A CSP approach to sequential consistency

Gavin Lowe[1] and Jim Davies[2]

Abstract

In this paper, we show how the process notation of Communicating Sequential Processes (CSP) may be used to describe a lazy caching protocol. We then show that the traces model for CSP can be used to specify the property of sequential consistency. The model-based approach of CSP allows us to demonstrate that the protocol description satisfies the specification, thus verifying that the lazy caching protocol is sequentially consistent.

1 Introduction

In shared-memory multiprocessor systems, the time taken to perform memory access operations is critical. In most designs, this is reduced by equipping each processor with a cache: a local image of the shared store. If each cache contains a copy of the locations that the corresponding processor is most likely to access, then any delay due to shared memory access will be minimised.

However, if a system contains multiple copies of the same datum, care must be exercised if the system is to behave in a predictable and satisfactory fashion. Whenever a processor updates some location, any caches which contain a copy of that location must be updated to match. This is the rôle of the consistency protocol.

Many consistency protocols operate by marking other copies as invalid, so that subsequent access requires a read from shared memory. This marking must be done immediately: no further reads or writes can occur until all caches have been marked. In highly-distributed multiprocessor systems, the delay caused by such atomic 'write and mark' operations is unacceptable. Such systems require a more relaxed view of data consistency.

In [Lam79], Lamport introduced the notion of *sequential consistency*:

> *The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.*

[1]Programming Research Group, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, OX1 3QD, England. Currently working in ESPRIT BRA Project No. 6021: "Building Correct Reactive Systems (REACT)".

[2]Department of Computer Science, University of Reading, Reading, RG6 2AY, England. Funded by DRA Malvern.

This notion is now widely employed in multiprocessor designs.

In [ABM93] the authors propose an algorithm—the *lazy caching protocol*—for ensuring sequential consistency. In this paper we use the traces model of Communicating Sequential Processes (CSP) [Hoa85] to verify this protocol.

In section 2 we give a brief overview of the syntax of CSP, and in section 3 we describe the traces model, and how it may be used to specify and verify processes. We describe the lazy caching protocol in section 4, and show how it may be encoded as a CSP process in section 5. In section 6 we express the notion of sequential consistency as a trace specification, and in sections 7 and 8 we verify that the protocol meets the specification: we use the proof system to derive a property that holds of all runs of the protocol, and show that this is enough to imply sequential consistency.

2 Processes

Communicating Sequential Processes is a language for describing patterns of communication. Each pattern is represented by an abstract program, or *process*, which records the points at which certain communications may take place. These processes may be combined to produce a description of a system in terms of its components.

In CSP, we use abstract entities called events to model important points and actions in a history, or execution, of a system. We may then obtain a workable description of the system in terms of these events; this description may contain information about the order in which certain events may occur, the times at which they may occur, and how they might be blocked or prevented from occurring.

The simplest process may perform no events, and is written Stop. This represents the end of a communication pattern. We introduce events using the prefix operator $\longrightarrow$: the process $a \longrightarrow \mathit{Stop}$ represents a system that is able to engage in a single event a before stopping. A choice between patterns of communication is provided by either of the choice operators $\Box$ and $\sqcap$, representing external (deterministic) and internal (nondeterministic) choice, respectively.

Processes may be placed in parallel combination using the $\parallel$ operator; in the parallel composition

$$P \underset{A}{\parallel} Q$$

the two processes P and Q evolve independently, but must cooperate upon every occurrence of any event from the shared set A. This set represents the interface where the two meet and data may be transferred. If the interface consists of precisely those events that are common to the two process descriptions, then we will omit the shared set parameter.

Processes may be placed in sequential combination using the ; operator. The sequential composition $P\ ;\ Q$ behaves as process P until that process terminates successfully, and then behaves as Q. We use the process *Skip* to mark a successful end to a pattern of communication.

We will use indexed forms of the above operators to represent networks and series of processes:

$$\|_{i \in 1\,..\,n} P_i$$

denotes a parallel combination of processes with index i ranging over a finite set $1\,..\,n$, while

$$⨟_{i \in 1\,..\,n} P_i$$

denotes the same collection of processes, executing sequentially.

In this application, processes will share only compound events, representing value-passing communications along named channels. A process that is ready to transmit value v on channel c and then behave according to the description P would be written as $c!v \longrightarrow P$, while a process that is ready to accept a value on channel c and then behave as Q would be written $c?x \longrightarrow Q$. The subsequent behaviour Q may be parameterised by x, a variable that will be bound to the value transmitted along the channel c.

We may conceal or abstract away events from a process description using the hiding operator $\setminus$. The process $P \setminus A$ behaves as P except that events from the set A are no longer visible to the environment of the process; they are encapsulated within. Finally, we will want to relabel processes. The process $i.P$ is the same as P except all events a are renamed to $i.a$.

3 Traces

A variety of denotational semantic models have been formulated for the process language of CSP. In each model, a process is associated with the set of observations that may be made during its execution. In the simplest model each process is associated with the set of event sequences or *traces* that may be recorded during execution. We may use this model to specify safety properties: requirements that no undesirable events should occur.

3.1 Semantics

If Σ is the set of all communication events, then the trace semantics of our language is given by a semantic function

$$\mathit{Traces} : \mathit{CSP} \to \mathbf{P}(\mathrm{seq}\,\Sigma)$$

mapping processes to sets of sequences of events. As usual, the semantic function is defined by structural recursion upon the language syntax: e.g.,

$$\begin{aligned} \mathit{Traces}[\![\mathit{Stop}]\!] &= \{\langle\rangle\} \\ \mathit{Traces}[\![a \longrightarrow P]\!] &= \{\langle\rangle\} \cup \{\langle a\rangle ^\frown tr \mid tr \in \mathit{Traces}[\![P]\!]\} \end{aligned}$$

where $\langle\rangle$ denotes the empty trace and $\frown$ denotes concatenation of traces.

If we impose an order upon the semantic model, then we may give a meaning to recursively-defined processes. Given an equation $P = F(P)$, where F is a syntactic function constructed from process operators, we define the semantics of P to be the least fixed point of the corresponding function in the semantic model. For this to be a good definition, we must insist that each recursive call of P on the right-hand side is guarded by at least one communication event.

