Eindhoven University of Technology

Department of Mathematics and Computing Science

# Verifying Sequentially Consistent Memory using Interface Refinement

by

R. Gerth

93/48

Computing Science Note 93/48
Eindhoven, December 1993

Copies can be ordered from:
Mrs. M. Philips
Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513
5600 MB EINDHOVEN
The Netherlands
ISSN 0926-4515

# Verifying Sequentially Consistent Memory using Interface Refinement

Rob Gerth*
Eindhoven University of Technology†

December, 1993

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

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

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

> *the result of any execution [of the memory] is the same as if the operations [memory accesses] of all the processors were executed in some sequential order,*

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

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

*and the operations of each individual processor appear in this sequence in the order specified by its program.*

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

The aim of this paper is to show how sequential consistency can be interpreted as an instance of interface refinement [GKS92] and by verifying a sequentially consistent memory protocol—the lazy caching protocol of [ABM93]—. Although the protocol is proven correct in that paper, the proof is on a semantical level and is not grounded in a verification methodology. This makes the proof quite hard to follow and hard to generalize to more complex protocols such as release consistent or non-blocking memory.

In the next section we explain and define sequential consistency. The lazy caching protocol is introduced in the Section 3. The heart of the paper is formed by Section 3 and 4. The latter contains the proof of sequential correctness of the protocol. Section 3 gives an overview of the verification methodology. Investigating sequential consistency made us realize that the methodology to prove interface refinement as defined in [GKS92] can be considerably simplified. Accordingly, this Section is also of independent interest. In Section 5 some conclusions are drawn.

## 1 Sequential consistency

In order to understand Lamport's definition, we first fix the behavior of a standard, 'serial' shared memory. This is done in Figures 1 and 2.

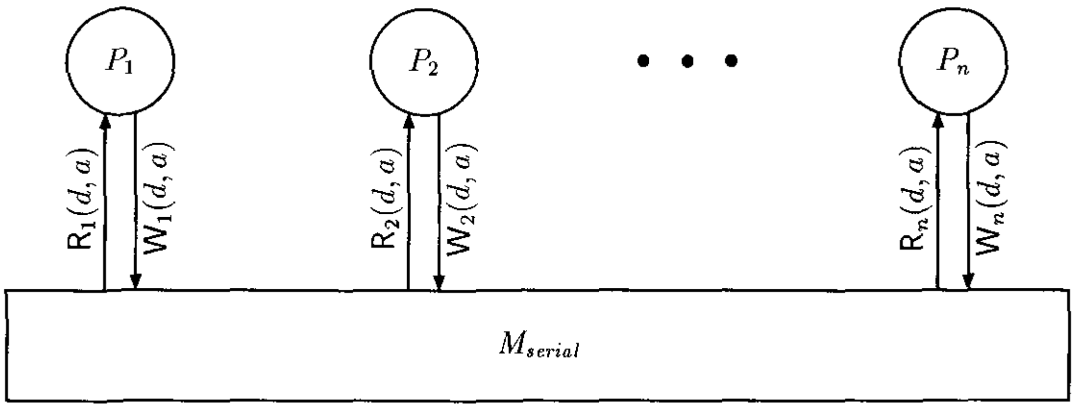


Figure 1: Architecture of $M_{serial}$

The interface of the memory comprises of read ($\mathsf{R}_i(d,a)$) and write ($\mathsf{W}_i(d,a)$) events for each processor $P_i$. The processors and the memory have to synchronize on these read and write events. The transition system in Figure 2 indicates that these are the only external events that $M_{serial}$ participates in and that it has no internal events. A read event $\mathsf{R}_i(d,a)$, issued by $P_i$, can only occur if the memory holds value $d$ at location $a$: $Mem[a] = d$. Write events $\mathsf{W}_i(d,a)$ can always occur with the expected result. The *external behavior* of the serial memory, Beh($M_{serial}$), is defined as the maximal (hence infinite) sequences of read and write events generated according to the transition system of Figure 2. Hence, the memory *serializes* the reads and writes of the processors.

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

Two objections that might be levied against this choice of interface are: events cannot overlap because they do not extend in time; and: read events specify the value that is read and thus do not really model read actions. Note that the second objection applies to the [ABM93] interface as well. The answer to both objections is that what is of importance are the points at which the memory system changes state and the values that can be read from memory as a result of these changes. Hence, write events should merely be viewed as the initiators of state changes while read events indicate which values can be returned. Thus, the precise way in which a process initiates a read or a write is of no importance to the modeling.

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

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

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

Figure 2: $M_{serial}$

We rephrase Lamport's proposal of correct behavior of sequentially consistent memory (SCM) thus *any external behavior, $\sigma$, [of the SCM] corresponds with an external behavior, $\tau$, of $M_{serial}$ so that the order in which the operations of each individual processor appear in $\sigma$ coincides with order in which they appear in $\tau$.*

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

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

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

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

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

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

Sequential consistency has been the canonical distributed memory model for a long time. In practice, however, different, still weaker memory models tend to be implemented as the synchronization overhead of SCM is still too large. For instance, the *processor consistency* model would allow the above behavior at the processors. See [Mos93] for an overview of distributed memory models.

