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

Shiro Takata,[†1,†2] Kenji Taguchi,[†3] Kazuki Joe[†4]
and Akira Fukuda [†1]

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 Araki[1]. The formal specification technique is based on a language that has combination characteristics of the Z notation and CCS (Calculus of Communicating Systems)[2].

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 language[1]. Taguchi and Araki also proposed the state-based CCS semantics that integrates Z and CCS semantics in order to verify given systems[1].

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 Hutto[3]. 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. A formal definition, implementation and verification of the causal memory have already been presented by Ahamad and Hutto[4]. Regardless of their results, they are inefficient for us to formalize every memory consistency model since they just use algebra.

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 clocks[5),6)] based on the causally-precedes relation defined by Lamport[7)] 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 clocks[8)] 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 Lamport[7)]. 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:

$$t(e_i) \prec t(e_j) \stackrel{\text{def}}{=} (\forall k : 1 \ldots n \bullet t(e_i)[k] \leq t(e_j)[k]) \wedge (\exists l : 1 \ldots n \bullet t(e_i)[l] < t(e_j)[l])$$

$$t(e_i) \preceq t(e_j) \stackrel{\text{def}}{=} (t(e_i) \prec t(e_j)) \vee (t(e_i) = t(e_j))$$

$$t(e_i) \preceq t(e_j) \Leftrightarrow e_i \rightarrow e_j$$

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 clocks*[5)], $t_i[i]$ 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:

$$\forall k : 1 \ldots n \bullet t_j[k] = \max(t_j[k], t_i[k])$$

†1 Nara Institute of Science and Technology
†2 Keihanna Interaction Plaza Inc.
†3 Kyushu University
†4 Wakayama University

Figure 1 illustrates a history of events in the

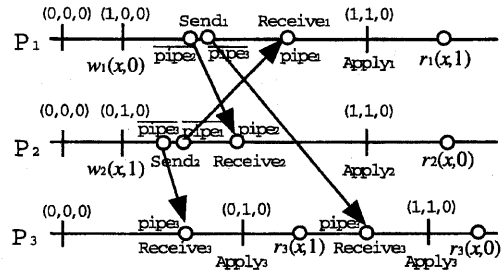

Fig. 1 A history of events in a causal memory system

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:

$$\langle \mathcal{E}, Act, \{\xrightarrow{\alpha} \mid \alpha \in Act\}\rangle$$

which consists of the set $\mathcal{E}$ of agent expressions in CCS, the set Act of actions, and the transition relation $\xrightarrow{\alpha} \subseteq \mathcal{E} \times \mathcal{E}$ for each $\alpha \in Act$. For example, a process E which evolves another process E' by an action α 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[1]:

$$\langle St, Op, \{\xrightarrow{\alpha} \mid \alpha \in Op\}\rangle$$

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

$$s \xrightarrow{\alpha} s'$$

In addition, Taguchi and Araki[1] 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.

$$\langle \mathcal{E} \times St, Act \cup Op, \{\xrightarrow{\alpha} \mid \alpha \in Act \cup Op\}\rangle$$

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

$$\langle \alpha.E, s\rangle \xrightarrow{\alpha} \langle E, s'\rangle \Leftrightarrow \alpha.E \xrightarrow{\alpha} E \wedge s \xrightarrow{\alpha} s'$$

provided that $s, s' \models [\![\Theta]\!]$, where Θ is the first-order representation of an operation schema α.

In[1], Taguchi and Araki also provide the following transition rules.

Prefix operator (1)

$$\frac{}{\langle \alpha.E, s\rangle \xrightarrow{\alpha} \langle E, s'\rangle} \ (\alpha \in Op, s \xrightarrow{\alpha} s')$$

Prefix operator (2)

$$\frac{}{\langle \alpha.E, s\rangle \xrightarrow{\alpha} \langle E, s\rangle} \ (\alpha \in Act)$$

Prefix operator (3)

$$\frac{}{\langle \overline{\alpha}(x!).E, s\rangle \xrightarrow{\overline{\alpha}(c)} \langle E, s\rangle} \ (s[\![x!]\!] = c)$$

Prefix operator (4)

$$\frac{}{\langle \alpha(x?).E, s\rangle \xrightarrow{\alpha(c)} \langle E, s'\rangle} \ (s' = s\{c/x?\})$$

