Specification and Verification of Memory Consistency Models for Shared-Memory Multiprocessor Systems

SHIRO TAKATA,†‡, KENJI TAGUCHI,†‡, KAZUKI JOE,†‡ and AKIRA FUKUDA,†‡

In this paper we formally specify memory consistency models for shared-memory multiprocessor systems, focusing on causal memory consistency model, by use of a formal specification technique proposed by Taguchi and Araki. The formal specification technique includes a language, which is based on the combination of the Z notation and CCS (Calculus of Communicating Systems), and the state-based CCS semantics, which integrates Z and CCS semantics. Then, we verify that the specified causal memory meets the causal memory consistency condition using the extended state-based CCS semantics.

1. Introduction

In this paper we propose a formal method to specify and verify memory consistency models for shared-memory multiprocessor systems using a formal specification technique proposed by Taguchi and Araki1,2. The formal specification technique is based on a language that has combination characteristics of the Z notation and CCS (Calculus of Communicating Systems)3.

The Z notation is a model-based specification language based on set theory and first-order predicate logic. It has rich data structures and facilities to define various operations. Thus it is suited for modeling states and operations. But the Z notation does not have enough facilities to specify concurrency aspects. CCS is a process algebra that is suitable vehicle for modeling mathematical structure of concurrency aspects. However CCS has no explicit modeling facilities for states and operations. Therefore, the combination of the Z notation and CCS, which complements each other, would result in a versatile specification language3.

Taguchi and Araki also proposed the state-based CCS semantics that integrates Z and CCS semantics in order to verify given systems3.

Memory consistency models for shared-memory multiprocessor systems define the behavior of memory with respect to read and write operations. In this paper, we focus on causal memory consistency model proposed by Hutto4. Causal memory is an implementation of the memory mechanism which satisfies the causal memory consistency condition: any read operation to shared-memory obtains the value which is consistent with other causally related read and write operations.

In this paper, we give more formal and therefore sufficient specification of the process components (states and operations in Z) and the concurrency aspects of the causal memory by the combination use of Z and CCS. We also specify the causal memory consistency model and verify that the specification of causal memory meets the causal memory consistency condition using the state-based CCS semantics and its extension to a sequence of actions in finite length.

This paper is structured as follows. In section 2, weak vector clocks5,6 based on the causally-precedes relation defined by Lamport7 is described. In section 3, the state-based CCS semantics and its extension are explained. In section 4, definition of causal memory consistency model using the extended state-based CCS is given. In section 5, a description of causal memory is described by using the combination of Z and CCS. Verification of causal memory is presented in section 6. Finally, we conclude and indicate our future works in section 7.

2. Weak Vector Clocks

Vector clocks5 are used in distributed systems to determine whether a pair of events e_i, e_j has causal relation denoted by e_i \rightarrow e_j where \rightarrow is the causally-precedes relation defined by Lamport7. Using the vector clocks, a timestamp is recorded when any event is detected, and the causal relationship of pairs of events is determined by comparing the timestamps. The timestamp is an n-tuple of integers, where n is the number of processes. Given two events e_i, e_j and their associated vector timestamps t(e_i), t(e_j), the following relation holds:

\[ t(e_i) < t(e_j) \equiv (\forall k: 1 \ldots n \cdot t(e_i)[k] \leq t(e_j)[k]) \wedge (\exists l: 1 \ldots n \cdot t(e_i)[l] < t(e_j)[l]) \]

\[ t(e_i) \leq t(e_j) \equiv (t(e_i) < t(e_j)) \lor (t(e_i) = t(e_j)) \]

\[ t(e_i) \leq t(e_j) \Leftrightarrow e_i \rightarrow e_j \]

With the traditional vector clocks, the local counter t_i[i] of a process P_i increases whenever the P_i executes each event. In contrast, with weak vector clocks5,6, t_i[i] increases only when P_i executes an event that potentially leads to change the system property which is expressed by some state variables.

In either case, P_i sends a message that contains P_i's state change information with its vector timestamp t_i to all other processes whenever its vector clock changes. When such a message is received, P_j updates its vector timestamp t_j as follows:

\[ \forall k: 1 \ldots n \cdot t_j[k] = \max(t_i[k], t_j[k]) \]
Figure 1 illustrates a history of events in the causal memory system described in section 5 which adopts weak vector clocks. Note that \(t_i[j]\) is the number of write operations by \(P_j\) because \(t_i\) is initialized to the 0 vector.

