domain S such that $f[x] = e(x)$ for all x in S. For example, the squaring

$$
\begin{array}{l}
\textsc{module } \mathit{LazyCache} \\
\hline
\textsc{extends } \mathit{MemParams},\ \mathit{ChannelInterface},\ \mathit{Sequences},\ \mathit{Naturals} \\
\hline
\textsc{variable } c, \mathit{in}, \mathit{out}, \mathit{mem} \\
\hline
\mathit{NotData} \ \triangleq\ \textsc{choose } i : i \notin \mathit{Data} \\
\mathit{Restrict}(f) \ \triangleq \\
\quad \{g \in [\mathit{Addr} \rightarrow \mathit{Data} \cup \{\mathit{NotData}\}] : \forall a \in \mathit{Addr} : g[a] \in \{f[a], \mathit{NotData}\}\} \\[1ex]
\mathit{Init} \ \triangleq\ \land\ \mathit{mem} \in [\mathit{Addr} \rightarrow \mathit{InitData}] \\
\qquad\quad\ \land\ c \in [\mathit{Proc} \rightarrow \mathit{Restrict}(\mathit{mem})] \\
\qquad\quad\ \land\ \mathit{in} = [i \in \mathit{Proc} \mapsto \langle\,\rangle] \\
\qquad\quad\ \land\ \mathit{out} = [i \in \mathit{Proc} \mapsto \langle\,\rangle] \\
\mathit{Read}(i, d, a) \ \triangleq\ \land\ \mathit{out}[i] = \langle\,\rangle \\
\qquad\qquad\qquad\ \land\ \forall j \in 1 \mathrel{..} \mathit{Len}(\mathit{in}[i]) : \mathit{Len}(\mathit{in}[i][j]) = 2 \\
\qquad\qquad\qquad\ \land\ c[i][a] = d \\
\qquad\qquad\qquad\ \land\ \mathit{ChanOp}(\mathit{ch}[i], \langle i, \text{“Rd”}, d, a\rangle) \\
\qquad\qquad\qquad\ \land\ \forall j \in \mathit{Proc} \setminus \{i\} : \textsc{unchanged } \mathit{ch}[j] \\
\qquad\qquad\qquad\ \land\ \textsc{unchanged } \langle c, \mathit{in}, \mathit{out}, \mathit{mem}\rangle \\
\mathit{Write}(i, d, a) \ \triangleq\ \land\ \mathit{ChanOp}(\mathit{ch}[i], \langle i, \text{“Wr”}, d, a\rangle) \\
\qquad\qquad\qquad\ \land\ \mathit{out}' = [\mathit{out} \textsc{ except } ![i] = \mathit{out}[i] \circ \langle\langle d, a\rangle\rangle] \\
\qquad\qquad\qquad\ \land\ \forall j \in \mathit{Proc} \setminus \{i\} : \textsc{unchanged } \mathit{ch}[j] \\
\qquad\qquad\qquad\ \land\ \textsc{unchanged } \langle c, \mathit{in}, \mathit{mem}\rangle \\
\mathit{MemWrite}(i) \ \triangleq \\
\quad \land\ \mathit{out}[i] \neq \langle\,\rangle \\
\quad \land\ \mathit{out}' = [\mathit{out} \textsc{ except } ![i] = \mathit{Tail}(\mathit{out}[i])] \\
\quad \land\ \mathit{in}' = [j \in \mathit{Proc} \mapsto \mathit{in}[j] \circ \textsc{if } j = i \textsc{ then } \langle \mathit{Head}(\mathit{out}[i]) \circ \langle \text{“*”} \rangle\rangle \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textsc{else } \langle \mathit{Head}(\mathit{out}[i])\rangle \quad] \\
\quad \land\ \mathit{mem}' = [\mathit{mem} \textsc{ except } ![\mathit{Head}(\mathit{out}[i])[2]] = \mathit{Head}(\mathit{out}[i])[1]] \\
\quad \land\ \textsc{unchanged } \langle c, \mathit{ch}\rangle \\
\mathit{CacheUpdate}(i) \ \triangleq\ \land\ \mathit{in}[i] \neq \langle\,\rangle \\
\qquad\qquad\qquad\qquad \land\ \mathit{in}' = [\mathit{in} \textsc{ except } ![i] = \mathit{Tail}(\mathit{in}[i])] \\
\qquad\qquad\qquad\qquad \land\ c' = [c \textsc{ except } ![i][\mathit{Head}(\mathit{in}[i])[2]] = \mathit{Head}(\mathit{in}[i])[1]] \\
\qquad\qquad\qquad\qquad \land\ \textsc{unchanged } \langle \mathit{out}, \mathit{ch}, \mathit{mem}\rangle
\end{array}
$$

Figure 5: The lazy caching algorithm (beginning).

$$
\begin{aligned}
MemRead(i) \;\triangleq\;& \land\ \exists\, a \in Addr\ :\\
&\qquad in' = [in \text{ EXCEPT } ![i] = in[i] \circ \langle\langle mem[a], a \rangle\rangle]\\
& \land\ \text{UNCHANGED}\ \langle out, c, mem, ch \rangle
\end{aligned}
$$

$$
\begin{aligned}
CacheInval(i) \;\triangleq\;& \exists\, f \in Restrict(c[i])\ :\\
&\quad \land\ c' = [c \text{ EXCEPT } ![i] = f]\\
&\quad \land\ \text{UNCHANGED}\ \langle in, out, mem, ch \rangle
\end{aligned}
$$

$$
\begin{aligned}
Next(i) \;\triangleq\;& \lor\ \exists\, d \in Data, a \in Addr\ :\ Read(i, d, a) \lor Write(i, d, a)\\
& \lor\ MemWrite(i) \lor CacheUpdate(i) \lor MemRead(i)\\
&\quad \lor\ CacheInval(i)
\end{aligned}
$$

$$
vars \;\triangleq\; \langle c, in, out, mem, ch \rangle
$$

$$
\begin{aligned}
Spec \;\triangleq\;& \land\ Init\\
& \land\ \Box[\exists\, i \in Proc\ :\ Next(i)]_{vars}\\
& \land\ \forall\, i \in Proc\ :\ \mathrm{WF}_{vars}(CacheUpdate(i)) \land \mathrm{WF}_{vars}(MemWrite(i))
\end{aligned}
$$

Figure 6: The lazy caching algorithm (continued).

function on natural numbers is $[n \in Nat \mapsto n * n]$. Tuples are enclosed by angle brackets. An n-tuple is a function whose domain is the set $\{1, \ldots, n\}$ of natural numbers, so $\langle v_1, \ldots, v_n \rangle[i]$ (the function $\langle v_1, \ldots, v_n \rangle$ applied to i) equals v_i, for $1 \leq i \leq n$.

Mathematical operators not provided by TLA$^+$ must be defined. Our specifications use operators on finite sequences, including *Head*, *Tail*, $\circ$ (concatenation), and *Len* (length) that are defined in the *Sequences* module of Figure 8. Finite sequences are represented as tuples, so $\langle v, w \rangle$ equals $\langle v \rangle \circ \langle w \rangle$, and $\langle\,\rangle$ is the empty sequence. Other operators defined by the *Sequences* module are explained later. The reader interested in how ordinary mathematics is formalized in TLA$^+$ can work out the details of the definitions in module *Sequences* with the aid of Figures 1 and 2.

A memory assigns data values to addresses, so its contents are represented by a function in $[Addr \rightarrow Data]$. A cache assigns data values to some addresses. We could represent the contents of a cache by a function whose domain is a subset of *Addr*. However, we find it more convenient to choose some value *NotData* not in *Data* and represent a cache's contents by a function f in $[Addr \rightarrow Data \cup \{NotData\}]$, where $f[a] = NotData$ means that the cache does not contain a data value for address a. The first definition in module *LazyCache* is of the constant *NotData*.

"Restricting" the contents of a cache means removing data values from it. The operator *Restrict* is defined so that *Restrict*(f) is the set of restrictions

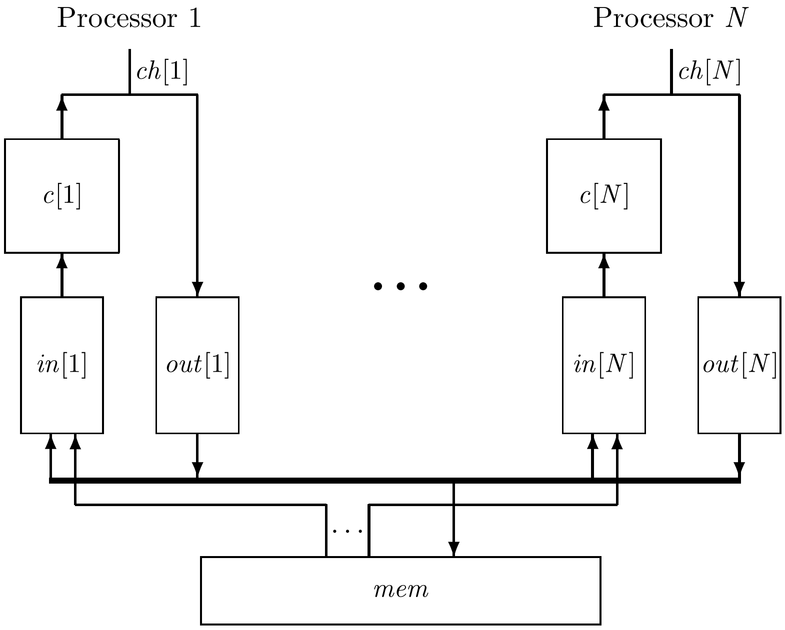

Figure 7: The state machine describing the lazy caching algorithm, where $c[i]$ is a cache, $out[i]$ is a queue of $\langle data, address \rangle$ pairs for writes by processor i that have not yet been performed to memory, and $in[i]$ is a queue of $\langle data, address \rangle$ pairs and $\langle data, address, \text{“}*\text{”} \rangle$ triples of pending writes to $c[i]$, a “$*$” indicating that the write was issued by processor i.

of f, for any cache contents f. Formally, $Restrict(f)$ is the set of all functions g in $[Addr \rightarrow Data \cup \{NotData\}]$ such that $g[a]$ equals $f[a]$ or $NotData$, for all a in $Addr$.

2.2.3 The Initial Condition and Actions

Module *LazyCache* next defines seven formulas that are formal statements of the initial condition and the six event descriptions of Figure 3.

Init The predicate *Init* describes the initial values of the variables c, in, out, and mem. (As we will see from module *ChannelInterface*, the initial value of ch doesn't matter.) The predicate has four conjuncts.[5] The first

[5] TLA$^+$ uses the notation that a list of expressions bulleted by $\land$ denotes their conjunction, and a list of expressions bulleted by $\lor$ denotes their disjunction. Indentation is used to eliminate parentheses [13]. (We also continue to use $\land$ and $\lor$ as infix operators.)

$$
\begin{array}{l}
\text{MODULE } \mathit{Sequences} \\
\text{EXTENDS } \mathit{Naturals} \\
\hline
\mathit{Len}(s) \;\triangleq\; \text{CHOOSE } n : (n \in \mathit{Nat}) \land ((\text{DOMAIN } s) = (1 \,..\, n)) \\
\mathit{Head}(s) \;\triangleq\; s[1] \\
\mathit{Tail}(s) \;\triangleq\; [i \in 1 \,..(\mathit{Len}(s) - 1) \mapsto s[i+1]] \\
s \circ t \;\triangleq\; [i \in 1 \,..(\mathit{Len}(s) + \mathit{Len}(t)) \mapsto \text{IF } \; i \leq \mathit{Len}(s) \;\text{ THEN } \; s[i] \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\;\; \text{ELSE } \; t[i - \mathit{Len}(s)]] \\
\mathit{Seq}(S) \;\triangleq\; \text{UNION } \{[(1 \,..\, n) \to S] : n \in \mathit{Nat}\} \\
\mathit{SelectSeq}(s, \mathit{test}(\_)) \;\triangleq\; \text{LET } \; F[\, t \in \mathit{Seq}(\{s[i] : i \in 1 \,..\, \mathit{Len}(s)\})\,] \;\triangleq \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{IF } \; t = \langle\,\rangle \;\text{ THEN } \; \langle\,\rangle \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\;\; \text{ELSE } \;\text{IF } \; \mathit{test}(\mathit{Head}(t)) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{THEN } \; \langle \mathit{Head}(t) \rangle \circ F[\,\mathit{Tail}(t)] \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{ELSE } \; F[\,\mathit{Tail}(t)] \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{IN } \quad F[s] \\
\mathit{SubSeq}(s, m, n) \;\triangleq\; [i \in 1 \,..(1 + n - m) \mapsto s[i + m - 1]] \\
\hline
\end{array}
$$

Figure 8: Module *Sequences*

asserts that *mem* is the contents of a memory that assigns to each address an element of *InitData*. The second conjunct asserts that c is a function with domain *Proc*, the set of processor names, and that $c[i]$ (the contents of processor i's cache) is a restriction of *mem*, for each processor i. The last two conjuncts assert that *in* and *out* are functions with domain *Proc* such that $in[i]$ and $out[i]$ are the empty sequence, for each processor i.

Read The operator *Read* is defined so that the action $Read(i, d, a)$ corresponds to the description of the event $\mathsf{R}_i(d, a)$ in Gerth's state machine. An *action* is a boolean-valued expression that may contain primed and unprimed variables. It specifies a *step*, which is a pair of states. Unprimed variables refer to the variables' values in the first state of the step; primed variables refer to their values in the second state. A step satisfying $Read(i, d, a)$ represents an $\mathsf{R}_i(d, a)$ event in Gerth's specification.

Action $Read(i, d, a)$ is the conjunction of six formulas. The first asserts that the queue $out[i]$ is empty. The second asserts that there is no "$*$" entry in the queue $in[i]$. More precisely, it asserts that for each positive natural j less than or equal to the length of $in[i]$, the jth element $in[i][j]$ of $in[i]$ is of length 2. The third conjunct asserts that cache $c[i]$ assigns d to address a. These three conjuncts contain no primed variables, so they are conditions on the first state of the step. They correspond to the "allowed if" condition for $\mathsf{R}_i(d, a)$ in Figure 3.

The fourth conjunct of $Read(i, d, a)$ uses the operator $ChanOp$ imported from module $ChannelInterface$. For any variable x and value v, formula $ChanOp(x, v)$ asserts that the values of x and x' describe a step that represents the sending of v over channel x.[6] Thus, this conjunct asserts that the tuple $\langle i, \text{"}\mathsf{Rd}\text{"}, d, a\rangle$, indicating a read of value d from address a by processor i, is sent over $ch[i]$. The precise definition of $ChanOp$, which is given later, does not matter. However, we should check that it defines a $ChanOp(x, v)$ step to be a reasonable representation of the event of sending v on channel x. For example, this would not be the case if a step could satisfy both $ChanOp(x, v)$ and $ChanOp(x, w)$, for $w \neq v$.

