Cache Coherency in SCI: Specification and a Sketch of Correctness*

Amy Felty[1] and Frank Stomp[2]

[1]Bell Laboratories, 700 Mountain Avenue, Murray Hill, NJ 07974, USA;
Email: felty@research.bell-labs.com.
[2] Wayne State University, Department of Computer Science, Detroit, MI 48202, USA.
Email: fstomp@cs.wayne.edu

Keywords: Formal Verification; Temporal Logic; Cache Coherency; SCI (Scalable Coherent Interface); IEEE Standard; Distributed Systems

Abstract. SCI – Scalable Coherent Interface – is an IEEE standard for specifying communication between multiprocessors in a shared memory model. In this paper we model part of SCI by a program written in a UNITY-like programming language. This part of SCI is formally specified in Manna and Pnueli's Linear Time Temporal Logic (LTL). We give a sketch of our proof that the program satisfies its specification. The proof has carried out within LTL. It uses history variables. Structuring of the proof hass achieved by careful formulation of lemmata and the use of auxiliary predicates as an abstraction mechanism.

1. Introduction

In this paper we formalize and sketch verification of part of the SCI (Scalable Coherent Interface) protocol [IEE90]. The correctness proof can be found in the longer version [FS99] of the current paper. This protocol is an IEEE standard for specifying communication between shared memory multiprocessors. It is called scalable because the protocol is intended to be performed in a system which may consist of up to 64,000 processors. Our correctness proof has been carried out for

* In *Formal Aspects of Computing*, 11(5):475–497, 1999. ©Springer-Verlag London Limited 1999. A preliminary version of this paper has appeared in the Proceedings of the eleventh Annual Conference on Computer Assurance (COMPASS96), see [FS96].
Correspondence and offprint requests to: Frank Stomp, Wayne State University, Department of Computer Science, Detroit, MI 48202, USA.

an *arbitrary, finite number of processors* (and not for some maximum number of processors).

Our model of the protocol has been extracted from the informal description in a document from 1990 when the SCI protocol had been proposed as an IEEE standard. (It became a standard in 1992.) The SCI protocol is large and complex. For this reason, we consider only the cache coherence portion of this protocol. In addition, we model only an abstraction of this portion. For example, we do not keep track of processors which want to read only (and not write) and we consider the problem with one cache line only. (Multiple cache lines require a straightforward extension of the proof. In essence, we need copies of our current proof.) Also, in our model of the protocol we assume messages sent from one process to another process arrive at the latter process in the same order as sent. In the standard this is not necessarily the case.

For any proof of this complexity and size, it is essential for both the verifier and the reader to structure the proof. We have done so by formulating a number of lemmata, each of which can be proved directly or using previously formulated (and proved) lemmata. Also, we have introduced a number of auxiliary predicates as an abstraction mechanism. In addition, for rather big lemmata we have proved properties under certain assumptions, which are then later discharged.

Our part of the SCI protocol is formally modeled by a program written in a guarded command programming language similar to UNITY [CM88]. Its specification is formulated in Manna and Pnueli's Linear Time Temporal Logic (LTL) [MP91]. We have proved within LTL that the program meets its specification. A *history variable* has been used in order to reason about the program's communication behavior. In addition to this variable we have also used *logical variables*, also called *ghost variables* or *freeze variables*, in our correctness proof. They are used to *freeze* the values of the history and of certain program variables during a computation when reasoning about the program. As mentioned above the correctness proof can be found in [FS99]. The current paper presents a sketch of that proof.

The presence of multiple caches introduces the problem of *coherence*. According to [CF78] *a memory scheme is coherent if the value returned on a read is the value given by the latest store with the same address.* Coherence is usually achieved by *snooping*. In essence, all processors listen to a bus, and either invalidate or update their caches when data is written into memory. This kind of cache coherence protocol relies on a broadcasting mechanism. They do not scale well because the bus becomes a bottleneck. In non-snooping cache coherence algorithms it is often the case that memory keeps track of the caches which may be affected when data is written into memory. In this case the bottleneck is at the memory controller. To overcome these bottlenecks the SCI protocol decentralizes the communication. Broadcast is replaced by point-to-point communication which requires more messages per read/write but messages are sent only to the relevant processes. For bookkeeping, doubly linked lists of processes are used to keep track of the caches which need to be updated when data is modified.

An attempt to validate the program verified in the current paper has been made by the model checking community. Holzmann [Hol95] using SPIN [Hol91], Kurshan [Kur95] using COSPAN [Kur89], and Long and McMillan [LM95] using SMV [McM93] report to have validated the program for up to five processes. The problem that each of the model checkers face in this case is the *state explosion problem.*

Other work on the formal specification of the SCI protocol has been carried out by Gjessing et al. [GKMK91, GMK91]. Using stepwise refinement, multiple layers of the protocol are formalized at different levels of abstraction and functions are defined which map one level to the next. The lowest level of formal description is comparable with the C-code specification in the SCI document. This formalization work is part of an ongoing effort to fully verify the SCI cache coherence protocol. Stern and Dill [SD95] describe another ongoing project of automatically verifying the SCI protocol. They have discovered several errors in the C-code which defines that protocol. An overview of the SCI protocol and related projects can be found in [Gus92].

There is a vast amount of work done on other cache coherence algorithms, see, e.g., [ABM93, SD87, SS88, IEE92] to name just a few of them. The algorithm proposed by Afek, Brown, and Merritt in [ABM93] explores a formalization of Lamport's notion of sequential consistency [Lam79]. (Whereas cache coherence ensures that processors have a consistent view of the cache, sequential consistency addresses the question of what order data writes are observed by other processors [PH96].) This algorithm has recently been the subject of various verification methods: Brinksma [Bri95] uses queue-like action transducers; Gerth [Ger95] uses a generalized version of refinement; Graf [Gra95] uses abstraction and model checking; Janssen, Poel, and Zwiers [JPZ95] apply a compositional approach; Jonsson, Pnueli, and Rump [JPR95] apply a partial order transducer; Katz [Kat95] uses ISTL [KP87]; Ladkin, Lamport, Olivier, and Roegel [LLOR95] apply the temporal logic TLA [Lam94]; Lowe and Davies [LD95] use CSP [Hoa85]. In each of these proofs the emphasis is on sequential consistency. Pong and Dubois [PD95] present a general technique for verifying cache coherence protocols. They use a symbolic representation of the system state keeping track of whether the caches have 0, 1, or multiple copies. We are not convinced that their technique is applicable to the algorithm analyzed in the current paper, because of the doubly linked list. Other cache coherence protocols have been validated in [CGH$^+$95] and in [MS91], using the model checker SMV.

The rest of this paper is organized as follows: In the next section we introduce some basic notions and notation. Both the informal and formal descriptions of the algorithm analyzed in our paper are given in Section 3. The algorithm's formal specification in formulated in Section 4. Section 5 contains a sketch of our proof that the algorithm satisfies its specification. Mechanizing the correctness proof using a theorem prover is currently being pursued. Finally, Section 6 draws some conclusions.

2. Preliminaries

The system considered in this paper consists of a process m called *memory* and a number of processes called *processors* to distinguish them from m. The set of all processors is denoted by $\mathcal{P}$. The term *process* denotes either a processor or memory. Every process has its own identity distinct from the identities of all other processes.

The cache coherence algorithm is modeled in a guarded command language similar to UNITY [CM88]. The program consists of a state formula and a (finite) set of guarded actions. The state formula describes the states in which the program may start its execution.

Our program consists of send- and receive-commands as well as the more conventional statements such as assignments and conditionals. To be more precise, every process p in the system maintains its own message queue $\mathit{buf}[p]$ to record messages which have been sent to, but not yet received by, p. Sending message M from process p to q is achieved by p executing the send-command $\mathit{buf}[q]!M$. This causes message M to be appended to queue $\mathit{buf}[q]$.

As usual in the description of network algorithms, we distinguish between different types of messages. A type is identified with a string of characters. A message of type T and arguments *args* is represented by $T(\mathit{args})$. To allow a receiving process to determine the identity of the sender of a message, the first component of *args* is always the identity of that message's sender. (This restriction could be relaxed. We refrain from doing so because it eases our proof.)

A receive-command is of the form $\mathit{buf}[p]?T(\mathit{args})$. Command $\mathit{buf}[p]?T(\mathit{args})$ can be executed by process p only if $\mathit{buf}[p]$'s first message is of type T in the state of its execution. In this case, we say that the receive-command is *enabled* in that state. Its execution causes process p to receive the first message of the queue, and to delete this message from the queue.

The guard of an action is either a boolean condition or a receive-statement. In case of a boolean condition, we say that guard g is enabled in some state if g evaluates to true in that state.

In the semantics of programs, we use history variable h which can take sequences as values. The empty sequence is denoted by ϵ. Every element in sequence h is of the form

- $\langle \mathit{Snd}, p, M, q\rangle$, to denote that process p has sent message M to process q, or
- $\langle \mathit{Rec}, p, M, q\rangle$, to denote that process q has received message M from process p.

As usual, variable h is updated whenever a send- or receive-command is executed. E.g., if $\mathit{buf}[q]!M$ is executed by process p, then $\langle \mathit{Snd}, p, M, q\rangle$ is appended to h.

