Invited Address I

Cache Consistency by Design

EXTENDED ABSTRACT

Ed Brinksma *

Tele-Informatics and Open Systems Group,
University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands
brinksma@cs.utwente.nl

Abstract. In this paper we present a proof outline of the sequential consistency the lazy caching protocol of Afek, Brown, and Merritt. We 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. What we present, in fact, is a rationalized design of the distributed caching memory. We will use a simple process-algebraic formalism for the specification of the various design stages. We will not follow a strictly algebraic exposition, however. At some points the correctness will follow using direct semantic arguments, and we will also employ higher-order constructs like *action transducers* to relate behaviours. The distribution of the design/proof over five transformation steps provides a good insight into the variations that could have been allowed at each point of the design while still maintaining sequential consistency. The design/proof in fact establishes the correctness of a whole family of related memory architectures. The factorization in smaller steps also allows for a closer analysis of the fairness assumptions about the distributed memory.

1 Introduction

In this paper we present a proof outline for the sequential consistency the lazy caching protocol of [ABM93] as formulated in [Ger94]. A detailed presentation of this proof can be found in [Bri94]. We 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 (outline), in fact, presents a rationalized design of the distributed caching memory.

We will use 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. We 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.

* This work has been supported by the EU as part of ESPRIT BRA project 6021 *Building Correct Reactive Systems* (REACT).

S. T. Vuong et al. (eds.), *Protocol Specification, Testing and Verification XIV*

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 1 and 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. 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

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

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 1. In the application part of the paper we will provide concrete instances of the set of actions Act en the process definition environment.

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	$B_1 \|\|_G B_2$ $(G \subseteq Act)$	$L(B_1) \cup L(B_2)$
hiding	B/G $(G \subseteq Act)$	$L(B) - G$
renaming	$B[H]$ $(H : Act \to Act)$	$H(L(B))$
instantiation	p $(p \Leftarrow B_p,\ L(B_p) \subseteq L_p)$	L_p

Table 1. syntax of a simple process algebraic language

In addition to the process algebraic combinators introduced by table 1 we will use generalizations 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

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

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 $\prod_{i \in \mathcal{I}}^G B_i$.

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.

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. structured operational semantics

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

Definition 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 $\mu \in Act \cup \{\tau\}$ $\exists B_1'$ $B_1 \xrightarrow{\mu} B_1'$ 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, written $B_1 \sim B_2$, 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 1. *The relation $\sim$ is a congruence with respect to all the combinators introduced in table 1 and satisfies the laws listed in table 3.* □

(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 3. Some transformation laws

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

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

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

Definition 2. 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 $a \in Act \cup \{\epsilon\}$ $\exists B_1' \; B_1 \stackrel{a}{\Rightarrow} B_1'$ implies $\exists B_2' \; B_2 \stackrel{a}{\Rightarrow} B_2'$ 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, written $B_1 \approx B_2$, 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. *The relation $\approx$ is a congruence with respect to all the combinators introduced in table 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 3.
notation Two behaviour expressions B_1, B_2 are *trace equivalent*, written $B_1 \approx_{trace} B_2$, iff $Traces(B_1) = Traces(B_2)$. □

Fact 3. *The relation $\approx_{trace}$ is a congruence with respect to all the combinators introduced in table 1 and $\sim \subseteq \approx \subseteq \approx_{trace}$.* □

Fact 4. *Let $B_1 ||_* B_2$ be defined as in Table 3.*

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

□

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), contexts define *action transducers* with *transductions* between them. Such transductions will be denoted by doubly decorated arrows, as in

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

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 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.

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

□

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

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

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

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 4. 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 5. 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 6. 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

$$\forall \rho, \sigma \in Act^*, \upsilon \in Q^* \;\; T^\varepsilon \xRightarrow[\sigma]{\rho} T^\upsilon \text{ implies } \rho\lceil A = \sigma\upsilon\lceil A$$

□

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 7. *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. See [Bri94].