Recursion

$$\frac{\langle E, s\rangle \xrightarrow{\alpha} \langle F, s'\rangle}{\langle P, s\rangle \xrightarrow{\alpha} \langle F, s'\rangle} \ (P \stackrel{\text{def}}{=} E)$$

Sum(Non-deterministic Choice)

$$\frac{\langle E_1, s\rangle \xrightarrow{\alpha} \langle F, s'\rangle}{\langle E_1 + E_2, s\rangle \xrightarrow{\alpha} \langle F, s'\rangle} \qquad \frac{\langle E_2, s\rangle \xrightarrow{\alpha} \langle F, s'\rangle}{\langle E_1 + E_2, s\rangle \xrightarrow{\alpha} \langle F, s'\rangle}$$

Concurrent Composition (1)

$$\frac{\langle E, s\rangle \xrightarrow{\alpha} \langle E', s'\rangle}{\langle E \mid F, s\rangle \xrightarrow{\alpha} \langle E' \mid F, s'\rangle} \qquad \frac{\langle F, s\rangle \xrightarrow{\alpha} \langle F', s'\rangle}{\langle E \mid F, s\rangle \xrightarrow{\alpha} \langle E \mid F', s'\rangle}$$

Concurrent Composition (2)

$$\frac{\langle E, s\rangle \xrightarrow{\overline{\alpha}(c)} \langle E', s'\rangle \quad \langle F, s\rangle \xrightarrow{\alpha(c)} \langle F', s'\rangle}{\langle E \mid F, s\rangle \xrightarrow{\tau} \langle E' \mid F', s'\rangle}$$

Restriction

$$\frac{\langle E, s\rangle \xrightarrow{\alpha} \langle F, s'\rangle}{\langle E \setminus L, s\rangle \xrightarrow{\alpha} \langle F \setminus L, s'\rangle} \ (\alpha \notin L, \alpha \in Act)$$

Renaming

$$\frac{\langle E, s\rangle \xrightarrow{\beta} \langle F, s'\rangle}{\langle E[f], s\rangle \xrightarrow{\alpha} \langle F[f], s'\rangle} \ (\alpha \in Act, \alpha = f(\beta))$$

Stirling[9] defined a natural extension of a single transition relation $\xrightarrow{\alpha}$ to a sequence of actions in finite length, or *traces* $\alpha_1 \ldots \alpha_n$ to provide the following transition rules:

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

$$\frac{}{E \xrightarrow{\epsilon} E} \qquad \frac{E \xrightarrow{\alpha} E' \quad E' \xrightarrow{\omega} F}{E \xrightarrow{\alpha\omega} F}$$

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

Let ω be a sequence of actions including operation schemas $\alpha_1 \ldots \alpha_n$.

$$\langle E, s_1\rangle \xrightarrow{\omega} \langle F, s_n'\rangle \Leftrightarrow E \xrightarrow{\omega} F \wedge s_1 \xrightarrow{\omega} s_n'$$

provided that $\forall i : 1 \ldots n \bullet s_i, s_i' \models [\![\Theta_i]\!]$, where s_i and s_i' are regarded as old state and new state of a operation schema α_i as a transition, respectively, Θ_i is the first-order representation of α_i, and n is the number of operation schemas in ω.

Trace

$$\frac{}{\langle E, s\rangle \xrightarrow{\epsilon} \langle E, s\rangle}$$

$$\frac{\langle E, s_1\rangle \xrightarrow{\alpha} \langle E', s_1'\rangle \quad \langle E', s_1'\rangle \xrightarrow{\omega} \langle F, s_n'\rangle}{\langle E, s_1\rangle \xrightarrow{\alpha\omega} \langle F, s_n'\rangle} \ (\alpha \in Op)$$

$$\frac{\langle E, s_1\rangle \xrightarrow{\alpha} \langle E', s_1\rangle \quad \langle E', s_1\rangle \xrightarrow{\omega} \langle F, s_n'\rangle}{\langle E, s_1\rangle \xrightarrow{\alpha\omega} \langle F, s_n'\rangle} \ (\alpha \in Act)$$

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 $\mathcal{P}$ 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 $w_i(x, v)$ here, stores the value v in location x.
- A read operation, denoted by $r_i(x, v)$ here, notifies P_i that v is stored in location x.