Our program always starts in a state satisfying $h = \epsilon$. Let $\mathbf{P}\equiv\langle\Theta, A\rangle$ denote this program, where Θ describes the state in which the program may start its execution, and where A describes the program's set of actions. Let τ denote the idling action [MP91]. A computation sequence of $\mathbf{P}$ is an infinite sequence $s_0 \overset{a_0}{\to} s_1 \overset{a_1}{\to} s_2 \cdots$ of states s_n and actions $a_n \in A \cup \{\tau\}$ $(n \geq 0)$, such that s_0 satisfies formula Θ, and for all $n \geq 0$ the following is satisfied:

- Either some action $a_n \in A$ is enabled in state s_n, and s_{n+1} is the state resulting when a_n is executed in s_n; or no action in state s_n is enabled, $s_n = s_{n+1}$, and $a_n = \tau$.
- Every action in A which is enabled from some point onwards in the sequence is eventually taken (weak fairness [Fra86]).

An obvious property which holds continuously during execution of the program is: The sequence of messages received by process q from process p is a prefix of the sequence of messages sent by p to q. Let $h{\downarrow}(\mathit{Rec}, p, q)$ denote the sequence of messages in sequence h that have been received by process q from process p; it is obtained by projection of h onto elements of the form $\langle \mathit{Rec}, p, T(\mathit{args}), q\rangle$. Similarly, let $h{\downarrow}(\mathit{Snd}, p, q)$ denote the sequence of messages in sequence h that have been sent by process p to process q. The property of the program mentioned above is then expressed by $h{\downarrow}(\mathit{Rec}, p, q) \preceq h{\downarrow}(\mathit{Snd}, p, q)$, where $\preceq$ denotes the usual prefix operator on sequences.

As discussed, message queue $\mathit{buf}[p]$ takes sequences consisting of elements of the form $T(\mathit{args})$ as values. Intuitively, $\mathit{buf}[p]$ is the sequence of messages sent to, but not yet been received by p. Let $buf[p]{\downarrow}q$ denote the sequence of messages in $buf[p]$ of the form $T(q, args')$, i.e., those messages sent by q to p but not yet received by p. (Recall that the first component of a message is the identity of a process.)

For sequences h_1, h_2, let $h_1 \oplus h_2$ denote the sequence obtained by appending h_2 to h_1; and let $h_1 \ominus h_2$ denote the difference between sequences h_1 and h_2, i.e.,

$$\begin{aligned} h_1 \ominus h_2 = & h, \text{ if } h_2 \oplus h = h_1 \\ & \epsilon, \text{ otherwise.} \end{aligned}$$

The following holds continuously during execution of the program: $h{\downarrow}\langle Snd, p, q\rangle \ominus h{\downarrow}\langle Rec, p, q\rangle = buf[q]{\downarrow}p$, i.e., the sequence of messages sent from p to q not yet received by q can be found (in the sames order as sent) in q's buffer. In other words, if some message is in process p's buffer, then that message has been sent to p (hence, recorded in h), and that message has not yet been received by p. Thus, messages sent from one process to another process are received in the same order as sent.

Throughout this paper we use Manna and Pnueli's Linear Time Temporal Logic LTL [MP91]. In particular, we use the temporal operators $\Box$ (always), $\Diamond$ (eventually), O (next), W (weak-until), and U (strong-until). Note that the $\Diamond$-operator can be derived from the $\Box$-operator; and that the U-operator can be derived from the W- and the $\Diamond$-operators. In correctness proofs of programs one usually establishes invariants, i.e., properties which are true throughout computation. LTL offers proof rules to establish such properties. As an example, assume that φ is a state property. The proof rule below, cf. S_INV in [MP91] shows how to prove $\Box\varphi$ for some program P consisting of state formula Θ and set A of guarded actions. Here, φ' denotes some state property.

$$\frac{\Theta \Rightarrow \varphi', \qquad \{\varphi'\}a\{\varphi'\}, \text{ for all } a \in A, \qquad \varphi' \Rightarrow \varphi}{P \vdash \Box\varphi}$$

Thus, a verifier has to formulate some state property φ' stronger than φ in order to apply this rule. The reason is that property φ is, in general, too weak to prove that it is preserved by all actions of the program. Property φ' may have to characterize a large number of additional properties in order to establish φ, as is the case for complicated programs such as the one analyzed in the current paper.

3. Program

We now present the informal and formal descriptions of the program studied in the rest of this paper.

3.1. Informal Description

Memory m maintains its (own) variables cv_m, $status_m$, and $head_m$. Variable cv_m (m's cache value) records the cache from m's point of view. For ease of exposition, we assume that the value of cv_m is always some natural number. The initial value of cv_m is irrelevant.

Variable $status_m$ has initial value *Home*. This variable can take the following values, where the informal explanation is given from m's point of view.

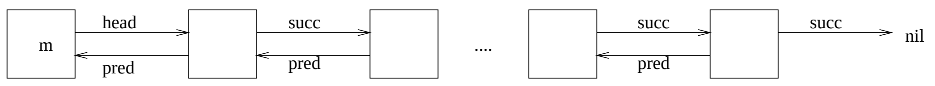

Fig. 1. *In an idealized view, memory and processors which have indicated to read or write the cache form a doubly liked list.*

- *Home*, if no read- or write-queries are in progress,
- *Fresh*, if only read-queries are in progress, or
- *Gone*, if at least one write-query is in progress.

The value of variable $head_m$ is either *nil* or a processor's identity. Value *nil* is different from all such identities; it is the initial value of $head_m$. Intuitively, $head_m$ records the processor to which m has *last* sent a response to a read- or write-query and for which a read- or write-query is in progress. (It is *nil* if no such processor exists.) Roughly, a *read/write query is in progress for a processor* after it has indicated that it wants to read or write until it goes off the doubly linked list mentioned in Section 1. During this period a series of messages is exchanged and the processor might be granted permission to read or write the cache.

Every processor p maintains its (own) variables cv_p, cs_p, $pred_p$, $succ_p$, and $status_p$. Variable cv_p (p's cache value), whose initial value is irrelevant, records the cache from processor p's point of view. Similarly to memory's variable cv_m, it is assumed that the value of cv_p is always a natural number.

To describe the interpretation of variable cs_p (p's cache status), we introduce the notion of the *owner* of the cache: If there are no write-queries in progress, then we say that m is the owner of the cache; otherwise, the processor to which m has *last* sent a response to a read- or write-query and for which a read- or write-query is in progress is the owner of the cache. This description is not precise, but suffices for the informal explanation of the algorithm. The notion of the owner of the cache will be formally defined in Section 4.

Variable cs_p's initial value is *invalid*. This is also its value when p has no read- or write- requests in progress. (In this case, the processor has no interest in the cache value and might have an incorrect value. The processor must reissue a query to get the correct value.) It has value *dirty*, if for some processor, possibly different from p, a write-request is in progress and p is the owner of the cache. It is *fresh*, otherwise, e. g., if p indicates that it wants to read and no other processor wants to modify the cache. As with the notion of the owner of the cache, the description of the intuition behind variable cs_p is again imprecise. E. g., the value of cs_p may be *invalid* if p has made a read- or write-query but is not yet part of the shared list which we introduce below.

The initial value of the variables $pred_p$ and $succ_p$ is *nil*. This is also the value of these variables when no read- or write-request is in progress. When processors issue read- or write-requests (to memory), they will always receive a response back (from memory). Intuitively, when such a request is in progress for processor p, $succ_p$ records the processor q such that the following is true: Prior to p, m has most recently sent a response to q, and a read- or write-request is in progress for q. (It is *nil*, if no such processor exists.) Analogously, when a read- or a write-request is in progress for processor p, $pred_p$ records the *next* processor after p

that received a response from memory and for which a read- or write-request is in progress. (It is m, if no such processor exists.) Thus, in an idealized view, the processors for which there is a read- or write-request in progress form a doubly linked list. For processor p, $succ_p$ identifies the next element in the list (the processor which prior to p issued a read- or a write-query), and $pred_p$ identifies the previous element (the processor which following p issued a read- or a write-query) in the list. Fig. 1 depicts the idealized view. This view may be corrupted because the processes perform their computation concurrently with respect to each other. Following [IEE90] we call this list the *shared list.*

The last variable to be discussed is $status_p$, for processor p. It can take the values:

- *Off* , if no read- or write-request is in progress for processor p.
- *Pending*, if p has issued a read- or a write-request and it is waiting for a response (from memory).
- *Inqueue*, if p has received the response from memory to its read- or write-request and p attempts to prepend to the shared list.
- *Inlist*, if p has succeeded in joining the shared list.
- *Delright*, if p attempts to go off the shared list and notifies the processor identified by $succ_p$ of this.
- *Delleft*, if p attempts to go off the shared list and notifies the processor identified by $pred_p$ of this.
- *Ftod* ($\underline{\text{F}}$resh $\underline{\text{to}}$ $\underline{\text{d}}$irty), if p has a read-query in progress and issues a request to m to modify the cache.
- *Purging*, if p has permission to write and is in the phase of deleting all other processors from the shared list.

We are now ready to discuss the algorithm. We relate the discussion to actions in the formal description of the algorithm given in Section 3.2.

If processor p is in the *Off* state ($status_p = Off$ holds), then it can send a message *read_cache_freshQ*(p) to memory indicating that p wants to read the cache; or a message *read_cache_goneQ*(p) indicating that p wants to modify the cache. Processor p then goes to the *Pending* state waiting for a response from memory. (Cf. the actions labeled p1 and p2 in Section 3.2.)