Lemma 8 (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 $T^\epsilon[B] \stackrel{\sigma}{\Rightarrow} T^\upsilon[B']$. Because B continuously allows all actions in Q, we have in particular that $B' \stackrel{\upsilon}{\Rightarrow} B''$ and therefore $T^\upsilon[B'] \stackrel{\varsigma}{\Rightarrow} T^\epsilon[B'']$. 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. □

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 (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.

$$Mem_{ser}(\overline{x}) \Leftarrow \sum_{\substack{i \in I \\ a \in A, d \in D}} W_i(d,a).Mem_{ser}(\overline{x}\{d/x_a\}) + \sum_{\substack{i \in I \\ a \in A}} R_i(x_a,a).Mem_{ser}(\overline{x}) \tag{1}$$

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 9. 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 $\forall \sigma \in Traces(B_1)$ $\exists \sigma' \in Traces(B_2)$ 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 5.

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) 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 $\mathcal{R}_j$.

$$Locmem_j(\overline{x}) \Leftarrow \sum_{\substack{i \in I \\ a \in A, d \in D}} W_i(d,a).Locmem_j(\overline{x}\{d/x_a\}) + \sum_{a \in A} R_j(x_a,a).Locmem_j(\overline{x}) \tag{2}$$

Our first refinement is now given by the process definition $Refinement_1$ in (3).

$$Refinement_1 \Leftarrow \prod_{j \in I}^{\mathcal{W}} Locmem_j(\overline{0}) \tag{3}$$

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

Lemma 10. *$Mem_{ser}(\overline{0}) \sim Refinement_1$*

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. □

Corollary 11. *$Refinement_1$ is weak sequential consistent with $Mem_{ser}(\overline{0})$*

Proof. Follows directly from $\sim \subseteq \approx_{trace}$ (fact 3). □

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 (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 $\mathcal{U} =_{df} \bigcup_{i \in I} \mathcal{U}_i$.

$$\begin{aligned} Cache_j(\overline{x}) \Leftarrow & \sum_{\substack{i \in I \\ a \in A, d \in D}} W_i(d,a).Cache_j(\overline{x}\{d/x_a\}) \\ & + \sum_{a \in A, d \in D} U_j(d,a).Cache_j(\overline{x}\{d/x_a\}) \\ & + \sum_{a \downarrow \overline{x}} R_j(x_a,a).Cache_j(\overline{x}) \\ & + \sum_{\overline{y} \in r(\overline{x})} \tau.Cache_j(\overline{y}) \end{aligned} \tag{4}$$

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 $\mathcal{U}/\mathcal{R} : Act \to Act$ 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.

$$Refinement_2 \Leftarrow \prod_{j\in I}^{\mathcal{W}} (Locmem_j(\overline{0})[\mathcal{U}/\mathcal{R}] \;||_{\mathcal{U}_j\cup\mathcal{W}}\; Cache_j(\overline{y}_{j0}))/\mathcal{U} \tag{5}$$

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

The correctness of this step follows from the following lemma.

Lemma 12. *$\forall \overline{x} \in D^A, \overline{y} \in r(\overline{x}), j \in I (Locmem_j(\overline{x})[\mathcal{U}/\mathcal{R}] \;||_{\mathcal{U}_j\cup\mathcal{W}}\; Cache_j(\overline{y}))/\mathcal{U} \;\approx\; Locmem_j(\overline{x})$*

Proof. The relation $\{\langle (Locmem_j(\overline{x})[\mathcal{U}/\mathcal{R}] \;||_{\mathcal{U}_j\cup\mathcal{W}}\; Cache_j(\overline{y}))/\mathcal{U}, Locmem_j(\overline{x})\rangle \mid \overline{x} \in D^A, \overline{y} \in r(\overline{x})\}$ is a weak bisimulation relation. □

Corollary 13. *$Refinement_2$ 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) it follows from that $Refinement_2 \approx Refinement_1$. Combining this with $\approx \subseteq \approx_{trace}$ (fact 3) and corollary 11 the desired result now follows directly. □

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 3.