3. The State-Based CCS Semantics and its Extension

In [2], Milner provides the operational semantics of CCS in terms of the following labeled transition system:

\[
(E, Act, \alpha \in Act)
\]

which consists of the set \(E\) of agent expressions in CCS, the set \(Act\) of actions, and the transition relation \(\alpha \subseteq E \times E\) for each \(\alpha \in Act\). For example, a process \(E\) which evolves another process \(E'\) by an action \(\alpha\) is denoted by the following transition relation:

\[
E \xrightarrow{\alpha} E'
\]

In the same way, Taguchi and Araki regard operation schemas in Z as transitions from old states to new states so that they provide the operational semantics of Z in terms of the following labeled transition system [3]:

\[
(S, Op, \alpha \in Op)
\]

which consists of the set \(S\) of states in Z, the set \(Op\) of operation schemas, and the transition relation \(\alpha \subseteq S \times S\) for each \(\alpha \in Op\). For example, a state \(s\) which evolves another state \(s'\) by an operation schema \(\alpha\) is denoted by the following transition relation:

\[
s \xrightarrow{\alpha} s'
\]

In addition, Taguchi and Araki [3] combine the Z notation with CCS to provide the operational semantics of this combined language, named the state-based CCS, in terms of the following labeled transition system:

\[
(E \times S, Act \cup Op, \alpha \in Act \cup Op)
\]

There is a restriction \(Act \cap Op = \emptyset\) which makes distinction between actions in CCS and operation schemas in Z. For example, a process \(E, s\) with the state \(s\) which evolves another process \(E'\) with the state \(s'\) by an operation schema \(\alpha\) in Z is denoted by the following transition relation:

\[
(E, s) \xrightarrow{\alpha} (E', s') \quad (\alpha \in Op, s \xrightarrow{\alpha} s')
\]

provided that \(s, s' \models [\theta]\), where \(\theta\) is the first-order representation of an operation schema \(\alpha\).

In [3], Taguchi and Araki also provide the following transition rules.

Prefix operator (1)

\[
\frac{\alpha, E, s \xrightarrow{\alpha} (E, s')} {E \xrightarrow{\alpha} (E', s')} \quad (\alpha \in Op, s \xrightarrow{\alpha} s')
\]

Prefix operator (2)

\[
\frac{(\alpha, E, s) \xrightarrow{\alpha} (E, s)} {\alpha \in Act}
\]

Prefix operator (3)

\[
\frac{(\alpha(z), E, s) \xrightarrow{\alpha} (E, s)} {s \models [\theta]} \quad (\alpha \in Act)
\]

Prefix operator (4)

\[
\frac{(s', s) \xrightarrow{\alpha} (E, s')} {s' = s(z)}
\]

Recursion

\[
\frac{(E, s) \xrightarrow{\alpha} (E', s')} {P \xrightarrow{\alpha} E}
\]

Sum (Non-deterministic Choice)

\[
\frac{(E_1, s) \xrightarrow{\alpha} (F, s') \quad (E_2, s) \xrightarrow{\alpha} (F, s') \quad (E_3, s) \xrightarrow{\alpha} (F, s') \quad (E_4, s) \xrightarrow{\alpha} (F, s') \quad (E_5, s) \xrightarrow{\alpha} (F, s') \quad (E_6, s) \xrightarrow{\alpha} (F, s') \quad (E_7, s) \xrightarrow{\alpha} (F, s') \quad (E_8, s) \xrightarrow{\alpha} (F, s')} {E \xrightarrow{\alpha} (F, s')}
\]

Concurrent Composition (1)

\[
\frac{(E, s) \xrightarrow{\alpha} (E', s') \quad (F, s) \xrightarrow{\alpha} (F', s')} {E \parallel F \xrightarrow{\alpha} (E', s')}
\]

Concurrent Composition (2)

\[
\frac{(E, s) \xrightarrow{\alpha} (E', s') \quad (F, s) \xrightarrow{\alpha} (F', s')} {E \parallel F \xrightarrow{\alpha} (E', s')}
\]