The fifth and sixth conjuncts assert what doesn't change. The formula UNCHANGED e means $e' = e$, so the fifth conjunct asserts that the step does not change any other channel—hence nothing is sent on the other channels. The sixth conjunct asserts that the tuple $\langle c, in, out, mem\rangle$ does not change—hence c, in, out, and mem are left unchanged. (In Figure 3, an event whose action does not mention a variable leaves the variable unchanged. In a TLA specification, an action that does not specify the new value of a variable allows the variable to assume any value.)

Write Action $Write(i, d, a)$ corresponds to the description of the state machine event $\mathsf{W}_i(d, a)$. The TLA$^+$ construct $[f \text{ EXCEPT } ![x] = e]$ denotes a function g that is the same as f except with $g[x]$ equal to e. Thus, the second conjunct asserts that the element $\langle d, a\rangle$ is appended to $out[i]$, and $out[j]$ is unchanged for j in the set $Proc \setminus \{i\}$ of other processors.

MemWrite Action $MemWrite(i)$ corresponds to the state machine event $\mathsf{MW}_i(d, a)$ when $\langle d, a\rangle$ is the head of $out[i]$, which is the only case in which that event is allowed. Letting d be $Head(out[i])[1]$ and a be $Head(out[i][2])$ (so $\langle d, a\rangle$ is the head of $out[i]$), the action asserts that $out[i]$ is nonempty, its head is removed, the triple $\langle d, a, \text{"}*\text{"}\rangle$ is appended to $in[i]$, the pair $\langle d, a\rangle$ is appended to all the other queues $in[j]$, and $mem[a]$ is set to d.

CacheUpdate Action $CacheUpdate(i)$ similarly corresponds to state machine event $\mathsf{CU}_i(d, a)$ when the head of $in[i]$ is $\langle d, a\rangle$ or $\langle d, a, \text{"}*\text{"}\rangle$, the

[6]There is no notion of a sender or a receiver, so it might be better to say that the step represents a v event on channel x.

only case in which the event is allowed. The tuple is removed from the head of $in[i]$ and the cache $c[i]$ is updated accordingly.

MemRead A step allowed by action $MemRead(i)$ corresponds to a state machine event $\mathsf{MR}_i(d, a)$ for some d and a. This event is allowed only when a is an address and d equals $mem[a]$. It appends the pair $\langle mem[a], a\rangle$ to $in[i]$ and leaves everything else unchanged.

CacheInval A step allowed by action $CacheInval(i)$ corresponds to a CI_i state machine event. It sets $c[i]$ to some restriction of its original value.

2.2.4 The Complete Specification

The definitions described thus far capture all the information explicit in Figure 3 except for the fairness condition. To write the complete TLA specification, we must also express what is implicit in that figure—the range of i, d, a and the fact that those six events are the only ones allowed.

The TLA specification corresponding to a state machine has the canonical form $I \wedge \Box[N]_v \wedge L$, where I is the initial predicate, N is the next-state action, v is the tuple of all relevant variables, and L is a fairness condition. A behavior satisfies this formula iff I holds in the initial state, every successive pair of states is a step that either satisfies N or else leaves v unchanged, and the fairness condition L is satisfied. (The reason for allowing "stuttering steps" that do not change v is explained in [10].) For the lazy caching algorithm, I is the initial predicate $Init$. We next describe the next-state action N, which describes all possible events of the state machine—that is all steps that change v.

We first describe all possible processor i events—ones subscripted by i in Figure 3. These events correspond to steps of action $Next(i)$ of module *LazyCache*. A step satisfies this action iff it satisfies $Read(i, d, a)$ or $Write(i, d, a)$ for some d in $Data$ and a in $Addr$, or satisfies $CacheUpdate(i)$, $MemWrite(i)$, $MemRead(i)$, or $CacheInval(i)$. Our specification should assert that every nonstuttering step is a $Next(i)$ step, for some processor i. Thus, it equals $Init \wedge \Box[\exists\, i \in Proc : Next(i)]_{vars} \wedge L$, where $vars$ is the tuple of relevant variables and L expresses the desired fairness conditions. We now describe L.

Gerth required that all the events in Figure 3 except CI_i satisfy a *weak fairness condition*—that is, if the event is continuously enabled, then it must eventually occur. Weak fairness is expressed in TLA with the formula $\mathrm{WF}_v(A)$, which asserts that if an A action that changes v is continuously enabled, then a step satisfying that action must eventually occur. We find it unnatural to require processors to keep executing operations forever, so we place no fairness requirements on the *Read* and *Write* actions. Because of the simplifications Gerth made to the algorithm, fairness of MR events is not needed to prove its correctness. We therefore place no fairness requirement on *MemRead* actions. We require weak fairness of only the actions $CacheUpdate(i)$ and $MemWrite(i)$, for every processor i.[7]

The complete specification of the lazy caching algorithm is given by formula *Spec* of module *LazyCache*.

Observe that there are no type declarations in TLA$^+$. Type correctness of our specification *Spec* is expressed by the following theorem, which asserts that the variables c, in, out, and mem are always elements of the proper set. (Module *Sequences* defines $Seq(S)$ to be the set of all finite sequences of elements in S, and $\Box$ is the usual "always" operator of temporal logic [21].)

$$
\begin{aligned}
Spec \Rightarrow \Box\,(\; & \land\; mem \in [Addr \rightarrow Data] \\
& \land\; c \in [Proc \rightarrow [Addr \rightarrow Data \cup \{NotData\}]] \\
& \land\; in \in [Proc \rightarrow Seq((Data \times Addr) \cup (Data \times Addr \times \{\text{“}*\text{”}\}))] \\
& \land\; out \in [Proc \rightarrow Seq(Data \times Addr)]\;\;)
\end{aligned}
$$

This theorem is easy to prove; the proof steps are similar to the ones performed in conventional type checking.

3 Sequential Consistency

We now specify sequential consistency for an arbitrary database. (A memory is a database in which the only operations are reading and writing.) We specify a serial database in Section 3.1 and use that specification in Section 3.2 to specify a sequentially consistent database. In Section 3.3, we give an equivalent specification in the spirit of Gerth's.

3.1 A Serial Database

Module *SerialDB* of Figure 9 specifies a serial database that communicates with its environment by means of a communication channel. The module first imports module *ChannelInterface*, where the operator *ChanOp* is defined. It then declares the following parameters: a variable *dch* that represents the communication channel; a constant *Op* that represents the set of

[7]Gerth has separate fairness requirements on $\mathsf{CU}_i(d, a)$ and $\mathsf{MW}_i(d, a)$ for each d and a, but we find it more convenient just to require fairness of $CacheUpdate(i)$ and $MemWrite(i)$. It is not hard to show that the two sets of conditions are equivalent.

MODULE *SerialDB*

EXTENDS *ChannelInterface*

VARIABLE *dch*

CONSTANT *Op*, *InitDB*, *OKOp*(_ , _ , _)

$$INext(mem) \triangleq \exists\, o \in Op : ChanOp(dch, o) \land OKOp(o, mem, mem')$$

$$ISpec(mem) \triangleq (mem \in InitDB) \land \Box[INext(mem)]_{\langle mem,\, dch \rangle}$$

$$Spec \triangleq \boldsymbol{\exists}\, mem : ISpec(mem)$$

Figure 9: Module *SerialDB*.

possible database operations; a constant *InitDB* that represents the set of legal initial values of the database; and an operator *OKOp* that describes the legal database operations. We interpret $OKOp(o, old, new)$ equal to TRUE to mean that operation o is a legal operation when the value of the database before the operation is *old* and the value after the operation is *new*. Module *CacheCorrectness* in Figure 16 (Section 4.1) defines *OKOp* for a memory.

The module next defines $ISpec(mem)$ to be a formula in the canonical form $I \land \Box[N]_v$ that specifies a serial database whose state is represented by the variable *mem*. (Since database operations need never occur, there is no fairness requirement.) Action $OKOp(o, mem, mem')$ asserts that operation o is allowed to change the database from the old value *mem* to the new value mem'. The next-state action $INext(mem)$ therefore asserts that some operation o in *Op* is sent on channel *dch* and makes a legal change to the database value *mem*.

Formula $ISpec(mem)$ contains the free variable *mem*. The specification of our database should describe only the sequence of operations sent on channel *dch*; it should not mention any other variable. Hence, *dch* is the only variable parameter of module *SerialDB*, which is why we had to introduce *mem* as a formal parameter of *ISpec*. Our specification of the serial database is formula $ISpec(mem)$ with the variable *mem* hidden. In TLA, variables are hidden with the temporal existential quantifier $\boldsymbol{\exists}$. Formula *Spec* of module *SerialDB* therefore specifies a serial database with channel *dch* described by the constant parameters *Op*, *InitDB*, and *OKOp*.

$$
\begin{array}{l}
\textsc{module } \mathit{SeqDBParams} \\
\hline
\textsc{extends } \mathit{Naturals},\ \mathit{Sequences} \\
\textsc{variable } \mathit{ch} \\
\textsc{constant } N,\ \mathit{POp},\ \mathit{InitDB},\ \mathit{OKOp}(\_,\_,\_) \\
\hline
\mathit{Proc} \ \triangleq\ 1 \,..\, N \\
\mathit{Op} \ \triangleq\ \textsc{union } \{\mathit{POp}[i] : i \in \mathit{Proc}\} \\
\hline
\textsc{assume } \mathit{NAssump} \ \triangleq\ (N \in \mathit{Nat}) \land (N > 0) \\
\textsc{assume } \mathit{OpsDisjoint} \ \triangleq\ \forall\, i, j \in \mathit{Proc} : (i \neq j) \Rightarrow (\mathit{POp}[i] \cap \mathit{POp}[j] = \{\}) \\
\hline
\end{array}
$$

Figure 10: Parameters for the specifications of a sequentially consistent database.

3.2 Our Specification of Sequential Consistency

3.2.1 Common Parameters

We put declarations and definitions that are common to both specifications of sequential consistency into module *SeqDBParams*, shown in Figure 10. The parameters *InitDB* and *OKOp* have the same interpretation as in module *SerialDB*. Module *SeqDBParams*'s other parameters are: a variable *ch* that represents an array of channels, where processor i communicates with the database over channel $ch[i]$; a constant N that represents the number of processors; and a constant *POp* representing an array of sets, where $POp[i]$ is the set of operations that can be performed by processor i.

The module defines *Proc* to be the set $\{1, \ldots, N\}$ of processors and *Op* to be the set of all operations—that is, the union of the sets $POp[i]$, for all processors i. The set *Op* plays the role of the parameter of the same name in module *SerialDB*.

The module next states two assumptions. Assumption *NAssump* asserts that N is a positive natural. For convenience, we assume that the operations sent by different processors are different. Formally, this means that $POp[i]$ and $POp[j]$ are disjoint sets of operations, if i and j are different processors. Module *SeqDBParams* asserts, and assigns the name *OpsDisjoint* to, this assumption.

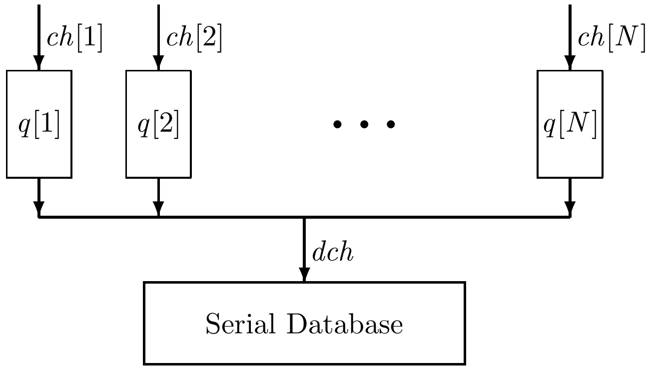

Figure 11: A state machine for specifying sequential consistency.

3.2.2 Sequential Consistency

Intuitively, sequential consistency means that there is some interleaving of the operations sent on channels $ch[1]$, ..., $ch[N]$ that forms a correct sequence of operations for a serial database. The idea behind our specification is illustrated by Figure 11. Consider a state machine that performs the following operations, for each processor i:

Enq Send an operation from $POp[i]$ on channel $ch[i]$ and append that operation to the tail of queue $q[i]$.

Deq Remove the operation from the head of $q[i]$ and send it on channel dch.

Let the state machine satisfy weak fairness of the *Deq* action for each i, which implies that every operation that is put into a queue by an *Enq* operation is eventually removed by a *Deq*. It is clear that the operations sent on the channels $ch[i]$ are sequentially consistent if the sequence of operations sent on channel dch satisfies the specification of a serial database. Moreover, for any sequentially consistent operations sent on the $ch[i]$, there is some way of performing the *Deq* operations that makes the operations sent on dch form a correct execution of the serial database. In other words, the operations on the $ch[i]$ are sequentially consistent iff there is some sequence of values assumed by the queues $q[i]$ and the channel dch that is a correct execution of the state machine and satisfies the specification of the serial database. Let *QSpec* be the TLA formula describing the state machine, and let *Spec* be the specification of the serial database from module *SerialDB*. The operations on the $ch[i]$ are sequentially consistent iff the formula $\exists\, dch, q : QSpec \land Spec$ is satisfied. This formula is the TLA specification of sequential consistency.

$$
\begin{array}{l}
\hline
\textsc{module } \mathit{SeqDB1} \\
\textsc{extends } \mathit{SeqDBParams},\ \mathit{ChannelInterface},\ \mathit{Sequences} \\
\hline
\mathit{SDB}(\mathit{dch}) \ \triangleq\ \textsc{instance } \mathit{SerialDB} \\
\mathit{Enq}(i, \mathit{dch}, q) \ \triangleq\ \exists\, o \in \mathit{POp}[i] : \\
\qquad\qquad \land\ \mathit{ChanOp}(\mathit{ch}[i], o) \\
\qquad\qquad \land\ q[i]' = q[i] \circ \langle o \rangle \\
\qquad\qquad \land\ \textsc{unchanged } \mathit{dch} \\
\qquad\qquad \land\ \forall\, j \in \mathit{Proc} \setminus \{i\} : \textsc{unchanged } \langle q[j], \mathit{ch}[j] \rangle \\
\mathit{Deq}(i, \mathit{dch}, q) \ \triangleq\ \land\ q[i] \neq \langle\,\rangle \\
\qquad\qquad \land\ \mathit{ChanOp}(\mathit{dch}, \mathit{Head}(q[i])) \\
\qquad\qquad \land\ q[i]' = \mathit{Tail}(q[i]) \\
\qquad\qquad \land\ \textsc{unchanged } \mathit{ch}[i] \\
\qquad\qquad \land\ \forall\, j \in \mathit{Proc} \setminus \{i\} : \textsc{unchanged } \langle q[j], \mathit{ch}[j] \rangle \\
\mathit{QSpec}(\mathit{dch}, q) \ \triangleq\ \land\ \forall\, i \in \mathit{Proc} : q[i] = \langle\,\rangle \\
\qquad\qquad \land\ \Box[\exists\, i \in \mathit{Proc} : \mathit{Enq}(i, \mathit{dch}, q) \lor \mathit{Deq}(i, \mathit{dch}, q)]_{\langle \mathit{ch}, \mathit{dch}, q \rangle} \\
\qquad\qquad \land\ \forall\, i \in \mathit{Proc} : \mathrm{WF}_{\langle \mathit{ch}, \mathit{dch}, q \rangle}(\mathit{Deq}(i, \mathit{dch}, q)) \\
\mathit{Spec} \ \triangleq\ \boldsymbol{\exists}\, \mathit{dch}, q : \mathit{QSpec}(\mathit{dch}, q) \land \mathit{SDB}(\mathit{dch})!\mathit{Spec} \\
\hline
\end{array}
$$

Figure 12: A specification of sequential consistency.