Definition 14. 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.

$$Refinement_3 \Leftarrow \prod_{j\in I}^{\mathcal{W}} (Locmem_j(\overline{0})[\mathcal{U}/\mathcal{R}] \;||_{\mathcal{U}_j\cup\mathcal{W}}\; K_j^\epsilon[Cache_j(\overline{y}_{j0})])/\mathcal{U} \tag{6}$$

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

We can now prove the following lemma.

Lemma 15.

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

Proof. This essentially follows from the preservation lemma 8. See [Bri94].

Corollary 16. *$Refinement_3$ is weak sequential consistent with $Mem_{ser}(\overline{0})$*

Proof. Assume that

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

then according to fact 4 for each $j \in I$ with $\sigma_j = \sigma\lceil(\mathcal{W}\cup\mathcal{R}_j)$ we have

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

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 4 again, in the opposite direction, combining the σ'_j and find a σ' with $\sigma'\lceil(\mathcal{W}\cup\mathcal{R}_j) = \sigma'_j\lceil(\mathcal{W}\cup\mathcal{R}_j)$

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

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 [Ger94]. 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 17. 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.

$$\begin{aligned} Cache'_j(\overline{x}) \Leftarrow & \sum_{a\in A, d\in D} U_j(d,a).Cache'_j(\overline{x}\{d/x_a\}) \\ & + \sum_{a\downarrow\overline{x}} R_j(x_a,a).Cache'_j(\overline{x}) \\ & + \sum_{\overline{y}\in r(\overline{x})} \tau.Cache'_j(\overline{y}) \end{aligned} \tag{7}$$

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

$$Refinement_{3'} \Leftarrow \prod_{j\in I}^{\mathcal{W}} (Locmem_j(\overline{0})[\mathcal{U}/\mathcal{R}] \;||_{\mathcal{U}_j\cup\mathcal{W}}\; L_j^{\epsilon}[Cache'_j(\overline{y}_{j0})])/\mathcal{U} \tag{8}$$

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

Proof. Left to the reader. □

Corollary 19. *$Refinement_{3'}$ is weak sequential consistent with $Mem_{ser}(\overline{0})$*

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

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.

$$Refinement_4 \Leftarrow (Mem_{ser}(\overline{0})[\mathcal{U}/\mathcal{R}] \;||_{\mathcal{U}\cup\mathcal{W}} \prod_{j\in I}^{\mathcal{W}} L_j^{\epsilon}[Cache'_j(\overline{y}_{j0})])/\mathcal{U} \tag{9}$$

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

Lemma 20.

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

Proof.

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

$\sim$ {*law 4 of table 3*}

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

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

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

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

$$(\prod^*_{j\in I} Locmem_j(\overline{0})[\mathcal{U}/\mathcal{R}] \;||_*\; \prod^*_{j\in I} L^\epsilon_j[Cache'_j(\overline{y}_{j0})])/\mathcal{U}$$

$\sim$ {*law 5 of table 3 and lemma 10*}

$$(Mem_{ser}(\overline{0})[\mathcal{U}/\mathcal{R}] \;||_*\; \prod^*_{j\in I} L^\epsilon_j[Cache'_j(\overline{y}_{j0})])/\mathcal{U}$$

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

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

□

Corollary 21. *$Refinement_4$ is weak sequential consistent with $Mem_{ser}(\overline{0})$*

Proof. As $\sim$ preserves traces. □

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

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

□

The corresponding refinement is expressed by process definition *$Refinement_5$* below (recall that in the beginning of this section we put $I = \{1, \ldots, n\}$).

$$Refinement_5 \Leftarrow \tag{10}$$
$$(M^\epsilon_1 \circ \ldots \circ M^\epsilon_n)[(Mem_{ser}(\overline{0})[\mathcal{U}/\mathcal{R}] \;||_{\mathcal{U}\cup\mathcal{W}} \prod_{j\in I}{}^{\mathcal{W}} L^\epsilon_j[Cache'_j(\overline{y}_{j0})])/\mathcal{U}]$$

