Vol. 40 No. SIG 2 (TOM 1) 情報処理学会論文誌 Feb. 1999

*Regular Paper*

# 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 and verify memory consistency models for shared-memory multiprocessor systems, focusing on the causal memory consistency model, by use of a formal method proposed by Taguchi and Araki. This formal method includes the combination of the Z notation and value-passing CCS (Calculus of Communicating Systems), and the state-based CCS semantics which has the ability to describe the evolution of processes and the transition of states simultaneously. So we specify separately the functional aspects and the concurrent aspects of the causal memory in the Z notation and value-passing CCS respectively and define the causal memory consistency model in terms of the state-based CCS semantics. We also verify that the specified causal memory meets the defined causal memory consistency model.

## 1. Introduction

DSM (Distributed Shared Memory) systems are a recent trend in parallel computer architectures and system software. Various memory consistency models have been proposed for more efficient use of these systems. Memory consistency models define the behavior of multiple memory accesses in DSM systems. For example, in the sequential consistency model, any series of memory operations must be observed in the same order by different processors. Processor consistency models sometimes allow a read operation to pass by previous write operations. In the case of release consistency models, any memory operations can be seen in arbitrary orders if they are not issued from critical sections of a given parallel program.

In general, the weaker memory consistency models are, the more complicated the behavior of memory requests in the resultant DSM systems is. Therefore, at some point, a formal specification of such memory consistency models will be required to understand and compare the conventional memory consistency models as well as explore new models.

To abstract the memory consistency models, two aspects of the models, *functional aspects* which model states and operations as state transitions and *concurrency aspects* which specify reactions with the environment and synchronization of operations by communication, are discussed in this paper.

Taguchi and Araki[1] proposed a formal method which combines the Z notation[2] and value-passing CCS (Calculus of Communicating Systems)[3], a variant of CCS which allows value passing between processes.

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 a suitable vehicle for modeling mathematical structures of concurrency aspects. However CCS has no explicit modeling facilities for states and operations. Therefore, the combination of the Z notation and CCS, which complement each other, would result in a versatile specification language[1),4)].

In this formal method, functional aspects and concurrency aspects are separated for the design of information systems. Taguchi and Araki advocate the use of Z in the specification of functional aspects and the use of value-passing CCS in the specification of concurrency aspects. In order to provide a sound theoretical basis for this formal method, they proposed the state-based CCS semantics. The main characteristic of these semantics is its ability to describe the evolution of processes and transition of states simultaneously.

†1 Nara Institute of Science and Technology
†2 Keihanna Interaction Plaza Inc.
†3 Graduate School of Information Science and Electrical Engineering, Kyushu University
†4 Wakayama University Faculty of Systems Engineering

Modal and temporal logics have been used for specifying and verifying properties of concurrent systems. They are also used for describing the capabilities of processes in process algebra[5]. Taguchi and Araki proposed a Hennessy-Milner logic for processes in value-passing CCS which enables us to express properties such as liveness and safety ascribed both to states and to actions[1].

In this paper we formally specify and verify memory consistency models for shared-memory multiprocessor systems, focusing on the causal memory consistency model, using the formal method above proposed by Taguchi and Araki.

Causal memory is an implementation of the memory mechanism which satisfies the causal memory consistency model: 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 causal memory have already been presented by Ahamad and Hutto[6]. Regardless of their results, they are inefficient for us to formalize every memory consistency model since they described only program order and restrictions between operations and vector clocks using algebra, pseudo-code and natural language and we have to explicitly specify functional and concurrency aspects, which are important factors for us to analyze and design DSM.

In contrast, we specify separately the state aspects and the concurrent aspects of the causal memory in the Z notation and value-passing CCS respectively and define the causal memory consistency model in terms of the state-based CCS semantics. We also verify that the specified causal memory meets the defined causal memory consistency model using the state-based CCS semantics.

This paper is structured as follows. In section 2, weak vector clocks[7),8)] based on the causally-precedes relation defined by Lamport[9)] is described. In section 3, the state-based CCS semantics is explained. In section 4, definition of the causal memory consistency model in terms of the state-based CCS semantics is given. In section 5, a description of causal memory is described by using the combination of the Z notation and value-passing CCS. Verification of causal memory is presented in section 6. Finally, we conclude and indicate our future works in section 7.