## A formal definition

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

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

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

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

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

## 2 The lazy caching protocol

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

The architecture of $M_{distr}$ is depicted in Figure 3; the transition system in Figure 4. The protocol is thus geared towards a bus based architecture. Here, too, the interface

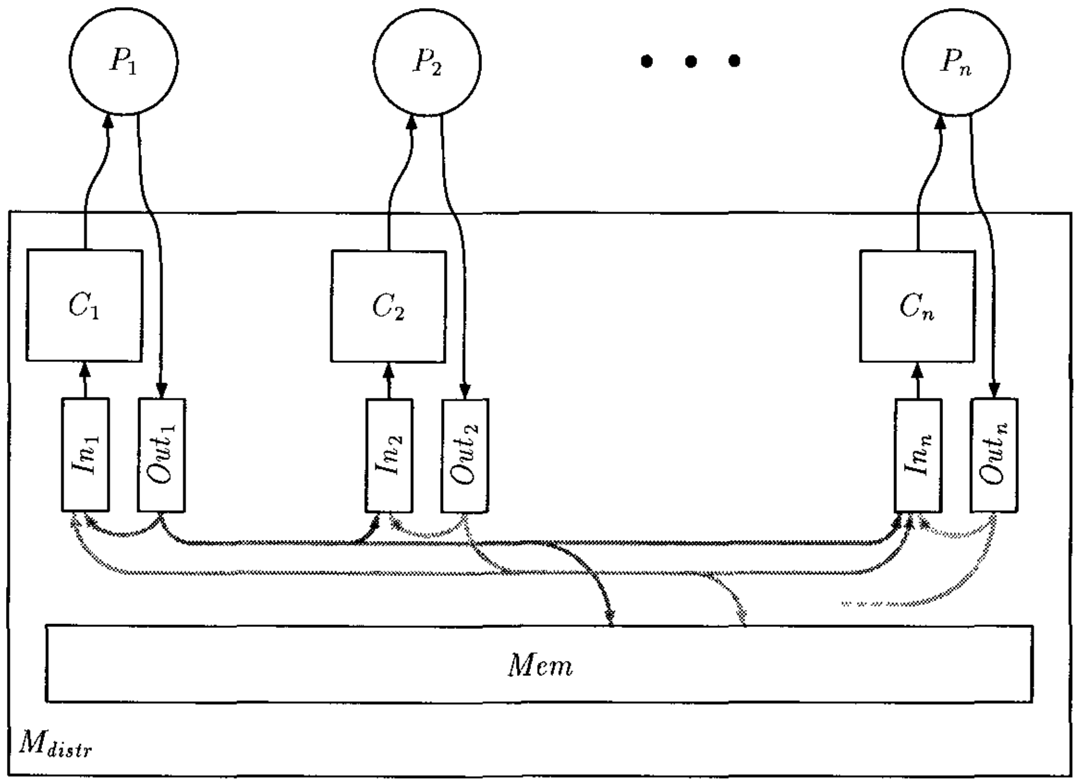


Figure 3: Architecture of $M_{distr}$

of the memory comprises of the read and write events of the processors. $M_{distr}$, however, interposes caches $C_i$ between the shared memory $Mem$ and the processes $P_i$. Each cache $C_i$ contains a part of the memory $Mem$ and has two queues associated with it: an out-queue

$Out_i$ in which $P_i$'s write requests are buffered and an in-queue $In_i$ in which the pending cache updates are stored. These queues model the asynchronous behavior of write events in a sequential consistent memory. The gray arrows indicate the information flows from the out queues to the in queues and to *Mem*.

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

A read event $\mathsf{R}_i(d,a)$, predictably, stalls until a copy of location $a$ is present in $C_i$ but also until the copy contains a 'correct' value in the following sense: sequential consistency implies that a processor $P_i$ reads the value at a location $a$ that was most recently written by $P_i$ unless some other processor updated $a$ in the mean time. Hence, a read event $\mathsf{R}_i(d,a)$ cannot occur unless all pending writes in $Out_i$ are processed as well as the cache update requests from $In_i$ that correspond to writes of $P_i$. For this reason, such cache update request are marked (with a $*$).

The transition system in Figure 4 makes all this precise.