Restriction

\[
\frac{(E, s) \xrightarrow{\alpha} (F', s') \quad (E, s) \xrightarrow{\alpha} (F, s') \quad (E, s) \xrightarrow{\alpha} (F, s') \quad (E, s) \xrightarrow{\alpha} (F, s') \quad (E, s) \xrightarrow{\alpha} (F, s') \quad (E, s) \xrightarrow{\alpha} (F, s') \quad (E, s) \xrightarrow{\alpha} (F, s') \quad (E, s) \xrightarrow{\alpha} (F, s')} {E \xrightarrow{\alpha} (F', s')}
\]

Renaming

\[
\frac{(E, s) \xrightarrow{\alpha} (F, s')} {E' \xrightarrow{\alpha} (F, s')}
\]

Stirling's definition of a natural extension of a single transition relation \(\alpha\) to a sequence of actions in finite length, or traces \(\alpha_1 \ldots \alpha_n\), to provide the following transition rules:

Let \(\omega\) be such a sequence with \(\epsilon\) as an empty trace. The notation \(E \xrightarrow{\omega} F\) represents \(E\) may perform the trace \(\omega\) and become \(F\).

\[
E \xrightarrow{\omega} E' \xrightarrow{\omega} F \xrightarrow{\omega} F'
\]

We propose the following natural extension of transition relations and trace transition rules:

Let \(\omega\) be a sequence of actions including operation schemas \(\alpha_1 \ldots \alpha_n\).

\[
(E, s_1) \xrightarrow{\alpha_1} (E', s'_1) \xrightarrow{\alpha_2} (E', s'_2) \xrightarrow{\alpha_3} \ldots \xrightarrow{\alpha_n} (E', s'_n)
\]

provided that \(V : 1 \ldots n \Rightarrow s_i, s'_i \models [\theta_i]\), where \(s_i\) and \(s'_i\) are regarded as old state and new state of a operation schema \(\alpha_i\) as a transition, respectively, \(\theta_i\) is the first-order representation of \(\alpha_i\), and \(n\) is the number of operation schemas in \(\omega\).

Trace

\[
\frac{(E, s) \xrightarrow{\alpha} (E, s')} {E \xrightarrow{\alpha} (E', s') \xrightarrow{\alpha} (E', s'') \xrightarrow{\alpha} \ldots \xrightarrow{\alpha} (E', s_n) \xrightarrow{\alpha} (F, s_n)}
\]

\[
\frac{(E, s) \xrightarrow{\alpha} (E, s')} {E \xrightarrow{\alpha} (E', s') \xrightarrow{\alpha} \ldots \xrightarrow{\alpha} (E', s_n) \xrightarrow{\alpha} \ldots \xrightarrow{\alpha} (E, s_n) \xrightarrow{\alpha} (F, s_n)}
\]

4. Definition of Causal Memory Consistency Model

This section explains the causal memory consistency model proposed by Hutto and Ahamad in [3],[4], then defines the model in terms of the extended state-based CCS semantics.
4.1 Shared Memory Parallel Computer Model

In [4], Hutto and Ahamad define a shared memory parallel computer model as follows:

- It is a finite set \( P \) of processes \( \{P_1, \ldots, P_n\} \) that interact by a series of read and write operations via a shared memory that consists of a finite set of locations.
- A write operation by a process \( P_i \), denoted by \( w_i(z, v) \), here, stores the value \( v \) in location \( z \).
- A read operation, denoted by \( r_i(z, v) \) here, notifies \( P_i \) that \( v \) is stored in location \( z \).

A local execution history \( L_i \) of process \( P_i \) is a sequence of read and write operations. An execution history \( H = (L_1, L_2, \ldots, L_n) \) is a collection of local histories. Let \( A \) be a set of all operations in \( H \) and \( A_{\text{write}} \) be a set of all operations by \( P_i \) and all write operations in \( H \). Two kinds of program orders, serialization and "respect" are defined as follows:

- \( o_1 \triangleright o_2 \), if operation \( o_1 \) precedes \( o_2 \) in \( L_i \).
- \( S_i \) for \( P_i \) is a serialization of \( A \), if \( S_i \) is an linear sequence containing exactly the operations in \( A \) such that each read operation from a location returns the value written by the most recent preceding write to the location. If a read operation has no preceding write, an initial value is assumed to be returned.

- **Serialization** \( S_i \) of \( A \) respects order \( \rightarrow \), if, for any operations \( o_1, o_2 \) and \( A \), \( o_1 \rightarrow o_2 \) implies that \( o_1 \) precedes \( o_2 \) in \( S_i \).