The TLA$^+$ version of the specification appears in module *SeqDB*1 of Figure 12. The module imports *SeqDBParams*, which contains the specification's parameters, the assumption *OpsDisjoint*, and the definition of *Op*. It also imports modules *ChannelInterface* (for the definition of *ChanOp*) and *Sequences* (for the definitions of operations on sequences).

Our specification uses the specification of a serial database, which appears in module *SerialDB*. Simply importing that module would import its parameters *dch* and *Op*, which should not be parameters of *SeqDB*1. (Channel *dch* is an internal variable of the sequentially consistent database; the set *Op* is defined in terms of *POp*, so it is not a parameter.) Instead, we want to include the definitions from *SerialDB* with the module's parameters instantiated as follows: *Op* instantiated with the set of the same name defined in the imported module *SeqDBParams*; *InitDB* and *OKOp* instantiated with the parameters of the same name imported from *SeqDBParams*; and *dch* replaced by an explicit formal parameter. The statement

$$\mathit{SDB}(\mathit{dch}) \ \triangleq\ \textsc{instance } \mathit{SerialDB}$$

includes the definitions from module *SerialDB*, with the names of all defined operators prefixed by "$SDB(dch)!$", and with the aforementioned instantiations.[8] For example, $SDB(d)!INext(m)$ equals

$$\exists\, o \in Op \,:\, SDB(d)!ChanOp(d, o) \land OKOp(o, m, m')$$

for any expressions d and m, where $SDB(d)!ChanOp$ equals the operator *ChanOp* (imported by module *SerialDB*) from module *ChannelInterface*. The symbols *Op* and *OKOp* are not prefixed by "$SDB(\ldots)!$" because they are parameters, not defined operators.

Module *SeqDB*1 next defines the TLA formula that specifies the state machine pictured in Figure 11, excluding the box labeled "Serial Database". Since the variables *dch* and q are not parameters of the module, they must be explicit parameters of the definition. Formula $QSpec(dch, q)$ is the canonical-form TLA formula that describes the state machine.

Finally, formula *Spec* is the complete specification of sequential consistency. More precisely, it specifies the sequentially consistent system with array *ch* of channels described by the parameters *Proc*, *POp*, *InitDB*, and *OKOp*.

3.2.3 A Closer Look at the Specification

Our specification *Spec* of module *SeqDB*1 looks deceptively simple. However, we shall now show that it cannot be written as a conventional state machine.

Expanding the definition of $SDB(dch)!Spec$ shows that *Spec* equals

$$\boldsymbol{\exists}\, dch, q \,:\, (QSpec(dch, q) \;\land\; \boldsymbol{\exists}\, mem \,:\, SDB(dch)!ISpec(mem))$$

Since *mem* does not occur in $QSpec(dch, q)$, this is equivalent to

$$\boldsymbol{\exists}\, dch, q, mem \,:\, (QSpec(dch, q) \land SDB(dch)!ISpec(mem))$$

Formulas $QSpec(dch, q)$ and $SDB(dch)!ISpec(mem)$ are both in canonical form. Since $\Box$ distributes over conjunction, a straightforward calculation allows us to rewrite $QSpec(dch, q) \land SDB(dch)!ISpec(mem)$ in the canonical form $I \land \Box[N]_v \land L$, where the next-state action N is

$$\begin{aligned}\exists\, i \in Proc \,:\, &\lor\; Enq(i, dch, q) \land (mem' = mem)\\ &\lor\; Deq(i, dch, q) \land SDB(dch)!INext(mem)\end{aligned}$$

[8] In the absence of explicit instantiation, a parameter is instantiated by the symbol of the same name. We have chosen our names to avoid having to introduce the TLA$^+$ construct for explicit instantiation of parameters.

v is the tuple $\langle ch, dch, q, mem \rangle$, and L is the conjunction of weak fairness conditions on $Deq(i, dch, q)$ actions.

Formula $I \wedge \Box[N]_v$ is the TLA representation of the state machine of Figure 11, including the "Serial Database" box. However, L does not correspond to any conventional fairness condition on state machines. The next-state action N allows a Deq action to occur only when it is also an $SDB(dch)!INext(mem)$ action. In other words, a Deq step can occur only if it sends a correct serial database operation on dch. However, weak fairness on $Deq(i, dch, q)$ requires that this operation must eventually be performed if $q[i]$ is nonempty, regardless of whether or not doing so would violate the serial database specification for dch. This requirement is not a conventional state-machine fairness condition. TLA formulas are more expressive than state machines.

Viewed as a machine, our specification is bizarre. The machine is allowed to perform any arbitrary operation on $ch[i]$, regardless of its legality. However, when the operation reaches the head of $q[i]$, it must eventually be a correct operation for the serial database. In the case of a sequentially consistent memory, a read of address a on $ch[i]$ may return any arbitrary value d. However, at some time after the read operation reaches the head of $q[i]$, the value of $mem[a]$ must equal d. That value could have been written by an operation on another channel $ch[j]$ that occurred after the read.

In the terminology of [1], our specification is not machine closed. Conventional state-machine specifications are always machine closed. Machine closure is a necessary condition for a specification to be executable in practice. The description of an algorithm should be machine closed, but a high-level specification need not be.

3.3 A Gerth-Like Specification of Sequential Consistency

3.3.1 Channels

Gerth specified sequential consistency directly in terms of the sequence of values transmitted over a channel. TLA formulas are written in terms of variables that describe the current state. To express Gerth's specification in TLA, we must introduce a *history* variable for each channel to record the sequence of values that have been sent over the channel. This is done in module *ChannelInterface* of Figure 13, which defines the following two operators.

MODULE *ChannelInterface*

EXTENDS *Sequences, Naturals*

$$
\begin{array}{lcl}
\mathit{ChanOp}(c, v) & \triangleq & c' = \langle v, \text{ IF } c[2] = \text{"a" THEN "b" ELSE "a"} \rangle \\
\mathit{ChanHist}(c, h) & \triangleq & \land\ h = \langle\,\rangle \\
 & & \land\ \Box[h' = \text{ IF } c' \neq c \text{ THEN } h \circ \langle c'[1] \rangle \text{ ELSE } h]_{\langle h, c\rangle}
\end{array}
$$

Figure 13: Module *ChannelInterface*

ChanOp A step satisfying action $\mathit{ChanOp}(c, v)$ must represent the event of sending the value v on a channel represented by the variable c. The obvious way to represent this event is by setting the value of c to v. However, this won't work because sending the same value v again would result in no change to c, and no change to c must represent nothing being sent on the channel. We therefore define $\mathit{ChanOp}(c, v)$ to set c to a pair $\langle v, \ldots \rangle$, where the second component is chosen to ensure that the new value of c does not equal its old value. There are any number of ways to do this. In module *ChannelInterface*, we let $\mathit{ChanOp}(c, v)$ set the second component alternately to "a" and "b". More precisely, $\mathit{ChanOp}(c, v)$ asserts that c' equals $\langle v, \text{"a"} \rangle$ unless the second component of c equals "a", in which case c' equals $\langle v, \text{"b"} \rangle$.[9]

ChanHist The temporal formula $\mathit{ChanHist}(c, h)$ asserts that the value of h always equals the sequence of values that have been sent over c. Two alternative definitions are

$$(h = \langle\,\rangle) \land \Box[\exists\, v : \mathit{ChanOp}(c, v) \land (h' = h \circ \langle v \rangle)]_{\langle h, c\rangle}$$

$$(h = \langle\,\rangle) \land \Box\begin{bmatrix} \land\ \forall\, v : \mathit{ChanOp}(c, v) \Rightarrow (h' = h \circ \langle v \rangle) \\ \land\ (c' = c) \Rightarrow (h' = h) \end{bmatrix}_{\langle h, c\rangle}$$

Using the TLA proof rules [14, Figure 5], it is easy to show that the formula $\Box[\exists\, v : \mathit{ChanOp}(c, v)]_c$, which asserts that every change to c is a *ChanOp* event, implies that all three definitions are equivalent.

Representing Channels Our method of representing channels is artificial; the use of pairs and the values "a" and "b" were completely arbitrary. The definition of *ChanOp* is artificial because the interface itself does not

[9] In TLA$^+$, $c[2]$ is some value—even if 2 is not in the domain of c. Hence, a specification need not specify any initial value for the channel.

correspond to any real form of communication. Asynchronous communication requires two separate events—the sending of a value and its acknowledgement. In a real memory, a read operation consists of a processor request (which may cause the cache to be updated) followed by the memory's response. The more realistic interface has a more natural representation. Gerth chose the artificial interface to simplify the problem. The simpler interface makes it impossible to specify some correctness properties of a real memory—in particular, the property that every request is eventually followed by a reply.

A more elegant description of a channel can be obtained by writing axioms for *ChanOp* instead of defining it explicitly. However, lists of axioms are notoriously difficult to understand. It is easy to write an incomplete specification by omitting axioms. Few people would think of including the axiom $\mathit{ChanOp}(c, v) \Rightarrow (c' \neq c)$ when specifying *ChanOp*. We believe that a clear, inelegant definition is better than an obscure, elegant list of axioms.

3.3.2 The Gerth-Like Specification

Gerth's specification of a sequentially consistent database with array ch of channels essentially asserts that there exists a serial database with channel dch such that, for each processor i, the sequence of operations sent on $ch[i]$ equals the sequence of operations from processor i sent on dch. Since $\mathit{ChanHist}(c, h)$ asserts that h is the history of operations sent on channel c, if $\mathit{SDB}(dch)!\mathit{Spec}$ is the specification of a serial database with channel dch, then Gerth's specification is[10]

$$\begin{aligned} \boldsymbol{\exists}\, dch \;:\; & \wedge\ \mathit{SDB}(dch)!\mathit{Spec} \\ & \wedge\ \forall\, i \in \mathit{Proc} \;:\; \boldsymbol{\forall}\, hch, hdch \;: \\ & \qquad \mathit{ChanHist}(dch, hdch) \wedge \mathit{ChanHist}(ch[i], hch) \Rightarrow G \end{aligned}$$

where G asserts the appropriate relation between the history variable hch for $ch[i]$ and the history variable $hdch$ for dch. Our problem is writing G in temporal logic.

Since channel dch is internal, it doesn't matter when the operations appear there. Without loss of generality, we can let G require that operations not appear on dch before they appear on $ch[i]$. Gerth's specification is then obtained by requiring that the sequence of operations sent at any time on channel $ch[i]$ eventually equals the sequence of operations by processor i

[10] Universal quantification over flexible variables is defined by $\boldsymbol{\forall}\, x : F \triangleq \neg \boldsymbol{\exists}\, x : \neg F$.

MODULE *SeqDB2*

EXTENDS *SeqDBParams, Sequences, ChannelInterface, Naturals*

$$
\begin{array}{l}
SDB(dch) \;\triangleq\; \text{INSTANCE } SerialDB \\
Proj(i, s) \;\triangleq\; \text{LET } Test(e) \;\triangleq\; e \in POp[i] \text{ IN } SelectSeq(s, Test) \\
Spec \;\triangleq\; \exists\!\!\!\exists\, dch : \land\ SDB(dch)!Spec \\
\qquad\qquad\qquad\quad \land\ \forall i \in Proc : \forall\!\!\!\forall\, hch, hdch : \\
\qquad\qquad\qquad\qquad\quad ChanHist(dch, hdch) \land ChanHist(ch[i], hch) \Rightarrow \\
\qquad\qquad\qquad\qquad\qquad \forall os : \Box((hch = os) \Rightarrow \Diamond(os = Proj(i, hdch)))
\end{array}
$$

Figure 14: A history-based specification of sequential consistency.

MODULE *DB1equivDB2*

EXTENDS *SeqDBParams*

$$
\begin{array}{l}
DB1 \;\triangleq\; \text{INSTANCE } SeqDB1 \\
DB2 \;\triangleq\; \text{INSTANCE } SeqDB2 \\
\text{THEOREM } Thm \;\triangleq\; DB1!Spec \equiv DB2!Spec
\end{array}
$$

Figure 15: The two specifications of sequential consistency are equivalent.

that have been sent on *dch*. This assertion can be written

$$\forall os : \Box((hch = os) \Rightarrow \Diamond(os = Proj(i, hdch)))$$

where $Proj(i, \sigma)$ is the subsequence of operations in σ sent by processor i, and $\Diamond$ is the usual temporal operator meaning *eventually* [21]. The complete specification appears as formula *Spec* of module *SeqDB*2 in Figure 14. It uses the operator *SelectSeq*, which is defined in module *Sequences* so that $SelectSeq(s, Test)$ is the subsequence of s consisting of all elements e with $Test(e)$ equal to TRUE.

3.3.3 Relating the two Specifications

We hope it is intuitively clear that the two specifications of sequential consistency, formulas *Spec* of modules *SeqDB*1 and *SeqDB*2, both allow the same sets of behaviors for the array *ch* of channels. Formally, this means that the two formulas are equivalent. Their equivalence is expressed in TLA$^+$ by the theorem named *Thm* of module *DB*1*equivDB*2 in Figure 15.

We will prove that the lazy caching algorithm implements our specification of sequential consistency. Formally, this means proving that formula *Spec* of module *LazyCache* implies formula *Spec* of module $SeqDB1$. To prove that the algorithm also implements the Gerth-like specification, it suffices to prove that formula *Spec* of module $SeqDB1$ implies formula *Spec* of module $SeqDB2$, which is half of the theorem in module $DB1equivDB2$.

4 The Proof

Our goal here is not to convince the reader that the lazy caching algorithm is correct, but to indicate how a convincing proof is obtained. Naive readers may be convinced by a nonrigorous proof, but published "proofs" of incorrect concurrent algorithms have demonstrated the need for rigor. Rigorous proofs are long, detailed, and tedious. They are difficult to present on paper and are best suited to hypertext. We therefore just describe how the proof is obtained. Section 4.1 states our result formally and describes the outline of the proof. Sections 4.2–4.4 specify two intermediate algorithms and sketch the more interesting parts of the proof.