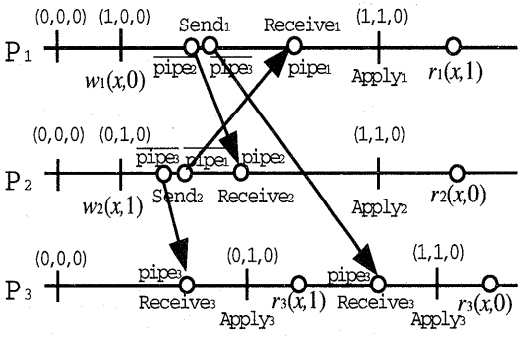



**Fig. 1** A history of events in a causal memory system

## 2. Weak Vector Clocks

*Vector clocks*[10)] are used in distributed systems to determine whether a pair of events $e_i$, $e_j$ have a causal relation denoted by $e_i \rightarrow e_j$ where $\rightarrow$ is the causally-precedes relation defined by Lamport[9)]. Using the vector clocks, a timestamp is recorded when any event is detected, and the causal relationship of pairs of events are 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 relations hold:

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

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

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

With the traditional vector clocks, the local counter $t[i]$ of a process $P_i$ increases whenever $P_i$ executes an event. In contrast, with *weak vector clocks*[7)], $t[i]$ increases only when $P_i$ executes an event that potentially leads to a change in 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, all other processes, $P_j$, learn that the process $P_i$ has potentially changed some properties and update their local state and vector timestamp. In the case of weak vector clocks, $P_j$ updates its vector timestamp $t_j$ as follows:

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

Figure 1 illustrates a history of events in the causal memory system described in section 5 which adopts weak vector clocks. Process $P_i$ increases its local counter $t_i[i]$ only

when $P_i$ executes a write operation $w_i(x, v)$ that changes the value of its local memory denoted by $M_i(x) = v$. $P_i$ sends a broadcast message of $M_i(x) = v$ and $t_i$. Receiving the broadcast message, all other processes, $P_j$, execute an apply operation for the consistency of $M_i(x) = v$ to its local memory and its local vector timestamp. Thus the following relations hold.

$t_1(w_1(x,0)) \preceq t_2(r_2(x,0)) \Leftrightarrow w_1(x,0) \rightarrow r_2(x,0)$
$t_1(w_1(x,0)) \preceq t_3(r_3(x,0)) \Leftrightarrow w_1(x,0) \rightarrow r_3(x,0)$
$t_2(w_2(x,1)) \preceq t_1(r_1(x,1)) \Leftrightarrow w_2(x,1) \rightarrow r_1(x,1)$
$t_2(w_2(x,1)) \preceq t_3(r_3(x,1)) \Leftrightarrow w_2(x,1) \rightarrow r_3(x,1)$

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

Taguchi and Araki combine the syntax of the Z notation and value-passing CCS and then define a labeled transition system, called the state-based CCS semantics, that reflects the state transitions of Z variables and evolutions of value-passing CCS processes simultaneously and give transition rules for all operations[1]. In this section the state-based CCS semantics and its transition rules are explained. An example of state transitions and evolutions of processes using the transition rules will be shown in section 5.3.

### 3.1 Labeled Transition Systems

In[3], 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 $\alpha$ is denoted by the following transition relation:

$$E \xrightarrow{\alpha} E'$$

Taguchi and Araki regard operation schemas in Z as transitions from old states to new states so they provide the operational semantics of Z in terms of the following labeled transition system:

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

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 $\alpha$ is denoted by the following transition relation:

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

### 3.2 The State-Based CCS Semantics

In addition, they provide the operational semantics of the combination language of the Z notation and value-passing 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 distinctions 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 $\alpha$ 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 $\Theta$ is the first-order representation of an operation schema $\alpha$.

### 3.3 Transition Rules

In the state-based CCS semantics it is possible to access variables of the Z specification within value-passing CCS expression. But there are the following restrictions. The state of the Z specification can only be changed by Z operation schemas and only input and output variables defined in the Z specification can be used as variables within value-passing CCS expression. Thus, the action which reflects the variables can not be used within value-passing CCS expression.

**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)**

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

Z has a convention for the use of variables. A variable with ?, e.g., $x?$ is regarded as an input variable and a variable with !, e.g., $x!$ as an output variable. So in order to receive a value for an input variable of the Z specification via an input port from the environment, say $\alpha(x?)$ is used and in order to send a value for an output variable of the Z specification via an output port to the environment, say $\overline{\alpha}(x!)$ is used respectively as the following prefix operators.

**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)**

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

If the following prefix operator is a prefix op-

erator(2) and (3), the following s' is s.

**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)**

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

When an output value, say c, is communicated from an output port $\overline{\alpha}$ to an input port $\alpha$, the following rule is applied.

**Concurrent Composition (2)**

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

When the action $\alpha$ and $\overline{\alpha}$ do not involve values, the resulting communication is a synchronization. In such a case the following rule is applied.