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

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

## 3 Interface Refinement

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

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

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

| | |
|---|---|
| Initially: | $\forall a \;\; Mem[a] = 0$ $\wedge \forall i = 1 \ldots n \;\; C_i \subset Mem \wedge In_i = \{\} \wedge Out_i = \{\}$ |
| Fairness: | no action other than $\mathsf{CI}_i$ can be always enabled but never taken |

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

Figure 4: $M_{distr}$

If we compare the definitions of sequential consistency

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

and that of standard (trace) refinement

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

we detect a pattern:

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

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

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

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

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

**Definition 3.1 (Interface refinement)** *Let $\phi$ be some LTL formula. Then*

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

For example, standard trace refinement, $C$ $\mathsf{ref}$ $A$, is defined as $C$ $\mathsf{ref}_\phi$ $A$ by taking, e.g.,

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

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

## 3.1 Sequential consistency as interface refinement

For this we make a simplifying assumption

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

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

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

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

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

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

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

**Lemma 3.2** *$M_{distr}$ s.c. $M_{serial}$ iff $M_{distr}$ $\mathsf{ref}_{\phi}$ $M_{serial}$ with $\phi = \Box \bigwedge_{i=1\ldots n} \phi_i$ and*

$$\phi_i \quad \equiv \quad f_i(h_0) \lceil i \preceq h_1 \lceil i \quad \textit{for some } f_i \textit{ that increases i.o. on } M_{distr} \tag{1}$$

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

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

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

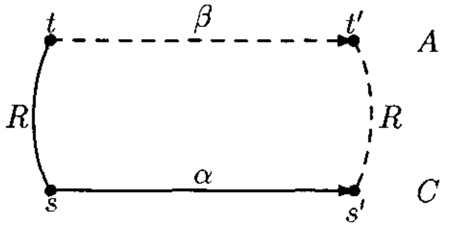
## 3.2 A proof rule

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

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

**Definition 3.3 (Weak simulation)** *Given transition systems $C$ and $A$, a relation $R \subseteq S(C) \times S(A)$ is a* weak simulation of $C$ in $A$, *$C \hookrightarrow_R A$, provided*

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



The inductive clause (2) is illustrated in the figure on the right. Given a state sequence $\sigma \in [\![C]\!]$, a weak simulation $C \hookrightarrow_R A$ constructs a state sequence $\tau$ of $A$ in which every state in $\tau$ is related to some state in $\sigma$. However, we do not necessarily have $\tau \in [\![A]\!]$. First of all, $A$ may have fairness constraints which $\tau$ may violate. Secondly, $\tau$ may be finite because from some moment onwards $R$ relates the transitions in $\sigma$ with $\epsilon$ transitions 'in' $\tau$. Fairness constraints are no problem for us, as $M_{serial}$ will play the rôle of $A$ and it does not have fairness constraints. Forcing $\tau$ to be infinite will be done implicitly, later on.

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

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

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

This observation immediately suggests the proof rule in Figure 5.

---

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

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

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

Figure 5: Proof rule for establishing sequential consistency

---

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

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

The proof rule for general interface refinement, $C$ $\mathsf{ref}_\phi$ $A$, is based on the same ideas. The first step, again, is establishing a weak simulation. The second step changes because now $\phi$ need not be of the form $\Box\bar{\phi}$ and it is this form that determined the second premiss in rule 5. For instance, if $\phi \equiv \Box\Diamond\bar{\phi}$ for some state assertion $\bar{\phi}$, then we need to establish $\bar{\phi}$ at infinitely many state pairs along every pair of $R$-related state sequences. For this we introduce an auxiliary state formula $d$ such that $C \models \Box\Diamond d$ and demand that $(s,t) \in R \;\&\; s \models d \;\Rightarrow\; s{\times}t \models \bar{\phi}$. In case $\phi \equiv \Box\Diamond\phi' \wedge \Diamond\phi''$ we would use two auxiliary state assertions $d'$ and $d''$ such that $C \models \Box\Diamond d' \wedge \Diamond d''$, etc. The normal form result of [MP91] tells us that a finite number of auxiliary formulae always suffices. Specifically, we have that for every TL formula $\phi$ (without quantifiers) there is a propositional TL formula $\Psi$ with propositional variables $p_1, \ldots, p_n$ and state formulae $\phi_1, \ldots, \phi_n$ such that $\models \phi \leftrightarrow \Psi[\phi_1/p_1, \ldots, \phi_n/p_n]$, where $\cdot/\cdot$ denotes syntactic substitution; we usually write $\Psi(\phi_1, \ldots \phi_n)$. The following proof rule applies to the general case.

---

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

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

Figure 6: Proof rule for general interface refinement

---

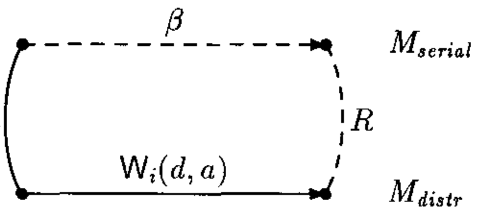
The sequential consistency proof rule is obtained by taking $d_i \equiv$ true and by noting that the formula in Lemma 3.2 is in normal form.