The semantic model can also be used to justify a notion of equivalence for process terms:

$$P \equiv Q \quad \Leftrightarrow \quad \mathit{Traces}[\![P]\!] = \mathit{Traces}[\![Q]\!]$$

which leads to a complete set of algebraic laws for rewriting processes: for example,

$$(a \longrightarrow \mathit{Skip}) \mathbin{;} P \quad \equiv \quad a \longrightarrow P$$

This states that the sequential composition of the process $a \longrightarrow \mathit{Skip}$ with P is equivalent to the process that performs a and then behaves as P.

3.2 Specification

Apart from ensuring consistency of process definitions, and justifying algebraic laws for equivalence and refinement, a denotational model may be used to support model-oriented specification. If each process is associated with a set of observations, then constraints upon observation sets may be used to express requirements upon process behaviour. We write behavioural specifications as predicates upon observations.

For example, suppose that a process P is capable of performing—amongst other events—both of the events a and b. If we wish to specify that P never performs an a after a b, then we have only to insist that in any trace tr of process P, the event a never appears after b. Formally,

$$\forall\, tr_1, tr_2 \bullet tr = tr_1 \frown \langle b\rangle \frown tr_2 \Rightarrow tr_2 \upharpoonright \{a\} = \langle\rangle$$

that is, if tr contains a b event, then the part of tr following the b (tr_2) contains no occurrences of event a. To express this final condition, we have used the trace projection operator $\upharpoonright$, which removes from the trace any event that is not in the chosen set (here $\{a\}$).

To show that a process satisfies a behavioural specification, we must show that every trace of the process satisfies the corresponding predicate. We define

$$P \ \mathbf{sat}\ S(tr) \quad \Leftrightarrow \quad \forall\, tr \bullet tr \in \mathit{Traces}[\![P]\!] \Rightarrow S(tr)$$

Although the **sat** is a relation between process syntax and predicates, it may be seen as a refinement relation. If we identify the process P with its semantic set of observations, and the predicate $S(tr)$ with its characteristic set of observations, then **sat** states that every observation of P is an observation of S.

3.3 A proof system

The definitions chosen for the various process operators guarantee that the semantics are pointwise compositional: the properties of an observation of a compound process can be derived from the properties of observations of the components. As a result, we may exhibit a compositional proof system for trace specifications. Each semantic equation may be inverted to yield a natural deduction inference rule.

$$\frac{}{\mathit{Stop}\ \mathbf{sat}\ tr = \langle\rangle}$$

The basic process *Stop* satisfies the specification 'the trace is empty'.

Any non-empty trace of a prefixed process must begin with the prefixed event; this event must be equal to the *head* of the trace:

$$\frac{P\ \mathbf{sat}\ S(tr)}{a \longrightarrow P\ \mathbf{sat}\ tr = \langle\rangle \lor \mathit{head}(tr) = a \land S(\mathit{tail}(tr))}$$

The *tail* of the trace must be performed by P; if we know that this process satisfies S, then we may conclude that S holds of $\mathit{tail}(tr)$.

A trace of a choice process may have been performed by either component; we are thus left with a disjunction of specifications:

$$\frac{\begin{array}{l} P\ \mathbf{sat}\ S(tr) \\ Q\ \mathbf{sat}\ T(tr) \end{array}}{P \mathbin{\Box} Q\ \mathbf{sat}\ S(tr) \lor T(tr)}$$

The parallel combinator, on the other hand, gives rise to a conjunction of specifications. In the case in which the interface between the components includes every event that is common to both process descriptions, we obtain the following inference rule:

$$\frac{\begin{array}{l} P\ \mathbf{sat}\ S(tr) \\ Q\ \mathbf{sat}\ T(tr) \end{array}}{P \parallel Q\ \mathbf{sat}\ S(tr \upharpoonright \alpha P) \land T(tr \upharpoonright \alpha Q)}$$

We write αP to denote the set of events that appear in the description of process P. The projection $tr \upharpoonright \alpha P$ reveals the sequence of events performed by component P in this parallel combination.

The rule for network parallel combination is a simple generalisation of the binary case; with the same assumption about interfaces we obtain:

$$\frac{\forall\, i \in 1 \mathinner{..} n \bullet P_i\ \mathbf{sat}\ S_i(tr)}{\|_{i \in 1 \mathinner{..} n} P_i\ \mathbf{sat}\ \forall\, i \in 1 \mathinner{..} n \bullet S_i(tr \upharpoonright \alpha P_i)}$$

while the rule for hiding involves an existential quantification:

$$\frac{P \ \textbf{sat}\ S(tr)}{P \setminus A \ \textbf{sat}\ \exists\, tr_1 \bullet tr = tr_1 \setminus A \wedge S(tr_1)}$$

We may be uncertain about the order of internal events, but we know that there must be some internal trace that is consistent with our observation. The hiding operator $\setminus$ on traces simply strips the given events from the trace.

Output communication is simply syntactic sugar (the corresponding rule is a particular case of the rule for prefixing)

$$\frac{P \ \textbf{sat}\ S(tr)}{c!v \longrightarrow P \ \textbf{sat}\ tr = \langle\rangle \vee head(tr) = c.v \wedge S(tail(tr))}$$

while input communication is a form of choice; the subsequent behaviour remains to be determined by the incoming value.

$$\frac{\forall\, x \bullet Q \ \textbf{sat}\ S_x(tr)}{c?x \longrightarrow Q \ \textbf{sat}\ tr = \langle\rangle \vee \exists\, v \bullet head(tr) = c.v \wedge S_v(tail(tr))}$$

The rule for recursion insists that we establish a base case (showing that the specification is satisfiable) and then demonstrate that the specification is preserved by recursive calls:

$$\frac{\begin{array}{l} Stop \ \textbf{sat}\ S(tr) \\ \forall\, X \bullet X \ \textbf{sat}\ S(tr) \Rightarrow F(X) \ \textbf{sat}\ S(tr) \end{array}}{P \ \textbf{sat}\ S(tr)} \quad [\ P = F(P)\ ]$$

This rule is sound only if the recursive process is well-defined (it is enough to show that each recursive call is guarded).