A local execution history L_i of process P_i is a sequence of read and write operations. An execution history $H = \langle L_1, L_2, \ldots, L_n \rangle$ is a collection of local histories. Let A be a set of all operations in H and A^H_{i+w} 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:

- $o_1 \xrightarrow[i]{} o_2$, if operation o_1 precedes o_2 in L_i.
- $o_1 \rightarrow o_2$, if operation o_1 precedes o_2 in H.
- S_i for P_i is a *serialization* of A, if S_i is a 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 $\perp$ is assumed to be returned.
- *Serialization* S_i of A *respects order* $\rightarrow$, if, for any operations o_1 and o_2 in A, $o_1 \rightarrow o_2$ implies that o_1 precedes o_2 in S_i.

Let ω be a sequence A^* of operations in H with ϵ as an empty trace. Let $per_i(o)$ be an operation that P_i "*perceives*" the operation o. For example, $per_i(r_i(x, v))$, $per_i(r_j(x, v))$, $per_i(w_i(x, v))$ and $per_i(w_j(x, v))$ are $r_i(x, v)$ by P_i, $r_j(x, v)$ by P_i, $w_i(x, v)$ by P_i and $Apply_i$ such that P_i applies $w_j(x, v)$ to its local memory, respectively. (See an operation schema $Apply_i$ that will be described in section 5.)

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

- $o_1, o_2 \in L_i, \exists\, \omega \in A^* \bullet$
  $\langle o_1.E_1, s_1 \rangle \xrightarrow{\omega} \langle o_2.E_2, s_2 \rangle$
- $o_1, o_2 \in H, \exists\, \omega \in A^* \bullet$
  $\langle o_1.E_1, s_1 \rangle \xrightarrow{\omega} \langle o_2.E_2, s_2 \rangle$
- $(\forall\, o \in A \bullet \exists\, per_i(o) \in S_i)$
  $\wedge$
  $j, k : 1 \ldots n, \forall\, per_i(r_k(x_1, v_1)) \in S_i \bullet$
  $((\exists\, per_i(w_j(x_1, v_1)) \in S_i,$
  $per_i(w_j(x_1, v_m)) \notin \omega \bullet$
  $\langle per_i(w_j(x_1, v_1)).E_j, s_j \rangle$
  $\xrightarrow{per_i(w_j(x_1, v_1))\omega}$
  $\langle per_i(r_k(x_1, v_1)).E_i, s_i \rangle)$
  $\vee$
  $(((\exists\, per_i(w_j(x_1, v_m)) \in S_i,$
  $per_i(w_j(x_1, v_m)) \notin \omega \bullet$
  $\langle start.E_0, s_0 \rangle \xrightarrow{\omega} \langle per_i(r_k(x_1, v_1)).E_i, s_i \rangle$
  $\vee\ per_i(w_j(x_1, v_m)) \notin S_i \Rightarrow v_1 = \perp)))$
- $\forall\, o_1, o_2 \in A, \exists\, \omega_1, \omega_2 \in A^* \bullet$
  $\langle o_1.E_1, s_1 \rangle \xrightarrow{\omega_1} \langle o_2.E_2, s_2 \rangle \Rightarrow$
  $\langle per_i(o_1).E'_1, s'_1 \rangle \xrightarrow{\omega_2} \langle per_i(o_2).E'_2, s'_2 \rangle$

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 $\mapsto$ on H is any relation with the following properties:

- if $o_1 \mapsto o_2$, then x and v exist such that $o_1 = w(x, v)$ and $o_2 = r(x, v)$;
- for any operation o_2, there is at most one o_1 such that $o_1 \mapsto o_2$;
- if $o_2 = r(x, v)$ for some x and there is no o_1 such that $o_1 \mapsto o_2$, then $v = \perp$; that is, a read with no write must read the initial value.

A *causality order* $o_1 \rightsquigarrow o_2$ on H if and only if one of the following cases holds:

- $o_1 \xrightarrow[i]{} o_2$ for some p_i (o_1 precedes o_2 in L_i);
- $o_1 \mapsto o_2$ (o_2 reads the value written by o_1); or
- there is some other operation o' such that $o_1 \rightsquigarrow o' \rightsquigarrow o_2$

(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^H_{i+w} that respects $\rightsquigarrow$.