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

Theorem 23. *For all $i \in I (M^\epsilon_1 \circ \ldots \circ M^\epsilon_i)[(Mem_{ser}(\overline{0})[\mathcal{U}/\mathcal{R}] \;||_{\mathcal{U}\cup\mathcal{W}} \prod_{j\in I}{}^{\mathcal{W}} L^\epsilon_j[Cache'_j(\overline{y}_{j0})])/\mathcal{U}]$ is weak sequential consistent with $Mem_{ser}(\overline{0})$.*

Proof. By induction on i using preservation lemma 8 it is straightforward to show that the application of each M^ϵ_i 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 21. □

Corollary 24. *$(M^\epsilon_1 \circ \ldots \circ M^\epsilon_n)[(Mem_{ser}(\overline{0})[\mathcal{U}/\mathcal{R}] \;||_{\mathcal{U}\cup\mathcal{W}} \prod_{j\in I}{}^{\mathcal{W}} L^\epsilon_j[Cache'_j(\overline{y}_{j0})])/\mathcal{U}]$ is weak sequential consistent with $Mem_{ser}(\overline{0})$.*

Proof. Take $i = n$. □

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 [Ger94], 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 3. 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

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

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 25 (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

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

□

To show the correctness of the distributed caching memory it suffices to extend some of the definitions and facts of section 2. We start with the equivalence corresponding to $Traces_\omega(B)$ defined by

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

Fact 5. *The relation $\approx_{trace_\omega}$ is a congruence with respect to all the combinators introduced in table 1 and $\approx \subseteq \approx_{trace_\omega} \subseteq \approx_{trace}$.* □

Fact 6. *Let $B_1||_* B_2$ be defined as in Table 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 8 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 [Ger94]: that no read, write, or update action is continuously enabled but never executed.

Lemma 26 (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 $q \in Q$ $\exists B''\ B' \stackrel{q}{\rightarrow} B''$. 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. See [Bri94]

Theorem 27.

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

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 11), 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 13), which can now be replaced by the argument that $\approx \subseteq \approx_{trace_\omega}$.
3. *buffering cache communication*: an infinite trace version of lemma 15 can be proved using fact 6 instead of fact 4, and the extended preservation lemma 26, which leads to the strong version of corollary 16. 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. □

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 [Ger94]. 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 26; the last is implicit in the proof of weak bisimulation equivalence in lemma 12. 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 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 15 concerning the cache buffer, and twice in the proof of theorem 23 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 10. 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

$$p(x) \quad \Leftarrow \quad \sum_{a \in Var} f(a,x).p(g(a,x)) + \sum_{a \in Inv} h(a,x).p(x) \tag{11}$$

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

Theorem 28. *Let $p(x)$ of the form defined by (11) above. Let $\mathcal{F}$ be a finite partitioning of Inv and define for all $F \in \mathcal{F}$*

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

Then

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

□

Sofar, we have not succeeded in formulating a suitably general formulation of the transformation principle behind the introduction of the local caches in lemma 12. 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.

References

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

[BB87] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.

[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.

[Bri94] Ed Brinksma. Cache consistency by design. In: Deliverable WP3 of ESPRIT BRA REACT (project 6021), June 1994. Submitted for publication.

[BW90] J.C.M. Baeten and W.P. Weijland. *Process algebra.* Cambridge University Press, 1990.

[Ger94] Rob Gerth. Introduction to sequential consistency and the lazy caching algorithm. In: Deliverable WP3 of ESPRIT BRA REACT (project 6021), June 1994. Submitted for publication.

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

[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.

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

[Plo81] G.D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[vG93] Rob J. van Glabbeek. The linear time - branching time spectrum ii. LectureNotes in Computer Science 715, pages 66 – 81. Springer-Verlag, 1993.