Verifying sequentially consistent memory

Eindhoven University of Technology

Department of Mathematics and Computing Science

Verifying Sequentially Consistent Memory

by

E. Brinksma, J.Davies,
R. Gerth, S. Graf,
W. Janssen, B. Jonsson,
S. Katz, G. Lowe,
M. Poel, A. Pnueli,
C. Rump and J. Zwiers.

94/44

ISSN 0926-4515

editors: prof.dr. J.C.M. Baeten
prof.dr. M. Rem

Computing Science Report 94/44
Eindhoven, October 1994

Verifying Sequentially Consistent Memory

Ed Brinksma[1], Twente[2] Jim Davies[3], Reading[4]
Rob Gerth (Editor)[1], Eindhoven[3] Susanne Graf[1], VERIMAG[4]
Wil Janssen[1], Twente[2] Bengt Jonsson[5], Uppsala[7] Shmuel Katz, The Technion[8]
Gavin Lowe[1], Oxford[9] Mannes Poel[1], Twente[2] Amir Pnueli[1], Weizmann[10]
Camilla Rump[11] , Lyngby[12] Job Zwiers[1], Twente[2]

August 1994

[1]Currently working in ESPRIT project P6021: "Building Correct Reactive Systems (REACT)".

[2]Computer Science Department, University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands

[3]Funded by DRA Malvern.

[4]Department of Computer Science, University of Reading, Reading RG6 2AY, England

[3]Department of Computing Science, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands. Email: robg@win.tue.nl

[4]Minipare, Zirst, Rue Lavoisier, F-38330 Montbonnot Saint Martin, France.

[5]Supported in part by the Swedish Board for Technical Development (NUTEK) as part of Esprit BRA project REACT, No. 6021

[7]Department of Computer Systems, Uppsala University, Box 325, 751 05 Uppsala, Sweden

[8]Department of Computer Science, The Technion, Haifa, Israel

[9]Programming Research Group, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, OX1 3QD, England

[10]Department of Applied Mathematics and Computer Science, Weizmann Institute of Science, Rehovot, Israel

[11]Currently working in ESPRIT BRA Project No. 7071: "Provably Correct Systems (ProCoS II)"

[12]Department of Computer Science, Technical University of Denmark, DK-2800 Lyngby, Denmark

Abstract

In distributed shared memory architectures, memory usually obeys weaker constraints than that of ordinary memory in (cache-less) single processor systems. One popular weakening is that of *sequential consistency*. Proving that a memory is sequentialy consistent does not easily fit the standard refinement and verification strategies. This paper takes a sequential consistent memory—the lazy caching protocol—and verifies it using a number of verification approaches. In almost all cases, existing approaches have to be generalized first.

Contents

Chapter 1

Introduction

R. Gerth

In large multiprocessor architectures the design of efficient shared memory systems is important because the latency imposed on the processors when reading or writing should be kept at a minimum. This is usually achieved by interposing a *cache memory* between each processor and the shared memory system. A cache is private to a processor and contains a subset of the memory; hopefully containing most of the locations (variables) that the processor needs to access; i.e., the 'cache-hit' probability should be high. Such caches induce replication of data and hence there is a problem of *cache consistency*: if one processor updates the value at some location, all caches in the system that contain a copy of the location need to be updated. This is often done by marking the location in the caches so that a subsequent access causes the location to be fetched from shared memory again; variations exist, though. Clearly, changing a location and marking that location in other caches must be done as one atomic operation if memory is to behave as expected.

If the multiprocessor architecture is also distributed then such 'write and mark' operations cause unacceptable latencies. For instance, the DASH [LLG+92] and KSR1 [BFKR92] architectures envisage up to 10000 workstations to be connected and to operate on a conceptually shared memory. Atomic write-and-marks produce massive network congestion because at any time there will be many writes in progress.

The approach taken in such distributed shared memory architectures is to relax the constraints on the behavior of a standard shared memory. Many of these relaxations are patterned after Lamport's proposal of *sequential consistency* [Lam79]. In a standard memory the value that is read at a location must be the value that has last been written to that location. A sequentially correct memory satisfies a less stringent requirement: in Lamport's words

> *the result of any execution [of the memory] is the same as if the operations [memory accesses] 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.*

The challenge that sequentially correct memory poses is not so much the verification of yet another complex protocol but rather the fact that sequential consistency does not comfortably fit the patterns of standard refinement strategies (trace inclusion, failure or ready trace equivalence, testing pre order, bisimulation, etc.).

The aim of this paper is to appraise how verifying sequential consistency can be accommodated for in a number of refinement methods. We do this by actually verifying a sequentially consistent memory—the lazy caching protocol of [ABM93]—using a variety of approaches. Although the protocol is proven correct in that paper, the proof is on a semantical level and is not grounded in a verification methodology. This makes the proof quite hard to follow and hard to generalize to more complex protocols such as release consistent or non-blocking memory.

In the next section we explain and define sequential consistency. The lazy caching protocol is introduced in the Section 3. The heart of the paper is formed by Chapters 2 till 8 which contain the various proofs.

In Chapter 2, process algebraic notions such as bisimulation and action transducers are used to derive the caching protocol through a number of refinement steps. Chapter 3 interpretes sequential consistency as a form of interface refinement and gives a direct refinement proof. Abstract interpretation techniques are used in Chapter 4 to reduce the verification problem to one that is amenable to automated verification using model checking techniques. In Chapter 5 CSP process notation and a trace based proof system is used to supply an assertional proof. The proof in Chapter 6 also uses stepwise refinement, but on a more abstract, conceptual level. The refinement proofs are based on partial order based techniques. Chapter 7 develops refinement transducers as a verification methodology and uses this to verify the caching protocol. These transducers can be seen as a syntactic elaboration of

the techniques of Chapter 4. Finally, Chapter 8 uses interleaving set temporal logic (ISTL) and the idea of representative sequences to verify the protocol.

1.1 Sequential consistency

In order to understand Lamport's definition, we first fix the behavior of a standard, 'serial' shared memory. This is done in Figures 1.1 and 1.2.

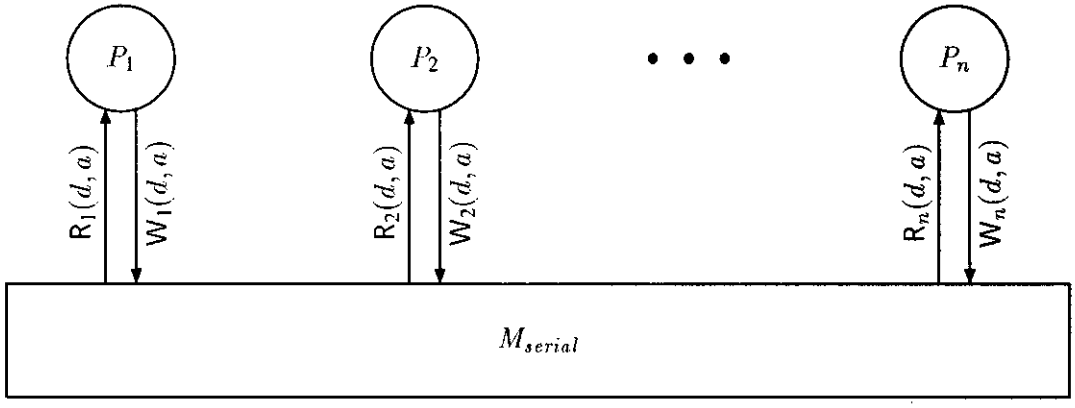


Figure 1.1: Architecture of M_{serial}

The interface of the memory comprises of read ($\mathsf{R}_i(d,a)$) and write ($\mathsf{W}_i(d,a)$) events for each processor P_i. The processors and the memory have to synchronize on these read and write events. The transition system in Figure 1.2 indicates that these are the only external events that M_{serial} participates in and that it has no internal events. A read event $\mathsf{R}_i(d,a)$, issued by P_i, can only occur if the memory holds value d at location a: $Mem[a] = d$. Write events $\mathsf{W}_i(d,a)$ can always occur with the expected result. The *external behavior* of the serial memory, $\mathrm{Beh}(M_{serial})$, is defined as the maximal (hence infinite) sequences of read and write events generated according to the transition system of Figure 1.2. Hence, the memory *serializes* the reads and writes of the processors.

The interface of the serial memory (and the caching protocol) in [ABM93] differs from the one we use. There, a $\mathsf{R}_i(d,a)$-event in either protocol is split into an (input) event $\mathsf{ReadRequest}_i(d,a)$, which is always enabled, and an (output) event $\mathsf{ReadReturn}_i(d,a)$ that behaves as the $\mathsf{R}_i(d,a)$-event. One reason for doing so is their use of I/O automata specifications in which input events must be always enabled. However, that paper also stipulates that a process i must not do otherwise than engage in a Return event after it has issued a Request. This means that the intended interface is synchronous so that not using I/O automata and having simple read and write external events seem to be the conceptually clearer approach.

Two objections that might be levied against this choice of interface are: events cannot overlap because they do not extend in time; and: read events specify the value that is read and thus do not really model read actions. Note that the second objection applies to the [ABM93] interface as well. The answer to both objections is that what is of importance are the points at which the memory system changes state and the values that can be read from memory as a result of these changes. Hence, write events should merely be viewed as the initiators of state changes while read events indicate which

values can be returned. Thus, the precise way in which a process initiates a read or a write is of no importance to the modeling.

We can use this definition of serial memory both to characterize the sequential orders in which the memory accesses of the processors can be executed—any order that corresponds to a behavior of M_{serial}—as well as to characterize the order of operations of each individual processor—since a processor belongs to the environment of M_{serial}, possible orderings are determined by the behaviors of M_{serial} as well.

E	Event	Allowed if	Action
$\surd$	$\mathsf{R}_i(d,a)$	$Mem[a] = d$	
$\surd$	$\mathsf{W}_i(d,a)$		$Mem[a] := d$

initially: $\forall a \;\; Mem[a] = 0$

Figure 1.2: M_{serial}

We rephrase Lamport's proposal of correct behavior of sequentially consistent memory (SCM) thus

> *any external behavior, σ, [of the SCM] corresponds with an external behavior, τ, of M_{serial} so that the order in which the operations of each individual processor appear in σ coincides with order in which they appear in τ.*

For instance, the graph below depicts a possible prefix of a behavior of an SCM and a corresponding serial behavior:

SCM		$\mathsf{W}_1(1,x)$	$\mathsf{W}_2(2,y)$	$\mathsf{R}_3(2,y)$	$\mathsf{R}_3(0,x)$	$\mathsf{R}_3(1,x)$
	P_1:	$\mathsf{W}_1(1,x)$				
	P_2:		$\mathsf{W}_2(2,y)$			
	P_3:			$\mathsf{R}_3(2,y)$	$\mathsf{R}_3(0,x)$	$\mathsf{R}_3(1,x)$
M_{serial}		$\mathsf{W}_2(2,y)$	$\mathsf{R}_3(2,y)$	$\mathsf{R}_3(0,x)$	$\mathsf{W}_1(1,x)$	$\mathsf{R}_3(1,x)$

Time flows from left to right. In particular notice that, although P_1 sets x to 1 before P_3 accesses that location, the first read of P_3 retrieves x's initial value 0. The effect of writes are thus seen to propagate slowly through the system. This is typical of sequentially consistent memory. Also notice that this SCM behavior is not possible for serial memory.

For completeness sake, we mention that the following behavior of the individual processes cannot be accommodated for by SCM:

P_1:	$\mathsf{W}_1(1,x)$					
P_2:		$\mathsf{W}_2(2,x)$				
P_3:			$\mathsf{R}_3(1,x)$		$\mathsf{R}_3(2,x)$	
P_4:				$\mathsf{R}_4(2,x)$		$\mathsf{R}_4(1,x)$

The problem is that P_3 and P_4 'observe' the writes of P_1 and P_2 in different order.

Sequential consistency has been the canonical distributed memory model for a long time. In practice, however, different, still weaker memory models tend to be implemented as the synchronization

overhead of SCM is still too large. For instance, the *processor consistency* model would allow the above behavior at the processors. See [Mos93] for an overview of distributed memory models.

A formal definition

Let $\cdot \restriction i$ denote the operation on behaviors of removing the events that do not originate from process P_i or that are not external. Then we have

A memory M is sequentially consistent w.r.t. M_{serial}, M *s.c.* M_{serial}, iff

$$\forall \sigma \in \text{Beh(M)}\ \exists \tau \in \text{Beh}(M_{serial})\ \forall i = 1 \ldots n \quad \sigma \restriction i = \tau \restriction i$$

This memory model enjoys an important advantage over its 'competitors': for reasoning about a program we may ignore the fact that the program runs on a sequential consistent memory and can assume instead that it runs on a standard serial memory. I.e., verification techniques need not be adapted and the programming model is that of standard shared memory.

We stress that this is the case only if the program has no means of communication, either implicitly or explicitly, other than through the memory. If a program can send messages or can sense the time at which reads and writes occur, then differences between sequential consistent and serial memory can be detected; see, e.g., [ABM93].

1.2 The lazy caching protocol

In [ABM93] a sequential correct memory that is not serial was proposed: the lazy caching protocol. We use a slightly adapted version of this protocol.

The architecture of M_{distr} is depicted in Figure 1.3; the transition system in Figure 1.4. The protocol is thus geared towards a bus based architecture. Here, too, the interface of the memory comprises of the read and write events of the processors. M_{distr}, however, interposes caches C_i between the shared memory Mem and the processes P_i. Each cache C_i contains a part of the memory Mem and has two queues associated with it: an out-queue Out_i in which P_i's write requests are buffered and an in-queue In_i in which the pending cache updates are stored. These queues model the asynchronous behavior of write events in a sequential consistent memory. The gray arrows indicate the information flows from the out queues to the in queues and to Mem.

A write event $\mathsf{W}_i(d, a)$ does not have immediate effect. Instead, a request (d, a) is placed in Out_i. When the write request is taken out of the queue, by an internal memory-write event $\mathsf{MW}_i(d, a)$, the memory is updated and a cache update request (d, a) is placed in every in-queue. This cache update is eventually removed from the top of some queue In_j by an internal cache update event $\mathsf{CU}_j(d, a)$ as a result of which cache memory C_j gets updated. Cache misses are modeled by internal cache invalidate events: CI_i can arbitrarily remove locations from cache C_i. Caches are filled both as the delayed result of write events as well as through internal memory-read events, $\mathsf{MR}_i(d, a)$. The latter events intend to model the effect of a cache-miss: in that case the read event suspends until the location is copied from memory.

A read event $\mathsf{R}_i(d, a)$, predictably, stalls until a copy of location a is present in C_i but also until the copy contains a 'correct' value in the following sense: sequential consistency implies that a processor P_i reads the value at a location a that was most recently written by P_i unless some other processor updated a in the mean time. Hence, a read event $\mathsf{R}_i(d, a)$ cannot occur unless all pending writes in

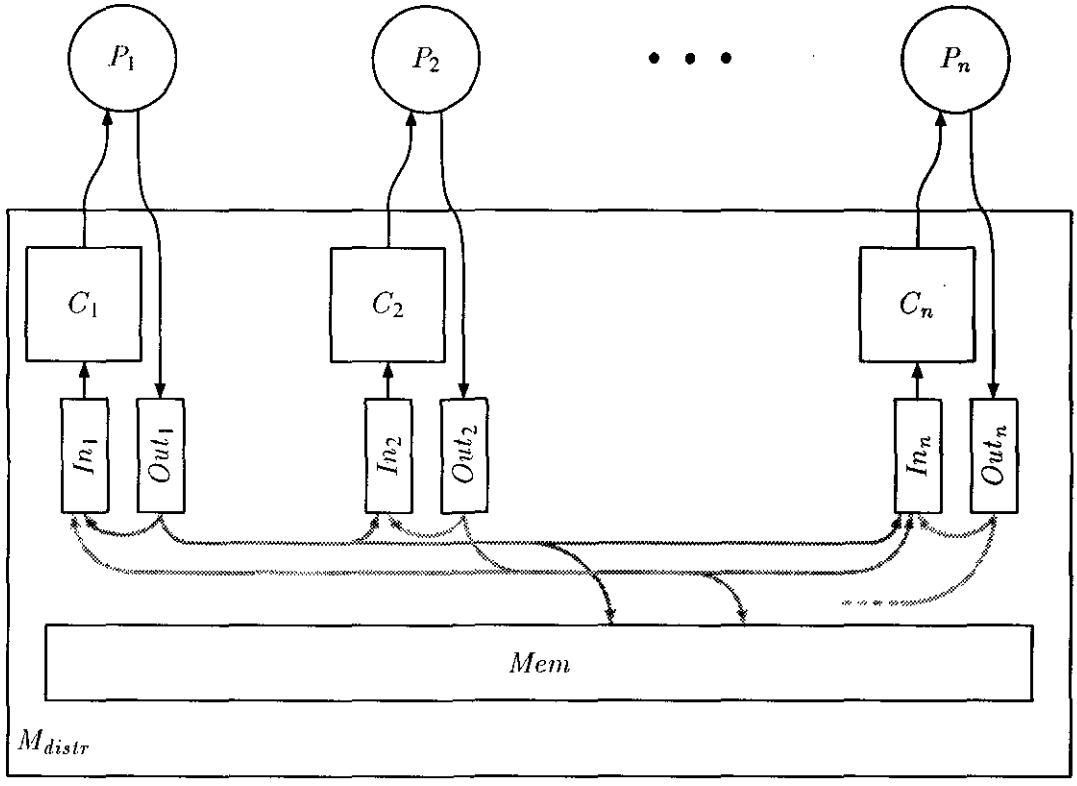


Figure 1.3: Architecture of M_{distr}

Out_i are processed as well as the cache update requests from In_i that correspond to writes of P_i. For this reason, such cache update request are marked (with a $*$).

The transition system in Figure 1.4 makes all this precise.

In this transition system caches are modeled as partial functions from the set of locations to the set of values. Cache update (CU) actions produce 'variant functions': $update(C_i, d, a)$ stands for the function f that coincides with C_i except 'at' a where $f(a) = d$. Cache invalidate (CI) actions yield 'restrictions' of functions: $restrict(C_i)$ stands for any function whose domain is included in that of C_i and which coincides with C_i on its domain.

For M_{distr} there is a distinction between the external behavior, Beh(M_{distr}) and the *internal behavior*, IBeh(M_{distr}) that comprises the maximal sequences of internal and external events that M_{distr} can generate (obviously we have Beh(M_{serial}) = IBeh(M_{serial})). Observe that for $s \in$ IBeh(M_{distr}), $s \restriction i$ denotes the subsequence of *external* read and write-events of P_i in s.

E	Event	Allowed if	Action
$\surd$	$\mathsf{R}_i(d,a)$	$C_i(a) = d \wedge Out_i = \{\}$ $\wedge$ no $*$-ed entries in In_i	
$\surd$	$\mathsf{W}_i(d,a)$		$Out_i := append(Out_i,(d,a))$
	$\mathsf{MW}_i(d,a)$	$head(Out_i) = (d,a)$	$Mem[a] := d$; $Out_i := tail(Out_i)$; $(\forall k \neq i :: In_k := append(In_k,(d,a)))$; $In_i := append(In_i,(d,a,*))$
	$\mathsf{MR}_i(d,a)$	$Mem[a] = d$	$In_i := append(In_i,(d,a))$
	$\mathsf{CU}_i(d,a)$	$head(In_i)$ 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: $\forall a \;\; Mem[a] = 0$ $\wedge \forall i = 1 \ldots n \;\; C_i \subset Mem \wedge In_i = \{\} \wedge Out_i = \{\}$

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

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

Figure 1.4: M_{distr}

Chapter 2

Cache Consistency by Design

E. Brinksma

2.1 Introduction

In this paper we present proof for the sequential consistency the lazy caching protocol of [ABM93] as formulated in [Ger95]. The proof will follow a strategy of *stepwise refinement*, developing the distributed caching memory in five transformation steps from a specification of the serial memory, whilst preserving the sequential consistency in each step. Thus our proof, in fact, presents a rationalized design of the distributed caching memory.

We will carry out our proof using a simple process-algebraic formalism for the specification of the various design stages. Process algebraic techniques [Hoa85, Mil89, BW90] are by their nature suitable for transformational proofs as they concentrate on laws that equate and/or compare different behaviour expressions. Such laws are natural candidates for design transformations. Our proof will not follow a strictly algebraic exposition, however. For some transformations we will show the correctness using semantic arguments directly, instead of pure syntactic derivations from basic laws. We will also employ the less standard feature of *action transducers* to relate behaviours in two of our design steps.

The structure of the rest of this paper is as follows.

- *section 2* introduces the process-algebraic formalism that we use;
- *section 3* explains about the use of action transducers, and introduces the notion of *queue-like* action transducers in particular;
- *section 4* gives a transformation style proof of the *weak* sequential consistency of the distributed cache memory. This property takes into account only finite sequences of the observable actions of a system;
- *section 5* improves the result to *strong* sequential consistency, also taking possibly infinite behaviour into account;
- *section 6* discusses the results that have been obtained and draws some conclusions.

2.2 A simple process-algebraic formalism

We will work with a simple process algebraic formalism to specify the different design stages in the course of our proof. Throughout this paper we will assume a working knowledge of process algebras. For a good introduction to the literature of process algebras the reader is referred to [Hoa85, Mil89, BW90]. Below, we give a short summary of those features that are essential for the development of our proof.

The syntax and semantics of our formalism are given in tables 2.1 and 2.2, respectively. The tables assume a given set of observable actions Act and an additional *silent* or *hidden action* τ. The *behaviour expressions* defined by the syntax table define the behaviour of systems in terms of labeled transition systems, where the transitions are labeled by elements in $Act \cup \{\tau\}$. These operational models can be derived for each behaviour expression with the aid of the inference rules given in table 2.2. For a detailed account of this so-called *structured operational semantics* or *SOS* style of definition, we refer to [Mil89, Plo81].

The behaviour expressions are defined in an environment of *process definitions* of the form

$$\{p \Leftarrow B_p \mid p \in \mathcal{P}\}$$

Name	Syntax B	Label set $L(B)$
inaction	$\mathbf{0}$	$\emptyset$
action-prefix	$\mu.B$ $(\mu \in Act)$	$\{\mu\} \cup L(B)$
	$\tau.B$	$L(B)$
choice	$B_1 + B_2$	$L(B_1) \cup L(B_2)$
composition	$B_1 \|\|_G B_2$	$L(B_1) \cup L(B_2)$
	$(G \subseteq Act)$	
hiding	B/G	$L(B) - G$
	$(G \subseteq Act)$	
renaming	$B[H]$	$H(L(B))$
	$(H : Act \rightarrow Act)$	
instantiation	p	L_p
	$(p \Leftarrow B_p, L(B_p) \subseteq L_p)$	

Table 2.1: syntax of a simple process algebraic language

Name	Axioms and inference rules
inaction	none
action-prefix	$\mu.B \xrightarrow{\mu} B$
	$(\mu \in Act \cup \{\tau\})$
choice	$B_1 \xrightarrow{\mu} B_1' \vdash B_1 + B_2 \xrightarrow{\mu} B_1'$
	$B_2 \xrightarrow{\mu} B_2' \vdash B_1 + B_2 \xrightarrow{\mu} B_2'$
composition	$B_1 \xrightarrow{\mu} B_1' \vdash_{\mu \notin G} B_1 \|\|_G B_2 \xrightarrow{\mu} B_1' \|\|_G B_2$
	$B_2 \xrightarrow{\mu} B_2' \vdash_{\mu \notin G} B_1 \|\|_G B_2 \xrightarrow{\mu} B_1 \|\|_G B_2'$
	$B_1 \xrightarrow{\mu} B_1', B_2 \xrightarrow{\mu} B_2' \vdash_{\mu \in G} B_1 \|\|_G B_2 \xrightarrow{\mu} B_1' \|\|_G B_2'$
hiding	$B \xrightarrow{\mu} B' \vdash_{\mu \notin G} B/G \xrightarrow{\mu} B'/G$
	$B \xrightarrow{\mu} B' \vdash_{\mu \in G} B/G \xrightarrow{\tau} B'/G$
renaming	$B \xrightarrow{\mu} B' \vdash B[H] \xrightarrow{H(\mu)} B'[H]$
instantiation	$B_p \xrightarrow{\mu} B' \vdash_{p \Leftarrow B_p} p \xrightarrow{\mu} B'$

Table 2.2: structured operational semantics

where $\mathcal{P}$ is a set of process identifiers p with action label type L_p, and B_p is a behaviour expression with action label set $L(B_p) \subseteq L_p$. We will use the the notation $p \Leftarrow B_p$ to denote the statement that '$p \Leftarrow B_p$ is an element of the environment of process definitions'. The environment may contain mutually recursive process definitions. The label types L_p are usually left undefined, and are implicitly understood to be the smallest label types satisfying the static constraints of table 2.1. In the application part of the paper we will provide concrete instances of the set of actions Act en the process definition environment.

In addition to the process algebraic combinators introduced by table 2.1 we will use generalizations

(1)	$B_1 \|\|_G B_2 = B_2 \|\|_G B_1$	
(2)	$B_1 \|\|_G (B_2 \|\|_G B_3) = (B_1 \|\|_G B_2) \|\|_G B_3$	
(3)	$B_1 \|\|_* (B_2 \|\|_* B_3) = (B_1 \|\|_* B_2) \|\|_* B_3$	where $B_1 \|\|_* B_2 =_{df} B_1 \|\|_{L(B_1) \cap L(B_2)} B_2$
(4)	$(B_1 \|\|_G B_2)/A = B_1/A \|\|_G B_2/A$	if $A \cap G = \emptyset$
(5)	$(B_1 \|\|_G B_2)[H] = B_1[H] \|\|_G B_2[H]$	if $H \lceil G = id_G$ and $H^{-1}(G) = G$

Table 2.3: Some transformation laws

for the choice and composition operators. If $\mathcal{B}$ denotes a *finite* set of behaviour expressions then $\sum \mathcal{B}$ and $\prod^G \mathcal{B}$ denote the repeated application of '+' and '$||_G$', respectively, to the elements of $\mathcal{B}$. E.g. if $\mathcal{B} = \{B_1, \ldots, B_n\}$ then

$$\begin{aligned} \sum \mathcal{B} &= B_1 + \ldots + B_n \\ \prod^G \mathcal{B} &= B_1 \|_G \ldots \|_G B_n \end{aligned}$$

This notation exploits the commutativity and associativity of the combinators '+' and '$||_G$' that will be justified below. If $\mathcal{B} = \{B_i | i \in \mathcal{I}\}$ we often write $\sum_{i \in \mathcal{I}} B_i$ and $\prod^G_{i \in \mathcal{I}} B_i$.

The standard identity over the behaviour expressions (and labeled transition systems) will be given by the *strong bisimulation equivalence* relation, which is a congruence with respect to all the given combinators. We recall the definition.

Let *BE* denote the set of behaviour expressions over given sets *Act* and $\mathcal{P}$ of actions and process identifiers, respectively.

Definition 2.2.1 *A relation $R \subseteq BE \times BE$ is a* strong simulation *relation iff for all $\langle B_1, B_2 \rangle \in R$ and for all $\mu \in \text{Act} \cup \{\tau\}$ $\exists B_1'\ B_1 \xrightarrow{\mu} B_1'$ implies $\exists B_2'\ B_2 \xrightarrow{\mu} B_2'$ and $\langle B_1', B_2' \rangle \in R$.*

A relation $R \subseteq BE \times BE$ is a strong bisimulation *relation iff both R and its inverse R^{-1} are strong simulation relations.*

Two behaviour expressions B_1, B_2 are strong bisimulation equivalent, notation $B_1 \sim B_2$, iff there exists a strong bisimulation relation R with $\langle B_1, B_2 \rangle \in R$. □

The following fact is a standard result in the process algebraic literature (cf. [Mil89])

Fact 2.2.2 *The relation $\sim$ is a congruence with respect to all the combinators introduced in table 2.1 and satisfies the laws listed in table 2.3.* □

We recall the following (standard) notations. Action names are variables over $Act \cup \{\tau\}$ and σ denotes a string of actions $a_1 \ldots a_n$.

$$\begin{aligned} B \xrightarrow{\sigma} B' &\Leftrightarrow_{df} \exists B_0, \ldots, B_n\ B \equiv B_0 \xrightarrow{a_1} B_1 \wedge \ldots \wedge B_{n-1} \xrightarrow{a_n} B_n \equiv B' \\ B \stackrel{\varepsilon}{\Rightarrow} B' &\Leftrightarrow_{df} \exists n\ B \xrightarrow{\tau^n} B' \\ B \stackrel{a}{\Rightarrow} B' &\Leftrightarrow_{df} \exists B_1, B_2\ B \stackrel{\varepsilon}{\Rightarrow} B_1 \wedge B_1 \xrightarrow{a} B_2 \wedge B_2 \stackrel{\varepsilon}{\Rightarrow} B' \\ B \stackrel{\sigma}{\Rightarrow} B' &\Leftrightarrow_{df} \exists B_0, \ldots, B_n\ B \equiv B_0 \stackrel{a_1}{\Rightarrow} B_1 \wedge \ldots \wedge B_{n-1} \stackrel{a_n}{\Rightarrow} B_n \equiv B' \\ Der(B) &=_{df} \{B' \mid \exists \sigma \in Act^*\ B \stackrel{\sigma}{\Rightarrow} B'\} \end{aligned}$$

We will also need a less strict relation than $\sim$.

Definition 2.2.3 *A relation $R \subseteq BE \times BE$ is a* weak simulation *relation iff for all $\langle B_1, B_2 \rangle \in R$ and for all $\alpha \in \mathrm{Act} \cup \{\epsilon\}$ $\exists B_1'\ B_1 \stackrel{\alpha}{\Rightarrow} B_1'$ implies $\exists B_2'\ B_2 \stackrel{\alpha}{\Rightarrow} B_2'$ and $\langle B_1', B_2' \rangle \in R$.*

A relation $R \subseteq BE \times BE$ is a weak bisimulation *relation iff both R and its inverse R^{-1} are weak simulation relations.*

Two behaviour expressions B_1, B_2 are weak bisimulation equivalent, notation $B_1 \approx B_2$, iff there exists a weak bisimulation relation R with $\langle B_1, B_2 \rangle \in R$. □

Again we have a standard result (cf. [Mil89]).

Fact 2.2.4 *The relation $\approx$ is a congruence with respect to all the combinators introduced in table 2.1 except for the choice combinator '+' (and its generalization $\sum$) and $\sim \subseteq \approx$ (i.e. $\approx$ satisfies all laws of $\sim$).* □

Finally, let us define $Traces(B) =_{df} \{\sigma \in Act^* \mid \exists B'\ B \stackrel{\sigma}{\Rightarrow} B'\}$, then we have the following well-known definition and results (cf. [Hoa85, vG93]).

Definition 2.2.5 *Two behaviour expressions B_1, B_2 are trace equivalent, notation $B_1 \approx_{trace} B_2$, iff* $\mathrm{Traces}(B_1) = \mathrm{Traces}(B_2)$. □

Fact 2.2.6 *The relation $\approx_{trace}$ is a congruence with respect to all the combinators introduced in table 2.1 and $\sim \subseteq \approx \subseteq \approx_{trace}$.* □

Fact 2.2.7 *Let $B_1||_*B_2$ be defined as in Table 2.3.*

$$Traces(B_1||_*B_2) = \{\sigma \in (L(B_1) \cup L(B_2))^* \mid \sigma\lceil L(B_1) \in Traces(B_1), \sigma\lceil L(B_2) \in Traces(B_2)\}$$ □

2.3 Queue-like action-transducers

Action-transducers are the operational counterpart of *contexts*, i.e. behaviour expressions with an open place or *hole* in them. Such open places, often denoted by the symbol '[ ]', can be regarded as variables that can be replaced with actual behaviour expressions to obtain instantiations of a given context. For example, the context $C[\ ] =_{df} a.\mathbf{0} + [\ ]$ can be instantiated by the expression $b.c.\mathbf{0}$, yielding $C[b.c.\mathbf{0}] = a.\mathbf{0} + b.c.\mathbf{0}$.

Whereas we can use behaviour expressions to define *states* with *transitions* between them (e.g. as defined by table 2.2), contexts define *action transducers* with *transductions* between them. Such transductions will be denoted by doubly decorated arrows, as in

$$T \stackrel{a}{\underset{b}{\rightarrow}} T'$$

which represents the transduction of action b into action a as action-transducer (state) T changes into T'. Informally, this should be understood as follows: whenever a behaviour B at the place of the formal parameter '[ ]' produces an a-action transforming into B', $T[B]$ will produce a b-action as its result and transform into $T'[B']$.

Example 2.3.1

$$a.B||_{\{a\}}[\,][a/b] \xrightarrow[b]{a} B||_{\{a\}}[\,][a/b]$$

where a/b denotes the obvious renaming function replacing b by a. □

The transduction $T \xrightarrow[b]{a} T'$ thus corresponds to the operational semantic rule

$$B \xrightarrow{b} B' \vdash T[B] \xrightarrow{a} T'[B']$$

Additionally, we also allow transducers to produce actions 'spontaneously' to cater for contexts like $a.[\,]$, which can produce an a-action without consuming an action of an instantiating behaviour. This will be denoted by transduction of the form $T \xrightarrow[0]{a} T'$, corresponding to the operational semantic rule

$$\vdash T[B] \xrightarrow{a} T'[B]$$

Example 2.3.2

$$a.B||_{\{a\}}a.[\,] \xrightarrow[0]{a} B||_{\{a\}}[\,]$$

□

In this paper we will not give a complete formal introduction to the concept of contexts as action-transducers. For this the reader is referred to [Lar90, Bri92]. Here, it will suffice to define systems of action-transducers by explicitly giving sets of transducer states and transductions between them.

A last step before defining transducer systems is the extension of the transduction notation to a suitable 'double-arrow' notation. Let $\sigma, \sigma' \in (Act \cup \{\tau, 0\})^*$. We write $\sigma \lhd \sigma'$ iff σ can be obtained from σ' by erasing any number of τ- or 0-occurrences in it. We define

$$T \xrightarrow[b_1 \ldots b_n]{a_1 \ldots a_n} T' \quad \Leftrightarrow_{df} \quad \exists T_0, \ldots, T_n \; T \equiv T_0 \xrightarrow[b_1]{a_1} T_1 \wedge \ldots \wedge T_{n-1} \xrightarrow[b_n]{a_n} T_n \equiv T'$$

$$T \xRightarrow[\sigma_2]{\sigma_1} T' \quad \Leftrightarrow_{df} \quad \exists \sigma_1', \sigma_2' \; T \xrightarrow[\sigma_2']{\sigma_1'} T' \; \wedge \; \sigma_1 \lhd \sigma_1' \; \wedge \; \sigma_2 \lhd \sigma_2'$$

We now proceed with the definition of the special kind of action-transducer systems that we need for our application, viz. the queue-like families of action transducers.

Definition 2.3.3 *Let $Q \subseteq Act$. A family of action-transducers $\mathcal{T}_Q = \{T^\sigma \mid \sigma \in Q^*\}$ is* queue-like *iff its tranductions are of the form:*

1. $\forall q \in Q, \sigma \in Q^* \;\; T^\sigma \xrightarrow[0]{q} T^{\sigma q}$
2. $\forall q \in Q, \sigma \in Q^* \;\; T^{q\sigma} \xrightarrow[q]{\tau} T^\sigma$
3. *for 0 or more* $\sigma \in Q^*, a \in (Act - Q) \; T^\sigma \xrightarrow[a]{a} T^\sigma$. □

Definition 2.3.4 *Let $\mathcal{T}_Q = \{T^\sigma \mid \sigma \in Q^*\}$ be a queue-like family of action-transducers. For each $A \subseteq Q$ we define the set $D_A \subseteq Act$ by*

$$D_A = \{a \in Act \mid T^\sigma \xrightarrow[a]{a} T^\sigma \text{ iff } \sigma\lceil A = \varepsilon\}$$

□

Definition 2.3.5 *Let $\mathcal{T}_Q = \{T^\sigma \mid \sigma \in Q^*\}$ be a queue-like family of action-transducers. We say that $\mathcal{T}_Q$ preserves $A \subseteq Act$ iff*

$$\forall \rho, \sigma \in \mathrm{Act}^*, \upsilon \in Q^* \; T^\epsilon \overset{\rho}{\underset{\sigma}{\Longrightarrow}} T^\upsilon \text{ implies } \rho\lceil A = \sigma\upsilon\lceil A$$

□

The following two lemmata express invariants of the *observable trace* transductions that are induced by families of queue-like action transducers. Of course, a string over any subset A of the set of actions Q that are subject to queing will be preserved. The lemmata indicate that A can always be extended with D_A, the set of actions that can be passed directly 'through' the context when no element of A is being queued. The intuition behind this result is that actions in D_A could therefore never 'overtake' actions in A, or vice versa, and thus upset the ordering of elements in the string.

Lemma 2.3.6 *Let $\mathcal{T}_Q = \{T^\sigma \mid \sigma \in Q^*\}$ be a queue-like family of action-transducers. For each $A \subseteq Q$ $\mathcal{T}_Q$ preserves $A \cup D_A$.*

Proof. Let $T^\epsilon \overset{\rho}{\underset{\sigma}{\Longrightarrow}} T^\upsilon$. We carry out the proof by induction on $|\rho| + |\sigma|$. The basic case that $|\rho| + |\sigma| = 0$ follows trivially as it implies that $\rho = \sigma = \upsilon = \varepsilon$.

Let us therefore suppose that the lemma holds for all $n < |\rho| + |\sigma|$. We can factorize $T^\epsilon \overset{\rho}{\underset{\sigma}{\Longrightarrow}} T^\upsilon$ into $T^\epsilon \overset{\rho_1}{\underset{\sigma_1}{\Longrightarrow}} T^{\upsilon_1} \xrightarrow[b]{a} T^\upsilon$ for some suitably chosen $\rho_1, \sigma_1, \upsilon_1, a$, and b. Since, by the definition of queue-like transductions, not both a and $b \in \{\tau, 0\}$ we can deduce that $|\rho_1| + |\sigma_1| < |\rho| + |\sigma|$ and therefore that $\rho_1\lceil(A \cup D_A) = \sigma_1\upsilon_1\lceil(A \cup D_A)$.

We now proceed by case analysis on the nature of the transduction $T^{\upsilon_1} \xrightarrow[b]{a} T^\upsilon$ as given in definition 2.3.3.

1. $T^{\upsilon_1} \xrightarrow[b]{a} T^\upsilon = T^{\upsilon_1} \xrightarrow[0]{q} T^{\upsilon_1 q}$.
   Then $\rho\lceil(A \cup D_A) = \rho_1 q\lceil(A \cup D_A) = \sigma_1\upsilon_1 q\lceil(A \cup D_A) = \sigma\upsilon\lceil(A \cup D_A)$.

2. $T^{\upsilon_1} \xrightarrow[b]{a} T^\upsilon = T^{q\upsilon} \xrightarrow[q]{\tau} T^\upsilon$.
   Then $\rho\lceil(A \cup D_A) = \rho_1\lceil(A \cup D_A) = \sigma_1\upsilon_1\lceil(A \cup D_A) = \sigma_1 q\upsilon\lceil(A \cup D_A)$
   $= \sigma\upsilon\lceil(A \cup D_A)$.

3. $T^{\upsilon_1} \xrightarrow[b]{a} T^\upsilon = T^\upsilon \xrightarrow[a]{a} T^\upsilon$.
   This is only possible if $a \notin Q$ and thus $a \notin A$. Assume that also $a \notin D_A$. In that case it follows that
   $\rho\lceil(A \cup D_A) = \rho_1 a\lceil(A \cup D_A) = \sigma_1\upsilon_1 a\lceil(A \cup D_A) = \sigma_1 a\upsilon 1\lceil(A \cup D_A)$
   $= \sigma\upsilon\lceil(A \cup D_A)$.
   In the other case that $a \in D_A$ it follows that $\upsilon_1 \cap A = \upsilon \cap A = \emptyset$.
   Therefore, we get
   $\rho\lceil(A \cup D_A) = \rho_1 a\lceil(A \cup D_A) = \sigma_1\upsilon_1 a\lceil(A \cup D_A) = \sigma_1 a\lceil(A \cup D_A)$
   $= \sigma\lceil(A \cup D_A) = \sigma\upsilon\lceil(A \cup D_A)$. □

□

Lemma 2.3.7 ((preservation lemma)) *Let $\mathcal{T}_Q = \{T^\sigma \mid \sigma \in Q^*\}$ be a queue-like family of action-transducers. Let B continuously allow all actions in Q, i.e. for all $B' \in Der(B)$ and all $q \in Q$ $\exists B''\ B' \xrightarrow{q} B''$. Then for all $A \subseteq Q$ we have*

$$\forall \sigma \in Traces(T^\epsilon[B])\ \exists \sigma' \in Traces(B)\ with\ \sigma\lceil(A \cup D_A) = \sigma'\lceil(A \cup D_A)$$

Proof. Assume that $T^\epsilon[B] \stackrel{\sigma}{\Rightarrow} T^v[B']$. Because B continuously allows all actions in Q, we have in particular that $B' \stackrel{v}{\Rightarrow} B''$ and therefore $T^v[B'] \stackrel{\epsilon}{\Rightarrow} T^\epsilon[B'']$. It follows that there exists a σ' with $T^\epsilon \stackrel{\sigma}{\underset{\sigma'}{\Longrightarrow}} T^\epsilon$ and $\sigma' \in Traces(B)$. The required preservation result now follows from an application of the previous lemma. □
□

2.4 Deriving the lazy caching memory

We start our derivation of the lazy caching protocol with a specification of the serial memory, which is given by the process $Mem(\overline{x})$ defined by (2.1) below. The contents of the memory is represented by the process parameter $\overline{x}$, which is a vector of elements in the data domain D indexed by the set A of memory addresses. For all $a \in A$ x_a denotes the a^{th} element of $\overline{x}$. The set $I = \{1, \ldots, n\}$ indexes the number of user interaction points of the memory, i.e. the number of locations where local read and write actions can be performed.

$$\begin{aligned} Mem_{ser}(\overline{x}) \Leftarrow & \sum_{\substack{i \in I \\ a \in A, d \in D}} W_i(d,a).Mem_{ser}(\overline{x}\{d/x_a\}) \\ + & \sum_{\substack{i \in I \\ a \in A}} R_i(x_a,a).Mem_{ser}(\overline{x}) \end{aligned} \tag{2.1}$$

Here, $W_i(d,a)$ represents the action of writing datum d in memory address a, and $R_i(d,a)$ reading datum d from memory location a. It will be useful to define the sets

- $\mathcal{W}_i =_{df} \{W_i(d,a) \mid d \in D, a \in A\}$ and $\mathcal{W} =_{df} \bigcup_{i \in I} \mathcal{W}_i$
- $\mathcal{R}_i =_{df} \{R_i(d,a) \mid d \in D, a \in A\}$ and $\mathcal{R} =_{df} \bigcup_{i \in I} \mathcal{R}_i$
- $\mathcal{L}_i =_{df} \mathcal{W}_i \cup \mathcal{R}_i$ and $\mathcal{L} =_{df} \bigcup_{i \in I} \mathcal{L}_i$

We can now formulate the correctness criterion in our setting as

Definition 2.4.1 *Let B_1 and B_2 be behaviour expressions with $L(B_i) \subseteq \mathcal{L}$. A behaviour B_1 is* weak sequential consistent *with B_2 iff*

$$\forall \sigma \in \text{Traces}(B_1)\ \exists \sigma' \in \text{Traces}(B_2)\ \textit{such that}\ \forall i \in I\ \sigma\lceil\mathcal{L}_i = \sigma'\lceil\mathcal{L}_i$$

□

This is a weaker requirement than the originally given definition of sequential consistency, which is concerned with maximal, and therefore possibly infinite traces (which are not in $Traces(B_1)$). We will first complete the design for this version of sequential consistency and will revisit the question of infinite traces in section 2.5.

2.4.1 Distributing the memory

Our first step in the design is to create a local copy of the memory for every user. The specification of the local memory for user $j \in I$ is given by the process definition of $Locmem_j(\overline{x})$ at (2.2) below. Note that $Locmem_j(\overline{x})$ still interacts in all actions in $\mathcal{W}$, but accepts only local read actions, i.e. those in $\mathcal{R}_j$.

$$Locmem_j(\overline{x}) \Leftarrow \sum_{\substack{i \in I \\ a \in A, d \in D}} W_i(d,a).Locmem_j(\overline{x}\{d/x_a\}) + \sum_{a \in A} R_j(x_a,a).Locmem_j(\overline{x}) \tag{2.2}$$

Our first refinement is now given by the process definition $Refinement_1$ in (2.3).

$$Refinement_1 \Leftarrow \prod_{j \in I}^{\mathcal{W}} Locmem_j(\overline{0}) \tag{2.3}$$

The correctness of this step is certified by the following lemma.

Lemma 2.4.2

$$Mem_{ser}(\overline{0}) \sim Refinement_1$$

Proof. The relation defined by

$$\{\langle Mem_{ser}(\overline{x}), \prod_{j \in I}^{\mathcal{W}} Locmem_j(\overline{x})\rangle \mid \overline{x} \in D^A\}$$

is a strong bisimulation. This follows directly as for all writing actions we have

$$\begin{array}{ll} & Mem_{ser}(\overline{x}) \xrightarrow{W_i(d,a)} Mem_{ser}(\overline{x}\{d/x_a\}) \\ \Leftrightarrow & \forall j \in I\ Locmem_j(\overline{x}) \xrightarrow{W_i(d,a)} Locmem_j(\overline{x}\{d/x_a\}) \\ \Leftrightarrow & \prod_{j \in I}^{\mathcal{W}} Locmem_j(\overline{x}) \xrightarrow{W_i(d,a)} \prod_{j \in I}^{\mathcal{W}} Locmem_j(\overline{x}\{d/x_a\}) \end{array}$$

and for all reading actions

$$\begin{array}{ll} & Mem_{ser}(\overline{x}) \xrightarrow{R_i(x_a,a)} Mem_{ser}(\overline{x}) \\ \Leftrightarrow & Locmem_i(\overline{x}) \xrightarrow{R_i(x_a,a)} Locmem_i(\overline{x}) \\ \Leftrightarrow & \prod_{j \in I}^{\mathcal{W}} Locmem_j(\overline{x}) \xrightarrow{R_i(x_a,a)} \prod_{j \in I}^{\mathcal{W}} Locmem_j(\overline{x}) \end{array}$$

□
□

Corollary 2.4.3 *$Refinement_1$ is weak sequential consistent with $Mem_{ser}(\overline{0})$*

Proof. Follows directly from $\sim \subseteq \approx_{trace}$ (fact 2.2.6). □
□

2.4.2 Introducing local caching

In the next step of our design we introduce a local cache that the user communicates with and that is updated by the local memory. Because of its direct interface with the user this cache has a more elaborate set of interactions that the chaches that we will ultimately design. The behaviour of the cache at interaction point $j \in I$ is given by the process definition $Cache_j(\overline{x})$ in (2.4) below. In addition to the (local) memory the caches have *update* actions $U_j(d,a)$. For convenience we define $\mathcal{U}_i =_{df} \{U_i(d,a) \mid d \in D, a \in A\}$ and $\mathcal{U} =_{df} \bigcup_{i \in I} \mathcal{U}_i$.

$$
\begin{aligned}
Cache_j(\overline{x}) \Leftarrow \quad & \sum_{\substack{i \in I \\ a \in A, d \in D}} W_i(d,a).Cache_j(\overline{x}\{d/x_a\}) \\
+ \quad & \sum_{a \in A, d \in D} U_j(d,a).Cache_j(\overline{x}\{d/x_a\}) \\
+ \quad & \sum_{a \downarrow \overline{x}} R_j(x_a,a).Cache_j(\overline{x}) \\
+ \quad & \sum_{\overline{y} \in r(\overline{x})} \tau.Cache_j(\overline{y})
\end{aligned}
\tag{2.4}
$$

Note that the local caches synchronize on all actions in $\mathcal{W}$, but accept only local read and update actions, i.e. only actions in $\mathcal{R}_j \cup \mathcal{U}_j$. Cache invalidation is modelled by allowing the elements of the memory vector $\overline{x}$ to take the *undefined value* $\uparrow$, and the introduction of the following predicate and set:

- $a \downarrow \overline{x}$ iff $x_a \neq \uparrow$
- $r(\overline{x}) =_{df} \{\overline{y} \mid \forall a \in A\ y_a = x_a \vee y_a = \uparrow\}$

Let $\mathcal{U}/\mathcal{R} : Act \rightarrow Act$ denote the renaming function that maps each read action $R_i(d,a)$ to the corresponding update action $U_i(d,a)$ for all i, d, and a, and all other actions to themselves. We are now ready to define the second refinement of our design as follows.

$$
Refinement_2 \Leftarrow \quad \prod_{j \in I}^{\mathcal{W}} (Locmem_j(\overline{0})[\mathcal{U}/\mathcal{R}] \mid\mid_{\mathcal{U}_j \cup \mathcal{W}} Cache_j(\overline{y}_{j0}))/\mathcal{U} \tag{2.5}
$$

$$
\text{for arbitrary } \overline{y}_{j0} \in r(\overline{0}).
$$

The correctness of this step follows from the following lemma.

Lemma 2.4.4 $\forall \overline{x} \in D^A, \overline{y} \in r(\overline{x}), j \in I$

$$
(Locmem_j(\overline{x})[\mathcal{U}/\mathcal{R}] \mid\mid_{\mathcal{U}_j \cup \mathcal{W}} Cache_j(\overline{y}))/\mathcal{U} \approx Locmem_j(\overline{x})
$$

Proof. The relation

$$
\{((Locmem_j(\overline{x})[\mathcal{U}/\mathcal{R}] \mid\mid_{\mathcal{U}_j \cup \mathcal{W}} Cache_j(\overline{y}))/\mathcal{U}, Locmem_j(\overline{x})) \mid \overline{x} \in D^A, \overline{y} \in r(\overline{x})\}
$$

is a weak bisimulation relation. It suffices to consider the following cases:

- $(Locmem_j(\overline{x})[\mathcal{U}/\mathcal{R}] \,||_{\mathcal{U}_j \cup \mathcal{W}}\, Cache_j(\overline{y}))/\mathcal{U} \stackrel{\varsigma}{\Rightarrow} B$:
  Then $B = (Locmem_j(\overline{x})[\mathcal{U}/\mathcal{R}] \,||_{\mathcal{U}_j \cup \mathcal{W}}\, Cache_j(\overline{y}'))/\mathcal{U}$ with $\overline{y}' \in r(\overline{x})$ where the silent transitions in $\stackrel{\varsigma}{\Rightarrow}$ consist of zero or more cache invalidations and/or updates. It suffices to take $Locmem_j(\overline{x}) \stackrel{\varsigma}{\Rightarrow} Locmem_j(\overline{x})$.
- $(Locmem_j(\overline{x})[\mathcal{U}/\mathcal{R}] \,||_{\mathcal{U}_j \cup \mathcal{W}}\, Cache_j(\overline{y}))/\mathcal{U} \stackrel{W_i(d,a)}{\longrightarrow} B$:
  Then $B = (Locmem_j(\overline{x}\{d/x_a\})[\mathcal{U}/\mathcal{R}] \,||_{\mathcal{U}_j \cup \mathcal{W}}\, Cache_j(\overline{y}\{d/y_a\}))/\mathcal{U}$. This is directly matched by $Locmem_j(\overline{x}) \stackrel{W_i(d,a)}{\longrightarrow} Locmem_j(\overline{x}\{d/x_a\})$.
- $(Locmem_j(\overline{x})[\mathcal{U}/\mathcal{R}] \,||_{\mathcal{U}_j \cup \mathcal{W}}\, Cache_j(\overline{y}))/\mathcal{U} \stackrel{R_j(x_a,a)}{\longrightarrow} B$:
  Then $B = (Locmem_j(\overline{x})[\mathcal{U}/\mathcal{R}] \,||_{\mathcal{U}_j \cup \mathcal{W}}\, Cache_j(\overline{y}))/\mathcal{U}$. This is directly matched by $Locmem_j(\overline{x}) \stackrel{R_j(x_a,a)}{\longrightarrow} Locmem_j(\overline{x})$.
- $Locmem_j(\overline{x}) \stackrel{\varsigma}{\Rightarrow} B$:
  Then $B = Locmem_j(\overline{x})$. This is therefore directly matched by
  $(Locmem_j(\overline{x})[\mathcal{U}/\mathcal{R}] \,||_{\mathcal{U}_j \cup \mathcal{W}}\, Cache_j(\overline{y}))/\mathcal{U} \stackrel{\varsigma}{\Rightarrow}$
  $(Locmem_j(\overline{x})[\mathcal{U}/\mathcal{R}] \,||_{\mathcal{U}_j \cup \mathcal{W}}\, Cache_j(\overline{y}))/\mathcal{U}$.
- $Locmem_j(\overline{x}) \stackrel{W_i(d,a)}{\longrightarrow} B$:
  Then $B = Locmem_j(\overline{x}\{d/x_a\})$. This is directly matched by
  $(Locmem_j(\overline{x})[\mathcal{U}/\mathcal{R}] \,||_{\mathcal{U}_j \cup \mathcal{W}}\, Cache_j(\overline{y}))/\mathcal{U} \stackrel{W_i(d,a)}{\longrightarrow}$
  $(Locmem_j(\overline{x}\{d/x_a\})[\mathcal{U}/\mathcal{R}] \,||_{\mathcal{U}_j \cup \mathcal{W}}\, Cache_j(\overline{y}\{d/y_a\}))/\mathcal{U}$.
- $Locmem_j(\overline{x}) \stackrel{R_j(x_a,a)}{\longrightarrow} B$:
  Then $B = Locmem_j(\overline{x})$. If $a \downarrow \overline{y}$ then this is directly matched by
  $(Locmem_j(\overline{x})[\mathcal{U}/\mathcal{R}] \,||_{\mathcal{U}_j \cup \mathcal{W}}\, Cache_j(\overline{y}))/\mathcal{U} \stackrel{R_j(x_a,a)}{\longrightarrow}$
  $(Locmem_j(\overline{x})[\mathcal{U}/\mathcal{R}] \,||_{\mathcal{U}_j \cup \mathcal{W}}\, Cache_j(\overline{y}))/\mathcal{U}$.
  If $y_a = \uparrow$ then first a cache update of address a must take place. This generates the following matching sequence of actions:
  $(Locmem_j(\overline{x})[\mathcal{U}/\mathcal{R}] \,||_{\mathcal{U}_j \cup \mathcal{W}}\, Cache_j(\overline{y}))/\mathcal{U} \stackrel{\tau}{\longrightarrow}$
  $(Locmem_j(\overline{x})[\mathcal{U}/\mathcal{R}] \,||_{\mathcal{U}_j \cup \mathcal{W}}\, Cache_j(\overline{y}\{x_a/y_a\}))/\mathcal{U} \stackrel{R_j(x_a,a)}{\longrightarrow}$
  $(Locmem_j(\overline{x})[\mathcal{U}/\mathcal{R}] \,||_{\mathcal{U}_j \cup \mathcal{W}}\, Cache_j(\overline{y}\{x_a/y_a\}))/\mathcal{U}$ □

□

Corollary 2.4.5 *$Refinement_2$ is weak sequential consistent with $Mem_{ser}(\overline{0})$*

Proof. Because $\approx$ is a congruence relation w.r.t. the parallel combinator $||_G$ (fact 2.2.4) it follows from that $Refinement_2 \approx Refinement_1$. Combining this with $\approx \subseteq \approx_{trace}$ (fact 2.2.6) and corollary 2.4.3 the desired result now follows directly. □

□

2.4.3 Buffering cache communication

In this refinement step we will buffer the communication of write/update actions to the cache, and only allow read actions if there are no local write actions buffered. This can be expressed using a family of queue-like action transducers in the sense of section 2.3.

Definition 2.4.6 *The family of queue-like action transducers* $\{K_j^\sigma \mid \sigma \in (\mathcal{W} \cup \mathcal{U}_j)^*\}$ *is for each* $j \in I$ *completely characterized by the following set of transductions:*

- $K_j^\sigma \xrightarrow[0]{U_j(d,a)} K_j^{\sigma.U_j(d,a)}$
- $K_j^\sigma \xrightarrow[0]{W_i(d,a)} K_j^{\sigma.W_i(d,a)}$ *for all* $i \in I$
- $K_j^{U_j(d,a).\sigma} \xrightarrow[U_j(d,a)]{\tau} K_j^\sigma$
- $K_j^{W_i(d,a).\sigma} \xrightarrow[W_i(d,a)]{\tau} K_j^\sigma$ *for all* $i \in I$
- $K_j^\sigma \xrightarrow[R_j(d,a)]{R_j(d,a)} K_j^\sigma$ *if* σ *contains no* $\mathcal{W}_j$*-actions*

□

The refinement is reflected in the following process definition.

$$Refinement_3 \Leftarrow \prod_{j \in I}^{\mathcal{W}} (Locmem_j(\overline{0})[\mathcal{U}/\mathcal{R}] \,||_{\mathcal{U}_j \cup \mathcal{W}}\, K_j^\epsilon[Cache_j(\overline{y}_{j0})])/\mathcal{U} \tag{2.6}$$

for arbitrary $\overline{y}_{j0} \in r(\overline{0})$.

We can now prove the following lemma.

Lemma 2.4.7

$$\begin{aligned}
&\forall j \in I, \sigma \in (\mathcal{W} \cup \mathcal{R}_j \cup \mathcal{U}_j)^*, \overline{x} \in D^A, \overline{y} \in r(\overline{x}) \\
&\qquad (\mathrm{Locmem}_j(\overline{0})[\mathcal{U}/\mathcal{R}] \,||_{\mathcal{U}_j \cup \mathcal{W}}\, K_j^\epsilon[Cache_j(\overline{y}_{j0})])/\mathcal{U} \stackrel{\sigma}{\Rightarrow} \\
&\exists \sigma' \in (\mathcal{W} \cup \mathcal{R}_j \cup \mathcal{U}_j)^* \\
&\qquad (\mathrm{Locmem}_j(\overline{0})[\mathcal{U}/\mathcal{R}] \,||_{\mathcal{U}_j \cup \mathcal{W}}\, Cache_j(\overline{y}_{j0}))/\mathcal{U} \stackrel{\sigma'}{\Rightarrow} \\
&\qquad \wedge\ \sigma\lceil(\mathcal{W}_j \cup \mathcal{R}_j) = \sigma'\lceil(\mathcal{W}_j \cup \mathcal{R}_j) \wedge\ \sigma\lceil\mathcal{W} = \sigma'\lceil\mathcal{W}
\end{aligned}$$

Proof. This essentially follows from the preservation lemma 2.3.7. Assume that

$$(Locmem_j(\overline{0})[\mathcal{U}/\mathcal{R}] \,||_{\mathcal{U}_j \cup \mathcal{W}}\, K_j^\epsilon[Cache_j(\overline{y}_{j0})])/\mathcal{U} \stackrel{\sigma}{\Rightarrow}$$

It follows there must exist a σ_1 with $\sigma_1/\mathcal{U} = \sigma$ and

$$Locmem_j(\overline{0})[\mathcal{U}/\mathcal{R}] \,||_{\mathcal{U}_j \cup \mathcal{W}}\, K_j^\epsilon[Cache_j(\overline{y}_{j0})] \stackrel{\sigma_1}{\Rightarrow}$$

By the properties of $||_{\mathcal{U}_1 \cup \mathcal{W}}$ (fact 2.2.7) for $\sigma_2 = \sigma_1\lceil(\mathcal{U}_j \cup \mathcal{W}_j)$ we have

$$Locmem_j(\overline{0})[\mathcal{U}/\mathcal{R}] \stackrel{\sigma_2}{\Rightarrow} \text{ and } K_j^\epsilon[Cache_j(\overline{y}_{j0})] \stackrel{\sigma_1}{\Rightarrow}$$

By the preservation lemma 2.3.7 there is a σ_1' with $Cache_j(\overline{y}_{j0}) \stackrel{\sigma_1'}{\Rightarrow}$ and

$$\sigma_1'\lceil(\mathcal{W}_j \cup \mathcal{R}_j) = \sigma_1\lceil(\mathcal{W}_j \cup \mathcal{R}_j) \text{ and } \sigma_1'\lceil(\mathcal{W} \cup \mathcal{U}_j) = \sigma_1\lceil(\mathcal{W} \cup \mathcal{U}_j)$$

which follows by taking $A = \mathcal{W}_j$ (then $D_A = \mathcal{R}_j$), and $A = \mathcal{W} \cup \mathcal{U}_j$ (then $D_A = \emptyset$), respectively. Recombining, we get

$$Locmem_j(\overline{0})[\mathcal{U}/\mathcal{R}] \,||_{\mathcal{U}_j \cup \mathcal{W}}\, Cache_j(\overline{y}_{j0}) \stackrel{\sigma_1'}{\Rightarrow}$$

Then taking $\sigma' = \sigma'_1/\mathcal{U}$ it follows that

$$(Locmem_j(\overline{0})[\mathcal{U}/\mathcal{R}] \;||_{\mathcal{U}_1 \cup \mathcal{W}}\; K_j^{\epsilon}[Cache_j(\overline{y}_{j0})])/\mathcal{U} \stackrel{\sigma'}{\Rightarrow}$$

with

$$\sigma\lceil(\mathcal{W}_j \cup \mathcal{R}_j) = (\sigma_1/\mathcal{U})\lceil(\mathcal{W}_j \cup \mathcal{R}_j) = (\sigma_1\lceil(\mathcal{W}_j \cup \mathcal{R}_j))/\mathcal{U} = (\sigma'_1\lceil(\mathcal{W}_j \cup \mathcal{R}_j))/\mathcal{U} = (\sigma'_1/\mathcal{U})\lceil(\mathcal{W}_j \cup \mathcal{R}_j) = \sigma'\lceil(\mathcal{W}_j \cup \mathcal{R}_j)$$

and likewise

$$\sigma\lceil\mathcal{W} = (\sigma_1/\mathcal{U})\lceil\mathcal{W} = (\sigma_1\lceil\mathcal{W})/\mathcal{U} = (\sigma'_1\lceil\mathcal{W})/\mathcal{U} = (\sigma'_1/\mathcal{U})\lceil\mathcal{W} = \sigma'\lceil\mathcal{W}$$

□
□

Corollary 2.4.8 *$Refinement_3$ is weak sequential consistent with $Mem_{ser}(\overline{0})$*

Proof. Assume that

$$\prod_{j\in I}^{\mathcal{W}} (Locmem_j(\overline{0})[\mathcal{U}/\mathcal{R}] \;||_{\mathcal{U}_j \cup \mathcal{W}}\; K_j^{\epsilon}[Cache_j(\overline{y}_{j0})])/\mathcal{U} \stackrel{\sigma}{\Rightarrow}$$

then according to fact 2.2.7 for each $j \in I$ with $\sigma_j = \sigma\lceil(\mathcal{W} \cup \mathcal{R}_j)$ we have

$$(Locmem_j(\overline{0})[\mathcal{U}/\mathcal{R}] \;||_{\mathcal{U}_j \cup \mathcal{W}}\; K_j^{\epsilon}[Cache_j(\overline{y}_{j0})])/\mathcal{U} \stackrel{\sigma_j}{\Rightarrow}$$

Also, it follows that for all $j \in I$ the σ_j must agree on their common actions in $\mathcal{W}$, i.e. $\sigma_{j_1}\lceil\mathcal{W} = \sigma_{j_2}\lceil\mathcal{W}$ for $j_1, j_2 \in I$.