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

$i, j : 1 \ldots n, \forall\, r_i(x_1, v_1) \in A \bullet$
$((\exists\, w_j(x_1, v_1) \in A, \omega \in A^* \bullet$
$\langle w_j(x_1, v_1).E_j, s_j \rangle \xrightarrow{\omega} \langle r_i(x_1, v_1).E_i, s_i \rangle)$
$\vee$
$(((\exists\, w_j(x_1, v_m) \in A, w_j(x_1, v_m) \notin \omega \bullet$
$\langle start.E_0, s_0 \rangle \xrightarrow{\omega} \langle r_i(x_1, v_1).E_i, s_i \rangle)$
$\vee\ w_j(x_1, v_m) \notin A \Rightarrow v_1 = \perp)))$

By iteration of applying trace transition rule:

$$\frac{\exists\, o_1, o', o_2 \in A, \omega_1, \omega_2 \in A^* \bullet \quad \langle o_1.E_1, s_1 \rangle \xrightarrow{\omega_1} \langle o'.E', s' \rangle \quad \langle o'.E', s' \rangle \xrightarrow{\omega_2} \langle o_2.E_2, s'_2 \rangle}{\langle o_1.E, s_1 \rangle \xrightarrow{\omega_1\omega_2} \langle o_2.E_2, s_2 \rangle}$$

Then, if there are a trace ω_1 corresponding to $o_1 \rightsquigarrow o'$ and a trace ω_2 corresponding to $o' \rightsquigarrow o_2$, a trace $\omega_1\omega_2$ corresponding to $o_1 \rightsquigarrow o' \rightsquigarrow o_2$ always exists.

Since the causally-precedes relation $\rightarrow$ which the extended state-based CCS semantics uses is a partial order, $\rightarrow$ on traces with vector clocks is acyclic.

Now we can define the causal memory consistency condition in terms of the CCS semantics as follows:

A history H is causal $\Leftrightarrow$
$j : 1 \ldots n, \forall\, i : 1 \ldots n \bullet$
$(\,(\forall\, o \in A^H_{i+w} \bullet \exists\, per_i(o) \in S_i \textit{ of } A^H_{i+w})$
$\wedge$
$(\forall\, r_i(x_1, v_1) \in S_i \textit{ of } A^H_{i+w} \bullet$
$((\exists\, per_i(w_j(x_1, v_1)) \in S_i \textit{ of } A^H_{i+w},$
$per_i(w_j(x_1, v_m)) \notin \omega \bullet$
$\langle per_i(w_j(x_1, v_1)).E_j, s_j \rangle$
$\xrightarrow{per_i(w_j(x_1, v_1))\omega}$
$\langle r_i(x_1, v_1).E_i, s_i \rangle)$
$\vee\ (((\exists\, per_i(w_j(x_1, v_m)) \in S_i \textit{ of } A^H_{i+w},$
$per_i(w_j(x_1, v_m)) \notin \omega \bullet$
$\langle start.E_0, s_0 \rangle \xrightarrow{\omega} \langle r_i(x_1, v_1).E_i, s_i \rangle)$
$\vee\ per_i(w_j(x_1, v_m)) \notin S_i \textit{ of } A^H_{i+w} \Rightarrow v_1 = \perp))))$
$\wedge$
$((\forall\, o_1, o_2 \in L_j \textit{ of } A^H_{i+w} \bullet \exists\, \omega_1, \omega_2 \in A^* \bullet$
$\langle o_1.E_1, s_1 \rangle \xrightarrow{\omega_1} \langle o_2.E_2, s_2 \rangle) \Rightarrow$
$\langle per_i(o_1).E'_1, s'_1 \rangle \xrightarrow{\omega_2} \langle per_i(o_2).E'_2, s'_2 \rangle)$
$\wedge$
$(\forall\, r_i(x_1, v_1) \in S_i \textit{ of } A^H_{i+w} \bullet \exists\, \omega_1, \omega_2 \in A^* \bullet$
$\langle w_j(x_1, v_1).E_j, s_j \rangle \xrightarrow{\omega_1} \langle r_i(x_1, v_1).E_i, s_i \rangle) \Rightarrow$
$\langle per_i(w_j(x_1, v_1)).E'_j, s'_j \rangle \xrightarrow{\omega_2} \langle r_i(x_1, v_1).E'_i, s'_i \rangle))))$

5. A Description of Causal Memory

In this section, using the combination of Z and CCS, a formal specification of causal memory, which was proposed by Ahamad et al[4], 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 CCS.