Thus, a less idealized view of Fig. 1 would consist of the shared list and a set of processors which are trying to get onto the shared list. More precisely, certain processors would form the shared list whereas other processors (“closer” to memory in Fig. 1) constitute a set of processors attempting to append to the shared list.

If memory m receives message *read_cache_freshQ*(p), then it sends a message *read_cache_freshR* as a response to p. This message carries four arguments. The first one is the identity of m; the second one is the processor which will be p’s successor in the shared list (this value is *nil* if the shared list is empty and p will become the only processor in the shared list); the third argument is the value of cv_m; and the fourth argument is either *gone* if m is not the owner of the cache, or *ok* otherwise. Memory also updates its variable $head_m$ (from m’s point of view p is the new head of the shared list). If p is the first processor on the list from m’s point of view, then m goes (from the *Home* state) to the *Fresh* state. (Cf. the action labeled m1 in Section 3.2.)

If memory m receives message *read_cache_goneQ*(p), then it sends a message *read_cache_goneR* back to p. This message also carries four arguments with the same interpretation as the ones in *read_cache_freshR*. As in the case of message *read_cache_freshQ*(p), m updates its variable $head_m$. Finally, m goes to the *Gone* state. (There is at least one write-request in progress.) (Cf. the action labeled m2 in Section 3.2.)

When p receives message *read_cache_freshR*(m, q, cv, arg) it assigns m to $pred_p$. Now, if q is *nil* then p immediately goes to the shared list, and p becomes the only processor in the list. It records the value of m's cache and also records that this is a fresh copy. Otherwise, if q is not *nil*, then p attempts to prepend to the shared list by sending message *prependQ*(p) to q. If $arg = gone$ holds then cs_p remains *invalid* and the proper value of the cache will be transferred to p at a later stage in the computation. This possibility occurs if memory was not the cache owner at the time it responded to p's query. In this case p must get its cache value and cache status from its successor in the shared list later, and may then become the owner of the cache. Additional action is taken only if $arg = ok$ holds. If this is so, then processor p records the value of m's cache and records that it now has a fresh copy of the cache. (Cf. the action labeled p3 in Section 3.2.) In the case of a *read_cache_goneR* message, p also records that it has become the owner of the cache, by assigning value *dirty* to its variable cs_p. (Cf. the action labeled p4 in Section 3.2.)

Upon receipt of message *prependQ*(p), a processor q grants permission to processor p to prepend to the shared list provided that q is in the *Inlist* state, by sending message *prependR*(q, q, ok, cv_q, cs_q) back to p. The first argument is, as for all messages, the identity of the sender; the second argument is the identity of the head of the shared list; the third argument indicates permission to prepend to the shared list. If this permission is granted, then q records that processor p is q's new predecessor. (For this purpose, the variable $pred_q$ is used.) If q was the owner of the cache, then it passes ownership on to p. Processor q then records that it is not the owner of the cache any more (by assigning *fresh* to its variable cs_q). It sends message *prependR*(q, nil, ok, cv_q, cs_q) when q is in the phase of notifying its predecessor that it is going off the shared list, and that the shared list becomes empty. In this case, p can safely prepend. Processor q sends message *prependR*$(q, r, retry, cv_m, cs_m)$ in all other cases to notify p that p cannot prepend (yet) and that it should redirect its request to processor r. Argument r is $succ_p$ if processor q is going off the shared list. Otherwise processor q is not going off the shared list and $r = q$ holds. (Cf. the action labeled (p5) in Section 3.2.)

After processor p has received message *prependR*(q, r, arg, cv, cs), p retries to prepend to the shared list if $arg = retry$ holds. It does so by sending message *prependQ*(p) to processor r. If, on the other hand, $arg = ok$ holds, then p gets onto the shared list and becomes the new head of the list. More precisely, p goes to the *Inlist* state, and records that r, which is either the identity of a processor or *nil*, is its successor. Processor p also assigns the values of cv and cs to cv_p and cs_p, respectively, if cs_p was *invalid*. (Cf. the action labeled p6 in Section 3.2.) (The value of cs_p is *invalid* if memory was not the owner of the cache when it sent its response to p's read- or write-query. In this case q was the owner and has transferred ownership to p.)

In the *Inlist* state, processor p has several possibilities:

(a) It may attempt to modify the cache when it is the owner of the cache. This case occurs if $cs_p = dirty$ holds. Our correctness proof shows that this occurs

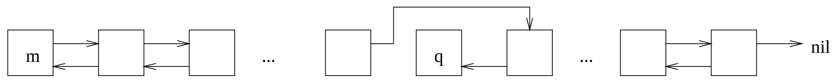

Fig. 2. *When a processor purges its successor q in the shared list, it updates its successor. The new successor will thereafter be purged off the shared list.*

when p is at the head of the shared list. If no other processors are in the shared list, then p simply modifies the cache. If other processors are part of the shared list, then p notifies them to go off the list. Other processors are in the shared list if $\mathit{succ}_p \neq \mathit{nil}$ holds. To purge processors from the list, p sends message $\mathit{purgeQ}(p)$ to its successor in the list. In order to record that p is purging processors, p goes to the *Purging* state. (Cf. the action labeled p7 in Section 3.2.)

A processor q receiving message *purgeQ* records that it is off the shared list by setting both its variables succ_q and pred_q to *nil*. Processor q also sets its variable cs_q to *invalid*. If q is in the *Inlist* state, then it simply goes to the *Off* state. Otherwise, as we will show, processor q has issued some query, e. g., a delright-query, to some other processor, and waits until it has received a response to that query before q goes to the *Off* state. In either case, q sends a message $\mathit{purgeR}(q, r)$ back to processor p. Argument r is the processor that follows q in the shared list if such a processor exists; otherwise, $r = \mathit{nil}$ holds. (Cf. the action labeled p16 in Section 3.2.) We have illustrated this in Fig. 2. Note that this figure demonstrates once again that the view depicted in Fig. 1 is too idealized. Of course, the idealized view serves as a starting point for understanding the complicated nature of the cache coherence algorithm studied in the current paper.

When p receives message $\mathit{purgeR}(q, r)$, it continues purging processor r until it has received a message $\mathit{purgeR}(q', \mathit{nil})$, for some processor q'. This means that the shared list consists only of processor p. In this case, p can safely modify the cache; and p goes back into the *Inlist* state. (Cf. the action labeled p17 in Section 3.2.)

(b) Processor p is at the head of the shared list, and may attempt to modify the cache, even though it is not the owner of the cache. This happens when p has issued a read query before, but now decides that it wants to modify the cache.

From our correctness proof it follows that in this case, $\mathit{cs}_p = \mathit{fresh}$ and $\mathit{pred}_p = m$ holds. Processor p issues a query (to memory) to transfer ownership of the cache to p by sending message $\mathit{modifydataQ}(p)$ to m and going into the *Ftod* state to wait for a response. (Cf. the action labeled p8 in Section 3.2.)

Upon receipt of message $\mathit{modifydataQ}(p)$, memory grants permission to p to modify the cache if p is also the head of the shared list from m's point of view. It does so by sending message $\mathit{modifydataR}(m, \mathit{ok})$ to processor p and going into the *Gone* state. (Now, there exists at least one process which attempts to modify the cache.) If p is not the head of the shared list from m's point of view, then m does not grant permission to modify the cache by sending message $\mathit{modifydataR}(m, \mathit{reject})$ to processor p. (Cf. the action labeled m4 in Section 3.2.)

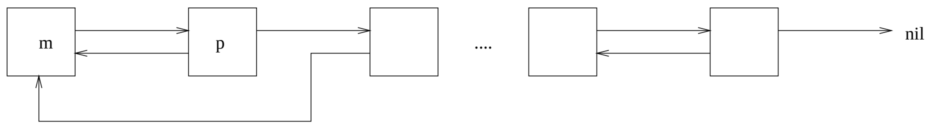

Fig. 3. *When processor p gets a positive response from its successor to p's delright query, p's successor updates its own predecessor in the shared list. In this case m is that predecessor. Thereafter m should update its own variable to point to the head of the shared list.*

When processor p receives response *modifydataR* from memory, p goes back into the *Inlist* state. If it has been granted permission to modify the cache, then p records this by changing its variable cs_p from *fresh* to *dirty*. (Ownership of the cache has been transformed from m to p.) (Cf. the action labeled p9 in Section 3.2.)