## 4 Correctness Proof of $M_{distr}$ s.c. $M_{serial}$

### 4.1 Constructing a weak simulation $R$

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

In the situation indicated on the right, $\beta$ should not be the corresponding $\mathsf{W}_i(d,a)$-event. If it is, $\mathsf{R}_j(e,a)$-actions in the distributed memory that read an earlier value $e$ at location $a$ become disabled in the serial memory. This suggests that the corresponding serial write be postponed until the write has been *completed*, that is, until no processor can read an older value from the distributed memory system, i.e., from its cache. As a consequence, any read-action that reads the value of an uncompleted write-action is postponed as well.



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

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

In order to formalize the above ideas, we adapt the transition system of $M_{distr}$; see Figure 7. Every write-action uniquely tags the value that it writes so that cache and memory update actions can be traced back to the specific action that 'caused' them.

Obviously, we still have

**Lemma 4.1** $\mathrm{Beh}(M^T_{distr}) = \mathrm{Beh}(M_{distr})$

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

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

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

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

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

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

Initially: $\forall a \;\; Mem[a] = 0{\star}\mathbf{0}$
$\wedge \, \forall i = 1 \ldots n \;\; C_i \subset Mem \wedge In_i = \{\} \wedge Out_i = \{\} \wedge \mathbf{t}_i = \mathbf{0}$

Fairness: no action other than $\mathsf{CI}_i$ can be always enabled but never taken

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

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

Figure 7: Adapted $M^T_{distr}$

- $(k,\alpha)$ **occurs in** $h$ iff $\alpha$ occurs in $h$ (i.e., $h = h_0\alpha h_1$ for some $h_0$, $h_1$) and $h$ contains at least $k$ occurrences of type($\alpha$)-events

- $(k,\alpha)$ **occurs before** $(l,\beta)$ **in** $h$ iff $(l,\beta)$ **occurs in** $h$ and there is a prefix $h'$ of $h$ such that $(k,\alpha)$ **occurs in** $h'$ but not $(l,\beta)$ **occurs in** $h'$

- $(k, \mathsf{W}_i(d,a))$ **is completed in** $h$ iff $\forall j = 1 \ldots n$ $\mathsf{CU}_j(d{\star}(k{\star}i), a)$ occurs in $h$

- $\alpha$ **completes** $(k,\beta)$ **in** $h$ iff not $(k,\beta)$ **is completed in** $h$ but $(k,\beta)$ **is completed in** $h\alpha$

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

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

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

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

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

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

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

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

**Lemma 4.2** *Let* $k > 0$ *and* $(k,\alpha) \neq (l,\beta)$. *The following formulae are invariants of* $M^T_{distr}$:

1. $(k, \mathsf{R}_i(d,a))$ **occurs in** $\mathsf{h} \rightarrow (l, \mathsf{W}_j(d,a))$ **is read by** $(k, \mathsf{R}_i(d,a))$ **in** $\mathsf{h}$ *for some* $(l, \mathsf{W}_j(d,a))$
2. $(k,\alpha)$ **occurs in** $\mathsf{h} \wedge \text{type}(\alpha) = \mathsf{W}_i \rightarrow \Diamond(k,\alpha)$ **is completed in** $\mathsf{h}$
3. $(k,\alpha)$ **is read by** $(l,\beta)$ **in** $\mathsf{h} \rightarrow (k,\alpha)$ **occurs before** $(l,\beta)$ **in** $\mathsf{h}$
4. $(k, \mathsf{MW}_i(d,a))$ **occurs before** $(l, \mathsf{MW}_j(d',a'))$ **in** $\mathsf{h} \rightarrow$
   $\neg(l, \mathsf{CU}_f(d',' a))$ **occurs before** $(k, \mathsf{CU}_f(d,a))$ **in** $\mathsf{h}$
5. $\alpha \in Act_i \wedge \beta \in Act_i \rightarrow$
   $(k,\alpha)$ **reads before** $(l,\beta)$ **in** $\mathsf{h} \rightarrow (k,\alpha)$ **occurs before** $(l,\beta)$ **in** $\mathsf{h}$
   $\wedge\, (k,\alpha)$ **distributes before** $(l,\beta)$ **in** $\mathsf{h} \rightarrow (k,\alpha)$ **occurs before** $(l,\beta)$ **in** $\mathsf{h}$
6. $\alpha \in Act_i \wedge \beta \in Act_i \wedge \alpha \equiv \mathsf{W}_i(d,a) \wedge (k,\alpha)$ **occurs before** $(l,\beta)$ **in** $\mathsf{h} \rightarrow$
   $\text{type}(\beta) = \mathsf{W}_i \rightarrow (k,\alpha)$ **distributes before** $(l,\beta)$ **in** $\mathsf{h}$
   $\wedge\, \text{type}(\beta) = \mathsf{R}_i \rightarrow (k, \mathsf{CU}_i(d\star(k\star i), a))$ **occurs before** $(l,\beta)$ **in** $\mathsf{h}$
   $\wedge\, (l,\beta)$ **is completed in** $\mathsf{h} \rightarrow (k,\alpha)$ **is completed in** $\mathsf{h}$
7. $(k, \mathsf{W}_i(d',a'))$ **occurs before** $(l, \mathsf{R}_i(d,a))$ **in** $\mathsf{h}$
   $\wedge\, (l, \mathsf{R}_i(d,a))$ **reads before** $(m, \mathsf{W}_j(\bar{d},a))$ **in** $\mathsf{h} \rightarrow$
   $(k, \mathsf{W}_i(d',a'))$ **distributes before** $(m, \mathsf{W}_j(\bar{d},a))$ **in** $\mathsf{h}$