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 $OutQueue_i$ and $InQueue_i$, a local execution history L_i of A which is a set of read and write operations, and a serialization S_i of A^{H}_{i+w} which is a set of all operations by P_i and all write operations in H. $OutQueue_i$ 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. $InQueue_i$ 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:

$[\mathcal{M}, A, Val]$

$$write\_tuple == \mathbb{N}_1 \times \mathcal{M} \times Val \times \mathrm{seq}\,\mathbb{N}$$

$$NumberOfProcesses : \mathbb{N}_1$$
$$MaxOutQueue, MaxInQueue : \mathbb{N}_1$$
$$MaxSerial, MaxLocalHis : \mathbb{N}_1$$
$$priority\_queue : (\mathrm{seq}\ write\_tuple) \times write\_tuple \rightarrow \mathrm{seq}\ write\_tuple$$

s_i

$$i : \mathbb{N}_1$$
$$M_i : \mathcal{M} \nrightarrow (Val \cup \{\bot\})$$
$$t_i : \mathrm{seq}\,\mathbb{N}$$
$$OutQueue_i : \mathrm{seq}\ write\_tuple$$
$$InQueue_i : \mathrm{seq}\ write\_tuple$$
$$L_i : \mathrm{seq}\ A$$
$$S_i : \mathrm{seq}\ A$$

$$\#t_i = NumberOfProcesses$$
$$\#OutQueue_i \leq MaxOutQueue$$
$$\#InQueue_i \leq MaxInQueue$$
$$\#L_i \leq MaxLocalHis$$
$$\#S_i \leq MaxSerial$$

P_i has an initialization operation schema $InitP_i$ and five basic operation schemas; $Read_i$, $Write_i$, $Send_i$, $Receive_i$, and $Apply_i$.

$InitP_i$

$$s_i'$$
$$pn_i? : \mathbb{N}_1$$

$$i' = pn_i?$$
$$M_i' = \lambda x : \mathcal{M} \bullet \bot$$
$$t_i' = \lambda n : 1 .. NumberOfProcesses \bullet 0$$
$$OutQueue_i' = <>$$
$$InQueue_i' = <>$$
$$L_i' = <>$$
$$S_i' = <>$$

A read operation schema $Read_i$ is executed whenever a read operation to a location $x?$ is invoked by P_i. Then, the value $v!$ stored in $M_i(x?)$ is sent to P_i. A label $r_i(x?, v!)$ of the read operation is added to a local execution history L_i and a serialization S_i.

$Read_i$

$$\Delta s_i$$
$$x? : \mathcal{M}$$
$$v! : Value$$

$$v! = M_i x?$$
$$i' = i$$
$$M_i' = M_i$$
$$t_i' = t_i$$
$$OutQueue_i' = OutQueue_i$$
$$InQueue_i' = InQueue_i$$
$$L_i' = L_i^\frown < r_i(x?, v!) >$$
$$S_i' = S_i^\frown < r_i(x?, v!) >$$

A write operation schema $Write_i$ is executed whenever a write operation of a value $v?$ to a location $x?$ is invoked by P_i. P_i increases $t[i]$, writes $v?$ to $M_i(x?)$, and appends the tuple $(i, x?, v?, t_i)$ to $OutQueue_i$. This tuple is called a *write_tuple* which is a message to other processes. A label $w_i(x?, v?)$ of the write operation is appended to L_i and S_i.

$Write_i$

$$\Delta s_i$$
$$x? : \mathcal{M}$$
$$v? : Value$$

$$i' = i$$
$$M_i' = M_i \oplus \{x? \mapsto v?\}$$
$$t_i' i = t_i i + 1$$
$$k : 1 .. \#t_i \mid k \neq i \bullet t_i' k = t_i k$$
$$OutQueue_i' = OutQueue_i^\frown < (i, x?, v?, t_i') >$$
$$InQueue_i' = InQueue_i$$
$$L_i' = L_i^\frown < w_i(x?, v?) >$$
$$S_i' = S_i^\frown < w_i(x?, v?) >$$

The information about local write operations in $OutQueue_i$ must be notified to all other processes. A send operation schema $Send_i$ sends a nonempty prefix of $OutQueue_i$ to all other processes and removes it from $OutQueue_i$.