(c) Processor p attempts to go off the shared list.
In this case, p has to inform its predecessor and its successor in the shared list (if any) that it is attempting to go off the list. If p has a successor in the shared list, then it sends a message $delrightQ(p, pred_p, cs_p)$ to its successor q and goes into the *Delright* state. This message is to be interpreted as a request of p to q for p to go off the list. (Cf. the action labeled p10 in Section 3.2.)
When q has received message *delrightQ*, it grants p's query, provided that q itself is not waiting for any response due to an outstanding query and provided that q's predecessor is p indeed. Processor q does so by sending message $delrightR(q, ok)$ to p and by recording its new predecessor in the shared list. This case is depicted in Fig. 3. If ownership of the cache has to be passed from p to q, then q also copies the third argument of the *delrightQ* message into its variable cs_q. The query associated with the *delrightQ* message is not granted by q if q is waiting for a response to one of its own queries, or if p is not its predecessor in the shared list (from q's point of view.) In this case, q sends message $delrightR(q, reject)$ to p. (Cf. the action labeled p12 in Section 3.2.)
Now if processor p receives message *delrightR* it may be that p was purged off the list in the meantime. In this case, its variable cs_p will have value *invalid* and it will go directly to the *Off* state. If p has not been purged its behavior is as follows: If p receives a message $delrightR(q, reject)$, then p simply goes back into the *Inlist* state, because no permission had been granted to p to go off the list. If p, on the other hand, receives a message $delrightR(q, ok)$ then p has to inform its predecessor in the shared list that it is going off the list. Informing the predecessor that p is going off the shared list is also immediately done if p has no successors in the list (without going through the *Delright* state). To do so, p sends message $delleftQ(q, succ_p, cv_p)$ to the process (which might be memory) identified by variable $pred_p$, and goes into the *Delleft* state. (Cf. the actions labeled p11 and p13 in Section 3.2.)
To describe the response to message *delleftQ*, we distinguish two cases:

(c1) Message *delleftQ* is received by memory.

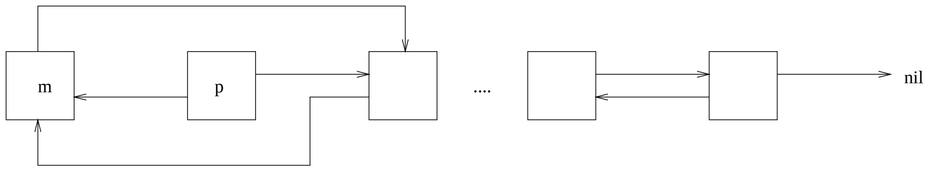

Fig. 4. *This is the situation after memory has updated its variable $head_m$ to record the new head of the shared list. Thereafter processor p will go to the Off state.*

If p is not the head of the shared list from m's point of view, then m sends a message *delleftR*(m, reject) to processor p. This message is not to be interpreted as a rejection to p of m to go off the list, but rather as information that p should retry to send other *delleftQ* messages later because the shared list is being modified.
If p is the head of the list from m's point of view, then m informs p that it can go off the shared list. Memory m does so, by sending a message *delleftR*(m, ok) to p. Memory m then copies the value of the third argument of message *delleftQ* into its variable cv_m (p could have been the owner of the cache and modified it). It records that the processor identified by the second argument of the *delleftQ* message is the new head of the shared list. Note that there is no such processor if this argument is *nil*. In this case, memory m goes back to the *Home* state because no read- or write queries are in progress any more. (Cf. the action labeled m3 in Section 3.2.) Fig. 4 shows how the list of processors looks like after memory positively responds to the delleft-request.

(c2) Message *delleftQ* is received by processor q.
First processor q checks if p is its successor in the shared list. It then also checks if it is either not waiting for a response, is waiting for a *modifydataR* message, or is waiting for a *delrightR* message. (These responses do not cause processor q to change the shared list.) If so, q sends message *delleftR*(q, ok) to processor p to inform p that it can safely go off the list. Processor q also updates its successor in the shared list (by using the second argument of the *delleftQ* message it received). There is no need for processor q to update its variable cv_q because p is not at the head of the shared list, hence not the owner of the cache.
In all other cases, q sends message *delleftR*(q, reject) to p, to inform p to resend the *delleftQ* message later (cf. case (c1) above). (Cf. the action labeled p14 in Section 3.2.)

Upon receipt of message *delleftR*, processor p immediately goes to the *Off* state, if some processor has purged him off the list (i.e., when $cs_p = \mathit{invalid}$), or if p has been informed that it is safe to go off the list (i.e., when $\mathit{arg} = \mathit{ok}$). (Cf. the case of *delrightR* messages.) Otherwise, if p receives message *delleftR*(q, reject), then p retries to go off the list by again sending message *delleftR* to its predecessor (which may be another process than when it first sent that message). (Cf. the action labeled p15 in Section 3.2.)

This completes our informal description of the algorithm.

3.2. Formal Description

As mentioned before, a program consists of an initial condition and a finite collection of actions. We first specify the initial condition, and thereafter the actions.

Initially, no communication has taken place and all the buffers are empty; process m is in the *Home* state and its variable $head_m$ has value *nil*; and every processor is in the *Off* state, its own cache-status is *invalid*, and its forward and backward pointers have value *nil*. Thus, the initial condition is the conjunction of

- $h = \epsilon$,
- $status_m = Home \land buf[m] = \epsilon \land head_m = nil$, and
- for all $p \in \mathcal{P}$, $status_p = Off \land buf[p] = \epsilon \land cs_p = invalid \land succ_p = nil \land pred_p = nil$.

The collection of actions is specified below. There, x:=? denotes the random assignment to model writing an arbitrary value into the cache.

Processor p

```
(p1) $status_p{=}Off \longrightarrow buf[m]!read\_cache\_freshQ(p);\ status_p{:=}Pending$
(p2) $status_p{=}Off \longrightarrow buf[m]!read\_cache\_goneQ(p);\ status_p{:=}Pending$
(p3) $buf[p]?read\_cache\_freshR(q,r,cv,arg) \longrightarrow pred_p{:=}q;$
         if $r{=}nil$
         then $status_p{:=}Inlist;\ cv_p{:=}cv;\ cs_p{:=}fresh$
         else $buf[r]!prependQ(p);\ status_p{:=}Inqueue$
              if $arg = ok$ then $cv_p{:=}cv;\ cs_p{:=}fresh$ fi
         fi
(p4) $buf[p]?read\_cache\_goneR(q,r,cv,arg) \longrightarrow pred_p{:=}q;$
         if $r{=}nil$
         then $status_p{:=}Inlist;\ cv_p{:=}cv;\ cs_p{:=}dirty$
         else $buf[r]!prependQ(p);\ status_p{:=}Inqueue$
              if $arg = ok$ then $cv_p{:=}cv;\ cs_p{:=}dirty$ fi
         fi
(p5) $buf[p]?prependQ(q) \longrightarrow$ if $status_p{=}Inlist$
         then $buf[q]!prependR(p,p,ok,cv_p,cs_p);\ pred_p{:=}q$
              if $cs_p{=}dirty$ then $cs_p{:=}fresh$ fi
         else if $status_p{=}Delleft$
              then if $succ_p{=}nil$
                   then $buf[q]!prependR(p,nil,ok,cv_p,cs_p);$
                        $cs_p{:=}invalid;\ pred_p{:=}nil$
                   else $buf[q]!prependR(p,succ_p,retry,cv_p,cs_p);$
                        $cs_p{:=}invalid;\ pred_p{:=}nil;\ succ_p{:=}nil$
                   fi
              else $buf[q]!prependR(p,p,retry,cv_p,cs_p)$
              fi
         fi
(p6) $buf[p]?prependR(q,r,arg,cv,cs) \longrightarrow$ if $arg = ok$
         then $status_p{:=}Inlist;\ succ_p{:=}r;$
              if $cs_p{=}invalid$ then $cv_p{:=}cv;\ cs_p{:=}cs$ fi
         else $buf[r]!prependQ(p)$
         fi
(p7) $status_p{=}Inlist \land cs_p{=}dirty \longrightarrow$ if $succ_p{\neq}nil$
         then $buf[succ_p]!purgeQ(p);\ status_p{:=}Purging;\ succ_p{:=}nil$
         else $cv_p{:=}?$
```

```
                                          fi
(p8) $status_p{=}Inlist \wedge cs_p{=}fresh \wedge pred_p{=}m \longrightarrow buf\,[m]!modifydataQ(p);\ status_p{:=}Ftod$
(p9) $buf\,[p]?modifydataR(q,arg) \longrightarrow status_p{:=}Inlist;$ if $arg = ok$ then $cs_p{:=}dirty$ fi
(p10) $status_p{=}Inlist \wedge succ_p{\neq}nil \longrightarrow buf\,[succ_p]!delrightQ(p,pred_p,cs_p);\ status_p{:=}Delright$
(p11) $status_p{=}Inlist \wedge succ_p{=}nil \longrightarrow buf\,[pred_p]!delleftQ(p,nil,cv_p);\ status_p{:=}Delleft$
(p12) $buf\,[p]?delrightQ(q,r,cs) \longrightarrow$ if $status_p{=}Inlist \wedge pred_p{=}q$
                                          then $buf\,[q]!delrightR(p,ok);\ pred_p{:=}r;$
                                              if $cs{=}dirty$ then $cs_p{:=}cs$ fi
                                          else $buf\,[q]!delrightR(p,reject)$
                                          fi
(p13) $buf\,[p]?delrightR(q,arg) \longrightarrow$ if $cs_p{=}invalid$
                                         then $status_p{:=}Off$
                                         else if $arg = reject$
                                              then $status_p{:=}Inlist$
                                              else $buf\,[pred_p]!delleftQ(p,succ_p,cv_p);$
                                                   $status_p{:=}Delleft$
                                              fi
                                         fi
(p14) $buf\,[p]?delleftQ(q,r,cv) \longrightarrow$ if   $succ_p{=}q$
                                          $\wedge\ (status_p{=}Inlist \vee status_p{=}Ftod \vee status_p{=}Delright)$
                                        then $buf\,[q]!delleftR(p,ok);\ succ_p{:=}r$
                                        else $buf\,[q]!delleftR(p,reject)$
                                        fi
(p15) $buf\,[p]?delleftR(q,arg) \longrightarrow$ if $cs_p{=}invalid \vee arg = ok$
                                       then $succ_p{:=}nil;\ pred_p{:=}nil;\ cs_p{:=}invalid;\ status_p{:=}Off$
                                       else $buf\,[pred_p]!delleftQ(p,succ_p,cv_p)$
                                       fi
(p16) $buf\,[p]?purgeQ(q) \longrightarrow cs_p{:=}invalid;\ buf\,[q]!purgeR(p,succ_p);\ pred_p{:=}nil;\ succ_p{:=}nil;$
                                   if $status_p{=}Inlist$ then $status_p{:=}Off$ fi
(p17) $buf\,[p]?purgeR(q,r) \longrightarrow$ if $r{=}nil$ then $status_p{:=}Inlist;\ cv_p{:=}?$ else $buf\,[r]!purgeQ(p)$ fi
```