**Proof.** We shall not give completely formal proofs here.

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

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

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

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

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

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

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

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

### 4.1.1 Dag $<_h$

Define the dag $<_h$ on the set

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

as the smallest relation satisfying

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

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

### 4.1.2 Simulation $R$

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

$$\begin{tikzcd}
t \arrow[r, dashed, "\beta"] \arrow[d, dash, "R"'] & t' \arrow[d, dash, dashed, "R"] \\
s \arrow[r, "\alpha"] & s'
\end{tikzcd}$$

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

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

**Lemma 4.3**

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

We defer the proof of Clause (1); Clause (2) is a direct consequence of Lemma 4.2(2) and the fact that each process issues infinitely many writes.

From the inductive definition of $R$, Lemma 4.2(2) and Lemma 4.3(1), we immediately conclude that $M^T_{distr} \hookrightarrow_R M_{serial}$; provided we can show that $<_h$ is indeed a dag so that minimal elements always exists.

**Theorem 4.4** *$<_h$ is a dag.*

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

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

**Lemma 4.5** *Let $(k, \mathsf{W}_i(d,a)) <^+_h (l,\beta)$.*

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

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

First assume that $(k,\alpha) <_h (l,\beta)$. Then either (i) $j = i$ and $(k,\alpha)$ **occurs before** $(l,\beta)$ **in** $h$ or (ii) $j \neq i$ and $(k,\alpha)$ **is read by** $(l,\beta)$ **in** $h$ or $(k,\alpha)$ **distributes before** $(l,\beta)$ **in** $h$. For case (i) the Lemma follows from Lemma 4.2(6). Case (ii) follows immediately from the definitions of the **is read by** and **distributes before** relations.

Next, suppose that $(k,\alpha) <_h^+ (m,\gamma) <_h (l,\beta)$. By induction the Lemma holds for $(k,\alpha) <_h^+ (m,\gamma)$. According to the definition of $<_h$ there are four cases. If $\gamma \notin Act_j$ and $(m,\gamma)$ **reads before** $(l,\beta)$ **in** $h$, then the result follows from Lemma 4.2(7). The other cases are as (i) and (ii) above. □

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

**Proof of Theorem 4.4.** Suppose that $<_h$ admits a cycle. Then, we must have $(\bar{k},\bar{\alpha}) <_h^+ (\bar{l},\bar{\beta})$ and $(\bar{l},\bar{\beta}) <_h^+ (\bar{k},\bar{\alpha})$ for some $\bar{\alpha}$ and $\bar{\beta}$. W.l.o.g., we may assume that $(\bar{k},\bar{\alpha}) <_h (\bar{l},\bar{\beta})$. So, by definition of $<_h$, there must be an $(\bar{m},\bar{\gamma})$ such that $(\bar{l},\bar{\beta}) <_h^+ (\bar{m},\bar{\gamma}) <_h^+ (\bar{k},\bar{\alpha})$ and not $\bar{\alpha},\bar{\beta},\bar{\gamma} \in Act_i$ for some $i$.

By transitivity of $<_h^+$ this means that we have (A) $(k,\mathsf{W}_i(d,a))$ **is read by** $(l,\beta)$ **in** $h$ and $(l,\beta) <_h^+ (k,\mathsf{W}_i(d,a))$ or (B) $(k,\mathsf{W}_i(d,a))$ **distributes before** $(l,\beta)$ **in** $h$ and $(l,\beta) <_h^+ (k,\mathsf{W}_i(d,a))$ or (C) $(k,\alpha)$ **reads before** $(l,\mathsf{W}_i(d,a))$ **in** $h$ and $(l,\mathsf{W}_i(d,a)) <_h^+ (k,\alpha)$ with $\alpha,\beta \notin Act_i$. We immediately obtain $(k,\mathsf{MW}_i(d\star(k\star i),a))$ **occurs before** $(k,\mathsf{W}_i(d,a))$ **in** $h$ for case (A) and by Lemma 4.5 $(k,\mathsf{MW}_i(d\star(k\star i),a))$ **occurs before** $(k,\mathsf{MW}_i(d\star(k\star i),a))$ **in** $h$ for case (B) and $(l,\mathsf{MW}_i(d\star(l\star i),a))$ **occurs before** $(k,\alpha)$ **in** $h$ for case (C). The first two cases give immediate contradictions; the last one via Lemma 4.2(7) from which we infer that $(l,\mathsf{W}_i(d,a))$ **distributes before** $(l,\mathsf{W}_i(d,a))$ **in** $h$ which is impossible. □

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

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

**Proof of Lemma 4.3(1).** In the proof we refer to the figure in the definition of the simulation $R$ on Page 16. First observe that $(k,\gamma) \in \mathrm{dom}(<_{s'(\mathsf{h})})$ implies that $(k,\gamma)$ **occurs in** $s'(\mathsf{h})$ for any $(k,\gamma)$. This is immediate from the definition of $<_h$.