**Concurrent Composition (3)**

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

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

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

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

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

For example, let $\omega$ 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 an 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**

$$\frac{}{\langle E, s\rangle \xrightarrow{\varepsilon} \langle E, s\rangle} \qquad \frac{\langle E, s\rangle \xrightarrow{\alpha} \langle E', s'\rangle \quad \langle E', s'\rangle \xrightarrow{\omega} \langle F, s''\rangle}{\langle E, s\rangle \xrightarrow{\alpha\omega} \langle F, s''\rangle}$$

## 4. Definition of Causal Memory Consistency Model

This section explains the causal memory consistency model proposed by Hutto and Ahamad in[6),11)], and then defines the model in terms of the state-based CCS semantics.

### 4.1 Shared Memory Parallel Computer Model

In[6)], 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 $H$.
- $S_i$ 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 $\omega$ be a sequence $A^*$ of operations in $H$

with $\varepsilon$ as an empty trace. Let $per_i(o)$ be an operation that $P_i$ "*perceives*"as 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 the operation schema $Apply_i$ and abbreviations $r_i(x?,v!)$ and $w_i(x?,v?)$ that will be described in section 5). Let $\langle \mathcal{E} \times St, Act \cup Op, \{\xrightarrow{\alpha} \mid \alpha \in Act \cup Op\}\rangle$ be a state-based CCS semantics of the shared memory parallel computer model above. $E_0$,$E_1$,$E_1'$,$E_2$,$E_2'$,$E_i$,$E_i'$,$E_j$,$E_j'$ range over $\mathcal{E}$ and $s_0$,$s_1$,$s_1'$,$s_2$,$s_2'$,$s_i$,$s_i'$,$s_j$,$s_j'$ over $St$. Now we define the above definitions in terms of the CCS semantics as follows respectively:

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

### 4.2 Definition of Causal Memory Consistency Model

In[6], 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 $v = \bot$; 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 $A_{i+w}^H$ that respects $\rightsquigarrow$.

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

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

By iteration of applying **trace** transition rules:

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

Then, if there is a trace $\omega_1$ corresponding to $o_1 \rightsquigarrow o'$ and a trace $\omega_2$ 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 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 model in terms of the CCS semantics as follows:

*A history H is causal* $\Leftrightarrow$
$j : 1 \ldots n, \forall i : 1 \ldots n \bullet$
$(\,(\,(\forall o \in A_{i+w}^H \bullet \exists per_i(o) \in S_i \text{ of } A_{i+w}^H)$
$\wedge$
$(\forall r_i(x_l?, v_l!) \in S_i \text{ of } A_{i+w}^H \bullet$
$((\exists per_i(w_j(x_l?, v_l?)) \in S_i \text{ of } A_{i+w}^H,$
$per_i(w_j(x_l?, v_m?)) \notin \omega \bullet$
$\langle per_i(w_j(x_l?, v_l?)).E_j, s_j\rangle$
$\xrightarrow{per_i(w_j(x_l?,v_l?))\omega} \langle r_i(x_l?, v_l!).E_i, s_i\rangle)$
$\vee\ (((\exists per_i(w_j(x_l?, v_m?)) \in S_i \text{ of } A_{i+w}^H,$
$per_i(w_j(x_l?, v_m?)) \notin \omega \bullet$
$\langle start.E_0, s_0\rangle \xrightarrow{\omega} \langle r_i(x_l?, v_l!).E_i, s_i\rangle)$
$\vee\ per_i(w_j(x_l?, v_m?)) \notin S_i \text{ of } A_{i+w}^H)$
$\Rightarrow v_l = \bot))\,)\,)$
$\wedge$
$(\forall o_1, o_2 \in L_j \text{ of } A_{i+w}^H, \exists \omega_1, \omega_2 \in A^* \bullet$
$\langle o_1.E_1, s_1\rangle \xrightarrow{\omega_1} \langle o_2.E_2, s_2\rangle \Rightarrow$
$\langle per_i(o_1).E_1', s_1'\rangle \xrightarrow{\omega_2} \langle per_i(o_2).E_2', s_2'\rangle)$

$$\wedge$$
$$(\forall r_i(x_l?, v_l!) \in S_i \text{ of } A^H_{i+w}, \exists \omega_1, \omega_2 \in A^* \bullet$$
$$\langle w_j(x_l?, v_l?).E_j, s_j\rangle \xrightarrow{\omega_1} \langle r_i(x_l?, v_l!).E_i, s_i\rangle \Rightarrow$$
$$\langle per_i(w_j(x_l?, v_l?)).E'_j, s'_j\rangle \xrightarrow{\omega_2} \langle r_i(x_l?, v_l!).E_i, s_i\rangle))$$

## 5. A Description of Causal Memory

In this section, using the combination of the Z notation and value-passing CCS, a formal specification of causal memory, which was proposed by Ahamad et al[6], is described.