Memory m

```
(m1) $buf\,[m]?read\_cache\_freshQ(p) \longrightarrow$ if $status_m = Gone$
                                             then $buf\,[p]!read\_cache\_freshR(m,head_m,cv_m,gone);$
                                             else $buf\,[p]!read\_cache\_freshR(m,head_m,cv_m,ok)$
                                             fi;
                                             $head_m{:=}p;$
                                             if $status_m{=}Home$ then $status_m{:=}Fresh$ fi
(m2) $buf\,[m]?read\_cache\_goneQ(p) \longrightarrow$ if $status_m{=}Gone$
                                             then $buf\,[p]!read\_cache\_goneR(m,head_m,cv_m,gone)$
                                             else $buf\,[p]!read\_cache\_goneR(m,head_m,cv_m,ok)$
                                             fi;
                                             $head_m{:=}p;\ status_m{:=}Gone$
(m3) $buf\,[m]?delleftQ(p,q,cv) \longrightarrow$ if $head_m{=}p$
                                        then $cv_m{:=}cv;\ buf\,[p]!delleftR(m,ok);\ head_m{:=}q;$
                                            if $q{=}nil$ then $status_m{:=}Home$ fi
                                        else $buf\,[p]!delleftR(m,reject)$
                                        fi
(m4) $buf\,[m]?modifydataQ(p) \longrightarrow$ if $head_m{=}p$
                                        then $buf\,[p]!modifydataR(m,ok);\ status_m{:=}Gone$
                                        else $buf\,[p]!modifydataR(m,reject)$
                                        fi
```

4. Specification

We now present the formal specification of the program in the previous section.

As remarked, every process has its own view of the cache. We stipulated that the value of the cache is the value of the owner of the cache. This is not quite true, however, because it might be that ownership (and hence, the value of the cache) is being transferred from one process to another process. Hence the informal requirement that the processor p with $cs_p = dirty$ is the owner of the cache also needs to be refined in order to ensure the obviously desired property that at any time during computation exactly one process is the owner of the cache.

First we formally define the notion of the owner of the cache. The owner is m, if m is in the *Home*- or *Fresh*-state. Otherwise, it is either processor p for which $cs_p = dirty$ holds and which has not been granted permission to go off the shared list; or it is the processor to which ownership of the cache is being transferred. A processor with $cs_p = dirty$ is granted permission to go off the shared list, if it receives message $delrightR(q, ok)$ from some process q, or if it has no successor in the shared list and receives message $delleftR(q, ok)$ from some process q. (If p is in the *Delleft*-state and has a successor, then p has been in the *Delright*-state before and received message $delrightR(q, ok)$ from its successor q.) Ownership is transferred from one process to another through a message if that message causes the process to go into a state with $cs_p = dirty$. This can happen when one of the following messages is in transit: $read\_cache\_goneR(m, r, cv, arg)$ with $(r = nil \lor arg = ok)$, $prependR(q, r, ok, cv, dirty)$, $modifydataR(m, ok)$. The formal definition of the owner of the cache is given next. Our correctness proof we shows that at any time during computation there exists exactly one owner of the cache. Therefore, if a processor is the owner then $status_m = Gone$ holds.

Hereafter, we often omit types of data in formal definitions whenever immaterial. Also, all free variables in a formula are assumed to be universally quantified.

Definition 4.1.

$$cache\_owner = \begin{cases} m, \; if & status_m = Home \lor status_m = Fresh \\ p, \; if & p \in \mathcal{P} \\ & \land \left(\begin{array}{l} cs_p = dirty \land status_p \neq Delleft \\ \qquad \land \neg \exists q. delrightR(q, ok) \in buf[p] \\ \lor \; cs_p = dirty \land status_p = Delleft \land succ_p = nil \\ \qquad \land \neg \exists q. delleftR(q, ok) \in buf[p] \\ \lor \; \exists r, cv, arg.\; read\_cache\_goneR(m, r, cv, arg) \in buf[p] \\ \qquad \land (r = nil \lor arg = ok) \\ \lor \; \exists q, r. prependR(q, r, ok, cv, dirty) \in buf[p] \\ \lor \; modifydataR(m, ok) \in buf[p] \end{array} \right) \end{cases}$$

∎

The value of the cache is the value of the cache owner's copy of the cache if the owner's cache status has value *dirty*. If ownership is being transferred to a process by means of a message, then that message carries the value of the cache as an argument, except for message $modifydataR(m, ok)$. The latter case is the only time that a processor p with $cs_p = fresh$ is granted permission to modify the cache, and we define the value of the cache by cs_p. The correctness proof shows that before the cache value is modified, cv_p is the same as cv_m (m is the previous owner of the cache).

Definition 4.2.

$$cache\_value = \begin{cases} cv_m, & \text{if } cache\_owner = m \\ cv_p, & \text{if } p \in \mathcal{P} \wedge cs_p = dirty \wedge status_p \neq Delleft \\ & \qquad \wedge \neg \exists q.delrightR(q, ok) \in buf[p] \\ cv_p, & \text{if } p \in \mathcal{P} \wedge cs_p = dirty \wedge status_p = Delleft \wedge succ_p = nil \\ & \qquad \wedge \neg \exists q.delleftR(q, ok) \in buf[p] \\ cv, & \text{if } \exists p, q, r.\ \ read\_cache\_goneR(m, r, cv, arg) \in buf[p] \\ & \qquad \wedge (r = nil \vee arg = ok) \\ & \quad \vee\ prependR(q, r, ok, cv, dirty) \in buf[p] \\ cv_p, & \text{if } p \in \mathcal{P} \wedge modifydataR(m, ok) \in buf[p] \end{cases}$$

∎

We say that a processor is *idle*, if it is either in the *Off* state or if it has sent a read- or write-query that has not yet been received by memory; a processor is *entering* if memory has received the read- or write-query and the processor is in the *Pending*- or the *Inqueue*-state; a processor is *leaving*, if it is about to go off the list, more precisely, if the processor is in the *Delleft*- or *Delright*-state and it has either been purged by another processor or a message *delleftR*(q, ok) has been sent to that processor; finally, a processor which is not *idle*, not *entering*, and not *leaving*, is called *visiting*. A processor is called *staying* if it is *visiting* and it is has not been granted any permission to go off the list.

Definition 4.3. For processors $p \in \mathcal{P}$, define

(a) *idle*(p), if $status_p{=}Off \ \vee\ read\_cache\_freshQ(p){\in}buf[m]$
$\vee\ read\_cache\_goneQ(p){\in}buf[m]$.

entering(p), if $\neg idle(p) \wedge (status_p{=}Pending \vee status_p{=}Inqueue)$.

leaving(p), if $(status_p{=}Delleft \vee status_p{=}Delright)$
$\wedge\ (cs_p{=}invalid \vee \exists q.delleftR(q, ok) \in buf[p])$.

visiting(p), otherwise.

(b) $staying(p) \equiv visiting(p) \wedge status_p{\neq}Delleft \wedge \neg\exists q.delrightR(q, ok) \in buf[p]$.

∎

A process p is said to have a consistent view of the cache if $cv_p = cache\_value$ holds. We require that during computation there always exists a unique owner of the cache, that *staying* processors always have a consistent view of the cache, and that only the owner of the cache can modify the cache. We also require that the owner of the cache will eventually have a proper copy of the cache, and that a processor which is in the *Purging*-state will eventually be able to modify the cache. The latter occurs if a process receives a message *purgeR* and it goes into the *Inlist*-state. We cannot prove that processors which have indicated that they want to modify the cache will eventually do so, because this property is not true. (Such processors may be purged off the list when another processor has become the owner.) Also, processors that indicated that they want to read only might later get permission to write. This can be avoided by maintaining an additional variable for every processor indicating whether it issued a read- or a write query. We have abstracted away from this in the model of our paper. The discussion above leads to the following formal specification of the program:

Definition 4.4. The following is required to hold continuously during computation of the program:

(a) $\exists! p.(p \in \mathcal{P} \cup \{m\} \;\wedge\; \mathit{cache\text{-}owner} = p)$.
(There exists always exactly one owner of the cache.)

(b) $\forall p \in \mathcal{P}.(\mathit{staying}(p) \;\Rightarrow\; cv_p = \mathit{cache\_value})$.
(*Staying* processors have a consistent view of the cache.)