Using the above lemma we find σ'_j with $\sigma_j\lceil(\mathcal{W}_j \cup \mathcal{R}_j) = \sigma'_j\lceil(\mathcal{W}_j \cup \mathcal{R}_j)$ and $\sigma_j\lceil\mathcal{W} = \sigma'_j\lceil\mathcal{W}$. The latter equality implies that for $j_1, j_2 \in I$ we have $\sigma'_{j_1}\lceil\mathcal{W} = \sigma_{j_1}\lceil\mathcal{W} = \sigma_{j_2}\lceil\mathcal{W} = \sigma'_{j_2}\lceil\mathcal{W}$. This means that we can apply fact 2.2.7 again, in the opposite direction, combining the σ'_j and find a σ' with $\sigma'\lceil(\mathcal{W} \cup \mathcal{R}_j) = \sigma'_j\lceil(\mathcal{W} \cup \mathcal{R}_j)$

$$\prod_{j\in I}^{\mathcal{W}} (Locmem_j(\overline{0})[\mathcal{U}/\mathcal{R}] \;||_{\mathcal{U}_j \cup \mathcal{W}}\; K_j^{\epsilon}[Cache_j(\overline{y}_{j0})])/\mathcal{U} \stackrel{\sigma'}{\Rightarrow}$$

It follows that $\sigma'\lceil(\mathcal{W}_j \cup \mathcal{R}_j) = \sigma\lceil(\mathcal{W}_j \cup \mathcal{R}_j)$ for all $j \in I$, i.e. *$Refinement_3$* is weak sequential consistent with *$Refinement_2$*, and thus with $Mem_{ser}(\overline{0})$. □
□

We proceed with a cosmetic transformation that is not really necessary for the design, but brings our specification closer in line with the specification given in the problem statement in [Ger95]. There, the cache communication buffer identifies all update and non-local write interactions once they have been buffered. The contents of local write interactions is marked for identification with a special symbol ('$*$'). To achieve this in our design we introduce a revised class of queue-like transducer families.

Definition 2.4.9 *The family of queue-like action transducers $\{L_j^{\sigma} \mid \sigma \in (\mathcal{W} \cup \mathcal{U}_j)^*\}$ is for each $j \in I$ completely characterized by the following set of transductions:*

- $L_j^\sigma \xrightarrow[0]{U_j(d,a)} L_j^{\sigma.(d,a)}$
- $L_j^\sigma \xrightarrow[0]{W_j(d,a)} K_j^{\sigma.(d,a,*)}$
- $L_j^\sigma \xrightarrow[0]{W_i(d,a)} K_j^{\sigma.(d,a)}$ $\quad i \neq j$ □
- $L_j^{\alpha(d,a).\sigma} \xrightarrow[U_j(d,a)]{\tau} L_j^\sigma$ $\quad \alpha(d,a) \in \{(a,d),(a,d,*)\}$
- $L_j^\sigma \xrightarrow[R_j(d,a)]{R_j(d,a)} L_j^\sigma$ $\quad$ *if σ contains no $*$-actions*

The corresponding revision of the cache specification is given by the process definition of $Cache'_j(\overline{x})$ below.

$$Cache'_j(\overline{x}) \Leftarrow \sum_{a\in A, d\in D} U_j(d,a).Cache'_j(\overline{x}\{d/x_a\}) + \sum_{a\downarrow\overline{x}} R_j(x_a,a).Cache'_j(\overline{x}) + \sum_{\overline{y}\in r(\overline{x})} \tau.Cache'_j(\overline{y}) \tag{2.7}$$

The overall refinement step that is implied by these changes is given by the process definition $Refinement_{3'}$.

$$Refinement_{3'} \Leftarrow \prod_{j\in I}^{\mathcal{W}} (Locmem_j(\overline{0})[\mathcal{U}/\mathcal{R}] \,||_{\mathcal{U}_j\cup\mathcal{W}}\, L_j^\epsilon[Cache'_j(\overline{y}_{j0})])/\mathcal{U} \tag{2.8}$$

for arbitrary $\overline{y}_{j0} \in r(\overline{0})$.

Essentially, $L_j^\epsilon[Cache'_j(\overline{y}_{j0})]$ differs from $K_j^\epsilon[Cache_j(\overline{y}_{j0})]$ only in the way in which the internal events corresponding to the buffer-cache communication are produced; the resulting transition systems are identical.

Lemma 2.4.10

$$L_j^\epsilon[Cache'_j(\overline{y}_{j0})] \;\sim\; K_j^\epsilon[Cache_j(\overline{y}_{j0})]$$

Proof. Left to the reader. □ □

Corollary 2.4.11 *$Refinement_{3'}$ is weak sequential consistent with $Mem_{ser}(\overline{0})$*

Proof. As $\sim$ is a congruence w.r.t. the operators used and preserves traces. □ □

2.4.4 Centralizing background memory

As the local memories have served their purpose in producing the local (buffered) caches they can now be recombined into a central background memory. Therefore, our penultimate design step is specified as follows.

$$Refinement_4 \Leftarrow \quad (Mem_{ser}(\overline{0})[\mathcal{U}/\mathcal{R}] \,||_{\mathcal{U}\cup\mathcal{W}} \prod_{j\in I}^{\mathcal{W}} L_j^{\varepsilon}[Cache'_j(\overline{y}_{j0})])/\mathcal{U} \tag{2.9}$$

$$\text{for arbitrary } \overline{y}_{j0} \in r(\overline{0}).$$

Lemma 2.4.12

$$(Mem_{ser}(\overline{0})[\mathcal{U}/\mathcal{R}] \,||_{\mathcal{U}\cup\mathcal{W}} \prod_{j\in I}^{\mathcal{W}} L_j^{\varepsilon}[Cache'_j(\overline{y}_{j0})])/\mathcal{U} \;\sim\; \prod_{j\in I}^{\mathcal{W}} (Locmem_j(\overline{0})[\mathcal{U}/\mathcal{R}] \,||_{\mathcal{U}_j\cup\mathcal{W}} L_j^{\varepsilon}[Cache'_j(\overline{y}_{j0})])/\mathcal{U}$$

Proof.

$\prod_{j\in I}^{\mathcal{W}}(Locmem_j(\overline{0})[\mathcal{U}/\mathcal{R}] \,||_{\mathcal{U}_j\cup\mathcal{W}} L_j^{\varepsilon}[Cache'_j(\overline{y}_{j0})])/\mathcal{U}$

$\sim$ *{law 4 of table 2.3}*

$(\prod_{j\in I}^{\mathcal{W}}(Locmem_j(\overline{0})[\mathcal{U}/\mathcal{R}] \,||_{\mathcal{U}_j\cup\mathcal{W}} L_j^{\varepsilon}[Cache'_j(\overline{y}_{j0})]))/\mathcal{U}$

$\sim$ $\{L(Locmem_{j_1}(\overline{0})[\mathcal{U}/\mathcal{R}]) \cap L(Locmem_{j_2}(\overline{0})[\mathcal{U}/\mathcal{R}]) = \mathcal{W}\ (j_1 \neq j_2)$,
$L(Locmem_j(\overline{0})[\mathcal{U}/\mathcal{R}]) \cap L(L_j^{\varepsilon}[Cache'_j(\overline{y}_{j0})]) = \mathcal{U}_j \cup \mathcal{W}\}$

$(\prod_{j\in I}^{*}(Locmem_j(\overline{0})[\mathcal{U}/\mathcal{R}] \,||_{*} L_j^{\varepsilon}[Cache'_j(\overline{y}_{j0})]))/\mathcal{U}$

$\sim$ *{laws 1 and 3 of table 2.3}*

$(\prod_{j\in I}^{*} Locmem_j(\overline{0})[\mathcal{U}/\mathcal{R}] \,||_{*} \prod_{j\in I}^{*} L_j^{\varepsilon}[Cache'_j(\overline{y}_{j0})])/\mathcal{U}$

$\sim$ *{law 5 of table 2.3 and lemma 2.4.2}*

$(Mem_{ser}(\overline{0})[\mathcal{U}/\mathcal{R}] \,||_{*} \prod_{j\in I}^{*} L_j^{\varepsilon}[Cache'_j(\overline{y}_{j0})])/\mathcal{U}$

$\sim$ $\{L(Mem_{ser}(\overline{0})[\mathcal{U}/\mathcal{R}]) \cap L(\prod_{j\in I}^{*} L_j^{\varepsilon}[Cache'_j(\overline{y}_{j0})]) = \mathcal{U} \cup \mathcal{W}$,
$L(L_{j_1}^{\varepsilon}[Cache'_{j_1}(\overline{y}_{j_1 0})]) \cap L(L_{j_2}^{\varepsilon}[Cache'_{j_2}(\overline{y}_{j_2 0})]) = \mathcal{W}\ (j_1 \neq j_2)\}$

$(Mem_{ser}(\overline{0})[\mathcal{U}/\mathcal{R}] \,||_{\mathcal{U}\cup\mathcal{W}} \prod_{j\in I}{}^{\mathcal{W}} L_j^{\varepsilon}[Cache'_j(\overline{y}_{j0})])/\mathcal{U}$

□
□

Corollary 2.4.13 *$Refinement_4$ is weak sequential consistent with $Mem_{ser}(\overline{0})$*

Proof. As $\sim$ preserves traces. □
□

2.4.5 Adding the user interface

The last step in our design is the buffering of local write interactions with the users. Local read interaction is permitted only when the local write buffer is empty. Again, this can be conveniently modelled using families of queue-like action transducers.

Definition 2.4.14 *The family of queue-like action transducers* $\{M_j^\sigma \mid \sigma \in \mathcal{W}_j{}^*\}$ *is for each* $j \in I$ *completely characterized by the following set of transductions:*

- $M_j^\sigma \xrightarrow[0]{W_j(d,a)} M_j^{\sigma.W_j(d,a)}$
- $M_j^{W_j(d,a).\sigma} \xrightarrow[W_j(d,a)]{\tau} M_j^\sigma$
- $M_j^\varepsilon \xrightarrow[R_j(d,a)]{R_j(d,a)} M_j^\varepsilon$
- $M_j^\sigma \xrightarrow[\alpha]{\alpha} M_j^\sigma \qquad \alpha \in \{R_i(d,a), W_i(d,a) | j \neq i \in I\}$

□

The corresponding refinement is expressed by process definition *Refinement*$_5$ below (recall that in the beginning of this section we put $I = \{1, \ldots, n\}$).

$$\textit{Refinement}_5 \Leftarrow \tag{2.10}$$
$$(M_1^\varepsilon \circ \ldots \circ M_n^\varepsilon)[(Mem_{ser}(\overline{0})[\mathcal{U}/\mathcal{R}] \;||_{\mathcal{U}\cup\mathcal{W}} \prod_{j\in I}^{\mathcal{W}} L_j^\varepsilon[Cache'_j(\overline{y}_{j0})])/\mathcal{U}]$$

for arbitrary $\overline{y}_{j0} \in r(\overline{0})$.

Theorem 2.4.15 *For all* $i \in I$

$$(M_1^\varepsilon \circ \ldots \circ M_i^\varepsilon)[(Mem_{ser}(\overline{0})[\mathcal{U}/\mathcal{R}] \;||_{\mathcal{U}\cup\mathcal{W}} \prod_{j\in I}^{\mathcal{W}} L_j^\varepsilon[Cache'_j(\overline{y}_{j0})])/\mathcal{U}]$$

is weak sequential consistent with $Mem_{ser}(\overline{0})$.

Proof. By induction on i using preservation lemma 2.3.7 it is straightforward to show that the application of each M_i^ε preserves the actions in $\mathcal{W}_i \cup \mathcal{R}_i$ and in $\mathcal{W}_j \cup \mathcal{R}_j$ for $j \neq i$, choosing $A = \mathcal{W}_i$ and $A = \emptyset$, respectively. The sequential consistency with $Mem_{ser}(\overline{0})$ then follows from corollary 2.4.13. □
□

Corollary 2.4.16

$$(M_1^\varepsilon \circ \ldots \circ M_n^\varepsilon)[(Mem_{ser}(\overline{0})[\mathcal{U}/\mathcal{R}] \;||_{\mathcal{U}\cup\mathcal{W}} \prod_{j\in I}^{\mathcal{W}} L_j^\varepsilon[Cache'_j(\overline{y}_{j0})])/\mathcal{U}]$$

is weak sequential consistent with $Mem_{ser}(\overline{0})$.

Proof. Take $i = n$. □
□

2.5 Strong sequential consistency

Having completed the design and proven it correct in terms of weak sequential consistency we come back to the original formulation of the problem in [Ger95], where sequential consistency is required with respect to the *maximal* observable traces, i.e. possibly infinite traces, of the systems involved. This is a strictly stronger requirement, as can be learned from the following example.

Example 2.5.1 *Consider a serial memory with only two user interfaces and only a single memory location initially holding the value 0. Suppose now a distributed implementation displays the infinite trace*

$$W_1(1)(R_2(0))^\omega \text{ or } W_1(1)R_2(0)R_2(0)R_2(0)\ldots$$

that is, user 1 writes the value 1 into the memory and user 2 keeps on reading the initial value 0 infinitely often.

Note that every finite prefix of this trace is weak sequential consistent with the serial memory. For all n $W_1(1)(R_2(0))^n$ is weak sequential consistent with $(R_2(0))^n W_1(1)$, which is a valid behaviour of the serial memory. For the infinite trace $W_1(1)(R_2(0))^\omega$ there exists no analogous permutation, as can be readily checked. □

The above example shows that when infinite strings are considered sequential consistency implies a *liveness* property: a write by one user is eventually read by the other. In this section we will show that the lazy caching memory in fact satisfies this stronger requirement, and will require only minor adaptations of the proofs for weak sequential consistency.

First, let A^ω denote the set of finite *and* infinite strings over A. Then we define the set of finite and infinite traces of a behaviour B as

$$Traces_\omega(B) =_{df} \{\sigma_0.\sigma_1.\sigma_2.\cdots \in Act^\omega \mid \exists\{B_i\}_{i\in\mathbf{N}}\, B \equiv B_0, B_i \stackrel{\sigma_i}{\Rightarrow} B_{i+1}\}$$

Definition 2.5.2 ((strong sequential consistency)) *Let B_1 and B_2 be behaviour expressions with $L(B_i) \subseteq \mathcal{L}$. A behaviour B_1 is* strong sequential consistent *with B_2 iff*

$$\forall\sigma \in \text{Traces}_\omega(B_1)\ \exists\sigma' \in \text{Traces}_\omega(B_2) \text{ such that } \forall i \in I\ \sigma\lceil\mathcal{L}_i = \sigma'\lceil\mathcal{L}_i$$

□

To show the correctness of the distributed caching memory it suffices to extend some of the definitions and facts of section 2.2. We start with the equivalence corresponding to $Traces_\omega(B)$ defined by

$$B_1 \approx_{trace_\omega} B_2 \text{ iff } Traces_\omega(B_1) = Traces_\omega(B_2)$$

Fact 2.5.3 *The relation $\approx_{trace_\omega}$ is a congruence with respect to all the combinators introduced in table 2.1 and $\approx \subseteq \approx_{trace_\omega} \subseteq \approx_{trace}$.* □

Fact 2.5.4 *Let $B_1||_* B_2$ be defined as in Table 2.3.*

$$Traces_\omega(B_1||_* B_2) =$$
$$\{\sigma \in (L(B_1) \cup L(B_2))^\omega \mid \sigma\lceil L(B_1) \in Traces_\omega(B_1), \sigma\lceil L(B_2) \in Traces_\omega(B_2)\}$$

□

The proofs of these facts are standard, and are left to the reader.

The last generalization that we need is the extension of lemma 2.3.7 to strings in Act^ω. This is the only part of the proof in which we will need the *weak fairness* assumption given in the problem description in [Ger95]: that no read, write, or update action is continuously enabled but never executed.

Lemma 2.5.5 ((extended preservation lemma)) *Let $\mathcal{T}_Q = \{T^\sigma \mid \sigma \in Q^*\}$ be a queue-like family of action-transducers. Let B continuously allow all actions in Q, i.e. for all $B' \in Der(B)$ and all $q \in Q$ $\exists B''$ $B' \xrightarrow{q} B''$. Then for all $A \subseteq Q$ we have*

$$\forall \sigma \in Traces_\omega(T^\epsilon[B])\ \exists \sigma' \in Traces_\omega(B)\ \textit{with}\ \sigma\lceil(A \cup D_A) = \sigma'\lceil(A \cup D_A)$$

Proof. We may assume that σ is an infinite trace, otherwise the proof of lemma 2.3.7 applies. By the definition of an infinite trace we then get that $\sigma = \sigma_0.\sigma_1.\sigma_2.\cdots$ with

$$\exists\{T^{\upsilon_i}[B_i]\}_{i\in\mathbf{N}}\ T^{\upsilon_1}[B_1] \stackrel{\sigma_i}{\Rightarrow} T^{\upsilon_{i+1}}[B_{i+1}]\ \textit{with}\ T^{\upsilon_0}[B_0] \equiv T^\epsilon[B]$$

Factorizing these transitions into transductions of the context and transitions of (the derivatives of) B we get

$$\exists\{\sigma'_i\}_{i\in\mathbf{N}}\ T^{\upsilon_i} \underset{\sigma'_i}{\overset{\sigma_i}{\Longrightarrow}} T^{\upsilon_{i+1}}\ \textit{and}\ B_i \stackrel{\sigma'_i}{\Rightarrow} B_{i+1}$$

It follows from lemma 2.3.6 that $(\sigma'_0.\cdots.\sigma'_i)\lceil(A \cup D_A)$ is prefix of $(\sigma_0.\cdots.\sigma_i)\lceil(A \cup D_A)$ for all i.

Now define $\sigma' = \sigma'_0.\sigma'_1.\sigma'_2.\cdots$, and suppose that $\sigma\lceil(A \cup D_A) \neq \sigma'\lceil(A \cup D_A)$, then it follows that $\sigma\lceil(A \cup D_A) = \sigma'\lceil(A \cup D_A).\sigma''\lceil(A \cup D_A)$ for some σ'' with $\sigma''\lceil(A \cup D_A) \neq \epsilon$. The latter entails in particular that $\sigma''\lceil A \neq \epsilon$ as the elements in D_A would, by construction, already occur in σ'. Also, it follows that $\sigma'\lceil(A \cup D_A)$ is finite, i.e. that there exists an N such that $\sigma'_i\lceil(A \cup D_A) = \epsilon$ for all $i > N$. By the transduction rules for queue-like transducers this implies that υ_i is a prefix of υ for all transducers T^υ that occur in the derivation of $T^{\upsilon_j} \underset{\sigma'_j}{\overset{\sigma_j}{\Longrightarrow}} T^{\upsilon_{j+1}}$ for $j \geq i > N$.

Because $\sigma''\lceil A \neq \epsilon$ we get that $\upsilon_i \neq \epsilon$ from some $M > N$ onwards. As B continuously allows all actions in Q, in particular the first element u_0 of υ_M, this action is continuously enabled as $T^{\upsilon_i} \xrightarrow[u_0]{\tau} T^{\upsilon'}$ for $i > M$ and $\upsilon_i = u_0.\upsilon'$. But it is never selected, because $i > N$ and υ_i is not a prefix of υ'. This contradicts our fairness assumption. Therefore $\sigma\lceil(A \cup D_A) = \sigma'\lceil(A \cup D_A)$. □

□

Theorem 2.5.6

$$(M_1^\epsilon \circ \ldots \circ M_n^\epsilon)[(\mathrm{Mem}_{ser}(\overline{0})[\mathcal{U}/\mathcal{R}] \,||_{\mathcal{U}\cup\mathcal{W}} \prod_{j\in I}^{\mathcal{W}} L_j^\epsilon[Cache'_j(\overline{y}_{j0})])/\mathcal{U}]$$

is strong sequential consistent with $Mem_{ser}(\overline{0})$.

Proof. We check proofs of the refinement steps for the weak sequential case:

1. *distributing the memory*: this was proved using that $\sim \subseteq \approx_{trace}$ (see corollary 2.4.3), which can now be replaced by the argument that $\sim \subseteq \approx_{trace_\omega}$.
2. *introducing local caching*: this was proved using that $\approx \subseteq \approx_{trace}$ (see corollary 2.4.5), which can now be replaced by the argument that $\approx \subseteq \approx_{trace_\omega}$.
3. *buffering cache communication*: an infinite trace version of lemma 2.4.7 can be proved using fact 2.5.4 instead of fact 2.2.7, and the extended preservation lemma 2.5.5, which leads to the strong version of corollary 2.4.8. The subsequent modification in $Refinement_3$, can be imitated as $\approx_{trace_\omega}$ is invariant under renaming of internal actions.
4. *centralizing background memory*: this is more or less the inverse of refinement 1, and therefore follows again by $\sim \subseteq \approx_{trace_\omega}$, and the fact that $\approx_{trace_\omega}$ is a congruence.
5. *adding the user interface*: this follows by using the extended version of the preservation lemma. □

□

2.6 Conclusions

In this paper we have presented a proof of the sequential consistency of the lazy caching protocol of [ABM93]. It is based on the application of a number of transformation steps, deriving the distributed caching memory in several steps from the sequential memory, whilst maintaining the property of sequential consistency. Thus the proof can also be seen as a rationalized reconstruction of the design of the lazy caching protocol, and a *a posteriori* attempt at *correctness by design*. One of the potential benefits of such an approach is that more general results can be obtained than the correctness of a specific design only. In this case the factorization of the proof in separate design steps gives substantial insight in design alternatives, and in fact provides us with correctness proofs for a whole family of distributed caching designs. Being based on the same transformation principles the following variations can be proven correct by minimal rearrangements of the proof:

1. *user interface buffers*: we can allow asymmetry between users in the sense that some may have buffered and others may not have a buffered user interface.

2. *cache buffers*: we can also allow asymmetry between caches in the sense that some may have buffered access and others not.

3. *local memories*: we may choose some users to have access to a complete local memory instead of a cache.

4. *background memories*: we may choose to have several write-synchronizing background memories for smaller user groups (e.g. to expedite cache updates).

The structured presentation of the proof also allows for a rather precise analysis of the blanket fairness assumption (no action other than cache invalidations can always be enabled but never taken) in general exposition in [Ger95]. Weak fairness is required in the following places:

1. processing local writes stored in the user interface buffers into the memory and the local cache buffers;

2. processing writes and updates stored in a local cache buffer into the local cache;

3. processing memory updates into the local cache buffers.

The first two are used in (the application of) the extended preservation lemma 2.5.5; the last is implicit in the proof of weak bisimulation equivalence in lemma 2.4.4. The latter exploits a notion of fairness that is 'built-in' in the notion of weak bisimulation equivalence. In the context of ACP it appears as *Koomen's fair abstraction rule* [BW90].

Although we have used a process-algebraic notation for the specification of the various design stages, and have applied a number of well-known laws from the process-algebraic literature, our proof is, in fact, heterogeneous in nature. The process-algebraic syntax is used to define labelled transitions systems. We have allowed, however, some of the fairness requirements to be superimposed on this representation, thereby leaving a proper process-algebraic framework. Also, we have not used a structured syntax to define action transducers, but have defined them directly in terms of their transductions. As already mentioned, the transducers have their syntactic counterparts in behaviour expression contexts, i.e. behaviour expressions with open places or 'holes' in them. Contexts corresponding to the transducers that we have used could be expressed in terms of our process-algebraic formalism if we accept simple compound data types such as *strings* and their associated

operations as given (otherwise one could turn to languages like LOTOS to formalize such notions [BB87]). In these cases, however, their syntactic representation is much more involved than their operational one, and would distract from the essential feature that figures in the proof, viz. that they are action transducers that induce *observable action-sequence* transductions. As sequential consistency is an invariant of such transductions, that is precisely the way we want to view them.

The correctness of a number of transformations has been shown in terms of direct semantic proofs, viz. by producing strong and weak bisimulations, and by reasoning in terms of action transducers. As a consequence, it can be disputed as to what extent our proof can be seen as one based on the application of *correctness-preserving* transformations (CPTs). Although our transformations do preserve the desired correctness criterion, this term is usually reserved for generic design principles whose correctness has been established beforehand (cf. for example [Bol92]), to be contrasted with the procedure of '*invent and verify*'. In addition to the applied standard process algebraic laws listed in table 2.3, however, most other parts of the proof could retrospectively qualify as CPTs. The formulation of our transduction based proofs, the (extended) preservation lemma, for example, is generic in the sense that it applies to all queue-like transducers. This enables its repeated application in proof, viz. twice in the proof of lemma 2.4.7 concerning the cache buffer, and twice in the proof of theorem 2.4.15 concerning the user interface buffer. In order not to burden our proof with such concerns we have foregone the formulation of a generic transformation principle corresponding to the equivalence proven in lemma 2.4.2. The idea behind the proof is quite general, however, viz. that a process maybe split into parts according to a partitioning of all those of its actions that do not affect its state, where each part should still be able to synchronize on all actions that do influence the state in order to maintain it. We present a generic formulation of this transformation without proof.

Let $p(x)$ be a parameterized process defined by

$$p(x) \Leftarrow \sum_{a \in Var} f(a,x).p(g(a,x)) + \sum_{a \in Inv} h(a,x).p(x) \tag{2.11}$$

where x ranges over a given domain D, Var and Inv are given index sets, and $f : Var \times D \rightharpoonup Act$, $g : Var \times D \rightharpoonup D$, and $h : Inv \times D \rightharpoonup Act \cup \{\tau\}$ are functions with f injective and $rge(f) \cap rge(h) = \emptyset$.

Theorem 2.6.1 *Let $p(x)$ of the form defined by (2.11) above. Let $\mathcal{F}$ be a finite partitioning of Inv and define for all $F \in \mathcal{F}$*

$$p_F(x) \Leftarrow \sum_{a \in Var} f(a,x).p_F(g(a,x)) + \sum_{a \in F} h(a,x).p_F(x)$$

Then

$$p(x) \sim \prod_{F \in \mathcal{F}}^{rge(f)} p_F(x)$$

□

Sofar, we have not succeeded in formulating a suitably general formulation of the transformation principle behind the introduction of the local caches in lemma 2.4.4. It seems that the semantic idea behind it is not readily expressible in generic syntactic terms. Summarizing, we can say that the problem of proving the lazy caching protocol correct has also served as a source of inspiration for the formulation of new correctness preserving design transformations. Although much of our proof can be interpreted as the application of such transformations, parts remain that rely on the '*invent and verify*' approach. As a whole the proof illustrates that an opportunistic combination of different methods can lead to an insightful example of correctness by design.

Chapter 3

Sequential Consistency as Interface Refinement

R. Gerth

3.1 Interface Refinement

The proof of sequential consistency will be based on our notion of interface refinement. The approach that we shall use is based on a much streamlined version of the one published in [GKS92]. This section intends to supply a quick introduction to interface refinement and a (derived) proof rule that is specifically engineered for proving sequential consistency. A full account of the general, streamlined approach will be published elsewhere.

We assume some general knowledge of linear temporal logic and of transition systems.

If we compare the definitions of sequential consistency

$$C \; s.c. \; A \;\; \text{iff} \;\; \forall \sigma \in \text{Beh}(C) \; \exists \tau \in \text{Beh}(A) \; \forall i = 1 \ldots n \;\; \sigma \lceil i = \tau \lceil i$$

and that of standard (trace) refinement

$$C \; \mathsf{ref} \; A \;\; \text{iff} \;\; \forall \sigma \in \text{Beh}(C) \; \exists \tau \in \text{Beh}(A) \;\; \sigma = \tau$$

we detect a pattern:

$$C \; \mathsf{ref}_R \; A \;\; \text{iff} \;\; \forall \sigma \in \text{Beh}(C) \; \exists \tau \in \text{Beh}(A) \;\; (\sigma, \tau) \in R$$

I.e., these cases can be viewed as refinements, except that the way in which an abstract behavior σ gets implemented as τ may change. Consequently, the refinement relation is *parameterized* with a relation R that determines how behaviors are implemented. For example, the relation is that of equality for ordinary refinement. This pattern is also shared by, e.g., the condition of serializebility of database transactions and by Lamport's 'stutter closed' refinement.

We assume that such relations are *specified* in some logic. I.e., a relation R is now given by a formula ϕ and $(\sigma, \tau) \in R$ iff $\sigma, \tau \models \phi$, for a suitably defined satisfaction relation $\models$.

The logic will be a linear temporal logic (LTL); although we shall only use always ($\Box$) and eventuality ($\Diamond$) properties. An LTL is usually valuated on (infinite) sequences of states. To express constraints on (internal) behaviors, we assume the logic to be extended with a *history variable* h that valuates at a point in a state sequence to the sequence of events that have occurred up to this point. A second complication is that here, the LTL is used to compare *two* state sequences. By convention, two (equal length) state sequences determine a single such sequence through taking the pointwise product of the states in the sequences. In the logic we can then use projection functions to refer to the separate sequences again. Write h_c and h_a for the projections of history h; 'c' for concrete and'a' for abstract.

We need to establish some notation. We generically assume that C and A are interpreted transition systems that have disjoint sets of (free) variables; see M_{serial} or M_{distr} for examples. Write $S(A)$ for the set of states of A; $I(A)$ for its initial states; and '$s \xrightarrow{\alpha} s'$ in A' if the event α is executed in state s of A and produces state s'. Remember that these states also valuate the variables; in particular the variable h, so that $s(\mathsf{h}) = \varepsilon$ if $s \in I(A)$ and if $s \xrightarrow{\alpha} s'$ in A then $s'(\mathsf{h}) = s(\mathsf{h})^\frown\alpha$. We often write just $s \xrightarrow{\alpha} s'$ if the transition system is clear from the context. Write $[\![A]\!]$ for the set of maximal sequences of states, obtained by repeatedly applying $\xrightarrow{\alpha}$ starting in some initial state of A. We assume that there are no finite state sequences in $[\![A]\!]$; as is the case for e.g. M_{serial} and M_{distr}. Because states valuate h, every state sequence $\sigma \in [\![A]\!]$ uniquely determines an event sequence, $\sigma^e \in \text{IBeh}(A)$; hence $\text{IBeh}(A) = \{\sigma^e \mid \sigma \in [\![A]\!]\}$. For states s and t write '$s \times t$' for their product or pairing. For (infinite) sequences of states σ and τ write $\sigma \times \tau$ for the sequence obtained by the pointwise product of the states in σ and τ. Write $\mathcal{H}$ for the set of (finite) event sequences h, h', ... History variables take there value from $\mathcal{H}$.

Definition 3.1.1 (Interface refinement) *Let ϕ be some LTL formula. Then*

$$C \ \mathsf{ref}_\phi \ A \ \text{ iff } \ \forall \sigma \in [\![C]\!] \ \exists \tau \in [\![A]\!] \ \sigma \times \tau \models \phi \, .$$

For example, standard trace refinement, C ref A, is defined as C ref_ϕ A by taking, e.g.,

$$\phi \equiv \Box \bar{\phi} \ \text{ and } \ \bar{\phi} \equiv \mathrm{last}(\mathsf{h}_c) = \mathrm{last}(\mathsf{h}_a) \, .$$

For $\sigma \in [\![C]\!]$ and $\tau \in [\![A]\!]$ we have by definition of $\Box$ that $\sigma \times \tau \models \phi$ holds just in case $\sigma \times \tau, k \models \bar{\phi}$ holds at every position k; i.e., for every state pair in $\sigma \times \tau$. If $s \times t$ is the k-th such state pair, then this is equivalent to $s \times t \models \bar{\phi}$ which holds precisely if (†) $last(s(\mathsf{h})) = last(t(\mathsf{h}))$. I.e., $(s \times t)(\mathsf{h}_c) = s(\mathsf{h})$ and $(s \times t)(\mathsf{h}_a) = t(\mathsf{h})$. Thus, (†) expresses that the event that produced s in C is the same as the one that produced t in A.

3.1.1 Sequential consistency as interface refinement

For this we make a simplifying assumption

> *Every process issues infinitely many writes to M_{distr}. Stated differently, on any $\sigma \in [\![M_{distr}]\!]$ and for any $i = 1 \ldots n$, $\sigma^e \restriction i$ contains infinitely may W_i events.*

This simplification is not essential for the proof; it does make it slightly easier.

Sequential consistency is a condition on maximal, hence, infinite sequences. To express this in an LTL, we must rewrite to a condition on states, i.e., on prefixes of the sequences, that must hold at various points along the sequences. A first try is

$$M_{distr} \text{ s.c. } M_{serial} \text{ iff } M_{distr} \ \mathsf{ref}_{\bar{\phi}} \ M_{serial} \text{ with } \bar{\phi} = \Box \textstyle\bigwedge_{i=1\ldots n} \bar{\phi}_i \text{ and}$$

$$\bar{\phi}_i \ \equiv \ \exists \mathsf{H} \, (\mathsf{h}_c = \mathsf{H} \wedge \Diamond H \restriction i \preceq \mathsf{h}_a \restriction i)$$

In $\sigma \times \tau \models \bar{\phi}_i$, the function of the quantification is to 'freeze' prefixes of the distributed behavior σ so that they can be matched against prefixes of the serial behavior τ. As every prefix of σ is eventually matched against a prefix of τ and because σ is infinite, we must have $\sigma \restriction i = \tau \restriction i$.

Another way of doing this is to associate with every prefix of τ a prefix of σ that can be matched against it. This approach leads to an easier proof. Now, however, we must make sure that we match ever longer prefixes of σ. Hence, we change $\bar{\phi}_i$ by replacing the existentially quantified temporal variable H by a 'choice function' f_i that maps a history to a prefix of that history. Say that $f : \mathcal{H} \to \mathcal{H}$ *increases i.o. on* A iff for every chain $h^0 \preceq h^1 \cdots$ such that $\lim_{n\to\infty} h^n = \mathrm{IBeh}(A)$ we have $\lim_{n\to\infty} |f_i(h^n)| = \infty$. Then

Lemma 3.1.2 *M_{distr} s.c. M_{serial} iff M_{distr} ref_ϕ M_{serial} with $\phi = \Box \bigwedge_{i=1\ldots n} \phi_i$ and*

$$\phi_i \ \equiv \ f_i(h_0) \restriction i \preceq h_1 \restriction i \quad \textit{for some } f_i \textit{ that increases i.o. on } M_{distr} \tag{3.1}$$

For completeness sake, we supply a proof. It is basically expanding definitions:

Proof. The left to right direction is obvious. Now assume that M_{distr} *s.c.* M_{serial} is not true. So, for some $\sigma = s_0 s_1 \cdots \in [\![M_{distr}]\!]$ and for every $\tau \in [\![M_{serial}]\!]$ we have $\sigma^e \restriction i \neq \tau^e \restriction i$ for some i. Fix such a σ, i and τ, and take any f_i that increases i.o. on M_{distr}.

For some index j we must have $(s_0 s_1 \cdots s_j)^e \restriction i \preceq \tau^e \restriction i$ and $(s_0 s_1 \cdots s_{j+1})^e \restriction i \not\preceq \tau^e \restriction i$ which is equivalent to $s_j(\mathsf{h}) \restriction i \preceq \tau^e \restriction i$ but $s_{j+1}(\mathsf{h}) \restriction i \not\preceq \tau^e \restriction i$. Now, consider $\Box \phi_i$. As f_i increases i.o. on M_{distr}, there is an index k such that $s_{j+1}(\mathsf{h}) \preceq f_i(s_k(\mathsf{h}))$. But then $\sigma, \tau, k \not\models \phi_i$ whence $\sigma, \tau \not\models \phi$. Since this conclusion holds for every $\tau \in [\![M_{serial}]\!]$ and any f_i, we conclude that M_{distr} ref_ϕ M_{serial} cannot hold. □

3.1.2 A proof rule

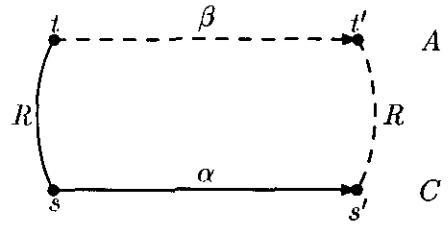
The first step in verifying sequential consistency C ref$_\phi$ A, or interface refinement in general, is to relate behaviors in the two systems with each other. The second step is then to prove that related behaviors satisfy the appropriate specification

A general technique for relating state sequences is that of *simulation* (backward or forward simulation, possibility mappings, implementation functions).

Definition 3.1.3 (Weak simulation) *Given transition systems C and A, a relation $R \subseteq S(C) \times S(A)$ is a* weak simulation of C in A, $C \hookrightarrow_R A$, *provided*

1. *for any $s \in I(C)$ there is an $t \in I(A)$ such that $(s,t) \in R$,*
2. *if $(s,t) \in R$ and $s \xrightarrow{\alpha} s'$ in C then there is an $t' \in S(A)$ such that $(s',t') \in R$ and either there is an event β such that $t \xrightarrow{\beta} t'$ in A or $t = t'$ (we say $\beta = \epsilon$ in this case)*

The inductive clause (2) is illustrated in the figure on the right. Given a state sequence $\sigma \in [\![C]\!]$, a weak simulation $C \hookrightarrow_R A$ constructs a state sequence τ of A in which every state in τ is related to some state in σ. However, we do not necessarily have $\tau \in [\![A]\!]$. First of all, A may have fairness constraints which τ may violate.



Secondly, τ may be finite because from some moment onwards R relates the transitions in σ with ϵ transitions 'in' τ. Fairness constraints are no problem for us, as M_{serial} will play the rôle of A and it does not have fairness constraints. Forcing τ to be infinite will be done implicitly, later on.

R is called a *weak* simulation because A is allowed to 'stutter' and because there are no constraints on the events of the transitions of C and A, nor on the related states. This is different from more standard forms of simulation where there are constraints on the events—e.g., $\alpha \equiv \beta$—or on the related states.

In our view, such condition are really implicitly defining how behaviors must be implemented and that is precisely what we want to avoid at this point. E.g., forcing $\alpha \equiv \beta$ in related transitions is forcing related sequences to be equal. If we set up such a stronger simulation between the states of C and A we are showing ordinary refinement.

Given a weak simulation, $C \hookrightarrow_R A$, the second step is to show that R–related sequences σ and τ satisfy $\sigma \times \tau \models \phi$. For sequential consistency this is easy, as it reduces to proving ϕ_i for every $i = 1 \ldots n$ in every related state pair.

This observation immediately suggests the proof rule in Figure 3.1.

A and C are transition systems such that A has no fairness constraints[1]:

$$\frac{C \hookrightarrow_R A, \qquad \forall s,t \;\; (s,t) \in R \;\Rightarrow\; s \times t \models \phi_i \; (i = 1 \ldots n)}{\models C \; s.c. \; A}$$

with $\phi_i \equiv f_i(h_0) \restriction i \preceq h_1 \restriction i$ for some f_i that increases i.o. on C

Figure 3.1: Proof rule for establishing sequential consistency

---

Soundness of the rule is immediate. Observe that because f_i must increase i.o. on C so that ϕ_i maps ever longer prefixes of $[\![C]\!]$ to prefixes of $[\![A]\!]$, the weak simulation R cannot associate a finite state sequence of $[\![A]\!]$ to one in $[\![C]\!]$.

The proof rule for general interface refinement, C ref_ϕ A, is based on the same ideas. The first step, again, is establishing a weak simulation. The second step changes because now ϕ need not be of the form $\Box\bar{\phi}$ and it is this form that determined the second premiss in rule 3.1. For instance, if $\phi \equiv \Box\Diamond\bar{\phi}$ for some state assertion $\bar{\phi}$, then we need to establish $\bar{\phi}$ at infinitely many state pairs along every pair of R-related state sequences. For this we introduce an auxiliary state formula d such that $C \models \Box\Diamond d$ and demand that $(s,t) \in R \;\&\; s \models d \;\Rightarrow\; s{\times}t \models \bar{\phi}$. In case $\phi \equiv \Box\Diamond\phi' \wedge \Diamond\phi''$ we would use two auxiliary state assertions d' and d'' such that $C \models \Box\Diamond d' \wedge \Diamond d''$, etc. The normal form result of [MP91] tells us that a finite number of auxiliary formulae always suffices. Specifically, we have that for every TL formula ϕ (without quantifiers) there is a propositional TL formula Ψ with propositional variables $p_1, \ldots, p_n$ and state formulae $\phi_1, \ldots, \phi_n$ such that $\models \phi \leftrightarrow \Psi[\phi_1/p_1, \ldots, \phi_n/p_n]$, where $\cdot/\cdot$ denotes syntactic substitution; we usually write $\Psi(\phi_1, \ldots \phi_n)$. The following proof rule applies to the general case.

A and C are transition systems such that A has no fairness constraints; Ψ is a propositional TL formula with propositional variables $p_1, \ldots, p_n$; and $\phi_1, \ldots, \phi_n$ and $d_1, \ldots, d_n$ are state formulae.

$$\frac{\begin{array}{l}\models \Psi(\phi_1, \ldots \phi_n) \rightarrow \phi \\ C \models \Psi(d_1, \ldots, d_n) \\ C \hookrightarrow_R A \\ s \models d_i \;\&\; (s,t) \in R \;\Rightarrow\; s{\times}t \models \phi_i \quad (i = 1 \ldots n)\end{array}}{\models C \;\mathsf{ref}_\phi\; A}$$

Figure 3.2: Proof rule for general interface refinement

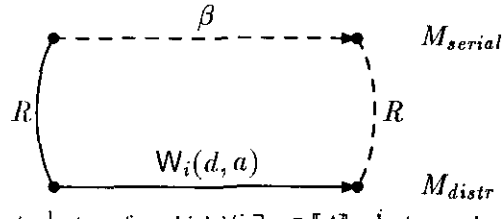
The sequential consistency proof rule is obtained by taking $d_i \equiv$ true and by noting that the formula in Lemma 3.1.2 is in normal form.

3.2 Correctness Proof of M_{distr} s.c. M_{serial}

3.2.1 Constructing a weak simulation R

The problem in defining a simulation is to decide when to 'allow' the serial memory to make a transition.

In the situation indicated on the right, β should not be the corresponding $\mathsf{W}_i(d,a)$-event. If it is, $\mathsf{R}_j(e,a)$-actions in the distributed memory that read an earlier value e at location a become disabled in the serial memory. This suggests that the corresponding serial write be postponed until the write has been *completed*,



[1] Formally: $[\![A]\!]$ must be closed in the sense that for any chain $\sigma^0 \preceq \sigma^1 \preceq \cdots$ for which $\forall i\ \exists \sigma \in [\![A]\!]\ \sigma^i \preceq \sigma$ we have $\lim_{i\to\infty} \sigma^i \in [\![A]\!]$

that is, until no processor can read an older value from
the distributed memory system, i.e., from its cache.
As a consequence, any read-action that reads the value of an uncompleted write-action is postponed as well.

We shall define R inductively, using a dag $<_h$. Given a state s of M_{distr}, the minimal elements of $<_{s(\mathsf{h})}$, i.e., the elements that are not the target of any edge in the dag, define the actions that M_{serial} can 'safely' execute. E.g., a write event α in $s(\mathsf{h})$ cannot be minimal as long as the event is still not completed and a read event, $\mathsf{R}_i(d,a)$, is not minimal as long as the write event that writes value d at location a has not occurred. Then, if $(s,t) \in R$ and $s \xrightarrow{\alpha} s'$ we take $(s',t') \in R$ for any t' such that $t \xrightarrow{\beta} t'$ where β is a minimal (enabled) event in $<_{s'(\mathsf{h})}$ that has not yet appeared in $t(\mathsf{h})$ (or ε if there are none).

Thus, along a state sequence $s_0 s_1 \cdots$ of M_{distr} the dag $<_{s_i(\mathsf{h})}$ functions as a scheduler of the events of M_{serial} and forces the R-related M_{serial} computation $t_0 t_1 \cdots$ to be always compatible with $s_0 s_1 \cdots$ so that at no point we can have $s_i(\mathsf{h}) \restriction i \not\preceq (t_0 t_1 \cdots)^e \restriction i$.

In order to formalize the above ideas, we adapt the transition system of M_{distr}; see Figure 3.3. Every write-action uniquely tags the value that it writes so that cache and memory update actions can be traced back to the specific action that 'caused' them.

Obviously, we still have

Lemma 3.2.1 $\mathrm{Beh}(M^T_{distr}) = \mathrm{Beh}(M_{distr})$

This is because $C_i(a) = d$ in M_{serial} iff $\exists \mathrm{n}\ C_i(a) = d \star \mathrm{n}$ in M_{distr} and the enabling condition of the other events are independent from any specific value of the data.

Since actions can occur more than once on behaviors the subsequent discussion is couched in terms of events, i.e., action-occurrences: (k,α) is the k-th occurrence of a type(α)-action in the behavior or history under discussion, where type(α) is defined as R_i or W_i depending on whether $\alpha \equiv \mathsf{R}_i(d,a)$ or $\alpha \equiv \mathsf{W}_i(d,a)$ for some d, a. Also write addr(α) for the location that the action α refers to. Write Act_i for the i-labeled actions and Ext_i for the i-labeled external actions.

For uniformity of notation and proof, the initial values of the memory are represented as pseudo-actions $\mathsf{W}_0(0,a)$ for every location a. Every M^T_{distr}-behavior is implicitly prepended with a sequence $(\mathsf{W}_0(0,a))_a$ where a ranges over all locations.

From now on, h (h') will denote prefixes of distributed (serial) memory internal behaviors

We define the following predicates. The more complex definitions are preceded by their intuitive meaning:

- (k,α) **occurs in** h iff α occurs in h (i.e., $h = h_0 \alpha h_1$ for some h_0, h_1) and h contains at least k occurrences of type(α)-events
- (k,α) **occurs before** (l,β) **in** h iff (l,β) **occurs in** h and there is a prefix h' of h such that (k,α) **occurs in** h' but not (l,β) **occurs in** h'
- $(k, \mathsf{W}_i(d,a))$ **is completed in** h iff $\forall j = 1 \ldots n\ \mathsf{CU}_j(d \star (k \star i), a)$ occurs in h
- α **completes** (k,β) **in** h iff not (k,β) **is completed in** h but (k,β) **is completed in** $h\alpha$

E	Event	Allowed if	Action
$\surd$	$\mathsf{R}_i(d,a)$	$C_i(a) = d\star\mathbf{n}$ **for some n** $\wedge\, Out_i = \{\}$ $\wedge$ no $*$-ed entries in In_i	
$\surd$	$\mathsf{W}_i(d,a)$		$\mathbf{t}_i := \mathbf{t}_i + \mathbf{1}$; $Out_i := append(Out_i, (d\star(\mathbf{t}_i\star\mathbf{i}), a))$
	$\mathsf{MW}_i(d,a)$	$head(Out_i) = (d,a)$	$Mem[a] := d$; $Out_i := tail(Out_i)$; $(\forall k \neq i :: In_k := append(In_k, (d,a)))$; $In_i := append(In_i, (d,a,*))$
	$\mathsf{MR}_i(d,a)$	$Mem[a] = d$	$In_i := append(In_i, (d,a))$
	$\mathsf{CU}_i(d,a)$	$head(In_i)$ 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: $\forall a\ \ Mem[a] = 0\star\mathbf{0}$
$\wedge\, \forall i = 1 \ldots n\ \ C_i \subset Mem \wedge In_i = \{\} \wedge Out_i = \{\} \wedge \mathbf{t}_i = \mathbf{0}$

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

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

$\cdot\star\cdot$ is some pairing function; say $n\star m = 2^n 3^m$.

Figure 3.3: Adapted M^T_{distr}

- (k,α) **is read by** (l,β) **in** h iff (k,α) is the (unique) write-event that caused the value read by (l,β) to be written. By convention, write-events are always read by themselves. More formally:

  (l,β) **occurs in** h and (i) β is a write-event and $(k,\alpha) = (l,\beta)$ or (ii) $\beta \equiv \mathsf{R}_i(d,a)$ for some i, d, a, $\alpha \equiv \mathsf{W}_j(d,a)$ for some j and either $j \neq 0$ and the last CU_i-event before (l,β) in h that refers to location a writes value $d\star(k\star j)$ or $j = 0$, $k = 0$, $d = 0$ and there are no CU-events before (l,β) in h that refer to location a

- $(k,\mathsf{W}_i(d,a))$ **distributes before** $(l,\mathsf{W}_j(d',a'))$ **in** h iff every cache 'sees' (i.e., is affected by) the $(k,\mathsf{W}_i(d,a))$-event before it sees the second event. More formally:

  $k = i = d = 0$ or $(k,\mathsf{MW}_i(d\star(k\star i),a))$ **occurs before** $(l,\mathsf{MW}_j(d'\star(l\star j),a'))$ **in** h

- $(k,\mathsf{R}_i(d,a))$ **reads before** $(l,\mathsf{W}_j(d',a))$ in h iff $(k,\mathsf{R}_i(d,a))$ reads a value at an address that will be overwritten by the $(l,\mathsf{W}_j(d',a))$-event. More formally:

  for some $(m,\mathsf{W}_k(d,a))$ we have that $(m,\mathsf{W}_k(d,a))$ **is read by** $(k,\mathsf{R}_i(d,a))$ **in** h and $(m,\mathsf{W}_k(d,a))$ **distributes before** $(l,\mathsf{W}_j(d',a))$ **in** h

- (k, α) **is ready in** h, h' iff $k > 0$, α is the k-th type(α)-event in h and there are precisely $\mathbf{k-1}$ type(α)-events in h'

We now state some important properties of the caching protocol. The whole correctness proof will be based on just these properties of M_{distr}.

Lemma 3.2.2 *Let $k > 0$ and $(k, \alpha) \neq (l, \beta)$. The following formulae are invariants of M^T_{distr}:*

1. $(k, \mathsf{R}_i(d, a))$ **occurs in** $\mathsf{h} \rightarrow (l, \mathsf{W}_j(d, a))$ **is read by** $(k, \mathsf{R}_i(d, a))$ **in** h *for some* $(l, \mathsf{W}_j(d, a))$

2. (k, α) **occurs in** $\mathsf{h} \wedge \mathrm{type}(\alpha) = \mathsf{W}_i \rightarrow \Diamond(k, \alpha)$ **is completed in** h

3. (k, α) **is read by** (l, β) **in** $\mathsf{h} \rightarrow (k, \alpha)$ **occurs before** (l, β) **in** h

4. $(k, \mathsf{MW}_i(d, a))$ **occurs before** $(l, \mathsf{MW}_j(d', a'))$ **in** $\mathsf{h} \rightarrow$
$\neg(l, \mathsf{CU}_f(d', ' a))$ **occurs before** $(k, \mathsf{CU}_f(d, a))$ **in** h

5. $\alpha \in Act_i \wedge \beta \in Act_i \rightarrow$
(k, α) **reads before** (l, β) **in** $\mathsf{h} \rightarrow (k, \alpha)$ **occurs before** (l, β) **in** h
$\wedge\ (k, \alpha)$ **distributes before** (l, β) **in** $\mathsf{h} \rightarrow (k, \alpha)$ **occurs before** (l, β) **in** h

6. $\alpha \in Act_i \wedge \beta \in Act_i \wedge \alpha \equiv \mathsf{W}_i(d, a) \wedge (k, \alpha)$ **occurs before** (l, β) **in** $\mathsf{h} \rightarrow$
$\mathrm{type}(\beta) = \mathsf{W}_i \rightarrow (k, \alpha)$ **distributes before** (l, β) **in** h
$\wedge\ \mathrm{type}(\beta) = \mathsf{R}_i \rightarrow (k, \mathsf{CU}_i(d \star (k \star i), a))$ **occurs before** (l, β) **in** h
$\wedge\ (l, \beta)$ **is completed in** $\mathsf{h} \rightarrow (k, \alpha)$ **is completed in** h

7. $(k, \mathsf{W}_i(d', a'))$ **occurs before** $(l, \mathsf{R}_i(d, a))$ **in** h
$\wedge\ (l, \mathsf{R}_i(d, a))$ **reads before** $(m, \mathsf{W}_j(\bar{d}, a))$ **in** $\mathsf{h} \rightarrow$
$(k, \mathsf{W}_i(d', a'))$ **distributes before** $(m, \mathsf{W}_j(\bar{d}, a))$ **in** h

Proof. We shall not give completely formal proofs here.

(1) Every value needs to be written; remember the convention to prepend histories with virtual $(0, \mathsf{W}_0(0, a)$-actions.

(2) This is a consequence of the fairness constraint on M^T_{distr} and the fact that MW_i and CU_i-events are enabled as long as Out_i and In_i are non-empty.

(3) This follows from the unique tagging of the data being written

(4) Follows from the fact that (d, a) enters queue In_f before (d', a') does.

(5) Let $\alpha \equiv \mathsf{R}_i(d, a)$, $\beta \equiv \mathsf{W}_i(d', a)$ and let (m, γ) **is read by** (k, α) **in** h with $\gamma \equiv \mathsf{W}_j(d, a)$. Since $(m, \mathsf{W}_j(d, a))$ **distributes before** $(l, \mathsf{W}_i(d', a))$ **in** h by definition of **reads before**, $(l, \mathsf{MW}_i(d' \star (m \star i), a))$ **occurs before** (k, α) **in** h would entail that *not* (m, γ) **is read by** (k, α) **in** h holds: α becomes enabled only after Out_i is flushed and In_i does not contain any $*$-ed entries but $(d' \star (l \star i), a, *)$ enters In_i after $(d \star (m \star j), a)$ does. The second implication is proven analogously

(6) Follows from the fact that W_i events are queued in Out_i and that a subsequent R_i event flushes the Out_i queue and the $*$-ed entries in the In_i queue as well (; remember that a W_i event eventually contributes a $*$-ed entry to In_i).

(7) Let (†) $(n, \mathsf{W}_r(d, a))$ **is read by** $(l, \mathsf{R}_i(d, a))$ **in** h. By definition of **reads before** we have $(n, \mathsf{W}_r(d, a))$ **distributes before** $(m, \mathsf{W}_j(\bar{d}, a))$ **in** h. If the consequent is false then we also have $(m, \mathsf{W}_j(\bar{d}, a))$ **distributes before** $(k, \mathsf{W}_i(d', a'))$ **in** h. We obtain $(n', \mathsf{CU}_i(d \star (n \star r), a))$ **occurs before** $(m', \mathsf{CU}_i(\bar{d} \star (m \star j), a))$ **in** hand $(m', \mathsf{CU}_i(\bar{d} \star (m \star j), a))$ **occurs before** $(k', \mathsf{CU}_i(d' \star (k \star i), a'))$ **in** h. As $\mathsf{W}_i(d', a')$ and $\mathsf{R}_i(d, a)$ both originate in the same process, we must have $(k', \mathsf{CU}_i(d' \star (k \star i), a'))$ **occurs before** $(l, \mathsf{R}_i(d, a))$ **in** h. This contradicts (†) since this CU_i-event processes a $*$-ed entry in In_i. □

Now we can define the dag and the simulation relation based on it:

Dag $<_h$

Define the dag $<_h$ on the set

$$\{(k,\alpha) \mid k > 0,\ \alpha \text{ is the k-th type}(\alpha)\text{-event in } h\} \cup \{\bot\}$$

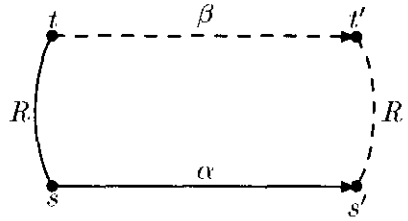
as the smallest relation satisfying

1. if $\alpha, \beta \in Act_i$ and (k,α) **occurs before** (l,β) **in** h then $(k,\alpha) <_h (l,\beta)$
2. if (k,α) **is read by** (l,β) **in** h then $(k,\alpha) <_h (l,\beta)$
3. if (k,α) **reads before** (l,β) **in** h then $(k,\alpha) <_h (l,\beta)$
4. if (k,α) **distributes before** (l,β) **in** h then $(k,\alpha) <_h (l,\beta)$
5. if not (k,α) **is completed in** h then $\bot <_h (k,\alpha)$

Here, we write $u <_h v$ to indicate that the dag has an edge from u to v.

Simulation R

The simulation relation R is inductively defined as the smallest relation that includes the pairs (s,t) for initial states s of M^T_{distr} and t of M_{serial} and that satisfies for all $(s,t) \in R$ and $s \xrightarrow{\alpha} s'$ that there is a state t' and an event β such that $(s',t') \in R$ and $t \xrightarrow{\beta} t'$ subject to the following constraint:



Let $T = \min(<_{s'(\mathsf{h})} \restriction \mathcal{R}_{s',t})$ with
$\mathcal{R}_{s',t} = \{(k,\alpha) \mid (k,\alpha)$ **is ready in** $s'(\mathsf{h}), t(\mathsf{h})\} \cup \{\bot\}$.
If $T \cap Act = \emptyset$ then $\beta = \tau$ else there is an l such that $(l,\beta) \in T$. Moreover, if α **completes** (n,γ) **in** $s(\mathsf{h})$ and $\gamma \in Act_i$ then $(l,\beta) \in T \cap Act_i$

So, M_{serial} executes an action that is minimal in the dag determined by $s'(\mathsf{h})$ from which all events that have already occured in $t(\mathsf{h})$ are removed. To ensure that M_{serial} executes actions from every P_i, there is the additional constraint that if M_{distr} *completes* a P_i-write action from s then M_{serial} must execute a P_i-action from t. It is only at such points that we can be sure that there is a P_i-action amongst the minimal ones.

Lemma 3.2.3

1. *Let $(s,t) \in R$ and $s \xrightarrow{\alpha} s'$ (in M^T_{distr}). Then for every $\beta \not\equiv \bot$, if $(l,\beta) \in \min(<_{s'(\mathsf{h})} \restriction \mathcal{R}_{s',t})$ for some l, then β is enabled in t*
2. $M^T_{distr} \models \Box((k,\alpha) \in \mathrm{dom}(<_{\mathsf{h}}) \rightarrow \Diamond(k,\alpha) \in \min(<_{\mathsf{h}}))$

We defer the proof of Clause (1); Clause (2) is a direct consequence of Lemma 3.2.2(2) and the fact that each process issues infinitely many writes.

From the inductive definition of R, Lemma 3.2.2(2) and Lemma 3.2.3(1), we immediately conclude that $M^T_{distr} \hookrightarrow_R M_{serial}$; provided we can show that $<_h$ is indeed a dag so that minimal elements always exists.