Let \( \omega \) be a sequence \( A^\omega \) of operations in \( H \) with \( \varepsilon \) as an empty trace. Let \( \text{per}(\omega) \) be an operation that \( P_i \) "perceives" the operation \( o \). For example, \( \text{per}(r_i(z, v)) \), \( \text{per}(w_i(z, v)) \) and \( \text{per}(w_i(z, v)) \) are \( r_i(z, v) \) by \( P_i \), \( r_i(z, v) \) by \( P_i \), \( w_i(z, v) \) to its local memory, respectively. (See the operation schema \( \text{Apply}_i \) that will be described in section 5.)

Now we define the above definitions in terms of the CCS semantics as follows respectively:

- \( o_1, o_2 \in L_i \), \( \exists \omega \in A^\omega \cdot (o_1 \triangleright o_2) \)
- \( o_1, o_2 \in H \), \( \exists \omega \in A^\omega \cdot (o_1 \triangleright o_2) \)
- \( \forall o \in \omega \in \exists \text{per}(o) \in S_i \)
- \( f, k : 1, \ldots, n \), \( \forall \text{per}(r_i(z_i, v_i)) \in S_i \cdot \text{per}(w_i(z_i, v_i)) \notin \omega \cdot \text{per}(w_i(z_i, v_i)) \notin \omega \cdot \text{per}(w_i(z_i, v_i)) \notin \omega \)

4.2 Definition of Causal Memory Consistency Model

In [4], Hutto and Ahamad define write-into order and causality order for the definition of the causal memory consistency model as follows:

A write-into order \( \rightarrow \) on \( H \) is any relation with the following properties:

- if \( o_1 \rightarrow o_2 \), then \( x \) and \( v \) exist such that \( o_1 = w_i(x, v) \) and \( o_2 = r_i(z, v) \);
- for any operation \( o_2 \), there is at most one \( o_1 \) such that \( o_1 \rightarrow o_2 \);
- if \( o_2 = r_i(z, v) \) for some \( z \) and there is no \( o_1 \) such that \( o_1 \rightarrow o_2 \), then \( v = \lambda \); that is, a read with no write must read the initial value.

**Causality order** \( o_1 \prec o_2 \) on \( H \) if and only if one of the following cases holds:

- \( o_1 \prec o_2 \) for some \( P_1 \) (\( o_1 \) precedes \( o_2 \) in \( L_i \));
- \( o_1 \prec o_2 \) (\( o_2 \) reads the value written by \( o_1 \)); or
- there is some other operation \( o' \) such that \( o_1 \prec o' \prec o_2 \)

(If the relation is cyclic, then it is not the causality order.)

A history \( H \) is causal if it has the causality order such that:

**CM:** For each process \( P_i \), there is a serialization \( S_i \) of \( A_{\text{write}}^\omega \) that respects \( \prec \).

Now we define write-into order on \( H \) in terms of the CCS semantics.

\[
i, j : 1, \ldots, n, \forall r_i(z_i, v_i) \in A^i \cdot (\exists w_i(z_i, v_i) \in A^i \cdot r_i(z_i, v_i) \\
((w_i(z_i, v_i), E_i, s_i) \in (r_i(z_i, v_i), E_i, s_i)) \\
(\exists w_j(z_j, v_j) \in A^j, w_j(z_j, v_j) \notin \omega \cdot s_i \in \text{Start}_{E_i, s_i} \cdot r_i(z_i, v_i) \in E_i, s_i) \wedge \\
((\exists w_i(z_i, v_i, v_i) \in A^i, w_i(z_i, v_i, v_i) \notin \omega \cdot (\text{Start}_{E_i, s_i} \rightarrow (r_i(z_i, v_i), E_i, s_i)) \\
\wedge \forall o_1, o_2 \in A^i, \exists \omega \in A^\omega \cdot (o_1 \triangleright o_2) \\
(\exists o_1 \in A, \exists \omega, \exists o_2 \in A^* \\
(o_1, E_i, s_i) \equiv (o_2, E_i, s_i) \\
(o_1, o_2) \in \text{per}(\omega, E_i, s_i)) \wedge \\
(\exists o_1 \in A, \exists \omega, \exists o_2 \in A^* \\
(o_1, E_i, s_i) \equiv (o_2, E_i, s_i) \\
(o_1, o_2) \in \text{per}(\omega, E_i, s_i)) \wedge \\
(\forall r_i(z_i, v_i) \in S_i \cdot r_i(z_i, v_i) \in A^i) \\
(w_i(z_i, v_i), E_i, s_i) \equiv (r_i(z_i, v_i), E_i, s_i) \\
(\exists o_1 \in A, \exists \omega, \exists o_2 \in A^* \\
(o_1, o_2) \in \text{per}(\omega, E_i, s_i)) \wedge \\
(\exists o_1 \in A, \exists \omega, \exists o_2 \in A^* \\
(o_1, o_2) \in \text{per}(\omega, E_i, s_i)) \wedge \\
(\forall r_i(z_i, v_i) \in S_i \cdot (r_i(z_i, v_i), E_i, s_i)) \wedge \\
(w_i(z_i, v_i), E_i, s_i) \equiv (r_i(z_i, v_i), E_i, s_i) \\
(\exists o_1 \in A, \exists \omega, \exists o_2 \in A^* \\
(o_1, o_2) \in \text{per}(\omega, E_i, s_i)) \wedge \\
(\forall r_i(z_i, v_i) \in S_i \cdot (r_i(z_i, v_i), E_i, s_i)) \wedge \\
(w_i(z_i, v_i), E_i, s_i) \equiv (r_i(z_i, v_i), E_i, s_i))
\]