The definition of **sat** allows us to derive logical rules for manipulating proof obligations: for example,

$$\frac{\begin{array}{l} P \ \textbf{sat}\ S(tr) \\ \forall\, tr \bullet S(tr) \Rightarrow T(tr) \end{array}}{P \ \textbf{sat}\ T(tr)}$$

It can be shown that the resulting proof system is sound and complete with respect to the trace semantics.

4 The lazy caching algorithm

In [ABM93] the authors propose a novel algorithm for ensuring sequential consistency. Each processor cache is equipped with input and output queues, allowing (1) cache updates to be postponed, while the processor reads possibly

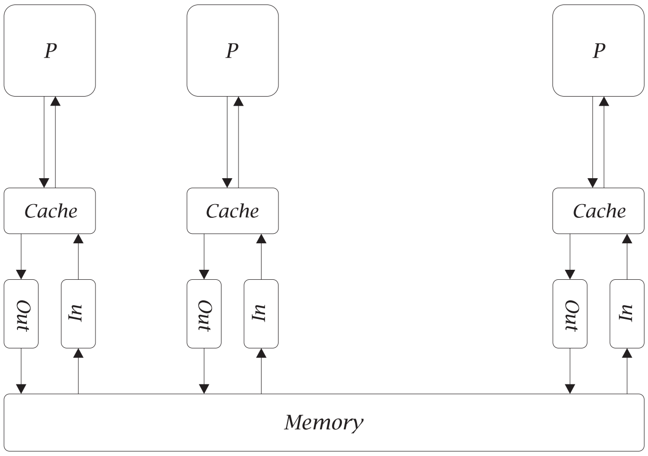

Figure 1: A lazy caching architecture

out-of-date data (2) memory updates to be queued, leaving sequences of write operations pending at each node. A suitable system is illustrated in Figure 1.

A write event does not have an immediate effect upon the shared memory state; instead, a request is placed in the output queue. Whenever a request is taken out of this queue, the memory is updated and a cache update request is placed in every input queue. In the case of the node responsible for the write event, the cache update request is marked when it arrives in the input queue: we say that it is *starred*.

A read event cannot occur until (1) the cache has a copy of the address concerned, (2) the output queue is empty, and (3) there are no more starred requests in the input queue. This discipline is enough to ensure that memory accesses are sequentially consistent throughout the system.

To obtain a CSP process description, we define a *Node* process for each processor. This buffers all communication to and from the shared memory, and consists of three components: a local cache, an output buffer, and an input buffer. Each user process communicates with the system via two channels r and w, used for reading and writing values, respectively.

The write channel w is connected directly to the output buffer, but the read channel is linked to all three components. Although the values passed on r are determined entirely by the cache, as we shall see, both buffers must agree to the communication taking place. There are three further channels in our description:

- ci, used for passing values from the input buffer to the cache;

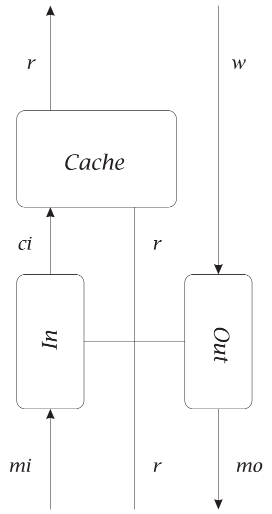

Figure 2: Cache and queues at a single node

- mi, used for passing values from the memory to the input buffer;
- mo, used for passing values from the output buffer to the memory.

These are internal channels, and will be hidden from the user process. The resulting *Node* process is illustrated in Figure 2.

To serve a collection of n user processes, we will require a network of node processes, labelled from 1 to n. These processes communicate via a shared memory, which forwards update messages to all input buffers. In this section, we will use a process *Memory* to describe the service provided by the shared memory interface.

Each node process is a parallel combination of three components:

$$Node \quad = \quad Cache \parallel In \parallel Out$$

The system contains a parallel combination of such nodes:

$$Nodes \quad = \quad ||_{i \in 1..n}\, i.Node$$

where $i.Node$ is the result of prefixing all communication events with an index i.

The two internal channels mi and mo will connect the memory to the input and output buffers at each node, while a third channel ci will connect the cache to the input buffer. These channels are not part of the interface to the memory system, so we conceal them using the hiding operator. The final description of the system is thus

$$System \quad = \quad (Nodes \parallel Memory) \setminus Internals$$

where *Memory* represents the service provided by the shared store, and

$$Internals \quad = \quad \{i.mi, i.mo, i.ci \mid i \in 1 \mathrel{..} n\}$$

denotes the set of all internal communications.

5 Process description

The output buffer is a queue for data messages. Each data message takes the form $a.d$, where a is an address and d is a data value. The communication $w.a.d$ from the user program is an instruction to update address a with value d. The output buffer is always ready to receive communications on channel w; these communications will be forwarded to the memory along channel mo in the order in which they were received.

If we model the output buffer as a CSP process Out, then this process will be indexed by a sequence variable, representing the current state of the buffer. Initially, this sequence is empty; at all times, it consists of the sequence of data messages that have yet to be forwarded to the memory.

$$\begin{array}{rcl}
Out & = & Out_{\langle\rangle} \\[1ex]
Out_{\langle\rangle} & = & w?a.d \longrightarrow Out_{\langle a.d \rangle} \\
& & \Box \\
& & r?a.d \longrightarrow Out_{\langle\rangle} \\[1ex]
Out_{\langle a'.d' \rangle ^\frown s} & = & w?a.d \longrightarrow Out_{\langle a'.d' \rangle ^\frown s ^\frown \langle a.d \rangle} \\
& & \Box \\
& & mo!a'.d' \longrightarrow Out_s
\end{array}$$

Whenever the output queue is empty, the process will allow communications to take place on channel r: the value passed is ignored by Out.