Theorem 3.2.4 *$<_h$ is a dag.*

The proof is based on a Lemma that relates the ordering of MW-events to the ordering of read and write events.

Write $(k,\alpha) <_h^+ (l,\beta)$ to indicate that the dag $<_h$ admits a path from (k,α) to (l,β).

Lemma 3.2.5 *Let* $(k, \mathsf{W}_i(d,a)) <_h^+ (l,\beta)$.

- *If* $\beta = \mathsf{R}_j(d',a')$ *then* $(k, \mathsf{MW}_i(d\star(k\star i), a)$ **occurs before** (l,β) **in** h
- *If* $\beta = \mathsf{W}_j(d',a')$ *then* $(k, \mathsf{W}_i(d,a))$ **distributes before** $(l, \mathsf{W}_k(d',a'))$ **in** h

Proof. We use induction along a path from $(k, \mathsf{W}_i(d,a))$ to (l,β). Let $\alpha \equiv \mathsf{W}_i(d,a)$ and $\beta \in Act_j$.

First assume that $(k,\alpha) <_h (l,\beta)$. Then either (i) $j = i$ and (k,α) **occurs before** (l,β) **in** h or (ii) $j \neq i$ and (k,α) **is read by** (l,β) **in** h or (k,α) **distributes before** (l,β) **in** h. For case (i) the Lemma follows from Lemma 3.2.2(6). Case (ii) follows immediately from the definitions of the **is read by** and **distributes before** relations.

Next, suppose that $(k,\alpha) <_h^+ (m,\gamma) <_h (l,\beta)$. By induction the Lemma holds for $(k,\alpha) <_h^+ (m,\gamma)$. According to the definition of $<_h$ there are four cases. If $\gamma \notin Act_j$ and (m,γ) **reads before** (l,β) **in** h, then the result follows from Lemma 3.2.2(7). The other cases are as (i) and (ii) above. □

We are ready to show that $<_h$ is a dag

Proof of Theorem 3.2.4. Suppose that $<_h$ admits a cycle. Then, we must have $(\bar{k},\bar{\alpha}) <_h^+ (\bar{l},\bar{\beta})$ and $(\bar{l},\bar{\beta}) <_h^+ (\bar{k},\bar{\alpha})$ for some $\bar{\alpha}$ and $\bar{\beta}$. W.l.o.g., we may assume that $(\bar{k},\bar{\alpha}) <_h (\bar{l},\bar{\beta})$. So, by definition of $<_h$, there must be an $(\bar{m},\bar{\gamma})$ such that $(\bar{l},\bar{\beta}) <_h^+ (\bar{m},\bar{\gamma}) <_h^+ (\bar{k},\bar{\alpha})$ and not $\bar{\alpha},\bar{\beta},\bar{\gamma} \in Act_i$ for some i.

By transitivity of $<_h^+$ this means that we have (A) $(k, \mathsf{W}_i(d,a))$ **is read by** (l,β) **in** h and $(l,\beta) <_h^+ (k, \mathsf{W}_i(d,a))$ or (B) $(k, \mathsf{W}_i(d,a))$ **distributes before** (l,β) **in** h and $(l,\beta) <_h^+ (k, \mathsf{W}_i(d,a))$ or (C) (k,α) **reads before** $(l, \mathsf{W}_i(d,a))$ **in** h and $(l, \mathsf{W}_i(d,a)) <_h^+ (k,\alpha)$ with $\alpha,\beta \notin Act_i$. We immediately obtain $(k, \mathsf{MW}_i(d\star(k\star i), a))$ **occurs before** $(k, \mathsf{W}_i(d,a))$ **in** h for case (A) and by Lemma 3.2.5 $(k, \mathsf{MW}_i(d\star(k\star i), a))$ **occurs before** $(k, \mathsf{MW}_i(d\star(k\star i), a))$ **in** h for case (B) and $(l, \mathsf{MW}_i(d\star(l\star i), a))$ **occurs before** (k,α) **in** h for case (C). The first two cases give immediate contradictions; the last one via Lemma 3.2.2(7) from which we infer that $(l, \mathsf{W}_i(d,a))$ **distributes before** $(l, \mathsf{W}_i(d,a))$ **in** h which is impossible. □

There remains the proof that minimal elements of $<_h$ are always enabled. For this, we need the following two trivial facts about M_{serial}.

1. $\mathsf{W}_i(d,a)$ is enabled in any state,
2. $\mathsf{R}_i(d,a)$ is enabled in state t iff the last write-event in $t(h)$ that referred to location a has the form $\mathsf{W}_j(d,a)$ for some j

Proof of Lemma 3.2.3(1). In the proof we refer to the figure in the definition of the simulation R on Page 37. First observe that $(k,\gamma) \in \mathrm{dom}(<_{s'(\mathsf{h})})$ implies that (k,γ) **occurs in** $s'(\mathsf{h})$ for any (k,γ). This is immediate from the definition of $<_h$.

Since writes are always enabled we may assume that $\beta \equiv \mathsf{R}_j(d,a)$. Now, suppose that β is *not* enabled in t. Then the last write event that referred to location a in $t(\mathsf{h})$ was $\alpha \equiv \mathsf{W}_i(d',a)$ for some i with $d' \neq d$; let this be the $k-th$ W_i-event in $t(\mathsf{h})$. Since (k,α) **occurs in** $t(h)$, we must have (k,α) **is completed in** $s(h)$ by definition of R. As (l,β) **occurs in** $s'(\mathsf{h})$ we have (n,γ) **is read by** (l,β) **in** $s'(\mathsf{h})$ for some n and $\gamma \equiv \mathsf{W}_r(d,a)$ so that $(n,\gamma) <_{s'(\mathsf{h})} (l,\beta)$. Also, since $(l,\beta) \in \min(<_{s'(h)} \lceil \mathsf{R}_{s',t})$ we must have (n,γ) **is completed in** $s(h)$.

Now, if (l,β) **reads before** (k,α) **in** $s'(\mathsf{h})$ then $(l,\beta) <_{s'(\mathsf{h})} (k,\alpha)$, whereas (k,α) **occurs in** $t(\mathsf{h})$ but *not* (l,β) **occurs in** $t(\mathsf{h})$. This contradicts the definition of R.

Hence, since both write actions are completed we must have (k,α) **distributes before** (n,γ) **in** $s'(\mathsf{h})$. We conclude that $(k,\alpha) <_{s'(\mathsf{h})} (n,\gamma)$ so that we cannot have (n,γ) **occurs before** (k,α) **in** $t(\mathsf{h})$. As (k,α) is the last write event referring to location a in $t(\mathsf{h})$, we must have $(n,\gamma) \in \mathrm{dom}(<_{s'(\mathsf{h})} \lceil \mathsf{R}_{s',t})$ so that (l,β) is *not* minimal. Contradiction.

We conclude that α cannot be of this form and, hence, that β is enabled in t. □

3.2.2 Concluding the proof

For the last step of verifying that $(s,t) \in R \Longrightarrow s \times t \models \phi_i$ for every $i = 1 \ldots n$, we need to instantiate the choice functions f_i and define

$$
\begin{array}{rcll}
f_i(h) = h' & & \text{where } h' \text{ is the prefix of } h \text{ such that } |h'| = n_i(h) \text{ with} & \\
 & & n_i \text{ inductively defined by } n_i(\varepsilon) = 0 \text{ and} & \\
n_i(h\alpha) & = & \begin{cases} n_i(h)+1, & \text{if } \alpha \textbf{ completes } (n,\gamma) \textbf{ in } h \wedge \gamma \in Act_i \\ n_i(h), & \text{otherwise} \end{cases} &
\end{array}
$$

So, the length of $f_i(h)$ is the number of completed W_i-events in h. This is the obvious choice because the definition of R guarantees that M_{serial} performs a P_i-action, only in case M_{distr} completes a W_i-action.

By Lemma 3.2.2(3) and the assumption that every processor contributes infinitely many writes, each f_i increases i.o. on M_{distr}.

Now, fix some $i = 1 \ldots n$ and $(s,t) \in R$. As $\phi_i \equiv f_i(\mathsf{h}_c) \lceil i \preceq \mathsf{h}_a \lceil i$, we have to show that $f_i(s(\mathsf{h})) \lceil i \preceq t(\mathsf{h}) \lceil i$.

Since both R and f_i are defined inductively, we prove this inductively.

The base case is clear as then $s(\mathsf{h}) = t(\mathsf{h}) = \varepsilon$ and $n_i(\varepsilon) = 0$.

For the inductive step, we may assume that $f_i(s'(\mathsf{h})) \lceil i \not\preceq t(\mathsf{h}) \lceil i$. We refer to the definition of R on Page 37. Hence, we must have (i) α **completes** (m,γ) **in** $s(\mathsf{h})$ with $\gamma \in Act_i$. Let $f_i(s'(\mathsf{h})) \lceil i = f_i(s(\mathsf{h})) \lceil i \hat{} \delta$ and let δ be the n-th event of type(δ). Then, (ii) (n,δ) **is ready in** $s'(\mathsf{h}), t(\mathsf{h})$ must be true. By induction, it suffices to prove that $\delta \equiv \beta$, where β is the transition taken in t according to R.

From Clause 2 in the definition of $<_{s'(\mathsf{h})}$, we have $\bot \not<_{s'(\mathsf{h})} (m,\gamma)$ so that $T \cap Act_i \neq \emptyset$ by Lemma 3.2.3(1). In fact $|T \cap Act_i| = 1$ as Clause (1) says that $<_{s'(\mathsf{h})}$ extends the ordering on event occurrences induced by $s'(\mathsf{h}) \lceil i$. Because (i) holds, we know from Lemma 3.2.2(6) that for any W_i event δ, if (r,δ) **occurs in** $s(f_i(\mathsf{h}))$ then (r,δ) **is completed in** $s(f_i(\mathsf{h}))$; whence $\bot \not<_{s'(\mathsf{h})} (r,\delta)$. By (ii) and the fact that $\bot$ is never covered by read events, we then have $(n,\delta) \in T \cap Act_i$ and also $\delta \equiv \beta$ since $(l,\beta) \in T \cap Act_i$ for some l by definition of R.

3.3 Conclusions

We have worked out the proof in considerable detail. The proof rule demands that a weak simulation be constructed as the first step. This can be interpreted as defining a scheduler that schedules the appropriate event in M_{serial} for every M_{distr}-event. For verifying sequential consistency this is a quite natural approach because the purpose of the protocol is to ensure that the event sequences that each process engages in can also be obtained from a serial memory. In this respect, there is a correspondence with the verification approach of [ABM93]. An important ingredient of the proof is the 'delayed' checking of sequential consistency of prefixes, which is inherent to our approach to interface refinement. This makes the definition of $<_h$ easier, although a penalty is paid in the form of a slightly more complex proof of Lemma 3.2.3(1). In contrast, the scheduler used in [ABM93] needs to maintain sequential consistency of the complete history instead of (ever longer) prefixes of history.

The actual proof tries to abstract from the details of the protocol. I.e., $<_h$ is defined in terms of some relations on the external behavior of the protocol and the proof is based on a number of correctness properties of the protocol. For the same reason, we have not used auxiliary variables other than for the purpose of making events unique. In fact, we view this proof as a first step towards a proper analysis of sequential consistency: The dag $<_h$ characterizes the constraints that the protocol maintains in order to generate sequentially consistent behavior. However, as is, $<_h$ is defined using internal events of M_{distr}; e.g., the **distributes before** relation refers to MW-events. Accordingly, one might ask for the weakest online[2] scheduler defined in terms of constraints on the external events only that maintains sequential consistency. In fact, we have already obtained more efficient protocols for network based architectures and are generalizing the protocols towards weaker memory models such as release consistency.

Acknowledgments

We thank Ruurd Kuiper for his help and Michael Merritt for posing the problem and for catching the last bug in the definition of $<_h$.

[2] In the sense of only depending on the current state.

Chapter 4

Characterization of a Sequentially Consistent Memory and Verification of a Cache Memory by Abstraction

S.Graf

4.1 Introduction

We propose to verify the distributed cache memory presented in [ABM93] and [Ger95] by using the verification method proposed in [BBLS92, LGS+94, CGL92, Lon93]. This method, based on the principle of abstract interpretation [CC77], proposes to verify a set of ∀CTL* [SG90] formulas on a composed program, as follows: define an appropriate abstract program, obtained compositionally from the given program, and verify the required properties on it. Our way of computing abstract programs is very similar to that proposed in [CGL92, Lon93], but our concept of compositionality is different from that proposed in [Lon93] or in [Pnu85]. We construct a global abstraction of the system by composing abstractions of its components, whereas the other method consists in deducing properties of the composed system from properties of its components. Both approaches are useful, but in the distributed cache memory that we want to verify the global properties cannot be deduced easily from properties of the components. An abstraction of each component is obtained applying the principle of abstract interpretation by means of a relation ϱ relating the domain of its variables and the domain of the set of some abstract variables.

In [GL93, Loi94] is described a tool allowing to verify finite state systems in a fully automatic way by using this method. Here, we show that the same method is also tractable in practice for infinite state systems where a complete automatization is not possible. In fact, if — depending on the formula one wants to verify — for each component P_i one can guess an appropriate abstraction relation ϱ_i verification becomes often a relatively simple task as

- the corresponding finite state abstract program is reasonably easy to obtain,
- the verification of the properties on the obtained abstract program can be done fully automatically.

Despite the fact that ∀CTL* contains also *liveness* properties, this method does in general not support directly the verification of liveness properties as they do not hold on most of the finite abstractions. Here, we verify liveness property of the cache memory by applying the induction rules given in [Pnu85, JPR94] to a set of safety properties.

In Section 4.2, we recall all the ingredients we need for our verification method:

- a simple program formalism similar to that used e. g., in [Pnu86],
- a method to compute abstract programs, consisting in defining for each operator on the concrete domains a corresponding *abstract operator* — the only step in the proposed method that cannot be fully automated,
- the temporal logic CTL* and its fragments, used for the description of properties,
- the preservation results allowing to deduce the validity of a property on the concrete program from its validity on the abstract program and
- the compositionality results allowing to compute an abstract program by composing abstractions of its components.

We illustrate all the definitions and results on a small buffer example. In Section 4.3, we give a set of temporal logic formulas guaranteeing that whenever a system satisfies it, then it is a "sequentially consistent memory" [Lam79]. This set has been chosen in such a way that its satisfaction on a given

program can be deduced from the satisfaction of a finite number of representatives. In Section 4.4, we verify this set of properties on the distributed cache memory system. It turns out that, using our method, this verification is almost as simple as the verification of the tiny buffer, as we need almost the the same abstract operations.

4.2 A verification method using abstraction

4.2.1 A program description formalism

We adopt a simple program formalism which is not meant as a real programming language but which is sufficient to illustrate our method. A complex system is a parallel composition of basic programs of the following form

Name :	P
Variables :	$x_1 : T_1, ..., x_n : T_n$
Transitions :	$(\ell_1)\ action_1(x_1, ..., x_n,\ x'_1, ..., x'_n)$
	...
	$(\ell_p)\ action_p(x_1, ..., x_n,\ x'_1, ..., x'_n)$
Initial States :	$init(x_1, ..., x_n)$

where P is an identifier used to refer to the program in a composition expression, x_i are variables of type T_i and $L_P = \{\ell_1, ..., \ell_n\}$ is the set of program labels. Each $action_i$ is an expression with variables in the set of program variables and a set of primed variables which is a copy of the set of state variables; as in [Pnu86, Lam95], $action_i$ represents a transition relation on the domain of the program variables by interpreting the valuations of $X_P = (x_1, ..., x_n)$ as the state *before*, and the valuations of $X'_P = (x'_1, ..., x'_n)$ as the state *after* the transition. We denote the set of valuations of X_P by $Val(X_P)$.

Semantics : Program P defines in an obvious manner a transition system $S_P = (Q_P, R_P)$ where

- $Q_P = Val(X_P)$ is the set of states,
- $R_P \subseteq Q_P \times Q_P$ is a transition relation defined by $R_P = \{(q, q') \mid \exists i\ .\ action_i(q, q')\}$.

The predicate $init$ defines the set of initial states. It is used in the formulas specifying the program: properties are in general of the form $init \Rightarrow \phi$ where ϕ expresses the property one wants to verify.

We do not distinguish variables representing inputs as they need not be treated in a particular manner. However, we annotate in the programs the variables which are meant as inputs as this makes programs easier to read.

Labels are used to name "events" or "actions". If ℓ_i is a label and (v, v') a pair of valuations such that $action_i(v, v')$ is true, then the transition from state v to state v' is an event ℓ. If e is the valuation of the "input" variables extracted from v, then we call this event also $\ell_i(e)$. Events are used for the expression of properties.

Example 4.2.1 (an infinite lossy buffer) *The following program represents an unbounded buffer taking as input elements* e *of some data domain* elem. *The event* $push(e)$ *enters* e *(if it has never been entered yet) into the buffer or arbitrarily "loses" it, and* $pop(e)$ *takes* e *out of the buffer if it is its first element.*

Name :	lossy buffer	
Variables :	$e : elem$ (Input)	
	E : *set* of $elem$ (already occurred events $push(e)$)	
	$B : buffer$ of $elem$	
Transitions :	(push(e))	$allowed(e, E, E') \wedge (append(B, e, B') \vee \mathbf{unch}(B))$
	(pop(e))	$first(B, e) \wedge tail(B, e, B') \wedge \mathbf{unch}(E)$
Initial States :	$empty(B)$	

E contains the elements e such that $push(e)$ has already occurred, and $\mathbf{allowed}(e, E, E')$ is necessarily *false* if $e \in E$. All other predicates have the intuitive meanings: $\mathbf{append}(B, e, B')$ holds if B' is obtained by appending element e at the end of the buffer B; $\mathbf{tail}(B, e, B')$ holds if B' is obtained by eliminating e from B if e is its first element ($\mathbf{first}(B, e)$ is *true*) otherwise $B' = B$; $\mathbf{empty}(B)$ is *true* if B is the empty buffer. $\mathbf{unch}(X)$, where $X = x_1, ...x_n$ is a tuple of program variables, represents the transition relation which lets all variables in X unchanged, i. e., $\mathbf{unch}(X) = \bigwedge_i (x_i' = x_i)$.

We use predicates of the form $append(B, e, B')$ instead of $B' = Append(B, e)$ where $Append$ is a function, as abstract operations are in general nondeterministic. This is also the way of representing operations which is proposed, e. g. in [CGL92, Lam95].

Composed programs : In [GL93] we obtain our results for more general parallel composition operators, but here we need only asynchronous composition. If P_1 and P_2 are programs defined on a tuple of state variables X_1, respectively X_2, then $P_1 \parallel P_2$ is the parallel composition of P_1 and P_2 defining the transition system $S = (Val(X_1 \cup X_2), R)$ where

$$R = R_{P_1} \wedge \mathbf{unch}(X_2 - X_1) \vee R_{P_2} \wedge \mathbf{unch}(X_1 - X_2)$$

Each transition of $P_1 \parallel P_2$ is either a transition of P_1 which leaves all variables which are declared in P_2 but not in P_1 unchanged or the other way round.

4.2.2 Abstract programs

As proposed in [CGL92, LGS+94], given a program P and a predicate ϱ on the variables of P and a tuple of *abstract variables* $X^A = (x_1^A, ...x_m^A)$, representing a relation between the concrete and the abstract domain (a function in [CGL92]), then an *abstraction of* P is a program P^A defined on X^A that can "simulate" any transition of P:

Definition 4.2.2 *(Abstract programs)*
Let P be a program on variables X and ϱ a predicate on X and a tuple $X^A = (x_1^A, ...x_m^A)$ of abstract variables*; then, any program P^A defined on X^A, such that for each action action of P there exists an action $action^A$ of P^A with the same label, such that*

$$\exists X \exists X' \,.\, \varrho(X, X^A) \wedge \varrho(X', X^{A'}) \wedge action(X, X') \;\Rightarrow\; action^A(X^A, X^{A'}) \text{ (1) and}$$
$$\exists X \,.\, \varrho(X, X^A) \wedge init(X) \;\Rightarrow\; init^A(X^A)$$

is an abstraction *or more precisely a* ϱ-abstraction *of P.*

When verifying composed programs, it is interesting to compute an abstract program compositionally, i. e., by composing abstract component programs. From a more general result given in [LGS+94], we obtain the following result which is sufficient for the verification of the distributed cache memory system.

Proposition 4.2.3 *Let P_1 and P_2 be programs and ϱ_i* total functions *from the domain of the variables of P_i into some abstract domains such that $\varrho_1 \cap \varrho_2$ is total and P_1^A, P_2^A are ϱ_i-abstractions of P_i, then $P_1^A \parallel P_2^A$ is a $(\varrho_1 \cap \varrho_2)$-abstraction of $P_1 \parallel P_2$.*

Computation of abstract programs in practice : The idea of abstract interpretation [CC77] is to replace every *function* on the concrete domain used in the program by a corresponding *abstract function* on the abstract domain, and then to analyze the so obtained simpler abstract program instead of the concrete one. Consider the program $Prog^A$ obtained by replacing every basic predicate $op(X, X')$ (such as $tail$, $first$,...) on the concrete variables by a predicate $op^A(X^A, X^{A'})$ on the abstract variables is a ϱ-abstraction of *Prog* if, instead of (1), for every basic operation

$$\exists X \exists X' \,.\, \varrho(X, X^A) \wedge \varrho(X', X^{A'}) \wedge op(X, X') \;\Rightarrow\; op^A(X^A, X^{A'}) \qquad (2)$$

holds. If the expressions in *Prog* are negation free (as in our buffer), then $Prog^A$ is in fact a ϱ-abstraction of *Prog*. The definition of abstract predicates op^A is the only part of our verification method which cannot be fully automatized. But as we will see, we only need a restricted number of such abstract operations in order to verify a whole class of programs. For example, in the domain of protocol verification, the used data structures are "messages" on which no operations are carried out, "memories" or "registers" in which data can be stored, integers which are mostly used as counters and "buffers" with the usual operations $append$, $tail$, $first$,..., as in our examples.

Example 4.2.4 (An abstract lossy buffer) *To illustrate the idea, consider again the buffer of Example 4.2.1. In order to show that the buffer has the property of "order preservation" (see Example 4.2.9), it is sufficient to show that the order of any pair of elements $(\mathbf{e}_1, \mathbf{e}_2) \in elem \times elem$ is preserved. All the information we need about the content of the buffer B is, if and in which order, it contains the elements $\mathbf{e}_1$ and $\mathbf{e}_2$. Furthermore, as each element is supposed to be put into the buffer at most once, we need not distinguish amongst the valuations of B containing $\mathbf{e}_i$ more than once. Similarly, for the input variable e we only need to distinguish if its value is $\mathbf{e}_1$, $\mathbf{e}_2$ or any other value. Concerning the value of E determinating which events $push(e)$ are still allowed, we only need to know if the event $push(\mathbf{e}_1)$, respectively $push(\mathbf{e}_2)$ is still possible or not. This leads us naturally to the abstract domain defined by the abstract variables,*

$$e_A : elem_A^2 = \{0, 1, 2\}$$
$$E_A : set\ of\ elem_A^2$$
$$B_A : buffer_A^2 = \{\epsilon, \mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_1 \bullet \mathbf{e}_2, \mathbf{e}_2 \bullet \mathbf{e}_1, \bot\}$$

and the following abstraction relation ϱ^2 defining the correspondence between the concrete and the abstract variables

$$\varrho^2(e, E, B,\ e_A, E_A, B_A) = \varrho^2_{elem}(e, e_A) \wedge \varrho^2_{set\_of\_elem}(E, E_A) \wedge \varrho^2_{buffer}(B, B_A)$$

where for $c : elem$ and $c_A : elem_A^2$

$$\varrho^2_{elem}(e, e_A) = ((e_A = 0) \equiv (e \notin \{\mathbf{e}_1, \mathbf{e}_2\})) \wedge ((e_A = 1) \equiv (e = \mathbf{e}_1)) \wedge ((e_A = 2) \equiv (e = \mathbf{e}_2))$$

for $E : set\ of\ elem$ and $E_A : set\ of\ elem_A^2$,

$$\varrho^2_{set\_of\_elem}(E, E_A) = (1 \in E_A) \equiv (\mathbf{e}_1 \in E) \wedge (2 \in E_A) \equiv (\mathbf{e}_2 \in E)$$

and for $B : buffer\ of\ elem$ and $B_A : buffer_A^2$

$$\begin{aligned}\varrho^2_{buffer}(B, B_A) = &((B_A = \epsilon) \equiv empty(B_{|\{\mathbf{e}_1, \mathbf{e}_2\}})) \wedge \\ &((B_A = \mathbf{e}_1) \equiv (B_{|\{\mathbf{e}_1, \mathbf{e}_2\}} = \mathbf{e}_1)) \wedge \\ &((B_A = \mathbf{e}_2) \equiv (B_{|\{\mathbf{e}_1, \mathbf{e}_2\}} = \mathbf{e}_2)) \wedge \\ &((B_A = \mathbf{e}_1 \bullet \mathbf{e}_2) \equiv (B_{|\{\mathbf{e}_1, \mathbf{e}_2\}} = \mathbf{e}_1 \bullet \mathbf{e}_2)) \wedge \\ &((B_A = \mathbf{e}_2 \bullet \mathbf{e}_1) \equiv (B_{|\{\mathbf{e}_1, \mathbf{e}_2\}} = \mathbf{e}_2 \bullet \mathbf{e}_1)) \wedge \\ &((B_A = \bot)\ in\ all\ other\ cases\ ))\end{aligned}$$

where $B_{|\{\mathbf{e}_1,\mathbf{e}_2\}}$ is the buffer B restricted to the elements $\mathbf{e}_1$ and $\mathbf{e}_2$. In order to construct an abstract program, we have to define abstract predicates for all the basic predicates used in the concrete buffer program, such as allowed, append, tail, **unch**, *etc.*

In the case that every abstract variable is related to a single concrete variable, the abstract predicate associated with **unch**(v) *is obviously* **unch**(v_A) *for any abstract variable v_A related to v. The following abstract predicates satisfy the condition (2).*

$$\begin{aligned} allowed^2_A(e_A, E_A, E_A') = &(E_A' \equiv E_A) \ \wedge \ (e_A = 0) \ \vee \\ &(1 \notin E_A) \wedge (1 \in E_A') \ \wedge \ (e_A = 1) \ \vee \\ &(2 \notin E_A) \wedge (2 \in E_A') \ \wedge \ (e_A = 2) \end{aligned}$$

$$\begin{aligned} append^2_A(B_A, e_A, B_A') = &(B_A = B_A') \qquad\qquad \wedge (e_A = 0) \vee \\ &(B_A \in \{\epsilon, \mathbf{e}_2\}) \wedge (B_A' = \mathbf{e}_1 \bullet B_A) \vee (B_A \notin \{\epsilon, \mathbf{e}_2\}) \wedge (B_A' = \bot) \wedge (e_A = 1) \vee \\ &(B_A \in \{\epsilon, \mathbf{e}_1\}) \wedge (B_A' = \mathbf{e}_2 \bullet B_A) \vee (B_A \notin \{\epsilon, \mathbf{e}_1\}) \wedge (B_A' = \bot)) \wedge (e_A = 2) \end{aligned}$$

$$\begin{aligned} tail^2_A(B_A, e_A, B_A') = &(B_A = B_A') \qquad\qquad\qquad\qquad \wedge (e_A = 0) \vee \\ &((B_A \in \{\mathbf{e}_1, \mathbf{e}_1 \bullet \mathbf{e}_2\}) \ \Rightarrow \ (B_A = B_A' \bullet \mathbf{e}_1)) \wedge (e_A = 1) \vee \\ &((B_A \in \{\mathbf{e}_2, \mathbf{e}_2 \bullet \mathbf{e}_1\}) \ \Rightarrow \ (B_A = B_A' \bullet \mathbf{e}_2)) \wedge (e_A = 2) \end{aligned}$$

$$empty^2_A(B_A) = (B_A = \epsilon)$$

$$\begin{aligned} first^2_A(B_A, e_A) = &(e_A = 0) \vee \\ &(B_A \in \{\mathbf{e}_1, \mathbf{e}_1 \bullet \mathbf{e}_2, \bot\}) \wedge (e_A = 1) \vee \\ &(B_A \in \{\mathbf{e}_2, \bot\}) \wedge (e_A = 2) \end{aligned}$$

$tail$ is an example of a predicate defining a function on the concrete domain, but which is nondeterministic on the given abstract domain; $tail^2_A(\bot, 1, B_A')$ necessarily evaluates to true *for any value of B_A' (the value of the buffer in the next state).*

Using these abstract predicates, the definition of a program representing a ϱ-abstraction of the buffer program becomes trivial. We just replace variables by corresponding abstract variables and every occurrence of a predicate by corresponding abstract one. The resulting abstract program looks almost as the concrete program but defines a very small finite transition system.

Name :	*Abstract lossy buffer*	
Variables :	$e_A : elem^2_A$ *(input)*	
	$E_A : set\ of\ elem^2_A$	
	$B_A : buffer^2_A$	
Transitions:	$(push(e_A))$	$allowed^2_A(e_A, E_A, E_A') \wedge append^2_A(B_A, e_A, B_A')$
	$(pop(e_A))$	$first^2_A(B_A, e_A) \wedge tail^2_A(B_A, e_A, B_A') \wedge$ **unch**(E_A)
	$(lose(e_A))$	$in^2_A(B_A, e_A) \wedge delete^2_A(B_A, e_A, B_A') \wedge$ **unch**(E_A)
Init :	$empty^2_A(B_A)$ *(the translation of the concrete initial predicate)*	

The useful abstractions are often obtained by using this kind of abstract domains. Here, we gave in detail the more complicated abstraction of a buffer particularizing two different data elements. But often, it is sufficient to particularize in the same way a single data element. The corresponding abstraction relations ϱ^1_{elem}, $\varrho^1_{set\_of\_elem}$, ϱ^1_{buffer} and abstract predicates $allowed^1_A$, $append^1_A$, $tail^1_A$,... can be defined by simplifying the above definitions in an obvious manner. For the verification of the cache memory we use also existential abstractions of buffers. The corresponding abstract predicates $append^{ex}(e_A)$, $tail^{ex}(e_A)$,... evaluate to *true* if e_A is an allowed value of the existentially abstracted buffer. In [CGL92] a similar method is proposed and in [Lon93] some "standard" abstractions are proposed for bounded integers and operations on them.

4.2.3 Temporal Logic

It remains to recall the definition of temporal logic. Here we restrict ourselves to subsets of CTL* [EH83] for the expression of properties. The preservation results in [LGS+94] are given for subsets of the more powerful branching time μ-calculus [Koz83] augmented by past time modalities. μ-calculus and CTL* can express both branching time and linear time properties; μ-calculus by using nested fixed points and CTL* by using explicitly state *and* path formulas. Our tool presented in [GL93, Loi94] only deals with state formulas; however formulas with nested fixed points are in general not very intuitive, so we prefer here for readability reasons to stick to CTL* even if we lose some of the expressive power.

Definition 4.2.5 CTL* *is the set of state formulas given by the following definition.*

1. *Let $\mathcal{P}$ be a set of atomic (a) state respectively (b) path formulas.*
2. *If ϕ and ψ are (a) state respectively (b) path formulas then $\phi \wedge \psi$, $\phi \vee \psi$ and $\neg\phi$ are (a) state respectively (b) path formulas.*
3. *If ϕ is a path formula then $\mathbf{A}\phi$ and $\mathbf{E}\phi$ are state formulas.*
4. *If ϕ and ψ are (a) state or (b) path formulas then $\mathbf{X}\phi$, $\phi\mathbf{U}\psi$ and $\phi\mathbf{W}\psi$ are path formulas.*

U is a "strong until" and **W** a "weak until" operator, a sequence satisfies $\phi\mathbf{W}\psi$ if ϕ holds as long no state satisfying ψ has been encountered, and $\phi\mathbf{U}\psi$ expresses the same property and moreover the obligation that such a state satisfying ψ exists. That means that **U** and **W** are related as follows: $\phi\mathbf{W}\psi = \neg(\neg\psi\mathbf{U}\neg(\phi \vee \psi))$ and $\phi\mathbf{W}\psi = (\phi\mathbf{U}\psi) \vee (\mathbf{G}\phi)$, where, as usual, we use also the abbreviations $\phi_1 \Rightarrow \phi_2$ denoting implication, $\mathbf{F}\phi$ denoting $true\mathbf{U}\phi$ (expressing "eventually" ϕ) and $\mathbf{G}\phi$ denoting $\phi\mathbf{W}false$ (expressing "always" ϕ).

CTL is the subset of CTL* obtained by allowing in all rules only the choice (a) whereas PTL is the subset obtained by allowing only the choice (b) and restricting Rule 3 by allowing only the path quantifier **A**. $\forall$CTL and $\forall$CTL* [SG90] are the subsets of CTL respectively CTL* obtained by allowing negations only on atomic formulas and restricting Rule 3 by allowing only the universal path quantifier **A**; that means that PTL is contained in $\forall$CTL*.

The *semantics* of CTL* is defined over Kripke structures of the form $M = (S, \mathcal{I})$ where $S = (Q, R)$ is a transition system and $\mathcal{I}$ is a interpretation function mapping elements of $\mathcal{P}$ into sets of states of S.

Definition 4.2.6 *A* path *in a transition system $S = (Q, R)$ is an infinite sequence $\pi = q_0 q_1 \ldots$ such that for every $i \in \mathcal{N}$. $R(q_i, q_{i+1})$. We denote by π_n the nth state of path π and by π^n the sub-path of π starting in π_n.*

Definition 4.2.7 *Let be $M = (S, \mathcal{I})$ a Kripke structure, $q \in Q$ and π a path of M. Then the satisfaction ($\models_M$) of* CTL* *formulas on M is defined inductively as follows.*

1. *Let be $p \in \mathcal{P}$. Then,*
   $q \models_M p$ *iff* $q \in \mathcal{I}(p)$ *and* $\pi \models_M p$ *iff* $\pi_0 \in \mathcal{I}(p)$.
2. *Let ϕ and ψ be (a) state respectively (b) path formulas. Then,*
   (a) $q \models_M \neg\phi$ *iff* $q \not\models_M \phi$, $q \models_M \phi \wedge \psi$ *iff* $q \models_M \phi$ *and* $q \models_M \psi$, $q \models_M \phi \vee \psi$ *iff* $q \models_M \phi$ *or* $q \models_M \psi$.
   (b) *analogous by replacing q by π*

3. *Let ϕ be a path formula. Then,*
   $q \models_M \mathbf{A}\phi$ *iff for every path* π *starting in* q, $\pi \models_M \phi$
   $q \models_M \mathbf{E}\phi$ *iff there exists a path* π *starting in* q *such that* $\pi \models_M \phi$.

4. *Let ϕ and ψ be (a) state respectively (b) path formulas. Then,*

   (a) $\pi \models_M \mathbf{X}\phi$ *iff* $\pi_1 \models_M \phi$,
   $\pi \models_M \phi \mathbf{U} \psi$ *iff* $\exists n \in \mathcal{N} \,.\, (\pi_n \models_M \psi$ *and* $\forall k < n \,.\, \pi_k \models_M \phi)$,
   $\pi \models_M \phi \mathbf{W} \psi$ *iff* $\forall n \in \mathcal{N} \,.\, ((\forall k \leq n \,.\, \pi_k \models_M \neg\psi)$ *implies* $\pi_n \models_M \phi)$.

   (b) the same definition obtained by replacing in (a) all states π_i *by subsequences* π^i.

We say that $M \models \phi$ if and only if $q \models_M \phi$ for all states of M.

From the more general results given in [LGS+94] we obtain the following proposition concerning preservation of properties of ∀CTL*.

Proposition 4.2.8 *(Preservation of* ∀CTL*)*
Let Prog *be a program, ϱ a total relation from the domain of* Prog *into some abstract domain, and* Prog_A *a ϱ-abstraction of* Prog. *Then, for any $\phi \in$* ∀CTL*, $\mathcal{P}$ the set of atomic formulas occurring in ϕ and $\mathcal{I}$ an interpretation function mapping $\mathcal{P}$ into sets of states of S_{Prog}, we have*

$$Im[\varrho^{-1}] \circ Im[\varrho] \circ \mathcal{I}(p) \subseteq \mathcal{I}(p) \;\; (*) \text{ for all } p \in \mathcal{P} \text{ occurring non negated in } \phi$$
implies
$$(S_{\text{Prog}_A}, Im[\varrho] \circ \mathcal{I}) \models \phi \quad \Rightarrow \quad (S_{\text{Prog}}, \mathcal{I}) \models \phi$$

where $Im[\varrho]$ is the image function of ϱ. Condition (*) is called consistency *of ϱ with $\mathcal{I}(p)$.*

This proposition expresses that, if $\phi \in$ ∀CTL* holds on a ϱ-abstraction of the program *Prog* by translating the interpretations of all atomic propositions occurring in the formula by $Im(\varrho)$ into predicates on the abstract domain, and if all these predicates are consistent with ϱ, then we can deduce that ϕ holds on *Prog*. Consistency is not needed for predicates that occur only negated in ϕ as $Im[\varrho^{-1}](\overline{Im[\varrho](\mathcal{I}(p))}) \subseteq \overline{\mathcal{I}(p)}$. We conclude that, if ϕ holds on $Prog_A$ using the abstract interpretation $\overline{Im[\varrho](\mathcal{I}(p))}$ of $\neg p$, then a stronger property than ϕ using the concrete interpretation $\overline{\mathcal{I}(p)}$ of $\neg p$ holds on *Prog*. In particular, for the verification of a formula of the form $init \Rightarrow \phi$, $init$ need not to be consistent with ϱ.

Example 4.2.9 *For a buffer, the property of* order preservation *— that means the fact elements are taken out in the same order in which they are put into the buffer — can be expressed on the set of "observable" atomic predicates*

$$\mathcal{P} = \{init, enable(push(e)), after(push(e)), enable(pop(e)), after(pop(e)), ...\},$$

by the following parameterized formula — that is a CTL* *formula containing globally universally quantified rigid variables:*

$$\forall e', e \in elem \;:\; init \;\Rightarrow\; \mathbf{A}(\; [\neg after(push(e)) \;\mathbf{W}\; after(push(e'))] \;\Rightarrow\; [\neg enable(pop(e)) \;\mathbf{W}\; after(pop(e'))]\;)$$

This formula can be transformed into a ∀CTL formula in which only the predicates $after(push(e))$ and $after(pop(e'))$ occur non negated. The transformation into an ∀CTL* formula is immediate, due to the fact that for every operator exists a dual one; in order to see that they are also in ∀CTL one can use a result given in [EH83]. In order to verify that the buffer of Example 4.2.1 has the property of order preservation, it is sufficient to verify the formula obtained by instanciating $\mathbf{e}_1$ for e and $\mathbf{e}_2$ for

e' on the small transition system associated with the abstract program defined in Example 4.2.4. In fact, as e_1 and e_2 represent an arbitrary pair of data values, this verification of a single representative of the set of formulas is sufficient. It remains to give the interpretations of the atomic propositions: $enable(\ell)$ is interpreted as the set of states in which event ℓ is possible, a predicate that is expressed by $\exists X' \,.\, action_\ell(X, X')$ if ℓ is just a label and by $\exists X' \,.\, action_l(X, X')[e/x]$ if $\ell = l(e)$ where l is a label and e a valuation of input variables x. $after(\ell)$ is interpreted as the set of states in which ℓ has just occurred; in order to make this predicate expressible, we introduce an explicit boolean program variable $after\_\ell$ for every proposition $after(\ell)$ occurring in the formula under consideration which is set to *true* exactly after any event ℓ and to *false* after all others. The so obtained program is equivalent to the original one as the values of the original variables do not depend of this new variable; this means that $after\_\ell$ is added by superposition [CM88].

Now, it is easy to obtain the consistency of predicates of the form $after(\ell)$ by not abstracting the variable $after\_\ell$. In the sequel, we suppose that for every predicate $after(\ell)$ occurring in the considered formula such a variable is defined, but we do not mention it explicitly in order to keep the programs simple.

4.3 Abstract specification of a sequentially consistent memory

If we want to use our method in order to verify that the distributed cache memory defined in [ABM93] is a "sequentially consistent memory" [Lam79], we must give a temporal logic characterization of this property. A system with observable events of the form $read_i(a, d)$ and $write_i(a, d)$ — where the index i determines the process P_i performing the event, a is a memory location and d a data element — is a sequentially consistent memory if any of its computation sequences projected on observable events can be reordered — by respecting the order of the events with the same indices — into a sequence of a central memory — that means a sequence in which $read_i(a, d)$ is only possible if the last write event concerning location a is of the form $write_j(a, d)$ for some j.

For the exact characterization of this property one needs full first order temporal logic, whereas we want to restrict ourselves to a set of propositional but parameterized formulas in order to be able to evaluate them by a model checking tool on a finite abstract model. Therefore, our characterization is necessarily stronger than required. In order to be able to give a convenient set of formulas we need the assumption — which can be made without loss of generality — that every pair of the form (a, d) can occur at most once as the parameter of some $write$ event. This is still not sufficient in order to express these requirements in terms the "observable" atomic propositions of the form $enable(\ell)$ and $after(\ell)$ for observable events ℓ and the predicate $init$. However, suppose we can identify auxiliary predicates $tia_i(a, d)$ expressing "$write(a, d)$ has been taken into account by process P_i" which is weaker than $enable(read_i(a, d))$ but such that each $write(a, d)$ is necessarily followed by $tia_i(a, d)$ in all processes and from that moment on until $tia_i(a, d)$ becomes "*false* forever" nothing else than d can be read on address a by process P_i. Then, the expression of "sequentially consistent with a memory" becomes possible. We have to express that elements written by the same process are taken into account in the same order by all processes; and for any two elements, even if they are not written by the same process, they are taken into account in all processes in the same order.

Proposition 4.3.1 *(Properties guaranteeing sequential consistency)*
Let be S a transition system and the set of predicates

$$\mathcal{P} = \{init, enable(read_i(a, d)), after(read_i(a, d)), enable(write_i(a, d)), after(write_i(a, d))\}_{i:index,(a,d):address \times datum}$$

with the interpretation function $\mathcal{I}$ defined in the previous section. If one can define an interpretation function $\mathcal{I}_{aux}$ for the set of predicates

$$\mathcal{P}_{aux} = \{tia_i(a,d))\}_{i:index,(a,d):address\times datum}$$

such that $M = (S, \mathcal{I} \cup \mathcal{I}_{aux})$ satisfies the following set of properties, then the program generating model M is a sequentially consistent memory.

(C1) $\forall (a,d) : address \times datum, i : index$
$init \Rightarrow \mathbf{AG}(enable(read_i(a,d)) \Rightarrow tia_i(a,d))$

(C2) $\forall (a,d),(a,d') : address \times datum \,.\, d \neq d', i : index$
$init \Rightarrow \mathbf{AG}(tia_i(a,d) \Rightarrow \mathbf{A}[\neg tia_i(a,d') \,\mathbf{W}\, \mathbf{AG}(\neg tia_i(a,d))])$

(C3) $\forall (a,d) : address \times datum, i,j : index$
$init \Rightarrow \mathbf{AG}[after(write_j(a,d)) \Rightarrow \mathbf{AF}(tia_i(a,d))]$

(S1) $\forall (a,d) : address \times datum, i : index$
$init \Rightarrow \mathbf{AG}[\, after(write_i(a,d)) \Rightarrow \mathbf{A}(\neg enable(read_i) \,\mathbf{W}\, tia_i(a,d))]$

(S2) $\forall (a,d) : address \times datum, i : index$
$init \Rightarrow \mathbf{A}(\neg tia_i(a,d) \,\mathbf{W}\, \bigvee_{j:index} after(write_j(a,d)))$

(S3) $\forall (a,d),(a',d') : address \times datum \,.\, d \neq d', i,j : index$
$init \Rightarrow \mathbf{A}([\neg after(write_j(a,d)) \,\mathbf{W}\, after(write_j(a',d'))] \Rightarrow$
$[\neg tia_i(a,d) \,\mathbf{W}\, tia_i(a',d')])$

(S4) $\forall (a,d),(a',d') : address \times datum \,.\, d \neq d', i,j : index$
$init \Rightarrow \mathbf{A}([\neg tia_i(a,d) \,\mathbf{W}\, tia_i(a',d')] \Rightarrow$
$[\neg tia_j(a,d) \,\mathbf{W}\, tia_j(a',d')])$

Properties (C1) to (C3) express the before mentioned conditions on the auxiliary predicates $tia_i(a,d)$.

Property (S1) expresses the requirement that in every process P_i as soon as an event $write_i(a,d)$ has occurred, nothing can be read anymore until this $write$ event has been taken into account. This requirement looks very strong. However, the weaker and more intuive requirement that, after $write_i(a,d)$ only events $read_i(a)$ are forbidden until (a,d) has been taken into account in process P_i is not sufficient: Suppose that process P_1 reads (a,d_1), then (a',d'_1), then (a,d_2) and then (a',d'_2) which guarantees by (S4) that (a,d_1) is taken into account before (a,d_2) in all processes an analogously for the primed pairs. If in process P_2, $write_2(a,d_2)$ is followed by $read_2(a',d'_1)$ and in process P_3, $write_3(a',d'_2)$ is followed by $read_3(a,d_1)$, then these sequences cannot be merged and completed into a sequence of a central memory, but it may satisfy all the above properties when (S1) is replaced by the proposed weaker property.

Property (S2) guaranties that every (a,d) is not taken into account in any process P_i before $write_j(a,d)$ has occurred for some j. This property could be weakened; what we need to express is that every pair (a,d) that is taken into account must be written by some process P_j, and only in P_j it is necessarily written before it is taken into account.

Property (S3) guarantees that the $write$ events of process P_j are taken into account by any process P_i in the order in which they occurred: whenever (a',d') is written before (a,d) by process P_j, then (a,d) is not taken into account by process P_i before (a',d') has already been taken into account.

Property (S4) expresses that in any pair of processes the $write$ events are taken into account in a compatible order.

Both (S3) and (S4) have the intended meaning only because of (C3). For example an execution sequence, in which process P_1 reads (a', d'_1) then (a, d_1) and then (a, d_2), process P_2 reads (a', d'_1) then (a', d'_2) and then (a, d_1), and process P_3 reads (a, d_2) and then (a', d'_1), can obviously not be merged and completed to a sequence of a central memory, but may be completed to a sequence which satisfies all the above properties except (C3), because, if in every process the pairs (a, d) which are not effectively read are not taken into acount, it is possible to obtain pairwise compatibility of the order in which $write$ events are taken into account but this does not imply global compatibility. In order to obtain global compatibility, we add the requirement expressed by formula (C3) that all $write$ events are taken into account in all processes. Obviously, one could weaken (C3) by requiring only that every $write$ event is either taken into account by all processes or by none of them.

Now a few remarks concerning the choice of appropriate predicates $tia_i(a, d)$: in case of a central memory, $tia_i(a, d)$ can obviously be chosen $enable(read_i(a, d)$ (i. e., $\mathrm{M}[a] = d$). We will show that the distributed memory system that we want to verify satisfies the set of properties given above if we choose $tia_i(a, d)$ to be $\mathrm{C}_i[a] = d$ where C_i is the cache memory of process P_i; for this choice, the condition (C1) is trivially satisfied.

All the above formulas can be translated into $\forall$CTL formulas. Notice that despite of the fact that the original abstract specification does not contain an liveness condition, we need the liveness property (C3) in order this characterization to be sufficient.

Proof of Proposition 4.3.1

Remains to show that every system that satisfies the requirement of Proposition 4.3.1 is sequentially consistent. In order to do so, we show that every computation sequence π of a system satisfying this requirement, can be finitely reordered respecting the order of the events of each individual process into a computation sequence of a central memory: build the projections π_i of visible events of each process P_i and the sequence TIA of pairs (a, d) in the order defined by "the first state in which $tia_i(a, d)$ is true" which is the same in all sequences π_i by property (S4). Then, build a sequence π_{seq} of a central memory using the following procedure.

$\pi_{seq} := \epsilon$; $\forall a : address\,.\, lw(a) := \epsilon$; $nw := first(TIA)$;
$b := true$;
while b do
    $b := false$;
    for $i : index$
        if $\exists a\,.\, first(\pi_i) = r(a, lw(a))$ then ($add(\pi_{seq}, first(\pi_i))$; $b := true$; $tail(\pi_i)$));
        if $first(\pi_i) = w(nw) \wedge nw = (a, d) \wedge \nexists j.r(a, lw(a)) \in \pi_j$, then
            $(add(\pi_{seq}, first(\pi_i))$; $b := true$; $tail(\pi_i)$; $lw(a) := d$; $tail(TIA)$; $nw := first(TIA))$;
    endfor
endwhile
if not $\forall i.empty(\pi_i)$ then "error state";

During the whole execution, $lw(a)$ contains the last element that has been written on address a, and nw contains the first element of TIA which is the next element to be written.

4.4 Verification of a distributed cache memory

In our program formalism, the cache memory proposed by [ABM93] is described as a system of the form $P_1 \parallel P_2 ... \parallel P_n$ where each process P_i is defined as follows:

Name :	P_i	
Variables :	Input :	$a : address, d : datum$
	local :	AD_i : set of $address \times datum_i$, (which data are already written)
		$\mathbf{C}_i : array[address]$ of $datum \cup \{\epsilon\}$ (local cache memory)
		$Out_i : buffer$ of $(address \times datum_i)$
	shared :	$\mathbf{M} : array[address]$ of $datum$ (global memory)
		$In_k : buffer$ of $((address \times datum) \times Bool)$, $k : index$
Transitions :		
$(write_i(a,d))$		$allowed((a,d), AD_i, AD_i') \wedge append(Out_i, (a,d), Out_i') \wedge$ **unch**$(\mathbf{C}_i, \mathbf{M}, In_1, ...In_n)$
$(read_i(a,d))$		$(\mathbf{C}_i[a] = d) \wedge empty(Out_i) \wedge empty(In_{i\mid(address \times datum) \times true}) \wedge$ **unch**$(AD_i, \mathbf{C}_i, Out_i, \mathbf{M}, In_1, ...In_n)$
$(mw_i(a,d))$		$first(Out_i, (a,d)) \wedge tail(Out_i, (a,d), Out_i') \wedge update(\mathbf{M}, (a,d), \mathbf{M}') \wedge$ $\forall k : index \,.\, append(In_k, ((a,d), i = k), In_k') \wedge$ **unch**$(AD_i, \mathbf{C}_i)$
$(cu_i(a,d))$		$first(In_i, ((a,d) \times Bool)) \wedge tail(In_i, ((a,d) \times Bool), In_i') \wedge$ $update(\mathbf{C}_i, (a,d), \mathbf{C}_i') \wedge$ **unch**$(AD_i, Out_i, \mathbf{M}, \{In_j, j \neq i\})$
$(mr_i(a,d))$		$(\mathbf{M}[a] = d) \wedge append(In_i, ((a,d), false), In_i') \wedge$ **unch**$(AD_i, \mathbf{C}_i, Out_i, \mathbf{M}, \{In_j, j \neq i\})$
$(cl_i(a))$		$clear(\mathbf{C}_i, a, \mathbf{C}_i') \wedge$ **unch**$(AD_i, Out_i, \mathbf{M}, In_1, ...In_n)$
Init :		$(\forall b : address \,.\, (\mathbf{C}_i[b] = \mathbf{M}[b] = \epsilon)) \wedge$ $empty(Out_i) \wedge empty(In_i)$

where $append$, $tail$, $first$ and $empty$ are as in the Example 4.2.1. $update$ is defined by
$update(M, (a,d), M') \equiv (M'[a] = d) \wedge (\forall b : address \,.\, (b \neq a \Rightarrow M'[b] = M[b]))$
and $clear$ by
$clear(M, a, M') \equiv (M'[a] = \epsilon) \wedge (\forall b : address \,.\, (b \neq a \Rightarrow M'[b] = M[b]))$.

The only difference between our system and the one given in [Ger95] concerns the fact that each pair (a,d) can be the parameter of at most one event $write$. The way we obtain this, is by defining the type $datum$ by $datum = \bigcup_i datum_i$, such that each process "signs" the data it writes, and by using in each process a variable AD_i of type set of $address \times datum_i$ which stores the information if the event $write_i(a,d)$ has already occurred or not, as in the example of the buffer.

We verify the parameterized formulas of Proposition 4.3.1 on different abstract systems. Our aim is not necessarily to find the smallest abstract system that can be used for the verification of each formula, but we want to apply, whenever possible, the already predefined abstractions in order to show that the application of the method is simple and can be done systematically. The cache memory uses the data types and operations of the buffer of Example 4.2.1; it uses also a data type "memory"= $array[address]$ of $datum$. As for buffers, we use three different types of abstractions of a variable X of type $memory$ depending on the formula to be verified: we may

- completely forget about it (we do this for all but 1 or 2 cache memories $\mathbf{C}_i$)
- keep information about a single pair (**a**,**d**) by taking an abstract boolean variable X_A and an abstraction relation $\varrho^1_{memory}(X, X_A) = X_A \equiv (X[\mathbf{a}] = \mathbf{d})$.
- keep information about two pairs $(\mathbf{a}_1, \mathbf{d}_1)$ and $(\mathbf{a}_2, \mathbf{d}_2)$ by taking two abstract boolean variables X^1_A and X^2_A and an analogous abstraction relation $\varrho^2_{memory}(X, X^1_A, X^2_A)$.

Suppose the type $elem$ to be $address \times datum$ and take an abstract variable e_A of type $elem^1_A = \{0, 1\}$ already used in the buffer example and the abstraction relation

$$\varrho^1_{elem}((a, d), e_A) = (e_A = 0) \wedge ((a, d) \neq (\mathbf{a},\mathbf{d})) \vee (e_A = 1) \wedge ((a, d) = (\mathbf{a},\mathbf{d})),$$

exactly as in Example 4.2.4; then, it is easy to define abstract predicates $update^1_A$ and $clear^1_A$ by

$$update^1_A(X_A, e_A, X'_A) = (e_A = 0) \wedge (X'_A \Rightarrow X_A) \vee (e_A = 1) \wedge X'_A$$

expressing that if $(a, d) \neq (\mathbf{a},\mathbf{d})$, $X[\mathbf{a}] = \mathbf{d}$ can only be true in the next state if it is already true in the present state, and if $(a, d) = (\mathbf{a},\mathbf{d})$, then in the next state $X[\mathbf{a}] = \mathbf{d}$, independently of the value of $X[\mathbf{a}]$ in the present state. And similarly,

$$clear^1_A(X_A, e_A, X'_A) = (e_A = 0) \wedge (X'_A \Rightarrow X_A) \vee (e_A = 1) \wedge \neg X'_A$$

We define analogously abstractions with superscripts ^{ex} and ^{2}, concerning existential abstractions, respectively abstractions where information about two elements is conserved (in the case that 2 elements are considered, one has to distinguish the cases $\mathbf{a}_1 = \mathbf{a}_2$ and $\mathbf{a}_1 \neq \mathbf{a}_2$).

In order to obtain convenient abstractions of the buffers In_i, we need also a slightly different abstraction of a buffer. In fact, due to the action $memory\ read$, the buffer In_i may contain several occurrences of the pair (a, d), one of the form $((a, d), true)$ and severals of the form $((a, d), false)$. Treating all these occurrences as $((a, d), true)$ allows the event $read_i$ less often, but does not invalidate our set of formulas. However, we need an abstract buffer type dealing with multiple occurrences of the distinguished elements. We define an abstract buffer with the same abstract domain as in Section 4.2, but with different abstraction relation and operations. So, if $\mathbf{e}_1, \mathbf{e}_2$ represent the distinguished elements $((a, d) \times Bool)$ respectively $((a', d') \times Bool)$, then $\mathbf{e}_1 \bullet \mathbf{e}_2$ represents any buffers that has multiple occurrences of elements of the form $((a, d) \times Bool)$ and of the form $((a', d') \times Bool)$ in such a way that the first element of the form $((a, d) \times Bool)$ occurs after the last element of the form $((a', d') \times Bool)$. The abstract operations $first$ and $empty$ remain obviously unchanged, but the abstract operations $append$ and $tail$ change. For example, $append^{2alt}_A(\mathbf{e}_1 \bullet \mathbf{e}_2, ((a, d), true)), B'_A)$ is *true* if $B'_A = \mathbf{e}_1 \bullet \mathbf{e}_2$, and $tail^{2alt}_A(\mathbf{e}_2 \bullet \mathbf{e}_1, ((a, d), true)), B'_A)$ is *true* if $B'_A = \mathbf{e}_2 \bullet \mathbf{e}_1$ or if $B'_A = \mathbf{e}_2$.

Using these definitions and those already given in Example 4.2.1, the definition of appropriate abstract finite state programs of the cache memory becomes simple.

Abstract system for property (S1) : Each instance of property (S1) involves only events of a single process P_i. However, even if we succeed to verify it on P_i we can *not* deduce its satisfaction on the composed system. In fact, if we replace all processes different from P_i by the process "$Chaos$", (S1) does not hold any more on the composed abstract program. We use here another approach to compositionality: by Proposition 4.2.3, we can abstract each process P_j individually and build a global model by composing these small abstract programs. We choose the abstraction relation for all processes P_j with $j \neq i$ in such a way that shared variables are abstracted in the same way as in P_i and we forget about all local variables; this has as effect to avoid adding certain changes of shared variables which are not allowed by the concrete processes P_j.

Intuitively, (S1) is guaranteed by the fact that for any $i \in I$ after the event $write_i$(**a**,**d**) the action $read_i$ is blocked until (**a**,**d**) has traversed the buffers Out_i and In_i and has been taken into account by the event $cashupdate_i$ in the cache $\mathbf{C}_i$. That means that we need to observe the cache $\mathbf{C}_i$ and all variables which may cause $enable(read_i(a,d)$ to become $true$, that is the buffers Out_i and In_i but also the global memory M which affects $\mathbf{C}_i$ via the action $memory\ read$ (mr_i); it is not necessary to observe the buffers Out_j for $j \neq i$: as $\mathbf{d} \in datum_i$ the actions (mw_j) should not be able to push (**a**,**d**) into In_i. This leads naturally to the following abstraction relation for process P_i:

$$\varrho_i^{S1}((a,d), AD_i, \mathbf{C}_i, Out_i, \mathbf{M}, In_1, ...In_n,\ e_A, E_A, \mathbf{C}_{iA}, Out_{iA}, \mathbf{M}_A, In_{iA}) =$$

$$\begin{array}{lll} \varrho^{1}_{elem}((a,d), e_A) & \wedge & \varrho^{1}_{set\_of\_elem}(AD_i, E_A) \quad \wedge \\ \varrho^{1}_{memory}(\mathbf{C}_i, \mathbf{C}_{iA}) & \wedge & \varrho^{1}_{buffer}(Out_i, Out_{iA}) \quad \wedge \\ \varrho^{1}_{memory}(\mathbf{M}, \mathbf{M}_A) & \wedge & \varrho^{1alt}_{buffer}(In_i, In_{iA}) \end{array}$$

and for process P_j, $j \neq i$ we use the same abstraction as in ϱ_i for the shared variables and forget about all local variables

$$\varrho_j^{S1}((a,d), AD_j, \mathbf{C}_j, Out_j, \mathbf{M}, In_1, ..., In_n,\ e_A, \mathbf{M}_A, In_{iA}) =$$

$$\begin{array}{lll} \varrho^{1}_{elem}((a,d), e_A) & \wedge & \varrho^{1}_{memory}(\mathbf{M}, \mathbf{M}_A) \quad \wedge \\ \varrho^{1alt}_{buffer}(In_i, In_{iA}) & & \end{array}$$

from which we obtain by replacing concrete by corresponding abstract predicates as defined before, the following abstract program P_i^A for index i,

Variables :	abstract input :	$e_A : Bool$
	local :	$E_A, \mathbf{C}_{iA} : Bool$
		$Out_{iA} : buffer^{1}_{A}$
	shared :	$\mathbf{M}_A : Bool$
		$In_{iA} : buffer^{1alt}_{A}$
Transitions :		
$(write_i(e_A))$	$allowed^{1}_{A}(e_A, E_A, E_A') \wedge append^{1}_{A}(Out_{iA}, e_A, Out'_{iA}) \wedge$ **unch**$(\mathbf{C}_{iA}, \mathbf{M}_A, In_{iA})$	
$(read_i(e_A))$	$(e_A \Rightarrow \mathbf{C}_{iA}) \wedge empty^{1}_{A}(Out_{iA}) \wedge empty^{1alt}_{A}(In_{iA}) \wedge$ **unch**$(E_A, \mathbf{C}_{iA}, Out_{iA}, \mathbf{M}_A, In_{iA})$	
$(mw_i(e_A))$	$first^{1}_{A}(Out_{iA}, e_A) \wedge tail^{1}_{A}(Out_{iA}, e_A, Out'_{iA}) \wedge$ $update^{1}_{A}(\mathbf{M}_A, e_A, \mathbf{M}'_A) \wedge append^{1alt}_{A}(In_{iA}, e_A, In'_{iA}) \wedge$ **unch**$(\mathbf{C}_{iA}, E_A)$	
$(cu_i(e_A))$	$first^{1alt}_{A}(In_{iA}, e_A) \wedge tail^{1alt}_{A}(In_{iA}, e_A, In'_{iA}) \wedge$ $update^{1}_{A}(\mathbf{C}_{iA}, e_A, \mathbf{C}'_{iA}) \wedge$ **unch**$(E_A, Out_{iA}, \mathbf{M}_A)$	
$(mr_i(e_A))$	$(\mathbf{M}_A = e_A) \wedge append^{1alt}_{A}(In_{iA}, e_A, In'_{iA}) \wedge$ **unch**$(E_A, \mathbf{C}_{iA}, Out_{iA}, \mathbf{M}_A)$	
$(cl_i(e_A))$	$clear^{1}_{A}(\mathbf{C}_{iA}, e_A, \mathbf{C}'_{iA}) \wedge$ **unch**$(E_A, Out_{iA}, \mathbf{M}_A, In_{iA})$	
Init :	$\neg \mathbf{C}_{iA} \wedge empty^{1}_{A}(Out_{iA}) \wedge \neg \mathbf{M}_A \wedge empty^{1alt}_{A}(In_{iA})$	

and P_j^A for all indices different from i,

Variables :	abstract input :	$e_A : Bool$
	shared:	$\mathrm{M}_A : Bool$
		$In_{iA} : buffer_A^{lalt}$
Transitions :		
$(write_j(e_A), read_j(e_A),$		
$cu_j(e_A), mr_j(e_A), cl_j(e_A))$		$\mathbf{unch}(\mathrm{M}_A, In_{iA})$
$(mw_j(e_A))$		$first_A^{ex}(e_A) \wedge append_A^{lalt}(In_{iA}, e_A, In'_{iA}) \wedge update_A^{\mathbf{l}}(\mathrm{M}_A, e_A, \mathrm{M}_A)$
Init :		$empty_A^{lalt}(In_{iA}) \wedge \neg \mathrm{M}_A$

in which we have already eliminated all abstract operations that are always *true*, such as $append_A^{ex}$, $update_A^{ex}$,.... Notice that the event $(mw_j(true))$ is in fact never executed as $first_A^{ex}(true) \equiv false$ because the buffer Out_j cannot contain a pair (**a**,**d**) with $\mathbf{d} \in datum_i$. Notice also that the composed system $P_1^A \parallel ... \parallel P_i^A \parallel ... \parallel P_n^A$ is $P_i^A \parallel P_j^A$, whatever the number of components is, as for all $j \neq i$, the programs P_j^A are identical and $P \parallel P$ and P represent the same transition system.

