Distrib. Comput. (1999) 12: 61–74

DISTRIBUTED COMPUTING

Cache consistency by design[⋆]

Ed Brinksma

Formal Methods and Tools Group, University of Twente, P.O. Box 217, NL-7500 AE Enschede, The Netherlands (e-mail: brinksma@cs.utwente.nl)

May 27, 1999

Summary. In this paper we present a proof of the sequential consistency of the lazy caching protocol of Afek, Brown, and Merritt. 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. The 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. We will not follow a strictly algebraic exposition, however. At some points the correctness will be shown 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.

Key words: Formal design – Caching protocols – Reactive systems – Process algebra – Correctness preserving – Transformations

1 Introduction

In this paper we present a proof of the sequential consistency of the lazy caching protocol of [ABM93] as formulated by Gerth in this issue. 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 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 outline of our refinement proof is as follows. It starts from a specification of a *serial memory* with n user interfaces, where each user can perform direct atomic read and write actions on the memory (see Fig. 4). This is a convenient starting point for our 'design', as the serial memory defines our correctness criterion: the distributed caching memory must be *sequentially consistent* with the serial memory. This means that the behaviour at each user interface of the distributed caching memory coincides with the projection on the same interface of a legal behaviour of the serial memory. In other words, if the user at one interface of the distributed caching memory cannot compare the order of his actions with that of the actions occurring at the other interfaces, he cannot distinguish his interface from that of a serial memory. The idea now is to transform a serial memory into other memory architectures step by step, where each new architecture is sequentially consistent if the previous one is. As the serial memory is trivially sequentially consistent, this implies that the final distributed caching memory, as well as all intermediate architectures are sequentially consistent. The transformation steps in the proof are all closely linked to the ingredients of the lazy caching architecture, which is depicted in Fig. 1.

First of all, we see that each user interface has a corresponding *caching memory* into which it can write, via two buffers, and from which it can read directly, but subject to some conditions. *Transformation step 1* will approximate this situation very crudely. Instead of caches it equips each user interface generously with a copy of the serial memory

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

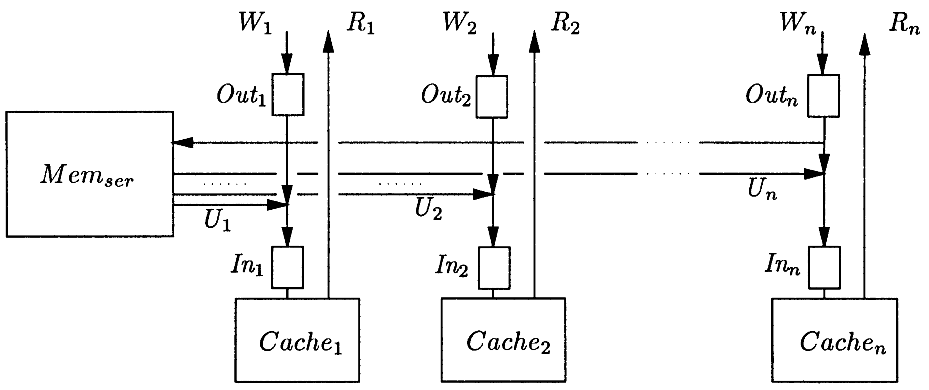

Fig. 1. The lazy caching memory

itself, where each user directly reads from its local memory, but each write action is atomically copied to *all* local memories (see Fig. 5). The latter ensures the global consistency of the local memories, and thus the sequential consistency.

Transformation step 2 introduces the local caches, but without removing the local memories (see Fig. 6). Each user reads directly from its local cache. When an invalidated address is read this will succeed only after it has been updated by the local memory. Because of the general fairness assumption stated in [**?**] this will occur after a finite time. A write action to a local cache is copied atomically to all local memories and caches. The correctness of this step follows from the fact that under the fairness assumption the caches do not add to the behaviour of the distributed memories.

Transformation step 3 adds a second ingredient of the target architecture, viz. the *In*-queues (see Fig. 7). These queues buffer the write and local update messages for the caches. Also, the constraint is added that local read actions are only allowed if no *local* write actions are queued, i.e. write messages of users at other interfaces can be in the queue at the time of reading. The addition of such an *In*-buffer and a constraint to the use of a cache provides a particular operational *context* for the caches. The correctness of the transformation follows from a general property of contexts of this kind: they preserve the order of any set of messages that may be queued *together with* those actions that can be executed when *no* messages of the given set are queued. We show this property to be sufficient for the preservation of sequential consistency.

Transformation step 4 combines the coherent local memories into one, central serial background memory that updates all local caches (see Fig. 8). This transformation exploits the associativity of the parallel compositions in the system, which allows us to see it as subsystem of coherent local memories in parallel with a subsystem of buffered local caches. The inverse of transformation step 1 in turn allows us to replace the former subsystem by a single serial memory.

Finally, *transformation step 5* adds the last ingredient to obtain the desired architecture, viz. the *Out*-queues. These queues buffer the local write messages, which upon leaving the queue are written atomically into the serial background memory and all *In*-queues. Local reads can occur only if the corresponding local *Out*-queue is empty. The queues and constraints are added separately for each user interface (see Fig. 9), each providing a context similar to those used in transformation step 3. Again, sequential consistency is guaranteed by the general preservation properties of such contexts.

The structure of the remainder of this article is as follows.

- *Section 2* introduces the process-algebraic formalism that we use;
- *Section 3* explains about *action transducers*, and in particular the new concept of a *queue-like* action transducer, which is used to represent the contexts of *In*- and *Out*-queues mentioned in the proof outline above;
- *Section 4* gives the formal transformation style proof of the *weak* sequential consistency of the distributed cache memory along the lines of the proof outline above. This proof takes only finite sequences of the observable actions of a system into account;
- *Section 5* improves the result to *strong* sequential consistency, which also deals with infinite behaviour;
- *Section 6* summarizes the results that have been obtained and discusses possible generalizations and extensions.

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* listed in the syntax table define the behaviour of systems in terms of labelled transition systems, where the transitions are labelled 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].