4.1 Outline of the Proof

We might expect sequential consistency to mean that the algorithm's specification, formula *Spec* of module *LazyCache*, implies formula *Spec* of module $SeqDB1$, which is our specification of sequential consistency. However, those two formulas have different parameters. The algorithm's memory is described by the parameters $Data$, $InitData$, and $Addr$, while sequential consistency is defined in terms of a more general database specified by the parameters POp, $InitDB$, and $OKOp$. Module *CacheCorrectness* in Figure 16 imports and declares the same parameters as the *LazyCache* module, and then includes module *LazyCache* as LC. It next defines the constants POp, $OKOp$, and $InitDB$ and includes $SeqDB1$ as $DB1$, implicitly substituting these three constants for the parameters of the same name in $SeqDB1$. The module then asserts theorem $LCimpliesDB1$, which expresses formally the sequential consistency of the lazy caching algorithm.

We split the proof of theorem $LCimpliesDB1$ into two parts by introducing an intermediate algorithm, called the complete cache. The complete cache is specified by formula *Spec* of module *CCache*, which appears in Section 4.2 below. The *CacheCorrectness* module includes this module and then asserts that the lazy caching algorithm implements the complete cache

MODULE *CacheCorrectness*

EXTENDS *MemParams*

VARIABLE c, in, out, mem

$LC \triangleq$ INSTANCE *LazyCache*

$POp \triangleq [i \in Proc \mapsto \{i\} \times \{\text{"Rd"}, \text{"Wr"}\} \times Data \times Addr]$

$$
\begin{aligned}
OKOp(o, old, new) \triangleq\ & \lor \land o[2] = \text{"Rd"} \\
& \quad \land o[3] = old[o[4]] \\
& \quad \land new = old \\
& \lor \land o[2] = \text{"Wr"} \\
& \quad \land new = [old \text{ EXCEPT } ![o[4]] = o[3]]
\end{aligned}
$$

$InitDB \triangleq [Addr \rightarrow InitData]$

$DB1 \triangleq$ INSTANCE *SeqDB1*

THEOREM $LCimpliesDB1 \triangleq LC!Spec \Rightarrow DB1!Spec$

$CC(cc, cin, cout) \triangleq$ INSTANCE *CCache*

THEOREM $LCimpliesCC \triangleq$

$LC!Spec \Rightarrow (\exists\, cc, cin, cout : CC(cc, cin, cout)!Spec)$

THEOREM $CCimpliesDB1 \triangleq$

$\forall\, cc, cin, cout : CC(cc, cin, cout)!Spec \Rightarrow DB1!Spec$

$ACC(cc, cin, cout, vcq, vrq, vdch) \triangleq$ INSTANCE *ACCache*

THEOREM $CCequivACC \triangleq$

$\forall\, cc, cin, cout : CC(cc, cin, cout)!Spec \equiv$

$(\exists\, vcq, vrq, vdch : ACC(cc, cin, cout, vcq, vrq, vdch)!ASpec)$

THEOREM $ACCimpliesDB1 \triangleq$

$\forall\, cc, cin, cout, vcq, vrq, vdch :$

$ACC(cc, cin, cout, vcq, vrq, vdch)!ASpec \Rightarrow DB1!Spec$

Figure 16: The theorems constituting the correctness proof.

(theorem *LCimpliesCC*) and that the complete cache is sequentially consistent (theorem *CCimpliesDB*1). The complete cache specification has three additional variable parameters, *cc*, *cin*, and *cout*, that are instantiated by parameters and are quantified in the statements of the theorems. As with ordinary first-order quantification, the formula $\forall x : F$ is valid iff F is. (The $\forall$ is needed in theorem *CCimpliesDB*1 because module parameters are the only free variables allowed by TLA$^+$ in a theorem.) By simple logic, theorems *LCimpliesCC* and *CCimpliesDB*1 imply theorem *LCimpliesDB*1.

We sketch the proof of the more interesting of these two theorems, *CCimpliesDB*1. The temporal existential quantifiers in the definition of *DB*1!*Spec* mean that the proof requires a refinement mapping [1]. To define the refinement mapping, we add to the complete cache three auxiliary variables, *vcq*, *vrq*, and *vdch*. The complete cache with auxiliary variables is specified by formula *ASpec* of module *ACCache*, which appears in Section 4.3. The *CacheCorrectness* module includes this module and then asserts two theorems. The first, *CCequivACC*, states that formula *ASpec* with the auxiliary variables hidden is equivalent to the complete cache specification. This is what it means for *vcq*, *vrq*, and *vdch* to be auxiliary variables. The second theorem states the sequential consistency of the complete cache with auxiliary variables. These two theorems imply *CCimpliesDB*1.

To prove that the lazy caching algorithm is sequentially consistent, we must prove theorems *LCimpliesCC*, *CCequivACC*, and *ACCimpliesDB*1 of module *CacheCorrectness*. Proving this for a Gerth-like definition of sequential consistency requires also proving the implication *DB*1!*Spec* $\Rightarrow$ *DB*2!*Spec* of *Thm* in module *DB*1*equivDB*2 (Figure 15). The equivalence of the two specifications of sequential correctness, while interesting, has nothing to do with caching algorithms. For reasons discussed below, the implementation of the complete cache by the lazy caching algorithm is not as interesting as the sequential consistency of the complete cache. We will therefore consider here only the proofs of *CCequivACC* and *ACCimpliesDB*1.

4.2 The Complete Cache

The subtle aspect of the lazy caching algorithm is its handling of read and write requests. The invalidation and refreshing of cache entries (actions *CacheInval*(i) and *MemRead*(i)) are standard. We construct the *complete cache* algorithm that describes the execution of read and write requests, but not the invalidation and refreshing of cache entries. We can then prove that the complete cache is implemented by the lazy caching algorithm (theorem *LCimpliesCC*) and implements the specification of sequential consistency (theorem $CCimpliesDB1$).

In the complete cache algorithm, each cache is a complete memory (a total function from locations to data values); there is no shared memory and no cache invalidation or refresh action. The algorithm is pictured as a state machine in Figure 17. To simplify the proof, we replace the queues $out[i]$ and $in[i]$ of the lazy caching algorithm with queues $cout[i]$ and $cin[i]$ containing complete operations $\langle i, op, d, a \rangle$, where i is the processor that issued the operation, op is the operation ("Rd" or "Wr"), d, is the data value, and a is the address. A $*$-ed entry in $c[i]$ becomes an operation in $cin[i]$ whose first component equals i. Some of these components are redundant—in particular, the queues contain only write operations, and $cout[i]$ contains only operations of processor i. All the caches $cc[i]$ have the same initial memory contents, and the cin and $cout$ queues are initially empty. The TLA$^+$ specification of this state machine appears in module $CCache$ of Figure 18.

Theorem $LCimpliesCC$ of module $CacheCorrectness$, which asserts that the lazy caching algorithm implements the complete cache, essentially proves the correctness of the algorithm's cache invalidation and refreshing operations. Since these operations are standard, the proof of the theorem does not reveal anything interesting about the lazy caching algorithm. Here, we consider only the proof of theorem $CCimpliesDB1$, which asserts the sequential

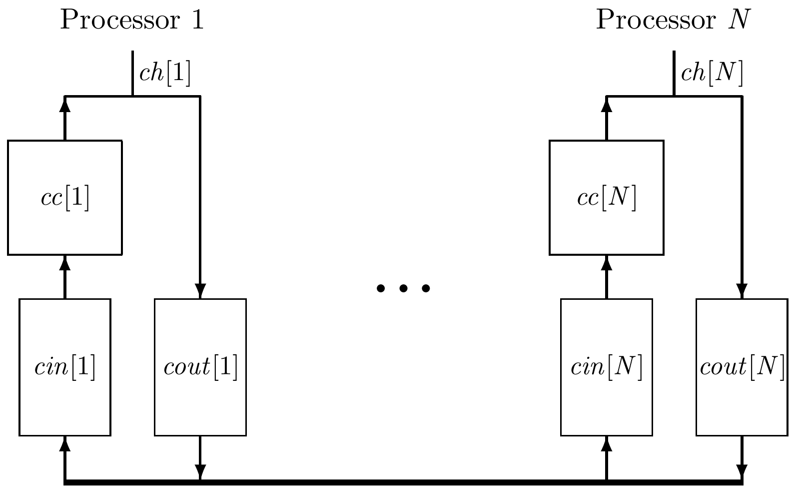

Figure 17: The state machine $CCache$ describing the complete cache algorithm, where $cc[i]$ is a complete cache, and $cout[i]$ and $cin[i]$ are queues of $\langle processor, operation, data, address \rangle$ tuples.

$$\text{MODULE } \mathit{CCache}$$

$\textsc{extends}$ *MemParams, ChannelInterface, Sequences, Naturals*

$\textsc{variable}$ *cc, cin, cout*

$$\begin{array}{lcl}
\mathit{Init} & \triangleq & \land\ cc \in [\mathit{Proc} \rightarrow [\mathit{Addr} \rightarrow \mathit{InitData}]] \\
& & \land\ \forall\, i, j \in \mathit{Proc} \,:\, cc[i] = cc[j] \\
& & \land\ cin = [i \in \mathit{Proc} \mapsto \langle\,\rangle] \\
& & \land\ cout = [i \in \mathit{Proc} \mapsto \langle\,\rangle]
\end{array}$$

$$\begin{array}{lcl}
\mathit{Read}(i, d, a) & \triangleq & \land\ cout[i] = \langle\,\rangle \\
& & \land\ \forall\, j \in (1 \,..\, \mathit{Len}(cin[i])) \,:\, cin[i][j][1] \neq i \\
& & \land\ cc[i][a] = d \\
& & \land\ \mathit{ChanOp}(ch[i], \langle i, \text{“Rd”}, d, a\rangle) \\
& & \land\ \forall\, j \in \mathit{Proc} \setminus \{i\} \,:\, \textsc{unchanged}\ ch[j] \\
& & \land\ \textsc{unchanged}\ \langle cc, cin, cout\rangle
\end{array}$$

$$\begin{array}{lcl}
\mathit{Write}(i, d, a) & \triangleq & \land\ \mathit{ChanOp}(ch[i], \langle i, \text{“Wr”}, d, a\rangle) \\
& & \land\ cout' = [cout \ \textsc{except}\ ![i] = cout[i] \circ \langle\langle i, \text{“Wr”}, d, a\rangle\rangle] \\
& & \land\ \forall\, j \in \mathit{Proc} \setminus \{i\} \,:\, \textsc{unchanged}\ ch[j] \\
& & \land\ \textsc{unchanged}\ \langle cc, cin\rangle
\end{array}$$

$$\begin{array}{lcl}
\mathit{MemWrite}(i) & \triangleq & \land\ cout[i] \neq \langle\,\rangle \\
& & \land\ cout' = [cout \ \textsc{except}\ ![i] = \mathit{Tail}(cout[i])] \\
& & \land\ cin' = [j \in \mathit{Proc} \mapsto cin[j] \circ \langle \mathit{Head}(cout[i])\rangle] \\
& & \land\ \textsc{unchanged}\ \langle cc, ch\rangle
\end{array}$$

$$\begin{array}{l}
\mathit{CacheUpdate}(i) \ \triangleq \\
\quad \land\ cin[i] \neq \langle\,\rangle \\
\quad \land\ cin' = [cin \ \textsc{except}\ ![i] = \mathit{Tail}(cin[i])] \\
\quad \land\ cc' = [cc \ \textsc{except}\ ![i][\mathit{Head}(cin[i])[4]] = \mathit{Head}(cin[i])[3]] \\
\quad \land\ \textsc{unchanged}\ \langle cout, ch\rangle
\end{array}$$

$$\begin{array}{lcl}
\mathit{Next}(i) & \triangleq & \lor\ \exists\, d \in \mathit{Data}, a \in \mathit{Addr} \,:\, \mathit{Read}(i, d, a) \lor \mathit{Write}(i, d, a) \\
& & \lor\ \mathit{MemWrite}(i) \lor \mathit{CacheUpdate}(i)
\end{array}$$

$$\mathit{vars} \ \triangleq\ \langle cc, cin, cout, ch\rangle$$

$$\begin{array}{lcl}
\mathit{Spec} & \triangleq & \land\ \mathit{Init} \\
& & \land\ \Box[\exists\, i \in \mathit{Proc} \,:\, \mathit{Next}(i)]_{\mathit{vars}} \\
& & \land\ \forall\, i \in \mathit{Proc} \,:\, \mathrm{WF}_{\mathit{vars}}(\mathit{CacheUpdate}(i)) \land \mathrm{WF}_{\mathit{vars}}(\mathit{MemWrite}(i))
\end{array}$$

Figure 18: The complete cache algorithm.

consistency of the complete cache.

4.3 The Complete Cache with Auxiliary Variables

We now describe how to prove theorem *CCimpliesDB1*. As we observed in Section 3.2.3, $DB1!Spec$ is equivalent to

$$\exists\, dch, q, mem \,:\, DB1!QSpec(dch, q) \wedge DB1!SDB(dch)!ISpec(mem)$$

One proves a theorem of the form $\forall\, x : F$ by simply proving F. Therefore, the proof of *CCimpliesDB1* is reduced to proving a theorem of the form

$$S1 \;\Rightarrow\; \exists\, dch, q, mem \,:\, S2 \tag{1}$$

In first-order logic, one proves an existentially quantified formula by finding instantiations for the quantified variables. For temporal existential quantification, the instantiations are called a *refinement mapping* [1]. We prove (1) by defining state functions $\overline{dch}$, $\overline{q}$, and $\overline{mem}$ in terms of the variables that appear in $S1$ and proving $S1 \Rightarrow \overline{S2}$, where $\overline{F}$ is the formula obtained by substituting $\overline{dch}$ for dch, $\overline{q}$ for q, and $\overline{mem}$ for mem in F, for any formula F.

Intuitively, $S1 \Rightarrow \overline{S2}$ means that if ch, cc, cin, and $cout$ change the way $S1$ allows them to, then $\overline{dch}$, $\overline{q}$, and $\overline{mem}$ change the way $S2$ allows dch, q, and mem to change. In other words, changes to the variables allowed by $S1$ cause changes to the barred variables that simulate the changes to the unbarred variables allowed by $S2$. We construct the refinement mapping by deciding which events allowed by $S1$ simulate the events allowed by $S2$.