Abstract system for property (S2): Property (S2) expresses the fact that any event $read_i$(**a**,**d**) is preceded by an event $write_j$(**a**,**d**) for some j. Thus, we need to observe as before the cache C_i the buffer In_i and the global memory M_A, but also all buffers Out_j. This leads to similar abstraction relations as for the verification of (S1), except that we need neither unicity of $write$ events and can forget about AD_i disabling $write_i$ events; however, we need abstract buffers Out_{jA} for all indices j as we only assume $\mathbf{d} \in datum$. Thus, the abstraction relations ϱ_j^{S2} are the same for all j:

$$\varrho_j^{S2}(A, D, AD_j, \mathrm{C}_j, Out_j, \mathrm{M}, In_1, ..., In_n,\ e_A, Out_{jA}, \mathrm{M}_A, In_{iA}) =$$
$$\varrho_{elem}^{\mathbf{l}}((A, D), e_A) \quad \wedge \quad \varrho_{buffer}^{\mathbf{l}}(Out_j, Out_{jA}) \quad \wedge$$
$$\varrho_{memory}^{\mathbf{l}}(\mathrm{M}, \mathrm{M}_A) \quad \wedge \quad \varrho_{buffer}^{lalt}(In_i, In_{iA})$$

For this abstraction, the obtained global abstract transition system does depend on the number n of processes as we have defined n abstract variables Out_{jA} with non-empty domain. In order to obtain an abstract transition system such that its size is independent of n, we can define — instead of the set of local abstract buffers Out_{jA} — a *single global* abstract buffer Out_A defined by a relation of the form

$$\varrho_{buffer}^{\mathbf{l},glob}(\bigcup_{j:index} Out_j,\ Out_A)$$

which obliges however to redefine abstract operations add_A^{glob}, $tail_A^{glob}$,...

Abstract system for properties (S3), (S4) and (C2) : For the verification of (S3) we need to observe events with two different parameters $(\mathbf{a}_1, \mathbf{d}_1)$ and $(\mathbf{a}_2, \mathbf{d}_2)$, such that $\mathbf{d}_1, \mathbf{d}_2 \in datum_j$; thus, we use the abstraction relations with superscript ^{2} as for the verification of order preservation in the preceding section. We define abstract variables E_A (in P_j^A) in order to guarantee uniqueness of the observed $write_j$ events, Out_{jA} (in P_j^A), C_{i_1}, C_{i_2} (in P_i^A) and shared variables In_{iA}, M_{A1} and M_{A2} and use the predefined abstraction relations and corresponding abstract operations.

The resulting global abstract transition system is again independent of the number of processes as all the abstract programs with indices different from i, j are identical. In the case $\mathbf{a}_1 = \mathbf{a}_2$, we need only to consider the case in which the indices i and j are different, as the property for $i = j$ is implied by (S1).

On the same abstraction, we can also verify the property (C2).

In presence of (S3), (S5) expresses that all $write$ events are taken into account in the same order in all processes P_{i_1} and P_{i_2}, also when they have been issued by two different processes P_{j_1} and P_{j_2}. Thus, for its verification we observe two pairs $(\mathbf{a}_1, \mathbf{d}_1)$ and $(\mathbf{a}_2, \mathbf{d}_2)$ such that $\mathbf{d}_1 \in datum_{j_1}$ and $\mathbf{d}_2 \in datum_{j_2}$. Consequently, we need abstract variables E^1_A, E^2_A (in $P^A_{j_1}$ respectively $P^A_{j_2}$), $Out_{j_1 A}$, $Out_{j_2 A}$ (in $P^A_{j_1}$ respectively $P^A_{j_2}$), $\mathbf{C}_{i_1 1}, \mathbf{C}_{i_1 2}$ (in $P^A_{i_1}$), $\mathbf{C}_{i_2 1}, \mathbf{C}_{i_2 2}$ (in $P^A_{i_2}$) and shared variables $\mathbf{M}_{A1}$, $\mathbf{M}_{A2}$, $In_{i_1 A}$ and $In_{i_2 A}$.

Verification of properties on abstract systems : The actual construction of global abstract transition systems and the verification of the formulas on them could be done automatically by our tool [GL93, Loi94]. By Proposition 4.2.8, we have to verify the consistency of the atomic propositions with the used abstraction relations. For all properties, consistency is obvious as in the corresponding $\forall$CTL* formulas only predicates of the form $after(\ell)$ or $tia_i(a, d)$ — which are consistent with the chosen abstraction relation — occur non negated.

Verification of Property (C3): As we have already mentioned, our verification does in general not allow to verify liveness properties directly: there exists no finite abstract system that verifies (C3). But under the hypothesis that the system is fair with respect to the events mw_i and cu_i for all i, one can deduce (C3) from the induction rules used in the proof given in [JPR94] and the following safety properties — which can be verified by using finite abstractions:

- $after(write_i(a, d)) \Rightarrow in(Out_i, (a, d))$
- $position(Out_i, n, (a, d)) \Rightarrow$
  $\mathbf{AX}(position(Out_i, n, (a, d)) \vee after(mw_i) \wedge position(Out_i, n-1, (a, d)))$
- $enable(mw_i(a, d)) \Rightarrow \mathbf{AX}(enable(mw_i(a, d)) \vee after(mw_i(a, d)) \wedge in(In_j, (a, d))$
- $position(In_j, n, (a, d)) \Rightarrow$
  $\mathbf{AX}(position(In_j, n, (a, d)) \vee after(cu_i) \wedge position(In_j, n-1, (a, d)))$
- $enable(cu_j(a, d)) \Rightarrow \mathbf{AX}(enable(cu_j(a, d)) \vee after(cu_j(a, d)) \wedge tia_j(a, d))$

where in and $position$ are predicates with obvious meanings.

4.5 Discussion

What have we achieved? A first impression could be that this verification of a cache memory looks much like a handwritten proof. However, it is quite different: starting right from the beginning, it is in fact rather lengthy to define all the abstraction relations and corresponding abstract predicates, even in order to verify some trivial buffer program. However, having done this once, in order to verify the much more complex cache memory system, we only need a few more definitions obtained a long the same line as the already given ones. In fact, there are many examples of systems, for which we have to verify exactly the same type of properties and which use analogous data structures and operations on them, such that the same abstract domains and operations can be used. Thus, we could build a "library" of useful abstract domains and operations in which new definitions can be added

when necessary. A similar approach has been followed by P. and R. Cousot and more recently by D. Long concerning "standard" abstractions of integers and operations on them.

The fact that for the verification of an individual property a large part of the system can be abstracted existentially is often necessary in order to obtain tractable global models. If the system is too large or the property is "too global" one can often get results by decomposing the property, depending on the particular system under study, as this has been proposed, e. g. in [Kur89].

For the verification of the cache memory, an additional complexity comes from the fact that we also have to define the set of formulas as originally the abstract specification is not given in these terms. We believe however that this set of properties is interesting by itself as it can be used for the verification of other systems supposed to implement sequentially consistent memories. The fact that our characterization is stronger than required by the definition of sequential consistency, is not a real problem, because in any particular case, it should be easy to decide which of the properties are allowed to be weakened and which not. In fact, this characterization can easily be adapted to weaker or stronger specifications which are frequently used in real implementations.

Another point which makes an abstract specification given as a set of properties so attractive, is the fact that the modification of a single property does not require to redo the whole verification process. Notice that our method is also incremental with respect to modifications of the program, as long as they allow to use the same or at least very similar abstraction relations and abstract operations, which is often the case. That means that exactly the time consuming and difficult part of the verification process need not to be redone. In the case that the obtained abstract program is not already identical to the previous one, the reconstruction of a model and the verification of the properties on it by means of some model checker poses no problem.

Acknowledgements: I would like to thank the referees for pointing out that the initial characterization of sequential consistency was not sufficient, Amir Pnueli for giving me some ideas how to get a satisfactory solution and Denis Dams and Joseph Sifakis for fruitful discussions.

Chapter 5

A CSP Approach to Sequential Consistency

G. Lowe and J. Davies

5.1 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.*

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 5.2 we give a brief overview of the syntax of CSP, and in section 5.3 we describe the traces model, and how it may be used to specify and verify processes. We describe the lazy caching protocol in section 5.4, and show how it may be encoded as a CSP process in section 5.5. In section 5.6 we express the notion of sequential consistency as a trace specification, and in sections 5.7 and 5.8 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.

5.2 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 || 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 $1 \ldots n$, while

$$\overset{\circ}{9}_{i \in 1..n} P_i$$

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 $i.a$.

5.3 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.

5.3.1 Semantics

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

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

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

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

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,

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

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.

5.3.2 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

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

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.

5.3.3 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.

$$\frac{}{Stop \textbf{ sat } tr = \langle\rangle}$$

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:

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

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 $tail(tr)$.

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

$$\frac{P \textbf{ sat } S(tr) \quad Q \textbf{ sat } T(tr)}{P \,\Box\, Q \textbf{ sat } S(tr) \vee T(tr)}$$

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:

$$\frac{P \textbf{ sat } S(tr) \quad Q \textbf{ sat } T(tr)}{P \parallel Q \textbf{ sat } S(tr \restriction \alpha P) \wedge T(tr \restriction \alpha Q)}$$

We write αP to denote the set of events that appear in the description of process P. The projection $tr \restriction \alpha P$ 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:

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

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.

5.4 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 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 5.1.

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

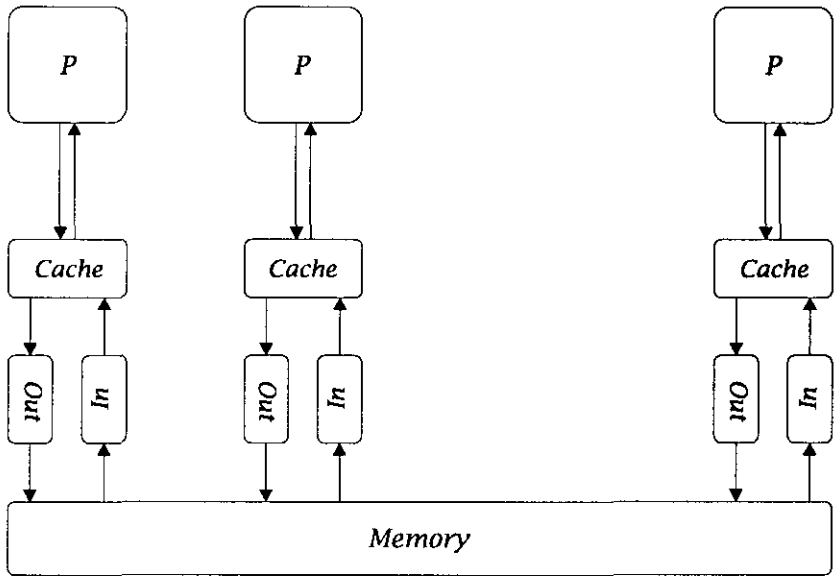


Figure 5.1: A lazy caching architecture

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;
- 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 5.2.

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.

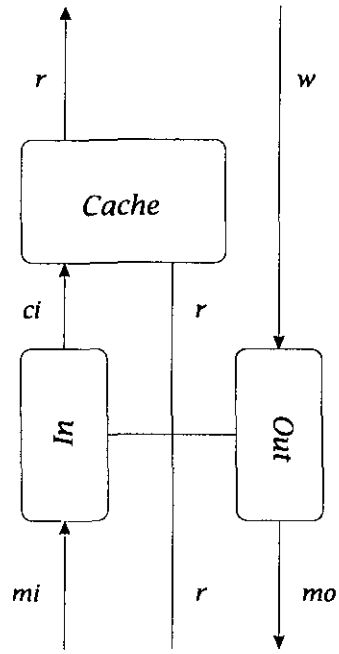


Figure 5.2: Cache and queues at a single node

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

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

denotes the set of all internal communications.

5.5 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.

$$Out \quad = \quad Out_{\langle\rangle}$$

$$
\begin{aligned}
Out_{\langle\rangle} &= w?a.d \longrightarrow Out_{\langle a.d\rangle} \\
&\quad \Box \\
&\quad r?a.d \longrightarrow Out_{\langle\rangle} \\
Out_{\langle a'.d'\rangle^\frown s} &= w?a.d \longrightarrow Out_{\langle a'.d'\rangle^\frown s^\frown \langle a.d\rangle} \\
&\quad \Box \\
&\quad mo!a'.d' \longrightarrow Out_s
\end{aligned}
$$

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.

$$
\begin{aligned}
In &= In_{\langle\rangle} \\
In_{\langle\rangle} &= mi?a.d.f \longrightarrow In_{\langle a.d.f\rangle} \\
&\quad \Box \\
&\quad r?a.d \longrightarrow In_{\langle\rangle}
\end{aligned}
$$

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.

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

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

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

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.

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

where $zero$ is given by

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

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.

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

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

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

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.

5.6 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 the condition that two traces tr and tr' agree upon the order of reads and writes as seen from each node is defined by the predicate $Consistent(tr, tr')$, 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.

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

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.

5.7 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

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

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

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

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.

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

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

$$\begin{aligned} i.In \quad & \textbf{sat} \quad i.IN(tr) \\ i.Out \quad & \textbf{sat} \quad i.OUT(tr) \\ i.Cache \quad & \textbf{sat} \quad i.CACHE(tr) \\ Memory \quad & \textbf{sat} \quad MEMORY(tr) \end{aligned}$$

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

5.7.1 System properties

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

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

Using the same rule once more, we obtain

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

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

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

A final application of the rule for parallel composition yields

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

where

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

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

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

5.8 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,

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

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,

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

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 $Nodes \parallel Memory$;
- 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 $k = \#(tr_0 \restriction i.ci)$. This is equivalent to showing that

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

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

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

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

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

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. we suppose that 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

$$\#((tr_0 \frown \langle i.w.a.d \rangle \frown tr_1 \frown tr_2 \frown \langle i.mo.a.d \rangle \frown tr_3) \restriction i.ci) = \#((tr_0' \frown \langle i.w.a.d \rangle \frown tr_1') \restriction I.mo$$

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

$$(tr_0 \frown \langle i.w.a.d \rangle \frown tr_1 \frown tr_2 \frown \langle i.mo.a.d \rangle \frown tr_3) \Downarrow i.ci.\star = (tr_0' \frown \langle i.w.a.d \rangle \frown tr_1') \Downarrow i.mo$$

Also from *Spec* :

$$(tr_0 \frown \langle i.w.a.d \rangle \frown tr_1 \frown tr_2 \frown \langle i.mo.a.d \rangle \frown tr_3) \Downarrow i.ci.\star = (tr_0 \frown \langle i.w.a.d \rangle \frown tr_1 \frown tr_2 \frown \langle i.mo.a.d \rangle \frown tr_3) \Downarrow i.mo$$

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

$$\#((tr_0'' \frown tr_1'' \frown \langle i.r.a'.d' \rangle \frown tr_2'' \frown tr_3'') \restriction i.mo = \#((tr_0'' \frown tr_1'' \frown \langle i.r.a'.d' \rangle \frown tr_2'' \frown tr_3'') \restriction i.w$$

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

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

Hence,

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

contradicting *Spec*.

5.9 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.

Chapter 6

The Compositional Approach to Sequential Consistency and Lazy Caching

W. Janssen, M. Poel, J. Zwiers

6.1 Introduction

Adequate decompositions often simplify the analysis and understanding of distributed systems. Such decompositions can to a large extent be formulated in a way that is independent of the underlying model of distributed systems. Formal correctness proofs that exploit this decomposition can have a top level structure that is largely independent from the particular formalism used as well. We substantiate this claim by explaining the *lazy caching* algorithm as proposed by Afek, Brown and Merritt [ABM93].
The algorithm in [ABM93] describes the implementation of a so called *sequentially consistent* shared memory [Lam79]. The result of our investigation is that the algorithm can be decomposed into essentially four simple protocols. The four protocols are of more general interest than just the lazy caching algorithm. For instance, the protocol concerned with replication of memories suggested a type of memories called *write-coherent* memories. Write-coherent memories have the property that they can be replicated, while preserving the write-coherency property. Such is not the case for the class of sequentially consistent memories as used in the lazy caching algorithm. A second aspect of our replication protocol is that it allows for less (or more) replicas than the number of processors accessing the shared memory. The lazy caching algorithm assumes exactly one replica, in the form of a cache, for each processor. Such variations of the original algorithm are more easily found in a *compositional* setup, since it allows one to deal with one aspect at a time.

The structure of the system used in the lazy caching algorithm is sketched in figure 6.1. In this figure we have shown a system communicating with four users, using three caches. Informally the algorithm behaves as follows. A number of user processes i communicate via channels W_i, Req_i, and Ret_i with a memory system. Write actions are sent via the W_i channel and are queued in separate so called *Out queues* for all users. At the bottom level of the system we have a number of cache memories and a general memory. The caches have queues connected to them as well. When writes leave the Out queue they are distributed to all caches (where they are queued again in queues In_j), and to the memory component. Reads are requested via Req_i channels, and the values requested are returned via the Ret_i channel. The protocol is such that read requests of user i are forwarded to the caches only if there are no queued writes in the Out queue of process i, and there are no writes by user i in the In queue connected to the cache it reads from.
Finally, the cache components can request values from the memory when needed. This models the fact that caches can have limited capacity only, whereas the memory always stores the latest values of all addresses. The values requested are put into the In queue connected to the cache as well.

As stated above, the top level of our proof does not depend on a particular formal model. As a consequence, we could make a choice which style of reasoning to use for the more detailed proofs concerned with the our four simple protocols. The fact that sequential consistency is always formulated in terms of traces strongly suggested a trace based model. A second consideration was that partial order methods, as we have exploited them in a shared variable setting [ZJ94, JPZ91], have proven themselves as techniques that yield considerable insight. Moreover, the distinctions between coherent memories, sequentially consistent memories, and write-coherent memories can be explained quite well by means of partial orders, and the related concept of *dependency relations* between the various communication channels of such memories. The model that we have chosen is a partial order version of the quiescent trace model advocated by Chandy and Misra, and Jonsson [Mis84, Jon85]. The model can specify both safety and liveness properties, yet has a very simple compositional rule for parallel composition; in essence, parallel composition of systems can be seen as logical conjunction on the level of trace specifications.

The model we use is based on the models and ideas underlying IO-systems [Jon85, Jon87] and

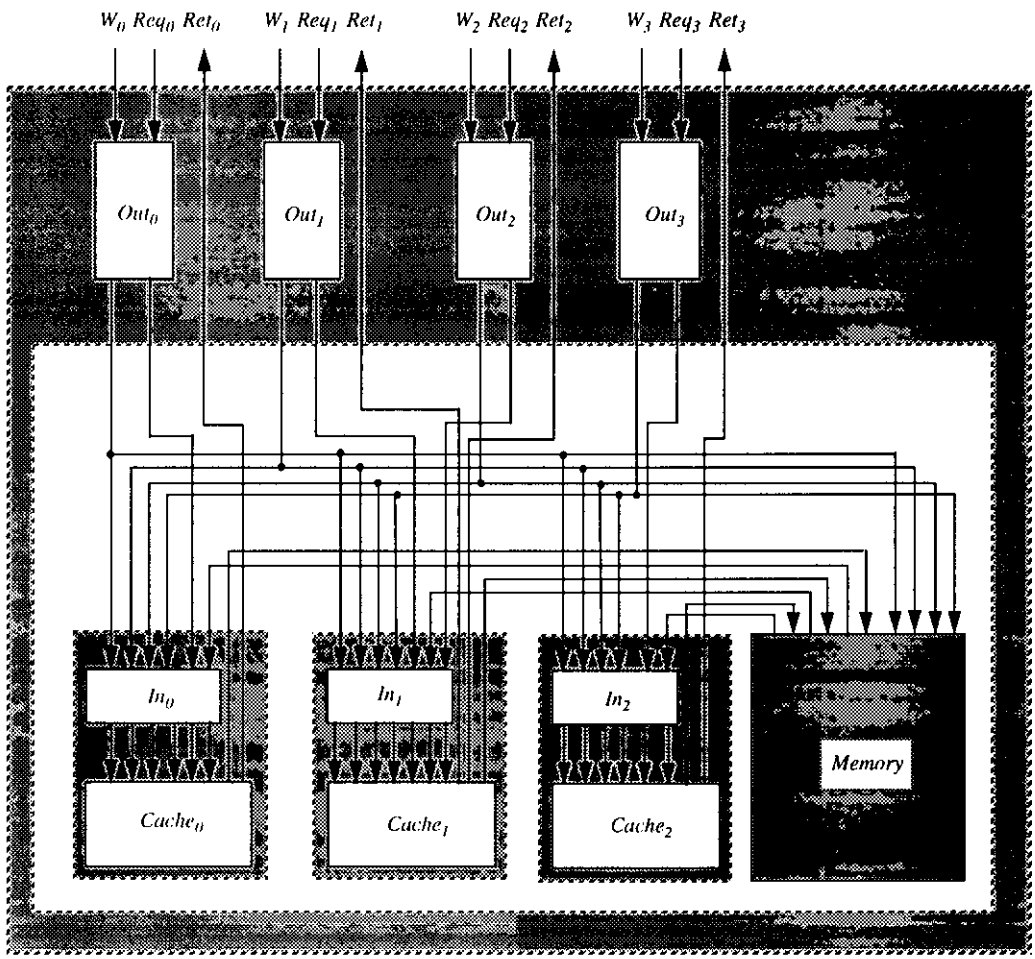


Figure 6.1: Structure of a lazy cache system with 4 users and 3 caches.

IO-automata [LT87], though reformulated in a partial order style taken from [ZJ94, JPZ91].
Informally, a component of a parallel system communicates with other components via directed communication channels. For each system execution H the communication events associated with some particular channel c form a linear history that we denote by $H \lceil c$. Communications along different channels c and d say, are not ordered unless a so called *dependency relation* exists between the actions along the two channels. Conceptually speaking one might think of such dependencies as being generated by a set of observers, each of which is capable of observing communications, including their relative ordering, along some given set of channels.
The idea of observers is of paramount importance to our view on the differences between coherent memories and sequentially consistent memories: For coherent memories one postulates a single, "global" observer at the interface between memory and user processes. For sequentially consistent memories there is no such global observer, at least not at the interface level. Rather there is an observer associated with the read and write actions issued by each particular user process. Consequently, read and write actions issued by a single user process are ordered, whereas actions stemming from different users processes remain unordered at the memory interface.

Our partial order model is inspired by the work of [Pra86, Gis84, Maz89]. It is similar to the models that we used in [ZJ94, JPZ91] except that here we focus on a communication based model, rather than on a shared memory model. The main reason for a communication based model is that interaction between various system components such as queues, busses or caches, is conveniently modeled by synchronizing communication actions.

The outline of the paper is as follows. After introducing a simple language for networks of processes we informally introduce the four protocols in our decomposition, and the memory types used. We show how the four protocols are composed to give a sequentially consistent system in the style of Affek, Merritt and Brown. Thereafter we introduce our specification language, and formally specify the memory types. Finally the formal proofs of the protocols and their properties are given.

Note and acknowledgement. This paper originates from a joint draft with Shmuel Katz. His contributions to the decomposition idea are gratefully acknowledged.

6.2 Networks of processes

6.2.1 The process language

In this section we introduce the process language and the model we use for describing networks of communicating processes. The process language appears later on as a sublanguage of the mixed term formalism we use for specification and design.

A *network* or *system* consists of the parallel composition of a number of processes. Processes communicate asynchronously via directed channels. The alphabet $\alpha(P)$ of a process P is the set of channels connected to that process. This set $\alpha(P)$ is the union of the input channels $I(P)$ and the set of output channels $O(P)$. For networks of parallel processes it is allowed that more than two processes are connected by a common channel, say c. In that case, only one process P can have c as an output channel. Any message sent by this P process is received by all others connected to c. If c is an input channel of process Q, then it is assumed that Q is always able to accept input via c. As a consequence, deadlock behavior as in CSP style process languages is not possible at all. A network of processes can reach a so called *quiescent state:* no output actions are possible anymore, at least until more input messages have been received. As described in [Mis84, Jon85], the semantics of such networks is adequately described by *quiescent traces:* sequences of communications along channels, corresponding to quiescent states. Below we introduce specifications S for quiescent traces.

We take such specifications S, together with a declaration of input and output channels, as the basic components of networks. That is, we do not provide an algorithmic language for basic components, but rather we admit process specifications S as basic components. As usual, we do not make a formal distinction between processes and networks of processes and we use "process" to refer to both basic components and networks. Channels c of a process P can be renamed into d by means of the renaming construct $P[d/c]$. A set of channels α of a network P can be made local to that network by the hiding construct $P\backslash\alpha$. The syntax of out process language is as follows. It has been taken from [Jon85].

$$P ::= (I, O : S) \quad | \quad P_1 \parallel P_2 \parallel \cdots \parallel P_n \quad | \quad P\backslash\alpha \quad | \quad P[d/c].$$

The input and output channels, $I(P)$ and $O(P)$ of processes P are defined in the following table. In all cases, the alphabet $\alpha(P)$ is defined as $I(P) \cup O(P)$.

P	$I(P)$	$O(P)$
$(I, O : S)$	I	O
$P_1 \parallel \cdots \parallel P_n$	$(I(P_1) \cup \cdots \cup I(P_n)) \setminus (O(P_1) \cup \cdots \cup O(P_n))$	$O(P_1) \cup \cdots \cup O(P_n)$
$P\backslash\alpha$	$I(P) \setminus \alpha$	$O(P) \setminus \alpha$
$P[d/c]$	$I(P)[d/c]$	$O(P)[d/c]$

The syntax of specifications S is spelled out in more detail below, where we introduce a mixed formalism, unifying processes and logic specifications. At this point it suffices to state that such specifications are predicate formulae with occurrences of so called *trace projections* of the form $H \restriction \alpha$. Such trace projections denote the (specified) trace, projected onto the alphabet α. It is assumed that for process $(I, O : S)$ all trace projections $H \restriction \alpha$ are on alphabets α such that $\alpha \subseteq I \cup O$.

6.2.2 The model

Processes P are interpreted as sets of traces of communication actions. Single communication actions a have the form $(c, v_0, \ldots, v_n)$, where c is the name of a channel, and where $v_0, \ldots, v_n$ are one or more values or attributes, sent along this channel. A trace H is then defined as a (finite or infinite) set of occurrences of communication actions, called *events*, together with an order among those events.

More precisely, a trace is a directed, acyclic graph $(V, \rightarrow)$ where V is a set of events, and where "$\rightarrow$" is what is called the *causal ordering* relation on events. We assume a symmetric and irreflexive *dependency relation* "$\sim$" on events.

Traces are required to be *strictly dependency closed,* which means that two events $e, e' \in V$ are ordered if and only if they are dependent events, i.e.:

$$e \sim e' \text{ iff } e \rightarrow e' \text{ or } e' \rightarrow e.$$

We use $His(\alpha)$ to denote the set of all possible traces over alphabet α.

The model that we use here is consistent with the (more complicated) models in [ZJ94]. The simplified model that need here has several isomorphic counterparts. For instance, we will often identify a trace of the form $(V, \rightarrow)$ with the partial order $(V, \rightarrow^+)$, where "$\rightarrow^+$" is the transitive closure of "$\rightarrow$". In fact, because of the strict dependency closure condition, a set of traces can also be represented by the set of all possible linearizations of those traces. (For the models in [ZJ94] this is not the case.) Much of what follows has been formulated neutrally in this respect, i.e. it can be understood both in the partial order and the isomorphic interleaving model. (When we come to specifications and in particular to the actual correctness proofs the concepts of partial orders turn out to be quite essential to clarify matters however.)

Process P with alphabet α is interpreted as a subset of $His(\alpha)$. To simplify notation, we identify here the term P and the set of traces it denotes. For a basic component that has the form of a specification $(I, O : S(H))$, with free (trace-typed) variable H, this is the set of all traces over $I \cup O$ that satisfy formula $S(H)$. To define parallel composition of processes P and Q we rely on the projection operation $H \restriction \alpha$.

$$P \parallel Q \stackrel{\text{def}}{=} \{H \in His(\alpha(P) \cup \alpha(Q)) \mid H\restriction\alpha(P) \in P, H\restriction\alpha(Q) \in Q\}.$$

Hiding is simply defined by means of projection: $P\backslash\beta$ denotes $P \restriction (\alpha(P) - \beta)$. Finally, renaming $P[d/c]$ is defined as usual, simply by renaming all occurrences of channel name c in P traces into d. We require that for a process $P[d/c]$ the channel d has the same dependencies within P as c does. This ensures that the resulting system is again strictly dependency closed if P is.

6.3 Memory types and interfaces

Our analysis of the lazy caching protocol is based on three different types of shared memories:

(i) Coherent memories,

(ii) Write-coherent memories,

(iii) Sequentially consistent memories.

In order to clarify the differences between these memory types one must consider the the interface between user processes and memory modules. We found it very useful to make a distinction in terms of so called *dependency relations* between the various forms of read and write actions. In our partial order model, dependency has a precise meaning in that it indicates which sort of events will be ordered and which ones will remain unordered. Within interleaving models, that could have been used instead, independence of actions can be understood thus: If two actions, say a and b are independent for system P, then the specification for P leaves the direct order of a and b unspecified.

All types of memories have the same interface, except for dependencies between the actions. There are N user processes $P_0, \ldots, P_{N-1}$, that can execute read and write actions. Write actions are executed simply by sending a memory address and a value to be written along channel W_i. A read access is executed by sending an address along read request channel Req_i, followed by receiving the value read via a read return channel Ret_i. It is assumed that user process P_i will *wait* after doing a request until the corresponding return action is received. Within models that allow for *synchronized* actions it is possible to combine read requests and read returns in one "joint" action. For models like IO automata, or the quiescent trace model that we use here, such is not possible due to requirement that input actions must always be enabled. The interface as described is essentially taken from [ABM93] except that we have left out "write return" actions. (Write return actions in [ABM93] do not carry any information; it appears that they have been included only to have a more symmetric protocol.)

Reads and writes are both parameterized with an address a and a value to read or to be written d. Thus every memory type has the same alphabet α: (let $n = N - 1$ and $I = \{0, \ldots, n\}$.)

$$\alpha \stackrel{\text{def}}{=} \{W_i, Req_i, Ret_i \mid i \in I\}$$

with different dependencies. (See figure 6.2.)

Conceptually speaking, each user process attached to a *coherent* memory observes the order of read and write actions performed by itself *and all other user processes.* So all memory accesses

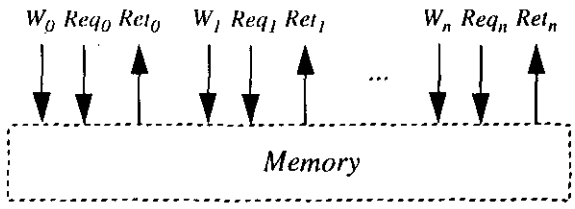


Figure 6.2: Interface of memory types.

appear to be totally ordered, in one globally interleaved trace. The precise specification of a coherent memory can be found in section 6.4.2, in the form of a predicate formula $CM(H)$ on traces H of read and write accesses. These details are not needed to understand sequential consistency and write-coherency.

For *sequentially consistent memories* we relax the global ordering condition on memory accesses. For such memories we only assume that a user process P_i can observe the order of its *own* read and write actions, and only *indirectly*, via the values returned by its read actions, it can observe the *values* that were written by other processes. But it cannot observe, in a direct sense, the order of its own actions with respect to actions performed by other processes P_j. Let $H \restriction A_i$ denote the projection of trace H onto the (read/write) actions A_i executed by process P_i. Each of these trace projections $H \restriction A_i$ can be seen as a totally ordered sequence, though no ordering is assumed between actions in $H \restriction A_i$ and actions in $H \restriction A_j$ for $i \neq j$. The specification of sequentially consistent memories has the form of a predicate formula $SCM(H)$:

> $SCM(H)$ holds for H iff there exists a (totally ordered) trace H' such that $CM(H')$ and, moreover, $H \restriction A_i = H' \restriction A_i$, for all processes P_i that access the memory.

In other words, H is sequentially consistent if there is a trace H' of a coherent memory that is equivalent to H in the sense that the two traces are identical if we omit from H' the ordering between actions that are independent from the sequentially consistent memory point of view.

Another interesting form of memory, that we use in the derivation of the caching algorithm, is so called *write-coherent memory*. This is a memory model "in between" coherent and sequentially consistent memories: All write actions are mutually dependent – similar to the case of coherent memories – but read actions for process P_i are dependent only on write actions for the same process P_i, i.e. they are *not* ordered with respect to read or write actions by other user processes. The trace based specification $WCM(H)$ is as follows. Let, A_i denote the read and write accesses or process P_i, and let W denote the combined set of all write accesses.

> $WCM(H)$ holds for H iff there exists a (totally ordered) trace H' such that $CM(H')$, $H \restriction A_i = H' \restriction A_i$, for all processes P_i that access the memory, and moreover, $H \restriction W = H' \restriction W$.

6.3.1 A memory builders toolkit

We discuss how memories of various types can be built from other memory modules and a few basic components like queues. We show that the caching algorithm by Afek, Brown, and Merritt can be constructed in this way and so we prove that it implements a sequentially consistent memory, indeed. Our approach makes it easy to consider variations of their algorithm. Moreover, it becomes easier

to consider related applications outside the realm of memories. One such application is that of a distributed databases with replicated data.

Queueing

An important ingredient of the caching algorithm by Afek, Brown, and Merritt is the queueing of write actions. According to the different memory models (coherent, write-coherent, or sequentially consistent) we distinguish three different queueing protocols.

For coherent memories queueing is not really important. Coherent memories are "normal" memories that behave as simple sequential algorithms. One can view such a coherent memory as a sequential memory with a single queue for writes for all user processes, and read returns to be allowed only if there are no pending writes. This queueing however is by no means vital for the behaviour. The only potential difference that one could observe at the interface is in terms of real-time properties: a read request after a series of writes is answered only after some delay.

For sequentially consistent memories read and write actions for one process P_i are, at the interface level, independent from similar actions stemming from other processes P_j. Consequently, queueing write requests for sequentially consistent memories is achieved by introducing *separate* queues, one for each user process. Process P_i enters its write requests and read requests in queue Q_i, and must wait after having made a read request until the corresponding read return is received. (Indeed it need not wait for any pending requests made by other processes.) Finally, it is possible to queue read return actions, too. Such queueing is not present in [ABM93], yet it seems a useful idea when a remote user process is reading a number of memory locations one after another. In section 6.5.1 we prove the following theorems:

(1a) A module consisting of a *sequentially consistent memory SCM* with N queues for read and write requests, and N queues for read returns, conforming to the protocol as described above, behaves as a *sequentially consistent memory* for N user processes.

(1b) Replacing the sequentially consistent memory in the protocol above by a coherent module still yields a sequentially consistent memory.

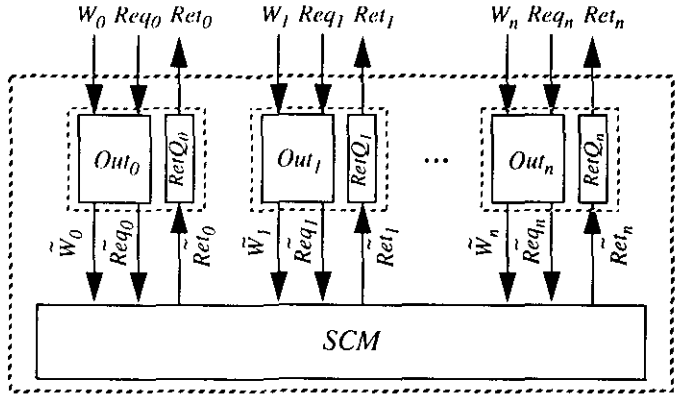


Figure 6.3: A coherent memory with WR-protocols.

The correctness proofs given in section 6.5 are straightforward. Construction (*1*) for instance, follows almost from the definition of sequential consistency: the coherent memory behaviour that the definition asserts, is actually present here at the (internal) interface between the coherent memory and

the queuing protocol. Moreover, this latter protocol ensures that for each individual process P_i the behaviour at the interface between P_i and the queues is essentially the same as that at the interface between the queues for P_i and the coherent memory.

Finally we discuss a second type of queues on top of coherent memories. Considering write requests, one sees that because of the total ordering of these at the interface level, we must have a single queue for for all of these requests, just like the situation for coherent memories. The protocol for read actions is more complex, since read actions for process P_i are ordered with respect to write actions for P_i only. A possible queuing protocol here is to tag write requests in the queue that were issued via channel W_i by the index i. Read requests via Req_i are delayed until no write requests *tagged by* i remain in the queue. This protocol is used within the design by [ABM93]. There, it is combined with replication so that each module has N writers, but only one reader P_i. Consequently, it is not necessary to tag by means of process indices, but rather a simple star "$*$" is attached to the write requests made by P_i.

We have the following important property, proven in section 6.5.3:

(2) A module consisting of a coherent memory plus a single queue for tagged write requests, conform the protocol as above, behaves as a write-coherent memory itself.

In fact, one could even replace the coherent component in the module above by a write coherent component, while retaining the write coherent behaviour.

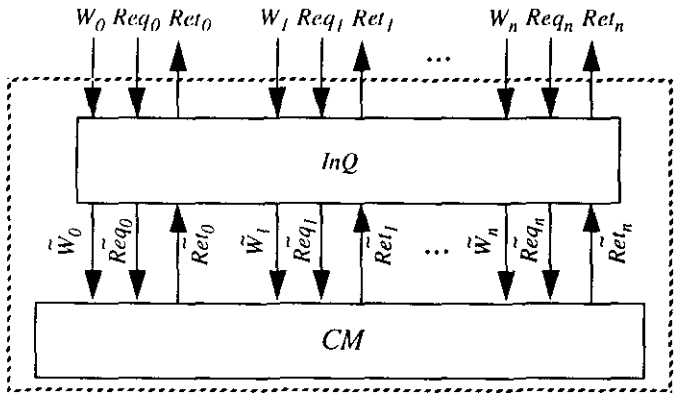


Figure 6.4: A write-coherent memory.

Replication

A write-coherent memory WCM for N user processes P_i, $0 \leq i < N$, can be obtained by *replication* of write-coherent memories as follows. Assume that WCM_l is write-coherent, for $0 \leq l < K$. (The number of replicas, K, need not be equal to the number of user processes N.)

- A write action along the W_i channel of WCM is implemented by issuing atomically similar write actions via the W_i channels of all WCM_l memories.

- All read requests and returns for user P_i are executed by issuing the same actions at only one of the WCM_l memories. Moreover, for a given user P_i, all read actions must be executed *at one and the the same* WCM_l memory.

We prove the following theorem in section 6.5.2:

. replication of write-coherent memories, connected to N user processes as described
behaves as a write-coherent memory itself.

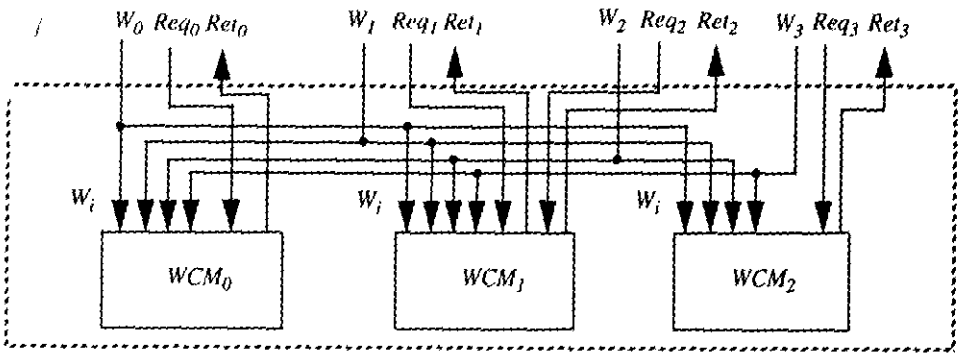


Figure 6.5: A replicated write-coherent memory with $N = 4, K = 3$.

Caches

Coherent memories are obviously write-coherent, so one or more of the memories in the replication construction discussed in the previous section could be coherent memories. Assume that Mem is such a coherent replica. We intend to use Mem as a kind of "back up" copy in case one or more of the other WCM_l memories would lose the value that it stores for some data item a say. Such a "loss" of data is used to model the behaviour of *cache misses*. A *data replication action* or *cache update* for WCM consists of requesting the value of some data item a from Mem, combined (atomically) with a write action of the corresponding value d for a at one replica WCM_l.

(4) Such a cache update action does not change the externally observable behaviour of the system as a whole, that is, the system still behaves as a write-coherent memory.

We prove this fact in section 6.5.4.

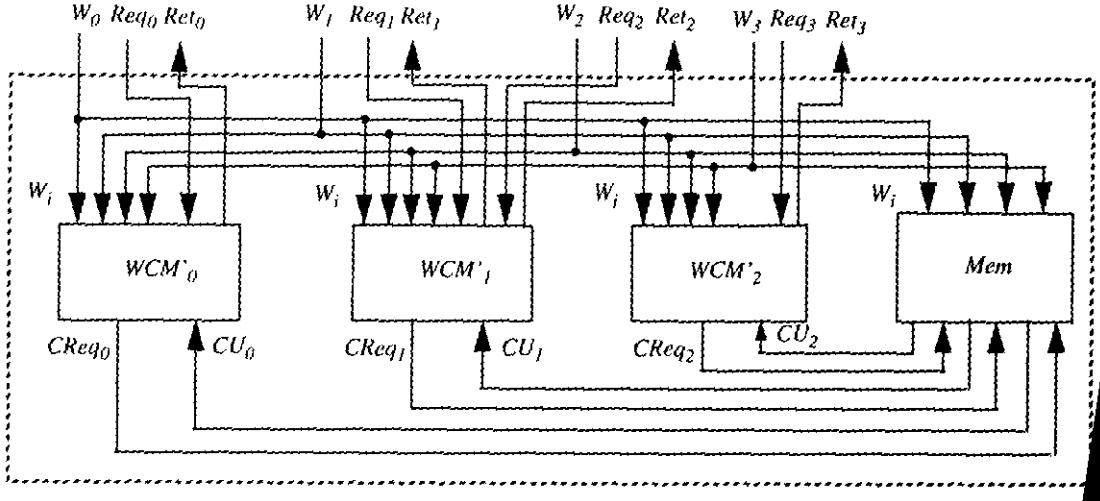


Figure 6.6: A replicated write-coherent memory with cache updates

6.3.2 The lazy caching protocol: top level proof

We have discussed memories, queues, and properties of the different types of memories. These components can now be combined in a number of steps, with as a result, a sequentially consistent memory along the lines of Afek, Brown, and Merritt. We sketch the different steps in the proof.

1. At the heart of the algorithm we have N cache memories $Cache_i$. These cache memories are coherent, that is, they are ordinary memories, possibly with limited capacity. Since coherency implies write-coherency, we may treat these caches as write-coherent memories. We can put queues "*In*" on top of these, while preserving write-coherency, as explained in the sections above.

2. Moreover, we can regard each of these *Cache/In* combinations as one replica of a combined module that, according to the theorems above, still is a write-coherent memory.

3. The theorem on caching above implies that a coherent memory "*Mem*" can be added to these replicas, and that in case of cache misses, a copy from *Mem* to one of the caches can be made, while preserving write-coherent behaviour.

4. Finally, we can put simple queues "*Out*" for write/read requests between the user processes and the (replicated) write-coherent memory. Again, we rely on the theorems above, and conclude that the result is a sequentially consistent memory.

In fact, besides the above top-level proof of sequential consistency of the lazy caching algorithm, we have similar proofs for a number of generalizations as a result of our decomposition as well. These include the use of multiple "back up" memories, different number of caches and users, and queuing of read return messages.

6.4 Specifications and proofs

6.4.1 The mixed term language

To specify processes we use a specification language inspired by similar languages for trace based reasoning [Zwi89]. The main difference with specifications for interleaving traces is that trace indexing is only allowed for sub traces that are guaranteed to be linearly ordered, and that an explicit precedence relation for events is used. (We remark that for fully interleaved traces such a precedence relation can be seen as a convenient abbreviation.)

We introduce trace expressions *Texpr*, event expressions *Event_expr*, and integer expressions *Int_expr*. We assume given sets of (typed) variables H, h, e, i and channels c, d. A set of channels is denoted by α. For indexed traces $te(ie)$, we require that all channels in the trace alphabet $\alpha(te)$ (defined below) are mutually dependent. This guarantees that te denotes a totally ordered trace.

$te \in Trace\_expr$,			
	$te ::=$	$< ee >$,	*single event trace, containing only ee.*
		$H \restriction \alpha$, $h \restriction \alpha$	*projected trace variables*
		$te_0 \bullet te_1$,	*layer composition*
		$te \restriction \alpha$,	*projection of te onto* α
		$te[d/c]$,	*trace te with channel c renamed into d*
$ee \in Event\_expr$,			
	$ee ::=$	e,	*event variable*
		$te(ie)$	*indexed trace expression: the i-th event in trace te.*
$ie \in Int\_expr$,			
	$ie ::=$	$0, 1, \ldots$	*integer constants*
		i	*integer variables*
		$\#te$	*the number of events in trace te*
		$ie_0 + ie_1, \ldots$	*standard arithmetical operations*

The only unusual operation in *Trace_expr* that needs explanation is *layer composition* $te_0 \bullet te_1$. We define this operation here only when te_0 denotes a finite trace. In essence, $te_0 \bullet te_1$ is the disjoint union of te_0 and te_1, with the order augmented such that events e in te_0 causally precede dependent events e' in te_1. For a trace te that denotes a run $h \stackrel{\text{def}}{=} (V, \rightarrow)$ we define $\#te = |V|$, that is, $\#te$ denotes the number of events in te.

For event attributes such as channel names, communicated values, memory addresses etc., we assume that there are corresponding classes of expressions, containing at least the following:

$ae \in Attr\_expr$,			
	$ae ::=$	$addr(ee)$	*address attribute of event ee*
		$val(ee)$	*value attribute of event ee*
		$chan(ee)$	*channel attribute of event ee*
		a, v, c	*constants of appropriate type*

For trace expressions te we define their *alphabet* $\alpha(te)$:

te	$\alpha(te)$
$\{ee\}$	$chan(ee)$
$H \restriction \alpha$	α
$te_0 \bullet te_1$	$\alpha(te_0) \cup \alpha(te_1)$
$te \restriction \alpha'$	$\alpha(te) \cap \alpha'$
$te[d/c]$,	$\alpha(te)[d/c]$

Finally we define a class of "mixed term" specifications, and an auxiliary class of quiescent trace specifications. This class is a unification of trace based specifications and processes. For a term of the form $P(H \restriction \alpha')$ we require that $\alpha' \subseteq \alpha(P)$. By convention, we use $P(H)$ for $P(H \restriction \alpha(P))$. For a process of the form $(I, O : S)$ we have a condition, stated below, that S actually specifies a ***quiescent trace set*** over input and output channels I and O.

$S \in Spec$,

$S ::=$	$P(H \restriction \alpha)$	*processes as "predicates"*
	$te_0 = te_1, ee_0 = ee_1,$	*equality of traces, events*
	$ee_0 \rightarrowtail ee_1$	*event precedence*
	$te_0 \leq te_1,$	*trace prefix relation*
	$ae_0 = ae_1, ie_0 = ie_1$	*equality of integers, attributes*
	$S_0 \wedge S_1, S_0 \Rightarrow S_1,$	*boolean connectives*
	$\exists h : His(\alpha).s(h),$	*quantification over traces*
	$\exists e \in te.s(e),$	*quantification over events in a given trace*
	$\exists i \in Nat.s(i)$	*quantification over integers*

$P \in QSpec$,

$P ::=$	$(I, O : S(H))$	*Specifications with input/output channels*
	$P_0 \parallel P_1,$	*Parallel composition*
	$P \backslash \alpha,$	*Hiding of the channels in* α
	$P[d/c]$	*Renaming of channel c into d*

For mixed terms S and trace variable h we define the *base* $\beta(S, h)$ as the union of all channel sets α such that $h \restriction \alpha$ occurs within S, for some free occurrence of h. When specifying some particular process P say, the variable "H" is used (by convention) to refer to the traces of P. Within this context we usually write P rather than $P(H)$, $\beta(S)$ for $\beta(S, H)$ etc.

Most of this language is interpreted as usual for (typed) predicate logic. We interpret all relations between expressions, including equality, *strictly* in the sense that the result is "false" whenever some of the operands are undefined.

In the sequel we will use the following abbreviations:

#c: For a channel c, $\#c$ denotes the number of events in the projection onto c: $\#c \stackrel{\text{def}}{=} \#(H \restriction c)$.

c(j): For a channel c, $c(j)$ denotes the j^{th} event in the projection onto c: $c(j) \stackrel{\text{def}}{=} (H \restriction c)(j)$.

c(ee): For a channel c and an event ee, $c(ee)$ denotes the predicate that the channel attribute of ee is c: $c(ee) \stackrel{\text{def}}{=} (chan(ee) = c)$.

last(H): For a linear finite trace H, $last(H)$ the last event of H: $last(H) = H(\#H)$.

$ee_0 \sim ee_1$: Two events ee_0 and ee_1 are dependent, denoted by $ee_0 \sim ee_1$, if ee_0 precedes ee_1 or visa versa: $ee_0 \sim ee_1 \stackrel{\text{def}}{=} (ee_0 \rightarrowtail ee_1) \vee ee_1 \rightarrowtail ee_0$.

$c \sim d$: Two channels c and d are dependent, denoted by $c \sim d$ if each communication event along channel c is dependent on each communication event along channel d: $c \sim d \stackrel{\text{def}}{=} \forall i, j.\ (H \restriction c)(i) \sim (H \restriction d)(j)$.

$Dep = \{A_0, A_1, \ldots A_n\}$: For each set of channels A_i in Dep we have that all channels in A_i are mutual dependent: $Dep = \{A_0, A_1, \ldots A_n\} \stackrel{\text{def}}{=} \forall i, 0 \leq i \leq n, \forall c, d \in A_i.\ c \sim d$. As a consequence $H \restriction C$ is a linear trace if $C \subseteq A_i$ for some i.

In our model we always assume as a global model property that every channel is dependent of itself, i.e. let *Chan* be the set of all channels then

$$\forall c \in Chan.\ c \sim c$$

As a consequence $H \restriction c$ is a linear trace for a channel c thus $(H \restriction c)(i)$ is well-defined for $1 \leq \#c$.

Quiescent trace specifications, too, are treated as predicate formulae, where the process connectives are treated as abbreviations. We define the following translation from quiescent trace specifications to logic specifications:

$$(I, O : S(H'))(H) \text{ is translated to } S(H),$$

$$(P_0 \parallel P_1)(H) \text{ abbreviates } P_0(H) \wedge P_1(H),$$

$$(P \backslash \alpha)(H) \text{ abbreviates } \exists H' : His(\beta(P,H)).\ (P(H') \wedge H = H' \restriction (\beta(P,H) \setminus \alpha),$$

$$(P[d/c])(H) \text{ abbreviates } \exists H' : His(\beta(P,H)).\ (P(H') \wedge H = H'[d/c]).$$

Another important abbreviation here is the "*sat*" relation between mixed terms:

$$S_0(H)\ sat\ S_1(H) \text{ abbreviates } \forall H : His(\beta(S_0) \cup \beta(S_1)).\ S_0 \Rightarrow S_1.$$

Quiescent trace specifications are, informally, specifications of processes P that are always enabled for input along all input channels $I(P)$. A process P is in a quiescent state iff it cannot produce any more output unless it receives more input. A quiescent trace is a trace of communications (possibly) leading to a quiescent state, or an infinite trace. As is known from the literature [Jon85] both safety and liveness properties can be specified in terms of the quiescent traces of processes. Moreover, in [Jon85] it is shown that within this context parallel composition can be (essentially) seen as logical conjunction. In order that a specification of the form $(I, O : S(H))$ defines a proper set of quiescent traces we require two conditions (taken from [Jon85]):

(i) In any state (not just quiescent ones), any input action e is enabled, and

(ii) From any state, a quiescent state can be reached by means of (possibly infinitely many) output actions alone.

Put formally: For any event e with $chan(e) \in I$:

$$\forall H, h : His(I \cup O).\ (S(H) \wedge h \leq H \wedge \#h < \infty) \Rightarrow \exists h' : His(O).\ S(h \bullet \{e\} \bullet h')$$

$$\forall H, h : His(I \cup O).\ (S(H) \wedge h \leq H \wedge \#h < \infty) \Rightarrow \exists h' : His(O).\ S(h \bullet h').$$

6.4.2 Specifying sequential consistency

In this section we will give some examples of specifications. First take a buffer process *Buff* with input channel *Enq* and output channel *Deq*. Since *Buff* must always be input enabled, it only can be a infinite or zero-place buffer. A usual CSP-style specification of this buffer is

$$\forall h \leq H.\ (val(h \restriction Deq) \leq val(h \restriction Enq))$$

That is, the values sent along channel *Deq* must be prefix of the values sent along channel *Enq*. For *Buff* to be quiescent it must be empty (if not it is output enabled), i.e.

$$val(H\,|`Deq) = val(H\,|`Enq).$$

or equivalently, using the first assumption

$$\#Deq = \#Enq$$

In general the channels *Enq* and *Deq* are not dependent, but n^{th} message along channel *Enq* must precede the n^{th} message along channel *Deq*. Recall that as a global model property any two different events which correspond to communications along the same channel are dependent. Summarizing the above remarks we see that the specification consists of two parts. The first part is a trace/run specification, in this case

$$\begin{aligned} &TraceBuff(H) \stackrel{\text{def}}{=} \\ &\quad \forall h \leq H.\ (val(h\,|`Deq) \leq val(h\,|`Enq)) \ \wedge\ \#Deq = \#Enq \end{aligned}$$

The dependencies are specified in the second part

$$\begin{aligned} &DepBuff \stackrel{\text{def}}{=} \\ &\quad Dep = \{\{Deq\}, \{Enq\}\} \ \wedge\ \forall j, 0 < j \leq (\#Enq).\ (Enq(j) \longrightarrow Deq(j)) \end{aligned}$$

Along the same lines one can specify a coherent memory *CM* with write channels W_i, read-request channels Req_i (both input channels) and read-return channels Ret_i, $i \in I$, as follows. The trace specification is

$$\begin{aligned} &TraceCM(H) \stackrel{\text{def}}{=} \\ &\quad \forall H' : His(\{W_i, Req_i, Ret_i \mid i \in I\}).\ (\#H' < \infty \wedge H' \bullet Ret_i(d, a) \leq H \Rightarrow \\ &\qquad\qquad d = val(last(H'\,|\{W_I(\cdot, a)\}))\ ) \ \wedge \\ &\quad (\forall h \leq H.(\ addr(h\,|Ret_i) \leq addr(h\,|`Req_i))) \ \wedge\ addr(H\,|`Req_i) \ =\ addr(H\,|`Ret_i) \end{aligned}$$

Observe again that the last clause specifies quiescence. The dependency specification is

$$\begin{aligned} &DepCM(H) \stackrel{\text{def}}{=} \\ &\quad Dep = \{\{W_i, Req_i, Ret_i \mid i \in I\}\} \ \wedge\ \forall i \in I, \forall j, 0 < j \leq \#Ret_i.\ (Req_i(j) \longrightarrow Ret_i(j)) \end{aligned}$$

Now the total specification is

$$CM \stackrel{\text{def}}{=} TraceCM \ \wedge\ DepCM$$

Given the specification of a coherent memory, we can specify a sequentially consistent memory *SCM* by

$$\begin{aligned} &TraceSCM(H) \stackrel{\text{def}}{=} \\ &\quad \exists H'.\ (TraceCM(H') \ \wedge\ \forall i \in I.\ (H\,|\{W_i, Req_i, Ret_i\} = H'\,|\{W_i, Req_i, Ret_i\}) \end{aligned}$$

$$\begin{aligned} &DepSCM(H) \stackrel{\text{def}}{=} \\ &\quad Dep = \{\{W_i, Req_i, Ret_i\} \mid i \in I\} \ \wedge\ \forall i \in I, \forall j, 0 < j \leq \#Ret_i.\ (Req_i(j) \longrightarrow Ret_i(j)) \end{aligned}$$

Note the difference between the channel dependencies for *CM* and *SCM*: for $i \neq j$ W_i is independent of W_j, in $DepSCM$ but not in $DepCM$. Now the total specification for *SCM* reads

$$SCM \stackrel{\text{def}}{=} TraceSCM \ \wedge \ DepSCM$$

Along the same lines one can specify a write-coherent memory *WCM* in which all the write channels are dependent. Let

$$W_I \stackrel{\text{def}}{=} \{W_i \mid i \in I\} \text{ and } R_I \stackrel{\text{def}}{=} \{Ret_i, Req_i \mid i \in I\}$$

$$\begin{aligned} &TraceWCM(H) \stackrel{\text{def}}{=} \\ &\quad \exists H' : His(W_I \cup R_I).\ (TraceCM(H') \wedge H' \restriction W_I = H \restriction W_I \wedge \\ &\qquad \forall i \in I.\ (H' \restriction \{W_i, Req_i, Ret_i\} \ = \ H \restriction \{W_i, Req_i, Ret_i\})\ ). \end{aligned}$$

and

$$\begin{aligned} &DepWCM(H) \stackrel{\text{def}}{=} \\ &\quad Dep = \{\{W_i, Req_i, Ret_i\} \mid i \in I\} \cup \{\{W_i \mid i \in I\}\} \ \wedge \\ &\qquad \forall i, j, 0 < j \leq \#Ret_i.\ (Req_i(j) \rightarrowtail Ret_i(j)) \end{aligned}$$

6.5 Correctness proofs

In this section we give detailed proofs of the claims made.

6.5.1 Queueing and sequential consistency

In this section we look at the behavior of a coherent or sequentially consistent memory with certain "queuing-protocols" on top of it. More precisely, we have a coherent memory or a sequentially consistent memory with write $\tilde{W}_i$, read-request $\tilde{Req}_i$, and read-return channels $\tilde{Ret}_i$ for every user i, $i \in I$. This memory is composed in parallel with separate, so called WR-protocols, which forms for each user i the interface between $\{W_i, Req_i, Ret_i\}$ and $\{\tilde{W}_i, \tilde{Req}_i, \tilde{Ret}_i\}$, see figure 6.7.