\[\]
5. A Description of Causal Memory

In this section, using the combination of Z and COS, a formal specification of causal memory, which was proposed by Ahamad et al.\(^6\), is described.

First, the states and operation schemas of each process are specified in Z. Weak vector clocks are adopted here as logical time of distributed systems like Marzullo and Neiger did\(^6\). Second, the concurrency aspects of causal memory are specified in COS.

5.1 Specifying the component of each process in Z

Each process \(P_i\) has a schema \(s_i\) of the state in Z. Each schema \(s_i\) consists of seven local data structures; a process identity number \(i\), a local memory \(M_i\) of the abstract shared causal memory \(\mathcal{M}\), a vector timestamp \(t_i\) which is used for updating the local timestamp, two message queues \(\text{OutQueue}_i\) and \(\text{InQueue}_i\), a local execution history \(L_i\) of \(A\) which is a set of read and write operations, and a serialization \(S_i\) of \(A_{\text{local}}^{w}\) which is a set of all operations by \(P_i\) and all write operations in \(H\). \(\text{OutQueue}_i\) is a first-in-first-out queue and contains information about write operations to local memory that have not been communicated to other processes yet. \(\text{InQueue}_i\) is ordered by vector timestamps and contains information about remote write operations to its remote memory that have not been written to local memory yet.

The schema \(s_i\) of \(P_i\)'s state is described using Z as follows:

\[
\begin{align*}
\text{write\_tuple} & = \text{seq } N_1 \times \mathcal{M} \times \text{Val} \times \text{seq } N_1 \\
\text{Number\_Of\_Processes} & = N_1 \\
\text{Max\_Out\_Queue} & = N_1 \\
\text{Max\_Serial\_Local\_Hiss} & = N_1 \\
\text{priority\_queue} & = (\text{seq } \text{write\_tuple}) \times \text{write\_tuple} \\
& \triangleright \text{seq } \text{write\_tuple} \\
\end{align*}
\]

\(P_i\) has an initialization operation schema \(\text{Init}_i\) and five basic operation schemas; \(\text{Read}_i\), \(\text{Write}_i\), \(\text{Send}_i\), \(\text{Receive}_i\), and \(\text{Apply}_i\).

\[\Delta s_i\]

\(\begin{align*}
\text{write\_tuple} & = \text{seq } N_1 \times \mathcal{M} \times \text{Val} \times \text{seq } N_1 \\
\text{Number\_Of\_Processes} & = N_1 \\
\text{Max\_Out\_Queue} & = N_1 \\
\text{Max\_Serial\_Local\_Hiss} & = N_1 \\
\text{priority\_queue} & = (\text{seq } \text{write\_tuple}) \times \text{write\_tuple} \\
& \triangleright \text{seq } \text{write\_tuple} \\
\end{align*}\]

\(A\) read operation schema \(\text{Read}_i\) is executed whenever a read operation to a location \(z\) is invoked by \(\text{P}_i\). Then, the value \(v!\) stored in \(\mathcal{M}_i(z)\) is sent to \(\text{P}_i\). A label \(\sigma(z, v!\) of the read operation is added to a local execution history \(L_i\) and a serialization \(S_i\).