When a message is received by P_i, a receive operation schema $Receive_i$ is executed. $Receive_i$ appends the message to $InQueue_i$ which is a priority queue sorted by vector timestamps. The $InQueue_i$ and an element with vector timestamps are inputted to a function *priority_queue*. The *priority_queue* attaches the element to $InQueue_i$, and returns the new $InQueue_i$. Although it is not hard to specify the function *priority_queue* in Z, the specification is not presented here because of lack of space.

$Send_i$

$$\Delta s_i$$
$$massage! : write\_tuple$$

$$Outqueue_i \neq <>$$
$$i' = i$$
$$M_i' = M_i$$
$$t_i' = t_i$$
$$massage! = head\ OutQueue_i$$
$$OutQueue_i' = tail\ OutQueue_i$$
$$InQueue_i' = InQueue_i$$
$$L_i' = L_i$$
$$S_i' = S_i$$

$Receive_i$

Δs_i

$massage? : write\_tuple$

$i' = i$

$M_i' = M_i$

$t_i' = t_i$

$OutQueue_i' = OutQueue_i$

$InQueue_i' = priority\_queue(InQueue_i, massage?)$

$L_i' = L_i$

$S_i' = S_i$

$Apply_i$ compares the local timestamp t_i with a remote timestamp t_j associated with the write operation which was executed by the remote process P_j. A write operation can be applied to local memory only if all components of t_j (other than the jth) are fewer than or equal to those of t_i and if the jth component of t_j is more than the jth component of t_i exactly by one. When a write operation is applied, it is removed from $InQueue_i$, the corresponding component of the local vector timestamp $t_i[j]$ is updated, and the new value v_j is written to $M_i(x_j)$. This means that such a write operation of $w_j(x_j, v_j)$ will be the most recent preceding write operation of the write-into order relation to the following read operation of $r_i(x_j, v_j)$, $w_j(x_j, v_j) \mapsto r_i(x_j, v_j)$ where the vector timestamp of $w_j(x_j, v_j)$ is less than or equal to that of $r_i(x_j, v_j)$. A label $w_j(x_j, v_j)$ of the write operation is appended to S_i.

$Apply_i$

Δs_i

$(j, x_j, v_j, t_j) : write\_tuple$

$InQueue_i \neq <>$

$i' = i$

$(j, x_j, v_j, t_j) = head\ InQueue_i$

$k : 1..\#t_i \mid k \neq j \bullet t_j\, k \leq t_i\, k$
$\quad \wedge\ t_j\, j = t_i\, j + 1$

$M_i' = M_i \oplus \{x_j \mapsto v_j\}$

$t_i'\, j = t_j\, j$

$k : 1..\#t_i \mid k \neq j \bullet t_i'\, k = t_i\, k$

$OutQueue_i' = OutQueue_i$

$InQueue_i' = tail\ InQueue_i$

$L_i' = L_i$

$S_i' = S_i^\frown < w_j(x_j, v_j) >$

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 $pipe_1 \cdots pipe_k$, and output ports $\overline{pipe_1} \cdots \overline{pipe_k}$.

Each process P_i consists of six operation schemas specified above in Z and four basic input or output ports; id_i, loc_i, val_i, and $pipe_i$. An input port id_i is initially executed by each process P_i to obtain its process identity number. The other ports loc_i, val_i, and $pipe_i$ are used for input or output ports in the following abbreviated actions: $r_i(x?, v!)$, $w_i(x?, v?)$, and $broadcast_i(m!)$.

An action $r_i(x?, v!)$ is an abbreviated action of blocked sequential actions: First, an input port loc_i is executed whenever a location $x?$ for a read action is received through the input port loc_i. Second, a read operation schema $Read_i$ is executed by P_i. Finally, the value $v!$ stored in $M_i(x?)$ is sent to P_i through an output port $\overline{val_i}$.

An action $w_i(x?, v?)$ is also an abbreviated action of blocked sequential actions: First, an input port $heap_i$ is executed whenever a value $v?$ to a location $x?$ for a write action is received through the input port $heap_i$. Second, a write operation schema $Write_i$ is executed by P_i.