This protocol can be seen as the specification of two separate infinite queues: one queue which queues the write and read-request actions, and another queue which queues the read-return actions. Such a write-read protocol satisfies the following specification WRP. For a channel c let $c(k)$ denote the k^{th} communication along channel c, i.e. $c(k) = (H \restriction c)(k)$ We have input channels $\{W, Req, \tilde{Ret}\}$, output channels $\{\tilde{W}, \tilde{Req}, Ret\}$. The specification of *WRP* consists of two parts: the trace specification $TraceWRP$, and the dependency specification *DepWRP*.

$$\begin{aligned} &TraceWRP(H) \stackrel{\text{def}}{=} \\ &\quad \forall h \leq H.\ (h \restriction \{\tilde{W}, \tilde{Req}\} \leq h \restriction \{W, Req\} \ \wedge \ h \restriction Ret \leq h \restriction \tilde{Ret}) \ \wedge \\ &\qquad \#W = \#\tilde{W} \ \wedge \ \#Req = \#\tilde{Req} \ \wedge \ \#Ret = \#\tilde{Ret}, \end{aligned}$$

$$\begin{aligned} &DepWRP(H) \stackrel{\text{def}}{=} \\ &\quad Dep = \{\{W_i, Req_i, Ret_i\}, \{\tilde{W}_i, \tilde{Req}_i, \tilde{Ret}_i\}\} \ \wedge \\ &\quad \forall 0 < j \leq \#W_i.\ (W_i(j) \rightarrowtail \tilde{W}_i(j)) \ \wedge \ \forall 0 < j \leq \#Req_i.\ (Req_i(j) \rightarrowtail \tilde{Req}_i(j)) \ \wedge \\ &\quad \forall 0 < j \leq \#Ret_i.\ (\tilde{Ret}_i(j) \rightarrowtail Ret_i(j)) \end{aligned}$$

The total specification is given by:

$$WRP \stackrel{\text{def}}{=} TraceWRP \ \wedge \ DepWRP$$

Observe that the first part of the *TraceWRP* specification is like a CSP (prefix) specification, and is needed to ensure the input enabledness. The second part of *TraceWRP* defines the quiescent states.

Assume that every user i which communicates by the interface WRP_i with the memory along the channels W_i, Req_i, Ret_i always waits after a read-request for the corresponding read-return before executing another write or read-request action. Thus user i should satisfy

$$\begin{aligned} &User_i(H) \stackrel{\text{def}}{=} \\ &\quad (\forall a, a' \in H.\, (Req(a) \wedge (W(a') \vee Req(a')) \wedge a \longrightarrow a') \Rightarrow \\ &\qquad (\exists a'' \in H.\, (Ret(a'') \wedge a \longrightarrow a'' \longrightarrow a'))) \ \wedge \\ &\quad Dep = \{\{W_i, Req_i, Ret_i\}\} \end{aligned}$$

For a specification S let $\tilde{S} \stackrel{\text{def}}{=} S[\tilde{c}/c \mid c \in \alpha(S)]$, i.e. every channel c in the alphabet of S is renamed into $\tilde{c}$. Similarly, for a set of channels A, $\tilde{A} \stackrel{\text{def}}{=} \{\tilde{a} \mid a \in A\}$. (We assume $\tilde{\tilde{a}} = \tilde{a}$.)

For two runs, say H and H', we use the notation $H \cong H'$ defined as $H = \tilde{H}'$, where again $\tilde{H}'$ is the run H with every channel c renamed into $\tilde{c}$.

Now take a sequentially consistent memory $\widetilde{SCM}$ which has input channels $\{\tilde{W}_i, \tilde{Req}_i \mid i \in I\}$ and output channels $\{\tilde{Ret}_i \mid i \in I\}$. Assume that every user i communicates with this memory via the interface WRP_i, where

$$WRP_i \stackrel{\text{def}}{=} WRP[W_i/W, Req_i/Req, Ret_i/Ret, \tilde{W}_i/\tilde{W}, \tilde{Req}_i/\tilde{Req}, \tilde{Ret}_i/\tilde{Ret}],$$

as given in figure 6.7. Moreover assume that the behavior of each user i satisfies $User_i$. Then the claim is that after projection on the external channels $\{W_i, Req_i, Ret_i \mid i \in I\}$ this composed process behaves like a sequentially consistent memory.

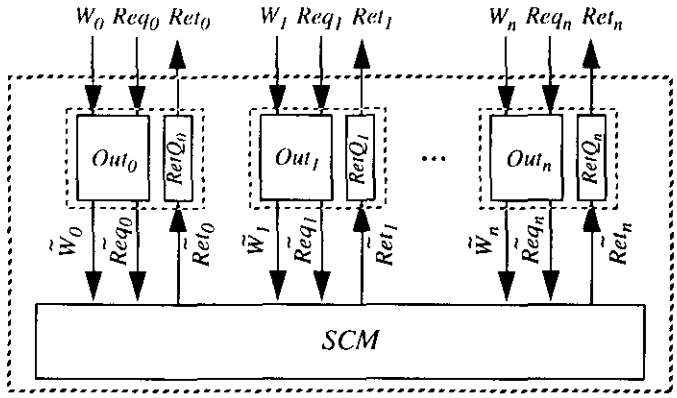


Figure 6.7: A coherent memory with WR-protocols.

To put it formally let $A_i \stackrel{\text{def}}{=} \{W_i, Req_i, Ret_i\}$ and $A = \bigcup_i A_i$

Theorem 6.5.1 *Sequentially consistent memories with WR-protocols*
Let $A = \{W_i, Req_i, Ret_i \mid i \in I\}$

$$P \stackrel{\text{def}}{=} (\widetilde{SCM} \parallel WRP_0 \parallel \cdots \parallel WRP_n)\backslash\tilde{A},$$

then under the assumption $\bigwedge_i \ User_i(H\restriction\{W_i, Req_i, Ret_i\})$ we have

$$P \ sat \ SCM$$

□

Proof. First observe that the dependency specifications of P and SCM are correct, i.e. $P \Rightarrow DepSCM$. Thus it is sufficient to show that $P \Rightarrow TraceSCM$

$$
\begin{array}{ll}
 & P(H) \\
\Rightarrow & \quad \{ \text{ by the definition of hiding and parallel composition } \} \\
 & \exists H_0 : His(A \cup \tilde{A}).\ H\restriction A = H_0\restriction A \wedge \widetilde{SCM}(H_0\restriction\tilde{A}) \wedge \bigwedge_i WRP_i(H_0\restriction(A_i \cup \tilde{A}_i)) \\
\Rightarrow & \quad \{ \text{ by definition of } \widetilde{SCM} \ \} \\
 & \exists H_0 : His(A \cup \tilde{A}).\ \exists H_1 : His(\tilde{A}).\ \widetilde{CM}(H_1\restriction\tilde{A}) \wedge H\restriction A = H_0\restriction A \wedge \\
 & \qquad \forall i \in I.\ (H_0\restriction\tilde{A}_i = H_1\restriction\tilde{A}_i \ \wedge \ WRP_i(H_0\restriction(A_i \cup \tilde{A}_i)) \\
\Rightarrow & \quad \{ \text{ by the lemma below } H_0\restriction A_i \cong H_0\restriction\tilde{A}_i \ \} \\
 & \exists H_0 : His(A \cup \tilde{A}).\ \exists H_1 : His(\tilde{A}).\ \widetilde{CM}(H_1\restriction\tilde{A}) \wedge H\restriction A = H_0\restriction A \wedge \\
 & \qquad \forall i \in I.\ (H_0\restriction\tilde{A}_i = H_1\restriction\tilde{A}_i \wedge H_0\restriction A_i \cong H_0\restriction\tilde{A}_i) \\
\Rightarrow & \quad \{ \text{ by the identities } H\restriction A = H_0\restriction A \text{ and } H_0\restriction\tilde{A}_i = H_1\restriction\tilde{A}_i \ \} \\
 & \exists H_0 : His(A \cup \tilde{A}).\ \exists H_1 : His(\tilde{A}).\ \widetilde{CM}(H_1\restriction\tilde{A}) \wedge \\
 & \qquad \forall i \in I.\ (H_0\restriction\tilde{A}_i = H_1\restriction\tilde{A}_i \wedge H\restriction A_i \cong H_1\restriction\tilde{A}_i) \\
\Rightarrow & \quad \{ \text{ propositional calculus } \} \\
 & \exists H_1 : His(\tilde{A}).\ \widetilde{CM}(H_1\restriction\tilde{A}) \wedge \forall i \in I.\ (H\restriction A_i \cong H_1\restriction\tilde{A}_i) \\
\Rightarrow & \quad \{ \text{ by renaming } \} \\
 & \exists H_2 : His(A).\ CM(H_2\restriction A) \wedge \forall i \in I.\ (H\restriction A_i = H_2\restriction A_i) \\
\Rightarrow & \quad \{ \text{ by definition of } SCM \ \} \\
 & TraceSCM(H\restriction A).
\end{array}
$$

Left to prove, in the above setting, the identity

$$H_0\restriction\{W_i, Req_i, Ret_i\} \cong H_0\restriction\{\tilde{W}_i, \tilde{Req}_i, \tilde{Ret}_i\}.$$

Lemma 6.5.2
Let H_0 be a run such that

1. $H_0\restriction\tilde{A}_i = H_1\restriction\tilde{A}_i \ \wedge \ \widetilde{CM}(H_1\restriction\tilde{A})$
2. $H\restriction A = H_0\restriction A \ \wedge \ User_i(H\restriction A)$
3. $WRP_i(H_0\restriction\{W_i, \tilde{W}_i, Req_i, \tilde{Req}_i, Ret_i, \tilde{Ret}_i\})$

Then

$$H_0\restriction\{W_i, Req_i, Ret_i\} \cong H_0\restriction\{\tilde{W}_i, \tilde{Req}_i, \tilde{Ret}_i\}$$

□

The proof of this lemma can be found in section 6.7.

Since every coherent memory is also a sequentially consistent memory, i.e. $CM \Rightarrow SCM$ we deduce as a corollary from the above theorem:

Corollary 6.5.3
A module consisting of a *coherent memory* $\widetilde{CM}$ with input channels $\{\tilde{W}_i, \tilde{Req}_i \mid i \in I\}$ and output channels $\{\tilde{Ret}_i \mid i \in I\}$, such that every user i satisfies $User_i$ and communicates with this memory via the interface WRP_i, behaves as a *sequentially consistent memory.* □

6.5.2 Replication of write-coherent memories

In the informal explanation we argued that write-coherent memories can be replicated in some sense, again resulting in a write-coherent memory. This replication is done by mapping writes of a process i to writes on all memories, and by mapping any request/return pair of a process onto a request/return pair on a single, fixed memory. The number of memories and the number of users need not be the same.

The structure of the system, as an example with three memories and four user processes is shown in figure 6.8. The user processes read from one of K, $K > 0$, write-coherent memories, WCM_l

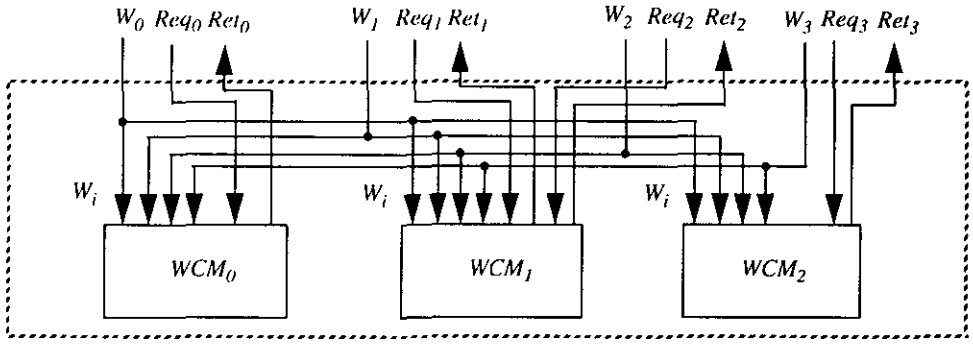


Figure 6.8: A replicated write-coherent memory with $N = 4, K = 3$.

for $0 \leq l < K$. Different users can read different memories, but every process reads from a fixed one. So a component WCM_l has an interface W_I, R_{J_l}, i.e. $WCM_l \stackrel{\text{def}}{=} WCM \restriction (W_I \cup R_{J_l})$, where $R_{J_l} \stackrel{\text{def}}{=} \{Req_i, Ret_i \mid i \in J_l\}$, $J_l \subseteq I$, and $J_l \cap J_{l'} = \emptyset$ for $l \neq l'$. Furthermore

$$\bigcup \{J_l \mid 0 \leq l < K\} = I. \tag{6.1}$$

We now have to prove that the replicated system as a whole behaves as a write-coherent memory.

Theorem 6.5.4 *Replication of write-coherent memories*
Let *Repl* be the replicated system, i.e.

$$Repl \stackrel{\text{def}}{=} (WCM_0 \parallel WCM_1 \parallel \cdots \parallel WCM_{K-1}),$$

then

$$Repl \; sat \; WCM.$$

□

Proof. By the definition of parallelism we have the following proof obligation

$$\bigwedge_{0 \le l < K} WCM(H \upharpoonright (W_I \cup R_{J_l})) \quad \Rightarrow \quad WCM(H \upharpoonright (W_I \cup R_I)) \tag{6.2}$$

Observe that by equation 6.1, *Repl⇒DepWCM*. Thus it is sufficient to show that

$$Repl \Rightarrow TraceWCM.$$

By the definition of $WCM(H)$ we must show the existence of a run H' such that $CM(H')$ holds and some ordering is preserved. Informally, we take the order of all writes from H', and the order of requests and returns Req_i and Ret_i with respect to all writes W_j at the memory WCM_l, such that $i \in J_l$. As the order of writes at all memories is the same, this defines a coherent run.

We use the following merging lemma for runs (proven in section 6.7.

Lemma 6.5.5
Let $H_0 = (A_0, \rightarrow_0)$ and $H_1 = (A_1, \rightarrow_1)$ be two runs, and let $A = A_0 \cap A_1$. If $H_0 \upharpoonright A = H_1 \upharpoonright A$, and $H_0 \upharpoonright A$ is *linear* in the sense that for any two actions there exists a chain between them, and furthermore $(A_0 - A)$ and $(A_1 - A)$ are *independent,* we have the following. For

$$H \stackrel{\text{def}}{=} (H_0 \cup H_1, \rightarrow_0 \cup \rightarrow_1).$$

H is a run, and $H \upharpoonright A_i = H_i$, for $i = 0, 1$. □

We furthermore use the following corollary for runs of a coherent memory.

Corollary 6.5.6
Let $H_0 : His(W_I \cup R_J)$, $H_0 = (A_0, \rightarrow_0)$ and $H_1 : His(W_I \cup R_{J'})$, $H_1 = (A_1, \rightarrow_1)$ such that $J \cap J' = \emptyset$, and assume H_0 and H_1 are both runs of a coherent memory, and $H_0 \upharpoonright W_I = H_1 \upharpoonright W_I$.
Then for $H \stackrel{\text{def}}{=} (A_0 \cup A_1, \rightarrow_0 \cup \rightarrow_1)$, we have that H is a run, $CM(H)$ holds, and $H \upharpoonright (W_I \cup R_J) = H_0 \upharpoonright (W_I \cup R_J)$ and $H \upharpoonright (W_I \cup R_{J'}) = H_0 \upharpoonright (W_I \cup R_{J'})$.

Proof. We apply lemma 6.5.5 to H_0 and H_1. (Linearity on W_I follows from the fact that all W_i are dependent, and therefore ordered, and R_J and $R_{J'}$ are independent.) This gives that H is a run. To see that $CM(H)$ holds, let $H' \bullet Ret_i(d, a) \le H$. Assume $i \in J$. Then, by monotonicity of projection

$$(H' \bullet Ret_i(d,a)) \upharpoonright (W_I \cup R_J) \;=\; (H' \upharpoonright (W_I \cup R_J)) \bullet Ret_i(d,a) \le H_0,$$

which gives by $CM(H_0)$ that

$$val(last(H' \upharpoonright W_I(\cdot, a))) \;=\; val(last((H' \upharpoonright (W_I \cup R_J)) \upharpoonright W_I(\cdot, a)) \;=\; d.$$

The case for $i \in J'$ follows by symmetry. The prefix properties follow directly from the fact that $H' \upharpoonright R_J = H_0 \upharpoonright R_J$ and $H' \upharpoonright R_{J'} = H_0 \upharpoonright R_{J'}$, plus the fact that $CM(H_0)$ and $CM(H_1)$ hold. □

Now we continue with the proof of the theorem.

$$\begin{array}{ll} & \bigwedge_{0 \le l < K} WCM(H \upharpoonright (W_I \cup R_{J_l})) \\ \Rightarrow & \quad \{ \text{ definition } WCM \} \\ & \exists H_0 : His(W_I \cup R_{J_0}).\ CM(H_0) \wedge H_0 \upharpoonright W_I = H \upharpoonright W_I \wedge \end{array}$$

$$
\begin{array}{ll}
 & \forall i \in J_0.\ H_0 \restriction \{W_i, Rcq_i, Ret_i\} \ = \ H \restriction \{W_i, Req_i, Ret_i\} \ \wedge \\
 & \qquad \vdots \\
 & \exists H_{K-1} : His(W_I \cup R_{J_{K-1}}).\ CM(H_{K-1}) \ \wedge \ H_{K-1} \restriction W_I \ = \ H \restriction W_I \ \wedge \\
 & \qquad \forall i \in J_{K-1}.\ H_{K-1} \restriction \{W_i, Req_i, Ret_i\} \ = \ H \restriction \{W_i, Req_i, Ret_i\} \\
\Rightarrow & \quad \{ \text{ corollary 6.5.6 } \} \\
 & \exists H' : His(W_I \cup R_I).\ CM(H') \ \wedge \ H' \restriction W_I = H \restriction W_I \ \wedge \\
 & \qquad \forall 0 \leq l < K.\ \forall i \in J_l.\ H' \restriction (W_i \cup R_i) \ = \ H \restriction (W_i \cup R_i) \\
\Rightarrow & \quad \{ \text{ equality 6.1 } \} \\
 & \exists H' : His(W_I \cup R_I).\ CM(H') \ \wedge \ H' \restriction W_I = H \restriction W_I \ \wedge \\
 & \qquad \forall i \in I.\ H' \restriction (W_i \cup R_i) \ = \ H \restriction (W_i \cup R_i) \\
\Rightarrow & \quad \{ \text{ definition of } WCM \} \\
 & TraceWCM(H)
\end{array}
$$

6.5.3 Queuing and write coherency

In the previous section we used write-coherent memories as blocks to be replicated. How do we get these write-coherent memories? Afek, Brown, and Merritt [ABM93] use a coherent memory CM_i where all processes write into, but only a single process reads from, and a special queue for the writes, the so-called *in-queue*. In this queue the writes of process i are tagged with a star, and read-requests of process i on the memory will only be executed when there are no starred writes in the queue.

We take a more general approach and show that the combination of a coherent memory with N writers and possibly N readers plus a special interface gives a write-coherent memory, under the assumption that every user always waits after a read-request for the corresponding read-return before executing another write or read-request action. The interface queues all writes in a single queue, the read-requests as well as the read-returns in separate queues for each user. Moreover a read-request of process i will only be transferred by the interface to the memory when there are no i-labeled writes in the write-queue.

The proof that this results in a write-coherent memory resembles the proof for sequential consistent memories. We give a specification of the tagged queue behaviour, and a specification of the coherent memory, and show that the result satisfies *WCM*. The structure of the system in sketched in figure 6.9. The formal specification *InQ* of the interface is as follows.

$$
\begin{array}{lll}
TraceInQ(H) & \stackrel{\text{def}}{=} & H \restriction W_I \cong (H \restriction \tilde{W}_I) \ \wedge \\
 & & \forall i \in I.\ (TraceWRP_i(H \restriction \{W_i, \tilde{W}_i, Req_i, \tilde{Req}_i, Ret_i, \tilde{Ret}_i\})),
\end{array}
$$

and

$$
DepInQ \stackrel{\text{def}}{=} Dep = \{\{W_i \mid i \in I\}, \{\tilde{W}_i, \tilde{Req}_i, \tilde{Ret}_i \mid i \in I\}\} \ \wedge \bigwedge_i DepWRP_i.
$$

Thus

$$
InQ \stackrel{\text{def}}{=} TraceInQ \ \wedge \ DepInQ
$$

Furthermore the memory module satisfies (by renaming) $\widetilde{CM}(H)$, so the system as a whole is given by

$$
QCM \stackrel{\text{def}}{=} (\widetilde{CM} \parallel InQ) \backslash_{\overline{(W_I \cup R_I)}}
$$

where again $R_I = \{Req_i, Ret_i \mid i \in I\}$.

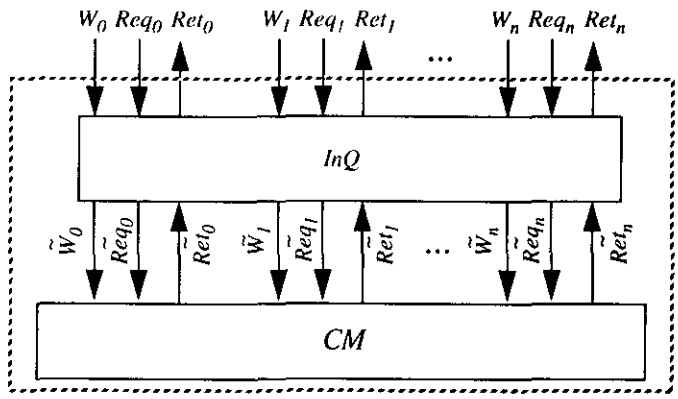


Figure 6.9: A write-coherent memory.

Theorem 6.5.7 *Write coherent memories*
In the above setting and under assumption $\bigwedge_i User_i(H \restriction \{W_i, Req_i, Ret_i\})$ we have

$$QCM \text{ sat } WCM$$

□

Proof. First observe that

$$DepInQ\backslash_{\overline{(W_I \cup R_I)}} \Rightarrow DepWCM.$$

Hence it is sufficient to prove that

$$QCM \Rightarrow TraceWCM$$

By the definition of hiding, this leads to the following proof obligation.

$$\exists H_0 : His(A \cup \tilde{A}).\ H \restriction A = H_0 \restriction A \wedge \widetilde{CM}(H_0) \wedge InQ(H_0) \Rightarrow WCM(H).$$

The proof can be given along the same lines as the proof for the WR-protocols, Theorem 6.5.1. Define A_i, $\tilde{A}_i$, etc. as in section 6.5.1.

$$\begin{array}{ll}
 & \exists H_0.\ H \restriction (W_I \cup R_I) = H_0 \restriction (W_I \cup R_I) \wedge \widetilde{CM}(H_0) \wedge InQ(H_0) \\
\Rightarrow & \quad \{ \text{ by definition } InQ(H_0) \} \\
 & \exists H_0 : His(A \cup \tilde{A}).\ \widetilde{CM}(H_0 \restriction \tilde{A}) \wedge H \restriction A = H_0 \restriction A \wedge H_0 \restriction W_I \cong H_0 \restriction \tilde{W}_I \wedge \\
 & \forall i \in I.\ (WRP_i(H \restriction (A_i \cup \tilde{A}_i)) \\
\Rightarrow & \quad \{ \text{ By lemma 6.5.2 } H_0 \restriction A_i \cong H_0 \restriction \tilde{A}_i \} \\
 & \exists H_0 : His(A \cup \tilde{A}).\ \widetilde{CM}(H_0 \restriction \tilde{A}) \wedge H \restriction A = H_0 \restriction A \wedge H_0 \restriction W_I \cong H_0 \restriction \tilde{W}_I \wedge \\
 & \forall i \in I.\ (H_0 \restriction A_i \cong H_0 \restriction \tilde{A}_i) \\
\Rightarrow & \quad \{ \text{ By the indentity } H \restriction A = H_0 \restriction A \ \} \\
 & \exists H_0 : His(A \cup \tilde{A}).\ \widetilde{CM}(H_0 \restriction \tilde{A}) \wedge H \restriction W_I \cong H_0 \restriction \tilde{W}_I \wedge \\
 & \forall i \in I.\ (H \restriction A_i \cong H_0 \restriction \tilde{A}_i) \\
\Rightarrow & \quad \{ \text{ by projection and renaming } \} \\
 & \exists H_I : His(A).\ CM(H_I \restriction A) \wedge H \restriction W_I = H_I \restriction W_I \wedge \\
 & \forall i \in I.\ (H \restriction A_i = H_I \restriction A_i) \\
\Rightarrow & \quad \{ \text{ by definition } WCM(H) \} \\
 & TraceWCM(H)
\end{array}$$

6.5.4 Adding cache misses

Up until now we assumed every (write-coherent) memory can immediately access any address. In the actual cache memories this need not be the case. Due to limited storage capacity, some addresses might be removed from the cache, in favour of others that are needed. In that case however, we need some "back up" memory that still has the most recent values of all memory locations, from which we can fetch that value if it is requested by some read action. The invalidation of cache addresses is not modeled explicitly.

We therefore add an extra K plus first memory component *Mem* to the system, which models that back up memory. We assume it behaves as a *coherent* memory, except for some renaming. We cannot use a write-coherent memory for this purpose as the order of all read request/return pairs and writes must be preserved. Different updates for different caches should all give the most recent value. It has N write ports W_i, for $i \in I$, and K read request and read return ports, called $CReq_j$ and CU_j, for $0 \leq j < K$. CU stands for Cache Update, as the reads are needed to update missing cache addresses. In fact, we could have several such back up memories, with fewer than K caches reading from each of them. Here we use a single back up memory, but the proof for multiple back up memories is almost literally the same. An example of a system is given in figure 6.10. We have to modify the

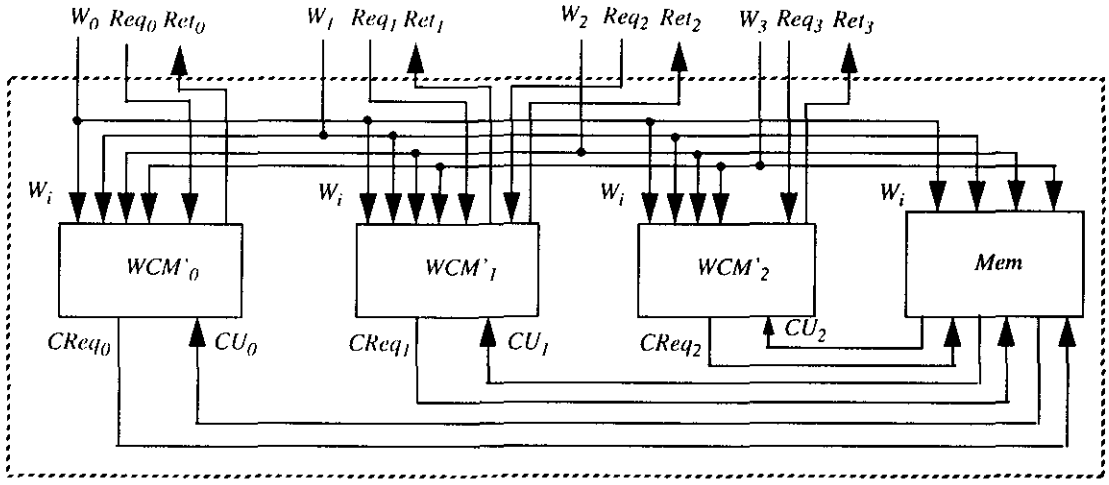


Figure 6.10: A replicated write-coherent memory with cache updates

write-coherent memories WCM_l slightly. They get an extra request port $CReq_l$ and cache update port CU_l as well, which behaves as a write, that is, it is dependent on all W_i, and any Ret_i reads the last value written by a write or by a cache update. We call such a component WCM'_l.

We have that the combination of a number of such WCM'_l components and a memory component behaves as a write-coherent memory. We do so for the combination of a single WCM' and a memory. The general case can be proven analogously, using the replication result.

The specification of WCM' is as follows, cf. section 6.4.2

$$WCM' \stackrel{\text{def}}{=} TraceWCM' \wedge DepWCM'$$

with

$$TraceWCM'(H) \stackrel{\text{def}}{=} \exists H' : His(\{W, CU, CReq, Req, Ret\}).$$

$$(TraceCM'(H') \wedge H' \upharpoonright \{W, CU\} = H \upharpoonright \{W, CU\} \wedge$$
$$(H' \upharpoonright \{W, Req, Ret\} = H \upharpoonright \{W, Req, Ret\})),$$

and

$$DepWCM'(H) \stackrel{\text{def}}{=}$$
$$Dep = \{\{W, CU\}, \{CU, CReq\}, \{W, Ret, Req\}\} \wedge$$
$$\forall 0 < j \leq \#Ret.\ Req(j) \rightarrowtail Ret(j)) \wedge \forall 0 < j \leq \#CU.\ CReq(j) \rightarrowtail CU(j)$$

The coherent memory CM' is defined by

$$TraceCM'(H) \stackrel{\text{def}}{=}$$
$$\forall h \bullet Ret(d, a) \leq H.\ a = val(last(H \upharpoonright \{W(\cdot, a), CU(\cdot, a)\})) \wedge$$
$$(\forall h \leq H.(\ addr(h \uparrow Ret) \leq addr(h \uparrow Req))) \wedge addr(H \upharpoonright Req) = addr(H \upharpoonright Ret)$$

and

$$DepCM' \stackrel{\text{def}}{=} Dep = \{\{W, CU, Req, Ret\}\} \wedge \forall 0 < j \leq \#CU.\ CReq(j) \rightarrowtail CU(j)$$

Thus

$$CM' \stackrel{\text{def}}{=} TraceCM' \wedge DepCM'$$

The memory component *Mem* of the system is a coherent memory with alphabet $\{W, CReq, CU\}$ and is given by following specification: take $I = \{0\}$ in the specification for *CM*, then

$$Mem \stackrel{\text{def}}{=} CM[W/W_0, CReq/Req_0, CU/Ret_0]$$

We prove that

$$(WCM' \parallel Mem) \backslash \{CReq, CU\} \ sat \ WCM.$$

First observe that

$$(WCM' \parallel Mem) \backslash \{CReq, CU\} \Rightarrow DepWCM,$$

thus it is sufficient to show that

$$(WCM' \parallel Mem) \backslash \{CReq, CU\} \Rightarrow TraceWCM$$

The proof is given as follows. Let $C = \{W, Req, Ret, CReq, CU\}$.

$$(WCM' \parallel Mem) \backslash \{CReq, CU\}$$
$$\Rightarrow \quad \{ \text{ by definition } \}$$
$$\exists H_0 : His(C).\ H \upharpoonright \{W, Req, Ret\} = H_0 \upharpoonright \{W, Req, Ret\} \wedge WCM'(H_0) \wedge Mem(H_0)$$
$$\Rightarrow \quad \{ \text{ definition } WCM', Mem \ \}$$
$$\exists H_0 : His(C).\ H \upharpoonright \{W, Req, Ret\} = H_0 \upharpoonright \{W, Req, Ret\} \wedge$$
$$\exists H_1 : His(C).\ CM'(H_1) \wedge H_1 \upharpoonright \{W, CU\} = H_0 \upharpoonright \{W, CU\} \wedge$$
$$(H_1 \upharpoonright \{W, Req, Ret\} = H_0 \upharpoonright \{W, Req, Ret\}) \wedge$$
$$\forall h \bullet CU(d, a) \leq H_0.\ a = val(last(H_0 \upharpoonright \{W(\cdot, a)\}))$$

$$
\begin{array}{ll}
\Rightarrow & \{ \text{ definition } CM', \text{calculus } \} \\
& \exists H_1 : His(C).\ H \mid\{W, Req, Ret\} = H_1 \mid\{W, Req, Ret\} \ \wedge \\
& \quad H_1 \restriction\{W, CU\} = H \restriction\{W, CU\} \wedge \\
& \quad \forall h \bullet Ret(d,a) \leq H_1.\ a = val(last(H_1 \mid\{W(\cdot,a), CU(\cdot,a)\})) \ \wedge \\
& \quad \forall h \bullet CU(d,a) \leq H_1.\ a = val(last(H_1 \restriction W(\cdot,a))) \\
\Rightarrow & \{ \text{ calculus } \} \\
& \exists H_1 : His(C).\ H \mid\{W, Req, Ret\} = H_1 \mid\{W, Req, Ret\} \ \wedge \\
& \quad H_1 \restriction\{W, CU\} = H \restriction\{W, CU\} \wedge \\
& \quad \forall h \bullet Ret(d,a) \leq H_1.\ a = val(last(H_1 \restriction W(\cdot,a))) \\
\Rightarrow & \{ \text{ take } H_2 \ = \ H_1 \restriction\{W, Req, Ret\} \ \} \\
& \exists H_2 : His(\{W, Req, Ret\}).\ CM(H_2) \wedge H \mid\{W, Req, Ret\} = H_2 \mid\{W, Req, Ret\} \\
\Rightarrow & \{ \text{ by definition } \} \\
& Trace\, WCM
\end{array}
$$

which finishes the proof.

6.6 Conclusion

We have introduced a threefold classification of shared memories:

- *Coherent memories* can be thought of as memories where read and write accesses are executed atomically, in some arbitrary, but totally ordered sequence.
- *Sequentially consistent memories* are described by partially ordered traces: read and write accesses stemming from one processor are totally ordered, but accesses stemming from different processors are *unordered*, at least at the external interface of the memory module. Moreover, each of these partially ordered external behaviours can be "linearized" into a behaviour of a coherent memory.
- *Write-coherent memories* are in some sense "in between" coherent memories and sequentially consistent ones. For write accesses, a single linear order is defined at the external interface. For read accesses stemming from some processor P only the relative order with respect to write accesses from P is defined, i.e. P's read accesses are independent of read or write accesses from other processors.

We have decomposed the lazy caching algorithm from [ABM93] into four simple protocols that explain how to build various forms of memories from other memory modules:

- Fifo queues for write accesses and read requests between a processor and a sequentially consistent memory preserve sequential consistency,
- Write-coherent memories can be replicated, while preserving write-coherency.
- When a write-coherent memory is implemented by means of replication, and some replica is actually a coherent memory, rather than just a write-coherent memory, then it one can allow *internal* actions that copy data from that coherent replica to any other replica. This transformation preserves write-coherency for the system as a whole.

- A joint queue can put in between a write-coherent memory and the processors accessing the memory. This queue should act as as a *joint* fifo queue for write actions, and moreover as a an *individual* fifo queue for the write actions and read requests stemming from each processor individually. This transformation, once again, preserves write-coherency of the memory module.

The correctness of these protocols has been shown based within a partial order quiescent trace model. It is clear from the actual proofs that most other trace based formalisms could have been used as well, such as interleaved quiescent traces or CSP style failure traces. In fact, although the detailed proofs would have been quite different, it seems that proofs based on simulation relations between state-transition systems or automata should work too. In all cases, the top level structure of our proof can be retained.

6.7 Proofs of some lemmas

This section gives the proofs of two lemmas, which were omitted from the main text.

First of all we prove lemma 6.5.2:

Let H_0 be a run such that

1. $H_0 \restriction \tilde{A}_i = H_1 \restriction \tilde{A}_i \ \wedge \ \widetilde{CM}(H_1 \restriction \tilde{A})$
2. $H \restriction A = H_0 \restriction A \ \wedge \ User_i(H \restriction A)$
3. $WRP_i(H_0 \restriction \{W_i, \tilde{W}_i, Req_i, \tilde{Req}_i, Ret_i, \tilde{Ret}_i\})$

Then

$$H_0 \restriction \{W_i, Req_i, Ret_i\} \cong H_0 \restriction \{\tilde{W}_i, \tilde{Req}_i, \tilde{Ret}_i\}$$

Proof of the lemma. Because of the assumptions $H_0 \restriction \{W_i, Req_i\} \cong H_0 \restriction \{\tilde{W}_i, \tilde{Req}_i\}$ and $H_0 \restriction Ret_i \cong H_0 \restriction \tilde{Ret}_i$ we only have to prove that the read-request actions are ordered consistently. That is

$$\forall a, a' \in H_0 \restriction \{W_i, Req_i\}.((a \rightarrowtail Ret(k) \rightarrowtail a') \Rightarrow (\tilde{a} \rightarrowtail \tilde{Ret}(k) \rightarrowtail \tilde{a}')$$

where $\tilde{a}$ ($\tilde{a}'$) the action in $H_0 \restriction \{\tilde{W}_i, \tilde{Req}_i\}$ corresponding to a, a' respectively. We will divide the proof in two cases

First case, assume

$$Ret(k) \rightarrowtail a'$$

with $a' \in H_0 \restriction \{W_i, Req_i\}$. Then by WRP_i, both $\tilde{a}'$ and $\tilde{Ret}(k)$ exists and moreover $a' \rightarrowtail \tilde{a}'$, $\tilde{Ret}(k) \rightarrowtail Ret(k)$, cf. figure 6.11. As $\tilde{Ret}(k) \sim \tilde{a}'$ and $\tilde{Ret}(k) \rightarrowtail^+ \tilde{a}'$, we get the arrow *1* in figure 6.11, i.e. $\tilde{Ret}(k) \rightarrowtail \tilde{a}'$. Ordering the other way would result in a cycle. Informally we use the transitivity of the ordering in the run to deduce the arrow *1*, as the resulting trace must be acyclic and dependency closed.

For the second case, assume,

$$a \rightarrowtail Ret(k)$$

with $a \in H_0 \restriction \{W_i, Req_i\}$, cf. figure 6.12. Then by WRP_i, both $\tilde{a}$ and $\tilde{Ret}(k)$ exists and moreover $a \rightarrowtail \tilde{a}$, $\tilde{Ret}(k) \rightarrowtail Ret(k)$ (arrows 1 in figure 6.12).

Now by assumption 1 of the lemma, we deduce the existence of $\tilde{Req}(k)$ and $\tilde{Req}(k) \rightarrowtail \tilde{Ret}(k)$ (arrow 2). By assumption 3, $Req(k)$ exists and $Req(k) \rightarrowtail \tilde{Req}(k)$, hence by "transitivity" $Req(k) \rightarrowtail Ret(k)$.

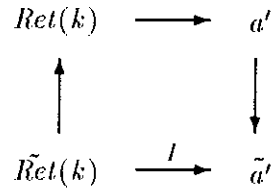


Figure 6.11: The case $Ret(k) \rightarrow a'$

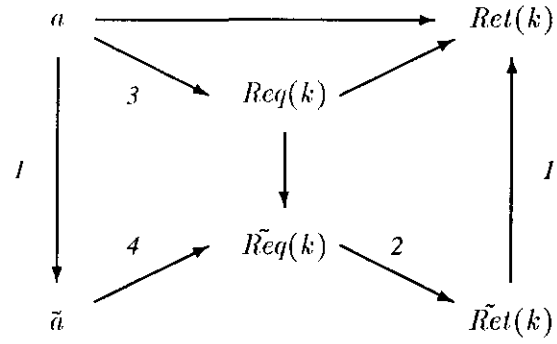


Figure 6.12: The case $a \rightarrow Ret(k)$

Now we claim that $a \rightarrowtail Req(k)$ (arrow 3). This claim is proven at the end. Hence by assumption 3, $\tilde{a} \rightarrowtail \tilde{Req}(k)$ (arrow 4). Now by "transitivity" we get that $\tilde{a} \rightarrowtail \tilde{Ret}(k)$ and we are done.

Left to prove that $a \rightarrowtail Req(k)$. Assume $Req(k) \rightarrowtail a$, then by assumption 2 of the lemma there exists a read-return, say $Ret(j)$, with $Req(k) \rightarrowtail Ret(j) \rightarrowtail a$. Since $a \rightarrowtail Ret(k)$ we deduce $j < k$. Thus we get the situation $Req(j) \rightarrowtail Req(k) \rightarrowtail Ret(j)$. Again we deduce the existence of a read-return $Ret(i)$ with $Req(j) \rightarrowtail Ret(i) \rightarrowtail Req(k)$, thus $i < j < k$. Now we are back at a similar situation $Req(i) \rightarrowtail Req(j) \rightarrowtail Ret(i)$, but now $i < j < k$. Hence by well-foundedness the situation $a \rightarrowtail Req(k)$ cannot occur.

This finishes the proof of the lemma and the theorem. □

Secondly we prove lemma 6.5.5:

Let $H_0 = (A_0, \rightarrow_0)$ and $H_1 = (A_1, \rightarrow_1)$ be two runs, and let $A = A_0 \cap A_1$. If $H_0 \restriction A = H_1 \restriction A$, and $H_0 \restriction A$ is *linear* in the sense that for any two actions there exists a chain between them, and furthermore $(A_0 - A)$ and $(A_1 - A)$ are *independent*, we have the following. For

$$H \stackrel{\text{def}}{=} (H_0 \cup H_1, \rightarrow_0 \cup \rightarrow_1).$$

H is a run, and $H \restriction A_i = H_i$, for $i = 0, 1$.

Proof. The projection property follows directly from the definition of H and the equality $H_0 \restriction A = H_1 \restriction A$.
To prove that H is a run we must prove dependency closedness and acyclicity. As for the former, let $a, b \in (A_0 \cup A_1)$, and assume $a \sim b$. Then $\{a, b\} \subseteq A_0$ or $\{a, b\} \subseteq A_1$. Thus a and b are ordered either by "$\rightarrow_0$" or "$\rightarrow_1$", respectively.
Now assume H contains a cycle. It must be the case that this cycle contains an action $a_0 \in A_0$ and an action $a_1 \in A_1$. Furthermore there exist actions $a, a' \in A$ such that

$$a_0 \rightarrow_0^+ a \rightarrow_1^+ a_1 \rightarrow_1^+ a' \rightarrow_0^+ a_0.$$

But due to linearity of $H_0 \restriction A$ we also have

$$a \rightarrow_0^+ a', \text{ or } a' \rightarrow_0^+ a.$$

The former case gives

$$a_0 \rightarrow_0^+ a \rightarrow_0^+ a' \rightarrow_0^+ a_0,$$

whereas the latter gives (using $H_0 \restriction A = H_1 \restriction A$)

$$a_1 \rightarrow_1^+ a' \rightarrow_1^+ a \rightarrow_1^+ a_1,$$

that is, both result in a cycle in either H_0 or H_1. This contradicts the fact that H_0 and H_1 are runs. □

Chapter 7

Proving Refinement Using Transduction

B. Jonsson, A. Pnueli and C. Rump

7.1 Introduction

Distributed computer systems can be specified at many levels of abstraction. For instance, a specification of a computer network can at one level describe an abstract file transfer service, and at another level include a description of a protocol for transmitting data over a physical link. An important problem is to verify that a (more concrete) lower-level specification correctly implements, or refines, a (more abstract) higher-level one.

Several criteria for the correctness of refinement have been suggested in the literature. A common criterion is based on the idea that a specification denotes a set of allowed observable behaviors, corresponding to different runs of a system. Refinement then corresponds to inclusion between sets of observable behaviors. A verification method should establish that for each computation of the concrete specification, there is an equivalent computation of the abstract one. Several proof methods have been suggested, notably refinement mappings [Lam83, Lam89] and (forward) simulation [Jon87, Jon91, LS90, LT87, Ora89, SL83, Sta88]. These methods are not complete for the case when, intuitively speaking, the abstract system has a nondeterministic choice which occurs earlier than the corresponding nondeterministic choice in the concrete system. For example, a system which outputs 10 a's and then decides to output either a b or a c certainly refines a system which at the beginning decides to output either 10 a's followed by a b or 10 a's followed by a c, but forward simulation is not powerful enough to verify this because at the point of outputting the first a, it is impossible to know which of the alternatives should be chosen in the abstract system in order to match the subsequent choice in the concrete system. The simulation method has therefore been extended with backward simulation or so-called prophecy variables [AL91, HHS87, Jos88, He 89, Jon91] to handle this case.

In this paper, we present a refinement proof method called *proof by transduction*. The main idea of the method is that for a pair of a concrete and an abstract system, we establish refinement by constructing a *transducer* consisting of the concrete system, the abstract system, and a queue of observable events. One must then prove that if the transducer runs through a concrete computation, then it can build up a corresponding abstract computation with some delay. The queue contains the sequence of concrete events which have not yet been matched by abstract ones. In the transducer, one can view the concrete system as a generator of a sequence of events, and the abstract system as an acceptor which accepts a sequences of events, generated by the concrete system, after some unspecified finite delay. In this way, the transducer may defer transitions in the abstract specification until the point in time when the relevant nondeterministic choices have been performed in the concrete system. Thus the method can reduce the number of prophecy variables needed in a proof of refinement. For instance, in the above example, the first step of the abstract system would be delayed until the concrete system has made the choice between b and c.

An important generalization of the transduction method is to prove refinement modulo some transformation of the interface. This kind of refinement has been termed *interface refinement* in [GKS92, BJO91]. A typical transformation could be to replace some observable event that represents a synchronization by a pair of request-confirm events. In the transducer, the queue between the concrete and the abstract system should then be replaced by a more complex component which allows the appropriate transformation on sequences of events.

A particular case of interface refinement is *partial order refinement*, by which we mean refinement that preserves only a subset of the orderings between events of a system. A typical case is sequential consistency in shared memory multiprocessor systems, where the ordering between events associated with each processor is preserved between the concrete and the abstract system. We present a method for the case of specifying and proving sequential consistency. In this method, a transducer is constructed in which the abstract system is an ideal serial memory, and the transducer is a component which

preserves exactly the orderings between events associated with each processor. The totally ordered queue of observable events is replaced by a partially ordered queue, which ensures that the ordering associated with each processor is maintained between the concrete and the abstract systems. In the case of proving sequential consistency, the partially ordered queue essentially consists of one separate totally ordered queue for each processor. The proof of refinement then consists in establishing that for each computation of the concrete system (e.g., a cache consistency protocol) there is a computation of the transducer where all events inserted into the queue by the transducer part corresponding to the concrete system are eventually removed by the part corresponding to the abstract system.

We illustrate the method for proving partial order refinement by applying it to prove sequential consistency of a rather simple cache consistency protocol, the so-called Lazy Caching protocol of Afek, Brown and Merritt [ABM93]. The main advantage of our proof is that whereas the proof of sequentially consistency in e.g., [ABM93] is based on reasoning about entire execution sequences, our proof is more concrete, and uses assertional reasoning about the state of the abstract and concrete system and the queue. The transducer queue can be said to represent the bookkeeping information about entire execution sequences that is needed in a proof like the one in [ABM93]. Thus we feel that our method makes the proof of sequential consistency more concrete and more explicit, in that all structures used in the proof are represented as concrete state variables in a transducer.

The paper is organized as follows. In the following section (7.2), we introduce our system model, a *fair named transition system* as a slightly extended version of a *fair transitions system*, and our notion of refinement. The rest of the paper consists of two parts: The first part introduces the transducer and the proof-by-transduction method for standard language inclusion refinement (section 7.4), presents the rules in Temporal Logic used to do the proofs (section 7.5), and illustrates the method on a simple example (section 7.6). In the second part, we generalize the proof-by-transduction method to handle partial order refinement (section 7.7), then specialize it to treat sequential consistency (section 7.8), and use this to prove the Lazy Cache Algorithm (sections 7.9-7.10).

7.2 System model and notion of refinement

We assume a universal *vocabulary* $\mathcal{V}$ of typed variables. We write $x : D$ to denote that variable x is of type D. We define a *state* s to be a type-consistent interpretation of $\mathcal{V}$, assigning to each variable $u \in \mathcal{V}$ of type D a value $s[u]$ over its domain. We denote by Σ the set of all states.

To express the visible part of the behavior of a system and compare two systems with different system variables, we use a universal set of *events* $\mathcal{E}$. Events can be viewed as an abstract representation of taking a transition.

We model systems as *fair named transition systems* (FNTS). A fair named transition system is a slightly extended version of a *fair transition system* [MP91] in which each transition is associated with an event. Taking a transition is interpreted as an occurrence (generation) of the event associated with the transition. The set of events is partitioned into the set $\mathcal{O}$ of *observable* events and the set $\mathcal{I}$ of *internal* events. An observation of the system is the sequence of observable events generated by a computation of the system.

Fair named transition system

A *fair named transition system* $\mathcal{S}$ is a six-tuple $(V, \Theta, \mathcal{T}, \mathcal{J}, \mathcal{C}, \mathcal{O})$, where

- $V \subseteq \mathcal{V}$ - is a finite set of *system variables*.

- Θ - is the *initial condition*. It is required that Θ be satisfiable, i.e., there exists at least one state satisfying Θ.

- $\mathcal{T}$ - is a (possibly infinite) set of *transitions*. Each transition $\tau \in \mathcal{T}$ is presented as

$$\tau : \rho_\tau(V, V') \quad \text{gen.} \quad \alpha_\tau$$

  where $\alpha_\tau \in \mathcal{E}$ is the event generated by τ, where V' is the set $\{x' \mid x \in V\}$ of primed system variables, and where $\rho_\tau(V, V')$ is an assertion called the *transition relation* which relates a state $s \in \Sigma$ to its possible τ-successors s' by referring to both unprimed and primed versions of the system variables in V. An unprimed version of a system variable refers to its value in s while a primed version of the same variable refers to its value in s'. We say that the transition τ *generates* the event α_τ. For example, a transition τ with $\rho_\tau : x' = x + 1$ and $\alpha_\tau : inc$ has the effect of incrementing the value of x and generating the event inc. Let E denote the set of events generated by the transitions of $\mathcal{S}$. Several transitions may generate the same event.

- $\mathcal{J} \subseteq \mathcal{T}$ is a set of *just* transitions.

- $\mathcal{C} \subseteq \mathcal{T}$ is a set of *compassionate* transitions.

- $\mathcal{O} \subseteq E$ is the set of *observable events*. Thus the set of internal events is $\mathcal{I} = E - \mathcal{O}$.

We require that the idle transition, τ_I, with the transition relation $\rho_{\tau_I} : V' = V$ and generated event $\alpha_I : idle$, always be in $\mathcal{T}$ and $idle \in \mathcal{I}$.

The partitioning of events induces a partitioning of transitions, the set of *observable transitions*, $\mathcal{T}_\mathcal{O} = \{\tau \mid \alpha_\tau \in \mathcal{O}\}$, and the set of *internal transitions*, $\mathcal{T}_\mathcal{I} = \{\tau \mid \alpha_\tau \in \mathcal{I}\}$.

We adopt the convention that all system variables not explicitly mentioned as primed in a transition relation are left unchanged by the relation. Thus, whenever we define a transition relation by means of an assertion ρ whose set of primed variables is U, we regard this as an abbreviation for $\rho_\tau(V, V') = \rho(V, U') \wedge \overline{v}' = \overline{v}$, where $\overline{v} = V - U$.

The transition relation $\rho_\tau(V, V')$ identifies a state s' as a τ-successor of state s if

$$\langle s, s' \rangle \vDash \rho_\tau(V, V'),$$

where $\langle s, s' \rangle$ is the interpretation which interprets $x \in V$ as the value $s[x]$ of x in state s and interprets $x' \in V'$ as $s'[x]$. A transition τ is *enabled* on a state s, written $s \vDash En(\tau)$ if

$$s \vDash \exists V' : \rho_\tau(V, V')$$

which is true iff s has a τ-successor.

A *scenario* of a fair named transition system $\mathcal{S}$ is a pair (σ, β) consisting of a model

$$\sigma : s_0, s_1, s_2, \ldots$$

and an infinite sequence of transitions

$$\beta : \tau_1, \tau_2, \tau_3, \ldots$$

satisfying:

- *Initiation* - s_0 satisfies the initial condition Θ.

- *Consecution* - For all $i \geq 0$, the state s_{i+1} is a τ_{i+1}-successor of the state s_i.
- *Justice* - For each transition $\tau \in \mathcal{J}$, it is not the case that τ is continually enabled at all states beyond some position in σ but appears only finitely many times in β. A transition *appears* finitely many times in β if there are finitely many indices $i_1 < \ldots < i_k$, such that $\tau_{i_j} = \tau$ for all $j = 1, \ldots, k$.
- *Compassion* - For each transition $\tau \in \mathcal{C}$, it is not the case that τ is enabled at infinitely many positions in σ but appears only finitely many times in β.

We refer to σ as the *computation* induced by the scenario, and to β as the *behavior* induced by the scenario.

A *run* of the system is any scenario satisfying the Initiation and Consecution requirements, but not necessarily any of the Justice or Compassion requirements.

An *observation* $\widetilde{\beta}$ corresponding to a behavior β is obtained from β by replacing each transition by the event it generates and then omitting all internal events. Formally, for an infinite sequence of transitions $\beta : \tau_1, \tau_2, \ldots$, let $Events(\beta)$ denote the sequence of the corresponding generated events, $\alpha_{\tau_1}, \alpha_{\tau_2}, \ldots$, and for a sequence X and a set E, let $X\restriction_E$ denote the projection mapping of the sequence X onto the set E.
Let $Obs(\mathcal{S})$ denote the set of all observations of a system $\mathcal{S}$.

Definition 7.2.1 (Refinement) *Given two systems $\mathcal{S}^C$ and $\mathcal{S}^A$, to which we respectively refer as a concrete and an abstract system, we say that the concrete system $\mathcal{S}^C$ refines the abstract system $\mathcal{S}^A$, denoted*

$$\mathcal{S}^C \sqsubseteq \mathcal{S}^A,$$

iff $\mathcal{O}^C = \mathcal{O}^A$ and $\mathrm{Obs}(\mathcal{S}^C) \subseteq \mathrm{Obs}(\mathcal{S}^A)$.

To represent infinite sets of transitions, we introduce the notion of *parameterized transitions*. Parameterized transitions are presented by a *transition scheme* of the form

$$\tau(p_1, \ldots, p_k) : \rho_\tau(p_1, \ldots, p_k, V, V') \quad \text{gen.} \quad \alpha_\tau(p_1, \ldots, p_k)$$

for $p_1 \in D_1, \ldots, p_k \in D_k$, where each p_i is a parameter associated with a particular domain D_i. A transition scheme identifies a (possibly infinite) set of transitions and their corresponding events. Each transition and corresponding event is obtained by selecting a particular instantiation $p_1 : d_1 \in D_1, \ldots, p_k : d_k \in D_k$ of the parameters $\overline{p} = \{p_1, \ldots, p_k\}$. Such an instantiation gives rise to the transition $\tau(d_1, \ldots, d_k)$ and the event $\alpha_\tau(d_1, \ldots, d_k)$.

For simplicity, we assume that only finitely many fair (just or compassionate) transitions are enabled in each state of a computation. This property is referred to as *finite fair enableness*. Our results also hold for the more general case, where a countably infinite number of transitions may be simultaneously enabled, but the proofs will be more involved (see [JPR94]).

A Note on Notation

We use $\bullet$ to denote concatenation of sequences as well as the concatenation of a single element to a sequence. The length of a sequence X is denoted by $|X|$.

As usual, for a sequence X of elements in some domain D, the notation $X[i]$ denotes the i'th element in X provided that $1 \leq i \leq Length(X)$, written $i \in dom(X)$. If $i \notin dom(X)$, $X[i]$ is taken to be $\perp$ which is different from all elements in D.

For a sequence $X \in D^*$ of length $|X| = n > 0$, we denote by $head(X)$ the element $X[1]$ and by $tail(X)$ the sequence $X[2], \ldots, X[n]$, obtained by removing the first element from X. If X is the empty sequence, then $head(X)$ is $\perp$, and $tail(X)$ is the empty sequence.

For an element $y \in D$ and a sequence $X \in D^*$, we use the predicate $y \in X$ as an abbreviation for $\exists\, i \in dom(X) : X[i] = y$.

7.3 Temporal Logic

We use linear time temporal logic [MP91] as a language for expressing properties of computations of fair named transition systems. We assume an underlying first-order language for expressing functions and relations over some standard domains such as the booleans and the integers. A formula in the underlying language is referred to as an *assertion*. We will use a restricted version of temporal logic, which consists of a first-order language augmented by the following temporal operators:

$\Box\, p$ – "p holds at all future positions",

$\Diamond\, p$ – "p holds at some future position",

x^+ – "the value of variable x in the next position",

A *local formula* is a formula in which the only temporal operator is the *next-value* operator $(\ )^+$, applied (once) to variables.

The truth of a temporal formula is evaluated relative to a position $j \geq 0$ in a computation $\sigma : s_0, s_1, s_2, \ldots$ as follows:

- For a formula p without temporal operators,
  $(\sigma, j) \models p$ iff the state s_j at position j in σ satisfies p.
- For a local formula $p(V, V^+)$
  $(\sigma, j) \models p(V, V^+)$ iff $\langle s, s' \rangle \models p(V, V')$, where we interpret $x \in V$ as $s[x]$ and $x' \in V'$ as $s'[x]$,
- $(\sigma, j) \models \Box\, p$ iff $(\sigma, i) \models p$ for all $i \geq j$,
- $(\sigma, j) \models \Diamond\, p$ iff $(\sigma, i) \models p$ for some $i \geq j$,

Boolean operators are evaluated as usual. Two common forms of temporal formulas are

- $\Box \Diamond\, p$ which means that p holds at infinitely many positions, and
- $\Box(p \rightarrow \Diamond\, q)$ which is abbreviated as $p \Rightarrow \Diamond\, q$, meaning that each state s that satisfies p is followed by a state (possibly s itself) that satisfies q.

For a transition τ, define $taken(\tau)$ as the local formula $\rho_\tau(V, V^+)$. We say that transition τ is *taken* at position j of a computation σ if $taken(\tau)$ holds at that position. Note that more than one transition may be considered as taken at position j. This may happen only if both $taken(\tau_1)$ and $taken(\tau_2)$ hold at j.

7.4 Proving Refinement Using a Transducer

In this section, we present the construction of a transducer for proving that some concrete system $\mathcal{S}^{\mathcal{C}}$ refines some abstract system $\mathcal{S}^{\mathcal{A}}$. The transducer is a *fair named transition system* constructed from $\mathcal{S}^{\mathcal{C}}$ and $\mathcal{S}^{\mathcal{A}}$ together with an *interface queue* of events in $\mathcal{O}$ and possibly some auxiliary variables. In the transducer, the concrete system acts as a generator of events which are transferred via the interface queue to the abstract system which in turn acts as an acceptor of sequences of events. To establish a refinement between $\mathcal{S}^{\mathcal{C}}$ and $\mathcal{S}^{\mathcal{A}}$, it must be verified that each sequence of observable events produced by $\mathcal{S}^{\mathcal{C}}$ can be accepted (after some unbounded finite delay induced by the interface queue) by the abstract system $\mathcal{S}^{\mathcal{A}}$. This property clearly implies refinement. In Theorem 7.4.2, we formulate sufficient conditions for this property.

Since we are now referring to two systems, one abstract and one concrete, we use superscript $^{\mathcal{A}}$ ($^{\mathcal{C}}$) when referring to parts of the abstract (concrete) system. Thus $\mathcal{S}^{\mathcal{C}}$ is given as $(V^{\mathcal{C}}, \Theta^{\mathcal{C}}, \mathcal{T}^{\mathcal{C}}, \mathcal{J}^{\mathcal{C}}, \mathcal{C}^{\mathcal{C}})$ and similarly for $\mathcal{S}^{\mathcal{A}}$. The terms *abstract* and *concrete* are sometimes merely used to refer to the systems on the right-hand side and the left-hand side of the refinement relation.

For aesthetic reasons, we refer to relations of the form $\rho_{(\tau^{\mathcal{A}})}$ as $\rho^{\mathcal{A}}_{\tau}$, to relations of the form $\rho_{\widehat{(\tau^{\mathcal{A}})}}$ as $\widehat{\rho^{\mathcal{A}}_{\tau}}$, and to events of the forms $\alpha_{\tau^{\mathcal{A}}}$ and $\alpha_{\tau^{\mathcal{C}}}$ as $\alpha^{\mathcal{A}}_{\tau}$ and $\alpha^{\mathcal{C}}_{\tau}$, respectively.