We draw attention to the fact that the parallel composition combinator $||_G$ allows for so-called *multiway synchro-*

Table 1. Syntax of a simple process algebraic language

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	$B[H]$	$H(L(B))$
	($H : Act \to Act$)	
instantiation	p	L_p
	($p \Leftarrow B_p$, $L(B_p) \subseteq L_p$)	

Table 2. Structured operational semantics

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	$B_p \xrightarrow{\mu} B' \vdash_{p \Leftarrow B_p} p \xrightarrow{\mu} B'$

nization between actions, i.e. the $||_G$-composition of any finite number of processes can synchronize on a given action $a \in G$. To make an action unavailable for further synchronization an additional *hiding*-combinator is part of our formalism. The combination of these two combinators allows for a powerful specification style that is essential for our proof of the caching protocol. A similar set of combinators is available in CSP and LOTOS [BB87]. ACP also possesses a hiding- or *abstraction*-combinator, and $||_G$-composition can be handled as a derived construct. In CCS synchronization is always combined with hiding, thus effectively allowing only binary synchronizations, which makes it unfit for our proof strategy.

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

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

where $\mathscr{P}$ 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* and the process definition environment.

In addition to the process algebraic combinators introduced by Table 1 we will use generalizations for the choice and composition operators. If $\mathscr{B}$ denotes a *finite* set of behaviour expressions then $\sum \mathscr{B}$ and $\prod^G \mathscr{B}$ denote the repeated application of '+' and '$||_G$', respectively, to the elements of $\mathscr{B}$. E.g. if $\mathscr{B} = \{B_1, \ldots, B_n\}$ then

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

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

The standard identity over the behaviour expressions (and labelled 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 $\mathscr{P}$ 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 $\mu \in Act \cup \{\tau\}$ it is the case that $\exists B_1{}'\ B_1 \xrightarrow{\mu} B_1{}'$ 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 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.* □

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

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

The $\Rightarrow$-notation is used for a generalized version of the transition relation that concentrates on *observable* behaviour. Note that $B \overset{\epsilon}{\Longrightarrow} B'$ expresses that B can change into B' *unobservedly*, i.e. by executing 0 or more τ-transitions. This includes the special case that no transitions are excuted at all (and therefore $B = B'$). $B \overset{a}{\Longrightarrow} B'$ expresses that B may move to B' when executing the observable action a, possibly preceded or followed by any finite number of invisible τ-steps. $\xrightarrow{\sigma}$ and $\overset{\sigma}{\Longrightarrow}$ are the generalizations for strings

Table 3. Some transformation laws

(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 $H \restriction G = id_G$ and $H^{-1}(G) = G$

of $\xrightarrow{a}$ and $\overset{a}{\Longrightarrow}$, respectively. *Der*($B$) denotes the set of behaviours that are *reachable* from B, also known as its *derivatives*. These are all behaviours that can be reached from B by some finite string of transitions.

Using the above notation we define also a less strict equivalence 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 $\alpha \in Act \cup \{\epsilon\}$ it is the case that $\exists B_1'$ $B_1 \overset{\alpha}{\Longrightarrow} B_1'$ implies $\exists B_2'$ $B_2 \overset{\alpha}{\Longrightarrow} B_2'$ 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 *The relation $\approx$ is a congruence with respect to all the combinators introduced in Table 1 except for the choice combinator +, and its generalization $\sum$. Moreover, $\sim \subseteq \approx$, i.e. $\approx$ satisfies all laws of $\sim$.* □

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

Definition 3 *Two behaviour expressions B_1, B_2 are* trace equivalent, *notation $B_1 \approx_{trace} B_2$, 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.*

$$\begin{aligned}&\mathit{Traces}(B_1||_*B_2) = \\ &\quad \{\sigma \in (L(B_1) \cup L(B_2))^* \mid \sigma \restriction L(B_1) \in \mathit{Traces}(B_1), \\ &\qquad\qquad \sigma \restriction L(B_2) \in \mathit{Traces}(B_2)\}\end{aligned}$$

□

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 $C[b.c.\mathbf{0}] = a.\mathbf{0}+b.c.\mathbf{0}$.

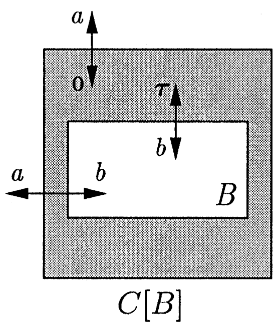

Fig. 2. Transductions of a context

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

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

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 b-action transforming into B', $T[B]$ will produce a a-action as its result and transform into $T'[B']$.

Example 1

$$a.B||_{\{a\}}[\ ][a/b] \underset{b}{\overset{a}{\rightarrow}} B||_{\{a\}}[\ ][a/b]$$

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

The transduction $T \underset{b}{\overset{a}{\rightarrow}} T'$ 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 $T \underset{0}{\overset{a}{\rightarrow}} T'$, corresponding to the operational semantic rule

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

Example 2

$$a.B||_{\{a\}}a.[\ ] \underset{0}{\overset{a}{\rightarrow}} B||_{\{a\}}[\ ]$$

□

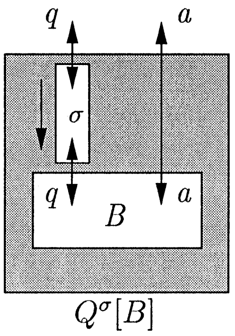

Fig. 3. A queue-like transducer

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

$$T \underset{b_1 \ldots b_n}{\xrightarrow{a_1 \ldots a_n}} T' \Leftrightarrow_{df} \exists T_0, \ldots, T_n \; T \equiv T_0 \underset{b_1}{\overset{a_1}{\rightarrow}} T_1 \wedge \ldots$$
$$\wedge\, T_{n-1} \underset{b_n}{\overset{a_n}{\rightarrow}} T_n \equiv T'$$
$$T \underset{\sigma_2}{\overset{\sigma_1}{\Longrightarrow}} T' \quad \Leftrightarrow_{df} \exists {\sigma_1}', {\sigma_2}' \; T \underset{{\sigma_2}'}{\overset{{\sigma_1}'}{\rightarrow}} T' \;\wedge\; \sigma_1 \lhd {\sigma_1}'$$
$$\wedge\, \sigma_2 \lhd {\sigma_2}'$$

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 $\mathscr{T}_Q = \{T^\sigma \mid \sigma \in Q^*\}$ is* queue-like *iff its transductions are of the form:*

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

These transducers correspond to the contexts depicted in Fig. 3. There are three sorts of transductions possible, corresponding to the double-headed arrows in the figure and the rules in the definition. There is a designated subset Q of *Act* representing the *queueable* actions. Actions of Q may be stored by the environment into the context queue (transduction rule 1). The contents of the queue is represented by a string in Q^* used as a superscript of T. Actions of Q that are at the head of the context queue may be consumed by the instantiating behaviour (transduction rule 2). This action is invisible (τ) to the environment. Finally, there may be states of the transducer (characterized by the contents σ of the queue) in which some non-queuable actions a of the environment coincide with actions a of the instantiating behaviour (transduction rule 3). Note that the first two rules require (de)queuing transductions to exist for all queuable actions in all context states, whereas the last rule *allows* the existence of certain transductions in certain context states.