We assume that these sequential actions in $r_i(x?, v!)$ and $w_i(x?, v?)$ 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.

$$r_i(x?, v!) \equiv loc_i(x?).Read_i.\overline{val_i}(v!)$$
$$w_i(x?, v?) \equiv heap_i(x?, v?).Write_i$$

A broadcast action $broadcast_i(m!)$ is an abbreviation action that sends a message $m!$ to all other processes through all output ports $\overline{pipe}$ except $\overline{pipe_i}$ as follows:

$$broadcast_i(m!) \equiv Send_i.\ \overline{pipe_1}(m!).\cdots.\overline{pipe_{i-1}}(m!).\ \overline{pipe_{i+1}}(m!).\cdots.\overline{pipe_k}(m!)$$

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

$$\begin{aligned} P_i \stackrel{\text{def}}{=}\ & r_i(x?, v!).P_i + \\ & w_i(x?, v?).P_i + \\ & broadcast_i(message!).P_i + \\ & pipe_i(message?).Receive_i.P_i + \\ & Apply_i.P_i \end{aligned}$$

Each process is connected with all other processes through input ports $pipe_1 \cdots pipe_k$ and output ports $\overline{pipe_1} \cdots \overline{pipe_k}$ to communicate the information about local writes to local memory. Then we specify a causal memory $C\mathcal{M}$ in CCS as follows:

$$K = \{pipe_1, \cdots, pipe_k\}$$
$$C\mathcal{M} \equiv id(1).InitP_1.\cdots.id(k).InitP_k.\ (P_1 \mid \cdots \mid P_k) \setminus K$$

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], Ahamad and Hutto prove the following Lemma 1,2 and Theorem 3.

Lemma 1: *Let H be a history of the implementation, $ts(o)$ be the timestamp of an operation o, and o_1 and o_2 be two operations such that $o_1 \rightsquigarrow o_2$. Then $ts(o_1) \preceq ts(o_2)$. Furthermore, if o_2 is a write operation by P_i, $ts(o_1)[i] < ts(o_2)[i]$, so $ts(o_1) \prec ts(o_2)$.*

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

Theorem 3: *Let H be a history of the implemen-*

tation. Then H is causal.

Proof : An inspection of operations $Read_i$, $Write_i$ 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,

$\forall i : 1 \ldots n \bullet \forall o \in A^H_{i+w} \bullet \exists per_i(o) \in S_i \; of \; A^H_{i+w}$

An inspection of operation schemas $Read_i$, $Write_i$ and $Apply_i$ shows that a local memory M_i is updated such that $M_i' = M_i \oplus \{x? \mapsto v?\}$ by P_i and the value $v!$ such that $v! = M_i x?$ 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:

$j : 1 \ldots n, \forall i : 1 \ldots n \bullet$
$(\forall r_i(x_l, v_l) \in S_i \; of \; A^H_{i+w} \bullet$
$((\exists per_i(w_j(x_l, v_k)) \in S_i \; of \; A^H_{i+w},$
$per_i(w_j(x_l, v_m)) \notin \omega \bullet$
$\langle per_i(w_j(x_l, v_k)).E_j, s_j \rangle$
$\xrightarrow{per_i(w_j(x_l,v_k))\omega} \langle r_i(x_l, v_l).E_i, s_i \rangle)$
$\vee$
$(((\exists per_i(w_j(x_l, v_m)) \in S_i \; of \; A^H_{i+w},$
$per_i(w_j(x_l, v_m)) \notin \omega \bullet$
$\langle start.E_0, s_0 \rangle \xrightarrow{\omega} \langle per_i(r_i(x_l, v_l)).E_i, s_i \rangle)$
$\vee \; per_i(w_j(x_l, v_m)) \notin S_i \; of \; A^H_{i+w} \Rightarrow v_l = \bot))))$

The state s_j' has a maple $x_l \mapsto v_k$ is not updated by P_i between the state s_j' and the state s_i, because $w_j(x_l, v_m) \notin \omega$. Thus $v_l = v_k (= M_i \, x_l)$.

Let o_1 and o_2 be operations in A^H_{i+w} such that $\rightsquigarrow$. By Lemma 1, $ts(o_1) \prec ts(o_2)$. One of the following case must holds. We assume that $j \neq i$.