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

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

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

We conclude that $\alpha$ cannot be of this form and, hence, that $\beta$ is enabled in $t$. □

## 4.2 Concluding the proof

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

$$f_i(h) = h' \qquad \text{where } h' \text{ is the prefix of } h \text{ such that } |h'| = n_i(h) \text{ with } n_i \text{ inductively defined by } n_i(\varepsilon) = 0 \text{ and}$$

$$n_i(h\alpha) = \begin{cases} n_i(h)+1, & \text{if } \alpha \textbf{ completes } (n,\gamma) \textbf{ in } h \wedge \gamma \in Act_i \\ n_i(h), & \text{otherwise} \end{cases}$$

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

By Lemma 4.2(3) and the assumption that every processor contributes infinitely many writes, each $f_i$ increases i.o. on $M_{distr}$.

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

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

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

For the inductive step, we may assume that $f_i(s'(\mathsf{h})) \restriction i \not\preceq t(\mathsf{h}) \restriction i$. We refer to the definition of $R$ on Page 16. Hence, we must have (i) $\alpha$ **completes** $(m,\gamma)$ **in** $s(\mathsf{h})$ with $\gamma \in Act_i$. Let $f_i(s'(\mathsf{h})) \restriction i = f_i(s(\mathsf{h})) \restriction i\,\hat{}\,\delta$ and let $\delta$ be the $n$-th event of type($\delta$). Then, (ii) $(n,\delta)$ **is ready in** $s'(\mathsf{h}), t(\mathsf{h})$ must be true. By induction, it suffices to prove that $\delta \equiv \beta$, where $\beta$ is the transition taken in $t$ according to $R$.

From Clause 2 in the definition of $<_{s'(\mathsf{h})}$, we have $\perp \not<_{s'(\mathsf{h})} (m,\gamma)$ so that $T \cap Act_i \neq \emptyset$ by Lemma 4.3(1). In fact $|T \cap Act_i| = 1$ as Clause (1) says that $<_{s'(\mathsf{h})}$ extends the ordering on event occurrences induced by $s'(\mathsf{h}) \restriction i$. Because (i) holds, we know from Lemma 4.2(6) that for any $\mathsf{W}_i$ event $\delta$, if $(r,\delta)$ **occurs in** $s(f_i(\mathsf{h}))$ then $(r,\delta)$ **is completed in** $s(f_i(\mathsf{h}))$; whence $\perp \not<_{s'(\mathsf{h})} (r,\delta)$. By (ii) and the fact that $\perp$ is never covered by read events, we then have $(n,\delta) \in T \cap Act_i$ and also $\delta \equiv \beta$ since $(l,\beta) \in T \cap Act_i$ for some $l$ by definition of $R$.

## 5 Conclusions

We have worked out the proof in considerable detail. The proof rule demands that a weak simulation be constructed as the first step. This can be interpreted as defining a scheduler that schedules the appropriate event in $M_{serial}$ for every $M_{distr}$-event. For verifying sequential consistency this is a quite natural approach because the purpose of the protocol is to ensure that the event sequences that each process engages in can also be obtained from a serial memory. In this respect, there is a correspondence with the verification approach of [ABM93]. An important ingredient of the proof is the 'delayed' checking of sequential consistency of prefixes, which is inherent to our approach to interface refinement. This makes the definition of $<_h$ easier, although a penalty is paid in the form of a slightly more complex proof of Lemma 4.3(1). In contrast, the scheduler used in [ABM93] needs to maintain sequential consistency of the complete history instead of (ever longer) prefixes of history.

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

## Acknowledgments

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

## References

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

[BFKR92] H. Burkhardt, S. Frank, B. Knobe, and J. Rothnie. Overview of the KSR1 computer system. Technical Report SR-TR-9202001, Kendall Square Research, Boston, 1992.

[GKS92] R. Gerth, R. Kuiper, and J. Segers. Interface refinement in reactive systems. In *Proceedings of the Conference on Concurrency (CONCUR)*, volume 630 of *Lecture Notes in Computer Sciecne*, pages 77–94. Springer Verlag, June 1992.

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

[Lam79] L. Lamport. How to make a multiprocessor that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28:690–691, 1979.