For the proofs in our derivation of the lazy caching memory we will be especially interested in queue-like action transducers because of the *observable trace transductions* that they induce. More in particular we will need to know those traces that are invariant under transduction. This motivates the following definition.

Definition 5 *Let $\mathscr{T}_Q = \{T^\sigma \mid \sigma \in Q^*\}$ be a queue-like family of action-transducers. We say that $\mathscr{T}_Q$ preserves $A \subseteq Act$ iff*

$$\forall \rho, \sigma \in Act^*, \upsilon \in Q^* \;\; T^\epsilon \underset{\sigma}{\overset{\rho}{\Longrightarrow}} T^\upsilon \;\; \textit{implies} \;\; \rho \restriction A = \sigma\upsilon \restriction A$$
□

It is not difficult to see that traces of queuable actions are preserved in the above sense by queue-like transducers, as (FIFO) queues preserve order. Strings of non-queuable actions are also preserved, as their execution by the context coincides with their execution by the instantiating behaviour. In order to study the preservation properties of strings of both queuable and non-queuable actions is it useful to consider the sets D_A of those non-queuable actions that can occur if the queue does *not* contain any of the actions in a given set $A \subseteq Q$.

Definition 6 *Let $\mathscr{T}_Q = \{T^\sigma \mid \sigma \in Q^*\}$ be a queue-like family of action-transducers. For each $A \subseteq Q$ we define the set $D_A \subseteq Act$ by*

$$D_A = \{a \in Act \mid \forall \sigma \; (T^\sigma \underset{a}{\overset{a}{\rightarrow}} T^\sigma \; \textit{iff} \; \sigma \restriction A = \epsilon)\}$$
□

The following lemma expresses the general preservation properties of queue-like transducers. They state that strings over A can always be mixed with actions in D_A without losing the preservation property. The intuition behind this result is that actions in D_A could never 'overtake' nor be 'overtaken' by actions in A and thus upset the ordering.

Lemma 7 *Let $\mathscr{T}_Q = \{T^\sigma \mid \sigma \in Q^*\}$ be a queue-like family of action-transducers. For each $A \subseteq Q$ $\;\mathscr{T}_Q$ preserves $A \cup D_A$.*

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

Let us therefore suppose that the lemma holds for all $n < |\rho| + |\sigma|$. We can factorize $T^\epsilon \underset{\sigma}{\overset{\rho}{\Longrightarrow}} T^\upsilon$ into $T^\epsilon \underset{\sigma_1}{\overset{\rho_1}{\Longrightarrow}} T^{\upsilon_1} \underset{b}{\overset{a}{\rightarrow}} T^\upsilon$ 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 $\rho_1 \restriction (A \cup D_A) = \sigma_1\upsilon_1 \restriction (A \cup D_A)$.

We now proceed by case analysis on the nature of the transduction $T^{\upsilon_1} \underset{b}{\overset{a}{\rightarrow}} T^\upsilon$ as given in Definition 4.

1. $T^{\upsilon_1} \underset{b}{\overset{a}{\rightarrow}} T^\upsilon = T^{\upsilon_1} \underset{0}{\overset{q}{\rightarrow}} T^{\upsilon_1 q}$.
   Then $\rho \restriction (A \cup D_A) = \rho_1 q \restriction (A \cup D_A)$

$= \sigma_1 v_1 q \upharpoonright (A \cup D_A)$
$= \sigma v \upharpoonright (A \cup D_A)$.

2. $T^{v_1} \underset{b}{\overset{a}{\rightarrow}} T^v = T^{qv} \underset{q}{\overset{\tau}{\rightarrow}} T^v$.
Then $\rho \upharpoonright (A \cup D_A) = \rho_1 \upharpoonright (A \cup D_A) = \sigma_1 v_1 \upharpoonright (A \cup D_A) = \sigma_1 q v \upharpoonright (A \cup D_A) = \sigma v \upharpoonright (A \cup D_A)$.
3. $T^{v_1} \underset{b}{\overset{a}{\rightarrow}} T^v = T^v \underset{a}{\overset{a}{\rightarrow}} T^v$.
This is only possible if $a \notin Q$ and thus $a \notin A$.
Assume that also $a \notin D_A$ then it follows that
$\rho \upharpoonright (A \cup D_A) = \rho_1 a \upharpoonright (A \cup D_A) = \sigma_1 v_1 a \upharpoonright (A \cup D_A) = \sigma_1 a v_1 \upharpoonright (A \cup D_A) = \sigma v \upharpoonright (A \cup D_A)$.
In the other case that $a \in D_A$ it follows that $v_1 \upharpoonright A = v \upharpoonright A = \epsilon$. Therefore, we get
$\rho \upharpoonright (A \cup D_A) = \rho_1 a \upharpoonright (A \cup D_A) = \sigma_1 v_1 a \upharpoonright (A \cup D_A) = \sigma_1 a \upharpoonright (A \cup D_A) = \sigma \upharpoonright (A \cup D_A) = \sigma v \upharpoonright (A \cup D_A)$. □

The following lemma casts the preservation property in the form that we will need in our proofs later.

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 $\sigma \in Traces(T^\epsilon[B])$ we have*

$$\exists \sigma' \in Traces(B)\ \ \forall A \subseteq Q$$
$$\textit{with}\ \ \sigma \upharpoonright (A \cup D_A) = \sigma' \upharpoonright (A \cup D_A)$$

Proof. Assume that $T^\epsilon[B] \overset{\sigma}{\Longrightarrow} T^v[B']$. Because B continuously allows all actions in Q, we have in particular that $B' \overset{v}{\Longrightarrow} B''$ and therefore $T^v[B'] \overset{\epsilon}{\Longrightarrow} T^\epsilon[B'']$. It follows that there exists a σ' with $T^\epsilon \underset{\sigma'}{\overset{\sigma}{\Longrightarrow}} T^\epsilon$ 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.