To define $\overline{mem}$, we choose an event in the complete cache state machine to simulate the sequential consistency state machine's event of dequeuing a write operation and updating the database. We let the dequeue event happen when the last copy of a write is removed from the cin queues. Writes are performed in the same order to all the caches $cc[i]$. Hence, simulating the dequeue when the write is performed to the last cache means that $\overline{mem}$ will equal a cache $cc[i]$ to which the fewest writes have been performed. (There may be several such caches, in which case they will all be equal.) Fewer writes have been performed to cache $cc[i]$ than to cache $cc[j]$ iff the queue $cin[i]$ is longer than the queue $cin[j]$. Letting $pMax$ be some processor such that the length of $cin[pMax]$ is greater than or equal to the length of $cin[j]$ for any other processor j, we define $\overline{mem}$ to equal $cc[pMax]$.

Having defined $\overline{mem}$, we quickly see that it is impossible to define $\overline{dch}$ and $\overline{q}$. The complete cache state machine does not contain the information needed to construct the queues $\overline{q}[i]$ because it does not remember read operations. It does not contain the information needed to define $\overline{dch}$ because the write operation that changes $\overline{mem}$, which must appear in $\overline{dch}$, is immediately forgotten.

To finish constructing our refinement mapping, we must add *auxiliary variables* [1] to the complete cache. Adding an auxiliary variable a to a specification S means finding a formula S^a such that $\exists\, a : S^a$ is equivalent to S. In our example, we need two types of auxiliary variables: history variables and stuttering variables. A history variable records what happened in the past, but doesn't affect how other variables change. A stuttering variable forces stuttering steps—ones in which the other variables are left unchanged.

To define $\overline{q}$, we want to add a queue vq such that $\overline{q}[i]$ equals $Proj(i, vq) \circ cout[i]$, for each processor i. (Recall that $Proj(i, vq)$ is the subsequence of operations in vq issued by processor i.) A write is appended to the tail of vq when it is moved to the cin queues by a *MemWrite* action. It is removed from vq, and hence from the appropriate queue $\overline{q}[i]$, when $\overline{mem}$ is updated—which is when the last copy of that write is removed from the cin queues. It follows that the sequence of writes in vq will always equal $cin[pMax]$.

A read is inserted into vq when it is issued by a *Read* action; the trick is to insert it in the right place. A read by processor i returns the value currently in $cc[i]$, so it should be inserted into vq behind all writes already performed to $cc[i]$, and in front of all writes not yet performed to $cc[i]$. Operations by processor i must appear in vq in the order in which they were issued. A *Read* action is enabled only when all writes by i have already been performed to $cc[i]$; we ensure that the read is placed behind any preceding reads by processor i by placing it immediately in front of the first write that has not yet been performed to $cc[i]$—or at the end of vq if there is no such write. There will be exactly $Len(cin[i])$ writes in vq that have not been performed to $cc[i]$, so the read is put in front of the $Len(cin[i])$'th write from the end of vq—or at the end of vq, if $Len(cin[i])$ equals zero. Since the sequence of writes in vq equals $cin[pMax]$, the read is put immediately in front of the $(Len(cin[pMax]) - Len(cin[i]) + 1)st$ write in vq—or at the end of vq if $Len(cin[i])$ equals zero.

A write is removed from the head of vq, and hence from the head of some $\overline{q}[i]$, by a *CacheUpdate* step. However, there is no action of the complete cache that can be used to remove a read from vq. It must be removed by a stuttering step—one that does not change cc, cin, $cout$, or ch. We represent vq as $vrq \circ vcq$, where vrq is a possibly empty sequence of reads, and, if vcq is nonempty, then the head of vcq is a write. We first add vcq as a history variable, then add vrq as a stuttering variable.

Finally, we add a channel *vdch* as a history variable and let $\overline{dch}$ equal *vdch*. The complete cache algorithm with these auxiliary variables is described by formula *ASpec* of module *ACCache* in Figures 19 and 20. Corresponding to the four "elementary" actions $Read(i, d, a)$, ..., $CacheUpdate(i)$ of module *CCache*, module *ACCache* defines the actions $ARead(i, d, a)$, ..., $ACacheUpdate(i)$. Each of these actions equals the corresponding action of *CCache* conjoined with (i) an enabling condition asserting that *vrq* is empty and (ii) formulas describing the new values of *vcq*, *vrq*, and *vdch*. The next-state action of *ASpec* is the disjunction of the $A\ldots$ actions with a new action *VRead*, which removes a read operation from the head of *vrq* and sends it on the *vdch* channel. This is the action that adds the requisite stuttering steps to the complete cache.

Module *ACCache* uses the TLA$^+$ LET construct for making definitions local to an expression, and the construct $f[x \in S] \triangleq e(x)$ for recursively defining a function f with domain S.

We prove theorem *CCimpliesDB*1 of module *CacheCorrectness* by proving theorem *CCequivACC*, which asserts that *vcq*, *vrq*, and *vdch* are auxiliary variables, and theorem *ACCimpliesDB*1, which asserts the sequential consistency of the complete cache with auxiliary variables.

To prove *CCequivACC*, we use the propositions of Figure 21. These propositions provide practical rules for adding history and stuttering variables; they can be derived from simpler results [2]. Theorem *CCequivACC* is proved by adding to the complete cache the history variable *vcq*, then the stuttering variable *vrq*, and then the history variable *vdch*. Adding an auxiliary variable a to a specification S means writing a formula S^a such that $\exists\, a : S^a$ is equivalent to S, so we prove *CCequivACC* by finding two formulas *BSpec* and *CSpec* such that

$$\begin{array}{rcll} Spec & \equiv & \exists\, vcq : CSpec & \text{by Proposition 1} \\ & \equiv & \exists\, vcq : \exists\, vrq : BSpec & \text{by Proposition 2} \\ & \equiv & \exists\, vcq : \exists\, vrq : \exists\, vdch : ASpec & \text{by Proposition 1} \end{array}$$

We next describe the proof of *ACCimpliesDB*1.

4.4 The Proof of Theorem *ACCimpliesDB1*

It is argued elsewhere that the way to avoid mistakes in proofs is to structure them hierarchically [15]. This is especially true for correctness proofs of computer systems, where the social process for detecting errors is largely missing. A hierarchical proof is a sequence of statements and their proofs, each

MODULE *ACCache*

EXTENDS *CCache, MemParams, ChannelInterface, Sequences, Naturals*

VARIABLE $vcq, vrq, vdch$

$$
\begin{array}{l}
Op \triangleq Proc \times \{\text{"Rd"}, \text{"Wr"}\} \times Data \times Addr \\
AInit \triangleq \land\ Init \\
\qquad\qquad \land\ vcq = \langle\,\rangle \\
\qquad\qquad \land\ vrq = \langle\,\rangle \\
pMax \triangleq \text{CHOOSE } i \in Proc : \forall\, j \in Proc : Len(cin[j]) \leq Len(cin[i]) \\
ARead(i, d, a) \triangleq \\
\quad \text{LET } Insert[k \in Nat, s \in Seq(Op)] \triangleq \\
\qquad\quad \text{IF } (s = \langle\,\rangle) \lor ((k = 0) \land (Head(s)[2] = \text{"Wr"})) \\
\qquad\qquad \text{THEN } \langle\langle i, \text{"Rd"}, d, a\rangle\rangle \circ s \\
\qquad\qquad \text{ELSE IF } Head(s)[2] = \text{"Rd"} \\
\qquad\qquad\qquad \text{THEN } \langle Head(s)\rangle \circ Insert[k, Tail(s)] \\
\qquad\qquad\qquad \text{ELSE } \langle Head(s)\rangle \circ Insert[k-1, Tail(s)] \\
\quad \text{IN } \land\ vrq = \langle\,\rangle \\
\qquad\ \land\ Read(i, d, a) \\
\qquad\ \land\ \lor\ \land\ Len(cin[pMax]) = Len(cin[i]) \\
\qquad\qquad\quad \land\ vrq' = \langle\langle i, \text{"Rd"}, d, a\rangle\rangle \\
\qquad\qquad\quad \land\ \text{UNCHANGED } vcq \\
\qquad\qquad \lor\ \land\ Len(cin[pMax]) \neq Len(cin[i]) \\
\qquad\qquad\quad \land\ vcq' = Insert[Len(cin[pMax]) - Len(cin[i]),\ vcq] \\
\qquad\qquad\quad \land\ \text{UNCHANGED } vrq \\
\qquad\ \land\ \text{UNCHANGED } vdch \\
AWrite(i, d, a) \triangleq \land\ Write(i, d, a) \\
\qquad\qquad\qquad\quad \land\ vrq = \langle\,\rangle \\
\qquad\qquad\qquad\quad \land\ \text{UNCHANGED } \langle vcq, vrq, vdch\rangle \\
AMemWrite(i) \triangleq \land\ MemWrite(i) \\
\qquad\qquad\qquad\quad \land\ vrq = \langle\,\rangle \\
\qquad\qquad\qquad\quad \land\ vcq' = vcq \circ \langle Head(cout[i])\rangle \\
\qquad\qquad\qquad\quad \land\ \text{UNCHANGED } \langle vrq, vdch\rangle
\end{array}
$$

Figure 19: The complete cache with auxiliary variables (beginning).

$$
\begin{array}{l}
ACacheUpdate(i) \;\triangleq \\
\quad \text{LET } vWr \;\triangleq\; \forall j \in Proc \setminus \{i\} : Len(cin[j]) < Len(cin[i]) \\
\qquad\quad nRds[s \in Seq(Op)] \;\triangleq\; \text{IF } (s = \langle\,\rangle) \lor (Head(s)[2] = \text{“Wr”}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{THEN } 0 \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{ELSE } 1 + nRds[Tail(s)] \\
\quad \text{IN } \;\; \land\; vrq = \langle\,\rangle \\
\qquad\quad \land\; CacheUpdate(i) \\
\qquad\quad \land\; \lor \land\; vWr \\
\qquad\qquad\quad \land\; ChanOp(vdch, Head(cin[i])) \\
\qquad\qquad\quad \land\; vrq' = SubSeq(Tail(vcq), 1, nRds[Tail(vcq)]) \\
\qquad\qquad\quad \land\; vcq' = SubSeq(Tail(vcq), nRds[Tail(vcq)] + 1, Len(Tail(vcq))) \\
\qquad\qquad \lor \land\; \neg vWr \\
\qquad\qquad\quad \land\; \text{UNCHANGED } \langle vcq, vrq, vdch \rangle \\
VRead \;\triangleq\; \land\; vrq \neq \langle\,\rangle \\
\qquad\qquad\;\; \land\; ChanOp(vdch, Head(vrq)) \\
\qquad\qquad\;\; \land\; vrq' = Tail(vrq) \\
\qquad\qquad\;\; \land\; \text{UNCHANGED } \langle cc, cout, cin, ch, vcq \rangle \\
ANext \;\triangleq \\
\quad \lor\; \exists\, i \in Proc : \lor\; \exists\, d \in Data, a \in Addr : ARead(i, d, a) \lor AWrite(i, d, a) \\
\qquad\qquad\qquad\quad \lor\; AMemWrite(i) \lor ACacheUpdate(i) \\
\quad \lor\; VRead \\
avars \;\triangleq\; \langle vars, vcq, vrq, vdch \rangle \\
AFair \;\triangleq\; \land\; \forall\, i \in Proc : \mathrm{WF}_{vars}(CacheUpdate(i)) \land \mathrm{WF}_{vars}(MemWrite(i)) \\
\qquad\qquad \land\; \mathrm{WF}_{avars}(VRead) \\
ASpec \;\triangleq\; AInit \land \Box[ANext]_{avars} \land AFair
\end{array}
$$

Figure 20: The complete cache with auxiliary variables (continued).

of which is either a sequence of statements or else an ordinary paragraph-style proof. A deeper hierarchy implies a more rigorous proof. One can obtain any desired degree of rigor—from informal, intuitive reasoning to a completely rigorous, formal proof—by choosing the depth appropriately. We describe our proof of theorem $ACCimpliesDB1$ of module *CacheCorrectness* by showing how its hierarchical structure is obtained.

Proposition 1 (History Variable) *If h and h' do not occur in Init, $\mathcal{A}_i$, or f, and h' does not occur in g_i, for all $i \in I$, then*

$$\begin{aligned} Init \;\wedge\; \Box[\exists\, i \in I : \mathcal{A}_i]_v \;\equiv\; \boldsymbol{\exists}\, h : &\wedge Init \wedge (h = f) \\ &\wedge \Box[\exists\, i \in I : \mathcal{A}_i \wedge (h' = g_i)]_{\langle h,v \rangle} \end{aligned}$$

Proposition 2 (Stuttering Variable) *If*

1. *s and s' do not occur in Init, Inv, $\mathcal{A}_i$, f, or g_i, and s' does not occur in h, for all $i \in I$.*
2. *There is a partially ordered set D with (unique) minimum element ϕ and well-founded partial order $\prec$ such that:*
   (a) $Init \Rightarrow (f \in D)$
   (b) $Inv \wedge Inv' \wedge \mathcal{A}_i \Rightarrow (g_i \in D)$
   (c) $Inv \wedge (s \in D) \wedge (s \neq \phi) \Rightarrow (h \in D) \wedge (h \prec s)$

then

$$\begin{aligned} \Box Inv \;\Rightarrow\; (Init \;\wedge\; &\Box[\exists\, i \in I : \mathcal{A}_i]_v \\ \equiv\; \boldsymbol{\exists}\, s : \;&\wedge\; Init \wedge (s = f) \\ &\wedge\; \Box \begin{bmatrix} \vee\; (s = \phi) \wedge (\exists\, i \in I : (s' = g_i) \wedge \mathcal{A}_i) \\ \vee\; (s \neq \phi) \wedge (s' = h) \wedge (v' = v) \end{bmatrix}_{\langle s,v \rangle} \\ &\wedge\; \mathrm{WF}_{\langle s,v \rangle}((s \neq \phi) \wedge (s' = h) \wedge (v' = v))\;) \end{aligned}$$

Figure 21: Rules for adding history and stuttering variables.