The input buffer is always ready to accept data messages from the memory on channel mi, and will forward these messages to the cache on channel ci. We may model the input buffer as a process In, similar in form to Out above.

$$\begin{array}{rcl}
In & = & In_{\langle\rangle} \\[1ex]
In_{\langle\rangle} & = & mi?a.d.f \longrightarrow In_{\langle a.d.f \rangle} \\
& & \Box \\
& & r?a.d \longrightarrow In_{\langle\rangle}
\end{array}$$

The behaviour of the input buffer is slightly different to that of the output buffer. It will allow communications on channel r whenever it contains no data messages that are marked with an asterisk: messages that have been flagged by the memory as urgent. Thus the buffer need not be empty for communications on r to occur; it may contain any number of non-urgent messages.

$$
\begin{array}{rcll}
In_{\langle a'.d'.f' \rangle ^\frown s} & = & mi?a.d.f \longrightarrow In_{\langle a'.d'.f' \rangle ^\frown s ^\frown \langle a.d.f \rangle} & \\
& & \Box & \\
& & ci!a'.d'.f' \longrightarrow In_s & \text{if } stars(\langle a'.d'.f' \rangle ^\frown s) \\
& = & mi?a.d.f \longrightarrow In_{\langle a'.d'.f' \rangle ^\frown s ^\frown \langle a.d.f \rangle} & \\
& & \Box & \\
& & ci!a'.d'.f' \longrightarrow In_s & \\
& & \Box & \\
& & r?a.d \longrightarrow In_{\langle a'.d'.f' \rangle ^\frown s} & \text{otherwise}
\end{array}
$$

The sequence predicate *stars* is true iff its argument includes a flagged data message. We may define

$$
stars(s) \quad \Leftrightarrow \quad s \upharpoonright \{ i.mi.a.d.\star \mid i \in 1 \mathinner{..} n \wedge a \in A \wedge d \in D \} = \langle \rangle
$$

where $\upharpoonright$ is the standard sequence/trace projection operator.

If we model the cache as a CSP process, then it will be indexed by a function variable, representing a mapping from address to data values. Initially, this takes the value *zero*, mapping each address to data value 0.

$$
\begin{array}{rcl}
Cache & = & Cache_{zero} \\
Cache_g & = & ci?a.d.f \longrightarrow Cache_{g \oplus a \mapsto d} \\
& & \Box \\
& & r?a!g(a) \longrightarrow Cache_g
\end{array}
$$

where *zero* is given by

$$
zero \quad \widehat{=} \quad \{ a \mapsto 0 \mid a \in Address \}
$$

At any time, the cache may be read via channel r; for any address a, it will return the value d stored at a, according to the local mapping g.

Because channel communication in CSP is fully synchronous, we may represent a read communication as a single event $r.a.d$. The composite notation used above abbreviates a choice construct

$$
r?a!g(a) \longrightarrow \ldots \quad = \quad \Box_{a \in A} \, r.a.g(a) \longrightarrow \ldots
$$

where A is the set of all addresses.

Our assumptions about the behaviour of the shared memory interface may also be described as a CSP process. This process accepts data messages from the output buffers and distributes them to all input buffers. In its initial state, the memory process will allow any communication of the form $i.r.a.d$: a read communication at node i. While the memory is distributing data, these communications are disabled.

$$\begin{array}{lcl} Memory & = & \Box_{i \in 1..n}\ i.mo?a.d \longrightarrow MemoryOut_{(i,a.d)} \,;\, Memory \\ & & \Box \\ & & \Box_{i \in 1..n}\ i.r?a.d \longrightarrow Memory \end{array}$$

The distribution process is a sequential composition of n processes, each passing the data message to a different input buffer.

$$\begin{array}{lcl} MemoryOut_{(i,a.d)} & = & ⨾_{j \in 1..n}\ \text{if}\ \ i = j\ \ \text{then}\ \ (j.mi.a.d.\star \longrightarrow Skip) \\ & & \qquad\qquad\qquad\quad \text{else}\ \ (j.mi.a.d \longrightarrow Skip) \end{array}$$

If the input buffer belongs to the same node process as the output buffer, then the memory will flag the data message with a star. This is the only aspect of the shared memory service that we need to consider.

6 Sequential consistency

We may express the property of sequential consistency as a trace specification. A trace tr is sequentially consistent if there is some trace of an ideal memory such that the order of reads and writes at each node is the same for both traces. Formally, we define a specification $SC(tr)$ on traces which holds exactly when tr is a trace of a sequentially consistent memory.

If predicate *Serial* holds exactly when tr is a trace of a serial memory, and predicate $Consistent(tr, tr')$ holds exactly when the two traces tr and tr' agree upon the order of reads and writes as seen from each node, then

$$SC(tr) \quad \Leftrightarrow \quad \exists\, tr' \bullet Serial(tr') \wedge Consistent(tr, tr')$$

That is tr is a trace of a sequentially consistent memory exactly when there is another trace tr' which is (1) a trace of a serial memory, and (2) consistent with tr.

A trace of an ideal memory is a trace in which every read communication passes the last data value written to the chosen address. If we write $\leq$ to denote the prefix relation between traces, then

$$Serial(tr') \quad = \quad \forall\, tr_0 \bullet tr_0 \frown \langle i.r.a.d \rangle \leq tr' \Rightarrow value_{I.w.a}(tr_0) = d$$

that is, for any prefix of tr' ending in an event of the form $i.r.a.d$, the value d passed must be the last value written to a. For convenience, we write I to denote the indexing set $1 \,..\, n$, and define

$$I.w \quad = \quad \{ i.w \mid i \in I \}$$

to denote the set of all write channels.

The function *value* returns the data value last written, or *0* if no write has occurred since the system was initialised.

$$\begin{array}{lcl} value_{C.a}(tr) = d & \Leftrightarrow & tr \upharpoonright C.a.D = \langle\rangle \wedge d = 0 \\ & & \vee \\ & & last(tr \upharpoonright C.a.D) = c.a.d \end{array}$$

The function *last* returns the last element of a trace, and the set of events $C.a.D$ marks the set of events

$$\{c.a.d \mid c \in C \wedge d \in D\}$$

for some set of channels C and the set of all data values D. Here, the set C is the set of all write channels $I.w$.

Two traces are consistent if they agree upon the order of external events at each node:

$$Consistent(tr, tr') \quad = \quad \forall\, i \bullet tr \upharpoonright E_i = tr' \upharpoonright E_i$$

where E_i is the external interface at node i, the set $\{i.r, i.w\}$. Two consistent traces may differ in the order of internal communications, and in the relative order of external communications at different nodes.