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

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 also be useful to define the sets

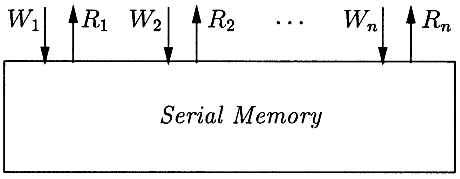

Fig. 4. A serial memory with n user interfaces

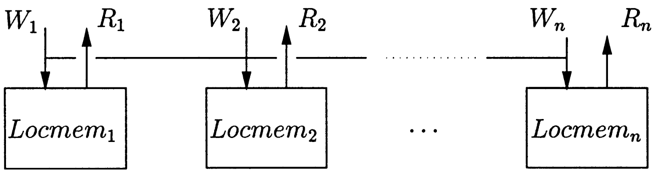

Fig. 5. A distributed memory

- $\mathcal{W}_i =_{df} \{W_i(d,a) \mid d \in D, a \in A\}$, the set of write actions at user interface i, and $\mathcal{W} =_{df} \bigcup_{i \in I} \mathcal{W}_i$, the set of all write actions,
- $\mathcal{R}_i =_{df} \{R_i(d,a) \mid d \in D, a \in A\}$, the set of read actions at user interface i, and $\mathcal{R} =_{df} \bigcup_{i \in I} \mathcal{R}_i$, the set of all read actions,
- $\mathcal{L}_i =_{df} \mathcal{W}_i \cup \mathcal{R}_i$, the set of read and write actions at user interface i, and $\mathcal{L} =_{df} \bigcup_{i \in I} \mathcal{L}_i$, the set of all read and write actions.

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* weakly sequentially consistent *with B_2 iff*

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

□

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 $\mathcal{R}_j$.

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

Our first refinement is now given by the process definition *Refinement*$_1$ in (3).

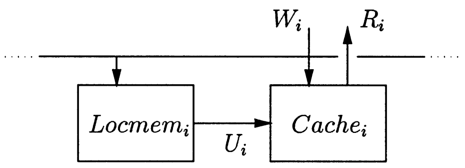

Fig. 6. Factoring out a local cache

$$\mathit{Refinement}_1 \Leftarrow \prod_{j\in I}^{\mathscr{W}} \mathit{Locmem}_j(\overline{0}) \tag{3}$$

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

Lemma 10

$$\mathit{Mem}_{ser}(\overline{0}) \sim \mathit{Refinement}_1$$

Proof. The relation defined by

$$\{\langle \mathit{Mem}_{ser}(\overline{x}), \prod_{j\in I}^{\mathscr{W}} \mathit{Locmem}_j(\overline{x})\rangle \mid \overline{x} \in D^A\}$$

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

$$\begin{aligned}
& \mathit{Mem}_{ser}(\overline{x}) \xrightarrow{W_i(d,a)} \mathit{Mem}_{ser}(\overline{x}\{d/x_a\}) \\
\Leftrightarrow\ & \forall j \in I\ \mathit{Locmem}_j(\overline{x}) \xrightarrow{W_i(d,a)} \mathit{Locmem}_j(\overline{x}\{d/x_a\}) \\
\Leftrightarrow\ & \prod_{j\in I}^{\mathscr{W}} \mathit{Locmem}_j(\overline{x}) \xrightarrow{W_i(d,a)} \prod_{j\in I}^{\mathscr{W}} \mathit{Locmem}_j(\overline{x}\{d/x_a\})
\end{aligned}$$

and for all reading actions

$$\begin{aligned}
& \mathit{Mem}_{ser}(\overline{x}) \xrightarrow{R_i(x_a,a)} \mathit{Mem}_{ser}(\overline{x}) \\
\Leftrightarrow\ & \mathit{Locmem}_i(\overline{x}) \xrightarrow{R_i(x_a,a)} \mathit{Locmem}_i(\overline{x}) \\
\Leftrightarrow\ & \prod_{j\in I}^{\mathscr{W}} \mathit{Locmem}_j(\overline{x}) \xrightarrow{R_i(x_a,a)} \prod_{j\in I}^{\mathscr{W}} \mathit{Locmem}_j(\overline{x})
\end{aligned}$$

□

Corollary 11 *$\mathit{Refinement}_1$ is weakly sequentially consistent with $\mathit{Mem}_{ser}(\overline{0})$*

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 than the caches that we will ultimately design. The behaviour of the cache at interaction point $j \in I$ is given by the process definition $\mathit{Cache}_j(\overline{x})$ in (4) below. In addition to the (local) memory the caches have *update* actions $U_j(d,a)$. For convenience we define $\mathscr{U}_i =_{df} \{U_i(d,a) \mid d \in D, a \in A\}$ and $\mathscr{U} =_{df} \bigcup_{i\in I} \mathscr{U}_i$.

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

Note that the local caches synchronize on all actions in $\mathscr{W}$, but accept only local read and update actions, i.e. only actions in $\mathscr{R}_j \cup \mathscr{U}_j$. 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 $x_a \neq\uparrow$, denoting that $\overline{x}$ is defined at address a, and
- $r(\overline{x}) =_{df} \{\overline{y} \mid \forall a \in A\ y_a = x_a \vee y_a =\uparrow\}$, denoting the set of all memory vectors $\overline{y}$ that coincide with $\overline{x}$ at all their defined addresses, i.e. $\overline{y}$ is obtained by invalidating $\overline{x}$ at any number of its addresses.

Let $\mathscr{U}/\mathscr{R} : \mathit{Act} \to \mathit{Act}$ 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.

$$\begin{aligned}
\mathit{Refinement}_2 \Leftarrow \prod_{j\in I}^{\mathscr{W}} & (\mathit{Locmem}_j(\overline{0})[\mathscr{U}/\mathscr{R}] \\
& \|_{\mathscr{U}_j\cup\mathscr{W}}\ \mathit{Cache}_j(\overline{y}_{j0}))/\mathscr{U}
\end{aligned} \tag{5}$$

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

The correctness of this step follows from the following lemma.