First, the functional aspects which model the states and operation schemas of each process are specified in the Z notation. Weak vector clocks are adopted here as logical time of distributed systems like Marzullo and Neiger did[8]. In addition, we use a vector timestamp which is represented by the sequence data type using the Z notation. Second, the concurrency aspects of causal memory are specified in value-passing CCS. Third, an example of the state transitions and evolutions of processes of causal memory using the state-based CCS semantics is shown.

### 5.1 Specifying the Functional Aspects of Causal Memory 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 $pn_i$, 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 $L_i$ which is a set of read and write operations by $P_i$, 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:

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

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

$NumOfProcesses : \mathbb{N}_1$
$MaxOutQueue, MaxInQueue : \mathbb{N}_1$
$MaxSerial, MaxLocalHis : \mathbb{N}_1$
$priority\_queue : (\text{seq}\ write\_tuple) \times write\_tuple \nrightarrow \text{seq}\ write\_tuple$

$s_i$
$pn_i : \mathbb{N}_1$
$M_i : \mathcal{M} \nrightarrow (Val \cup \{\bot\})$
$t_i : \text{seq}\,\mathbb{N}$
$OutQueue_i : \text{seq}\ write\_tuple$
$InQueue_i : \text{seq}\ write\_tuple$
$L_i : \text{seq}\ A$
$S_i : \text{seq}\ A$
---
$\#t_i = NumOfProcesses$
$\#OutQueue_i \leq MaxOutQueue$
$\#InQueue_i \leq MaxInQueue$
$\#L_i \leq MaxLocalHis$
$\#S_i \leq MaxSerial$

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

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 $P_i$. The label $r_i(x?, v!)$ of the read operation is added to a local execution history $L_i$ and a serialization $S_i$.

$InitP_i$
$s'_i$
$pn_i? : \mathbb{N}_1$
---
$pn'_i = pn_i?$
$M'_i = \lambda x : \mathcal{M} \bullet \bot$
$t'_i = \lambda n : 1 .. NumOfProcesses \bullet 0$
$OutQueue'_i = \langle\rangle$
$InQueue'_i = \langle\rangle$
$L'_i = \langle\rangle$
$S'_i = \langle\rangle$

$Read_i$
$\Delta s_i$
$x? : \mathcal{M}$
$v! : Value$
---
$v! = M_i x?$
$pn'_i = pn_i$
$M'_i = M_i$
$t'_i = t_i$
$OutQueue'_i = OutQueue_i$
$InQucue'_i = InQueue_i$
$L'_i = L_i \frown \langle r_i(x?, v!)\rangle$
$S'_i = S_i \frown \langle r_i(x?, v!)\rangle$

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. The label $w_i(x?, v?)$ of the write operation is appended to $L_i$ and $S_i$.

The information about local write operations in $OutQueue_i$ must be sent 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 $OutQueue_i$.

$Write_i$

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

---

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

$Send_i$

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

---

$Outqueue_i \neq \langle\rangle$
$pn_i' = pn_i$
$M_i' = M_i$
$t_i' = t_i$
$message! = head\ OutQueue_i$
$OutQueue_i' = tail\ OutQueue_i$
$InQueue_i' = InQueue_i$
$L_i' = L_i$
$S_i' = S_i$

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.

The information in $InQueue_i$ is used to update the view of a process to memory by an operation schema $Apply_i$. $Apply_i$ 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 $j$th) are less than or equal to those of $t_i$ and if the $j$th component of $t_j$ is more than the $j$th component of $t_i$ by exactly 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 ,$w_j(x_j, v_j)$, will be the most recent write operation of the write-into order relation preceding the following read operation, $r_i(x_j, v_j)$, $w_j(x_j, v_j) \rightarrow r_i(x_j, v_j)$ where the vector timestamp of $w_j(x_j, v_j)$ is less than or equal to that of $r_i(x_j, v_j)$. The label of the write operation, $w_j(x_j, v_j)$, is appended to $S_i$.