To show that the lazy caching algorithm guarantees sequential consistency, we must establish that

$$System \quad \textbf{sat} \quad SC(tr)$$

that is, that every trace tr of our implementation satisfies the specification SC defined above.

7 Component properties

As part of the verification process, we will identify the salient properties of each system component: the contributions that each makes towards our guarantee of sequential consistency.

The output buffer acts as a queue: the sequence of messages output must be a prefix of the sequence of messages input. Furthermore, read events may occur only whenever the buffer is empty. We define a behavioural specification for the output buffer at node i

$$\begin{array}{lcl} i.OUT(tr) & = & tr \Downarrow i.mo \leq tr \Downarrow i.w \\ & & \wedge \\ & & \forall\, tr_0 \bullet tr_0 {}^\frown \langle i.r.a.d\rangle \leq tr \Rightarrow tr_0 \Downarrow i.mo = tr_0 \Downarrow i.w \end{array}$$

If a read event occurs, then the input message sequence must match the output message sequence. We write $tr \Downarrow c$ to denote the sequence of data messages passed on channel c during trace tr.

The input buffer also acts as a queue. However, read events are enabled only when there are no flagged data messages held in the buffer. A suitable behavioural specification of the input buffer at node i would be

$$\begin{array}{rcl} i.IN(tr) & = & tr \Downarrow i.ci \leq tr \Downarrow i.mi \\ & & \wedge \\ & & \forall\, tr_0 \bullet tr_0 \frown \langle i.r.a.d \rangle \leq tr \Rightarrow tr_0 \Downarrow i.ci.\star = tr_0 \Downarrow i.mi.\star \end{array}$$

We write $tr \Downarrow c.\star$ to denote the sequence of flagged messages passed on channel c during trace tr.

The cache acts as a serial memory. Whenever a read event occurs, the data value passed must be the last value written to the address in question. For node i, a suitable behaviour specification would be

$$i.CACHE(tr) \;=\; \forall\, tr_0 \bullet tr_0 \frown \langle i.r.a.d \rangle \leq tr \Rightarrow value_{i.ci.a}(tr_0) = d$$

where $value$ is as defined above.

The memory process may allow read events only when every data message accepted from any channel $i.mo$ has been redistributed to all channels of the form $j.mi$. In any case, the starred data placed onto an $i.mi$ channel will be a prefix of the data taken off the corresponding mo channel.

$$\begin{array}{rcl} MEMORY(tr) & = & \forall\, i \bullet tr \Downarrow i.mi.\star \leq tr \Downarrow i.mo \\ & & \quad \wedge \\ & & \quad \forall\, tr_0 \bullet tr_0 \frown \langle i.r.a.d \rangle \leq tr \Rightarrow \\ & & \qquad\qquad \forall\, j \bullet tr_0 \Downarrow j.mo = tr_0 \Downarrow j.mi.\star \\ & & \qquad\qquad\quad \wedge \\ & & \qquad\qquad\quad tr_0 \Downarrow j.mi = tr_0 \Downarrow I.mo \end{array}$$

If the input buffer belongs to the source node for the current data message, then the outgoing copy must be flagged.

Using the rules of the proof system, it is easy to establish that each of the sequential processes exhibits a satisfactory pattern of communication: i.e., that

$$\begin{array}{rcl} i.In & \mathbf{sat} & i.IN(tr) \\ i.Out & \mathbf{sat} & i.OUT(tr) \\ i.Cache & \mathbf{sat} & i.CACHE(tr) \\ Memory & \mathbf{sat} & MEMORY(tr) \end{array}$$

where *In*, *Out*, *Cache*, and *Memory* are as defined in the previous section.

7.1 System properties

The component specifications may be combined using the proof rule for parallel composition. In this way, we can establish that

$$
\begin{array}{lll}
i.Cache \parallel i.In & \textbf{sat} & tr \Downarrow i.ci \le tr \Downarrow i.mi \wedge \\
& & \forall\, tr_0 \bullet tr_0 {}^\frown \langle i.r.a.d \rangle \le tr \\
& & \qquad \Rightarrow \\
& & \qquad tr_0 \Downarrow i.ci.\star = tr_0 \Downarrow i.mi.\star \wedge \\
& & \qquad value_{i.ci.a}(tr_0) = d
\end{array}
$$

Using the same rule once more, we obtain

$$
\begin{array}{lll}
i.Cache \parallel i.In \parallel i.Out & \textbf{sat} & tr \Downarrow i.ci \le tr \Downarrow i.mi \wedge \\
& & tr \Downarrow i.mo \le tr \Downarrow i.w \wedge \\
& & \forall\, tr_0 \bullet tr_0 {}^\frown \langle i.r.a.d \rangle \le tr \\
& & \qquad \Rightarrow \\
& & \qquad tr_0 \Downarrow i.ci.\star = tr_0 \Downarrow i.mi.\star \wedge \\
& & \qquad tr_0 \Downarrow i.mo = tr_0 \Downarrow i.w \wedge \\
& & \qquad value_{a,i.ci}(tr_0) = d
\end{array}
$$

The rule for network parallel combination may then be applied to yield the following statement (recall that $Nodes = \;||_{i \in 1..n}\; i.Node$):

$$
\begin{array}{lll}
Nodes & \textbf{sat} & \forall\, i \bullet tr \Downarrow i.ci \le tr \Downarrow i.mi \wedge \\
& & \quad tr \Downarrow i.mo \le tr \Downarrow i.w \wedge \\
& & \quad \forall\, tr_0 \bullet tr_0 {}^\frown \langle i.r.a.d \rangle \le tr \\
& & \qquad \Rightarrow \\
& & \qquad tr_0 \Downarrow i.ci.\star = tr_0 \Downarrow i.mi.\star \wedge \\
& & \qquad tr_0 \Downarrow i.mo = tr_0 \Downarrow i.w \wedge \\
& & \qquad value_{i.ci.a}(tr_0) = d
\end{array}
$$