Lemma 12 $\forall \overline{x} \in D^A, \overline{y} \in r(\overline{x}), j \in I$

$$\begin{aligned}
& (\mathit{Locmem}_j(\overline{x})[\mathscr{U}/\mathscr{R}]\ \|_{\mathscr{U}_j\cup\mathscr{W}}\ \mathit{Cache}_j(\overline{y}))/\mathscr{U} \\
& \approx \mathit{Locmem}_j(\overline{x})
\end{aligned}$$

Proof. The relation

$$\begin{aligned}
& \{\langle(\mathit{Locmem}_j(\overline{x})[\mathscr{U}/\mathscr{R}]\ \|_{\mathscr{U}_j\cup\mathscr{W}}\ \mathit{Cache}_j(\overline{y}))/\mathscr{U}, \\
& \mathit{Locmem}_j(\overline{x})\rangle \mid \overline{x} \in D^A, \overline{y} \in r(\overline{x})\}
\end{aligned}$$

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

- $(\mathit{Locmem}_j(\overline{x})[\mathscr{U}/\mathscr{R}]\ \|_{\mathscr{U}_j\cup\mathscr{W}}\ \mathit{Cache}_j(\overline{y}))/\mathscr{U} \overset{\epsilon}{\Longrightarrow} B$: Then $B{=}(\mathit{Locmem}_j(\overline{x})[\mathscr{U}/\mathscr{R}]\ \|_{\mathscr{U}_j\cup\mathscr{W}}\ \mathit{Cache}_j(\overline{y}'))/\mathscr{U}$ with $\overline{y}' \in r(\overline{x})$ where the silent transitions in $\overset{\epsilon}{\Longrightarrow}$ consist of zero or more cache invalidations and/or updates. It suffices to take $\mathit{Locmem}_j(\overline{x}) \overset{\epsilon}{\Longrightarrow} \mathit{Locmem}_j(\overline{x})$.
- $(\mathit{Locmem}_j(\overline{x})[\mathscr{U}/\mathscr{R}]\ \|_{\mathscr{U}_j\cup\mathscr{W}}\ \mathit{Cache}_j(\overline{y}))/\mathscr{U} \xrightarrow{W_i(d,a)} B$: Then $B = (\mathit{Locmem}_j(\overline{x}\{d/x_a\})[\mathscr{U}/\mathscr{R}]\ \|_{\mathscr{U}_j\cup\mathscr{W}} \mathit{Cache}_j(\overline{y}\{d/y_a\}))/\mathscr{U}$.

This is directly matched by $\mathit{Locmem}_j(\overline{x}) \xrightarrow{W_i(d,a)} \mathit{Locmem}_j(\overline{x}\{d/x_a\})$.

- $(\mathit{Locmem}_j(\overline{x})[\mathcal{U}/\mathcal{R}] \;||_{\mathcal{U}_j\cup\mathcal{W}}\; \mathit{Cache}_j(\overline{y}))/\mathcal{U} \xrightarrow{R_j(x_a,a)} B$: Then $B = (\mathit{Locmem}_j(\overline{x})[\mathcal{U}/\mathcal{R}] \;||_{\mathcal{U}_j\cup\mathcal{W}}\; \mathit{Cache}_j(\overline{y}))/\mathcal{U}$.
  This is directly matched by $\mathit{Locmem}_j(\overline{x}) \xrightarrow{R_j(x_a,a)} \mathit{Locmem}_j(\overline{x})$.
- $\mathit{Locmem}_j(\overline{x}) \overset{\epsilon}{\Longrightarrow} B$: Then $B = \mathit{Locmem}_j(\overline{x})$.
  This is therefore directly matched by
  $(\mathit{Locmem}_j(\overline{x})[\mathcal{U}/\mathcal{R}] \;||_{\mathcal{U}_j\cup\mathcal{W}}\; \mathit{Cache}_j(\overline{y}))/\mathcal{U} \overset{\epsilon}{\Longrightarrow} (\mathit{Locmem}_j(\overline{x})[\mathcal{U}/\mathcal{R}] \;||_{\mathcal{U}_j\cup\mathcal{W}}\; \mathit{Cache}_j(\overline{y}))/\mathcal{U}$.
- $\mathit{Locmem}_j(\overline{x}) \xrightarrow{W_i(d,a)} B$: Then $B = \mathit{Locmem}_j(\overline{x}\{d/x_a\})$.
  This is directly matched by
  $(\mathit{Locmem}_j(\overline{x})[\mathcal{U}/\mathcal{R}] \;||_{\mathcal{U}_j\cup\mathcal{W}}\; \mathit{Cache}_j(\overline{y}))/\mathcal{U} \xrightarrow{W_i(d,a)} (\mathit{Locmem}_j(\overline{x}\{d/x_a\})[\mathcal{U}/\mathcal{R}] \;||_{\mathcal{U}_j\cup\mathcal{W}}\; \mathit{Cache}_j(\overline{y}\{d/y_a\}))/\mathcal{U}$.
- $\mathit{Locmem}_j(\overline{x}) \xrightarrow{R_j(x_a,a)} B$: Then $B = \mathit{Locmem}_j(\overline{x})$.
  If $a \downarrow \overline{y}$ then this is directly matched by
  $(\mathit{Locmem}_j(\overline{x})[\mathcal{U}/\mathcal{R}] \;||_{\mathcal{U}_j\cup\mathcal{W}}\; \mathit{Cache}_j(\overline{y}))/\mathcal{U} \xrightarrow{R_j(x_a,a)} (\mathit{Locmem}_j(\overline{x})[\mathcal{U}/\mathcal{R}] \;||_{\mathcal{U}_j\cup\mathcal{W}}\; \mathit{Cache}_j(\overline{y}))/\mathcal{U}$.
  If $y_a = \uparrow$ then first a cache update of address a must take place.
  This generates the following matching sequence of actions:
  $(\mathit{Locmem}_j(\overline{x})[\mathcal{U}/\mathcal{R}] \;||_{\mathcal{U}_j\cup\mathcal{W}}\; \mathit{Cache}_j(\overline{y}))/\mathcal{U} \xrightarrow{\tau} (\mathit{Locmem}_j(\overline{x})[\mathcal{U}/\mathcal{R}] \;||_{\mathcal{U}_j\cup\mathcal{W}}\; \mathit{Cache}_j(\overline{y}\{x_a/y_a\}))/\mathcal{U} \xrightarrow{R_j(x_a,a)}$
  $(\mathit{Locmem}_j(\overline{x})[\mathcal{U}/\mathcal{R}] \;||_{\mathcal{U}_j\cup\mathcal{W}}\; \mathit{Cache}_j(\overline{y}\{x_a/y_a\}))/\mathcal{U}$ □

Corollary 13 *Refinement$_2$ is weakly sequentially 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 the above lemma that $\mathit{Refinement}_2 \approx \mathit{Refinement}_1$. Combining this with $\approx \subseteq \approx_{\mathit{trace}}$ (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:*

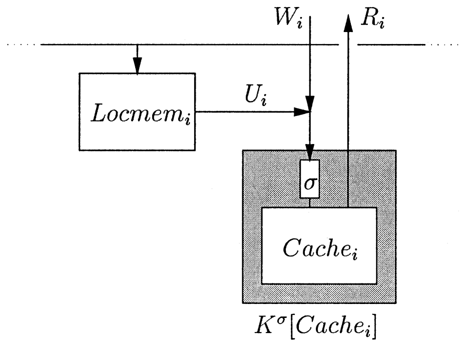

Fig. 7. Buffering a local cache

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

□

If we compare the above transductions to Definition 4 we see that the first two transductions correspond to queuing (case 1 of Definition 4) write and update messages, the next two transductions correspond to dequeuing (case 2) write and update messages, and the last transduction corresponds to direct execution (case 3) of read actions under a specific constraint on the contents of the queue.