[LLG+92] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The Stanford Dash multiprocessor. *IEEE Computer*, pages 63–79, 1992.

[Mos93] D. Mosberger. Memory consistency models. *ACM SIGOP Operating Systems Review*, 27(1):18–27, 1993.

[MP91] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, New York, 1991.

| | | |
|---|---|---|
| 91/18 | Rik van Geldrop | Transformational Query Solving, p. 35. |
| 91/19 | Erik Poll | Some categorical properties for a model for second order lambda calculus with subtyping, p. 21. |
| 91/20 | A.E. Eiben<br>R.V. Schuwer | Knowledge Base Systems, a Formal Model, p. 21. |
| 91/21 | J. Coenen<br>W.-P. de Roever<br>J.Zwiers | Assertional Data Reification Proofs: Survey and Perspective, p. 18. |
| 91/22 | G. Wolf | Schedule Management: an Object Oriented Approach, p. 26. |
| 91/23 | K.M. van Hee<br>L.J. Somers<br>M. Voorhoeve | Z and high level Petri nets, p. 16. |
| 91/24 | A.T.M. Aerts<br>D. de Reus | Formal semantics for BRM with examples, p. 25. |
| 91/25 | P. Zhou<br>J. Hooman<br>R. Kuiper | A compositional proof system for real-time systems based on explicit clock temporal logic: soundness and complete ness, p. 52. |
| 91/26 | P. de Bra<br>G.J. Houben<br>J. Paredaens | The GOOD based hypertext reference model, p. 12. |
| 91/27 | F. de Boer<br>C. Palamidessi | Embedding as a tool for language comparison: On the CSP hierarchy, p. 17. |
| 91/28 | F. de Boer | A compositional proof system for dynamic proces creation, p. 24. |
| 91/29 | H. Ten Eikelder<br>R. van Geldrop | Correctness of Acceptor Schemes for Regular Languages, p. 31. |
| 91/30 | J.C.M. Baeten<br>F.W. Vaandrager | An Algebra for Process Creation, p. 29. |
| 91/31 | H. ten Eikelder | Some algorithms to decide the equivalence of recursive types, p. 26. |
| 91/32 | P. Struik | Techniques for designing efficient parallel programs, p. 14. |
| 91/33 | W. v.d. Aalst | The modelling and analysis of queueing systems with QNM-ExSpect, p. 23. |
| 91/34 | J. Coenen | Specifying fault tolerant programs in deontic logic, p. 15. |
| 91/35 | F.S. de Boer<br>J.W. Klop<br>C. Palamidessi | Asynchronous communication in process algebra, p. 20. |

|  |  |  |
|---|---|---|
| 92/22 | R. Nederpelt<br>F.Kamareddine | A useful lambda notation, p. 17. |
| 92/23 | F.Kamareddine<br>E.Klein | Nominalization, Predication and Type Containment, p. 40. |
| 92/24 | M.Codish<br>D.Dams<br>Eyal Yardeni | Bottum-up Abstract Interpretation of Logic Programs, p. 33. |
| 92/25 | E.Poll | A Programming Logic for Fω, p. 15. |
| 92/26 | T.H.W.Beelen<br>W.J.J.Stut<br>P.A.C.Verkoulen | A modelling method using MOVIE and SimCon/ExSpect, p. 15. |
| 92/27 | B. Watson<br>G. Zwaan | A taxonomy of keyword pattern matching algorithms, p. 50. |
| 93/01 | R. van Geldrop | Deriving the Aho-Corasick algorithms: a case study into the synergy of programming methods, p. 36. |
| 93/02 | T. Verhoeff | A continuous version of the Prisoner's Dilemma, p. 17 |
| 93/03 | T. Verhoeff | Quicksort for linked lists, p. 8. |
| 93/04 | E.H.L. Aarts<br>J.H.M. Korst<br>P.J. Zwietering | Deterministic and randomized local search, p. 78. |
| 93/05 | J.C.M. Baeten<br>C. Verhoef | A congruence theorem for structured operational semantics with predicates, p. 18. |
| 93/06 | J.P. Veltkamp | On the unavoidability of metastable behaviour, p. 29 |
| 93/07 | P.D. Moerland | Exercises in Multiprogramming, p. 97 |
| 93/08 | J. Verhoosel | A Formal Deterministic Scheduling Model for Hard Real-Time Executions in DEDOS, p. 32. |
| 93/09 | K.M. van Hee | Systems Engineering: a Formal Approach<br>Part I: System Concepts, p. 72. |
| 93/10 | K.M. van Hee | Systems Engineering: a Formal Approach<br>Part II: Frameworks, p. 44. |
| 93/11 | K.M. van Hee | Systems Engineering: a Formal Approach<br>Part III: Modeling Methods, p. 101. |
| 93/12 | K.M. van Hee | Systems Engineering: a Formal Approach<br>Part IV: Analysis Methods, p. 63. |
| 93/13 | K.M. van Hee | Systems Engineering: a Formal Approach<br>Part V: Specification Language, p. 89. |
| 93/14 | J.C.M. Baeten<br>J.A. Bergstra | On Sequential Composition, Action Prefixes and Process Prefix, p. 21. |