A final application of the rule for parallel composition yields

$$
Memory \parallel Nodes \quad \textbf{sat} \quad Spec(tr)
$$

where

$$
\begin{array}{lll}
Spec(tr) & \widehat{=} & \forall\, i \bullet tr \Downarrow i.mi.\star \le tr \Downarrow i.mo \wedge \\
& & \quad tr \Downarrow i.ci \le tr \Downarrow i.mi \wedge \\
& & \quad tr \Downarrow i.mo \le tr \Downarrow i.w \wedge \\
& & \quad \forall\, tr_0 \bullet tr_0 {}^\frown \langle i.r.a.d \rangle \le tr \Rightarrow \\
& & \qquad tr_0 \Downarrow i.ci.\star = tr_0 \Downarrow i.mi.\star \wedge \\
& & \qquad tr_0 \Downarrow i.mo = tr_0 \Downarrow i.w \wedge \\
& & \qquad value_{i.ci.a}(tr_0) = d \wedge \\
& & \qquad \forall\, j \bullet tr_0 \Downarrow j.mo = tr_0 \Downarrow j.mi.\star \wedge \\
& & \qquad\quad tr_0 \Downarrow j.mi = tr_0 \Downarrow I.mo
\end{array}
$$

We have shown that whenever tr is a trace of $Nodes \parallel Memory$, then tr must satisfy the specification $Spec$. In addition, all prefixes of tr will also satisfy $Spec$.

From our definition of sequential consistency, it is easy to see that this property is invariant under the hiding of internal channels $Internals$: if tr' is such that $tr = tr' \setminus Internals$, then $SC(tr') \Leftrightarrow SC(tr)$.

Thus, we have reduced our proof obligation to

$$(\forall\, tr' \leq tr \bullet Spec(tr')) \quad \Rightarrow \quad SC(tr)$$

8 Verifying sequential consistency

To demonstrate that $Spec(tr)$ is sufficient for sequential consistency, we will show that, for any trace tr_1 satisfying $Spec$—and all of whose prefixes satisfy $Spec$—there is a consistent trace tr_2 which satisfies $Serial$. We may construct this trace by permuting tr_1:

$$tr_2 = (M_w \circ M_r)\ tr_1$$

where each M is a permutation function on traces.

The function M_r moves each read event $i.r$ to a position just after the last $j.mo$ event for which a corresponding $i.ci$ event has occurred. In other words, the read is moved to a position after just those mo events whose effects have filtered through node i's input buffer to have an effect on i's cache. If k $i.ci$ events have occurred, then the read is moved to just after the kth mo. Formally,

$$\begin{array}{l}
\forall\, tr : \mathrm{seq}\,\Sigma \bullet tr \setminus I.r = M_r(tr) \setminus I.r \wedge \\
\qquad\qquad \forall\, i \bullet tr \upharpoonright i.r = M_r(tr) \upharpoonright i.r \wedge \\
\qquad\qquad \forall\, tr_0 \bullet tr_0 ^\frown \langle i.r.a.d \rangle \leq M_r(tr) \Rightarrow \\
\qquad\qquad\qquad\qquad \exists\, tr_0' \bullet tr_0' {}^\frown \langle i.r.a.d \rangle \leq tr \wedge \\
\qquad\qquad\qquad\qquad\qquad \#(tr_0' \upharpoonright i.ci) = \#(tr_0 \upharpoonright I.mo)
\end{array}$$

We insist that: only read events are moved; the order of read events at a particular node is unchanged; a read event at node i is moved back to just after the last mo whose effect has been seen at i.

The function M_w moves each write event $i.w$ to a position just before the corresponding $i.mo$ event. In other words, write effects are moved to the point at which their effect is experienced by the memory. The kth $i.w$ is moved to a position just before the kth $i.mo$—if the corresponding $i.mo$ has not yet occurred, then we move $i.w$ to the end of the trace. Formally,

$$
\begin{array}{l}
\forall\, tr, tr' \bullet tr' = M_w(tr) \Rightarrow \\
\quad \forall\, i \bullet tr \Downarrow i.mo \leq tr \Downarrow i.w \;\wedge \\
\quad tr \setminus I.w = tr' \setminus I.w \;\wedge \\
\quad \forall\, i \bullet tr \upharpoonright i.w = tr' \upharpoonright i.w \;\wedge \\
\quad \forall\, tr_0 \bullet tr_0 \frown \langle i.mo.a.d \rangle \leq tr' \Rightarrow \mathit{last}\; tr_0 = i.w.a.d \;\wedge \\
\quad \forall\, tr_0, tr_1 \bullet tr_0 \frown \langle i.w.a.d \rangle \frown tr_1 = tr \Rightarrow \mathit{first}\; tr_1 = i.mo.a.d \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad tr_1 \setminus I.w = \langle\rangle
\end{array}
$$

We insist that: the function is only defined on those traces tr where the $i.mo$ events are a prefix of the $i.w$ events; only the write events are moved; write events from a particular node are not reordered; every mo in the resulting trace is preceded by the write event that caused it; every write is moved to a position where it either precedes the corresponding mo, or is followed only by other writes—the write events at the end of the trace are those for which the corresponding mo event has yet to occur.

The composition of these functions moves each write event to the point where its message is accepted by the memory, and each read event to the point where the last update for that address was placed on the corresponding *In* queue. In the resulting trace, each read event is placed after precisely those write events whose effects have reached the memory.

Having defined the functions, we must show that

- the composition $M_w \circ M_r$ is defined for every trace of the process $\mathit{Nodes} \parallel \mathit{Memory}$;
- any trace in the range of $M_w \circ M_r$ is a serial trace;
- any trace tr is consistent with its image $(M_w \circ M_r)\; tr$.

The first of these requirements is easily met. The others will need some careful reasoning.

Applicability

To show that our permutation functions may be successfully applied to any trace of our system, we must show that every trace tr_1 of the system is in the domain of the function M_r, and that $M_r(tr_1)$ lies in the domain of function M_w.

We must show that if $tr_0 \frown \langle i.r.a.d \rangle \leq tr_1$ then the event $i.r.a.d$ can be moved forward to follow k $I.mo$ events, where $k = \#(tr_0 \upharpoonright i.ci)$. This is equivalent to showing that

$$\#(tr_0 \upharpoonright i.ci) \leq \#(tr_0 \upharpoonright I.mo)$$

which is easy to establish: tr_0 is also a trace of the system, and so must satisfy *Spec*; the second and last conjuncts of *Spec* establish the result.