The third refinement is reflected in the following process definition.

$$\mathit{Refinement}_3 \Leftarrow \prod_{j\in I}^{\mathcal{W}} (\mathit{Locmem}_j(\overline{0})[\mathcal{U}/\mathcal{R}] \;||_{\mathcal{U}_j\cup\mathcal{W}}\; K_j^\epsilon[\mathit{Cache}_j(\overline{y}_{j0})])/\mathcal{U} \qquad (6)$$

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

We can now prove the following lemma.

Lemma 15

$$\begin{aligned}
&\forall j \in I, \sigma \in (\mathcal{W} \cup \mathcal{R}_j)^*, \overline{x} \in D^A, \overline{y} \in r(\overline{x})\\
&(\mathit{Locmem}_j(\overline{x})[\mathcal{U}/\mathcal{R}] \;||_{\mathcal{U}_j\cup\mathcal{W}}\; K_j^\epsilon[\mathit{Cache}_j(\overline{y})])/\mathcal{U} \overset{\sigma}{\Longrightarrow}\\
&\exists \sigma' \in (\mathcal{W} \cup \mathcal{R}_j)^*\\
&(\mathit{Locmem}_j(\overline{x})[\mathcal{U}/\mathcal{R}] \;||_{\mathcal{U}_j\cup\mathcal{W}}\; \mathit{Cache}_j(\overline{y}))/\mathcal{U} \overset{\sigma'}{\Longrightarrow}\\
&\wedge\, \sigma \restriction (\mathcal{W}_j \cup \mathcal{R}_j) = \sigma' \restriction (\mathcal{W}_j \cup \mathcal{R}_j)\\
&\wedge\, \sigma \restriction \mathcal{W} = \sigma' \restriction \mathcal{W}
\end{aligned}$$

Proof. This essentially follows from the preservation Lemma 8. Assume that

$(\mathit{Locmem}_j(\overline{x})[\mathcal{U}/\mathcal{R}] \;||_{\mathcal{U}_j\cup\mathcal{W}}\; K_j^\epsilon[\mathit{Cache}_j(\overline{y})])/\mathcal{U} \overset{\sigma}{\Longrightarrow}$

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

$$Locmem_j(\overline{x})[\mathscr{U}/\mathscr{R}] \;||_{\mathscr{U}_j\cup\mathscr{W}}\; K^{\epsilon}_j[Cache_j(\overline{y})] \overset{\sigma_1}{\Longrightarrow}$$

By the properties of $||_{\mathscr{U}_j\cup\mathscr{W}}$ (fact 4) for $\sigma_2 = \sigma_1 \upharpoonright (\mathscr{U}_j \cup \mathscr{W})$ we have

$$Locmem_j(\overline{x})[\mathscr{U}/\mathscr{R}] \overset{\sigma_2}{\Longrightarrow} \quad \text{and } K^{\epsilon}_j[Cache_j(\overline{y})] \overset{\sigma_1}{\Longrightarrow}$$

By the preservation Lemma 8 there is a σ'_1 with $Cache_j(\overline{y}) \overset{\sigma'_1}{\Longrightarrow}$ and

$$\begin{aligned}\sigma'_1 \upharpoonright (\mathscr{W}_j \cup \mathscr{R}_j) &= \sigma_1 \upharpoonright (\mathscr{W}_j \cup \mathscr{R}_j) \quad \text{and}\\ \sigma'_1 \upharpoonright (\mathscr{W} \cup \mathscr{U}_j) &= \sigma_1 \upharpoonright (\mathscr{W} \cup \mathscr{U}_j)\end{aligned}$$

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

$$Locmem_j(\overline{x})[\mathscr{U}/\mathscr{R}] \;||_{\mathscr{U}_j\cup\mathscr{W}}\; Cache_j(\overline{y}) \overset{\sigma'_1}{\Longrightarrow}$$

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

$$(Locmem_j(\overline{x})[\mathscr{U}/\mathscr{R}] \;||_{\mathscr{U}_j\cup\mathscr{W}}\; K^{\epsilon}_j[Cache_j(\overline{y})])/\mathscr{U} \overset{\sigma'}{\Longrightarrow}$$

with

$$\begin{aligned}\sigma \upharpoonright (\mathscr{W}_j \cup \mathscr{R}_j) &= (\sigma_1/\mathscr{U}) \upharpoonright (\mathscr{W}_j \cup \mathscr{R}_j)\\ &= (\sigma_1 \upharpoonright (\mathscr{W}_j \cup \mathscr{R}_j))/\mathscr{U}\\ &= (\sigma'_1 \upharpoonright (\mathscr{W}_j \cup \mathscr{R}_j))/\mathscr{U}\\ &= (\sigma'_1/\mathscr{U}) \upharpoonright (\mathscr{W}_j \cup \mathscr{R}_j)\\ &= \sigma' \upharpoonright (\mathscr{W}_j \cup \mathscr{R}_j)\end{aligned}$$

and likewise

$$\begin{aligned}\sigma \upharpoonright \mathscr{W} &= (\sigma_1/\mathscr{U}) \upharpoonright \mathscr{W} = (\sigma_1 \upharpoonright \mathscr{W})/\mathscr{U}\\ &= (\sigma'_1 \upharpoonright \mathscr{W})/\mathscr{U} = (\sigma'_1/\mathscr{U}) \upharpoonright \mathscr{W} = \sigma' \upharpoonright \mathscr{W}\end{aligned}$$

□

Corollary 16 *Refinement$_3$ is weakly sequentially consistent with Mem$_{ser}(\overline{0})$*

Proof. Assume that

$$\prod_{j\in I}^{\mathscr{W}} (Locmem_j(\overline{0}) \;\; [\mathscr{U}/\mathscr{R}] \;||_{\mathscr{U}_j\cup\mathscr{W}} \; K^{\epsilon}_j[Cache_j(\overline{y}_{j0})])/\mathscr{U} \overset{\sigma}{\Longrightarrow}$$