- $o_1 \overset{i}{\rightarrow} o_2$ by p_i. An inspection of operation schemas $Read_i$ and $Write_i$ shows that these operations are concatenated to L_i and S_i in the same oder in which these operation are executed by P_i. Then o_1 precedes o_2 in S_i.
- $o_1 \overset{j}{\rightarrow} o_2$ by p_j. This means that o_1 and o_2 are both writes. By Lemma 1, $ts(o_1) \prec ts(o_2)$. An inspection of operation schema $Receive_i$ and $Apply_i$ shows that o_1 is applied by P_i before o_2. Then o_1 precedes o_2 in S_i.
- $o_1 \mapsto o_2$ by p_i. This means that o_1 is a write operation and o_2 is a read operation. By Lemma 1, $ts(o_1) \prec ts(o_2)$. An inspection of operation schema $Receive_i$, $Apply_i$ and $Read_i$ shows that o_1 is applied by P_i before o_2. Then o_1 precedes o_2 in S_i.
- $o_1 \rightsquigarrow o' \rightsquigarrow o_2$
$\exists o_1, o', o_2 \in A, \omega_1, \omega_2 \in A^* \bullet$
$\dfrac{\langle o_1.E_1, s_1 \rangle \xrightarrow{\omega_1} \langle o'.E', s' \rangle \quad \langle o'.E', s' \rangle \xrightarrow{\omega_2} \langle o_2.E_2, s_2' \rangle}{\langle o_1.E, s_1 \rangle \xrightarrow{\omega_1\omega_2} \langle o_2.E_2, s_2 \rangle}$

Thus, the following condition holds.

$j : 1 \ldots n, \forall i : 1 \ldots n \bullet$
$((\forall o_1, o_2 \in L_j \; of \; A^H_{i+w} \bullet \exists \omega_1, \omega_2 \in A^* \bullet$
$\langle o_1.E_1, s_1 \rangle \xrightarrow{\omega_1} \langle o_2.E_2, s_2 \rangle) \Rightarrow$
$\langle per_i(o_1).E_1', s_1' \rangle \xrightarrow{\omega_2} \langle per_i(o_2).E_2', s_2' \rangle)$
$\wedge$
$(\forall r_i(x_l, v_l) \in S_i \; of \; A^H_{i+w} \bullet \exists \omega_1, \omega_2 \in A^* \bullet$
$\langle w_j(x_l, v_l).E_j, s_j \rangle \xrightarrow{\omega_1} \langle r_i(x_l, v_l).E_i, s_i \rangle) \Rightarrow$
$\langle per_i(w_j(x_l, v_l)).E_j', s_j' \rangle \xrightarrow{\omega_2} \langle r_i(x_l, v_l).E_i', s_i' \rangle)))$

Thus, the proof is complete.

7. Conclusion

In this paper we described causal memory using the formal specification technique proposed by Taguchi and Araki[1]. We presented an extension of the state-based CCS 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 CCS 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.

Acnowledgements

We would like to thank Keijiro Araki at Kyushu University.

References

1) Kenji Taguchi and Keijiro Araki. The State-based CCS Semantics for Concurrent Z Specification. In *Proc. of the 1st Int'l Conf. of Formal Engineering Methods*, pages 283–292. IEEE, November 1997.

2) R.Milner. *Communication and Concurrency.* Prentice Hall, 1989.

3) Phillip W. Hutto and Mustaque Ahamad. Slow memory: Weakening consistency to enhance concurrency in distributed shared memories. In *Proc. of the 10th Int'l Conf. on Distributed Computing Systems*, pages 302–311, May 1990.

4) Mustaque Ahamad, Gil Neiger, Prince Kohli, James E. Burns, and Phillip W. Hutto. Causal memory: Definitions, implementation and programming. Technical Report 93/55, College of Computing, Georgia Institute of Tecnology, September 1993.

5) Keith Marzullo and Laura Sabel. Using consistent subcuts for detecting stable properties. In *Proc. of the 5th Workshop on Distributed Algorithms and Graphs*, October 1991.

6) Keith Marzullo and Gil Neiger. Detection of global state predicates. Technical Report 91/39, College of Computing, Georgia Institute of Tecnology, 1991.

7) Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

8) Friedmann Mattern. Virtual time and global states of distributed systems. In Michel Cosnard, Patrice Quinton, Yves Robert, and Michel Raynal, editors, *Proc. of the 10th Int'l Workshop on Parallel and Distributed Algorithms*, pages 215–226. North-Holland, October 1988.

9) Colin Stirling. Modal and temporal logic for processes. In *Logic for Concurrency*, number 1043 in LNCS, pages 149–237. Springer-Verlag, 1996.