Definition 7.4.1 (The Refinement Transducer) *Given concrete and abstract systems $\mathcal{S}^{\mathcal{C}}$ and $\mathcal{S}^{\mathcal{A}}$ such that $V^{\mathcal{C}} \cap V^{\mathcal{A}} = \emptyset$ and $\mathcal{O}^{\mathcal{A}} = \mathcal{O}^{\mathcal{C}}$, a* transducer $\mathcal{S}^{T}$ over $\mathcal{S}^{\mathcal{C}}$ and $\mathcal{S}^{\mathcal{A}}$ *is a fair named transition system where*

- $V^{T} = V^{\mathcal{A}} \cup V^{\mathcal{C}} \cup \{Q : (\mathcal{O}^{\mathcal{C}})^{*}\} \cup U$, *i.e. the system variables consists of the system variables of the abstract and the concrete systems, together with a queue (sequence) Q of concrete observable events, and a (possibly empty) set of auxiliary variables, U.*
- Θ^{T}, *the initial condition, is a formula that satisfies:*

$$(\exists\, U : \Theta^{T}) \quad \leftarrow \quad \Theta^{\mathcal{A}} \wedge \Theta^{\mathcal{C}} \wedge Q = \Lambda$$

- $\mathcal{T}^{T}$, *the set of transitions is the union of two sets $\widehat{\mathcal{T}^{\mathcal{C}}}$ and $\widehat{\mathcal{T}^{\mathcal{A}}}$ such that each concrete transition $\tau^{\mathcal{C}} \in \mathcal{T}^{\mathcal{C}}$ has a corresponding transducer transition $\widehat{\tau^{\mathcal{C}}} \in \widehat{\mathcal{T}^{\mathcal{C}}}$ and each abstract transition $\tau^{\mathcal{A}} \in \mathcal{T}^{\mathcal{A}}$ has a corresponding transducer transition $\widehat{\tau^{\mathcal{A}}} \in \widehat{\mathcal{T}^{\mathcal{A}}}$. The transition relations are required to satisfy the following:*
  - *The transition relation $\widehat{\rho^{\mathcal{A}}_{\tau}}$ for the transducer transition $\widehat{\tau^{\mathcal{A}}}$ corresponding to the abstract transition $\tau^{\mathcal{A}}$ should satisfy*

$$\widehat{\rho^{\mathcal{A}}_{\tau}} \quad \leftarrow \quad \rho^{\mathcal{A}}_{\tau} \wedge deQ(\alpha^{\mathcal{A}}_{\tau}) \wedge (V^{\mathcal{C}})' = V^{\mathcal{C}}$$

    where $deQ(\alpha^{\mathcal{A}}_{\tau})$ is defined as $Q = \alpha^{\mathcal{A}}_{\tau} \bullet Q'$ if $\tau^{\mathcal{A}}$ is observable and $Q' = Q$ otherwise. Each transition $\widehat{\tau^{\mathcal{A}}}$ generates the special event null.
  - *The transition relation $\widehat{\rho^{\mathcal{C}}_{\tau}}$ for the transducer transition $\widehat{\tau^{\mathcal{C}}}$ corresponding to the concrete transition $\tau^{\mathcal{C}}$ should satisfy*

$$\rho^{\mathcal{C}}_{\tau} \wedge enQ(\alpha^{\mathcal{C}}_{\tau}) \wedge (V^{\mathcal{A}})' = V^{\mathcal{A}} \quad \leftrightarrow \quad \exists\, U' : \widehat{\rho^{\mathcal{C}}_{\tau}}$$

    where $enQ(\alpha^{\mathcal{C}}_{\tau})$ is defined as $Q' = Q \bullet \alpha^{\mathcal{C}}_{\tau}$ if $\tau^{\mathcal{C}}$ is observable and $Q' = Q$ otherwise. Each transition $\widehat{\tau^{\mathcal{C}}}$ generates the event $\alpha^{\mathcal{C}}_{\tau}$ which is the event generated by $\tau^{\mathcal{C}}$.

- $\mathcal{J}^T \subseteq \widehat{\mathcal{J}^C} \cup \widehat{\mathcal{T}^A}$. *That is, the justice set contains the transitions corresponding to a subset of the just transitions for the concrete level and a subset of the abstract transitions. The mapping* $\widehat{\cdot}$ *is extended to apply to sets of transitions in the obvious way.*

- $\mathcal{C}^T \subseteq \widehat{\mathcal{C}^C} \cup \widehat{\mathcal{T}^A}$. *The compassion set contains the transitions corresponding to a subset of the compassionate transitions for the concrete level and a subset of the abstract transitions.*

- $\mathcal{O}^T = \mathcal{O}^C$, *i.e. the set of observable events equals the set of observable events for* $\mathcal{S}^C$.

The system variables of the transducer are those of $\mathcal{S}^C$ and $\mathcal{S}^A$ together with the interface queue Q and a set U of auxiliary variables. These can be used to restrict (schedule) the occurrences of abstract transitions, and to simplify the proof of the verification conditions, to be presented in Theorem 7.4.2. The auxiliary variables are not allowed to restrict the possible behaviors of the concrete system. This is the motivation for the condition on Θ^T, which states that for each intended initial state of the transducer, satisfying $\Theta^A \wedge \Theta^C \wedge Q = \Lambda$, there are values of U such that Θ^T holds.

The transitions of the transducer are of two kinds: those that correspond to a transition of the concrete system ($\widehat{\mathcal{T}^C}$), and those that correspond to a transition of the abstract system ($\widehat{\mathcal{T}^A}$). The conditions on these ensure that the proper events are inserted and removed from the interface queue when these transitions are taken.

One way to understand the requirements on the transitions in $\mathcal{T}^T$ is to see that they are satisfied if we

- construct $\widehat{\mathcal{T}^C}$ from $\mathcal{T}^C$ by adding an operation that inserts an observable event into Q for each observable transition in $\mathcal{T}^C$. We may also add constraints on the auxiliary variables, if these do not constrain the original concrete transitions.

- construct $\widehat{\mathcal{T}^A}$ from $\mathcal{T}^A$ by adding an operation that removes an observable event from Q for each observable transition in $\mathcal{T}^A$. We may also add any additional constraints on any variables.

Note that since new enableness criteria on abstract transducer transitions may have been introduced as may new fairness requirements, it has to be verified that the transducer satisfy the finite fair enableness property.

7.4.1 Soundness of the Method

We can now (in Theorem 7.4.2) prove a soundness theorem for the transducer recipe. It says that given a concrete and abstract system, $\mathcal{S}^C$ and $\mathcal{S}^A$, if a transducer $\mathcal{S}^T$ over $\mathcal{S}^C$ and $\mathcal{S}^A$ satisfies the three requirements of matching-progress, justice satisfaction, and compassion satisfaction, then $\mathcal{S}^C \sqsubseteq \mathcal{S}^A$. Intuitively, the matching-progress requirement states that whenever the interface queue is nonempty, then there will be a subsequent occurrence of a transition that removes the first element from Q. Note that this transition must be a transition of $\mathcal{S}^A$. The motivation for the justice (compassion) satisfaction requirement is, that since the transducer is allowed to constrain an abstract transition τ^A in any way, the enableness criterion of the corresponding transducer transition $\widehat{\tau^A}$ may be different from those of τ^A. Thus, in general, a justice (compassion) requirement on $\widehat{\tau^A}$ is not enough to ensure a justice (compassion) requirement on τ^A. It must therefore be verified separately that the transducer respects the justice and compassion requirements for the abstract system.

For now, we are only going to sketch the soundness proof of the refinement transducer, and then in section 7.7, we shall give a complete soundness proof of the *partial order refinement transducer* which is a generalization of the refinement transducer.

Theorem 7.4.2 (Soundness) *If a transducer $\mathcal{S}^T$ over $\mathcal{S}^C$ and $\mathcal{S}^A$ satisfies:*

1. *Progress in Matching:*
$$Q \neq \Lambda \quad \Rightarrow \quad \diamondsuit(Q^+ = tail(Q))$$
2. *Justice Satisfaction: For each transition $\tau^A \in \mathcal{T}^A$,*
$$En(\tau^A) \quad \Rightarrow \quad \diamondsuit(\neg En(\tau^A) \vee taken(\widehat{\tau^A}))$$
3. *Compassion Satisfaction: For each transition $\tau^A \in \mathcal{C}^A$,*
$$\square \diamondsuit En(\tau^A) \quad \Rightarrow \quad \square \diamondsuit taken(\widehat{\tau^A})$$

then
$$\mathcal{S}^C \sqsubseteq \mathcal{S}^A$$

Proof: (Sketch only). Let
$$\gamma^C : \tau_1^C, \tau_2^C, \tau_3^C, \ldots$$
be a behavior of $\mathcal{S}^C$. We must prove that there is a behavior γ^A of $\mathcal{S}^A$ which induces the same observation, i.e., $\widetilde{\gamma^C} = \widetilde{\gamma^A}$.
By the conditions on $\mathcal{S}^T$, we conclude that $\mathcal{S}^T$ has a behavior β which is an interleaving of the sequence
$$\widehat{\tau_1^C}, \widehat{\tau_2^C}, \widehat{\tau_3^C}, \ldots$$
with transitions derived from $\widehat{\mathcal{T}^A}$. From the conditions on Q (including matching-progress) in the computation, we infer that the sequence of observable abstract transitions is the same as that of observable concrete transitions. Finally, we use the justice and compassion satisfaction requirements to conclude that the abstract part of β (with the hats ($\widehat{\cdot}$'s) removed) is a behavior of $\mathcal{S}^A$. □

7.5 Proof Rules

In this section, we present temporal proof rules that will be used to prove temporal properties of the form $p \Rightarrow \diamondsuit q$, e.g. matching progress properties of transducers. The rules can be used to infer a temporal conclusion from a list of non-temporal (i.e., first-order) premises. The statement made by a rule is that, if each of the assertional premises holds over all *S-accessible* states (states that may occur in a computation of system S), then the conclusion holds over all computations of S. This implies that, in establishing any of the premises, we may freely employ any previously established invariant of the system. The rules presented here are taken from [MP94].

7.5.1 A Single-Step Rule

A single-step rule, relying on justice, is provided by rule STEP presented in Figure 7.1.

Rule STEP can be used to prove *single-step* response properties, i.e., properties that can be achieved by a single activation of a just transition. The rule calls for the identification of an intermediate assertion φ and a just transition $\tau_h \in \mathcal{J}$, to which we refer as the *helpful transition*. The idea of the rule is to establish that each state that satisfies p is the beginning of a (possibly empty) period that satisfies φ, that φ holds as long as q has not become true, and that (by justice) transition τ_h must eventually be taken and make q true.

For assertions p, q, φ, and transition $\tau_h \in \mathcal{J}$,

J1. $p \rightarrow q \vee \varphi$

J2. $\rho_\tau \wedge \varphi \rightarrow q' \vee \varphi'$ for every $\tau \in \mathcal{T}$

J3. $\rho_{\tau_h} \wedge \varphi \rightarrow q'$

J4. $\varphi \rightarrow En(\tau_h)$

$p \Rightarrow \Diamond q$

Figure 7.1: Rule STEP (single-step response under justice).

Premise J1 of the rule states that, in any position satisfying p, either the goal assertion q already holds, or the intermediate assertion φ, bridging the passage from p to q, holds. The q-disjunct of this premise covers the case that the distance between the p-position and the q-position is 0. The φ-disjunct and the other premises cover the case that the distance between these two positions is positive.

Premise J2 requires that every transition leads from a φ-position to a position that satisfies $q \vee \varphi$. That is, either a position satisfying the goal assertion q is attained or, if not, then at least the intermediate φ is maintained.

Premise J3 requires that the helpful transition τ_h always leads from a φ-position to a q-position.

Premise J4 requires that the helpful transition τ_h is enabled at every φ-position.

Note that, if we have established an invariant ψ of the system, it is sufficient to prove the implication

$$\psi \wedge r \rightarrow s,$$

in order to establish a premise of the form

$$r \rightarrow s.$$

Rule STEP can be used to establish that every state satisfying p (p-state) is followed by a state satisfying another assertion q. The case where q expresses that some transition τ_q must eventually be taken can be expressed by the assertion

$$p \Rightarrow \Diamond taken(\tau_q),$$

Such response properties can be proven by rule J-TAKE of Figure 7.2 which is a variation on rule STEP.

7.5.2 Combining Response Properties

Rule STEP by itself is not a very strong rule, and is sufficient only for proving *one-step* response properties, i.e., properties that can be achieved by a single activation of a helpful transition.

In general, most response properties of the form $p \Rightarrow \Diamond q$ require several helpful steps in order to get from a p-position to a q-position. To establish such properties we may use several rules that enable us to combine response properties, each of which established by a single application of rule STEP. These rules are based on general properties of response formulas that allow us to form these combinations. We list some of these properties as proof rules.

For assertions p, φ, and transitions $\tau_h \in \mathcal{J}$, $\tau_q \in \mathcal{T}$,

J1. $p \rightarrow \varphi$

J2. $\rho_\tau \wedge \varphi \rightarrow \varphi' \vee \rho_{\tau_q}$ for every $\tau \in \mathcal{T}$

J3. $\rho_{\tau_h} \wedge \varphi \rightarrow \rho_{\tau_q}$

J4. $\varphi \rightarrow En(\tau_h)$

$$p \Rightarrow \Diamond taken(\tau_q)$$

Figure 7.2: Rule J-TAKE(eventual activation of transition).

For example, the following rule TRNS (transitivity) infers the response property $p \Rightarrow \Diamond r$ from the two response properties $p \Rightarrow \Diamond q$ and $q \Rightarrow \Diamond r$.

Rule TRNS (transitivity of response)

$$\frac{p \Rightarrow \Diamond q \qquad q \Rightarrow \Diamond r}{p \Rightarrow \Diamond r}$$

Another useful property of response formulas is that it is amenable to proof by cases. This possibility is presented by rule CASES.

Rule CASES (case analysis for response)

$$\frac{p \Rightarrow \Diamond r \qquad q \Rightarrow \Diamond r}{(p \vee q) \Rightarrow \Diamond r}$$

7.5.3 A Well-Founded Rule

The preceding rules can be used to establish response properties that need a bounded number of helpful steps. Some properties may require a number of helpful steps that depend on the state and cannot be bounded a priori. To handle these cases, we introduce a rule that depends on a well-founded domain as a measure of progress towards the goal q.

A *well-founded domain* $(\mathcal{A}, \succ)$ consists of a set $\mathcal{A}$ and a *well-founded* binary relation $\succ$ on $\mathcal{A}$. The relation $\succ$ is called *well-founded* if there does not exist an infinitely descending sequence $a_0, a_1, \ldots$ of elements of $\mathcal{A}$ such that

$$a_0 \succ a_1 \succ \cdots.$$

A typical example of a well-founded domain is $(\mathbb{N}, >)$, where $\mathbb{N}$ are the natural numbers (including

0) and $>$ is the greater-than relation.

Rule WELL, presented in Figure 7.3, can be used to establish a response property requiring an unbounded state-dependent number of helpful steps. The rule uses a well founded domain $(\mathcal{A}, \succ)$ and a *ranking function* δ mapping states to elements of $\mathcal{A}$

For assertions p, q, and φ,
a well-founded domain $(\mathcal{A}, \succ)$, and
a ranking function δ: $\mathcal{S} \mapsto \mathcal{A}$

$$\text{W1.} \quad p \rightarrow q \vee \varphi$$

$$\text{W2.} \quad \varphi \wedge \delta = u \Rightarrow \Diamond(q \vee \varphi \wedge u \succ \delta)$$

$$\overline{p \Rightarrow \Diamond q}$$

Figure 7.3: Rule WELL (well-founded response).

Premise W1 states that every p-state satisfies the goal assertion q or the intermediate assertion φ. In the first case, the goal is achieved within 0 helpful steps.

Premise W2 requires that every φ-state with rank $\delta = u$, is followed by another state which either satisfies q or satisfies φ with a rank lower than u. Since the domain is well-founded, the rank can decrease only finitely many times, ensuring that a q-state is eventually reached.

Rule WELL has as its premise W2, another response formula. This allows a recursive use of the rule, by which the temporal premise W2 is proved either by the simpler rule STEP, or by rule WELL again, only applied to simpler assertions. In many cases, the premise W2 is proved directly by rule STEP. In these cases it is advantageous to replace the temporal premise W2 by the non-temporal premises of rule STEP. This leads to rule S-WELL, presented in Figure 7.4.

The rule uses an intermediate assertion φ to describe the situation between the occurrence of p and the resulting occurrence of q. It also uses the function h which identifies, for each φ-state, a just transition which is helpful for this state. We refer to h, as the *helpful function*. Note that the helpful transition depends on the state.

The rule uses a well-founded domain $(\mathcal{A}, \succ)$ and a ranking function δ mapping states into the set $\mathcal{A}$. The ranking function measures the distance of an intermediate state from a goal state satisfying q. As the computation takes a helpful step, this measure decreases. Due to well-foundedness, the rank cannot decrease forever. Consequently, the computation must eventually reach a q-state.

Premise S1 requires that every p-position, satisfies q or φ.

Premise S2 requires that the application of an arbitrary transition τ to a φ-state s leads to a successor state s' satisfying one of the following:

- s' satisfies q, or
- s' satisfies φ with a rank $\delta(s')$ smaller than $\delta(s)$, or
- s' satisfies φ with a rank $\delta(s')$ equal to $\delta(s)$, and with identical helpful transition $h(s') = h(s)$.

Note that in the case of no observable progress, described by the third clause above, we require the persistence of the helpful transition

Premise S3 requires that the application of the helpful transition h, to a φ-state s, leads to a successor state s' which either satisfies q or satisfies φ with a rank lower than that of s.

Since premise S3 covers the case of $\tau = h$, it is sufficient to establish premise S2 only for $\tau \neq h_i$.

For assertions p, q, and φ,
a well-founded domain $(\mathcal{A}, \succ)$,
ranking function δ: $\mathcal{S} \mapsto \mathcal{A}$, and
helpful function h: $\mathcal{S} \mapsto \mathcal{J}$

S1. $p \rightarrow q \vee \varphi$

S2. $\rho_\tau \wedge \varphi \rightarrow \begin{bmatrix} q' \vee (\varphi' \wedge \delta \succ \delta') \\ \vee (\varphi' \wedge \delta = \delta' \wedge h = h') \end{bmatrix}$
for every $\tau \in \mathcal{T}$

S3. $\rho_h \wedge \varphi \rightarrow q' \vee \varphi' \wedge \delta \succ \delta'$

S4. $\varphi \rightarrow En(h)$

$p \Rightarrow \Diamond q$

Figure 7.4: Rule S-WELL (well-founded response with helpful sets).

Premise S4 requires that the helpful transition is enabled on every every φ-state.

7.6 The Buffer Example

We shall illustrate the proof-by-transduction method by an example. We have chosen a very simple problem to show how proof by transduction can replace the use of prophecy variables.

The example consists of a concrete and an abstract system, both of which operate on a buffer B which ranges over sequences of data messages and have two observable operations: *insert* and *remove*, and one internal operation: *delete*. An $insert(d)$ operation adds a message d to the end of the buffer, and a $remove(d)$ removes element d from the front of the buffer. A *delete* operation deletes an element from the buffer. The difference between the systems is that in the abstract system a $delete^{\mathcal{A}}$ can only remove the last element *inserted* to the buffer, whereas in the concrete system a $delete^{\mathcal{C}}(k)$ removes the k'th element, i.e. any element can be deleted.

Proving $\mathcal{S}^{\mathcal{C}} \sqsubseteq \mathcal{S}^{\mathcal{A}}$ using a regular state to state refinement mapping would require use of prophecy variables. The reason is, that even though one can execute the corresponding abstract $insert^{\mathcal{A}}(d)$ whenever a concrete $insert^{\mathcal{C}}(d)$ is executed, it is not known by that time whether this d element will be deleted, which then requires immediate execution of a $delete^{\mathcal{A}}$ operation in the abstract system, or not.

We assume that messages are taken from some data domain D.

Besides the operations on sequences previously introduced (see section 7.2) we use the following operations for the systems description.

For a sequence $X : D^*$, we shall denote by $drop(k, X)$, for $k \in [1..n]$, the sequence $X[1], \ldots, X[k-1], X[k+1], \ldots, X[n]$, obtained by removing the k'th element from X. For $k \notin [1..n]$, we define $drop(k, X) = X$. We write $last(X)$ for $X[|X|]$ and $rest(X)$ for $drop(|X|, X)$. Obviously, $last(X)$ is the last element of X and $rest(X)$ is the sequence minus its last element.

7.6.1 The Abstract System

We define the abstract system by the following fair named transition system, $\mathcal{S}^{\mathcal{A}}$:

$$\begin{array}{lcl} V^{\mathcal{A}} & = & \{B^{\mathcal{A}} : D^*\} \\ \Theta^{\mathcal{A}} & = & B^{\mathcal{A}} = \Lambda \\ \mathcal{T}^{\mathcal{A}} & = & \{insert^{\mathcal{A}}(d), remove^{\mathcal{A}}(d), delete^{\mathcal{A}}, \mid d \in D\} \\ \mathcal{J}^{\mathcal{A}} & = & \{remove^{\mathcal{A}}(d) \mid d \in D\} \\ \mathcal{O}^{\mathcal{A}} & = & \{insert(d), remove(d) \mid d \in D\} \end{array}$$

where the transitions in $\mathcal{T}^{\mathcal{A}}$ are defined by

$$\begin{array}{lll} insert^{\mathcal{A}}(d) & : (B^{\mathcal{A}})' = B^{\mathcal{A}} \bullet d & \text{gen. } insert(d) \\ remove^{\mathcal{A}}(d) & : B^{\mathcal{A}} = d \bullet (B^{\mathcal{A}})' & \text{gen. } remove(d) \\ delete^{\mathcal{A}} & : (B^{\mathcal{A}})' = rest(B^{\mathcal{A}}) & \text{gen. } delete \end{array}$$

Note that transition $delete^{\mathcal{A}}$ can only remove the last element from the buffer $B^{\mathcal{A}}$.

Since the idle transition is a standard part of any FNTS, we omit its specification from the presentation.

7.6.2 The Concrete System

The concrete system, $\mathcal{S}^{\mathcal{C}}$, is defined by the fair named transition system:

$$\begin{array}{lcl} V^{\mathcal{C}} & = & \{B^{\mathcal{C}} : D^*\} \\ \Theta^{\mathcal{C}} & = & B^{\mathcal{C}} = \Lambda \\ \mathcal{T}^{\mathcal{C}} & = & \{insert^{\mathcal{C}}(d), remove^{\mathcal{C}}(d), delete^{\mathcal{C}}(k), \mid d \in D, k \in \mathbf{N}\} \\ \mathcal{J}^{\mathcal{C}} & = & \{remove^{\mathcal{C}}(d) \mid d \in D\} \\ \mathcal{O}^{\mathcal{C}} & = & \{insert(d), remove(d) \mid d \in D\} \end{array}$$

where the transitions in $\mathcal{T}^{\mathcal{C}}$ are defined by

$$\begin{array}{lll} insert^{\mathcal{C}}(d) & : (B^{\mathcal{C}})' = B^{\mathcal{C}} \bullet d & \text{gen. } insert(d) \\ remove^{\mathcal{C}}(d) & : B^{\mathcal{C}} = d \bullet (B^{\mathcal{C}})' & \text{gen. } remove(d) \\ delete^{\mathcal{C}}(k) & : k \in dom(B^{\mathcal{C}}) \wedge (B^{\mathcal{C}})' = drop(k, B^{\mathcal{C}}) & \text{gen. } delete(k) \end{array}$$

Note that transitions $delete^{\mathcal{C}}(k)$ can delete an element at an arbitrary position of $B^{\mathcal{C}}$.

7.6.3 The Buffer Transducer

The transducer that proves the refinement relation between the concrete and the abstract buffer systems uses two auxiliary variables in addition to the interface queue $Q : (\mathcal{O}^{\mathcal{C}})^*$, ranging over sequences of events.

The abstract system can only delete elements that are at the end of the buffer. Thus boolean variable $depend$ is set to T when such deletion is necessary, indicating the element at the end of $B^{\mathcal{A}}$ should be deleted before any new element is added to $B^{\mathcal{A}}$.

Sequence variable $M : Mark^*$ is used to contain additional information associated with the events listed in Q. The transducer keeps $|Q| = |M|$ so that it is possible to view $M[i]$ as the information associated with the event $Q[i]$. Each entry in M ranges over the enumerated type $Mark : (nil, rem, del)$ with the following intended interpretation:

- For every $Q[i] = remove(d)$, $M[i] = rem$.
- The pair $Q[i] = insert(d)$ and $M[i] = nil$ represents an element that was inserted into $B^{\mathcal{C}}$ by the concrete system and is still in $B^{\mathcal{C}}$. The destiny of such an element is still undecided because it may still be deleted or eventually removed.
- The pair $Q[i] = insert(d)$ and $M[i] = rem$ represents an element that was both inserted into and removed from $B^{\mathcal{C}}$ by the concrete system. Consequently, this element is no longer in $B^{\mathcal{C}}$. The fact that $Q[i] = insert(d)$ implies that the abstract system has not inserted it yet into $B^{\mathcal{A}}$.
- The pair $Q[i] = insert(d)$ and $M[i] = del$ represents an element that was both inserted into and deleted from $B^{\mathcal{C}}$ by the concrete system. Consequently, this element is no longer in $B^{\mathcal{C}}$. The fact that $Q[i] = insert(d)$ implies that the abstract system has not inserted it yet into $B^{\mathcal{A}}$.

Consider a sequence M. Let $i_1, \ldots, i_k$ be the sequence of indices of null entries within M, i.e., $M[i_1] = \cdots = M[i_k] = nil$. The function $freex(j, M)$ computes the index of the j'th null entry within M, i.e., i_j, if $j \leq k$ and returns 0, if $j > k$. We define $firstfree(M)$ to be $freex(1, M)$.

For a sequence M, we use the notation $(M[i] := m)$ to denote the sequence which is identical to M in all elements except for the i'th element which equals m.

The transducer is presented by the transition system shown in figure 7.5. We omit specification of the events generated by the transducer transitions, since they are determined by Definition 7.4.1. Transitions $\widehat{insert}^{\mathcal{A}}(d)$ and $\widehat{remove}^{\mathcal{A}}(d)$ are enabled only when $dpend = \text{F}$. Variable *dpend* is set to T by transition $\widehat{insert}^{\mathcal{A}}(d)$ whenever it inserts an element d with a corresponding M entry *del* which signifies that this element was deleted by the concrete system. Setting *dpend* to T disables all other abstract transitions and enables $\widehat{delete}^{\mathcal{A}}$ which deletes the last element of $B^{\mathcal{A}}$.

Transition $\widehat{insert}^{\mathcal{C}}(d)$ inserts event $insert(d)$ into Q and marks it in M with nil. Transition $\widehat{remove}^{\mathcal{C}}(d)$ inserts event $remove(d)$ into Q with corresponding marking rem but also marks the first null entry in M with rem. As will be shown, the corresponding entry in Q contains the $insert(d)$ event that was responsible for inserting into $B^{\mathcal{C}}$ the element d that is currently removed. Marking it by rem signals to the abstract system that the destiny of this element has just been identified and it is safe to insert this element into $B^{\mathcal{A}}$, knowing it will not be deleted.

Transition $\widehat{delete}^{\mathcal{C}}(k)$ deletes the k'th element from $B^{\mathcal{C}}$ and marks the k'th null insert event as deleted. This signals the abstract system that it is now safe to insert the corresponding element d into $B^{\mathcal{A}}$, provided it will be immediately deleted by $delete^{\mathcal{A}}$.

7.6.4 Proof of the Buffer Transducer

Four invariants are needed to prove the matching-progress and the justice satisfaction of the buffer transducer. To express these invariants, we introduce some additional notation.

Consider a queue $Q \in (\mathcal{O}^{\mathcal{C}})^*$ and a marking sequence $M \in Mark^*$ of the same length. Based on the correspondence between entries in Q and entries in M, we can classify $insert$-entries as follows: An $insert$-entry is called a *null insert*, a *remaining insert*, or a *deleted insert* if its corresponding mark is nil, rem, or del, respectively. An $insert$-entry is called an *undeleted insert* if its corresponding mark is not del. We define $Q\restriction_{nilinsert}$, $Q\restriction_{reminsert}$, and $Q\restriction_{undelinsert}$, to be the sequence of data elements that correspond to null $insert$ entries, remaining $insert$ entries, and undeleted $insert$ entries, respectively. We define $Q\restriction_{remove}$ to be the sequence of data elements corresponding to $remove$-entries in Q.

We define

$$\overline{B}^{\mathcal{A}} = \text{if } dpend \text{ then } rest(B^{\mathcal{A}}) \text{ else } B^{\mathcal{A}}$$

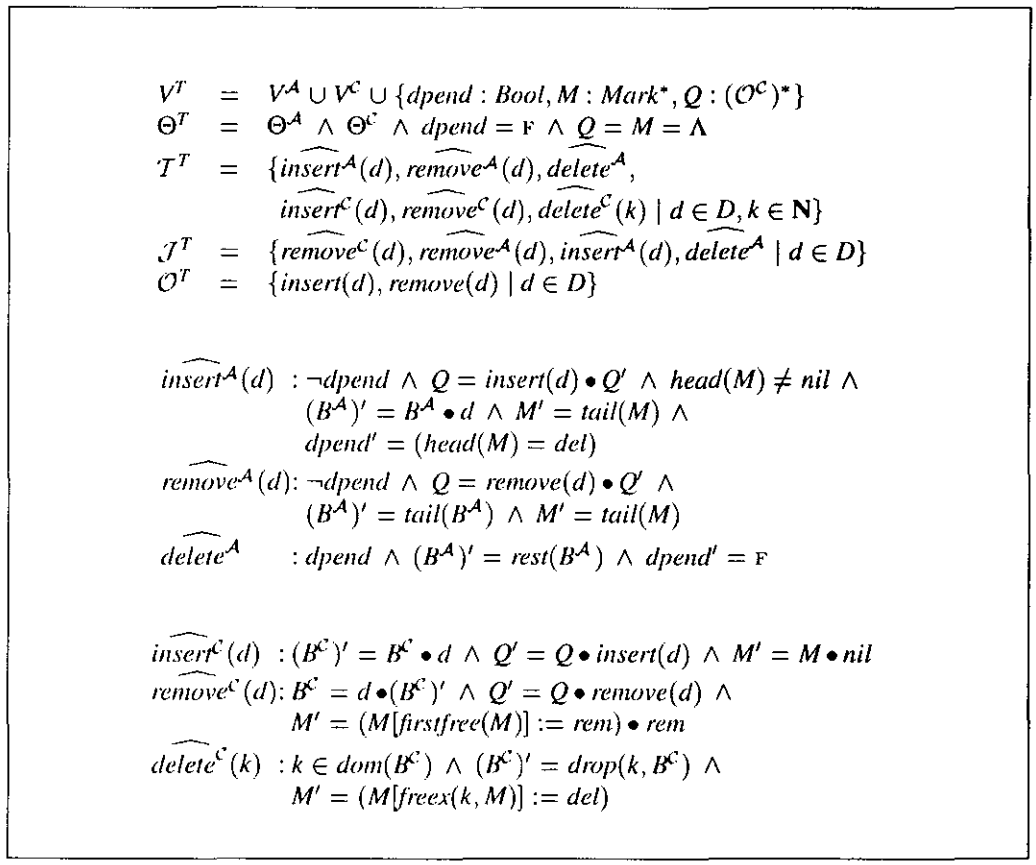


Figure 7.5: The buffer transducer

Thus, $\overline{B}^{\mathcal{A}}$ represents a stable $B^{\mathcal{A}}$, without its last element if this element is soon to be deleted as indicated by a true $dpend$.

The proof of the progress properties required by Theorem 7.4.2 is based on the following invariants:

$$
\begin{array}{ll}
I_1: & |Q| = |M| \\
I_2: & \overline{B}^{\mathcal{A}} \bullet Q\lceil_{reminsert} = Q\lceil_{remove} \\
I_3: & Q\lceil_{undelinsert} = Q\lceil_{reminsert} \bullet Q\lceil_{nilinsert} \\
I_4: & Q\lceil_{nilinsert} = B^{\mathcal{C}}
\end{array}
$$

Invariant I_1 states that Q and M have equal lengths.

Invariant I_2 states that the concatenation of the (stable) abstract buffer to the $insert$ elements marked as remaining yields the same sequence of data elements as those associated with $remove$-entries in Q.

Invariant I_3 states that the sequence of all undeleted $insert$-entries in Q consists of the sequence of remaining $insert$'s followed by the sequence of null $insert$'s. It implies that all remaining $insert$'s precede all null $insert$'s in Q.

Invariant I_4 states that the sequence of elements corresponding to null $insert$'s is identical to the concrete buffer $B^{\mathcal{C}}$.

The method of proving refinement by transduction requires that we establish two response prop-

erties (since we have no compassion requirements). The first requirement is progress in matching and the second is the satisfaction of abstract justice. We will prove each requirement in turn.

7.6.5 Progress in Matching

Progress in matching requires that each observable event that is currently at the head of the queue is eventually removed. This can be formulated by the response formula

$$head(Q) = \alpha \quad \Rightarrow \quad \diamondsuit\, taken(\widehat{\alpha^{\mathcal{A}}}). \tag{7.1}$$

for $\alpha \in \mathcal{O}^{\mathcal{C}}$. In our case, the observable events are all of the form $\nu(d)$ where $\nu \in \{insert, remove\}$. We will prove the progress property in several stages.

First we show that if an observable transition is enabled, implying that the corresponding event must be at the head of Q, it is eventually taken. This property is expressed by the following lemma:

Lemma 7.6.1 $En(\widehat{\nu^{\mathcal{A}}}(d)) \quad \Rightarrow \quad \diamondsuit\, taken(\widehat{\nu^{\mathcal{A}}}(d)) \qquad$ *for* $\nu \in \{insert, remove\}$.

Proof: Both cases are proven by rule J-TAKE, taking $p = \varphi : En(\widehat{\nu^{\mathcal{A}}}(d))$ for the assertions and $\tau_h : \widehat{\nu^{\mathcal{A}}}(d)$ for the helpful transition. The only nontrivial premise is J2. Observing that premise J3 implies premise J2 for $\tau = \tau_h = \widehat{\nu^{\mathcal{A}}}(d)$, it is sufficient to prove, for each $\tau \neq \widehat{\nu^{\mathcal{A}}}(d)$, the implication:

$$\rho_\tau \wedge \underbrace{En(\widehat{\nu^{\mathcal{A}}}(d))}_{\varphi} \quad \rightarrow \quad \underbrace{En'(\widehat{\nu^{\mathcal{A}}}(d))}_{\varphi'}$$

which claims that no transition other than $\widehat{\nu^{\mathcal{A}}}(d)$ can disable $\widehat{\nu^{\mathcal{A}}}(d)$, once it is enabled. For the case $\nu = insert$, the enabling condition is $\neg dpend \wedge head(Q) = insert(d) \wedge head(M) \neq nil$, and it is easily seen that, once true, this can only be changed by the transition $\widehat{insert^{\mathcal{A}}}(d)$. For the case $\nu = remove$, the enabling condition is $\neg dpend \wedge head(Q) = remove(d)$, and, once true, this can only be changed by transition $\widehat{remove^{\mathcal{A}}}(d)$. This establishes that, once an observable abstract transition is enabled, it is eventually taken. □

We will proceed to show that if observable event $\nu(d)$ is at the head of Q then, eventually, $\widehat{\nu^{\mathcal{A}}}(d)$ becomes enabled.

One possible obstacle to the enableness of the two observable abstract transitions is that variable $dpend$ is true. We now establish that, whenever this happens, variable $dpend$ eventually become false. This property is stated by the following lemma, which also guarantees that, when $dpend$ becomes false, the same observable event α is still at the head of Q.

Lemma 7.6.2 $head(Q) = \nu(d) \wedge dpend \quad \Rightarrow \quad \diamondsuit(head(Q) = \nu(d) \wedge \neg dpend)$

Proof: By rule STEP, using the following constructs:

$$\begin{aligned} p = \varphi: &\quad head(Q) = \nu(d) \wedge dpend \\ q: &\quad head(Q) = \nu(d) \wedge \neg dpend \\ \tau_h: &\quad \widehat{delete^{\mathcal{A}}} \end{aligned}$$

□

As the last step in the proof of progress in matching, we show that if $\nu(d)$ is at the head of Q and $dpend = \mathrm{F}$, then $\widehat{\nu^{\mathcal{A}}}(d)$ eventually becomes enabled. This is stated by the following lemma:

Lemma 7.6.3 $head(Q) = \nu(d) \wedge \neg dpend \quad \Rightarrow \quad \Diamond(En(\widehat{\nu^{\mathcal{A}}}(d)))$

Proof: We consider separately the two cases:

$\nu = remove$. Since $En(\widehat{remove^{\mathcal{A}}}(d))$ is $\neg dpend \wedge head(Q) = remove(d)$, the left-hand side of the lemma's claim implies $En(\widehat{remove^{\mathcal{A}}}(d))$ immediately, so the claim is valid.

$\nu = insert$. For this case the enabling condition $En(\widehat{insert^{\mathcal{A}}}(d))$ requires, in addition, that $head(M) \neq nil$. It is, therefore, sufficient to prove

$$\neg dpend \wedge head(Q) = insert(d) \wedge head(M) = nil \quad \Rightarrow \quad \Diamond\Big(\neg dpend \wedge head(Q) = insert(d) \wedge head(M) \neq nil\Big) \tag{7.2}$$

By invariant I_4 : $Q\restriction_{nilinsert} = B^{\mathcal{C}}$, $head(Q) = insert(d) \wedge head(M) = nil$ implies $head(B^{\mathcal{C}}) = d$. The helpful transition $\widehat{remove^{\mathcal{C}}}(d)$ is thus enabled, and will, when taken, append $remove(d)$ to the end of Q and set $M[i]$ to rem, where i is the first null $insert$ in Q. Since $insert(d)$ is a null $insert$ at the head of Q, $i = 1$ and $\widehat{remove^{\mathcal{C}}}(d)$ sets $M[1]$ to rem, achieving $head(M) \neq nil$. The helpful transition $\widehat{remove^{\mathcal{C}}}(d)$ can be disabled by transition $\widehat{delete^{\mathcal{C}}}(1)$, which sets $M[1]$ to del, also achieving $head(M) \neq nil$. Thus, formula (7.2) can be proven by rule STEP, with the following choice:

p: $\neg dpend \wedge head(Q) = insert(d) \wedge head(M) = nil$
φ: $\neg dpend \wedge head(Q) = insert(d) \wedge head(M) = nil \wedge head(B^{\mathcal{C}}) = d$
q: $\neg dpend \wedge head(Q) = insert(d) \wedge head(M) \neq nil$
τ_h: $\widehat{remove^{\mathcal{C}}}(d)$

□

Obviously, Lemmas 7.6.2, 7.6.3, and 7.6.1 establish progress in matching, as expressed by formula (7.1).

7.6.6 Abstract Justice

The second requirement that should be proven is that if a just abstract transition $\alpha^{\mathcal{A}}$ (the original transition of the abstract system, not its transducer counterpart) is enabled, then either it becomes disabled some time later, or the corresponding transducer transition is eventually taken. Since all just abstract transitions are of the form $remove^{\mathcal{A}}(d)$, we have to prove the following formula.

$$\underbrace{head(B^{\mathcal{A}}) = d}_{En(remove^{\mathcal{A}}(d))} \quad \Rightarrow \quad \Diamond\Big(head(B^{\mathcal{A}}) \neq d \vee \widehat{taken(remove^{\mathcal{A}}(d))}\Big)$$

(note that $head(B^{\mathcal{A}}) \neq d$ is implied by $B^{\mathcal{A}} = \Lambda$). In view of Lemma 7.6.1, it is sufficient to prove

$$head(B^{\mathcal{A}}) = d \quad \Rightarrow \quad \Diamond\Big(head(B^{\mathcal{A}}) \neq d \vee head(Q) = remove(d)\Big) \tag{7.3}$$

If $head(B^{\mathcal{A}}) = d$ then either $head(\overline{B}^{\mathcal{A}}) = d$ or $depend = \tau$, $last(B^{\mathcal{A}}) = d$, and $|B^{\mathcal{A}}| = 1$. In the latter case, we can use a proof similar to that of Lemma 7.6.2 to establish

$$head(B^{\mathcal{A}}) = d \quad \Rightarrow \quad \Diamond\Big(head(B^{\mathcal{A}}) \neq d \vee head(\overline{B}^{\mathcal{A}}) = d\Big)$$

We can therefore proceed under the assumption that $head(\overline{B}^{\mathcal{A}}) = d$. By invariant I_2, this implies that $head(Q\restriction_{remove}) = d$ and, therefore, $remove(d) \in Q$. Assume that j is the smallest subscript such that

$Q[j] = remove(d)$. By property (7.1), each element at the head of Q is eventually removed, moving $remove(d)$ closer to the head of Q. Eventually, we will reach a state in which $head(Q) = remove(d)$, as required by property (7.3).

This concludes the proof of the requirement of abstract justice for the buffer transducer.

Since $\mathcal{C}^{\mathcal{A}} = \emptyset$ and the transducer satisfies the matching-progress and justice satisfaction requirements, we conclude that $\mathcal{S}^{\mathcal{C}} \sqsubseteq \mathcal{S}^{\mathcal{A}}$.

7.7 Partial Order Refinement by Transduction

In the previous section, we have used proof-by-transduction to prove inclusion between the sets of observations of two systems. However, our proof method can be used for more general refinement criteria, which are not defined simply as inclusion between sets of observations. An example is a refinement criterion which requires that for each observation $\widetilde{\beta^{\mathcal{C}}}$ of the concrete system, there is an observation $\widetilde{\beta^{\mathcal{A}}}$ of the abstract system, which is in some way related to $\widetilde{\beta^{\mathcal{C}}}$, for instance through some particular transformation. This more general kind of refinement has been termed *interface refinement* in the work by Brinksma, Jonsson, and Orava [BJO91] and by Gerth, Kuiper, and Segers [GKS92]. The more restricted standard refinement criterion corresponds to the special case where the relation between observations is equality.

A particular instance of interface refinement occurs when the transformation is defined to respect partial orderings between observable events (standard refinement respects the total ordering between all observable events in an observation). In this way, it is possible to specify phenomena such as serializability, sequential consistency, etc. The partial ordering can be defined to respect, for each member of some set of observers, the order of events that the particular observer can see. The observers can be taken as the individual processors in the case of sequential consistency, and as the individual transactions in the case of serializability.

In this section, we shall first define a general framework for partial order refinement. We then go on to show how the proof-by-transduction method can be generalized to a proof method also for partial order refinement. The idea of this generalization is to replace the interface queue by a data structure that can attain the appropriate transformation from sequences of input events to sequences of output events. In the next section we show how partial order refinement can be specialized to sequential consistency.

Let E be a set of events. A *dependency D on E* is a reflexive and symmetric relation on E. For two finite or infinite equal-length sequences of events $\beta : e_0, e_1, \ldots$ and $\overline{\beta} : \overline{e}_0, \overline{e}_1, \ldots$ and a permutation $\pi : \{1, \ldots, |\beta|\} \to \{1, \ldots, |\beta|\}$ (in the infinite case $\pi : \mathrm{N} \to \mathrm{N}$), we write $\overline{\beta} = \pi(\beta)$ iff $e_i = \overline{e}_{\pi(i)}$ for every $i = 0, 1, \ldots, |\beta|$. We say that π is *D-respecting on β* if $e_i \; D \; e_j$ implies $\pi(i) < \pi(j)$ whenever $0 \leq i < j \leq |\beta|$. We say that β and $\overline{\beta}$ are *D-equivalent*, written $\beta \approx_D \overline{\beta}$, if there is a D-respecting permutation π on β such that $\overline{\beta} = \pi(\beta)$.

Definition 7.7.1 (Partial Order Refinement) *Let $\mathcal{S}^{\mathcal{C}}$ and $\mathcal{S}^{\mathcal{A}}$ be fair named transition systems, and let D be a dependency on the set $\mathcal{O}$ of observable events. Then $\mathcal{S}^{\mathcal{C}}$ is a partial order refinement of $\mathcal{S}^{\mathcal{A}}$ with respect to D, written $\mathcal{S}^{\mathcal{C}} \sqsubseteq_D \mathcal{S}^{\mathcal{A}}$, iff $\mathcal{O}^{\mathcal{A}} = \mathcal{O}^{\mathcal{C}}$ and for any observation $\widetilde{\beta^{\mathcal{C}}}$ of $\mathcal{S}^{\mathcal{C}}$ there exists an observation $\widetilde{\beta^{\mathcal{A}}}$ of $\mathcal{S}^{\mathcal{A}}$ such that $\widetilde{\beta^{\mathcal{A}}} \approx_D \widetilde{\beta^{\mathcal{C}}}$.*

Sometimes when the dependency is understood, we just write $\mathcal{S}^{\mathcal{C}} \sqsubseteq \mathcal{S}^{\mathcal{A}}$.

The partial order refinement transducer

Partial order refinement can be established by transduction in a way analogous to ordinary refinement. The difference is that the interface queue Q is no longer a totally ordered FIFO queue, but a *partially ordered multi-set* (*pomset*) (see e.g., [Pra86], [Gai89], or [Maz89]).

We assume a universal set of events $\mathcal{E}$ and a given dependency relation D.

A *pomset* over a set of events E is a structure of the form $(C, \mu, <)$ where C is a set called the *carrier*, μ is a mapping from C to E, and $<$ is a partial order over C. A pomset $(C, \mu, <)$ is *D-compatible* if for every $a, b \in C$, $\mu(a)\ D\ \mu(b)$ only if $a < b$ or $b < a$. Note that we may have $a, b, c \in C$ such that $a < b < c$, $\mu(a) = \mu(c)$ and $\mu(a)\ D\ \mu(b)$. Thus the ordering is on the carrier and not on the labeling events. Two pomsets $(C_i, \mu_i, <_i)$, $i = 1, 2$, are isomorphic (equal for all practical purposes) if there exists a bijection $f : C_1 \rightarrow C_2$ such that, for every $a \in C_1$, $\mu_1(a) = \mu_2(f(a))$ and, for every $a, b \in C_1$, $a <_1 b$ iff $f(a) <_2 f(b)$.

The empty pomset is denoted by Λ. Let $X = (C_x, \mu_x, <_x)$ and $Y = (C_y, \mu_y, <_y)$ be pomsets over E. The concatenation of X and Y, written $X \bullet_D Y$, is defined if C_x and C_y are disjoint (if they are not, use disjoint isomorphic copies) and yields the pomset $(C_x \cup C_y, \mu, <)$ where μ is the combined mapping of μ_x and μ_y and $<$ is the transitive closure of $<_x \cup <_y \cup \{a < b \mid a \in C_x, b \in C_y : \mu(a)\ D\ \mu(b)\}$. Obviously, $X \bullet_D Y$ is D-compatible if X and Y are D-compatible pomsets. The restriction of X to a certain set of events $\mathcal{O}$, written $X\restriction_{\mathcal{O}}$, is the pomset $(C'_x, \mu'_x, <'_x)$ where C'_x is the subset $\mu_x^{-1}(\mathcal{O})$ of C_x that is mapped (by μ_x) to events in $\mathcal{O}$, and μ'_x and $<'_x$ are the restrictions of μ_x and $<_x$ to C'_x, respectively.

The $\bullet_D$ operator is similar to the *layered composition* operator in the work by Zwiers et al. (see, e.g., [JZ93]) which originates from the communication-closed layers principle by Elrad and Francez [EF82].

We let $Pomset(E)$ denote the class of all pomsets over the set E of events.

This leads to the following definition of a transducer for proving partial order refinement.

Definition 7.7.2 (The Partial Order Refinement Transducer) *Given concrete and abstract systems $\mathcal{S}^{\mathcal{C}}$ and $\mathcal{S}^{\mathcal{A}}$ such that $V^{\mathcal{C}} \cap V^{\mathcal{A}} = \emptyset$ and $\mathcal{O}^{\mathcal{A}} = \mathcal{O}^{\mathcal{C}}$, a* partial order transducer $\mathcal{S}^T$ over $\mathcal{S}^{\mathcal{C}}$ and $\mathcal{S}^{\mathcal{A}}$ with dependency D *is a fair named transition system where*

- $V^T = V^{\mathcal{A}} \cup V^{\mathcal{C}} \cup \{Q : Pomset(\mathcal{O}^{\mathcal{C}})\} \cup U$, *i.e. the system variables consists of the system variables of the abstract and the concrete systems, together with a pomset Q of concrete observable events, and a (possibly empty) set of auxiliary variables, U.*
- Θ^T, *the initial condition, is a formula that satisfies:*

$$(\exists\, U : \Theta^T) \quad \leftrightarrow \quad \Theta^{\mathcal{A}} \wedge \Theta^{\mathcal{C}} \wedge Q = \Lambda$$

- $\mathcal{T}^T$, *the set of transitions is the union of two sets $\widehat{\mathcal{T}^{\mathcal{C}}}$ and $\widehat{\mathcal{T}^{\mathcal{A}}}$ such that each concrete transition $\tau^{\mathcal{C}} \in \mathcal{T}^{\mathcal{C}}$ has a corresponding transducer transition $\widehat{\tau^{\mathcal{C}}} \in \widehat{\mathcal{T}^{\mathcal{C}}}$ and each abstract transition $\tau^{\mathcal{A}} \in \mathcal{T}^{\mathcal{A}}$ has a corresponding transducer transition $\widehat{\tau^{\mathcal{A}}} \in \widehat{\mathcal{T}^{\mathcal{A}}}$. The transition relations are required to satisfy the following:*
  - *The transition relation $\widehat{\rho_\tau^{\mathcal{A}}}$ for the transducer transition $\widehat{\tau^{\mathcal{A}}}$ corresponding to the abstract transition $\tau^{\mathcal{A}}$ should satisfy*

$$\widehat{\rho_\tau^{\mathcal{A}}} \quad \rightarrow \quad \rho_\tau^{\mathcal{A}} \wedge deQ(\alpha_\tau^{\mathcal{A}}) \wedge (V^{\mathcal{C}})' = V^{\mathcal{C}}$$

    where $deQ(\alpha_\tau^{\mathcal{A}})$ is defined as $Q = \alpha_\tau^{\mathcal{A}} \bullet_D Q'$ if $\tau^{\mathcal{A}}$ is observable and $Q' = Q$ otherwise. Each transition $\widehat{\tau^{\mathcal{A}}}$ generates the special event null.

– *The transition relation* $\widehat{\rho_\tau^{\mathcal{C}}}$ *for the transducer transition* $\widehat{\tau^{\mathcal{C}}}$ *corresponding to the concrete transition* $\tau^{\mathcal{C}}$ *should satisfy*

$$\rho_\tau^{\mathcal{C}} \wedge enQ(\alpha_\tau^{\mathcal{C}}) \wedge (V^{\mathcal{A}})' = V^{\mathcal{A}} \quad \leftrightarrow \quad \exists U' : \widehat{\rho_\tau^{\mathcal{C}}}$$

where $enQ(\alpha_\tau^{\mathcal{C}})$ *is defined as* $Q' = Q \bullet_D \alpha_\tau^{\mathcal{C}}$ *if* $\tau^{\mathcal{C}}$ *is observable and* $Q' = Q$ *otherwise. Each transition* $\widehat{\tau^{\mathcal{C}}}$ *generates the event* $\alpha_\tau^{\mathcal{C}}$ *which is the event generated by* $\tau^{\mathcal{C}}$.

- $\mathcal{J}^T \subseteq \widehat{\mathcal{J}^{\mathcal{C}}} \cup \widehat{\mathcal{T}^{\mathcal{A}}}$. *That is, the justice set contains the transitions corresponding to a subset of the just transitions for the concrete level and a subset of the abstract transitions. The mapping* $\widehat{\cdot}$ *is extended to apply to sets of transitions in the obvious way.*
- $\mathcal{C}^T \subseteq \widehat{\mathcal{C}^{\mathcal{C}}} \cup \widehat{\mathcal{T}^{\mathcal{A}}}$. *The compassion set contains the transitions corresponding to a subset of the compassionate transitions for the concrete level and a subset of the abstract transitions.*
- $\mathcal{O}^T = \mathcal{O}^{\mathcal{C}}$, *i.e. the set of observable events equals the set of observable events for* $\mathcal{S}^{\mathcal{C}}$.

Note that this definition is almost the same as the definition of the standard transducer, the only difference being that the interface queue is a pomset, and that insertions and deletions to the interface queue are made with respect to the dependency relation. Recall that it has to be verified that the transducer satisfies the property of finite fair enableness.

7.7.1 Soundness of the Method

We shall prove a soundness theorem for the partial order refinement transducer which is as similar to the one for the linear case as the similarities in the transducer recipes suggest. The only difference is that it does not suffice to require that a non-empty Q always gets shorter, since there can be several minimal elements in the interface pomset Q at the same time. As a result, it is possible that infinitely many elements are removed from Q, yet some other elements remain continually stuck in Q forever. Instead we specifically make sure that each event in Q is eventually removed.

Theorem 7.7.3 (Soundness of Partial Order Refinement) *If a partial order refinement transducer* $\mathcal{S}^T$ *over* $\mathcal{S}^{\mathcal{C}}$ *and* $\mathcal{S}^{\mathcal{A}}$ *with dependency* D *satisfies:*

1. *Progress in Matching: For each event* $\alpha \in \mathcal{O}^{\mathcal{C}}$, *there exists an abstract transition* $\tau^{\mathcal{A}} \in \mathcal{T}_{\mathcal{O}}^{\mathcal{A}}$ *generating the event* α *such that*

$$\alpha \in Q \quad \Rightarrow \quad \Diamond\, taken(\widehat{\tau^{\mathcal{A}}})$$

2. *Justice Satisfaction: For each transition* $\tau^{\mathcal{A}} \in \mathcal{J}^{\mathcal{A}}$,

$$En(\tau^{\mathcal{A}}) \quad \Rightarrow \quad \Diamond(\neg En(\tau^{\mathcal{A}}) \vee taken(\widehat{\tau^{\mathcal{A}}}))$$

3. *Compassion Satisfaction: For each transition* $\tau^{\mathcal{A}} \in \mathcal{C}^{\mathcal{A}}$,

$$\Box \Diamond En(\tau^{\mathcal{A}}) \quad \Rightarrow \quad \Box \Diamond taken(\widehat{\tau^{\mathcal{A}}})$$

then

$$\mathcal{S}^{\mathcal{C}} \sqsubseteq_{\approx D} \mathcal{S}^{\mathcal{A}}$$

The proof of the theorem follows below.

The behaviors of a transducer $\mathcal{S}^T$ over $\mathcal{S}^{\mathcal{A}}$ and $\mathcal{S}^{\mathcal{C}}$ consist of transitions from the original abstract and concrete systems, $\mathcal{S}^{\mathcal{A}}$ and $\mathcal{S}^{\mathcal{C}}$, but with hats ($\widehat{\cdot}$'s) on. To compare behaviors of $\mathcal{S}^T$ with behaviors in $\mathcal{S}^{\mathcal{A}}$ and $\mathcal{S}^{\mathcal{C}}$, we have to project onto $\widehat{T^{\mathcal{A}}}$ and $\widehat{T^{\mathcal{C}}}$ respectively, and then (syntactically) remove all hats. Given a behavior $\beta : \widehat{\tau_1}, \widehat{\tau_2}, \ldots$, $\natural(\beta)$ yields the corresponding sequence of transitions, but with all hats removed: $\natural(\beta) : \tau_1, \tau_2, \ldots$. For a scenario (σ, β) of $\mathcal{S}^T$, define $\beta^{\mathcal{C}}$ and $\beta^{\mathcal{A}}$ to be the projections of β onto $\widehat{T^{\mathcal{C}}}$ and $\widehat{T^{\mathcal{A}}}$ respectively, and define $\sigma^{\mathcal{C}}$ and $\sigma^{\mathcal{A}}$ to be the projections of σ onto $\Sigma^{\mathcal{C}}$ and $\Sigma^{\mathcal{A}}$ respectively. For every prefix β_i of β, the notation $\beta_i^{\mathcal{C}}$ and $\beta_i^{\mathcal{A}}$ is defined analogously. We let $\widetilde{\beta^{\mathcal{A}}}$ refer to the sequence of observable abstract events $Events(\natural(\beta^{\mathcal{A}}))\restriction_{\mathcal{O}^{\mathcal{A}}}$, i.e., the events of $\natural(\beta^{\mathcal{A}})$ projected onto the observable events of $\mathcal{S}^{\mathcal{A}}$. $\widetilde{\beta_i^{\mathcal{A}}}$, $\widetilde{\beta^{\mathcal{C}}}$, $\widetilde{\beta_i^{\mathcal{C}}}$ etc. are defined in a similar way. We refer to transitions in $\widehat{T^{\mathcal{A}}}$ or $\widehat{T^{\mathcal{C}}}$ as abstract or concrete transducer transitions, respectively.

Before we present the soundness proof, we prove three important lemmas that state properties of partial order refinement transducers. The first lemma, Lemma 7.7.4, states the safety property of the transducer that holds for any prefix $\widetilde{\beta_i}$ of an observation of the transducer: For any linearization q of Q (i.e. a sequence of Q's elements consistent with the partial order of Q), the extension $\widetilde{\beta_i^{\mathcal{A}}} \bullet q$ of the abstract part $\widetilde{\beta_i^{\mathcal{A}}}$ of $\widetilde{\beta_i}$ with q is D-equivalent to $\widetilde{\beta_i^{\mathcal{C}}}$, the concrete part of $\widetilde{\beta_i}$. The sequence q is intended to represent a possible order in which the elements in Q could be removed in the future part of the behavior.

The second lemma, Lemma 7.7.5, states that if a transducer satisfies the matching-progress property for partial order refinement transducers, which says that every event in the interface pomset Q is eventually removed, then $\widetilde{\beta^{\mathcal{A}}} \approx_D \widetilde{\beta^{\mathcal{C}}}$.

The third lemma, Lemma 7.7.6, states that for any scenario $(\sigma^{\mathcal{C}}, \gamma^{\mathcal{C}})$ of the original concrete system, $\mathcal{S}^{\mathcal{C}}$, there exist a scenario (σ, β) of the transducer $\mathcal{S}^T$, such that $\natural(\beta^{\mathcal{C}}) = \gamma^{\mathcal{C}}$, i.e., the transducer can generate all behaviors of the original concrete system.

Having proved these lemmas, what is left to do in the soundness theorem is to state the necessary justice and compassion satisfaction requirements and then simply show that a transducer satisfying these requirements can generate a behavior of the original abstract system, $\mathcal{S}^{\mathcal{A}}$.

Before we move on to proving the lemmas, we introduce some notation.

For a partial function f, let $dom(f)$ be the set of elements x for which $f(x)$ is defined, and let $range(f)$ be the set $\{f(x) \mid x \in dom(f)\}$. A *partial permutation* π is a partial, injective function $\pi : \mathbb{N} \xrightarrow{\sim} \mathbb{N}$ for which $range(\pi) = \{1, \ldots, n\}$ for some $n \in \mathbb{N}_0$. The cardinality of $range(\pi)$ is denoted $size(\pi)$.

For a sequence β and some m let $\beta|_m$ denote the sequence of the m first elements of β (if $m \geq |\beta|$ then $\beta|_m = \beta$). An infinite sequence $\beta_0, \beta_1, \ldots$ of sequences of elements in some domain B, is said to converge to the sequence $\beta : B^\omega$ if for all $m > 0$, there exists $n \geq 0$ such that $\beta_i|_m = \beta|_m$ for all $i \geq n$. In this case we define $\lim_{i\to\infty} \beta_i = \beta$.

Given a transducer $\mathcal{S}^T$ over systems $\mathcal{S}^{\mathcal{A}}$ and $\mathcal{S}^{\mathcal{C}}$, let (σ, β) with $\sigma : s_0, s_1, \ldots$ and $\beta : \tau_1, \tau_2, \ldots$ be a scenario of $\mathcal{S}^T$. Define $\beta_i = \beta|_i$. Then $\lim_{i\to\infty} \beta_i = \beta$, and obviously $\lim_{i\to\infty} \widetilde{\beta_i} = \widetilde{\beta}$. We refer to the value of Q in the state s_i by Q_i.

Let $\widetilde{\beta^{\mathcal{A}}}$ be the sequence $\alpha_1^{\mathcal{A}}\alpha_2^{\mathcal{A}}\alpha_3^{\mathcal{A}}\ldots$ and let $\widetilde{\beta^{\mathcal{C}}}$ be $\alpha_1^{\mathcal{C}}\alpha_2^{\mathcal{C}}\alpha_3^{\mathcal{C}}\ldots$.

Corresponding to each β_i we define a partial function $\pi_i : \mathbf{N} \rightharpoonup \mathbf{N}$ by:

$$\pi_0 = \emptyset \quad \text{(the } \textit{empty mapping}\text{)}$$

$$\pi_{i+1} = \begin{cases} \pi_i \cup \{k \mapsto |\widetilde{\beta_i^{\mathcal{A}}}| + 1\} & \text{if } |\widetilde{\beta_{i+1}^{\mathcal{A}}}| > |\widetilde{\beta_i^{\mathcal{A}}}| = n \text{ and there exists } k \text{ such that} \\ & \quad k \text{ is the least } k \in \mathbf{N} - dom(\pi_i) \text{ for which} \\ & \quad \alpha_{n+1}^{\mathcal{A}} = \alpha_k^{\mathcal{C}} \\ \pi_i & \text{otherwise} \end{cases}$$

Intuitively the π_i's are intended to slowly build a permutation $\widetilde{\pi}$ that can be used to prove that $\widetilde{\beta^{\mathcal{A}}} \approx_D \widetilde{\beta^{\mathcal{C}}}$. Since D is reflexive, the permutation will always map least index to least index whenever two events are equal. We shall show that π_i is a partial permutation for each i, and that its range is $\{1, \ldots, n\}$ where $n = |\widetilde{\beta_i^{\mathcal{A}}}|$.