| | | |
|---|---|---|
| 93/15 | J.C.M. Baeten<br>J.A. Bergstra<br>R.N. Bol | A Real-Time Process Logic, p. 31. |
| 93/16 | H. Schepers<br>J. Hooman | A Trace-Based Compositional Proof Theory for Fault Tolerant Distributed Systems, p. 27 |
| 93/17 | D. Alstein<br>P. van der Stok | Hard Real-Time Reliable Multicast in the DEDOS system, p. 19. |
| 93/18 | C. Verhoef | A congruence theorem for structured operational semantics with predicates and negative premises, p. 22. |
| 93/19 | G-J. Houben | The Design of an Online Help Facility for ExSpect, p.21. |
| 93/20 | F.S. de Boer | A Process Algebra of Concurrent Constraint Programming, p. 15. |
| 93/21 | M. Codish<br>D. Dams<br>G. Filé<br>M. Bruynooghe | Freeness Analysis for Logic Programs - And Correctness?, p. 24. |
| 93/22 | E. Poll | A Typechecker for Bijective Pure Type Systems, p. 28. |
| 93/23 | E. de Kogel | Relational Algebra and Equational Proofs, p. 23. |
| 93/24 | E. Poll and Paula Severi | Pure Type Systems with Definitions, p. 38. |
| 93/25 | H. Schepers and R. Gerth | A Compositional Proof Theory for Fault Tolerant Real-Time Distributed Systems, p. 31. |
| 93/26 | W.M.P. van der Aalst | Multi-dimensional Petri nets, p. 25. |
| 93/27 | T. Kloks and D. Kratsch | Finding all minimal separators of a graph, p. 11. |
| 93/28 | F. Kamareddine and<br>R. Nederpelt | A Semantics for a fine λ-calculus with de Bruijn indices, p. 49. |
| 93/29 | R. Post and P. De Bra | GOLD, a Graph Oriented Language for Databases, p. 42. |
| 93/30 | J. Deogun<br>T. Kloks<br>D. Kratsch<br>H. Müller | On Vertex Ranking for Permutation and Other Graphs, p. 11. |
| 93/31 | W. Körver | Derivation of delay insensitive and speed independent CMOS circuits, using directed commands and production rule sets, p. 40. |
| 93/32 | H. ten Eikelder and<br>H. van Geldrop | On the Correctness of some Algorithms to generate Finite Automata for Regular Expressions, p. 17. |
| 93/33 | L. Loyens and J. Moonen | ILIAS, a sequential language for parallel matrix computations, p. 20. |

| | | |
|---|---|---|
| 93/34 | J.C.M. Baeten and<br>J.A. Bergstra | Real Time Process Algebra with Infinitesimals, p.39. |
| 93/35 | W. Ferrer and<br>P. Severi | Abstract Reduction and Topology, p. 28. |
| 93/36 | J.C.M. Baeten and<br>J.A. Bergstra | Non Interleaving Process Algebra, p. 17. |
| 93/37 | J. Brunekreef<br>J-P. Katoen<br>R. Koymans<br>S. Mauw | Design and Analysis of<br>Dynamic Leader Election Protocols<br>in Broadcast Networks, p. 73. |
| 93/38 | C. Verhoef | A general conservative extension theorem in process algebra, p. 17. |
| 93/39 | W.P.M. Nuijten<br>E.H.L. Aarts<br>D.A.A. van Erp Taalman Kip<br>K.M. van Hee | Job Shop Scheduling by Constraint Satisfaction, p. 22. |
| 93/40 | P.D.V. van der Stok<br>M.M.M.P.J. Claessen<br>D. Alstein | A Hierarchical Membership Protocol for Synchronous Distributed Systems, p. 43. |
| 93/41 | A. Bijlsma | Temporal operators viewed as predicate transformers, p. 11. |
| 93/42 | P.M.P. Rambags | Automatic Verification of Regular Protocols in P/T Nets, p. 23. |
| 93/43 | B.W. Watson | A taxomomy of finite automata construction algorithms, p. 87. |
| 93/44 | B.W. Watson | A taxonomy of finite automata minimization algorithms, p. 23. |
| 93/45 | E.J. Luit<br>J.M.M. Martin | A precise clock synchronization protocol,p. |
| 93/46 | T. Kloks<br>D. Kratsch<br>J. Spinrad | Treewidth and Patwidth of Cocomparability graphs of Bounded Dimension, p. 14. |
| 93/47 | W. v.d. Aalst<br>P. De Bra<br>G.J. Houben<br>Y. Komatzky | Browsing Semantics in the "Tower" Model, p. 19. |