4.4.1 The High Level Outline

To prove theorem $ACCimpliesDB1$, we must prove

$$ACC(cc, cin, cout, vcq, vrq, vdch)!ASpec \Rightarrow DB1!Spec$$

We now drop the prefixes "$ACC(cc, cin, cout, vcq, vrq, vdch)!$" and "$DB1!$", so the theorem to be proved is $ASpec \Rightarrow Spec$.[11] As we observed in Section 4.3, this theorem is equivalent to

$$ASpec \Rightarrow \boldsymbol{\exists}\, dch, q, mem : QSpec(dch, q) \wedge SDB(dch)!ISpec(mem)$$

[11]The only ambiguous name is *Spec*, which will mean *DB*1!*Spec* rather than the formula of the same name imported by *ACCache* from module *CCache*.

which we prove by proving

$$ASpec \Rightarrow \overline{QSpec(dch, q)} \wedge \overline{SDB(dch)!ISpec(mem)} \tag{2}$$

where $\overline{F}$ is obtained by substituting $\overline{dch}$, $\overline{q}$, $\overline{mem}$ for dch, q, mem in formula F, and

$$\begin{array}{lcl}
\overline{q} & \triangleq & [i \in Proc \mapsto Proj(i, vrq \circ vcq) \circ cout[i]\,] \\
\overline{mem} & \triangleq & cc[pMax] \\
\overline{dch} & \triangleq & vdch
\end{array}$$

When reasoning in a formal logic, the structure of the formula indicates the structure of the proof. Propositional logic tells us that to prove (2), we must prove $ASpec \Rightarrow \overline{QSpec}$ and $ASpec \Rightarrow \overline{SDB(dch)!ISpec(mem)}$. Both of these formulas have the form

$$AInit \wedge \Box[ANext]_{avars} \wedge AFair \;\Rightarrow\; \overline{Init} \wedge \Box[\overline{Next}]_{\overline{v}} \wedge \overline{L}$$

where L is a fairness condition. (For $\overline{SDB(dch)!ISpec(mem)}$, formula $\overline{L}$ is just TRUE.) The standard TLA proof of such a formula has the following structure, where Inv is a suitable predicate called the *invariant*.

1. $AInit \wedge \Box[ANext]_{avars} \Rightarrow \Box Inv$
   1.1. $AInit \Rightarrow Inv$
   1.2. $Inv \wedge [ANext]_{avars} \Rightarrow Inv'$
   1.3. Q.E.D.
      PROOF: 1.1, 1.2, and rule INV1 of [14].
2. $AInit \Rightarrow \overline{Init}$
3. $\Box Inv \wedge \Box[ANext]_{avars} \Rightarrow \Box[\overline{Next}]_{\overline{v}}$
   3.1. $[Inv \wedge Inv' \wedge ANext]_{avars} \Rightarrow [\overline{Next}]_{\overline{v}}$
   3.2. Q.E.D.
      PROOF: Rules TLA2 and INV2 of [14].
4. $\Box Inv \wedge \Box[ANext]_{avars} \wedge AFair \Rightarrow \overline{L}$
5. Q.E.D.
   PROOF: 1–4 and propositional logic.

We combine the proofs for $ASpec \Rightarrow \overline{QSpec}$ and $ASpec \Rightarrow \overline{SDB(dch)!Spec}$, using the fact that barring distributes over all the TLA$^+$ operators of Figure 1 except WF, SF, and ENABLED, to get the following structure for the proof of (2). (Remember that $\overline{dch}$ equals $vdch$.)

1. $AInit \wedge \Box[ANext]_{avars} \Rightarrow \Box Inv$
   1.1. $AInit \Rightarrow Inv$
   1.2. $Inv \wedge [ANext]_{avars} \Rightarrow Inv'$

1.3. Q.E.D.

PROOF: 1.1, 1.2, and rule INV1.

2. $AInit \Rightarrow (\forall\, i \in Proc\ :\ \overline{q}[i] = \langle\,\rangle) \wedge (\overline{mem} \in InitDB)$

3. $\Box Inv \wedge \Box[ANext]_{avars} \Rightarrow$

   $\wedge\ \Box[\exists\, i \in Proc\ :\ Enq(i, vdch, \overline{q}) \vee Deq(i, vdch, \overline{q})]_{\langle ch, vdch, \overline{q}\rangle}$

   $\wedge\ \Box[SDB(vdch)!INext(\overline{mem})]_{\langle \overline{mem},\, vdch\rangle}$

   3.1. $[Inv \wedge Inv' \wedge ANext]_{avars} \Rightarrow$

   $\wedge\ [\exists\, i \in Proc\ :\ Enq(i, vdch, \overline{q}) \vee Deq(i, vdch, \overline{q})]_{\langle ch, vdch, \overline{q}\rangle}$

   $\wedge\ [SDB(vdch)!INext(\overline{mem})]_{\langle \overline{mem},\, vdch\rangle}$

   3.2. Q.E.D.

   PROOF: Rules TLA2 and INV2 of [14].

4. $\Box Inv \wedge \Box[ANext]_{avars} \wedge AFair \Rightarrow$

   $\forall\, i \in Proc\ :\ \overline{\mathrm{WF}_{\langle ch, dch, q\rangle}(Deq(i, dch, q))}$

5. Q.E.D.

   PROOF: 1–4 and propositional logic.

Step 2 asserts that the initial state of the complete cache machine implements a correct initial state of the sequential consistency machine. Its proof is easy. We now consider the other steps that need to be proved: the two substeps of step 1, step 3.1, and step 4.

4.4.2 Step 1: The Invariance Proof

As in all assertional reasoning, finding a suitable invariant *Inv* is the key to the proof. One first guesses a definition of *Inv* and then iteratively refines it—usually making it stronger—until steps 1.2 and 3.1 are both provable. With experience, one gets fairly good at guessing the invariant, and few iterations are needed.

Our predicate *Inv* is the conjunction of five assertions with the following intuitive meanings.

1. Variables have the type of values we expect them to. (Such a "type correctness" assertion is a standard part of an invariant.)

2. For every processor i, the queue $cin[i]$ is a suffix of $cin[pMax]$, and $cc[i]$ equals the memory obtained by applying the first $Len(cin[pMax]) - Len(cin[i])$ operations in $cin[pMax]$ to $cc[pMax]$.

3. The sequence of writes in vcq equals $cin[pMax]$.

4. For any read operation $\langle i, \text{“}\mathsf{Rd}\text{”}, d, a\rangle$ in $vrq \circ vcq$, if σ is the sequence of write operations that precede it in $vrq \circ vcq$, and m is the memory

obtained by updating $cc[pMax]$ with the sequence σ of writes, then $d = m[a]$.

5. There are no reads by processor i in vcq after the $(Len(cin[pMax]) - Len(cin[i]) + 1)st$ write.

Conjunct 4 is the key to why the algorithm works. It states that $vrq \circ vcq$ is a correct sequence of operations for a serial memory whose contents are $\overline{mem}$ (which equals $cc[pMax]$). This sequence is the serialization of all operations that have not yet been performed to $\overline{mem}$ and are not still in the *cout* queues. Thus, the serialization order of writes is determined by the order in which *MemWrite* events move writes from the *cout* queues to the *cin* queues.

The precise definition of *Inv* is in Figure 22. It uses the following operators: $WriteSel(s)$ is the subsequence of write operations in s, $ApplyOps[s, m]$ is the memory contents obtained by applying the sequence s of writes to a memory with contents m, and $InsertPos[k, s]$ is the index of the $(k + 1)st$ write in s, or equals $Len(s) + 1$ if there are at most k writes in s.

Having defined *Inv*, we can now continue our proof. Step 1.1 ($AInit \Rightarrow Inv$) is straightforward. The structure of *Inv* leads us to prove 1.2 as follows:

1.2. $Inv \wedge [ANext]_{avars} \Rightarrow Inv'$
  1.2.1. $Inv \wedge [ANext]_{avars} \Rightarrow Inv.1'$
  ⋮
  1.2.5. $Inv \wedge [ANext]_{avars} \Rightarrow Inv.5'$
  1.2.6. Q.E.D.
    PROOF: 1.2.1–1.2.5, since *Inv* equals $Inv.1 \wedge \ldots \wedge Inv.5$.

Next, the structure of $[ANext]_{avars}$ tells us how to structure the proofs of 1.2.1–1.2.5. For example, the proof of 1.2.4 is:

1.2.4. $Inv \wedge [ANext]_{avars} \Rightarrow Inv.4'$
  1.2.4.1. $Inv \wedge (i \in Proc) \wedge (d \in Data) \wedge (a \in Addr) \wedge ARead(i, d, a) \Rightarrow Inv.4'$
  1.2.4.2. $Inv \wedge (i \in Proc) \wedge (d \in Data) \wedge (a \in Addr) \wedge AWrite(i, d, a) \Rightarrow Inv.4'$
  1.2.4.3. $Inv \wedge (i \in Proc) \wedge AMemWrite(i) \Rightarrow Inv.4'$
  1.2.4.4. $Inv \wedge (i \in Proc) \wedge ACacheUpdate(i) \Rightarrow Inv.4'$
  1.2.4.5. $Inv \wedge VRead \Rightarrow Inv.4'$
  1.2.4.6. $Inv \wedge (avars' = avars) \Rightarrow Inv.4'$
  1.2.4.7. Q.E.D.

$$
\begin{array}{l}
\mathit{WriteSel}(s) \;\triangleq\; \text{LET } \mathit{test}(e) \;\triangleq\; e[2] = \text{“Wr”} \\
\qquad\qquad\qquad\;\; \text{IN } \;\; \mathit{SelectSeq}(s, \mathit{test}) \\
\mathit{ApplyOps}[s \in \mathit{Seq}(\mathit{Op}), m \in [\mathit{Addr} \rightarrow \mathit{Data}]] \;\triangleq \\
\quad \text{IF } \; s = \langle\,\rangle \\
\quad\quad \text{THEN } \; m \\
\quad\quad \text{ELSE } \; \mathit{ApplyOps}[\,\mathit{Tail}(s),\; [m \text{ EXCEPT } ![\mathit{Head}(s)[4]] = \mathit{Head}(s)[3]]\,] \\
\mathit{InsertPos}[k \in \mathit{Nat}, s \in \mathit{Seq}(\mathit{Op})] \;\triangleq \\
\quad \text{IF } \; (s = \langle\,\rangle) \lor ((k = 0) \land (\mathit{Head}(s)[2] = \text{“Wr”})) \\
\quad\quad \text{THEN } \; 1 \\
\quad\quad \text{ELSE } \; \text{IF } \; \mathit{Head}(s)[2] = \text{“Rd”} \\
\quad\quad\quad\quad\quad \text{THEN } \; 1 + \mathit{InsertPos}[k, \mathit{Tail}(s)] \\
\quad\quad\quad\quad\quad \text{ELSE } \; 1 + \mathit{InsertPos}[k-1, \mathit{Tail}(s)] \\
\mathit{Inv} \;\triangleq \\
\quad 1.\land \; \land \; \mathit{cc} \in [\mathit{Proc} \rightarrow [\mathit{Addr} \rightarrow \mathit{Data}]] \\
\quad\quad\;\; \land \; \mathit{cin} \in [\mathit{Proc} \rightarrow \mathit{Seq}(\mathit{Proc} \times \{\text{“Wr”}\} \times \mathit{Data} \times \mathit{Addr})] \\
\quad\quad\;\; \land \; \land \; \mathit{cout} \in [\mathit{Proc} \rightarrow \mathit{Seq}(\mathit{Proc} \times \{\text{“Wr”}\} \times \mathit{Data} \times \mathit{Addr})] \\
\quad\quad\quad\;\; \land \; \forall\, i \in \mathit{Proc} \,:\, \forall\, j \in 1 \,..\, \mathit{Len}(\mathit{cout}[i]) \,:\, \mathit{cout}[i][j][1] = i \\
\quad\quad\;\; \land \; \mathit{vcq} \in \{s \in \mathit{Seq}(\mathit{Op}) \,:\, (s \neq \langle\,\rangle) \Rightarrow (\mathit{Head}(s)[2] = \text{“Wr”})\,\} \\
\quad\quad\;\; \land \; \mathit{vrq} \in \mathit{Seq}(\mathit{Proc} \times \{\text{“Rd”}\} \times \mathit{Data} \times \mathit{Addr}) \\
\quad 2.\land \; \forall\, i \in \mathit{Proc} \,: \\
\quad\quad\quad \land \; \mathit{cin}[i] = \mathit{SubSeq}(\mathit{cin}[\mathit{pMax}],\, 1 + \mathit{Len}(\mathit{cin}[\mathit{pMax}]) - \mathit{Len}(\mathit{cin}[i]), \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathit{Len}(\mathit{cin}[\mathit{pMax}])) \\
\quad\quad\quad \land \; \mathit{cc}[i] = \mathit{ApplyOps}[\mathit{SubSeq}(\mathit{cin}[\mathit{pMax}], 1, \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathit{Len}(\mathit{cin}[\mathit{pMax}]) - \mathit{Len}(\mathit{cin}[i])),\; \mathit{cc}[\mathit{pMax}]\,] \\
\quad 3.\land \; \mathit{WriteSel}(\mathit{vcq}) = \mathit{cin}[\mathit{pMax}] \\
\quad 4.\land \; \text{LET } \; s \;\triangleq\; \mathit{vrq} \circ \mathit{vcq} \\
\quad\quad\;\; \text{IN } \;\; \forall\, n \in 1 \,..\, \mathit{Len}(s) \,: \\
\quad\quad\quad\quad\quad (s[n][2] = \text{“Rd”}\,) \Rightarrow \\
\quad\quad\quad\quad\quad\quad (s[n][3] = \mathit{ApplyOps}[\,\mathit{WriteSel}(\mathit{SubSeq}(s, 1, n-1)), \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathit{cc}[\mathit{pMax}]\,]\,[s[n][4]]\,) \\
\quad 5.\land \; \forall\, i \in \mathit{Proc} \,: \\
\quad\quad\quad \forall\, j \in \mathit{InsertPos}[\mathit{Len}(\mathit{cin}[\mathit{pMax}]) - \mathit{Len}(\mathit{cin}[i]),\; \mathit{vcq}\,] \,..\, \mathit{Len}(\mathit{vcq}) \,: \\
\quad\quad\quad\quad (\mathit{vcq}[j][1] = i) \Rightarrow (\mathit{vcq}[j][2] = \text{“Wr”}\,)
\end{array}
$$

Figure 22: The invariant for the correctness proof of the complete cache.

PROOF: 1.2.4.1–1.2.4.6, the definition of *ANext*, and simple predicate logic, since $[A]_v \triangleq A \vee (v' = v)$, for any A and v.

The proof of all steps 1.2.∗.6 are trivial, since the tuple *avars* contains all the variables that appear in *Inv*. The other 25 steps are the standard ones for an invariance proof. They show that, starting in a state with the invariant true, executing one step of the algorithm produces a state in which each conjunct of the invariant is true. The quadratic number of steps—the number of disjuncts in the next-state relation times the number of conjuncts of the invariant—is characteristic of approaches based on invariance, such as the Owicki-Gries method [22].