(c) $$\begin{aligned} \mathit{cache\_value} \neq O(\mathit{cache\_value}) & \\ \Rightarrow \quad & \mathit{cache\_owner} \in \mathcal{P} \wedge cv_{\mathit{cache\_owner}} = \mathit{cache\_value} \\ & \wedge\, \mathit{cache\_owner} = O(\mathit{cache\_owner}) \\ & \wedge\, O(cv_{\mathit{cache\_owner}}) = O(\mathit{cache\_value}). \end{aligned}$$
(Only a processor which is the owner can modify the cache value.)

(d) $(\mathit{cache\text{-}owner} = p)\; U\; (\mathit{cache\text{-}owner} = p \wedge cv_p = \mathit{cache\_value})$.
(The owner of the cache eventually has a proper copy of the cache.)

(e) $$\begin{aligned} & [(p \in \mathcal{P} \wedge \mathit{cache\_owner}{=}p \wedge \mathit{status}_p{=}\mathit{Purging}) \\ & \qquad\qquad U \\ & (\mathit{cache\_owner}{=}p \wedge \mathit{status}_p{=}\mathit{Purging} \wedge \exists q.\mathit{first}(\mathit{buf}[p]){=}\mathit{purgeR}(q, \mathit{nil}))] \\ \wedge\; & (\mathit{first}(\mathit{buf}[p]){=}\mathit{purgeR}(q', \mathit{nil}))\; U\; (\mathit{status}_p = \mathit{Inlist} \;\wedge\; \mathit{cache\text{-}owner} = p). \end{aligned}$$
(A processor in the *Purging*-state eventually receives a *purge* response and goes into the *Inlist*-state from which it can modify the cache. See the program text.) ∎

5. Correctness Proof

We now describe how we have shown that the program in Section 3.2 satisfies the specification formulated in Definition 4.4. A detailed proof is presented in [FS99].

5.1. Invariants

In this subsection we list a number of properties which continuously hold during execution of the program. Some of these properties deal with types; some other properties are formulated in order to show that there are no *unspecified receipts*. (For every process, if it can receive a message then it can execute at least one action which deals with that message.) The invariants are also used to establish that the program satisfies its specification.

Every message always carries the identity of the sender, a process, as the first component of the message's argument:

Lemma 5.1. The following properties continuously hold during execution of the program:

(a) $$(\langle \mathit{Snd}, p, T(p', \mathit{arg}), q\rangle \in h \vee \langle \mathit{Rec}, p, T(p', \mathit{arg}), q\rangle \in h) \Rightarrow (\quad p \in \mathcal{P} \cup \{m\} \wedge p = p').$$

(b) $T(p, \mathit{arg}) \in \mathit{buf}[q] \Rightarrow p \in \mathcal{P} \cup \{m\}$. ∎

The proofs of this property and of some properties formulated hereafter depend on general properties of the semantics, such as $\mathit{msg}(p, \mathit{arg}) \in \mathit{buf}[q] \Rightarrow \mathit{msg}(q, \mathit{arg}) \in h{\downarrow}\langle \mathit{Snd}, p, q\rangle \ominus h{\downarrow}\langle \mathit{Rec}, p, q\rangle$ (see Section 2). We omit most proofs in this paper; they can all be established using the techniques described in [MP91]

We have that queries are only sent by processors (and never by memory):

Lemma 5.2. For all message types msgQ, the following continuously holds during execution of the program:

$$\left(\begin{array}{ll} & (\langle Snd, p, msgQ(p, arg), q\rangle \in h \\ \vee & \langle Rec, p, msgQ(p, arg), q\rangle \in h \\ \vee & msgQ(p, arg) \in buf[q]) \end{array}\right) \Rightarrow p \in \mathcal{P}. \quad \blacksquare$$

Read-, write-, and modifydata-queries are only sent to memory:

Lemma 5.3. The following continuously holds during execution of the program:

$$\left(\begin{array}{ll} & \langle Snd, p, read\_cache\_freshQ(p), q\rangle \in h \\ \vee & \langle Snd, p, read\_cache\_goneQ(p), q\rangle \in h \\ \vee & \langle Snd, p, modifydataQ(p), q\rangle \in h \\ \vee & \langle Rec, p, read\_cache\_freshQ(p), q\rangle \in h \\ \vee & \langle Rec, p, read\_cache\_goneQ(p), q\rangle \in h \\ \vee & \langle Rec, p, modifydataQ(p), q\rangle \in h \\ \vee & read\_cache\_freshQ(p) \in buf[q] \\ \vee & read\_cache\_goneQ(p) \in buf[q] \\ \vee & modifydataQ(p) \in buf[q] \end{array}\right) \Rightarrow q = m. \quad \blacksquare$$

This lemma implies that there are no unspecified receipts for processors.

Read-, write-, and modifydata-responses are only sent by memory and to processors. This property as well as a number of other ones, which are needed to establish it, are formulated in the next lemma.

Lemma 5.4. The following continuously holds during execution of the program:

(a) $$\left(\begin{array}{ll} & \langle Snd, p, read\_cache\_freshR(p, r, cv, arg), q\rangle \in h \\ \vee & \langle Snd, p, read\_cache\_goneR(p, r, cv, arg), q\rangle \in h \\ \vee & \langle Rec, p, read\_cache\_freshR(p, r, cv, arg), q\rangle \in h \\ \vee & \langle Rec, p, read\_cache\_goneR(p, r, cv, arg), q\rangle \in h \\ \vee & read\_cache\_freshR(p, r, cv, arg) \in buf[q] \\ \vee & read\_cache\_goneR(p, r, cv, arg) \in buf[q] \end{array}\right)$$
$$\Rightarrow (p = m \wedge\ q \in \mathcal{P} \wedge\ (r = nil\ \vee\ r \in \mathcal{P}) \wedge\ (arg = ok \vee\ arg = gone)).$$

(b) $$\left(\begin{array}{ll} & \langle Snd, p, modifydataR(p, arg), q\rangle \in h \\ \vee & \langle Rec, p, modifydataR(p, arg), q\rangle \in h \\ \vee & modifydataR(p, arg) \in buf[q] \end{array}\right)$$
$$\Rightarrow (p = m \wedge q \in \mathcal{P} \wedge\ (arg = ok \vee\ arg = reject)).$$

(c) $head_m{=}nil \vee head_m{\in} \mathcal{P}$.

(d) For all $p \in \mathcal{P}$, $succ_p{=}nil \vee succ_p{\in} \mathcal{P}$.

(e) $$\left(\begin{array}{ll} & \langle Snd, p, prependR(p, r, arg, cv, cs), q\rangle \in h \\ \vee & \langle Rec, p, prependR(p, r, arg, cv, cs), q\rangle \in h \\ \vee & prependR(p, r, arg, cv, cs) \in buf[q] \end{array}\right)$$
$$\Rightarrow \left(\begin{array}{l} \quad p \in \mathcal{P}\ \wedge\ q \in \mathcal{P} \\ \wedge\ ((\ arg = ok \wedge (p = r \vee r = nil)) \\ \qquad \vee\ (arg = retry \wedge r \in \mathcal{P})). \end{array}\right)$$

(f) $$\left(\begin{array}{ll} & \langle Snd, p, delleftQ(p, r, cv), q\rangle \in h \\ \vee & \langle Rec, p, delleftQ(p, r, cv), q\rangle \in h \\ \vee & delleftQ(p, r, cv) \in buf[q] \end{array}\right) \Rightarrow\ (r = nil\ \vee r \in \mathcal{P}). \quad \blacksquare$$

It follows that, for all processors $p \in \mathcal{P}$, $succ_p \neq m$ holds.

We next show the values that some of the other variables can take:

Lemma 5.5. The following continuously holds during execution of the program:

(a) $status_m = Home \vee status_m = Fresh \vee status_m = Gone$.

(b) $status_m = Home \Leftrightarrow head_m = nil$.

(c) For all processors p, $status_p = Off \vee status_p = Pending \vee status_p = Inqueue \vee status_p = Inlist \vee status_p = Delleft \vee status_p = Delright \vee status_p = Ftod \vee status_p = Purging$. ∎

Prepend- and delright-queries are sent only to processors (and never to memory); and the value of $pred_p$, for processors p, is either nil or in set $\mathcal{P} \cup \{m\}$:

Lemma 5.6. The following continuously holds during execution of the program:

(a) $\left(\begin{array}{ll} & \langle Snd, p, prependQ(p), q\rangle \in h \\ \vee & \langle Rec, p, prependQ(p), q\rangle \in h \\ \vee & prependQ(p) \in buf[q] \end{array} \right) \Rightarrow q \in \mathcal{P}$.

(b) $\left(\begin{array}{ll} & \langle Snd, p, delrightQ(p,r,cv), q\rangle \in h \\ \vee & \langle Rec, p, delrightQ(p,r,cv), q\rangle \in h \\ \vee & delrightQ(p,r,cv) \in buf[q] \end{array} \right) \Rightarrow q \in \mathcal{P} \wedge (r = nil \vee r \in \mathcal{P} \cup \{m\})$.

(c) $pred_p = nil \vee pred_p \in \mathcal{P} \cup \{m\}$. ∎

Purge-queries and purge-responses are sent by processors to processors:

Lemma 5.7. The following continuously holds during execution of the program:

(a) $\left(\begin{array}{ll} & \langle Snd, p, purgeQ(p), q\rangle \in h \\ \vee & \langle Rec, p, purgeQ(p), q\rangle \in h \\ \vee & purgeQ(p) \in buf[q] \end{array} \right) \Rightarrow q \in \mathcal{P}$.