\[\Delta s_i\]

\(\begin{align*}
\text{write\_tuple} & = \text{seq } N_1 \times \mathcal{M} \times \text{Val} \times \text{seq } N_1 \\
\text{Number\_Of\_Processes} & = N_1 \\
\text{Max\_Out\_Queue} & = N_1 \\
\text{Max\_Serial\_Local\_Hiss} & = N_1 \\
\text{priority\_queue} & = (\text{seq } \text{write\_tuple}) \times \text{write\_tuple} \\
& \triangleright \text{seq } \text{write\_tuple} \\
\end{align*}\]

\(A\) write operation schema \(\text{Write}_i\) is executed whenever a write operation of a value \(v\) to a location \(z\) is invoked by \(\text{P}_i\). \(\text{P}_i\) increases \(t_i\), writes \(v\) to \(\mathcal{M}_i(z)\), and appends the tuple \((i, z, v, t_i)\) to \(\text{OutQueue}_i\). This tuple is called a \text{write\_tuple} which is a message to other processes. A label \(\nu_i(z, v)\) of the write operation is appended to \(L_i\) and \(S_i\).

\[\Delta s_i\]

\(\begin{align*}
\text{write\_tuple} & = \text{seq } N_1 \times \mathcal{M} \times \text{Val} \times \text{seq } N_1 \\
\text{Number\_Of\_Processes} & = N_1 \\
\text{Max\_Out\_Queue} & = N_1 \\
\text{Max\_Serial\_Local\_Hiss} & = N_1 \\
\text{priority\_queue} & = (\text{seq } \text{write\_tuple}) \times \text{write\_tuple} \\
& \triangleright \text{seq } \text{write\_tuple} \\
\end{align*}\]

The information about local write operations in \(\text{OutQueue}_i\) must be notified to all other processes. A send operation schema \(\text{Send}_i\) sends a nonempty prefix of \(\text{OutQueue}_i\) to all other processes and removes it from \(\text{OutQueue}_i\).

When a message is received by \(\text{P}_i\), a receive operation schema \(\text{Receive}_i\) is executed. \(\text{Receive}_i\) appends the message to \(\text{InQueue}_i\) which is a priority queue sorted by vector timestamps. The \(\text{InQueue}_i\) and an element with vector timestamps are inputted to a function \text{priority\_queue}. The \text{priority\_queue} attaches the element to \(\text{InQueue}_i\), and returns the new \(\text{InQueue}_i\). Although it is not hard to specify the function \text{priority\_queue} in Z, the specification is not presented here because of lack of space.

\[\Delta s_i\]

\(\begin{align*}
\text{write\_tuple} & = \text{seq } N_1 \times \mathcal{M} \times \text{Val} \times \text{seq } N_1 \\
\text{Number\_Of\_Processes} & = N_1 \\
\text{Max\_Out\_Queue} & = N_1 \\
\text{Max\_Serial\_Local\_Hiss} & = N_1 \\
\text{priority\_queue} & = (\text{seq } \text{write\_tuple}) \times \text{write\_tuple} \\
& \triangleright \text{seq } \text{write\_tuple} \\
\end{align*}\]
5.2 Specifying the concurrency aspect of causal memory in CCS

In this section, we specify the concurrency aspects of causal memory in CCS. We assume that causal memory has k processes. Each of the processes is connected with all other processes through input ports pipe1 ... pipek, and output ports pipe1 ... pipek.

Each process Pi consists of six operation schemas specified above in Z and four basic input or output ports: id, loc, val, and pipe. An input port id is initially executed by each process Pi to obtain its process identity number. The other ports loc, val, and pipe are used for input or output ports in the following abbreviated actions: r1(x?, v1), w1(x?, v1), and broadcasti(message1).

An action r1(x?, v1) is an abbreviated action of blocked sequential actions: First, an input port loci is executed whenever a location x2 for a read action is received through the input port loci. Second, a read operation schema Readi is executed by Pi. Finally, the value v! stored in Mj(x?) is sent to Pi through an output port valj.

An action w1(x?, v1) is also an abbreviated action of blocked sequential actions: First, an input port heapj is executed whenever a value v? to a location x2 for a write action is received through the input port heapj. Second, a write operation schema Writei is executed by Pi.