Some of the 25 remaining steps in the proof of 1.2 are trivial—for example, 1.2.3.2 holds because $AWrite(i, d, a)$ leaves *vcq* and *cin* unchanged, which implies that *pMax* is unchanged, and thus *Inv*.3 remains true. Some steps are routine—for example, the substeps of 1.2.1 require the kind of simple reasoning performed by a type checker. Some steps go to the heart of why the algorithm works. We consider one such step: 1.2.4.1. This step asserts that an *ARead* event leaves the crucial conjunct *Inv*.4 true. Its proof essentially shows that read operations are serializable. Here is an informal proof, based on the informal statement of *Inv* given above:

1.2.4.1. $Inv \wedge (i \in Proc) \wedge (d \in Data) \wedge (a \in Addr) \wedge ARead(i, d, a)$
$\Rightarrow\ Inv.4'$

PROOF: $ARead(i, d, a)$ implies *vrq* is empty, so an $ARead(i, d, a)$ step inserts a read of d from address a just before the $(Len(cin[pMax]) - Len(cin[i]) + 1)st$ write in *vcq*, or at the end if $Len(cin[i])$ equals zero, and hence after the first $Len(cin[pMax]) - Len(cin[i])$ writes. Since *Inv*.4 holds before the step, it holds after the step if $d = m[a]$, where m is the memory obtained by updating $cc[pMax]$ with the first $Len(cin[pMax]) - Len(cin[i])$ writes in *vcq*. By *Inv*.2, m equals $cc[i]$. Since $ARead(i, d, a)$ implies $Read(i, d, a)$, which implies $d = cc[i][a]$, we deduce $d = m[a]$.

This informal reasoning is useful for understanding the algorithm, but it is not reliable. An off-by-one error in where the read is inserted could easily pass unnoticed. The proof implicitly assumes that *cc* and *cin* are left unchanged by an *ARead* step, and incorrect assumptions are hard to detect if they are not made explicit. Step 1.2.4.1 is a precisely defined mathematical formula; it can be proved by rigorous mathematical reasoning. The hierarchical proof structure allows such proofs to be carried down to as fine a level of detail as necessary to reach any desired degree of reliability.

4.4.3 Step 3.1: Step Simulation

The form of step 3.1 immediately leads to this proof outline:

3.1. $[Inv \land Inv' \land ANext]_{avars} \Rightarrow$
$\quad \land\ [\exists\, i \in Proc : Enq(i, vdch, \overline{q}) \lor Deq(i, vdch, \overline{q})]_{\langle ch, vdch, \overline{q} \rangle}$
$\quad \land\ [SDB(vdch)!INext(\overline{mem})]_{\langle \overline{mem},\, vdch \rangle}$

3.1.1. $[Inv \land Inv' \land ANext]_{avars} \Rightarrow$
$\quad [\exists\, i \in Proc : Enq(i, vdch, \overline{q}) \lor Deq(i, vdch, \overline{q})]_{\langle ch, vdch, \overline{q} \rangle}$

3.1.2. $[Inv \land Inv' \land ANext]_{avars} \Rightarrow [SDB(vdch)!INext(\overline{mem})]_{\langle \overline{mem},\, vdch \rangle}$

3.1.3. Q.E.D.

Proof: 3.1.1 and 3.1.2.

The structure of the left-hand side of these implications leads to the same sort of decomposition as for steps 1.2.1–1.2.5. This would lead to steps such as

3.1.1.1. $Inv \land Inv' \land (i \in Proc) \land (d \in Data) \land (a \in Addr) \land ARead(i, d, a)$
$\quad \Rightarrow\ [\exists\, i \in Proc : Enq(i, vdch, \overline{q}) \lor Deq(i, vdch, \overline{q})]_{\langle ch, vdch, \overline{q} \rangle}$

However, we actually prove the stronger result that the left-hand side implies $Enq(i, vdch, \overline{q})$. The decomposition with the stronger results is:

3.1.1. $[Inv \land Inv' \land ANext]_{avars} \Rightarrow$
$\quad [\exists\, i \in Proc : Enq(i, vdch, \overline{q}) \lor Deq(i, vdch, \overline{q})]_{\langle ch, vdch, \overline{q} \rangle}$

3.1.1.1. $Inv \land Inv' \land (i \in Proc) \land (d \in Data) \land (a \in Addr) \land ARead(i, d, a)$
$\quad \Rightarrow\ Enq(i, vdch, \overline{q})$

3.1.1.2. $Inv \land Inv' \land (i \in Proc) \land (d \in Data) \land (a \in Addr) \land AWrite(i, d, a)$
$\quad \Rightarrow\ Enq(i, vdch, \overline{q})$

3.1.1.3. $Inv \land Inv' \land (i \in Proc) \land AMemWrite(i)$
$\quad \Rightarrow\ \textsc{unchanged}\ \langle ch, vdch, \overline{q} \rangle$

3.1.1.4. $Inv \land Inv' \land (i \in Proc) \land ACacheUpdate(i)$
$\quad \Rightarrow\ Deq(i, vdch, \overline{q}) \lor \textsc{unchanged}\ \langle ch, vdch, \overline{q} \rangle$

3.1.1.5. $Inv \land Inv' \land VRead\ \Rightarrow\ Deq(i, vdch, \overline{q})$

3.1.1.6. $(avars' = avars)\ \Rightarrow\ \textsc{unchanged}\ \langle ch, vdch, \overline{q} \rangle$

3.1.1.7. Q.E.D.

Proof: 3.1.1.1–3.1.1.6, the definitions of $ANext$ and $[\ldots]_{avars}$, and simple predicate logic.

3.1.2. $[Inv \land Inv' \land ANext]_{avars} \Rightarrow [SDB(vdch)!INext(\overline{mem})]_{\langle \overline{mem},\, vdch \rangle}$

3.1.2.1. $Inv \land Inv' \land (i \in Proc) \land (d \in Data) \land (a \in Addr) \land ARead(i, d, a)$
$\quad \Rightarrow\ \textsc{unchanged}\ \langle \overline{mem},\, vdch \rangle$

3.1.2.2. $Inv \land Inv' \land (i \in Proc) \land (d \in Data) \land (a \in Addr) \land AWrite(i, d, a)$
$\quad \Rightarrow\ \textsc{unchanged}\ \langle \overline{mem},\, vdch \rangle$

3.1.2.3. $Inv \wedge Inv' \wedge (i \in Proc) \wedge AMemWrite(i)$
$\Rightarrow$ UNCHANGED $\langle \overline{mem}, vdch \rangle$
3.1.2.4. $Inv \wedge Inv' \wedge (i \in Proc) \wedge ACacheUpdate(i)$
$\Rightarrow SDB(vdch)!INext(\overline{mem}) \vee$ UNCHANGED $\langle \overline{mem}, vdch \rangle$
3.1.2.5. $Inv \wedge Inv' \wedge VRead \Rightarrow SDB(vdch)!INext(\overline{mem})$
3.1.2.6. $(avars' = avars) \Rightarrow$ UNCHANGED $\langle \overline{mem}, vdch \rangle$
3.1.2.7. Q.E.D.
PROOF: 3.1.2.1–3.1.2.6, the definitions of $ANext$ and $[\ldots]_{avars}$, and simple predicate logic.

The proofs of steps 3.1.1.6 and 3.1.2.6 are trivial, since the tuple *avars* contains all the variables that appear in our formulas. To understand the other ten steps, remember that the state machine specifying sequential consistency is the composition of two machines: a queue machine with variables *ch*, *dch*, and *q* that moves operations into and out of the queues, and a serial database machine with variables *dch* and *mem*. Steps 3.1.1.1–3.1.1.5 assert that every event of the complete cache either simulates a step of the queue machine or else leaves that machine's variables unchanged. Steps 3.1.2.1–3.1.2.5 make the analogous assertions for the serial database machine. These steps correspond to the step simulation part of a traditional proof that one state machine simulates another [9, 19]. As usual, some of the proofs are trivial and some give further insight into the algorithm. An example of the latter is step 3.1.1.1, which asserts that an *ARead* event simulates a queue machine *Enq* event. The structure of $Enq(i, vdch, \overline{q})$ suggests the following proof outline.

3.1.1.1. $Inv \wedge Inv' \wedge (i \in Proc) \wedge (d \in Data) \wedge (a \in Addr) \wedge ARead(i, d, a)$
$\Rightarrow Enq(i, vdch, \overline{q})$
ASSUME: $Inv \wedge Inv' \wedge (i \in Proc) \wedge (d \in Data) \wedge (a \in Addr) \wedge ARead(i, d, a)$
PROVE: $Enq(i, vdch, \overline{q})$
3.1.1.1.1. $ChanOp(ch[i], \langle i, \text{“Rd”}, d, a \rangle)$
3.1.1.1.2. $\overline{q}[i]' = \overline{q}[i] \circ \langle \langle i, \text{“Rd”}, d, a \rangle \rangle$
3.1.1.1.3. UNCHANGED $vdch$
3.1.1.1.4. $\forall j \in Proc \setminus \{i\}$: UNCHANGED $\langle \overline{q}[j], ch[j] \rangle$
3.1.1.1.5. Q.E.D.
PROOF: 3.1.1.1.1–3.1.1.1.4, the definitions of POp and Enq, and predicate logic.

Here is an informal proof of 3.1.1.1.2.

3.1.1.1.2. $\overline{q}[i]' = \overline{q}[i] \circ \langle \langle i, \text{“Rd”}, d, a \rangle \rangle$
PROOF: An $ARead(i, d, a)$ event inserts $\langle i, \text{“Rd”}, d, a \rangle$ into either *vrq* or

vcq, and hence into $\overline{q}[i]$. We just have to show that it inserts the operation at the end of $\overline{q}[i]$. The operation is inserted into $vrq \circ vcq$ just before the $(Len(cin[pMax]) - Len(cin[i]) + 1)st$ write in vcq, or at the end if $Len(cin[i])$ equals zero. By $Inv.2$ and $Inv.3$, all the writes that follow it in $vrq \circ vcq$ are in $cin[i]$. Since $ARead(i, d, a)$ implies $Read(i, d, a)$, which implies there are no writes by i in $cin[i]$, no writes by i follow the newly inserted operation in $vrq \circ vcq$. By $Inv.5$, no reads by i follow it. By definition of $\overline{q}[i]$, this implies that the operation is inserted at the end of $\overline{q}[i]$.

4.4.4 Step 4: Fairness

To prove step 4, we must prove $\overline{\mathrm{WF}_{\langle ch, dch, q\rangle}(Deq(i, dch, q))}$ for each processor i. This suggests using rule WF2 of [14]. With that rule, one infers $\overline{\mathrm{WF}_w(B)}$ from $\mathrm{WF}_v(A)$, where A is the action that implements $\overline{B}$. However, as we saw in the proof of 3.1.1, $\overline{Deq(i, dch, q)}$ is implemented by the two separate actions $ACacheUpdate(i)$ and $VRead$. This gives us two ways to prove step 4: (i) prove that the complete cache specification implies weak fairness of $ACacheUpdate(i) \lor VRead$ and apply WF2, or (ii) expand the definition of WF in the conclusion and apply temporal logic reasoning directly. Either one works; we use the second.

The formula $\mathrm{WF}_v(A)$ is defined to equal $\Box\Diamond\neg\textsc{Enabled}\,\langle A\rangle_v \lor \Box\Diamond\langle A\rangle_v$, for any action A, where $\langle A\rangle_v$ is defined to equal $A \land (v' \neq v)$ and the predicate $\textsc{Enabled}\,\langle A\rangle_v$ asserts that there is some possible step starting in the current state that satisfies $\langle A\rangle_v$.

Proofs of fairness are typically by contradiction, deducing $F \Rightarrow G$ from $F \land \neg G \Rightarrow G$. Expanding the definition of WF leads to the following proof.

4. $\Box Inv \land \Box[ANext]_{avars} \land AFair \Rightarrow$
   $\quad \forall\, i \in Proc : \overline{\mathrm{WF}_{\langle ch, dch, q\rangle}(Deq(i, dch, q))}$
   PROOF: By predicate logic, it suffices to:
   ASSUME: $i \in Proc$
   PROVE: $\Box Inv \land \Box[ANext]_{avars} \land AFair \Rightarrow \overline{\mathrm{WF}_{\langle ch, dch, q\rangle}(Deq(i, dch, q))}$
   LET: $B \triangleq Deq(i, dch, q)$
   $\quad w \triangleq \langle ch, dch, q\rangle$
   4.1. $\Box Inv \land \Box[ANext]_{avars} \land AFair \land \Box\overline{\textsc{Enabled}\,\langle B\rangle_w} \land \Box[\neg\overline{B}]_{\overline{w}}$
   $\qquad \Rightarrow \Box\Diamond\langle\overline{B}\rangle_{\overline{w}}$
   4.2. $\Box Inv \land \Box[ANext]_{avars} \land AFair \land \Diamond\Box\overline{\textsc{Enabled}\,\langle B\rangle_w} \land \Diamond\Box[\neg\overline{B}]_{\overline{w}}$
   $\qquad \Rightarrow \Box\Diamond\langle\overline{B}\rangle_{\overline{w}}$
   4.2.1. $AFair \equiv \Box AFair$

PROOF: Definition of $AFair$, since $\Box\mathrm{WF} \equiv \mathrm{WF}$ and $\Box$ distributes over $\wedge$ and $\forall$.

4.2.2. $\Box Inv \wedge \Box[ANext]_{avars} \wedge \Box AFair \wedge \Box\overline{\textsc{Enabled}\,\langle B\rangle_w} \wedge \Box[\neg\overline{B}]_{\overline{w}}$
$\Rightarrow \Box\Diamond\langle\overline{B}\rangle_{\overline{w}}$

PROOF: 4.1 and 4.2.1.

4.2.3. $\Diamond\Box Inv \wedge \Diamond\Box[ANext]_{avars} \wedge \Diamond\Box AFair \wedge \Diamond\Box\overline{\textsc{Enabled}\,\langle B\rangle_w}$
$\wedge\ \Diamond\Box[\neg\overline{B}]_{\overline{w}} \ \Rightarrow\ \Diamond\Box\Diamond\langle\overline{B}\rangle_{\overline{w}}$

PROOF: 4.2.2, since $F_1 \Rightarrow F_2$ implies $\Diamond F_1 \Rightarrow \Diamond F_2$ and $\Diamond(\Box F_1 \wedge \ldots \wedge \Box F_n) \equiv (\Diamond\Box F_1 \wedge \ldots \wedge \Diamond\Box F_n)$, for any formulas F_i.

4.2.4. Q.E.D.

PROOF: 4.2.3, since $\Box F \Rightarrow \Diamond\Box F$ and $\Diamond\Box\Diamond F \equiv \Box\Diamond F$, for any F.