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

$$(Locmem_j(\overline{0})[\mathscr{U}/\mathscr{R}] \;||_{\mathscr{U}_j\cup\mathscr{W}}\; K^{\epsilon}_j[Cache_j(\overline{y}_{j0})])/\mathscr{U} \overset{\sigma_j}{\Longrightarrow}$$

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

Using the above lemma we find σ'_j with $\sigma_j \upharpoonright (\mathscr{W}_j \cup \mathscr{R}_j) = \sigma'_j \upharpoonright (\mathscr{W}_j \cup \mathscr{R}_j)$ and $\sigma_j \upharpoonright \mathscr{W} = \sigma'_j \upharpoonright \mathscr{W}$. The latter equality implies that for $j_1, j_2 \in I$ we have $\sigma'_{j_1} \upharpoonright \mathscr{W} = \sigma_{j_1} \upharpoonright \mathscr{W} = \sigma_{j_2} \upharpoonright \mathscr{W} = \sigma'_{j_2} \upharpoonright \mathscr{W}$. This means that we can apply fact 4 again, in the opposite direction, combining the σ'_j and find a σ' with $\sigma' \upharpoonright (\mathscr{W} \cup \mathscr{R}_j) = \sigma'_j \upharpoonright (\mathscr{W} \cup \mathscr{R}_j)$

$$\prod_{j\in I}^{\mathscr{W}} (Locmem_j(\overline{0})[\mathscr{U}/\mathscr{R}] \;||_{\mathscr{U}_j\cup\mathscr{W}} Cache_j(\overline{y}_{j0}))/\mathscr{U} \overset{\sigma'}{\Longrightarrow}$$

It follows that $\sigma' \upharpoonright (\mathscr{W}_j \cup \mathscr{R}_j) = \sigma \upharpoonright (\mathscr{W}_j \cup \mathscr{R}_j)$ for all $j \in I$, i.e. *Refinement*$_3$ is weakly sequentially 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 [**?**]. 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 $\{L^{\sigma}_j \mid \sigma \in (\mathscr{W} \cup \mathscr{U}_j)^*\}$ is for each $j \in I$ completely characterized by the following set of transductions:*

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

□

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

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

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

$$\begin{aligned}Refinement_{3'} \Leftarrow \prod_{j\in I}^{\mathscr{W}} (Locmem_j(\overline{0}) \;\; & [\mathscr{U}/\mathscr{R}] ||_{\mathscr{U}_j\cup\mathscr{W}} && (8)\\ & L^{\epsilon}_j[Cache'_j(\overline{y}_{j0})])/\mathscr{U}\\ & \text{for arbitrary } \overline{y}_{j0} \in r(\overline{0}).\end{aligned}$$

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

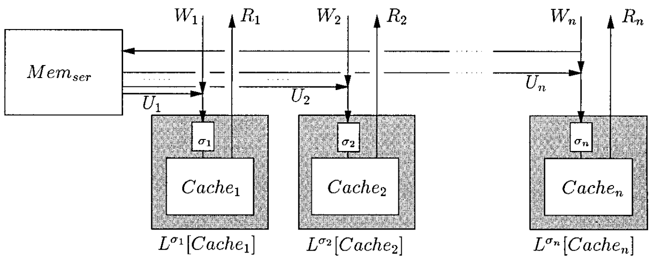

Fig. 8. Centralizing the memory

Lemma 18

$$L^\epsilon_j[Cache'_j(\overline{y}_{j0})] \quad \sim \quad K^\epsilon_j[Cache_j(\overline{y}_{j0})]$$

Proof. Left to the reader. □

Corollary 19 *Refinement$_{3'}$ is weakly sequentially 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.

$$\begin{aligned} Refinement_4 \Leftarrow\ & (Mem_{ser}(\overline{0})[\mathscr{U}/\mathscr{R}]\ ||_{\mathscr{U}\cup\mathscr{W}} \\ & \prod_{j\in I}^{\mathscr{W}} L^\epsilon_j[Cache'_j(\overline{y}_{j0})])/\mathscr{U} \qquad (9) \\ & \text{for arbitrary } \overline{y}_{j0} \in r(\overline{0}). \end{aligned}$$

Lemma 20

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

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

Proof.

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

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

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

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

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

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

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

$\sim$ {*law 5 of Table 3 and Lemma 10*}

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

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

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

□

Corollary 21 *Refinement$_4$ is weakly sequentially 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 $\{M^\sigma_j \mid \sigma \in \mathscr{W}_j^*\}$ 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 first and second transduction rules correspond to the queuing and dequeuing of local write actions. The third rule corresponds to the local read actions, and the last rule to all read and write actions at the other user interfaces. The last two rules are both instances of case 3 of Definition 4.

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

$$Refinement_5 \Leftarrow \qquad (10)$$

$$(M^\epsilon_1 \circ \ldots \circ M^\epsilon_n)[(Mem_{ser}(\overline{0})[\mathscr{U}/\mathscr{R}]\ ||_{\mathscr{U}\cup\mathscr{W}}$$

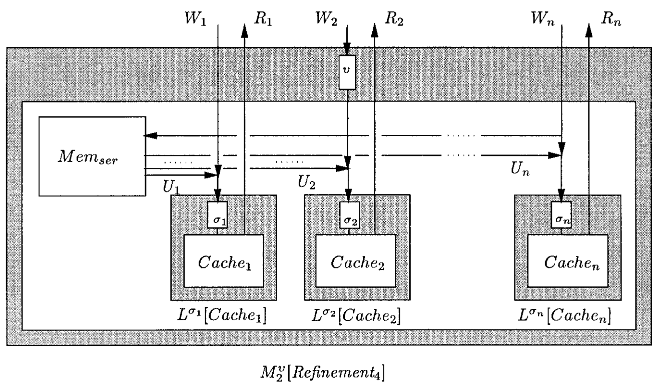

Fig. 9. Adding a user interface

$$\prod_{j\in I}^{\mathscr{W}} L_j^{\epsilon}[Cache'_j(\overline{y}_{j0})])/\mathscr{U}]$$

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

Here, $M_1^{\epsilon} \circ \ldots \circ M_n^{\epsilon}$ denotes the composition of n tranducer applications, one for each user interface. With the addition of these interfaces the last ingredients of the lazy caching memory of Fig. 1, viz. the *Out*-queues and their associated constraints, have been incorporated in the design. The following theorem and its corollary, therefore, express the correctness of the lazy caching memory.