For partial permutations π_i and π_j we define $\pi_i \subseteq \pi_j$ iff $dom(\pi_i) \subseteq dom(\pi_j)$ and $\forall k \in dom(\pi_i) : \pi_j(k) = \pi_i(k)$.

We are now ready to give the first lemma, which states the necessary safety properties of the transducer.

Lemma 7.7.4 (Transducer Safety) *Given a partial order transducer $\mathcal{S}^T$ over systems $\mathcal{S}^{\mathcal{A}}$ and $\mathcal{S}^{\mathcal{C}}$ with dependency D, let (σ, β) be a run of $\mathcal{S}^T$. If β_i, π_i, and Q_i are as defined above, then for all i,*

1. *π_i is a partial permutation and $size(\pi_i) = |\widetilde{\beta_i^{\mathcal{A}}}|$, and*
2. *for all linearizations q of Q_i, there exists a D-respecting permutation $\pi \supseteq \pi_i$ on $\widetilde{\beta_i^{\mathcal{C}}}$, such that $\widetilde{\beta_i^{\mathcal{A}}} \bullet q = \pi(\widetilde{\beta_i^{\mathcal{C}}})$.*

Proof: We refer to [JPR94] for a proof of the lemma. □

>From the definition of the partial permutations π_i it is now straightforward to infer that the π_i's are monotone increasing:

$$\pi_0 \subseteq \pi_1 \subseteq \pi_2 \subseteq \ldots \tag{7.4}$$

Using this and the safety property of the transducer we can infer that if all events added to the interface queue Q are eventually removed, then the transducer has the following important property: For all behaviors of the transducer there exists a permutation π such that the observation corresponding to the concrete part of the behavior is D-equivalent to the observation corresponding to the abstract part of the behavior.

Lemma 7.7.5 *For any scenario (σ, β) of a partial order transducer $\mathcal{S}^T$ over systems $\mathcal{S}^{\mathcal{A}}$ and $\mathcal{S}^{\mathcal{C}}$ with dependency D, satisfying that for each event $\alpha \in \mathcal{O}^{\mathcal{C}}$, there exists an abstract transition $\tau^{\mathcal{A}} \in \mathcal{T}_{\mathcal{O}}^{\mathcal{A}}$ generating the event α such that*

$$\alpha \in Q \quad \Rightarrow \quad \Diamond\, taken(\widehat{\tau^{\mathcal{A}}}) \tag{7.5}$$

there exist a D-respecting permutation π on $\widetilde{\beta^{\mathcal{C}}}$ such that

$$\widetilde{\beta^{\mathcal{A}}} = \pi(\widetilde{\beta^{\mathcal{C}}})$$

Proof: We refer to [JPR94] for a proof of the lemma. □

Next we show that given a scenario $(\varsigma^{\mathcal{C}}, \gamma^{\mathcal{C}})$ of the original concrete system $\mathcal{S}^{\mathcal{C}}$, there exists a scenario (σ, β) of the transducer such that $\natural(\beta^{\mathcal{C}}) = \gamma^{\mathcal{C}}$. The idea of the proof of this property is to go along and compute the transducer versions of the transitions of $\gamma^{\mathcal{C}}$ and then insert an appropriate number of abstract transducer transitions, or, in other words, to match an appropriate number of abstract transitions, between each such transition. Doing this we only have to worry about fairness requirements for transitions in $\widehat{\mathcal{T}^{\mathcal{A}}}$. To ensure that all fairness requirements are met, we use a slightly modified version of a standard scheduler for justice and compassion.

Lemma 7.7.6 *Given a transducer $\mathcal{S}^T$ over $\mathcal{S}^{\mathcal{A}}$ and $\mathcal{S}^{\mathcal{C}}$. For any scenario $(\varsigma^{\mathcal{C}}, \gamma^{\mathcal{C}})$ of $\mathcal{S}^{\mathcal{C}}$, there exists a scenario (σ, β) of $\mathcal{S}^T$ such that $\natural(\beta^{\mathcal{C}}) = \gamma^{\mathcal{C}}$.*

Proof: We refer to [JPR94] for a proof of the lemma. □

Let $(\varsigma^{\mathcal{C}}, \gamma^{\mathcal{C}})$ be a scenario of a transducer $\mathcal{S}^{\mathcal{C}}$ which satisfy the matching-progress, justice satisfaction, and compassion satisfaction properties given in Theorem 7.7.3 above. By Lemma 7.7.6, we have that there exists a scenario (σ, β) of $\mathcal{S}^T$ such that $\natural(\beta^{\mathcal{C}}) = \gamma^{\mathcal{C}}$. Since $\mathcal{S}^T$ satisfies the matching-progress property 1, we conclude by Lemma 7.7.5 that $\widetilde{\beta^{\mathcal{A}}} \approx_D \widetilde{\beta^{\mathcal{C}}}$. It thus remains to be shown, using the justice and compassion satisfaction requirements, that there exists a scenario $(\varsigma^{\mathcal{A}}, \gamma^{\mathcal{A}})$ of $\mathcal{S}^{\mathcal{A}}$ such that $\natural(\beta^{\mathcal{A}}) = \gamma^{\mathcal{A}}$. This is stated in the following lemma.

Lemma 7.7.7 *For any scenario (σ, β) of a partial order transducer $\mathcal{S}^T$ over systems $\mathcal{S}^{\mathcal{A}}$ and $\mathcal{S}^{\mathcal{C}}$ with dependency D, satisfying that for each transition $\tau^{\mathcal{A}} \in \mathcal{T}^{\mathcal{A}}$,*

$$En(\tau^{\mathcal{A}}) \quad \Rightarrow \quad \Diamond(\neg En(\tau^{\mathcal{A}}) \vee taken(\widehat{\tau^{\mathcal{A}}})) \tag{7.6}$$

and for each transition $\tau^{\mathcal{A}} \in \mathcal{C}^{\mathcal{A}}$,

$$\Box \Diamond En(\tau^{\mathcal{A}}) \quad \Rightarrow \quad \Box \Diamond taken(\widehat{\tau^{\mathcal{A}}}) \tag{7.7}$$

there exists a scenario $(\varsigma^{\mathcal{A}}, \gamma^{\mathcal{A}})$ of $\mathcal{S}^{\mathcal{A}}$ such that $\natural(\beta^{\mathcal{A}}) = \gamma^{\mathcal{A}}$.

Proof: We shall show that actually $(\sigma^{\mathcal{A}}, \natural(\beta^{\mathcal{A}}))$ is a scenario of $\mathcal{S}^{\mathcal{A}}$. By the transducer requirements on transitions,

$$\widehat{\rho_\tau^{\mathcal{A}}} \quad \rightarrow \quad \rho_\tau^{\mathcal{A}} \wedge \ldots$$

and

$$\exists U' : \widehat{\rho_\tau^{\mathcal{C}}} \quad \rightarrow \quad \ldots \wedge (V^{\mathcal{A}})' = V^{\mathcal{A}}$$

we immediately conclude that $(\sigma^{\mathcal{A}}, \natural(\beta^{\mathcal{A}}))$ is a run of $\mathcal{S}^{\mathcal{A}}$. Since $\mathcal{S}^T$ satisfies properties (7.6) and (7.7) above, we conclude that $(\sigma^{\mathcal{A}}, \natural(\beta^{\mathcal{A}}))$ is a scenario of $\mathcal{S}^{\mathcal{A}}$. □

It is now straightforward to deduce from lemmas 7.7.4–7.7.7 the soundness of the partial order refinement transducer as stated in Theorem 7.7.3.

7.8 Sequential Consistency

In this section, we describe how specification and verification of sequential consistency can be seen as a special case of partial order refinement, which can be established using a partial order transducer.

We assume that we are given two fair named transition systems $\mathcal{S}^{\mathcal{A}}$ and $\mathcal{S}^{\mathcal{C}}$ which specify systems that are used by a set of processes $P_1, \ldots, P_n$. The processes interact with the system by means of

events in the same set $\mathcal{E}$, and do not communicate among themselves by any other means than via the system specified by $\mathcal{S}^{\mathcal{A}}$ or $\mathcal{S}^{\mathcal{C}}$. Let $\mathcal{O}_i$ be the set of events that represent the interaction between P_i and the system. We assume that the sets $\mathcal{O}_1, \ldots, \mathcal{O}_n$ are pairwise disjoint, and that their union is the set of observable events for both $\mathcal{S}^{\mathcal{C}}$ and $\mathcal{S}^{\mathcal{A}}$.

The system $\mathcal{S}^{\mathcal{C}}$ is said to be sequentially consistent with $\mathcal{S}^{\mathcal{A}}$ [Lam79] if for each behavior $\beta^{\mathcal{C}}$ of $\mathcal{S}^{\mathcal{C}}$, there is a behavior $\beta^{\mathcal{A}}$ of $\mathcal{S}^{\mathcal{A}}$ such that

$$\widetilde{\beta^{\mathcal{C}}}\lceil_{\mathcal{O}_i} \approx_D \widetilde{\beta^{\mathcal{A}}}\lceil_{\mathcal{O}_i} \quad \text{for each } i = 1, \ldots, n$$

The property of sequential consistency can be rephrased in terms of partial order refinement as follows.

Theorem 7.8.1 (Sequential Consistency) *Let the dependency relation D be an equivalence relation which relates two events iff they belong to the same set $\mathcal{O}_i$ for some i. The system $\mathcal{S}^{\mathcal{C}}$ is sequentially consistent with $\mathcal{S}^{\mathcal{A}}$ iff $\mathcal{S}^{\mathcal{C}} \sqsubseteq_D \mathcal{S}^{\mathcal{A}}$.*

Proof: Straight-forward. □

Theorem 7.8.1 prescribes a systematic method for verifying sequential consistency. To prove that a system $\mathcal{S}^{\mathcal{C}}$ is sequentially consistent with another system $\mathcal{S}^{\mathcal{A}}$, we build a transducer comprising $\mathcal{S}^{\mathcal{C}}$, $\mathcal{S}^{\mathcal{A}}$, and a pomset Q. The pomset Q orders two events if and only if they are dependent. Since the dependency relation D is an equivalence relation with one equivalence class for each index i, the pomset Q defines a total ordering of the events indexed by i. One can therefore think of Q as consisting of one queue Q_i for each set i. The queue Q_i is a linearly ordered sequence of events in $\mathcal{O}_i$. The queues are independent of each other.

In Figure 7.6, we draw a schematic picture of the structure of the partial order transducer for verifying that $\mathcal{S}^{\mathcal{C}}$ is sequentially consistent with $\mathcal{S}^{\mathcal{A}}$. The transducer consists of the concrete and

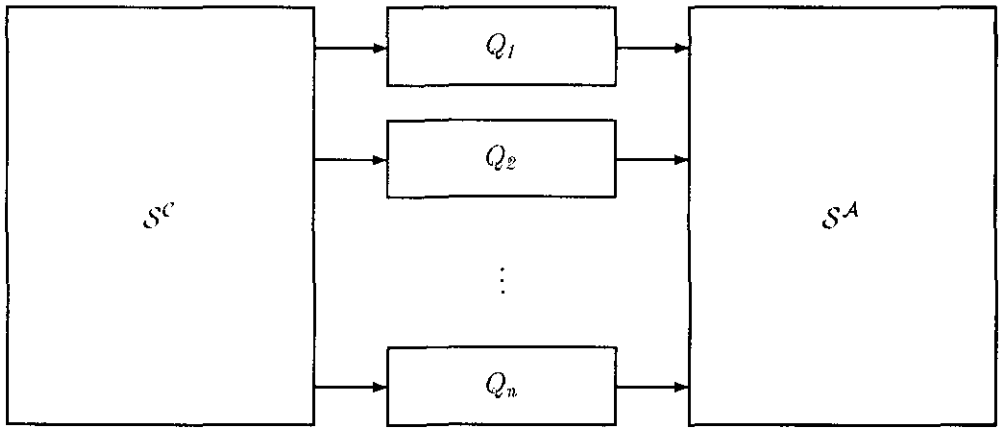


Figure 7.6: Structure of partial order transducer for sequential consistency

abstract versions of the system, $\mathcal{S}^{\mathcal{C}}$ and $\mathcal{S}^{\mathcal{A}}$. For each i, there is a FIFO queue from $\mathcal{S}^{\mathcal{C}}$ to $\mathcal{S}^{\mathcal{A}}$ that carries the observable events in $\mathcal{O}_i$. The transducer may also contain additional auxiliary variables. Intuitively, the transducer may be understood as follows.

The concrete system $\mathcal{S}^{\mathcal{C}}$ generates observable events and inserts each event into the appropriate queue Q_i. Via the queues, the observable events are transferred to the abstract system $\mathcal{S}^{\mathcal{A}}$. It is now the

task of the transducer to remove the observable events from the queues, in an order that corresponds to a behavior of $\mathcal{S}^{\mathcal{A}}$. The FIFO nature of the queues ensure that events in a single $\mathcal{O}_i$ are performed in the same order by $\mathcal{S}^{\mathcal{C}}$ and $\mathcal{S}^{\mathcal{A}}$. However, events in different sets of observable events may be performed in different orders by $\mathcal{S}^{\mathcal{C}}$ and $\mathcal{S}^{\mathcal{A}}$. The major difficulty in constructing the transducer is to construct an appropriate "scheduling mechanism" which ensures that $\mathcal{S}^{\mathcal{A}}$ removes elements from queues in an appropriate order. Often, this scheduling is not trivial, but must be accomplished via auxiliary variables.

A proof-by-transduction of sequential consistency can be compared to other proof methods for sequential consistency. The proof method used in [ABM93] is directly based on the previously presented definition of sequential consistency. That proof shows how to inductively build a scenario of $\mathcal{S}^{\mathcal{A}}$, given a scenario of $\mathcal{S}^{\mathcal{C}}$, by direct reasoning about computations and behaviors. Typically, this proof will build $\gamma^{\mathcal{A}}$ inductively, based on $\gamma^{\mathcal{C}}$, with each prefix of $\gamma^{\mathcal{A}}$ being based on some prefix of $\gamma^{\mathcal{C}}$. The transduction method achieves a similar effect, but $\gamma^{\mathcal{A}}$ is part of the behavior of a specified system (the transducer), for which we can use standard methods for proving invariant and progress properties. Put differently, the construction of the transducer contains the part of the proof of sequential consistency that requires more human ingenuity (e.g., ensuring proper scheduling of abstract actions), while checking the properties in Theorem 7.7.3 should be more routine.

7.9 The Cache Memory Example

We shall use the generalization of the proof by transduction method to prove the Lazy Cache Algorithm. We refer to [Ger95] for a description of the algorithm. We first present a formal specification of an idealized serial memory, thereafter a specification of the cache protocol. We then construct a transducer to prove that the cache protocol is sequentially consistent (with the serial memory).

Serial Memory

The abstract system is a serial memory which can be presented as the following fair named transition system. We assume that $addr$ is a given set of addresses of a memory system, and that $data$ is a set of values that can be stored in the locations given by $addr$.

$$\begin{array}{lcl} V^{\mathcal{A}} & = & \{Mem^{\mathcal{A}} : \text{Array}[addr] \text{ of } data\} \\ \Theta^{\mathcal{A}} & = & \text{T} \\ \mathcal{T}^{\mathcal{A}} & = & \{Read_i^{\mathcal{A}}(d,a), Write_i^{\mathcal{A}}(d,a) \mid 1 \leq i \leq n, d \in data, a \in addr\} \\ \mathcal{J}^{\mathcal{A}} & = & \emptyset \\ \mathcal{O}^{\mathcal{A}} & = & \{Read_i(d,a), Write_i(d,a) \mid 1 \leq i \leq n, d \in data, a \in addr\} \end{array}$$

Note that $\mathcal{J}^{\mathcal{A}}$ is different from the version presented in [Ger95].

Let d be of type $data$ and a of type $addr$. The notation $(X[a] := d)$ where X is an Array is an abbreviation for $X'[a] = d \ \wedge \ \forall\, b \neq a : X'[b] = X[b]$. Then the transitions are given by:

$$\begin{array}{lll} Read_i^{\mathcal{A}}(d,a)\text{: } Mem^{\mathcal{A}}[a] = d & \text{gen.} & Read_i(d,a) \\ Write_i^{\mathcal{A}}(d,a)\text{: } (Mem^{\mathcal{A}}[a] := d) & \text{gen.} & Write_i(d,a) \end{array}$$

The transition $Read_i^{\mathcal{A}}(d,a)$ can be performed if location a contains data value d and does not change the state of the system. The transition $Write_i^{\mathcal{A}}(d,a)$ can be performed in any state, and changes the content of location a to become d.

Cache Memory

The concrete system is a cache memory which we define by a fair named transition system:

$$
\begin{array}{lcl}
V^{\mathcal{C}} & = & \{Mem^{\mathcal{C}} : \text{Array}[addr] \text{ of } data, \\
 & & \quad In_i : ((datum \times addr) \cup (datum \times addr \times \{*\}))^*, \\
 & & \quad Out_i : (datum \times addr)^*, \\
 & & \quad C_i : \text{Array}[addr] \text{ of } (datum \cup \bot) \mid 1 \le i \le n\} \\
\Theta^{\mathcal{C}} & = & In_i = Out_i = \Lambda \wedge C_i = Mem^{\mathcal{C}} \\
\mathcal{T}^{\mathcal{C}} & = & \{Read_i^{\mathcal{C}}(d,a), Write_i^{\mathcal{C}}(d,a), MemWrite_i^{\mathcal{C}}(d,a), MemRead_i^{\mathcal{C}}(d,a), \\
 & & \quad CacheUpdate_i^{\mathcal{C}}(d,a), CacheInvalidate_i^{\mathcal{C}} \mid 1 \le i \le n, d \in data, a \in addr\} \\
\mathcal{J}^{\mathcal{C}} & = & \{MemWrite_i^{\mathcal{C}}(d,a), CacheUpdate_i^{\mathcal{C}}(d,a) \\
 & & \quad \mid 1 \le i \le n, d \in data, a \in addr\} \\
\mathcal{O}^{\mathcal{C}} & = & \{Read_i(d,a), Write_i(d,a) \mid 1 \le i \le n, d \in data, a \in addr\}
\end{array}
$$

Intuitively, $Mem^{\mathcal{C}}$ is the global memory, and C_i is the local cache of process i. For each i, the variable In_i is a sequence of elements of the form (d,a) or of the form $(d,a,*)$, and the variable Out_i is a sequence of pairs of the form (d,a). In the cache memory, these sequences are used as FIFO queues. Initially the queues are empty, and all caches are identical to the global memory.

For a sequence In of (d,a) and $(d,a,*)$ elements, let $In\lceil_*$ denote the projection of In onto $(d,a,*)$ elements. Then the transitions are given by:

$$
\begin{array}{lll}
Read_i^{\mathcal{C}}(d,a) & : C_i[a] = d \wedge Out_i = \Lambda \wedge In_i\lceil_* = \Lambda & \text{gen. } Read_i(d,a) \\
Write_i^{\mathcal{C}}(d,a) & : Out_i' = Out_i \bullet (d,a) & \text{gen. } Write_i(d,a) \\
MemWrite_i^{\mathcal{C}}(d,a) & : Out_i = (d,a) \bullet Out_i' \wedge & \text{gen. } MemWrite_i(d,a) \\
 & \quad (Mem^{\mathcal{C}}[a] := d) \wedge & \\
 & \quad In_i' = In_i \bullet (d,a,*) \wedge & \\
 & \quad \forall k \neq i : In_k' = In_k \bullet (d,a) & \\
MemRead_i^{\mathcal{C}}(d,a) & : Mem^{\mathcal{C}}[a] = d \wedge In_i' = In_i \bullet (d,a) & \text{gen. } MemRead_i(d,a) \\
CacheUpdate_i^{\mathcal{C}}(d,a) & : \begin{pmatrix} In_i = (d,a) \bullet In_i' \\ \vee \\ In_i = (d,a,*) \bullet In_i' \end{pmatrix} & \text{gen. } CacheUpdate_i(d,a) \\
 & \quad \wedge \ (C_i[a] := d) & \\
CacheInvalidate_i^{\mathcal{C}} & : C_i' = Restrict(C_i) & \text{gen. } CacheInvalidate_i
\end{array}
$$

Note that $\mathcal{J}^{\mathcal{C}}$ is different from the version presented in [Ger95].

Intuitively, a transition of the form $Write_i^{\mathcal{C}}(d,a)$ is a write operation, which simply appends the pair (d,a) to the end of the Out_i queue. A transition of the form $MemWrite_i^{\mathcal{C}}(d,a)$ applies the first write operation in Out_i to the global memory $Mem^{\mathcal{C}}$, and appends the operation (d,a) to all queues In_k. The element appended to In_i is equipped with an extra $*$ in order to mark that this operation originated from process i. This mark is used in read operations by process i. A transition of the form $CacheUpdate_i^{\mathcal{C}}(d,a)$ applies the first write operation in In_i to the local cache.

A transition of the form $Read_i^{\mathcal{C}}(d,a)$ is a read operation which can be performed if the data element in address a of the local cache is d, and two additional conditions are satisfied: (1) the queue Out_i must be empty, and (2) the queue In_i must not contain any elements of the form $(d_1,a_1,*)$, i.e., elements that originate from write operations by process i. These two requirements ensure that a read operation will read from a cache to which all past write operations by the same process have been applied.

$Restrict$ is any function such that $C_i' = Restrict(C_i)$ means that

$$\forall a \in addr : C_i'[a] = C_i[a] \vee C_i'[a] = \bot$$

Thus a $CacheInvalidate_i^{\mathcal{C}}$ transition restricts the domain of the cache C_i.

A transition of the form $MemRead_i^{\mathcal{C}}(d, a)$ is a memory read operation which can be performed if the value of address a in $Mem^{\mathcal{C}}$ is d. It appends a (d, a) element to the In_i queue.

Cache Transducer

We show sequential consistency for the lazy cache algorithm by proving that the FNTS $\mathcal{S}^{\mathcal{C}}$ for the cache memory is a partial order refinement of the FNTS $\mathcal{S}^{\mathcal{A}}$ for the serial memory with respect to the dependency relation D_{sc}. Two events are dependent iff they are observable and are performed by the same process, i.e.

$$\begin{aligned} D_{sc} \quad = \quad & \{(Write_i(d_1, a_1), Write_i(d_2, a_2)), (Read_i(d_1, a_1), Read_i(d_2, a_2)), \\ & (Read_i(d_1, a_1), Write_i(d_2, a_2)), (Write_i(d_1, a_1), Read_i(d_2, a_2)) \\ & \mid 1 \leq i \leq n, d_1, d_2 \in data, a_1, a_2 \in addr\} \end{aligned}$$

As described in Theorem 7.8.1, the problem of proving that $\mathcal{S}^{\mathcal{C}}$ is sequentially consistent with $\mathcal{S}^{\mathcal{A}}$ can be formulated as proving that

$$\mathcal{S}^{\mathcal{C}} \sqsubseteq_{\simeq_{D_{sc}}} \mathcal{S}^{\mathcal{A}}$$

We prove this refinement by transduction.

From $\mathcal{S}^{\mathcal{C}}$ and $\mathcal{S}^{\mathcal{A}}$, we form a transducer by adding the variable Q which is a partially ordered multi-set of events. In the transducer, we are only allowed to use the operator $\bullet_D$, defined above, when performing operations on Q, so the pomset Q orders two events if and only if they are dependent. Since the dependency relation D is an equivalence relation with one equivalence class for each index i, the pomset Q defines a total ordering of the events indexed by i. One can therefore think of Q as consisting of one queue of $Read_i$ and $Write_i$ events for each process i. The queues are independent of each other (see Figure 7.6).

If we form the most straight-forward transducer with the given Q, we will not be able to prove the progress in matching condition needed for the correctness of the refinement. The reason is that if the abstract transition system consumes events in the wrong order, then it may reach a state where events of one of the processes can no longer be processed. As an example, consider a scenario where at some point the address a of $Mem^{\mathcal{A}}$ and each of the caches contains the data element d. Assume now that some caches update the data element in address a to $\overline{d}$ by consuming elements from the corresponding In queues, and that thereafter a $\widehat{Write_i}^{\mathcal{A}}(\overline{d}, a)$ event is performed which changes the address a to $\overline{d}$ also in $Mem^{\mathcal{A}}$. If some cache C_j still has value d at address a, and a $\widehat{Read_j^{\mathcal{C}}}(d, a)$ operation occurs, then the event $Read_j(d, a)$ will be added to Q. However, this event can most probably never be removed from Q in an abstract $\widehat{Read_j^{\mathcal{A}}}(d, a)$ transition, since the content of a in $Mem^{\mathcal{A}}$ has already been changed from d to $\overline{d}$.

The problem with the above scenario is that a $\widehat{Write_i}^{\mathcal{A}}(\overline{d}, a)$ operation is performed before the corresponding write updates are applied to all caches. Observing that all caches experience the updates in the same order, determined by the global order of $\widehat{MemWrite_i}^{\mathcal{C}}$ operations, we can remedy this by adding information to the transducer, in the form of auxiliary variables, which restrict the possible sequences of abstract transitions in the following way. The auxiliary variables must ensure that

1. For each process i, the $\widehat{Read_i}^{\mathcal{A}}$ operations and all $\widehat{Write_j^{\mathcal{A}}}$ operations (for all j) must be performed in the same order as the cache C_i experiences the corresponding concrete operations,

where $\widehat{Read_i}^{\mathcal{A}}(d,a)$ corresponds to $\widehat{Read_i}^{\mathcal{C}}(d,a)$ and where $\widehat{Write_j}^{\mathcal{A}}(d,a)$ corresponds to performing a $\widehat{CacheUpdate_j}^{\mathcal{C}}(d,a)$ with the element in In_j that originates from the corresponding $\widehat{Write_j}^{\mathcal{C}}(d,a)$ operation.

2. A $\widehat{Write_i}^{\mathcal{A}}(d,a)$ operation must not be performed before all the corresponding $\widehat{CacheUpdate_j}^{\mathcal{C}}$ (d,a) operations have applied that write operation to each cache.

Due to these conditions, we must "remember" for each process the order of all read and all write operations that have been applied to its cache at least as long as there is a process which has not applied these write events to its cache. Therefore, we introduce for each i a variable RWQ_i, which is a sequence of $Write_i$ events, $Write$ events (modelling writes performed by other processes), and $Read_i$ events. Intuitively, RWQ_i contains all write operations that have been applied to the cache of process i, but have not yet been consumed by the abstract serial memory, together with read events of process i, so that the order of operations in RWQ_i is the same as the order in which the corresponding operations were performed on C_i.

Formally, for the elements in the RWQ_i queues, $Write_\epsilon(d,a)$ indicates that the element (d,a) was written by another process j, $j \neq i$, and $Write_i(d,a)$ indicates that the element (d,a) was written by the process i itself. We use ι to range over $\{1,\ldots,n,\epsilon\}$. Thus the names of the events in the RWQ_i queues range over $E^{RWQ} = E^{\mathcal{C}} \cup \{Write_\epsilon(d,a) \mid d \in data, a \in addr\}$. We use $Write(d,a)$, for any d, a, as a shorthand notation for $Write_\epsilon(d,a)$.

A $Write_\epsilon(d,a)$ or $Write_i(d,a)$ entry is appended to RWQ_i whenever a transition of the form $\widehat{CacheUpdate_i}^{\mathcal{C}}(d,a)$ is taken with a (d,a) or a $(d,a,*)$ entry at the front of In_i if the (d,a) or $(d,a,*)$ entry was placed there by a $\widehat{MemWrite_j}^{\mathcal{C}}$ or a $\widehat{MemWrite_i}^{\mathcal{C}}$ operation. Since transition $\widehat{MemRead_i}^{\mathcal{C}}$ can also add (d,a) entries to In_i, it is necessary to distinguish between (d,a) entries placed in In_i by a $\widehat{MemWrite_j}^{\mathcal{C}}$ operation and those placed there by a $\widehat{MemRead_i}^{\mathcal{C}}$ operation.

To keep this distinction, we use a number of parallel $MrMwQ_i$ queues, one for each In_i queue. Then every time a (d,a) or $(d,a,*)$ element is added to In_i by a $MemWrite$ operation, it is marked with an Mw in the $MrMwQ_i$ queue, whereas every time an (d,a) element is added to In_i by an $MemRead$ operation, it is marked with an Mr in the $MrMwQ_i$ queue.

The definition of the cache transducer is shown in Figure 7.7. We omit specification of the events generated by the transducer transitions, since they are determined by Definition 7.7.2.
If $Read_i(d,a)$ is the first event in RWQ_i and is minimal in Q, and d is the content of address a in $Mem^{\mathcal{A}}$, then transition $\widehat{Read_i}^{\mathcal{A}}(d,a)$ can be taken, consuming the event $Read_i(d,a)$ from Q and the head of RWQ_i. Analogously, if $Write_i(d,a)$ is minimal in Q and is at the first event in RWQ_i and $Write(d,a)$ is the first event in RWQ_j for all $j \neq i$, then transition $\widehat{Write_i}^{\mathcal{A}}(d,a)$ can be taken, consuming event $Write_i(d,a)$ from RWQ_i and Q, consuming event $Write(d,a)$ from all the RWQ_j queues, and performing a $Write_i^{\mathcal{A}}(d,a)$ operation.

Transition $\widehat{Read_i}^{\mathcal{C}}(d,a)$ performs the operation $Read_i^{\mathcal{C}}(d,a)$ and then appends the event $Read_i(d,a)$ to Q and to RWQ_i. Transition $\widehat{Write_i}^{\mathcal{C}}(d,a)$ performs the operation $Write_i^{\mathcal{C}}(d,a)$ and appends the event $Write_i(d,a)$ to Q. The operation $\widehat{MemWrite_i}^{\mathcal{C}}(d,a)$ performs the operation $MemWrite_i^{\mathcal{C}}(d,a)$ and appends an Mw element to all the $MrMwQ_j$ queues (for all j). Transition $\widehat{CacheUpdate_i}^{\mathcal{C}}(d,a)$ performs the $CacheUpdate_i^{\mathcal{C}}(d,a)$ operation. Then if the head of $MrMwQ_i$ is Mw (note that this is always the case if $(d,a,*)$ is at the head of In_i) it adds a $Write_i(d,a)$ or $Write(d,a)$ event to RWQ_i, depending on whether the element at the head of In_i originates form

$$
\begin{aligned}
V^T &= V^{\mathcal{A}} \cup V^{\mathcal{C}} \cup \\
&\quad \{Q : Pomset(\mathcal{O}^{\mathcal{C}}), RWQ_i : Events(E^{RWQ})^*, MrMwQ_i : (Mr \cup Mw)^*\} \\
\Theta^T &= \Theta^{\mathcal{C}} \wedge Q = \Lambda \\
\mathcal{T}^T &= \{\widehat{Read_i}^{\mathcal{A}}(d,a), \widehat{Write_i}^{\mathcal{A}}(d,a), \widehat{Read_i}^{\mathcal{C}}(d,a), \widehat{Write_i}^{\mathcal{C}}(d,a), \\
&\quad \widehat{MemWrite_i}^{\mathcal{C}}(d,a), \widehat{MemRead_i}^{\mathcal{C}}(d,a), \widehat{CacheUpdate_i}^{\mathcal{C}}(d,a), \\
&\quad \widehat{CacheInvalidate_i}^{\mathcal{C}} \mid 1 \le i \le n, d \in data, a \in addr\} \\
\mathcal{J}^T &= \{\widehat{MemWrite_i}^{\mathcal{C}}(d,a), \widehat{CacheUpdate_i}^{\mathcal{C}}(d,a), \\
&\quad \widehat{Read_i}^{\mathcal{A}}(d,a), \widehat{Write_i}^{\mathcal{A}}(d,a) \mid 1 \le i \le n, d \in data, a \in addr\} \\
\mathcal{O}^T &= \{Read_i(d,a), Write_i(d,a) \mid 1 \le i \le n, d \in data, a \in addr\}
\end{aligned}
$$

$$
\begin{aligned}
\widehat{Read_i}^{\mathcal{A}}(d,a)&: Mem^{\mathcal{A}}[a] = d \wedge \\
&\qquad Q = Read_i(d,a) \bullet_D Q' \wedge \\
&\qquad RWQ_i = Read_i(d,a) \bullet RWQ_i' \\
\widehat{Write_i}^{\mathcal{A}}(d,a)&: Q = Write_i(d,a) \bullet_D Q' \wedge \\
&\qquad RWQ_i = Write_i(d,a) \bullet RWQ_i' \wedge \\
&\qquad \forall j \neq i : RWQ_j = Write(d,a) \bullet RWQ_j' \wedge \\
&\qquad (Mem^{\mathcal{A}}[a] := d)
\end{aligned}
$$

$$
\begin{aligned}
&\widehat{Read_i}^{\mathcal{C}}(d,a) && : C_i[a] = d \wedge Out_i = \Lambda \wedge In_i\lceil_* = \Lambda \wedge \\
&&& \quad Q' = Q \bullet_D Read_i(d,a) \wedge \\
&&& \quad RWQ_i' = RWQ_i \bullet Read_i(d,a) \\
&\widehat{Write_i}^{\mathcal{C}}(d,a) && : Out_i' = Out_i \bullet (d,a) \wedge Q' = Q \bullet_D Write_i(d,a) \\
&\widehat{MemWrite_i}^{\mathcal{C}}(d,a) && : Out_i = (d,a) \bullet Out_i' \wedge \\
&&& \quad (Mem^{\mathcal{C}}[a] := d) \wedge \\
&&& \quad In_i' = In_i \bullet (d,a,*) \wedge \\
&&& \quad \forall k \neq i : In_k' = In_k \bullet (d,a) \wedge \\
&&& \quad \forall k : MrMwQ_k' = MrMwQ_k \bullet Mw \\
&\widehat{MemRead_i}^{\mathcal{C}}(d,a) && : Mem^{\mathcal{C}}[a] = d \wedge In_i' = In_i \bullet (d,a) \wedge MrMwQ_i' = MrMwQ_i \bullet Mr \\
&\widehat{CacheUpdate_i}^{\mathcal{C}}(d,a) && : \left(\begin{array}{l} In_i = (d,a) \bullet In_i' \wedge (head(MrMwQ_i) = Mw \rightarrow \\ \qquad\qquad RWQ_i' = RWQ_i \bullet Write(d,a)) \\ \vee \\ In_i = (d,a,*) \bullet In_i' \wedge RWQ_i' = RWQ_i \bullet Write_i(d,a) \end{array} \right) \wedge \\
&&& \quad (C_i[a] := d) \wedge MrMwQ_i' = tail(MrMwQ_i) \\
&\widehat{CacheInvalidate_i}^{\mathcal{C}} && : C_i' = Restrict(C_i)
\end{aligned}
$$

Figure 7.7: The cache transducer.

process i or some other process. Finally, the element at the head of $MrMwQ_i$ is consumed. Transition $\widehat{MemRead_i}^{\mathcal{C}}(d,a)$ performs the operation $MemRead_i^{\mathcal{C}}(d,a)$ and appends an Mr element to the $MrMwQ_i$ queue. Transition $\widehat{CacheInvalidate_i}^{\mathcal{C}}$ just performs the operation $CacheInvalidate_i^{\mathcal{C}}$.

We now proceed to prove the properties of the cache transducer that will allow us to use Theorem 7.7.3 to conclude that the cache memory is sequentially consistent with the serial memory. Since both $\mathcal{J}^{\mathcal{A}} = \emptyset$ and $\mathcal{C}^{\mathcal{A}} = \emptyset$, we only have to verify the matching progress property for the cache transducer.

7.10 Proof of the Cache Transducer

Before we list the invariants for the cache transducer, we introduce some useful notation.

For a sequence of events X, we define $X\restriction_{Write}$ to be the projection of X onto the set of events of the forms $Write(d,a)$ or $Write_i(d,a)$. This is used to project RWQ_i on all $Write$ and $Write_i$ events. We define projection of a sequence of events X onto a process number i, written $X\restriction_i$, to be the projection of X onto events of process i. Likewise, for a pomset Q, $Q\restriction_i$ is the projection of Q onto the events of process i.

The predicate OK takes a memory Mem and a sequence of $Write$ events and $Read_i$-events, RWQ, and checks that for each $Read_i(d,a)$-event if the memory is updated with the preceding $Write$-events, then $Mem[a] = d$:

$$\begin{array}{lcl} OK(Mem,\Lambda) & = & \mathrm{T} \\ OK(Mem, Write_i(d,a) \bullet RWQ) & = & OK(Update(Mem,d,a), RWQ) \\ OK(Mem, Write(d,a) \bullet RWQ) & = & OK(Update(Mem,d,a), RWQ) \\ OK(Mem, Read_i(d,a) \bullet RWQ) & = & Mem[a] = d \wedge OK(Mem, RWQ) \end{array}$$

where the $Update(Mem,d,a)$ denotes a memory which is like Mem except at address a where the value is d.

The function $Apply$ takes a memory Mem and a sequence of $Write$-events and $Read_j$-events, RWQ, and results in a memory which is updated according to the sequence of $Write$-events in RWQ:

$$\begin{array}{lcl} Apply(Mem,\Lambda) & = & Mem \\ Apply(Mem, Write_i(d,a) \bullet RWQ) & = & Apply(Update(Mem,d,a), RWQ) \\ Apply(Mem, Write(d,a) \bullet RWQ) & = & Apply(Update(Mem,d,a), RWQ) \\ Apply(Mem, Read_i(d,a) \bullet RWQ) & = & Apply(Mem, RWQ) \end{array}$$

In the following, we use the two basic properties of OK and $Apply$ that are expressed in Lemmas 7.10.1 and 7.10.2:

Lemma 7.10.1 $OK(Mem, RWQ_1 \bullet RWQ_2) =$
$OK(Mem, RWQ_1) \wedge OK(Apply(Mem, RWQ_1), RWQ_2)$

Checking that some RWQ is OK with respect to some memory Mem, can be split into checking that the first part of RWQ is OK with respect Mem and then checking that the second part is OK with respect to the memory obtained by applying the first part to Mem. The two parts of RWQ can be chosen arbitrarily.

Lemma 7.10.2 $Apply(Mem, RWQ_1 \bullet RWQ_2) = Apply(Apply(Mem, RWQ_1), RWQ_2)$

Applying some RWQ to some memory Mem can be split into applying the first part of RWQ to Mem and then applying the second part of RWQ to the memory obtained by the first application. The two parts of RWQ can be chosen arbitrarily.

We refer to [JPR94] for the proofs of the lemmas.

The function $Wout(i, X)$ simply converts a sequence X of (d, a)-elements to a sequence of $Write_i(d, a)$-elements. It is used to convert an Out_i sequence $(d_1, a_1), \ldots, (d_m, a_m)$ into the event sequence $Write_i(d_1, a_1), \ldots, Write_i(d_m, a_m)$.

$$\begin{array}{lcl} Wout(i, \Lambda) & = & \Lambda \\ Wout(i, (d, a) \bullet X) & = & Write_i(d, a) \bullet Wout(i, X) \end{array}$$

The function $Win(i, X, MrMwQ)$ converts a sequence X of (d, a) and $(d, a, *)$ elements to a sequence of $Write(d, a)$ and $Write_i(d, a)$ elements, while ignoring elements for which the corresponding element in a sequence $MrMwQ$ of Mw's and Mr's is Mr. The definition assumes X and $MrMwQ$ to be of equal lengths. It is always applied to In_i queues and a corresponding $MrMwQ$ sequence.

$$\begin{array}{lcl} Win(i, \Lambda, \Lambda) & = & \Lambda \\ Win(i, (d, a) \bullet X, Mr \bullet MrMwQ) & = & Win(i, X, MrMwQ) \\ Win(i, (d, a) \bullet X, Mw \bullet MrMwQ) & = & Write(d, a) \bullet Win(i, X, MrMwQ) \\ Win(i, (d, a, *) \bullet X, MrMwQ) & = & Write_i(d, a) \bullet Win(i, X, tail(MrMwQ)) \end{array}$$

We introduce the following shorthand notations:

$$\begin{array}{lcl} Win_i & = & Win(i, In_i, MrMwQ_i) \\ Wout_i & = & Wout(i, Out_i) \\ WriteQ_i & = & RWQ_i\restriction_{Write} \bullet Win_i \end{array}$$

Thus, Win_i denotes the sequence of $Write$ events whose parameters are currently contained in In_i, while $Wout_i$ denotes a similar sequence for the buffer Out_i. Sequence Win_i may contain both $Write_i$ entries for writes initiated by process i and $Write$ entries for writes initiated by other processes. Sequence $Wout_i$ contains only $Write_i$ entries. The sequence $WriteQ_i$ contains all the $Write$-events that process i has observed and that have not yet been performed by the abstract part of the transducer.

For a sequence X and an element e, we define $mindex(e, X)$ to be the smallest index $i \in dom(X)$ such that $X[i] = e$. If $e \notin X$, then $mindex(e, X)$ is taken to be 1.

7.10.1 Invariants for the Cache Transducer

We need four invariants. The proofs are given in [JPR94].

$$\begin{array}{ll} I_1: & WriteQ_i[m] = Write_\iota(d, a) \quad \rightarrow \\ & \quad \exists j : WriteQ_j[m] = Write_j(d, a) \wedge \forall k \neq j : WriteQ_k[m] = Write(d, a) \\ I_2: & Q\uparrow_i = RWQ_i\restriction_i \bullet Win_i\restriction_i \bullet Wout_i \\ I_3: & OK(Mem^{\mathcal{A}}, RWQ_i) \\ I_4: & C_i[a] = d \quad \rightarrow \quad Apply(Mem^{\mathcal{A}}, RWQ_i)[a] = d \end{array}$$

I_1 states that all processes experience the same order of $Write$ events and for each particular $Write$ event one and only one process marks this event as its own.

For each i, I_2 states that the sequence RWQ_i of process i will (eventually) mirror the experienced order of $Read_i$ and $Write_i$ events of the process itself in the concrete system, as given by $Q\restriction_i$. This invariant also implies that $Q\restriction_i$ is totally ordered.

For each i, I_3 states that the sequence RWQ_i contains a sequence of *Read* and *Write* events that can be applied to the current version of $Mem^{\mathcal{A}}$. That is, if we update the current version of $Mem^{\mathcal{A}}$ according to the sequence of $Write$ operations then, whenever we encounter a $Read_i(d, a)$ event, the updated memory at this point is such that $Mem^{\mathcal{A}}[a] = d$.

Since the transducer may perform $\widehat{CacheInvalidate_i}^{\mathcal{C}}$ operations, the cache C_i will not necessarily have a value at address a. The invariant I_4 states, for each i, that if the current version of C_i has a value d for the address a then if we apply the sequence of $Write$ operations contained in RWQ_i to the current version of $Mem^{\mathcal{A}}$, it will yield the same value d at address a.

7.10.2 Proof of Progress in Matching

We proceed to prove the matching-progress requirement for the cache transducer, using the four listed invariants. The invariants themselves will be proven in a subsequent subsection.

We shall prove that all events contained in the interface pomset are eventually consumed by abstract transitions. In the case of sequential consistency, it is sufficient to prove for each i, that the first element of the linear sequence Q_i that contains $\mathcal{O}_i$ is eventually removed whenever Q_i is nonempty. In the case that the first element of Q_i is a read event, we will show that in fact a $\widehat{Read_i}^{\mathcal{A}}$-event is enabled, and will eventually be performed (by justice). In the case that the first element of Q_i is a write event, the corresponding $\widehat{Write_i}^{\mathcal{A}}$-event need not be enabled, since this requires the corresponding $Write$-event to be first in each RWQ_j. However, we will show that this situation will eventually occur, when all preceding read events have been consumed, and that then (by justice), the $\widehat{Write_i}^{\mathcal{A}}$-event will be performed. After this sketchy outline, over to the proof.

The following lemma proves that any enabled observable abstract transition is eventually taken.

Lemma 7.10.3 *For ν_i in $\{Read_i, Write_i\}$,*

$$En(\widehat{\nu_i^{\mathcal{A}}}(d,a)) \quad \Rightarrow \quad \Diamond\, taken(\widehat{\nu_i^{\mathcal{A}}}(d,a))$$

Proof: Both cases are proven by rule J-TAKE, taking $p = \varphi : En(\widehat{\nu_i^{\mathcal{A}}}(d,a))$ for the assertions and $\tau_h : \widehat{\nu_i^{\mathcal{A}}}(d,a)$ for the helpful transition. The only nontrivial premise to be proven is J2 which claims that no transition other than $\widehat{\nu_i^{\mathcal{A}}}(d,a)$ itself can disable $\widehat{\nu_i^{\mathcal{A}}}(d,a)$, once it is enabled. We consider separately the two cases:

$\nu_i = Read_i$: For this case,

$$\begin{aligned} En(\widehat{Read_i}^{\mathcal{A}}(d,a)) \quad &= \quad Mem^{\mathcal{A}}[a] = d \ \wedge \\ &\qquad head(Q\restriction_i) = head(RWQ_i) = Read_i(d,a) \end{aligned}$$

Let us ascertain that no transition other than $\widehat{Read_i}^{\mathcal{A}}(d,a)$ can falsify this assertion, once it is true. The only transition that can falsify $Mem^{\mathcal{A}}[a] = d$ is some $\widehat{Write_j}^{\mathcal{A}}(e,a)$ for some $e \neq d$. No $Write_j^{\mathcal{A}}$ transitions are enabled when $head(RWQ_i) = Read_i(d,a)$. The only transition that can falsify $head(Q\restriction_i) = head(RWQ_i) = Read_i(d,a)$ is $\widehat{Read_i}^{\mathcal{A}}$ itself.

$\nu_i = Write_i$: For this case,

$$En(\widehat{Write_i}^{\mathcal{A}}(d,a)) \quad = \quad head(Q\restriction_i) = head(RWQ_i) = Write_i(d,a) \wedge \forall j \neq i : head(RWQ_j) = Write(d,a)$$

We show that no transition other than $\widehat{Write_i}^{\mathcal{A}}(d,a)$ can falsify this assertion, once it is true. The only transition that can falsify $head(Q\restriction_i) = head(RWQ_i) = Write_i(d,a)$ is $\widehat{Write_i}^{\mathcal{A}}(d,a)$ itself. Similarly, the only transitions that can falsify $head(RWQ_j) = Write(d,a)$ for $j \neq i$ are of the form $\widehat{Write_k}^{\mathcal{A}}(d,a)$ for some $k \neq j$. We will show that such a transition can be enabled only if $k = i$. For $\widehat{Write_k}^{\mathcal{A}}(d,a)$ to be enabled, it is necessary that $head(RWQ_k) = Write_k(d,a)$. However, $En(\widehat{Write_i}^{\mathcal{A}}(d,a))$ implies that, if $k \neq i$, then $head(RWQ_k) = Write(d,a) \neq Write_k(d,a)$. Thus, $k = i$.

□

The preceding lemma showed that if a transition of the form $\widehat{\nu_i}^{\mathcal{A}}(d,a)$ is enabled, where $\nu_i \in \{Read_i, Write_i\}$, it will eventually be taken. We proceed to show that if the corresponding event of such a transition is at the head of its respective RWQ_i queue, then the transition eventually becomes enabled.

For the case that $\nu_i = Read_i$, the following lemma establishes that if $Read_i(d,a)$ is at the head of RWQ_i then transition $\widehat{Read_i}^{\mathcal{A}}(d,a)$ is already enabled:

Lemma 7.10.4 $head(RWQ_i) = Read_i(d,a) \quad \Rightarrow \quad En(\widehat{Read_i}^{\mathcal{A}}(d,a))$

Proof: The enabling condition of $\widehat{Read_i}^{\mathcal{A}}(d,a)$ is

$$En(\widehat{Read_i}^{\mathcal{A}}(d,a)) \quad = \quad Mem^{\mathcal{A}}[a] = d \wedge head(Q\restriction_i) = head(RWQ_i) = Read_i(d,a)$$

The conjunct $head(RWQ_i) = Read_i(d,a)$ is given. By invariant I_2,

$$head(Q\restriction_i) = head(RWQ_i) = Read_i(d,a)$$

Invariant $I_3 = OK(Mem^{\mathcal{A}}, RWQ_i)$ and the fact that $Read_i(d,a)$ is the first element of RWQ_i imply $Mem^{\mathcal{A}}[a] = d$. Hence we have that $head(RWQ_i) = Read_i(d,a)$ implies $En(\widehat{Read_i}^{\mathcal{A}}(d,a))$. □

Next, we consider the case that the event at the head of RWQ_i is $Write_i(d,a)$. The enabling condition for the corresponding $\widehat{Write_i}^{\mathcal{A}}(d,a)$ transition consists of the conjunction $head(Q\restriction_i) = head(RWQ_i) = Write_i(d,a)$, which is implied by $head(RWQ_i) = Write_i(d,a)$ and invariant I_2, but also of the conjunct $head(RWQ_j) = Write(d,a)$ for every $j \neq i$. We will show that if $Write_i(d,a)$ is at the head of RWQ_i then, eventually, $head(RWQ_j) = Write(d,a)$ for every $j \neq i$. This ensures that $\widehat{Write_i}^{\mathcal{A}}(d,a)$ eventually becomes enabled.

Let us consider some $j \neq i$. Invariant I_1 with $m = 1$ implies that $head(WriteQ_j) = Write(d,a)$. Since $WriteQ_j = RWQ_j\restriction_{Write} \bullet Win_j$, it follows that either $Write(d,a)$ is the first write event in RWQ_j, or $Write(d,a)$ is the first element of Win_j and RWQ_j contains only read operations. We deal first with the latter case.

The following lemma establishes that if $Write(d,a)$ is the first element of Win_j then eventually it will move to RWQ_j.

Lemma 7.10.5 $head(RWQ_i) = Write_i(d,a) \wedge head(Win_j) = Write(d,a) \Rightarrow \Diamond(head(RWQ_i) = Write_i(d,a) \wedge Write(d,a) \in RWQ_j)$

Proof: Note that the lemma requires that when $Write(d,a)$ moves to RWQ_j, then $Write_i(d,a)$ is still at the head of RWQ_i.

To prove the lemma, we use rule STEP with the following constructs:

$$\begin{array}{ll} p: & head(RWQ_i) = Write_i(d,a) \wedge head(Win_j) = Write(d,a) \\ \varphi: & head(RWQ_i) = Write_i(d,a) \wedge head(Win_j) = Write(d,a) \wedge \\ & Write(d,a) \notin RWQ_j \\ q: & head(RWQ_i) = Write_i(d,a) \wedge Write(d,a) \in RWQ_j \\ \tau_h: & \widehat{CacheUpdate}_j^{\mathcal{C}}(d,a) \end{array}$$

Only premise J2 is non-trivial. J2 claims that no transition can falsify φ without establishing q. It is easily seen that only $\widehat{CacheUpdate}_j^{\mathcal{C}}(d,a)$ can falsify φ. □

We now handle the case that $Write_i(d,a)$ is at the head of RWQ_i and $Write(d,a)$ is the first write event in RWQ_j. The entry $Write(d,a)$ may still not be the first in RWQ_j, in which case it is preceded by several $Read_j^{\mathcal{A}}$ entries. The following lemma establishes that such a state is always followed by another state in which $Write(d,a)$ is at the head of RWQ_j.

Lemma 7.10.6 $head(RWQ_i) = Write_i(d,a) \wedge Write(d,a) \in RWQ_j \Rightarrow \Diamond(head(RWQ_i) = Write_i(d,a) \wedge head(RWQ_j) = Write(d,a))$

Proof: We use rule S-WELL, choosing constructs as follows:

$$\begin{array}{ll} p: & head(RWQ_i) = Write_i(d,a) \wedge Write(d,a) \in RWQ_j \\ \varphi: & head(RWQ_i) = Write_i(d,a) \wedge mindex(Write(d,a), RWQ_j) > 1 \\ q: & head(RWQ_i) = Write_i(d,a) \wedge head(RWQ_j) = Write(d,a)) \\ h: & \widehat{Read}_j^{\mathcal{A}}(d_1,a_1) \text{ whenever } head(RWQ_j) = Read_j(d_1,a_1) \end{array}$$

Note that h returns an enabled, just transition in every φ-state since by I_1, in such a state, $Write(d,a)$ is the first write entry in RWQ_j and its index is at least 2. It follows that the first entry in RWQ_j is a read entry which, by Lemma 7.10.4, corresponds to an enabled transition. □

Lemmas 7.10.5 and 7.10.6 can be combined using rules TRNS and CASES to yield

$$head(RWQ_i) = Write_i(d,a) \Rightarrow \Diamond(head(RWQ_i) = Write_i(d,a) \wedge head(RWQ_j) = Write(d,a)), \tag{7.8}$$

for every $j \neq i$.

Using this, we can establish that once a $Write_i(d,a)$ event is at the head of RWQ_i then, eventually, the corresponding $\widehat{Write}_i(d,a)$ transition will be enabled:

Lemma 7.10.7 $head(RWQ_i) = Write_i(d,a) \Rightarrow \Diamond En(\widehat{Write}_i^{\mathcal{A}}(d,a))$

Proof: By induction on $j \neq i$ and using property 7.8, we can establish the response property

$$head(RWQ_i) = Write_i(d,a) \Rightarrow \Diamond(head(RWQ_i) = Write_i(d,a) \wedge \bigwedge_{j \neq i} head(RWQ_j) = Write(d,a)),$$

which implies that, eventually, $\widehat{Write_i}(d,a)$ becomes enabled. To do so, it is necessary to slightly modify property (7.8) to ensure that, once $Write(d,a)$ reaches the head of some RWQ_{j_1}, it remains there until the whole transition becomes enabled. The modified version of property (7.8) can be proven using TRNS and a proof similar to that of Lemma 7.10.6. □

Lemmas 7.10.4 and 7.10.7 establish that if a $Read_i(d,a)$ or a $Write_i(d,a)$ event is at the head of RWQ_i, then the associated transition eventually becomes enabled. Lemma 7.10.3 guarantees that this transition is eventually taken.

Now we are ready to prove the matching-progress property of the cache transducer:

$$\nu_i(d,a) \in Q \quad \Rightarrow \quad \Diamond\, taken(\widehat{\nu_i^{\mathcal{A}}}(d,a)) \qquad \text{for all } \nu_i(d,a) \in \mathcal{O}$$

Proof: According to I_2, if $\nu_i(d,a) \in Q$, then $\nu_i(d,a)$ is in one of the queues RWQ_i, Win_i, or $Wout_i$. Using arguments similar to the proof of Lemma 7.10.5, we can show that $\nu_i(d,a)$ must progress from $Wout_i$ to Win_i, and from Win_i to RWQ_i. It is therefore sufficient to treat the case that $\nu_i(d,a) \in RWQ_i$. Consider the minimal index of $\nu_i(d,a)$ in RWQ_i. If $mindex(\nu_i(d,a), RWQ_i) = 1$ then, as explained above, lemmas 7.10.4, 7.10.7, and 7.10.3 ensure that $\widehat{\nu_i^{\mathcal{A}}}(d,a)$ is eventually taken. For the cases that $mindex(\nu_i(d,a), RWQ_i) > 1$, we will establish the response property

$$mindex(\nu_i(d,a), RWQ_i) = k > 1 \qquad \Rightarrow \qquad \Diamond(mindex(\nu_i(d,a), RWQ_i) = k-1) \tag{7.9}$$

To prove property 7.9 we consider the entry which is currently at the head of RWQ_i. By the trivial invariant $\nu(d,a) \in RWQ_i \quad \rightarrow \quad \nu \in \{Read_i, Write_i, Write\}$, there are three cases to consider:

$head(RWQ_i) = Read_i(d_1,a_1)$: In this case, transition $\widehat{Read_i}^{\mathcal{A}}(d_1,a_1)$ is currently enabled and, by lemma 7.10.3, will be eventually taken, decreasing the minimal index of $\nu_i(d,a)$ in RWQ_i.

$head(RWQ_i) = Write_i(d_1,a_1)$: By Lemma 7.10.7, transition $\widehat{Write_i}^{\mathcal{A}}(d_1,a_1)$ will eventually become enabled and, by Lemma 7.10.3, eventually taken, decreasing the minimal index of $\nu_i(d,a)$ in RWQ_i.

$head(RWQ_i) = Write(d_1,a_1)$: By invariant I_1, there exists some $j \neq i$ such that $Write_j(d_1,a_1)$ is the first write entry in $WriteQ_j$. Since $Write_j(d_1,a_1)$ can only be preceded by read entries in the concatenation $RWQ_j \bullet Win_i$, we can trace its progress until it reaches the head of RWQ_j. Once there, it will eventually become enabled and the transition $\widehat{Write_j}^{\mathcal{A}}(d_1,a_1)$, eventually taken. Taking this transition removes the entry $Write(d_1,a_1)$ from RWQ_i and decrements by 1 the minimal index of $\nu_i(d,a)$ in RWQ_i.

We can now use property 7.9 to prove premise W2 in rule WELL and establish that every $\nu_i(d,a) \in RWQ_i$ eventually gets to the head of RWQ_i where it is guaranteed to be eventually removed by transition $\widehat{\nu_i^{\mathcal{A}}}(d,a)$. □

Since $\mathcal{J}^{\mathcal{A}} = \emptyset$ and $\mathcal{C}^{\mathcal{A}} = \emptyset$, we can finally use Theorem 7.7.3 to conclude that $\mathcal{S}^{\mathcal{C}} \sqsubseteq \mathcal{S}^{\mathcal{A}}$.

7.11 Conclusion and Related Work

We have presented a method for proving refinement between concurrent systems, called *proof by transduction*. The main idea of the method is to construct, for a given pair of a concrete and an abstract system, a *transducer* consisting of the concrete system, the abstract system, and a queue of

observable events. The concrete system inserts events into the queue, which are then removed by the abstract system. Refinement is established by proving that for any behavior of the concrete system, we can find a corresponding behavior of the abstract system, in which all events inserted into the queue are eventually removed.

The main advantage of the transduction method is that the transducer may defer nondeterministic choices in the abstract system until the point in time when the relevant nondeterministic choices have been performed in the concrete system. Without such a delay, there are many cases in which refinement cannot be established by ordinary (forward) simulation, but one must instead use backward simulation [Jon91] or prophecy variables [AL91]. Thus the transduction method can often reduce the number of prophecy variables needed in a proof of refinement. The proof method can also prove refinement in many cases where the backward simulation technique would fail because of the finite-image condition needed when considering infinite behaviors. In the buffer example of the paper, we could construct a backward simulation between the concrete and the abstract system, which however would not be finitary, especially when the domain D of data values is infinite. A proof of refinement along these lines could be constructed as a combination of several backward simulations, each ensuring infinitely many instances of finite image [Jon91], but this would be rather cumbersome.

One of the important features of the transduction method is its straightforward generalization to partial order refinement. In this way, we can reduce many instances of interface refinement, such as sequential consistency etc. to standard refinement, simply by an appropriate choice of interface queue.

A proof by transduction of sequential consistency can be compared to proof methods for sequential consistency that are based on direct reasoning about computations and behaviors (e.g., as in [ABM93]). Typically, such a proof will build an abstract behavior inductively, based on successively longer prefixes of a given concrete behavior. The transduction method makes the structure and bookkeeping involved in such a proof explicit, representing the unmatched portion of the concrete behavior as the value of the interface queue.

The transduction method can be seen as a generalization of proof by simulations. In e.g. the work by Jonsson [Jon87], a method for establishing ordinary forward simulation between systems is presented, in which the concrete and abstract systems are combined, but without the queue. This method can be regarded as a special case of proof by transduction, where the queue of the transducer is always empty. Several other presentations of standard simulation are found in e.g. [AL91, LT87, LS90, Sta88, Ora89].

The definition of sequential consistency originates in the work by Lamport [Lam79], but the interest in weaker consistency models for shared memory has become much larger in recent years, due to the development of multiprocessor systems such as the DASH multiprocessor [LLG+90]. A framework for describing memory models, e.g. sequential consistency has been developed by Dubois, Scheurig, and Briggs [DSB86]. This framework is based on auxiliary definitions concerning the propagation of write and read operations between different processors, which would be difficult to formalize in an existing framework for verification of correctness. Definitions of and reasoning about memory models can be based on the definition of different ordering constraints between memory operations [SS88, GAG+92]. Other frameworks, e.g. by Afek, Brown, and Merritt [ABM93] and by Collier [Col92] describe memory models in terms of how the processors view complete execution histories.

Chapter 8

Sequential Consistency Using Global Equivalence Proofs and Temporal Logic

S. Katz

8.1 Introduction

Temporal logics have been defined that exploit information on partial order among events in a distributed system. The temporal logic we consider is based on the idea of a *partial order computation* (also called a *run*) which is simply a maximal set of occurrences of operations (called events) of a distributed system that have some partial ordering among them. The ordering includes any causality required among events, and may have additional restrictions. Events which are ordered are called *dependent*, and the others are *independent*. A program or system defines a collection of such runs. In the version of this approach to be shown here, presented previously in [KP90, KP92b, KP92a], the collection of all linearizations of the events that are consistent with the partial order are considered in a temporal logic framework. Each such linearization is viewed as generating a sequence of alternating events and global states, that represents an execution sequence. All such execution sequences generated from a given run are called an *interleaving set* and are considered equivalent. Here 'equivalence' is used in the sense that the only difference between the execution sequences in an interleaving set is that strictly independent operations are executed in a different order.