(b) $\left(\begin{array}{ll} & \langle Snd, p, purgeR(p,r), q\rangle \in h \\ \vee & \langle Rec, p, purgeR(p,r), q\rangle \in h \\ \vee & purgeR(p,r) \in buf[q] \end{array} \right) \Rightarrow p \in \mathcal{P} \wedge q \in \mathcal{P} \wedge (r = nil \vee r \in \mathcal{P})$.

∎

Delleft-responses are always sent to processors (never to memory); the second argument of the response is either ok or $reject$:

Lemma 5.8. The following continuously holds during execution of the program:

$$\left(\begin{array}{ll} & \langle Snd, p, delleftR(p,arg), q\rangle \in h \\ \vee & \langle Rec, p, delleftR(p,arg), q\rangle \in h \\ \vee & delleftR(p,arg) \in buf[q] \end{array} \right) \Rightarrow q \in \mathcal{P} \wedge (arg = ok \vee arg = reject).$$

∎

It follows from the Lemmata 5.4, 5.6, 5.7, and 5.8 that there are no unspecified receipts for m. In particular, m will never receive a message of the form $msgR(arg)$, i.e., one associated with a response.

An occurrence of message $msgQ$ is outstanding for processor p, if p has sent $msgQ$ to some process and not received message $msgR$ thereafter.

Definition 5.1.

(a) $Out(msgQ, p, i) \equiv$
$$\begin{aligned} & 0 < i \leq |h| \\ \wedge\ & \exists q \in \mathcal{P} \cup \{\mathrm{m}\}. \exists arg. h[i] = \langle Snd, p, msgQ(arg), q\rangle \\ \wedge\ & \forall q' \in \mathcal{P} \cup \{m\}. \forall arg'. \forall j.\ (i<j\leq |h| \Rightarrow h[j] \neq \langle Rec, q', msgR(arg'), p\rangle). \end{aligned}$$

(b) $outstanding(msgQ, p) \equiv \exists i.\ Out(msgQ, p, i)$. ▮

If $\neg outstanding(msgQ, p) \vee \exists! msg. \exists! i.\ Out(msgQ, p, i)$ holds, we say that there exists at most one outstanding query for processor p.

Hereafter, the operator $\triangledown$ denotes the "exclusive-or" operator, i.e., $A \triangledown B$ holds iff either A or B, but not both, holds. We now arrive at the first key invariant:

Lemma 5.9. The following continuously holds during execution of the program:

(a) Every processor has at most one outstanding query.

(b) For every processor p,
$status_p = Off \Rightarrow p$ has no outstanding queries.
$status_p = Pending \Rightarrow p$ has an outstanding read- or write query.
$status_p = Inqueue \Rightarrow p$ has an outstanding prepend query.
$status_p = Inlist \Rightarrow p$ has no outstanding queries.
$status_p = Delleft \Rightarrow p$ has an outstanding delleft query.
$status_p = Delright \Rightarrow p$ has an outstanding delright query.
$status_p = Purging \Rightarrow p$ has an outstanding purge query.
$status_p = Ftod \Rightarrow p$ has an outstanding modifydata query.

(c) $$\begin{aligned} h[i] = \langle Snd, p, msgR(arg), q\rangle \Rightarrow \exists j. \exists arg'.(\ & 1 \leq j < i \wedge h[j] = \langle Rec, q, msgQ(arg'), p\rangle \\ & \wedge \forall k. \forall arg''.(j<k<i \Rightarrow h[k] \neq \langle Snd, p, msgR(arg''), q\rangle)). \end{aligned}$$
(If p responds to process q, then there has been a request of q to p, and p has not responded to that request before.)

(d) $$\begin{aligned} Out(msgQ, p, i) \Leftrightarrow \exists q \in \mathcal{P} \cup \{m\}.\ (\ & \exists arg. msgQ(p, arg) \in buf[q] \\ & \triangledown \exists j. \exists arg'.\ (\ i < j \leq |h| \\ & \qquad \wedge h[j] = \langle Snd, q, msgR(arg'), p\rangle \\ & \qquad \wedge msgR(arg') \in buf[p])). \end{aligned}$$
(A process has an outstanding query iff either that query is in transit or p's buffer contains a response to that query.) ▮

Lemma 5.10. For every processor p, the following continuously holds during execution of the program:

(a) $$\begin{aligned} & (status_p = Delright \wedge cs_p \neq invalid \wedge pred_p = z) \\ & \qquad W \\ & (\ (status_p = Delright \wedge cs_p = invalid) \\ & \ \vee ([status_p = Inlist \vee status_p = Delleft] \wedge cs_p \neq invalid \wedge pred_p = z)). \end{aligned}$$

(b) $(status_p = Delright \wedge cs_p = cs \wedge cs \neq invalid \;\wedge\; delrightR(q, ok) \in buf[p])$
$$W$$
$((status_p = Delright \;\wedge\; cs_p = invalid) \vee (status_p = Delleft \;\wedge\; cs_p = cs))$.

(c) $(status_p = Delright \;\wedge cs_p = invalid)\ W\ status_p = Off$.

(d) $(status_p = Delleft \wedge pred_p = z_1 \;\wedge\; succ_p = z_2)$
$$W$$
$(cs_p = invalid \wedge (status_p = Delleft \vee status_p = Off))$.

(e) $(status_p = Delleft \wedge delleftR(q, ok) \in buf[p])\ W\ status_p = Off$.

(f) $(status_p = Delleft \;\wedge\; cs_p = invalid)\ W\ status_p = Off$. ∎

Recall that we have introduced the notions of a process being *idle*, *entering*, and *visiting* (see Definition 4.3). We have:

Lemma 5.11. For every processor p, the following continuously holds during execution of the program:

(a) $idle(p)\ W\ entering(p)$.

(b) $visiting(p)\ W\ (leaving(p) \vee status_p = Off)$.

(c) $leaving(p)\ W\ status_p = Off$. ∎

Let us call a processor *active* if it is either *entering* or *visiting*. By $active(p)$ we denote that processor p is active. We next assign ranks to *active* processors according to the order in which read and write queries are received by m. First we define an auxiliary function:

Definition 5.2. For processors p and natural numbers n define,

$$\begin{array}{l} Last\_activated(p) = n, \\ \text{if} \quad active(p) \\ \quad \wedge\ (\quad h[n] = \langle Rec, p, read\_cache\_freshQ(p), m\rangle \\ \qquad \vee\ h[n] = \langle Rec, p, read\_cache\_goneQ(p), m\rangle) \\ \quad \wedge\ \forall i.n < i \leq |h|.(\quad h[i] \neq \langle Rec, p, read\_cache\_freshQ(p), m\rangle \\ \qquad\qquad\qquad\qquad \vee\ h[i] \neq \langle Rec, p, read\_cache\_goneQ(p), m\rangle). \end{array}$$ ∎

Definition 5.3. For processors p such that $active(p)$ holds, define

$$\begin{array}{l} rank(p) = 0, \text{ if} \quad \exists n.Last\_activated(p) = n \\ \qquad\qquad \wedge \forall m.\forall q \in \mathcal{P}.(q \neq p \wedge active(q) \wedge Last\_activated(q) = m) \Rightarrow m > n. \\ rank(p) = n + 1, \text{ if } \exists q \in \mathcal{P} \quad active(q) \wedge rank(q) = n \\ \qquad\qquad\qquad \wedge\ Last\_activated(q) < Last\_activated(p) \\ \qquad\qquad\qquad \wedge\ \neg r \in \mathcal{P}. \quad active(r) \\ \qquad\qquad\qquad\qquad \wedge\ Last\_activated(q) < Last\_activated(r) \\ \qquad\qquad\qquad\qquad \wedge\ Last\_activated(r) < Last\_activated(p). \end{array}$$ ∎

We then have the following properties:

Lemma 5.12. For every processor p, q, the following continuously holds during execution of the program:

(a) $active(p) \Rightarrow \exists n.Last\_activated(p) = n$.

(b) $(p \neq q \wedge active(p) \wedge active(q)) \Rightarrow rank(p) \neq rank(q)$.

(c) $(active(p) \wedge rank(p) = n) W (\neg active(p) \vee rank(p) < n)$.

(d) $(active(p) \wedge rank(p) = n) \Rightarrow \forall m < n . \exists p' \in \mathcal{P} . (active(p') \wedge rank(p') = m)$. ▮

There are two lemmata which are critical for our correctness proof. They show various properties including how messages sent from one processor to another relate to the ranks of those processors. In both these lemmata we have formulated invariants of the program which hold under certain assumptions. This has been done to reduce the size of the lemmata. (Without these assumptions, the invariants cannot be proved.) The assumptions are discharged later. The lemmata depend on the property that communication is reliable and that the order of messages sent by one process to another is preserved. (There can be two messages from one process in some other process's buffer.) The two key lemmata demonstrate the phenomenon explained at the end of Section 2: In order to establish some invariant of the program, we have to prove a stronger property of that program. We have tried to break up these lemmata into smaller ones, but have not succeeded in doing so. One of the lemma consists of 17 clauses; the other one consists of 7 clauses. We believe that all the clauses in the lemmata are mutually dependent and that none of these clauses can be omitted. This observation is further supported by our mechanical verification effort of this correctness proof. The theorem prover Nuprl [C$^+$86] is now being employed in an ongoing project to mechanize the proof reported in the current paper. So far, we have not discovered any independent clauses which could have then be removed from the lemmata. We mention our work using the theorem prover in Section 6.