We have then to show that the trace $tr_1' = M_r(t_1)$ satisfies

$$tr_1' \Downarrow i.mo \leq tr_1' \Downarrow i.w$$

which follows from the corresponding result for tr_1, given that M_r only moves read events.

Serialization

To show that every trace in the range of $M_w \circ M_r$ is a serial trace, we must show that an appropriate value is returned on every read event. We begin by showing that if $Spec(tr_1)$ then the data messages read during $M_r(tr_1)$ agree with the data messages passed on the mo channels: i.e., that

$$\forall\, tr_0 \bullet tr_0 {}^\frown \langle i.r.a.d \rangle \leq M_r(tr_1) \Rightarrow value_{I.mo.a}(tr_0) = d$$

Suppose that

$$tr_0 {}^\frown \langle i.r.a.d \rangle \leq M_r(tr_1)$$

then from the form of M_r, there exists some tr_0' such that

$$tr_0' {}^\frown \langle i.r.a.d \rangle \leq tr_1 \ \wedge \ \#(tr_0' \upharpoonright i.ci) = \#(tr_0 \upharpoonright I.mo)$$

From $Spec(tr_1)$ we have that $value_{i.ci.a}(tr_0') = d$ whence

$$\begin{array}{lll} & tr_0' \Downarrow i.ci & \\ \leq & tr_0' \Downarrow i.mi & (\text{from } Spec(tr_0')) \\ = & tr_0' \Downarrow I.mo & (\text{from } Spec(tr_1)) \\ \leq & tr_1 \Downarrow I.mo & (tr_0' \leq tr_1) \\ = & M_r(tr_1) \Downarrow I.mo & (\text{definition of } M_r) \end{array}$$

Also, since $tr_0 \leq M_r(tr_1)$, we have

$$tr_0 \Downarrow I.mo \leq M_r(tr_1) \Downarrow I.mo$$

Hence $tr_0' \Downarrow i.ci$ and $tr_0 \Downarrow I.mo$ are both prefixes of $M_r(tr_1) \Downarrow I.mo$. But they are of the same length so they must be equal. We conclude that

$$value_{I.mo.a}(tr_0) = value_{i.ci.a}(tr_0') = d$$

as required.

We must now show that $(M_w \circ M_r)\ tr_1$ is serial:

$$\forall\, tr_0 \bullet tr_0 {}^\frown \langle i.r.a.d \rangle \leq M_w(M_r(tr_1)) \Rightarrow value_{I.w.a}(tr_0) = d$$

Suppose that

$$tr_0 {}^\frown \langle i.r.a.d \rangle \leq M_w(M_r(tr_1))$$

From the definition of M_w, there is some trace tr'_0 such that

$$tr'_0 {}^\frown \langle i.r.a.d \rangle \leq M_r(t_1) \ \wedge \ tr'_0 \setminus I.w = tr_0 \setminus I.w$$

From the above, we have that $value_{I.mo.a}(tr'_0) = d$. Also

$$\begin{array}{lll} & tr_0 \Downarrow I.w & \\ = & tr_0 \Downarrow I.mo & \text{(from } Spec(tr_1)\text{, and since } M_r \text{ moves only reads)} \\ = & tr'_0 \Downarrow I.mo & (M_w \text{ only moves write events}) \end{array}$$

We may conclude that $value_{I.w.a}(tr_0) = value_{I.mo.a}(tr'_0) = d$. The trace is question is thus a serial trace.

Consistency

To show that $M_w \circ M_r$ preserves consistency, it is sufficient to show that each of M_r and M_w preserves consistency when considered separately. We argue that neither M_r or M_w allows reordering between $i.r$ events, or between $i.w$ events. It is therefore enough to show that the transformations do not alter the relative order of $i.r$ and $i.w$ events.

To see that M_r cannot move an $i.r$ event past an $i.w$ event, suppose that a trace of the form $tr_0 {}^\frown tr_1 {}^\frown \langle i.r \rangle {}^\frown tr_2$ is transformed into $tr'_0 {}^\frown \langle i.r \rangle {}^\frown tr'_1 {}^\frown tr'_2$ where $tr_i \setminus I.r = tr'_i \setminus I.r$ for $i = 0, 1, 2$, and

$$\#((tr_0 {}^\frown tr_1) \upharpoonright i.ci) = \#(tr'_0 \upharpoonright I.mo)$$

We suppose for a contradiction that the $i.r$ passes an $i.w$ event, i.e. tr_1 is of the form $tr_3 {}^\frown \langle i.w.a.d \rangle {}^\frown tr_4$. Then from $Spec$ we have

$$(tr_0 {}^\frown tr_3 {}^\frown \langle i.w.a.d \rangle {}^\frown tr_4) \Downarrow i.mo = (tr_0 {}^\frown tr_3 {}^\frown \langle i.w.a.d \rangle {}^\frown tr_4) \Downarrow i.w$$

and

$$(tr_0 {}^\frown tr_3) \Downarrow i.mo \leq (tr_0 {}^\frown tr_3) \Downarrow i.w$$

Hence there must be a $i.mo.a.d$ event in tr_4. A similar argument shows that there must be $i.mi.a.d.\star$ and $i.ci.a.d.\star$ events in tr_4 after the $i.mo.a.d$, and if the $i.ci.a.d.\star$ is the kth $i.ci$ event, then the $i.mo.a.d$ is the kth $I.mo$ event. But then

$$\#((tr_0 {}^\frown tr_1) \upharpoonright i.ci) > \#(tr_0 \upharpoonright I.mo) = \#(tr'_0 \upharpoonright I.mo)$$

since $tr_0 \setminus I.r = tr'_0 \setminus I.r$, which contradicts the statement above.

To see that M_w does not alter the relative order of $i.r$ and $i.w$ events, note that the function M_w moves each $i.w$ event to just before the corresponding $i.mo$ event. Suppose, for a contradiction, that the $i.w$ passes a $i.r$ event. Then there are two cases to consider.