4.3. Q.E.D.

PROOF: 4.2 and propositional logic, since $\overline{\mathrm{WF}_{\langle ch, dch, q\rangle}(Deq(i, dch, q))}$ is defined to equal $\overline{\Box\Diamond\neg\textsc{Enabled}\,\langle B\rangle_w \vee \Box\Diamond\langle B\rangle_w}$, which is equivalent to $\Box\Diamond\neg\overline{\textsc{Enabled}\,\langle B\rangle_w} \vee \Box\Diamond\langle\overline{B}\rangle_{\overline{w}}$ because barring distributes over all operators except WF and $\textsc{Enabled}$, and

$\neg(\Box\Diamond\neg\overline{\textsc{Enabled}\,\langle B\rangle_w} \vee \Box\Diamond\langle\overline{B}\rangle_{\overline{w}})$
$\equiv\ \Diamond\Box\overline{\textsc{Enabled}\,\langle B\rangle_w} \wedge \Diamond\Box\neg\langle\overline{B}\rangle_{\overline{w}}$ $\quad[\Diamond \triangleq \neg\Box\neg]$
$\equiv\ \Diamond\Box\overline{\textsc{Enabled}\,\langle B\rangle_w} \wedge \Diamond\Box[\neg\overline{B}]_{\overline{w}}$ $\quad[\neg\langle A\rangle_v \equiv [\neg A]_v$, for any A and $v]$

so 4.2 shows that $\Box Inv \wedge \Box[ANext]_{avars} \wedge AFair \wedge \neg\overline{\mathrm{WF}_{\ldots}(\ldots)}$ implies $\overline{\mathrm{WF}_{\ldots}(\ldots)}$.

The proof of 4.1 has the following structure.

4.1. $\Box Inv \wedge \Box[ANext]_{avars} \wedge AFair \wedge \Box\overline{\textsc{Enabled}\,\langle B\rangle_w} \wedge \Box[\neg\overline{B}]_{\overline{w}}$
$\Rightarrow \Box\Diamond\langle\overline{B}\rangle_{\overline{w}}$

LET: $AA \ \triangleq\ \Box Inv \wedge \Box[ANext]_{avars} \wedge AFair \wedge \Box\overline{\textsc{Enabled}\,\langle B\rangle_w}$
$\wedge\ \Box[\neg\overline{B}]_{\overline{w}}$

$Qrd \ \triangleq\ (vrq \neq \langle\,\rangle) \wedge (Head(vrq)[1] = i)$

$Qwr \ \triangleq\ (vrq = \langle\,\rangle) \wedge (Head(vcq)[1] = i)$

4.1.1. $AA \Rightarrow \Diamond(Qrd \vee Qwr)$

4.1.2. $AA \wedge Qrd \Rightarrow \Box\Diamond\langle\overline{B}\rangle_{\overline{w}}$

4.1.3. $AA \wedge Qwr \Rightarrow \Box\Diamond\langle\overline{B}\rangle_{\overline{w}}$

4.1.4. Q.E.D.

PROOF: 4.1.1–4.1.3 and simple temporal reasoning, since $AA \equiv \Box AA$.

Steps 4.1.1 and 4.1.2 are proved informally as follows.

4.1.1. $AA \Rightarrow \Diamond(Qrd \vee Qwr)$

4.1.1.1. $\overline{\textsc{Enabled}\,\langle B\rangle_w} \equiv (cout[i] \neq \langle\,\rangle) \vee (Proj(i, vrq \circ vcq) \neq \langle\,\rangle)$

PROOF: $\textsc{Enabled}\,\langle B\rangle_w$ equals $q[i] \neq \langle\,\rangle$, and $\overline{q}[i]$ is defined to equal

$Proj(i, vrq \circ vcq) \circ cout[i]$.

4.1.1.2. $AA \land (cout[i] \neq \langle\rangle) \Rightarrow \Diamond(Proj(i, vrq \circ vcq) \neq \langle\rangle)$

PROOF: Weak fairness of $MemWrite(i)$ implies that if $cout[i]$ is nonempty, then its head (which by $Inv.1$ is an operation of processor i) must eventually be removed and appended to vcq.

4.1.1.3. $AA \land (Proj(i, vrq \circ vcq) \neq \langle\rangle) \Rightarrow \Diamond(Qrd \lor Qwr)$

PROOF: Weak fairness of $VRead$ implies that any operation in vrq (which by $Inv.1$ must be a read) eventually reaches the head of vrq and is then removed from vrq. By $Inv.3$, a write is in vcq iff it is in $cin[pMax]$. Weak fairness of the $CacheUpdate(j)$ actions and of $VRead$ therefore implies that any write in vcq eventually reaches the head of vcq and is then removed from vcq. Therefore, a read by processor i in $vrq \circ vcq$ eventually reaches vrq and thus eventually reaches the head of vrq, making Qrd true. A write by i in $vrq \circ vcq$ is in vcq (by $Inv.1$) and eventually reaches the head of vcq, making Qwr true.

4.1.1.4. Q.E.D.

PROOF: 4.1.1.1–4.1.1.3 and temporal logic reasoning, since AA implies $\overline{\text{ENABLED}\ \langle B\rangle_w}$.

4.1.2. $AA \land Qrd \Rightarrow \Box\Diamond\langle\overline{B}\rangle_{\overline{w}}$

4.1.2.1. $AA \land Qrd \Rightarrow \Box Qrd$

PROOF: Qrd can be made false only by removing a read by processor i from the head of vrq, which is a nonstuttering $\overline{B}$ step, and the conjunct $\Box[\neg\overline{B}]_{\overline{w}}$ of AA asserts that such a step never occurs.

4.1.2.2. $AA \land \Box Qrd \Rightarrow \Box\Diamond\langle\overline{B}\rangle_{\overline{w}}$

PROOF: Qrd implies that $VRead$ is enabled and that a $VRead$ step is a $\langle\overline{B}\rangle_{\overline{w}}$ step. Thus, $\Box Qrd$ and weak fairness of $VRead$ imply that infinitely many such steps occur, proving $\Box\Diamond\langle\overline{B}\rangle_{\overline{w}}$.

4.1.2.3. Q.E.D.

PROOF: 4.1.2.1 and 4.1.2.2.

The proof of 4.1.3 has the following outline. Informal proofs of the substeps are similar to the ones above and are left to the reader.

4.1.3. $AA \land Qwr \Rightarrow \Box\Diamond\langle\overline{B}\rangle_{\overline{w}}$

LET: $Qun \triangleq \forall j \in Proc \setminus \{pMax\} : Len(cin[j]) < Len(cin[pMax])$

4.1.3.1. $AA \land Qwr \Rightarrow \Diamond(Qun \land Qwr)$

4.1.3.2. $AA \land (Qun \land Qwr) \Rightarrow \Box(Qun \land Qwr)$

4.1.3.3. $AA \land \Box(Qun \land Qwr) \Rightarrow \Box\Diamond\langle\overline{B}\rangle_{\overline{w}}$

4.1.3.4. Q.E.D.

PROOF: 4.1.3.1–4.1.3.3 and temporal reasoning.

Each of our informal proofs can be replaced by hierarchical ones, which can be carried down to the point where each step is justified by predicate logic and TLA proof rules.

4.4.5 Discussion of the Proof

The key to managing any kind of complexity, including the complexity inherent in a nontrivial proof, is hierarchical structure. When reasoning in a formal logic such as TLA, the proof rules and the structure of the formulas determine the structure of the proof. For example, the first three levels in the proofs of steps 1 and 3 were determined completely syntactically; they could be generated mechanically. Most of the next level is also determined syntactically—for example, the fact that $Inv.2$ has the form $\forall i \in S : F(i) \wedge G(i)$ determines the high level outline of the proofs of 1.2.2.1–1.2.2.5.

Because TLA is a formal logic, every step in the proof hierarchy is a precisely defined mathematical formula. Prose appears only in the lowest-level proof. We can obtain a more reliable proof by replacing a prose proof with one that is hierarchically structured. Ultimately, we would arrive at a proof in which every step is purely syntactic, justified by the direct application of a single proof rule—either of TLA or of predicate logic. However, before that level of detail were reached, we could replace the prose by instructions to a mechanical theorem prover [7].

Steps 1–3 prove a safety property of the complete cache. They consist of action reasoning, with essentially no temporal logic. These steps are the TLA version of the standard invariance and step-simulation proofs of methods based on toy programming languages [22] and automata [9, 19]. However, our proofs are completely formal.

The proof of fairness in step 4 uses nontrivial temporal logic reasoning. We know of no better formalism than temporal logic for writing rigorous proofs of fairness properties. Had we done the proof in more detail, we would have relied heavily on the TLA rules WF1 and WF2, which use action reasoning to derive temporal logic formulas. (Rule WF1 is the TLA version of Manna and Pnueli's "single-step" rule for "just" transitions [20]. Because their method is not hierarchical, Manna and Pnueli have no analog of rule WF2.) Even for fairness properties, action reasoning forms the bulk of a detailed TLA proof.

When we fill in more levels of detail in our proof, we discover that certain facts about actions are used in several places. For example, a closer examination of our informal proof of step 4.1.2.1 reveals that it implicitly uses step 3.1.1.5. Our hierarchical proof style allows the proof of 4.1.2.1 to invoke step 3, but not any of its substeps [15]. Results that are used in several steps must either be moved to a higher level in the proof, or else proved in separate lemmas. We prefer to restructure a proof rather than adding an array of external lemmas, so the structure of a complete, detailed proof will differ from that of the proof presented here.

4.5 Epilogue

Structured hand proofs are much more reliable than conventional mathematical proofs, but not as reliable as mechanically checked ones. Writing such a proof is significantly easier than mechanical verification and we believe that, when done carefully, it is almost as reliable. The major parts of the proofs of the theorems in module *CacheCorrectness* have been carried out to a very detailed level. (All that remains is the low-level proof of theorem *CCequivACC*.) We recently tested how good these proofs were by using two tools under development: a TLA$^+$ parser, being written by Jean-Charles Grégoire, and the TLC model checker for a subclass of TLA$^+$ specifications, being written by Yuan Yu. We now describe what we found.

The specifications that appear above were all formated in LaTeX. The syntax of TLA$^+$ had changed in the three years since we submitted the previous version of this article, so we first modified the specifications to conform to the current syntax. All but one of these modifications were to the declarations; we had to rewrite one short formula because a construct had been removed from TLA$^+$.

We manually converted the LaTeX version of all the modules to ASCII and ran the parser on them. The parser can detect the usual syntactic errors as well as undefined or multiply-defined identifiers. The parser found no errors—except for ones introduced in the translation to ASCII.[12]

We then applied the TLC model checker to the specifications *LazyCache*, *CCache*, and *ACCache*.[13] To use TLC, one describes a finite-state model by giving explicit values to constant parameters (like the number N of processors and the set *Addr* of addresses) and specifying constraints on the maximum lengths of queues. When released, TLC should handle these three

[12] Because of a bug in the parser, a small part of the specification was not checked for undefined identifiers.

[13] The specifications *SeqDB*1 and *SeqDB*2 do not have the canonical form $I \wedge \Box[N]_v \wedge L$ required by TLC. Although they can be put into that form by simple logical manipulation (and ignoring hiding), the resulting specifications are not machine closed. TLC uses only the initial predicate and next-state action, so the reachable states it computes are not all reachable if the specification is not machine closed.

specifications as written, but the current version required us to make many small modifications to them.

TLC checks for invariance, enumerating all reachable states and reporting an error if it finds one that does not satisfy a specified invariant. We checked all three specifications with the type-correctness invariant. We also checked *ACCache* with the invariant *Inv* defined in Figure 22. We used models having two processors, two data values, and two addresses, and with the following maximum queue lengths:

	out/cout	*in/cin*	*vcq*	*vrq*
LazyCache	1	2		
CCache	2	3		
ACCache	1	3	4	2

Checking each of these models involved examining millions of states. We found one error: the fourth line in the definition of *ARead* in module *ACCache* originally was $\langle i, \text{“Rd”}, d, a \rangle \circ s$ instead of $\langle \langle i, \text{“Rd”}, d, a \rangle \rangle \circ s$. A minor problem with the current version of TLC required that we replace $\circ$ with a prefix operator, and we noticed this error when modifying the expression. Had we not seen it then, we would have found it quickly when TLC reported a trace that violated the type-correctness invariant.

Invariance for a finite model does not imply invariance for the actual specification. Fairly small models are usually enough to catch errors in a simple type invariant. Our model for *ACCache* is probably large enough to have discovered if *Inv* is not an invariant of the specification. However, an invariant of a specification is one that is true of all reachable states. We proved the stronger condition that *Inv* is an invariant of the next-state relation—meaning that it is true in any state reachable from a state satisfying *Inv*, not just in states reachable from an initial state. TLC cannot check this property for a large enough model.

These tests suggest that structured hand proofs are effective in eliminating errors in specifications. The absence of syntax errors in about 375 lines of specification and the apparent absence of type-invariance errors in 200 of those lines show that one reads a specification very carefully when reasoning about it. The proof of invariance of *Inv* is the largest single part of our proof, and TLC's inability to find an error in it speaks well for the proof method. But the one error we did find reminds us that hand proofs are not perfect.

References

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

[2] Martín Abadi, Leslie Lamport, and Stephan Merz. Refining specifications. To appear.

[3] Yehuda Afek, Geoffrey Brown, and Michael Merritt. Lazy caching. *ACM Transactions on Programming Languages and Systems*, 15(1):182–205, January 1993.

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

[5] K. Mani Chandy and Jayadev Misra. *Parallel Program Design.* Addison-Wesley, Reading, Massachusetts, 1988.

[6] Richard A. De Millo, Richard J. Lipton, and Alan J. Perlis. Social processes and proofs of theorems and programs. *Communications of the ACM*, 22(5):271–280, May 1979.

[7] Urban Engberg, Peter Grønning, and Leslie Lamport. Mechanical verification of concurrent systems with TLA. In G. v. Bochmann and D. K. Probst, editors, *Proceedings of the Fourth International Conference on Computer Aided Verification*, volume 663 of *Lecture Notes in Computer Science*, pages 44–55, Berlin, June 1992. Springer-Verlag. Proceedings of the Fourth International Conference, CAV'92.

[8] Rob Gerth. Introduction to sequential consistency and the lazy caching algorithm. *Distributed Computing*, 1995.

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

[10] Leslie Lamport. What good is temporal logic? In R. E. A. Mason, editor, *Information Processing 83: Proceedings of the IFIP 9th World Congress*, pages 657–668, Paris, September 1983. IFIP, North-Holland.

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

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

[13] Leslie Lamport. How to write a long formula. *Formal Aspects of Computing*, 6:580–584, 1994. First appeared as Research Report 119, Digital Equipment Corporation, Systems Research Center.

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

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

[16] Leslie Lamport. How to make a correct multiprocess program execute correctly on a multiprocessor. *IEEE Transactions on Computers*, 46(7):779–782, July 1997.

[17] Leslie Lamport and Stephan Merz. Specifying and verifying fault-tolerant systems. In H. Langmaack, W.-P. de Roever, and J. Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science*, pages 41–76. Springer-Verlag, September 1994.

[18] Leslie Lamport and Lawrence C. Paulson. Should your specification language be typed? Research Report 147, Digital Equipment Corporation, Systems Research Center, May 1997.

[19] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, San Mateo, California, 1995.

[20] Zohar Manna and Amir Pnueli. Completing the temporal picture. *Theoretical Computer Science*, 83(1):97–130, 1991.

[21] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, New York, 1991.

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