In the temporal logic $ISTL^*$, a branching time assertion is interpreted as being *true* for a distributed system, if it is true for every interleaving set of the system. (This is analogous to the standard interpretation of a linear temporal logic assertion being true of a system if it holds for every execution sequence.) Then it is easy to express that each equivalence class has some execution sequence satisfying a property p, simply as Ep. Such properties are often natural for distributed systems and allow expressing specifications for problems such as database serializability, distributed snapshots, and, as will be shown below, sequential consistency of cache-based shared memory systems.

In addition, for many properties it is true that $Ep \Rightarrow Ap$, i.e., if p is true of one execution in an interleaving set, then it is true for all the others in that set. For such properties, verification can be made more efficient by showing generically that p is a property for which $Ep \Rightarrow Ap$, then explicitly showing Ep, and using modus ponens to conclude Ap.

Thus properties of the form Ep can arise in a variety of contexts, and proof rules have been presented that allow concluding Ep. In such rules there are actually two tasks that are mixed together. One task is to show that p is true for the executions that are identified as the ones to be explicitly considered, and the other is to show that sufficient executions have been chosen to 'cover' all of the equivalence classes with at least one representative. The motivation for showing both properties at once is to allow a classic iterative proof on the computation, maintaining compositionality and modularity in the proof. At each step we can assume both that p is true for (some extension of) the parts of the computations considered so far, and that sufficient computations are being considered. This allows compositional proofs and proof rules to be used, but has the price of complicated proof rules[KP92b, PP90]. In the inductive step, it is necessary to show that the states reached so far all have a possible next state that will both maintain p and extend the existing computations to sufficient representatives.

Here a complete separation is suggested between showing that each of a chosen set of computations (called the *convenient computations*) fulfills the needed properties, and showing that every computation is equivalent to one of the convenient ones. The proof of the first aspect uses the usual iterative approaches, while the proof of the second aspect is global, and uses temporal logic assertions about the entire computation, along with formulas that encode which operations are independent of each other. The advantage of this separation is that different kinds of reasoning can be used for the two aspects, each most natural for the problem at hand.

This approach is demonstrated in the context of refinements of distributed systems, gradually replacing high level atomic operations by a collection of lower level operations that loosen the

synchrony among distributed processes, but still maintain some key properties. Each refinement is divided into two independent proof stages. The first stage shows that convenient executions of operations from the next lower level are a simple refinement of executions from the upper level, and can be demonstrated correct using standard refinement mappings.

Then we show that every additional execution sequence at the lower level is equivalent to one of the convenient ones. This stage could be considered as a 'loosening' of the ordering imposed by the convenient executions. The two-step reasoning at each level saves having to directly relate each lower level sequence through a mapping to an upper level one. Although such a mapping exists, it may require the use of history and prophecy variables, and be extremely difficult to express and justify. This is because the collection of lower level operations that can be considered the 'implementation' of an upper level one are interleaved with an arbitrary number of operations that implement other higher level operations. Thus it is difficult to obtain an iterative proof that is uniform for all the computations when a direct mapping is required.

As a first example of this approach, we treat the replacement of an abstract sequential global memory by a less synchronized version with queues between the processes and the global memory. In the abstract version, each process can execute atomic read and write operations directly from the memory. In the lower level version, a process can only write to a local queue, while later the head of the queue is written to the memory internally. This is one basic step in a series of refinements that can be used to derive a lazy caching protocol that maintains what is known as *sequential consistency*. Intuitively, this means that the projection of local events of each process is consistent with use of the serial memory, even if a version with queues and local caches is being used instead.

The cache consistency protocol we treat is presented in [ABM93] and in the introductory paper of this issue. It has served as the basis for a variety of attempts to prove its correctness, in the framework of the Esprit REACT project [Ger93]. Sequential consistency seems, by its very definition, to favor the interleaving set view that considers the set of all total orders of events that are consistent with a partial order, as the semantic object to be considered.

Once we introduce queues, it is easy to define convenient executions for them and show that these implement the more abstract level. The fact that each other lower level execution sequence is equivalent to some convenient sequence is of course a crucial aspect of the correctness proof. It will be necessary to restrict the use of the queues on the implementation level, in order to guarantee this property. This will be expressed as another term in a temporal logic formula. As we shall see below, care must be taken in defining which events are dependent, in order to obtain the appropriate equivalence relation and/or partial ordering.

The rest of this paper is structured as follows. We first explain in more detail the idea of (convenient) interleaving sequences and the dependency relation. The implications for independence of queue operations are also examined. The version of temporal logic used is then briefly described. Section 8.4 explains the conjuncts that define the independence relation and other temporal formulas that describe the lower level execution sequences for the first refinement. In Section 8.5 these are summarized and used in a semantic version of the proof, basically a description of the temporal reasoning necessary to show that other executions are equivalent to the convenient ones. In Section 8.6 further steps in deriving the cache consistency algorithm are described, again in terms of temporal formulas that express independence and restrict the possible execution sequences.

8.2 Defining dependencies and convenient executions

The convenient executions at the lower level are precisely those where the lower level operations that implement a higher level one are all done sequentially, with no other lower level operations interspersed. These are legal lower level executions, even if they are unlikely to occur in practice because the operations are distributed in a collection of asynchronously executing processors. A mapping function from each convenient execution to some abstract computation is generally simple and iterative. After this first stage, we have only shown that every convenient execution sequence is a refinement of some higher-level abstract execution. The loosening stage requires precise reasoning about which operations are independent in which states. Each operation is viewed as a guard c (i.e., a condition for applicability on the state s) followed by a command f that is simply a function of s (with the operation written $c \rightarrow f$), as in [ABM93]. Note that such an interpretation of an event is reasonable only when a state is assumed as a semantic object, as part of the definition of an execution sequence. Then two operations, say $op1$ and $op2$, are *independent* in a state s, denoted $s \Rightarrow I(op1, op2)$, if beginning in state s neither affects the truth of the other's guard, and the result of executing them in either order is the same, i.e.,

$$c1(s) \Rightarrow (c2(f1(s)) \Leftrightarrow c2(s))$$

$$c2(s) \Rightarrow (c1(f2(s)) \Leftrightarrow c1(s))$$

$$(c1(s) \wedge c2(s)) \Rightarrow (f1(f2(s)) = f2(f1(s)))$$

The definition above is known as *conditional independence*[KP92a] because a pair of operations may be dependent in some states, and independent in others. The states in which two operations are independent are defined by a state predicate. Two execution sequences are considered equivalent if they differ only in that independent operations were done in a different order, but all dependent operations are done in the same order. The reasoning used to show the equivalence of two computations is quite different from that used to show the mapping from a higher to a lower level. If we are given a collection of independent operations in various states, then two sequences are equivalent if they differ only by interchanging two adjacent operations beginning at a state where they are independent. The equivalence class we consider is the transitive closure of this 'exchange' relation.

When more complex data structures are assumed, the dependencies become more complicated, and the extra freedom is exploited by the lower level implementation.

As a particularly relevant example, we consider the dependencies for a queue q with operations $empty(q)$, $put(q, e)$, and $get(q, e)$, where e is a data element.
When the queue is non-empty, then $put(q, e)$ is independent of $get(q, f)$:

$$(\neg empty(q)) \Rightarrow I(put, get) \tag{8.1}$$

When the queue is empty, a put and a get operation will be dependent:

$$empty(q) \Rightarrow \neg I(put, get) \tag{8.2}$$

All adjacent pairs of put's are dependent:

$$\neg(I(put, put)) \tag{8.3}$$

All adjacent pairs of get's are dependent:

$$\neg(I(get, get)) \tag{8.4}$$

The first rule is intuitively true because a *put* and a *get* by different processes on a nonempty queue are done at opposite ends of the queue, and never involve the same item, while this is not so when the queue is initially empty. In that case the *get* operation must follow a *put*.

The other rules follow from the fact that the contents of the queue differs according to the order of *put*'s, while the states of the rest of the system differ if *get*'s are done in a different order. A formal proof of these dependencies could be based, for example, on an algebraic specification of the queue axioms.

8.3 The logic

The version of temporal logic used in this paper will be briefly summarized. This is an adaptation of the logic $ISTL^*$ introduced in [KP90], with additions to facilitate showing equivalence of execution sequences. Most of the operators are those of CTL^* [EH86], but interpreted as true for a system if they hold for each interleaving set. An interleaving set is defined as an equivalence class of computations under exchanges of operations that can be done when the independence relation I holds. The syntax is thus standard, and the semantics (implicitly) universally quantifies over the interleaving sets:
Ap – for every computation in each interleaving set, p is true
Ep – for some computation in each interleaving set, p is true
Fp – eventually for some state, p is true
Gp – for every state from the present, p is true
Xp – for the next state, p is true
pUq – p is true until q becomes true (and q does become true)

In order to facilitate reasoning about sequences of operations, we add some conventions. First, an operation name also serves as a state predicate that is true precisely when that operation was executed in the transition from the previous state. (An alternative temporal logic that treats operations more directly can be seen in Lamport's TLA [Lam95]). Then sequences of operations (or other predicates) can be denoted as
"$s; t$" – defined as $X(s \wedge Xt)$ (in the next state s holds, followed by a state with t). This expression relates to a single execution sequence and can be preceded by E or A.

Note that in the starred version of the logic, there is no restriction on which combinations of the temporal operators are allowed. When temporal logics are used in model checking of finite state programs, as is done for CTL, it is common to restrict the combinations in order to facilitate efficient checking. In particular, the modalities E and A are known as *state* modalities because they deal with all of the possible continuations from a given global state. Such modalities are required to alternate with the other modalities, known as *path* modalities since they deal with restrictions on a given path. Although many aspects of the specification below can be treated in $ISTL$ with alternating state and path modalities, here we do not treat whether such restrictions allow sufficient expressibility, since in any case, model checking techniques are not used.

Additional information on I within the temporal descriptions of computations means that more execution sequences can be proven equivalent. In some sense the equivalence classes are demonstrably larger and fewer convenient executions are required to guarantee that each equivalence class contains a convenient execution.

8.4 Expressing independence and allowed computations

The definition of sequential consistency used in this paper can be stated as follows.

A memory M is sequentially consistent with respect to a serial memory M_{serial}, iff

$$\forall \sigma \in Beh(M) \exists \tau \in Beh(M_{serial}) \forall i = 1 \ldots n \;\; \sigma|i = \tau|i$$

$Beh(M)$ is the set of execution sequences associated with a system M, and $Beh(M_{serial})$ is the set where read and write operations are atomically done on the global memory. The above asserts that the projections on each process are the same as those in some execution using a serial memory, even though the general behavior may have extra internal steps associated with the memory, so that a write operation may not affect the memory directly. This statement suggests the interleaving set approach, since it closely relates to the idea of convenient sequences: the behavior of the serial memory will be viewed as consisting of lower level convenient sequences, where all lower level executions are *equivalent* to such a convenient execution. That is, if we now view M_{serial} as a temporal logic predicate true of the lower level serial computations, we require $E\ M_{serial}$.

We must define an independence relation so that the system is sequentially consistent if every execution is equivalent to a convenient serial one. That is, we require formulas in $ISTL^*$ that express the independence of adjacent operations (i.e., when I is true), that characterize the convenient serial computations, and that characterize every computation (including restrictions on when values can be read). Once these have been defined, we need to show that assuming the formula that defines the independence of operations, and the formula that defines all computations, $E\ M_{serial}$ is true.

In defining the independence relation so that it reflects sequential consistency, the local operations of each processor must be unchanged in the equivalent convenient version. Thus we assume a total order among local operations of a single processor. Since this order must be maintained for all equivalent execution sequences, we obtain the identity of local projections for every two equivalent execution sequences, as required in the definition of sequential consistency. For any two operations a_i and b_i, executed by process i, we therefore require

$$\neg I(a_i, b_i) \tag{8.5}$$

Of course, local operations a_i and b_j of *different* processes are independent:

$$i \neq j \Rightarrow I(a_i, b_j) \tag{8.6}$$

We consider how to refine abstract *read* and *write* actions. An abstract *write* action can be implemented by adding to the end of a queue the pair consisting of the value to be written and the memory address, later removing that pair from the head of the queue, and then writing it in the memory. If we denote the action of putting the value-address pair in the queue by $W(d, v)$, and the action of removing the pair from the head of the queue and writing to the memory by $MW(d, v)$ (standing for *Memory Write*), such a pair is the implementation of the abstract *write*. Thus W is associated with a *put* operation, and MW combines a *get* with a memory write.

Similarly, an abstract *read* could be implemented by reading from the memory, adding the value-location pair to another queue, and later reading the value-address pair from the head of that queue into the local process. However, the treatment of reads will be postponed to a second level of refinement, so for the present we assume a direct atomic read action denoted $R(d, v)$, meaning that value d is read from address (or variable) v.

In order to capture the intuition of reading and writing into memory, we express that the value returned for a variable or memory location x in an action $R(c, x)$ is the last value written into it by a $MW(d, x)$ action, in the assertion:

$$(MW(d, v) \wedge (\neg MW(b, v)) UR(c, v))) \Rightarrow c = d \tag{8.7}$$

This is known as *read/write consistency* and is a fundamental assumption when truly atomic reads and writes are being used. However, when reads and writes occur at different processes, and are not atomic, we can weaken the requirement.

This requirement does not seem to appear explicitly in [ABM93]. However, the operations there are defined using a *Memory* data structure (an array representing the contents of memory), and the effects of the atomic operations are defined so that a value can be returned for a variable only if it is the latest value written to that variable. Thus the same consistency requirement is simply given implicitly.

If we now replace the abstract read and write actions of the serial memory by the lower level actions above, we arrive at a situation that can be viewed as the addition of abstract write queues to the serial memory. Since we have a collection of such write queues, the "lower" level involves operations on an Out_i queue between the processor i and the central memory, for each processor. Since there now is a queue for each processor, we denote a write to the end of the ith queue by W_i, and removing an element from the head of that queue plus writing to the memory by MW_i. Reading by process i is denoted by R_i. All of these have the same parameters as previously, namely the value and the address (or variable name). The events that are considered local to a process i are not independent, and these include all occurrences of W_i and R_i, but not MW_i. On this level only the MW_i and R_i operations directly involve the memory and are required to satisfy read/write consistency.

In the convenient executions, items are inserted by the process i using W_i operations into the corresponding Out_i queue and immediately removed and copied to the central memory by the MW_i action. In these very particular computations, every W_i is immediately followed by writing into the memory using MW_i, with no intervening operations anywhere in the system. The queues are thus always empty except when a single item has just been put in and has not yet been written to the memory in the next step. In temporal logic we can state the requirement for a convenient computation as simply

$$G(\, W_i(c,x) \Leftrightarrow XMW_i(c,x)) \tag{8.8}$$

That is, throughout the computation, if a W_i has occurred, it is immediately followed by the corresponding MW_i, and every MW_i is preceded by a W_i with the same parameters. Every adjacent $W_i; MW_i$ pair is clearly a trivial implementation of the direct write on the abstract level. Since the read events R_i are still atomic, all convenient execution sequences can be easily shown to implement the abstract sequences, by a trivial induction on the sequence.

Then we need to claim that every execution of the lower level satisfying the queue axioms and the memory consistency assumptions is equivalent under the independence relation I to one of the convenient executions. This is almost true, but we need to restrict the read operations of the lower level to maintain the total order among local actions of a single process. Consider a situation where a process has written a pair (d,x) to its Out queue, then executes a read operation (implemented as an R) on x, and only then does a MW execute on that queue, changing the memory. The value read is clearly whatever was in the memory before the last MW. This implies that there is a linearization consisting of

$$W_i(d,x);\, R_i(c,x);\, MW_i(d,x)$$

with $d \neq c$. But such a computation is not consistent with the dependency requirements, because we claim that it is not equivalent to any convenient computation. If we wish to find a convenient execution to which this one is equivalent, we must show that the R operation can be exchanged, either with the following MW or the preceding W. The former exchange would lead to

$$W_i(d,x);\, MW_i(d,x);\, R_i(c,x)$$

This is not a convenient execution, since it violates the restrictions on the value read being the last one written in the memory location (read/write consistency). Exchanging the R_i and W_i operations would lead to

$$R_i(c, x);\ W_i(d, x);\ MW_i(d, x)$$

This is a convenient sequence, but is not equivalent to the original one, because it does not have the same total order of the local operations in process i.

This difficulty is solved by simply requiring that the lower level operations be restricted so that any read operation by a process i, R_i, is 'delayed' until the Out_i queue is empty, i.e., until all of the 'pending' MW_i operations have been done. In that case the problematic computation described above is simply declared impossible. Of course, there is no such restriction for reads and writes from *different* processes. The restriction on the implementation is again a temporal logic formula and can be expressed in several ways. One approach treats the actions directly, using a # symbol to denote the number of times an operation has occurred:

$$AG(R_i \Rightarrow (\#W_i = \#MW_i))$$

That is, no R_i is between a W_i and an MW_i, because every W_i before R_i has a corresponding MW_i that also appears in the execution sequence before R_i. Another way to express this is to define a predicate $empty$ that is true when the queue is empty and simply state that

$$AG(R_i \Rightarrow empty(Out_i)). \tag{8.9}$$

Such a predicate can be defined using temporal formulas derived from well-known algebraic axioms to first define $number$ in terms of each operation (incrementing when an item is inserted and decrementing when one is removed) so that $empty$ can be seen as a derived predicate true when $number = 0$. We shall assume that expressions defining such predicates have been defined, and use the second alternative.

The independence relations define what exchanges of operations can be made, and thus which computations are equivalent. This needs to be introduced into the logic explicitly, through the formula

$$AG(\ I(a, b)\ \Rightarrow\ (("a; b") \Leftrightarrow ("b; a"))) \tag{8.10}$$

In words, if $I(a, b)$ holds in a state, then the sequences that begin in that state and then have "$a; b$" are equivalent to those with "$b; a$". at that point.

8.5 Proving refinements

The proof requirements of showing a refinement that satisfies sequential consistency are obtained by using the relations from the previous section. The independence relations for queues (1–4) will have W_i corresponding to put and MW_i to get for each queue Out_i. We also have the independence and dependence relations on all local actions in each process (5–6). To these we add the read/write consistency rules for simple memory locations (7), the delay condition on reads defined above (9), and the formula connecting I and equivalence. We then claim that an execution sequence satisfying these dependencies must be equivalent (under the relations I) to one where all $W - MW$ pairs from the same queue are adjacent (8), i.e., to one of the convenient sequences. Note that the convenient sequences are assumed to have already been shown to correspond to abstract atomic read/write consistency. In terms of $ISTL^*$, the restrictions on the possible lower level computations must imply $EConvenient$, where $Convenient$ is the temporal logic definition of the convenient sequences.

queues, for process i:

$$(\neg empty(Out_i)) \Rightarrow I(W_i, MW_i)$$

$$empty(Out_i) \Rightarrow \neg I(W_i, MW_i)$$

locality, for a, b operations W or R in processes i, j:

$$\neg I(a_i, b_i)$$

$$i \neq j \Rightarrow I(a_i, b_j)$$

read/write memory consistency, for all processes i, j, and k:

$$AG(\ (MW_i(d, v) \wedge (\neg MW_j(b, v))\, UR_k(c, v))) \Rightarrow c = d\ )$$

delay of reads, for process i:

$$AG(R_i \Rightarrow empty(Out_i)).$$

independence and equivalence, for operations a and b:

$$AG(\ I(a, b)\ \ \Rightarrow\ \ (\ (\text{“}a; b\text{”}) \Leftrightarrow (\text{“}b; a\text{”})\ ))$$

Figure 8.1: Conjuncts in the correctness formula of a refinement

The conjuncts in the correctness formula are summarized in Figure 8.1. With the restrictions we have added, this implication is not difficult to prove. Consider any sequence satisfying read/write consistency and read delays. Assuming the other formulas in Figure 1 (that define independence), we want to show

$$EG(\ W_i(c, x) \Leftrightarrow XMW_i(c, x)).$$

We prove by induction on the number of states (or operations, since the two alternate) between a $W_i(d, x)$ — $MW_i(d, x)$ pair that correspond to putting a value in the Out_i queue and later removing it. If the two are adjacent, this pair is part of a convenient execution. If there is one state between them, and in that state $MW_j(c, y)$ for any j, c, and y, the independence relations show that there is an equivalent computation with the MW_j before the $W_i(d, x)$. The same is true of any R_j or W_j where $j \neq i$. If in that state there is another W_i it can be exchanged with the following $MW_i(d, x)$ (and recall that there cannot be an R_i). In general, note that there cannot be a 'matching' pair between another such pair from the same process, because that would violate the queue axioms. Assume that for all pairs with n states between them, we can find equivalent computations where the pairs are adjacent. For a pair with $n + 1$ states between them, if the first state is anything except $W_i(c, y)$, the action and the resultant predicate can be exchanged with the previous $W_i(d, x)$, using the independence assertions, and the inductive hypothesis can be used. Otherwise, we have a situation of the form

$$W_i(d, x);\ W_i(c, y); \underbrace{\cdots}_{n}; MW_i(d, x)$$

Again using the inductive hypothesis, the n remaining actions can be exchanged either after the $MW_i(d, x)$, or before the pair of actions $W_i(d, x)$; $W_i(c, y)$ because any action (except R_i's, which are excluded by assumption) that can be exchanged with $W_i(d, x)$ can also be exchanged with $W_i(c, y)$. Finally, the $W_i(c, y)$ can be exchanged with the $MW_i(d, x)$, since they are independent.

The proof here is simply a systematic analysis of which pairs of operations are independent under what conditions, in order to show that any computation is equivalent to a convenient one. We show exchanges that bring a general computation 'closer' according to some measure to a convenient one.

Just as for the abstract *write* actions, we could refine *read* actions into a pair of actions $MR_i(c,x)$ and $R_i(c,x)$, where in this case the memory read MR_i reads from the main memory and puts the pair read at the end of a queue, while the process read action R_i takes from the head of the queue and reads the pair into the process. Note that the MR_i must precede the R_i. The convenient sequences would have $MR_i(c,x); R_i(c,x)$ subsequences. In fact, the reading is handled in another way in the cache consistency algorithm, seen in the following section.

8.6 Further refinements

Further top-down development of the lazy caching algorithm could similarly be divided into a series of refinements, with each described first by a convenient sequence, followed by a loosening stage to the rest of the computations at that level. Note that the convenient executions are lower level implementations of *any* computation from the upper level, and not just the convenient upper level ones. Although we will not treat the other levels in as great detail as above, the convenient executions and the type of reasoning necessary is described in this section. The idea of the implementation described in the introductory paper and in [ABM93] is that a local cache memory of bounded size is associated with each process, and updates to the global memory are also inserted in a queue for each process, from which they are transferred to the local memory.

On this level, In queues are used. A lower level MW_i operation, in addition to removing an element from the head of the Out_i queue and writing it to the memory, now also adds the update requests to the In queue of each process. Alternatively, we could view this strengthened MW_i operation as simply the previous MW_i that only wrote to the main memory, followed immediately by an automatic MR operation for each process, that adds that same value to the end of the In queue of the process. We prefer to treat the operation in this way because such a view maintains the option of later additional refinements that loosen the atomicity of writing to the In queues. The temporal predicate that describes the possible computations for now will simply require that every needed MW/MR sequence appears atomically with no intervening operations. That is, we have

$$AG(MW_i(c,x) \Rightarrow \text{“}MR_1(c,x);\cdots;MR_n(c,x)\text{”})$$

A CU_i event removes an update request from the head of the In_i queue, and writes in the cache according to the update. A read request R_i is now from the local cache rather than from the central memory or from an abstract queue. Thus read/write consistency must hold for the central memory between MW and MR operations, and within each cache for CU_i and R_i. That is, for each process i, j, and k,

$$AG(\ (MW_i(d,v) \wedge (\neg MW_j(b,v))\, UMR_k(c,v))) \Rightarrow c = d\ )$$

$$AG(\ (CU_i(d,v) \wedge (\neg CU_i(b,v))\, UR_i(c,v))) \Rightarrow c = d\ )$$

The convenient sequences for this level are simply those computations for which every MW_i event and the subsequent MR's for each process are immediately followed by a subsequence with a single CU_j event for every process j. That is, again the queues will be empty, have one item inserted, and immediately use that item to write in the local caches. We view a "MW-subsequences of MR's and CU's" as the implementation of a simple MW which only wrote to the central memory, and a subsequent MR. In this case, for these convenient execution sequences, each cache is the same as

the central memory when outside such subsequences. By the considerations above for the convenient sequences, a read from the cache will give the same result as the higher level read from the central memory.

The CR_i read actions of a process i will again be restricted in order to allow showing equivalence to convenient sequences. Recall that this is intended to prevent local inconsistency where a process could sense that a W_i action has not yet 'taken effect' when a subsequent local read is done. As before, read actions of process i cannot occur while Out_i is nonempty. Similarly, those items in the In_i queue that represent updates that were originally initiated by process i itself, must be removed from the In_i queue and written to the cache before a read of that variable by process i. This is needed, just like the flushing of the Out queue required before a memory read operation by that process, in order to guarantee local total order. In the description in the introductory paper, these are the 'starred' items in the In queues. In terms of operations, R_i occurs in an execution only after all CU_i events that correspond to previous local W_i actions have occurred. This will simply be an assumption true of executions in the implementation. Once again there are several possible ways to introduce this into the logical assertions. For simplicity, we assume that the stars explained in the introduction are associated with elements in the In_i queue that were inserted immediately as a result of a MW_i operation, and that a predicate *hasstars* is true when there are such items in the queue. Simple assertions that define this predicate in terms of the operations are again assumed. Then we have

$$AG(R_i \Rightarrow (empty(Out_i) \wedge \neg hasstars(In_i))$$

Note that the restrictions on read operations are trivially true for the convenient executions, because in those the Out and In queues are always empty when a cache read occurs.

In a real cache consistency algorithm, the possibility of cache misses must also be treated. The idea behind cache misses is that the cache has limited capacity and can therefore not mirror the whole of the central memory. Sometimes variables are removed from the cache so that values for other variables can be put into the cache. This is modelled by adding internal cache invalidate actions CI_i that remove value-address pairs from the cache, i.e.,

$$AG(CI_i(x) \Rightarrow (\forall c.\neg(c, x) \in Cache_i))$$

Note that this is part of the assertions that define the $\in$ predicate, and that the other assertion defining $\in$ is

$$AG(CU_i(c, x) \Rightarrow (c, x) \in Cache_i).$$

A cache read $R_i(d, x)$ can only occur if the pair (d, x) is in the cache. If there is no value for x in the cache because of a cache invalidate action, the read must be delayed until the needed pair is retrieved from the central memory. This is done by repeating MR operations to read a value for a variable from the central memory and putting it into the In queue of only the process that had the CI event, so that the pair eventually is put back into the cache by a cache update. As previously, this description can be captured by a temporal logic assertion further restricting when an R can occur:

$$AG(R_i(d, x) \Rightarrow (d, x) \in Cache_i).$$

The assumptions about possible computations and independence are summarized in Figure 2.

Convenient sequences at this level now consist of sequences where all MR and corresponding CU actions immediately follow one another, and each $CI_i(x)$ is followed immediately by $MR_i(c, x)$; $CU_i(c, x)$ for the value c in the central memory, whenever there is a later $R_i(c, x)$

(i.e., whenever the value is subsequently needed for reading from the cache). If there is no subsequent cache read of that variable, a $CI_i(x)$ event does not have to be followed by any other related event.

As previously, assuming these temporal formulas, (and straightforward but tedious formulas that define the predicates *empty* and *hasstar*) we must show that E *Convenient* holds, where *Convenient* is now a temporal formula describing the computations with writes that take immediate effect in all caches, and cache misses that are immediately rectified by reading from memory at arbitrary points. In showing that the convenient executions correctly implement the higher level, the subsequences $CI_i(x); MR_i(c,x); CU_i(c,x)$ that will occur after a cache miss (whenever x is still needed in that cache) in these convenient executions can be mapped to not having done anything on the abstract level. The value read from the central memory after a cache miss in this case is always the same as the one just erased because for the convenient executions the Out and In queues are always empty when a CI event occurs.

Of course, in a more realistic description, the capacity of the cache would be given, and in that case a CI operation may not be followed immediately by retrieving the value just removed, since then the space could not be used for the value of a different variable that is needed for a read operation first. In that case the convenient sequences could have the $MR_i(c,x); CU_i(c,x)$ later than the $CI_i(x)$, and immediately preceding the $R_i(c,x)$. That is, the needed value is retrieved just before it is read from the cache, and again the queues are always empty except when one element has just been put in and is about to be removed.

It remains to show that all other executions are equivalent to some convenient one. Again, all of the independence relations must be precisely analyzed. Note that when the $CI_i(x)$ operation is followed by other operations, and a $MR_i(d,x)$ occurs only after there have been intervening MW operations that change the main memory, the value read will be different than the one erased. However, since there is already evidently an entry in the In queue with the update, this is equivalent to first doing the update, and then the invalidate CI. Clearly, the relation $I(CI_i, MW_j)$ holds for all i and j. In fact, it can be shown that an occurrence of $CI_i(x)$ is independent of every other following operation except the last $CU_i(c,x)$ before the next $R_i(c,x)$. This can be used to show that each general sequence that satisfies the formulas is equivalent to one of the convenient ones, again using a proof by induction on the distance from the position of key operations (in this case, CI's) in a general execution sequence to their position in a convenient one.

8.7 Concluding remarks

In this paper we showed how to prove correction of the lazy caching algorithm through a series of refinements, starting from the definition of serial and sequentially consistent memory. Reasoning in terms of convenient sequences and their equivalence classes seems to be well-suited for this purpose. The independence relations and restrictions on possible implementations are easily expressed using $ISTL^*$. At each refinement, a two-stage proof is used, first showing that the convenient sequences are a simple refinement using usual mapping functions, and then separately showing every lower level computation equivalent to one of the convenient ones.

The steps in such proofs of equivalence are uniform. First, predicates are needed that make the independence of adjacent operations explicit. These can be justified from the underlying semantics of the model, or by properties of the data structures used. In the case of sequential consistency, the independence is further restricted by the problem specification, namely that there is a total ordering among local process writes and reads. These properties can often be shown once for a large collection of related problems.

Second, the properties of the general computations are described as global temporal logic predicates. These follow from a description of the implementation level. In the case of cache consistency, these include restrictions on when a read action is possible.

Next, the convenient computations are described, also using the temporal logic.

The claim to be proven is that under the equivalence defined by I, with the assumptions on the possible computations, E *Convenient* is true. The proof of this fact is done by induction showing that each computation is equivalent to one that is 'closer' to a convenient one. A systematic examination of which operations can be exchanged is done using the independence information. This aspect seems amenable to automation, since it involves a large number of very simple assertions.

In the example given, the main concern is on showing equivalence, and the convenient sequences are chosen so that the refinement proof is particularly easy. This does not always have to be the optimal division, and sometimes more effort will have to be devoted to showing that the convenient executions indeed satisfy the needed property.

Acknowledgement: Job Zwiers and Wil Janssen suggested the gradual refinement stages and showed connections to algebraic partial orders, and Rob Gerth helped to clarify the cache consistency protocol.

queues, for process i:

$$(\neg empty(Out_i)) \Rightarrow I(W_i, MW_i)$$

$$empty(Out_i) \Rightarrow \neg I(W_i, MW_i)$$

$$(\neg empty(In_i)) \Rightarrow I(MR_i, CU_i)$$

$$empty(In_i) \Rightarrow \neg I(MR_i, CU_i)$$

locality, for a, b operations W or R in processes i, j:

$$\neg I(a_i, b_i)$$

$$i \neq j \Rightarrow I(a_i, b_j)$$

read/write memory and cache consistency, for all processes i, j, and k:

$$AG(\ (MW_i(d,v) \wedge (\neg MW_j(b,v))UMR_k(c,v))) \Rightarrow c = d\ )$$

$$AG(\ (CU_i(d,v) \wedge (\neg CU_i(b,v))UR_i(c,v))) \Rightarrow c = d\ )$$

delay of reads, for process i:

$$AG(R_i \Rightarrow (empty(Out_i) \wedge \neg hasstars(In_i))$$

$$AG(R_i(d,x) \Rightarrow (d,x) \in Cache_i).$$

effect of cache invalidate and write, for process i:

$$AG(CI_i(x) \Rightarrow (\forall c.\neg(c,x) \in Cache_i))$$

$$AG(CU_i(c,x) \Rightarrow\ (c,x) \in Cache_i)$$

independence and equivalence, for operations a and b:

$$AG(\ I(a,b)\ \ \Rightarrow\ \ (\ (\text{“}a;b\text{”}) \Leftrightarrow (\text{“}b;a\text{”})\ ))$$

Figure 8.2: Conjuncts for computations with cache misses

Bibliography

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

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

[BB87] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.

[BBLS92] A. Bouajjani, S. Bensalem, C. Loiseaux, and J. Sifakis. Property preserving simulations. In *Workshop on Computer-Aided Verification (CAV), Montréal*, volume 630 of *Lecture Notes in Computer Science*. Springer Verlag, June 1992.

[BFKR92] H. Burkhardt, S. Frank, B. Knobe, and J. Rothnie. Overview of the KSR1 computer system. Technical Report SR-TR-9202001, Kendall Square Research, Boston, 1992.

[BJO91] E. Brinksma, B. Jonsson, and F. Orava. Refining interfaces of communicating systems. In Abramsky and Maibaum, editors, *Proc. Coll. on Combining Paradigms for Software Development*, volume 494 of *Lecture Notes in Computer Science*, pages 297–312. Springer Verlag, 1991.

[Bol92] T. Bolognesi. Catalogue of LOTOS correctness preserving transformations. Technical Report Lo/WP1/T1.2/N0045/V03, Esprit Project 2304 Lotosphere, April 1992.

[Bri92] Ed Brinksma. On the uniqueness of fixpoints modulo observation congruence. Lecture Notes in Computer Science 630, pages 62–76. Springer-Verlag, 1992.

[BW90] J.C.M. Baeten and W.P. Weijland. *Process algebra*. Cambridge University Press, 1990.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th POPL*, January 1977.

[CGL92] E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. In *Symposium on Principles of Programming Languages (POPL 92)*. ACM, January 1992.

[CM88] K. M. Chandy and J. Misra. *Parallel Program Design*. Addison-Wesley, Massachusetts, 1988.

[Col92] W. W. Collier. *Reasoning about Parallel Architectures*. Prentice Hall, 1992.

[DSB86] M. Dubois, C. Scheurig, and F.A. Briggs. Memory access buffering in multiprocessors. In *Proc. 13th Annual Int. Symp. on Computer Architecture*, pages 434–442, June 1986.

[EF82] T. Elrad and N. Francez. Decomposition of distributed programs into communication closed layers. *Science of Computer Programming*, 2, 1982.

[EH83] E. A. Emerson and J. Y. Halpern. 'Sometimes' and 'not never' revisited: On branching versus linear time. In *10th ACM Symposium on Principles of Programming Languages (POPL 83)*. ACM, 1983. also published in Journal of ACM , 33:151-178.

[EH86] E.A. Emerson and J.Y. Halpern. Sometimes and not never revisited: on branching versus linear time temporal logic. *Journal of the ACM*, 33:151–178, 1986.

[GAG+92] K. Gharachorloo, S. Adve, A. Gupta, J. Hennessy, and M.D. Hill. Programming for different memory consistency models. *Journal of Parallel and Distributed Computing*, 15:399–407, Aug. 1992.

[Gai89] Haim Gaifman. Modeling concurrency by partial orders and nonlinear transition systems. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354 of *Lecture Notes in Computer Science*, pages 467–488. Springer-Verlag, 1989.

[Ger93] R. Gerth(editor). Verifying sequentially consistent memory. Technical report, Esprit React report, 1993.

[Ger95] R. Gerth. Introduction to sequential consistency and the lazy caching algorithm. *Distributed Computing*, This issue, 1995.

[Gis84] J. L. Gischer. *Partial Orders and the Axiomatic Theory of Shuffle*. PhD thesis, Stanford University, 1984.

[GKS92] R. Gerth, R. Kuiper, and J. Segers. Interface refinement in reactive systems. In R. Cleaveland, editor, *Proceedings of the third International Conference on Concurrency Theory (CONCUR)*, volume 630 of *Lecture Notes in Computer Science*, pages 77–94. Springer Verlag, June 1992.

[GL93] S. Graf and C. Loiseaux. A tool for symbolic program verification and abstraction. In *Conference on Computer Aided Verification CAV 93, Heraklion Crete*, volume 697 of *Lecture Notes in Computer Science*. Springer Verlag, 1993.

[He 89] He Jifeng. Process simulation and refinement. *Formal Aspects of Computing*, 1:229–241, 1989.

[HHS87] C.A.R. Hoare, He Jifeng, and J.W. Sanders. Prespecification in data refinement. *Information Processing Letters*, 25:71–76, 1987.

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

[Jon85] B. Jonsson. A model and proof system for asynchronous networks. In *Proceedings 4th ACM Symposium on Principles of Distributed Computing*, pages 49–58. ACM, 1985.

[Jon87] B. Jonsson. Modular verification of asynchronous networks. In *Proc. 6th ACM Symp. on Principles of Distributed Computing*, pages 152–166, Vancouver, Canada, 1987. Extended Version as SICS Research Report R90010.

[Jon91] B. Jonsson. Simulations between specifications of distributed systems. In *Proc. CONCUR '91, Theories of Concurrency: Unification and Extension*, volume 527 of *Lecture Notes in Computer Science*, Amsterdam, Holland, 1991. Springer Verlag.

[Jos88] M.B. Josephs. A state-based approach to communicating processes. *Distributed Computing*, 3:9–18, 1988.

[JPR94] Bengt Jonsson, Amir Pnueli, and Camilla Rump. Proving refinement using transduction. Technical report, Department of Applied Mathematics and Computer Science, The Weizmann Institute of Science, Rehovot, Israel, 1994.

[JPZ91] W. Janssen, M. Poel, and J. Zwiers. Action systems and action refinement in the development of parallel systems. In *Proc. of CONCUR '91*, pages 298–316. Springer-Verlag, LNCS 527, 1991.

[JZ93] W. Janssen and J. Zwiers. Specifying and proving communication closedness in protocols. In *Proceedings of 13th IFIP symp. on Protocol Specification, Testing and Verification*. North-Holland, 1993.

[Koz83] D. Kozen. Results on the propositional μ-calculus. In *Theoretical Computer Science*. North-Holland, 1983.

[KP90] S. Katz and D. Peled. Interleaving set temporal logic. *Theoretical Computer Science*, 75:263–287, 1990.

[KP92a] S. Katz and D. Peled. Defining conditional independence using collapses. *Theoretical Computer Science*, 101:337–359, 1992.

[KP92b] S. Katz and D. Peled. Verification of distributed programs using representative interleaving sequences. *Distributed Computing*, 6:107–120, 1992.

[Kur89] R.P. Kurshan. Analysis of discrete event coordination. In *REX Workshop on Stepwise Refinement of Distributed Systems, Mook*, volume 430 of *Lecture Notes in Computer Science*. Springer Verlag, 1989.

[Lam79] L. Lamport. How to make a multiprocessor that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28:690–691, 1979.

[Lam83] L. Lamport. Specifying concurrent program modules. *ACM Trans. on Programming Languages and Systems*, 5(2):190–222, 1983.

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

[Lam95] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 1995. To appear.

[Lar90] Kim Guldstrand Larsen. Compositional theories based on an operational semantics of contexts. In J. W. de Bakker, W. P. de Roever, and Grzegorz Rozenberg, editors, *Stepwise Refinement of Distributed Systems — Models, Formalisms, Correctness*, volume 430 of *Lecture Notes in Computer Science*, pages 487–518. Springer-Verlag, 1990.

[LGS+94] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *To appear in Formal Methods in System Design*, 1994.

[LLG+90] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proc. 17th Annual Int. Symp. on Computer Architecture*, May 1990.

[LLG+92] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The Stanford Dash multiprocessor. *IEEE Computer*, pages 63–79, 1992.

[Loi94] C. Loiseaux. Vérification symbolique de programmes réactifs à l'aide d'abstractions. Thesis, Verimag, Grenoble, January 1994.

[Lon93] D. E. Long. Model checking, abstraction and compositional verification. Phd thesis, Carnegie Mellon University, July 1993.

[LS90] S. S. Lam and A. U. Shankar. Refinement and projection of relational specifications. In de Bakker, de Roever, and Rozenberg, editors, *Stepwise Refinement of Distributed Systems. Models, Formalisms, Correctness*, volume 430 of *Lecture Notes in Computer Science*, pages 454–486. Springer Verlag, 1990.

[LT87] N. A. Lynch and M. R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proc. 6th ACM Symp. on Principles of Distributed Computing*, pages 137–151, Vancouver, Canada, 1987.

[Maz89] A. Mazurkiewicz. Basic notions of trace theory. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354 of *Lecture Notes in Computer Science*, pages 285–363. Springer-Verlag, 1989.

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

[Mis84] J. Misra. Reasoning about networks of communicating processes. In *INRIA advanced Nato study institute on logics and models for verification and specification of concurrent systems, Nice, France*, 1984.

[Mos93] D. Mosberger. Memory consistency models. *ACM SIGOP Operating Systems Review*, 27(1):18–27, 1993.

[MP91] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, New York, 1991.

[MP94] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems*. Springer-Verlag, New York, 1994. To Appear.

[Ora89] F. Orava. Verifying safety and deadlock properties of networks of asynchronously communicating processes. In *Protocol Specification, Testing, and Verification IX*, pages 357–372, Enschede, The Netherlands, 1989. IFIP WG 6.1, North-Holland.

[Plo81] G.D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[Pnu85] A. Pnueli. In transition from global to modular temporal reasoning about programs. In *Logics and Models for Concurrent Systems*, volume 13 of *NATO, ASI Series F*. North-Holland, 1985.

[Pnu86] A. Pnueli. Specification and Development of reactive Systems. In *Conference IFIP, Dublin*. North-Holland, 1986.

[PP90] D. Peled and A. Pnueli. Proving partial order liveness properties. In *Proc. of 17th ICALP*, pages 553–571. Springer-Verlag, LNCS 443, 1990.

[Pra86] V. Pratt. Modeling concurrency with partial orders. *International Jornal of Parallel Programming*, 15(1):33–71, 1986.

[SG90] G. Shurek and O. Grumberg. The Modular Framework of Computer-aided Verification: Motivation, Solutions and Evaluation Criteria. In *Conference on Automatic Verification (CAV), Rutgers, NJ*, volume 531 of *Lecture Notes in Computer Science*. Springer Verlag, 1990.

[SL83] A.U. Shankar and S.S. Lam. An HDLC protocol specification and its verification using image protocols. *ACM Trans. on Computer Systems*, 1(4):331–368, Nov. 1983.

[SS88] D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *ACM Transactions on Programming Languages and Systems*, 10(2):282–312, April 1988.

[Sta88] E. W. Stark. Proving entailment between conceptual state specifications. *Theoretical Computer Science*, 56:135–154, 1988.

[vG93] Rob J. van Glabbeek. The linear time - branching time spectrum II. LectureNotes in Computer Science 715, pages 66 – 81. Springer-Verlag, 1993.

[ZJ94] J. Zwiers and W. Janssen. Partial order based design of concurrent systems. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Proceedings of the REX School/Symposium "A Decade of Concurreny", Noordwijkerhout, 1993*, Lecture Notes in Computer Science. Springer-Verlag, 1994.

[Zwi89] J. Zwiers. *Compositionality, Concurrency and Partial Correctness*, volume 321 of *Lecture Notes in Computer Science*. Springer Verlag, 1989.

Computing Science Reports

Department of Mathematics and Computing Science
Eindhoven University of Technology

In this series appeared:

91/01	D. Alstein	Dynamic Reconfiguration in Distributed Hard Real-Time Systems, p. 14.
91/02	R.P. Nederpelt H.C.M. de Swart	Implication. A survey of the different logical analyses "if...,then...", p. 26.
91/03	J.P. Katoen L.A.M. Schoenmakers	Parallel Programs for the Recognition of *P*-invariant Segments, p. 16.
91/04	E. v.d. Sluis A.F. v.d. Stappen	Performance Analysis of VLSI Programs, p. 31.
91/05	D. de Reus	An Implementation Model for GOOD, p. 18.
91/06	K.M. van Hee	SPECIFICATIEMETHODEN, een overzicht, p. 20.
91/07	E.Poll	CPO-models for second order lambda calculus with recursive types and subtyping, p. 49.
91/08	H. Schepers	Terminology and Paradigms for Fault Tolerance, p. 25.
91/09	W.M.P.v.d.Aalst	Interval Timed Petri Nets and their analysis, p.53.
91/10	R.C.Backhouse P.J. de Bruin P. Hoogendijk G. Malcolm E. Voermans J. v.d. Woude	POLYNOMIAL RELATORS, p. 52.
91/11	R.C. Backhouse P.J. de Bruin G.Malcolm E.Voermans J. van der Woude	Relational Catamorphism, p. 31.
91/12	E. van der Sluis	A parallel local search algorithm for the travelling salesman problem, p. 12.
91/13	F. Rietman	A note on Extensionality, p. 21.
91/14	P. Lemmens	The PDB Hypermedia Package. Why and how it was built, p. 63.
91/15	A.T.M. Aerts K.M. van Hee	Eldorado: Architecture of a Functional Database Management System, p. 19.
91/16	A.J.J.M. Marcelis	An example of proving attribute grammars correct: the representation of arithmetical expressions by DAGs, p. 25.

91/17	A.T.M. Aerts P.M.E. de Bra K.M. van Hee	Transforming Functional Database Schemes to Relational Representations, p. 21.
91/18	Rik van Geldrop	Transformational Query Solving, p. 35.
91/19	Erik Poll	Some categorical properties for a model for second order lambda calculus with subtyping, p. 21.
91/20	A.E. Eiben R.V. Schuwer	Knowledge Base Systems, a Formal Model, p. 21.
91/21	J. Coenen W.-P. de Roever J.Zwiers	Assertional Data Reification Proofs: Survey and Perspective, p. 18.
91/22	G. Wolf	Schedule Management: an Object Oriented Approach, p. 26.
91/23	K.M. van Hee L.J. Somers M. Voorhoeve	Z and high level Petri nets, p. 16.
91/24	A.T.M. Aerts D. de Reus	Formal semantics for BRM with examples, p. 25.
91/25	P. Zhou J. Hooman R. Kuiper	A compositional proof system for real-time systems based on explicit clock temporal logic: soundness and complete ness, p. 52.
91/26	P. de Bra G.J. Houben J. Paredaens	The GOOD based hypertext reference model, p. 12.
91/27	F. de Boer C. Palamidessi	Embedding as a tool for language comparison: On the CSP hierarchy, p. 17.
91/28	F. de Boer	A compositional proof system for dynamic proces creation, p. 24.
91/29	H. Ten Eikelder R. van Geldrop	Correctness of Acceptor Schemes for Regular Languages, p. 31.
91/30	J.C.M. Baeten F.W. Vaandrager	An Algebra for Process Creation, p. 29.
91/31	H. ten Eikelder	Some algorithms to decide the equivalence of recursive types, p. 26.
91/32	P. Struik	Techniques for designing efficient parallel programs, p. 14.
91/33	W. v.d. Aalst	The modelling and analysis of queueing systems with QNM-ExSpect, p. 23.
91/34	J. Coenen	Specifying fault tolerant programs in deontic logic, p. 15.

91/35	F.S. de Boer J.W. Klop C. Palamidessi	Asynchronous communication in process algebra, p. 20.
92/01	J. Coenen J. Zwiers W.-P. de Roever	A note on compositional refinement, p. 27.
92/02	J. Coenen J. Hooman	A compositional semantics for fault tolerant real-time systems, p. 18.
92/03	J.C.M. Baeten J.A. Bergstra	Real space process algebra, p. 42.
92/04	J.P.H.W.v.d.Eijnde	Program derivation in acyclic graphs and related problems, p. 90.
92/05	J.P.H.W.v.d.Eijnde	Conservative fixpoint functions on a graph, p. 25.
92/06	J.C.M. Baeten J.A. Bergstra	Discrete time process algebra, p.45.
92/07	R.P. Nederpelt	The fine-structure of lambda calculus, p. 110.
92/08	R.P. Nederpelt F. Kamareddine	On stepwise explicit substitution, p. 30.
92/09	R.C. Backhouse	Calculating the Warshall/Floyd path algorithm, p. 14.
92/10	P.M.P. Rambags	Composition and decomposition in a CPN model, p. 55.
92/11	R.C. Backhouse J.S.C.P.v.d.Woude	Demonic operators and monotype factors, p. 29.
92/12	F. Kamareddine	Set theory and nominalisation, Part I, p.26.
92/13	F. Kamareddine	Set theory and nominalisation, Part II, p.22.
92/14	J.C.M. Baeten	The total order assumption, p. 10.
92/15	F. Kamareddine	A system at the cross-roads of functional and logic programming, p.36.
92/16	R.R. Seljée	Integrity checking in deductive databases; an exposition, p.32.
92/17	W.M.P. van der Aalst	Interval timed coloured Petri nets and their analysis, p. 20.
92/18	R.Nederpelt F. Kamareddine	A unified approach to Type Theory through a refined lambda-calculus, p. 30.
92/19	J.C.M.Baeten J.A.Bergstra S.A.Smolka	Axiomatizing Probabilistic Processes: ACP with Generative Probabilities, p. 36.
92/20	F.Kamareddine	Are Types for Natural Language? P. 32.

92/21	F.Kamareddine	Non well-foundedness and type freeness can unify the interpretation of functional application, p. 16.
92/22	R. Nederpelt F.Kamareddine	A useful lambda notation, p. 17.
92/23	F.Kamareddine E.Klein	Nominalization, Predication and Type Containment, p. 40.
92/24	M.Codish D.Dams Eyal Yardeni	Bottum-up Abstract Interpretation of Logic Programs, p. 33.
92/25	E.Poll	A Programming Logic for Fω, p. 15.
92/26	T.H.W.Beelen W.J.J.Stut P.A.C.Verkoulen	A modelling method using MOVIE and SimCon/ExSpect, p. 15.
92/27	B. Watson G. Zwaan	A taxonomy of keyword pattern matching algorithms, p. 50.
93/01	R. van Geldrop	Deriving the Aho-Corasick algorithms: a case study into the synergy of programming methods, p. 36.
93/02	T. Verhoeff	A continuous version of the Prisoner's Dilemma, p. 17
93/03	T. Verhoeff	Quicksort for linked lists, p. 8.
93/04	E.H.L. Aarts J.H.M. Korst P.J. Zwietering	Deterministic and randomized local search, p. 78.
93/05	J.C.M. Baeten C. Verhoef	A congruence theorem for structured operational semantics with predicates, p. 18.
93/06	J.P. Veltkamp	On the unavoidability of metastable behaviour, p. 29
93/07	P.D. Moerland	Exercises in Multiprogramming, p. 97
93/08	J. Verhoosel	A Formal Deterministic Scheduling Model for Hard Real-Time Executions in DEDOS, p. 32.
93/09	K.M. van Hee	Systems Engineering: a Formal Approach Part I: System Concepts, p. 72.
93/10	K.M. van Hee	Systems Engineering: a Formal Approach Part II: Frameworks, p. 44.
93/11	K.M. van Hee	Systems Engineering: a Formal Approach Part III: Modeling Methods, p. 101.
93/12	K.M. van Hee	Systems Engineering: a Formal Approach Part IV: Analysis Methods, p. 63.
93/13	K.M. van Hee	Systems Engineering: a Formal Approach

		Part V: Specification Language, p. 89.
93/14	J.C.M. Baeten J.A. Bergstra	On Sequential Composition, Action Prefixes and Process Prefix, p. 21.
93/15	J.C.M. Baeten J.A. Bergstra R.N. Bol	A Real-Time Process Logic, p. 31.
93/16	H. Schepers J. Hooman	A Trace-Based Compositional Proof Theory for Fault Tolerant Distributed Systems, p. 27
93/17	D. Alstein P. van der Stok	Hard Real-Time Reliable Multicast in the DEDOS system, p. 19.
93/18	C. Verhoef	A congruence theorem for structured operational semantics with predicates and negative premises, p. 22.
93/19	G-J. Houben	The Design of an Online Help Facility for ExSpect, p.21.
93/20	F.S. de Boer	A Process Algebra of Concurrent Constraint Programming, p. 15.
93/21	M. Codish D. Dams G. Filé M. Bruynooghe	Freeness Analysis for Logic Programs - And Correctness?, p. 24.
93/22	E. Poll	A Typechecker for Bijective Pure Type Systems, p. 28.
93/23	E. de Kogel	Relational Algebra and Equational Proofs, p. 23.
93/24	E. Poll and Paula Severi	Pure Type Systems with Definitions, p. 38.
93/25	H. Schepers and R. Gerth	A Compositional Proof Theory for Fault Tolerant Real-Time Distributed Systems, p. 31.
93/26	W.M.P. van der Aalst	Multi-dimensional Petri nets, p. 25.
93/27	T. Kloks and D. Kratsch	Finding all minimal separators of a graph, p. 11.
93/28	F. Kamareddine and R. Nederpelt	A Semantics for a fine λ-calculus with de Bruijn indices, p. 49.
93/29	R. Post and P. De Bra	GOLD, a Graph Oriented Language for Databases, p. 42.
93/30	J. Deogun T. Kloks D. Kratsch H. Müller	On Vertex Ranking for Permutation and Other Graphs, p. 11.
93/31	W. Körver	Derivation of delay insensitive and speed independent CMOS circuits, using directed commands and production rule sets, p. 40.
93/32	H. ten Eikelder and H. van Geldrop	On the Correctness of some Algorithms to generate Finite Automata for Regular Expressions, p. 17.

93/33	L. Loyens and J. Moonen	ILIAS, a sequential language for parallel matrix computations, p. 20.
93/34	J.C.M. Baeten and J.A. Bergstra	Real Time Process Algebra with Infinitesimals, p.39.
93/35	W. Ferrer and P. Severi	Abstract Reduction and Topology, p. 28.
93/36	J.C.M. Baeten and J.A. Bergstra	Non Interleaving Process Algebra, p. 17.
93/37	J. Brunekreef J-P. Katoen R. Koymans S. Mauw	Design and Analysis of Dynamic Leader Election Protocols in Broadcast Networks, p. 73.
93/38	C. Verhoef	A general conservative extension theorem in process algebra, p. 17.
93/39	W.P.M. Nuijten E.H.L. Aarts D.A.A. van Erp Taalman Kip K.M. van Hee	Job Shop Scheduling by Constraint Satisfaction, p. 22.
93/40	P.D.V. van der Stok M.M.M.P.J. Claessen D. Alstein	A Hierarchical Membership Protocol for Synchronous Distributed Systems, p. 43.
93/41	A. Bijlsma	Temporal operators viewed as predicate transformers, p. 11.
93/42	P.M.P. Rambags	Automatic Verification of Regular Protocols in P/T Nets, p. 23.
93/43	B.W. Watson	A taxomomy of finite automata construction algorithms, p. 87.
93/44	B.W. Watson	A taxonomy of finite automata minimization algorithms, p. 23.
93/45	E.J. Luit J.M.M. Martin	A precise clock synchronization protocol,p.
93/46	T. Kloks D. Kratsch J. Spinrad	Treewidth and Patwidth of Cocomparability graphs of Bounded Dimension, p. 14.
93/47	W. v.d. Aalst P. De Bra G.J. Houben Y. Kornatzky	Browsing Semantics in the "Tower" Model, p. 19.
93/48	R. Gerth	Verifying Sequentially Consistent Memory using Interface Refinement, p. 20.

94/01	P. America M. van der Kammen R.P. Nederpelt O.S. van Roosmalen H.C.M. de Swart	The object-oriented paradigm, p. 28.
94/02	F. Kamareddine R.P. Nederpelt	Canonical typing and Π-conversion, p. 51.
94/03	L.B. Hartman K.M. van Hee	Application of Marcov Decision Processe to Search Problems, p. 21.
94/04	J.C.M. Baeten J.A. Bergstra	Graph Isomorphism Models for Non Interleaving Process Algebra, p. 18.
94/05	P. Zhou J. Hooman	Formal Specification and Compositional Verification of an Atomic Broadcast Protocol, p. 22.
94/06	T. Basten T. Kunz J. Black M. Coffin D. Taylor	Time and the Order of Abstract Events in Distributed Computations, p. 29.
94/07	K.R. Apt R. Bol	Logic Programming and Negation: A Survey, p. 62.
94/08	O.S. van Roosmalen	A Hierarchical Diagrammatic Representation of Class Structure, p. 22.
94/09	J.C.M. Baeten J.A. Bergstra	Process Algebra with Partial Choice, p. 16.
94/10	T. verhoeff	The testing Paradigm Applied to Network Structure. p. 31.
94/11	J. Peleska C. Huizing C. Petersohn	A Comparison of Ward & Mellor's Transformation Schema with State- & Activitycharts, p. 30.
94/12	T. Kloks D. Kratsch H. Müller	Dominoes, p. 14.
94/13	R. Seljée	A New Method for Integrity Constraint checking in Deductive Databases, p. 34.
94/14	W. Peremans	Ups and Downs of Type Theory, p. 9.
94/15	R.J.M. Vaessens E.H.L. Aarts J.K. Lenstra	Job Shop Scheduling by Local Search, p. 21.
94/16	R.C. Backhouse H. Doombos	Mathematical Induction Made Calculational, p. 36.
94/17	S. Mauw M.A. Reniers	An Algebraic Semantics of Basic Message Sequence Charts, p. 9.

94/18	F. Kamareddine R. Nederpelt	Refining Reduction in the Lambda Calculus, p. 15.
94/19	B.W. Watson	The performance of single-keyword and multiple-keyword pattern matching algorithms, p. 46.
94/20	R. Bloo F. Kamareddine R. Nederpelt	Beyond β-Reduction in Church's $\lambda\rightarrow$, p. 22.
94/21	B.W. Watson	An introduction to the Fire engine: A C++ toolkit for Finite automata and Regular Expressions.
94/22	B.W. Watson	The design and implementation of the FIRE engine: A C++ toolkit for Finite automata and regular Expressions.
94/23	S. Mauw and M.A. Reniers	An algebraic semantics of Message Sequence Charts, p. 43.
94/24	D. Dams O. Grumberg R. Gerth	Abstract Interpretation of Reactive Systems: Abstractions Preserving ∀CTL*, ∃CTL* and CTL*, p. 28.
94/25	T. Kloks	$K_{1,3}$-free and W_4-free graphs, p. 10.
94/26	R.R. Hoogerwoord	On the foundations of functional programming: a programmer's point of view, p. 54.
94/27	S. Mauw and H. Mulder	Regularity of BPA-Systems is Decidable, p. 14.
94/28	C.W.A.M. van Overveld M. Verhoeven	Stars or Stripes: a comparative study of finite and transfinite techniques for surface modelling, p. 20.
94/29	J. Hooman	Correctness of Real Time Systems by Construction, p. 22.
94/30	J.C.M. Baeten J.A. Bergstra Gh. Ştefanescu	Process Algebra with Feedback, p. 22.
94/31	B.W. Watson R.E. Watson	A Boyer-Moore type algorithm for regular expression pattern matching, p. 22.
94/32	J.J. Vereijken	Fischer's Protocol in Timed Process Algebra, p. 38.
94/33	T. Laan	A formalization of the Ramified Type Theory, p.40.
94/34	R. Bloo F. Kamareddine R. Nederpelt	The Barendregt Cube with Definitions and Generalised Reduction, p. 37.
94/35	J.C.M. Baeten S. Mauw	Delayed choice: an operator for joining Message Sequence Charts, p. 15.
94/36	F. Kamareddine R. Nederpelt	Canonical typing and Π-conversion in the Barendregt Cube, p. 19.

94/37	T. Basten R. Bol M. Voorhoeve	Simulating and Analyzing Railway Interlockings in ExSpect, p. 30.
94/38	A. Bijlsma C.S. Scholten	Point-free substitution, p. 10.
94/39	A. Blokhuis T. Kloks	On the equivalence covering number of splitgraphs, p. 4.
94/40	D. Alstein	Distributed Consensus and Hard Real-Time Systems, p. 34.
94/41	T. Kloks D. Kratsch	Computing a perfect edge without vertex elimination ordering of a chordal bipartite graph, p. 6.
94/42	J. Engelfriet	Concatenation of Graphs, p. 7.
94/43	R.C. Backhouse Bijsterveld	Category Theory as Coherently Constructive Lattice M . Theory: An Illustration, p. 35.