Firstly, consider the case where the $i.w$ moves to a position before the *original* position of the $i.r$, i.e. the original trace is of the form

$$tr_0 \frown \langle i.w.a.d \rangle \frown tr_1 \frown tr_2 \frown \langle i.mo.a.d \rangle \frown tr_3 \frown \langle i.r.a'.d' \rangle \frown tr_4$$

which is transformed by M_r to

$$tr'_0 \frown \langle i.w.a.d \rangle \frown tr'_1 \frown \langle i.r.a'.d' \rangle \frown tr'_2 \frown \langle i.mo.a.d \rangle \frown tr'_3 \frown tr'_4$$

which is transformed by M_w to

$$tr''_0 \frown tr''_1 \frown \langle i.r.a'.d' \rangle \frown tr''_2 \frown \langle i.w.a.d \rangle \frown \langle i.mo.a.d \rangle \frown tr''_3 \frown tr''_4$$

where

$$\begin{aligned}&\#((tr_0 \frown \langle i.w.a.d \rangle \frown tr_1 \frown tr_2 \frown \langle i.mo.a.d \rangle \frown tr_3) \upharpoonright i.ci) = \\ &\quad \#((tr'_0 \frown \langle i.w.a.d \rangle \frown tr'_1) \upharpoonright I.mo\end{aligned}$$

and $tr_i \setminus \{I.r, I.w\} = tr'_i \setminus \{I.r, I.w\} = tr''_i \setminus \{I.r, I.w\}$ for $i = 0 \mathrel{..} 4$. Then, using *Spec*,

$$\begin{aligned}&(tr_0 \frown \langle i.w.a.d \rangle \frown tr_1 \frown tr_2 \frown \langle i.mo.a.d \rangle \frown tr_3) \Downarrow i.ci.\star) = \\ &\quad (tr'_0 \frown \langle i.w.a.d \rangle \frown tr'_1) \Downarrow i.mo\end{aligned}$$

Also from *Spec* :

$$\begin{aligned}&(tr_0 \frown \langle i.w.a.d \rangle \frown tr_1 \frown tr_2 \frown \langle i.mo.a.d \rangle \frown tr_3) \Downarrow i.ci.\star = \\ &\quad (tr_0 \frown \langle i.w.a.d \rangle \frown tr_1 \frown tr_2 \frown \langle i.mo.a.d \rangle \frown tr_3) \Downarrow i.mo\end{aligned}$$

giving a contradiction.

Alternatively, consider the case where the $i.w$ moves to a position after the original position of the $i.r$, i.e. the original trace is of the form

$$tr_0 \frown \langle i.w.a.d \rangle \frown tr_1 \frown tr_2 \frown \langle i.r.a'.d' \rangle \frown tr_3 \frown \langle i.mo.a.d \rangle \frown tr_4$$

which is transformed by M_r to

$$tr'_0 \frown \langle i.w.a.d \rangle \frown tr'_1 \frown \langle i.r.a'.d' \rangle \frown tr'_2 \frown tr'_3 \frown \langle i.mo.a.d \rangle \frown tr'_4$$

which is transformed by M_w to

$$tr''_0 \frown tr''_1 \frown \langle i.r.a'.d' \rangle \frown tr''_2 \frown tr''_3 \frown \langle i.w.a.d \rangle \frown \langle i.mo.a.d \rangle \frown tr''_4$$

where

$$\begin{aligned}&\#((tr''_0 \frown tr''_1 \frown \langle i.r.a'.d' \rangle \frown tr''_2 \frown tr''_3) \upharpoonright i.mo = \\ &\quad \#((tr''_0 \frown tr''_1 \frown \langle i.r.a'.d' \rangle \frown tr''_2 \frown tr''_3) \upharpoonright i.w\end{aligned}$$

and $tr_i \setminus \{I.r, I.w\} = tr'_i \setminus \{I.r, I.w\} = tr''_i \setminus \{I.r, I.w\}$ for $i = 0 \mathrel{..} 4$, and

$$\begin{aligned}&(tr_0 \frown \langle i.w.a.d \rangle \frown tr_1 \frown tr_2 \frown \langle i.r.a'.d' \rangle \frown tr_3 \frown \langle i.mo.a.d \rangle \frown tr_4) \upharpoonright i.w = \\ &\quad (tr''_0 \frown tr''_1 \frown \langle i.r.a'.d' \rangle \frown tr''_2 \frown tr''_3 \frown \langle i.w.a.d \rangle \frown \langle i.mo.a.d \rangle \frown tr''_4) \upharpoonright i.w\end{aligned}$$

Hence,

$$
\begin{array}{ll}
 & \#((tr_0 \frown \langle i.w.a.d \rangle \frown tr_1 \frown tr_2) \upharpoonright i.mo) \\
\leq & \#((tr_0'' \frown tr_1'' \frown \langle i.r.a'.d' \rangle \frown tr_2'' \frown tr_3'') \upharpoonright i.mo) \\
= & \#((tr_0'' \frown tr_1'' \frown \langle i.r.a'.d' \rangle \frown tr_2'' \frown tr_3'') \upharpoonright i.w) \\
< & \#(tr_0 \frown \langle i.w.a.d \rangle \frown tr_1 \frown tr_2) \upharpoonright i.w)
\end{array}
$$

contradicting *Spec*.

9 Conclusion

In this chapter we showed that the CSP process notation can be used to describe a lazy caching algorithm for shared memory. We also showed that the language of CSP traces can be used to characterise the property of sequential consistency. Using the traces model of CSP, we were able to verify that the caching algorithm guaranteed sequential consistency for a suitable shared memory.

The trace specification provided a particularly concise characterisation of sequential consistency, and the process notation made it easy to describe the communicating behaviour of the chosen implementation. The proof that each trace of this implementation is sequentially consistent is lengthy and involved, but contains little or no junk: the complexity of the proof matches the complexity of the problem.

The approach taken here is relatively unsophisticated. We have not shown how the process notation may be refined further, towards software or hardware implementations. Neither have we demonstrated the more powerful models of CSP, which support liveness and timing specifications. But it is our hope that this chapter demonstrates the advantages of a uniform model-based approach: by choosing the simplest adequate notion of observation for each property, complex systems may be verified with a minimum of effort.

References

[ABM93] Y. Afek, G. Brown, and M. Merritt. Lazy caching. *ACM Transactions on Programming Languages and Systems*, 15(1):182–206, 1993.

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

[Lam79] L. Lamport. How to make a multiprocessor that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28, 1979.