In essence, some of the clauses in the lemmata are concerned with characterizing the the structure of nodes when they are on the shared list. The idea is that the head of shared list can be reached through pointer $head_m$. Processors on the shared list can be reached by following the succ pointers. The notion of rank is employed to prove that the shared list will never contain any cycles. The invariant expressing these properties is not immediately provable, but requires establishing a stronger invariant as noted in Section 1. Thus, we have added additional clauses to do so, such as one clause to cope with the situation that some processor may become part of the shared list. This approach also demonstrates the accumulative process for finding *provable* properties, because the addition of one clause may generate the additions of other clauses to ensure that all the added clauses are provable.

After having proved the two lemmata mentioned above, we have a lemma which combines the invariants proved under certain assumptions into another invariant. At this stage during the proof we also discharge the assumptions under which these invariants were derived. Thereafter, we are ready to show that the program is correct w.r.t. its specification:

Theorem 5.1. The program satisfies its specification.

6. Conclusion

The SCI protocol is an IEEE standard for specifying communication between multiprocessors in a shared memory model. In this paper we have considered the cache coherence portion of this protocol. We have modeled and sketched correctness of an abstraction of this portion. For example, we have not kept track of processors which want to read only (and not write) and we have considered the problem with one cache line only. (Multiple cache lines require a straightforward extension of the proof.) Also, we have used only three values for the cache status of a process, whereas in the full protocol more values are employed. We have presented a specification of our model and a proof sketch that the model meets this specification. The correctness proof has been carried out within Linear Time Temporal Logic and can be found in [FS99].

Our proof has been carried out by pen and paper. We realize that hand-written proofs may contain errors. For this reason we are now in the process of mechanizing our whole proof. This work is done jointly with Doug Howe using the theorem prover Nuprl. Another reason to advocate the use of mechanical tools to support human reasoning became evident when doing the correctness proof. Two lemmata are rather tedious to prove. Both these lemmata consists of a large number of clauses of which it has to be shown that each of them is an invariant. The correctness of a clause depends on several clauses which are defined later in the lemma. When one of the clauses turns out to be invalid (as has happened quite frequently when formulating the lemma), all previously verified clauses need to be reproved because they might depend on the modified one. A tool which could keep track of such dependencies or which could redo the proof would be of great help. We are convinced that such tools are even essential if such proofs are carried out on a regular basis.

We have used assumptions in lemmata in order to structure the correctness proof. These assumptions have been discharged at a later stage in the proof. In contrast to compositional appoaches, our assumptions may refer to global properties. We believe that our approach is worth further research, since it allows more transparent formulations of properties and structuring their proofs. This may have an impact on reducing complexity of automated proofs.

With Doug Howe we are currently mechanizing the correctness proof reported on in the current paper. This is an ongoing project and results about our mechanization, employing the theorem prover Nuprl [C^+86], can be found in [FHS98].

In previous work [BFS95], with Ramesh Bharadwaj, we have investigated how to combine model checking and theorem proving to verify a broadcasting protocol. The work reported in the current paper serves as a foundation for a case study to push the limits of formal verification by means of tools to really large programs, in particular programs which cannot be validated by model checking techniques (only). In the future we will try to mechanically verify even larger programs.

Acknowledgements

Thanks are due to David Long for discussing the model in Section 3. His original model has served as a basis for both our model as well for the models studied by Holzmann, Kurshan, and McMillan. We also thank Thor Jeremiassen and Michael Merritt for suggestions on improving the presentation.

References

[ABM93] Yehuda Afek, Geoffrey Brown, and Michael Merritt. Lazy caching. *ACM Transactions on Programming Languages and Systems*, 15(1):182–205, January 1993.

[BFS95] Ramesh Bharadwaj, Amy Felty, and Frank Stomp. Formalizing inductive proofs of network algorithms. In *Proceedings of the 1995 Asian Computing Science Conference (Lecture Notes in Computer Science 1023)*, December 1995.

[Bri95] Ed Brinksma. Cache consistency by design. Manuscript, 1995.

[C$^+$86] Robert L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, 1986.

[CF78] L. M. Censier and P. Feautrier. A new solution to coherence problems in multicache systems. *IEEE Transactions on Computers*, 27(12):1112–1118, December 1978.

[CGH$^+$95] Edmund M. Clarke, Orna Grumberg, Hiromi Hirashi, Somesh Jha, David E. Long, Kenneth L. McMillan, and Linda A. Ness. Verification of the futurebus+cache coherence protocol. *Formal Methods in Systems Design*, 6:217–232, 1995.

[CM88] K. M. Chandy and J. Misra. *Parallel Program Design—A Foundation.* Addison-Wesley, 1988.

[FHS98] Amy Felty, Douglas Howe, and Frank Stomp. Protocol verification in nuprl. In *Tenth International Workshop on Computer-Aided Verification (Lecture Notes in Computer Science 1427)*, pages 428–439. Springer-Verlag, 1998.

[Fra86] Nissim Francez. *Fairness.* Springer-Verlag, 1986.

[FS96] Amy Felty and Frank Stomp. A correctness proof of a cache coherence protocol. In *Eleventh Annual Conference on Computer Assurance*, pages 128–141. IEEE, 1996.

[FS99] Amy Felty and Frank Stomp. Specification and correctness of cache coherency in SCI. *Formal Aspects of Computing*, 11(E), 1999.

[Ger95] Rob Gerth. Sequential consistency as interface refinement. Manuscript, 1995.

[GKMK91] Stein Gjessing, Stein Krogdahl, and Ellen Munthe-Kaas. A top-down approach to the formal specification of SCI cache coherence. In *Proceedings of the 3th International Workshop on Computer-Aided Verification*, pages 83–91. Springer Verlag Lecture Notes in Computer Science 697, 1991.

[GMK91] Stein Gjessing and Ellen Munthe-Kaas. Formal specification of cache coherence in a shared memory multiprocessor. Technical Report 158, Department of Informatics, University of Oslo, 1991.

[Gra95] Susanne Graf. Characterization of a sequential consistent memory and verification of a cache memory by abstraction. Manuscript, 1995.

[Gus92] D. B. Gustavsson. The scalable coherent interface and related standard projects. *IEEE Micro*, 12(1):10–22, 1992.

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

[Hol91] Gerard J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall Software Series, 1991.

[Hol95] Gerard Holzmann, 1995. Personal Communication.

[IEE90] IEEE-P1596-05Nov90-doc197-iii. *Part IIIA: SCI Coherence Overview*, 1990. Unapproved Draft. Approved standard is described in IEEE Std. 1596-1992 "The Scalable Coherent Interface".

[IEE92] IEEE Computer Society. *IEEE Standard for Futurebus+–Logical Protocal Specification*, March 1992. IEEE Standard 896.1-1991.

[JPR95] Bengt Jonsson, Amir Pnueli, and Camilla Rump. Proving refinement using transduction. Manuscript, 1995.

[JPZ95] Wil Janssen, Mannes Poel, and Job Zwiers. The compositional approach to sequential consistency and lazy caching. Manuscript, 1995.

[Kat95] Shmuel Katz. Sequential consistency using global proofs and temporal logic. Manuscript, 1995.

[KP87] S. Katz and D. Peled. Interleaving set temporal logic. In *Proceedings of the 6th ACM Symposium on Distributed Computing*, pages 178–190, 1987.

[Kur89] R. P. Kurshan. Analysis of discrete event coordination. In *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness (REX Workshop)*, pages 414–453. Springer Verlag Lecture Notes in Computer Science 430, 1989.

[Kur95] Robert Kurshan, 1995. Personal Communication.

[Lam79] L. Lamport. How to make a multiprocessor that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, September 1979.

[Lam94] Leslie Lamport. The temporal logic of actions. *ACM Transctions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[LD95] Gavin Lowe and Jim Davies. Using CSP to verify sequential consistency. Manuscript, 1995.

[LLOR95] Peter Ladkin, Leslie Lamport, Bryan Olivier, and Denis Roegel. Lazy caching in LTA. Manuscript, 1995.

[LM95] David Long and Ken McMillan, 1995. Personal Communication.

[McM93] K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

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

[MS91] K. L. McMillan and J. Schwalbe. Formal verification of the encore gigamax cache consistency protocol. In *Proceedings of the 1991 International Symposium on Shared Memory Multiprocessors*, pages 164–177, April 1991.

[PD95] Fong Pong and Michael Dubois. A New Approach for the Verification of Cache-Coherence Protocols. *IEEE Transactions on Parallel and Distributed Systems*, 6(8), 1995.

[PH96] David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantative Approach.* Morgan Kaufmann Publishers, Inc, 1996. Second Edition.

[SD87] C. Schreurich and M. Dubois. Correct memory operation of cache-based multiprocessors. In *14th International Symposium on Computer Architecture*, pages 234–243, Los Angeles, 1987. IEEE Computer Society.

[SD95] Ulrich Stern and David L. Dill. Automatic verification of the SCI cache coherence protocol. In *Correct Hardware Design and Verification Methods: IFIP WG10.5 Advanced Research Working Conference Proceedings*, 1995.

[SS88] D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *ACM Transactions on Programming Languages and Systems*, 10(2):282–212, April 1988.