We assume that these sequential actions in r1(x?, v1) and w1(x?, v?) are blocked, which means that other processes which run concurrently are blocked during those executions. It is not hard to specify that kind of blocked operations in CCS, but makes the specification hard to be understood. Hence we simply assume that these actions are blocked here.

A broadcast action broadcasti(message1) is an abbreviation action that sends a message m! to all other processes through all output ports pipe except pipei as follows:

broadcasti(message1) ≡ Sendi.

We then specify the concurrency aspects of the processes Pi in CCS.

Pi ≡ r1(x?, v1).Pi + w1(x?, v1).Pi + broadcasti(message1).Pi + pipei(message1).Receivei.Pi + Applyi.Pi

Each process is connected with all other processes through input ports pipe1 ... pipek and output ports pipe1 ... pipek to communicate the information about local writes to local memory. Then we specify a causal memory C/M in CCS as follows:

K = {pipe1, ..., pipek}
C/M ≡ id(1).InitP1, ..., id(k).InitPk.

6. Verification of Causal Memory

In this section we should describe the verification that the specified causal memory described in section 5 meets the causal memory consistency condition described in section 4 using the extended state-based CCS semantics. But we present only the verification of the following theorem 3 here because of lack of space. In [4], Abadi and Hutto prove the following Lemma 1.2 and Theorem 3.1.

Lemma 1: Let H be a history of the implementation, ts(a) be the timestamp of an operation o, and o1 and o2 be two operations such that o1v o2. Then ts(a1) ≤ ts(a2). Furthermore, if o2 is a write operation by Pi, ts(a1)[i] < ts(a2)[i], so ts(a1) < ts(a2).

Lemma 2: Let H be a history of the implementation and suppose that w1(x, v) is a write operation of process Pi. Then each process Pi eventually applies w1(x, v) to its local memory.

Theorem 3: Let H be a history of the implementa-
tation. Then $H$ is causal.

Proof: An inspection of operation schemas $Read_i$, $Write_i$, and $Apply_i$ shows that the serialization $S_i$ for $P_i$ includes all writes in $H$ (by Lemma 2) and all read operation in $L_i$. Thus,

$$V : 1 \ldots n \cdot \forall \nu \in A^H_{-L_i} \cdot \exists \nu \in S_{i+1} \cdot A^H_{i+1}$$

An inspection of operation schemas $Read_i$, $Write_i$ and $Apply_i$ shows that a local memory $M_i$ is updated such that $M'_i = M_i \oplus \{ \nu \}$ by $P_i$, and the value $\nu!$ such that $\nu! = M'_i \oplus \nu$ is reported to $P_i$. Then each read operation schema notifies the value that most recently write operation schema has written. Thus, the following transitions exist:

$$j : 1 \ldots n \cdot \forall \nu : 1 \ldots n \cdot$$

$$(\forall \nu_i (v_i, v_i) \in S_i \cdot A^H_{i+1})$$

$$(\exists \nu \in S_{i+1} \cdot A^H_{i+1})$$

$$\nu \in S_{i+1}$$

$$\nu \in S_{i+1}$$

$$\nu \in S_{i+1}$$

$$\nu \in S_{i+1}$$

$$\nu \in S_{i+1}$$

$$\nu \in S_{i+1}$$

Thus, the following condition holds:

$$j : 1 \ldots n \cdot \forall \nu : 1 \ldots n \cdot$$

$$\nu \in S_{i+1}$$

$$\nu \in S_{i+1}$$

$$\nu \in S_{i+1}$$

$$\nu \in S_{i+1}$$

Thus, the proof is complete.

7. Conclusion

In this paper we described causal memory using the formal specification technique proposed by Taguchi and Araki. We presented an extension of the state-based CCA semantics to a sequence of actions with finite length in order to deal with a sequence of actions and operations. Using this extended state-based CCA semantics, we described causal memory consistency model and causal memory, then verified that causal memory meets causal memory consistency model.

We have much future work remain to be done. First, we should adopt this technique to other memory consistency models, especially release consistency models, and have to develop a new formal technique for specifying lock and release operations. Second, we will be able to propose a new memory consistency model and present a design of new parallel computer architectures with the new memory consistency model. Those models and architecture designs can be formally described and verified by our proposing formal techniques.

Acknowledgements

We would like to thank Keiji Araki at Kyushu University.

References