Theorem 23 *For all $i \in I$*

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

is weakly sequentially 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_i^{ϵ} preserves the actions in $\mathscr{W}_i \cup \mathscr{R}_i$ and in $\mathscr{W}_j \cup \mathscr{R}_j$ for $j \neq i$, choosing $A = \mathscr{W}_i$ and $A = \emptyset$, respectively. The sequential consistency with $Mem_{ser}(\overline{0})$ then follows from Corollary 21. □

Corollary 24

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

is weakly sequentially 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 [**?**], 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 weakly sequentially consistent with the serial memory. For all n $W_1(1)(R_2(0))^n$ is weakly sequentially 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

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

Definition 25 (strong sequential consistency) *Let B_1 and B_2 be behaviour expressions with $L(B_i) \subseteq \mathscr{L}$. A behaviour B_1 is* strongly sequentially consistent *with B_2 iff*

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

□

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

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

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.*

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

□

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 [**?**]: that no read, write, or update action is continuously enabled but never executed.

Lemma 26 (extended preservation lemma) *Let $\mathscr{T}_Q = \{T^\sigma \mid \sigma \in Q^*\}$ 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 $\sigma \in Traces_\omega(T^\epsilon[B])$ we have*

$$\exists \sigma' \in Traces_\omega(B)\ \ \forall A \subseteq Q \ \text{ with } \ \sigma \upharpoonright (A \cup D_A) = \sigma' \upharpoonright (A \cup D_A)$$

Proof. We may assume that σ is an infinite trace, otherwise the proof of Lemma 8 applies. By the definition of an infinite trace we then get that $\sigma = \sigma_0.\sigma_1.\sigma_2.\cdots$ with

$$\exists \{T^{v_i}[B_i]\}_{i\in\mathbf{N}}\ \ T^{v_i}[B_i] \overset{\sigma_i}{\Longrightarrow} T^{v_{i+1}}[B_{i+1}] \ \text{ with } \ T^{v_0}[B_0] \equiv T^\epsilon[B]$$

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

$$\exists \{\sigma'_i\}_{i\in\mathbf{N}}\ \ T^{v_i} \underset{\sigma'_i}{\overset{\sigma_i}{\Longrightarrow}} T^{v_{i+1}} \ \text{ and } \ B_i \overset{\sigma'_i}{\Longrightarrow} B_{i+1}$$

It follows from Lemma 7 that $(\sigma'_0.\cdots.\sigma'_i) \upharpoonright (A \cup D_A)$ is prefix of $(\sigma_0.\cdots.\sigma_i) \upharpoonright (A \cup D_A)$ for all i.

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

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

Theorem 27

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

is strongly sequentially consistent with Mem$_{ser}$($\bar{0}$).

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 an *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, and others may not, have a buffered user interface. This is obtained by applying the tranducers M_i^ϵ for only those user interfaces i that should have *Out*-queues.
2. *cache buffers*: we can also allow asymmetry between caches in the sense that some may have buffered access and others not. This is similar to the above: apply the transducers L_i^ϵ only to those local caches that should have *In*-queues.
3. *local memories*: we may choose some users to have access to a complete local memory instead of a cache. This is obtained by carrying out refinement step 2 for only those interfaces where a cache instead of a local serial memory is required. It is easy to see that (the proof of) Lemma 10 does not depend on the particular I. This means that for any nonempty subset of $I_0 \subseteq I$ the I_0-product of local memories is bisimilar to a serial memory of type I_0. This is sufficient for the subsequent creation of a central background memory in transformation step 4 for those interfaces that do have caches.
4. *background memories*: we may choose to have several write-synchronizing background memories for smaller user groups (e.g. to expedite cache updates). This relies on the same observation as the previous point, applying it for disjoint $I_0, \ldots, I_i \subseteq I$.

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 the general exposition in [**?**]. 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].

It may be good to point out that the way in which we have factored the proof into transformations is not unique. We could have chosen, for example, to first recombine the local memories (now step 4) and then add the *In*-queues to the caches (now step 3). In that case we could even have tried to carry out the first three transformations in one fell swoop, factoring all local caches out of a central memory. There may even be design strategies completely different from the approach that we have followed. We have tried to follow a strategy that we believe leads to elegant and insightful transformations, and minimizes the amount of "global reasoning" by making arguments depend only on actions local to the user interfaces whenever this is possible. Whether this is the best approach is to some extent a matter of taste, of course.

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 transition 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 may be 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

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

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

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

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

Then

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

□

Thus far, 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.

We thank an anonymous referee for pointing out a subtle error in the original presentation of the preservation lemmas 8 and 26.

References

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

[BB87] Bolognesi T., Brinksma, E.: Introduction to the ISO specification language LOTOS. Computer Networks and ISDN Systems, 14:25–59 (1987)

[Bol92] Bolognesi, T.: Catalogue of LOTOS correctness preserving transformations. Technical Report Lo/WP1/T1.2/N0045/V03, Esprit Project 2304 Lotosphere, April 1992

[Bri92] Brinksma, E.: On the uniqueness of fixpoints modulo observation congruence. Lecture Notes in Computer Science 630, pp. 62–76. Berlin Heidelberg New York: Springer 1992

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

[Hoa85] Hoare, C.A.R.: Communicating Sequential Processes. Englewood Cliffs, NJ: Prentice-Hall 1985

[Lar90] Larsen, K.G.: Compositional theories based on an operational semantics of contexts. In: de Bakker, J.W., de Roever, W.P., Rozenberg, G. (eds.) Stepwise Refinement of Distributed Systems — Models, Formalisms, Correctness, Lecture Notes in Computer Science, Vol. 430 pp. 487–518, Berlin Heidelberg New York: Springer 1990

[Mil89] Milner, R.: Communication and Concurrency. Englewood Cliffs, NJ: Prentice-Hall 1989

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

[vG93] van Glabbeek, R.J.: The linear time - branching time spectrum ii. LectureNotes in Computer Science Vol. 715, pp. 66–81, Berlin Heidelberg New York: Springer 1993

Ed Brinksma holds the chair of Formal Methods and Tools of the Faculty of Computer Science at the University of Twente at Enschede, The Netherlands. His main research interest lies in the application of formal methods to the design of distributed systems, including specification, verification, implementation, testing and software tool support. Ed was chairman of the ISO standardization committee of the formal specification language LOTOS (ISO IS 8807). His current research focusses on testing (test derivation), validation by model checking, formal methods for real-time systems, linking performance models to formal specifications, and the introduction formal methods into industrial working practices. He is leader of the Systems Validation Centre at the Telematics Institute, a leading national research institute that is jointly funded by the Dutch government and major computer science and telecom industries.