$Receive_i$

$\Delta s_i$
$message? : write\_tuple$

---

$pn_i' = pn_i$
$M_i' = M_i$
$t_i' = t_i$
$OutQueue_i' = OutQueue_i$
$InQueue_i' = priority\_queue($
$\qquad InQueue_i, message?)$
$L_i' = L_i$
$S_i' = S_i$

$Apply_i$

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

---

$InQueue_i \neq \langle\rangle$
$pn_i' = pn_i$
$(j, x_j, v_j, t_j) = head\ InQueue_i$
$k : 1 \,..\, \#t_i \mid k \neq j \bullet t_j\, k \leq t_i\, k$
$\qquad \wedge\ t_j\, j = t_i\, j + 1$
$M_i' = M_i \oplus \{x_j \mapsto v_j\}$
$t_i'\, j = t_j\, j$
$k : 1 \,..\, \#t_i \mid k \neq j \bullet t_i'\, k = t_i\, k$
$OutQueue_i' = OutQueue_i$
$InQueue_i' = tail\ InQueue_i$
$L_i' = L_i$
$S_i' = S_i \frown \langle w_j(x_j, v_j) \rangle$

### 5.2 Specifying the Concurrency Aspect of Causal Memory in CCS

In this section, we specify the concurrency aspects of causal memory in value-passing CCS.

We assume that the causal memory has $n$ processes. Each of the processes is connected with all other processes through input ports $pipe_1 \cdots pipe_n$, and output ports $\overline{pipe_1} \cdots \overline{pipe_n}$.

Each process $P_i$ consists of six operation

schemas specified above in Z and four basic input or output ports; $id_i$, $loc_i$, $val_i$, $heap_i$, 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 $loc_i$, $val_i$, $heap_i$, and $pipe_i$ are used for input or output ports in the following abbreviated actions: $r_i(x?, v!)$, $w_i(x?, v?)$, $broadcast_i(message!)$ and $receive_i(message?)$.

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

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

An action $w_i(x?, v?)$ is also an abbreviation of blocked sequential actions: First, an input port $heap_i$ is executed by $P_i$ whenever values for input variables $x?$ and $v?$ are received via the input port $heap_i$ from the environment. Second, a write operation schema $Write_i$ is executed by $P_i$.

$$w_i(x?, v?) \equiv heap_i(x?, v?).Write_i$$

An action $broadcast_i(message!)$ is an abbreviation of blocked sequential actions: First, a send operation schema $Send_i$ is executed by $P_i$. Second, a message $message!$ is sent to all other processes via all output ports $\overline{pipe}$ except $\overline{pipe_i}$ as follows:

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

An action $receive_i(message?)$ is an abbreviation of blocked sequential actions: First, an input port $pipe_i$ is executed by $P_i$ whenever a message for an input variable $message?$ is received via the input port $pipe_i$ from the output port $\overline{pipe_i}$ as a co-named port of this input port $pipe_i$. In CCS, this action is called $\tau$ action. Second, a receive operation schema $Receive_i$ is executed by $P_i$ as follows:

$$receive_i(message?) \equiv pipe_i(message?).\ Receive_i$$

We then specify the concurrency aspects of the processes $P_i$.

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

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

$$\begin{aligned} K &= \{pipe_1, \cdots, pipe_n\} \\ \mathcal{CM} &\equiv id(pn_1?).InitP_1.\cdots \\ &\quad .id(pn_n?).InitP_n.(P_1 \mid \cdots \mid P_n) \setminus K \end{aligned}$$

### 5.3 Example of $\mathcal{CM}$ Transitions

Now we show an example of the state transitions and evolutions of processes of the causal memory $\mathcal{CM}$ which has two processes, $P_1$ and $P_2$, using the state-based CCS semantics as follows.

Let $\uparrow$ be an indicator of locations, say $x \uparrow \in \mathcal{M}$ indicates a location $x$.

$$\begin{aligned} & \langle \mathcal{CM}, s_0 \rangle \\ \xrightarrow{id(1)}\ & \langle InitP_1.id(pn_2?).InitP_2.(P_1 \mid P_2) \setminus K, s_0' \rangle \\ \xrightarrow{InitP_1}\ & \langle id(pn_2?).InitP_2.(P_1 \mid P_2) \setminus K, s_1' \rangle \\ \xrightarrow{id(2)}\ & \langle InitP_2.(P_1 \mid P_2) \setminus K, s_2' \rangle \\ \xrightarrow{InitP_2}\ & \langle (P_1 \mid P_2) \setminus K, s_3' \rangle \\ \xrightarrow{heap_1(x\uparrow,1)}\ & \langle (Write_1.P_1 \mid P_2) \setminus K, s_4' \rangle \\ \xrightarrow{Write_1}\ & \langle (P_1 \mid P_2) \setminus K, s_5' \rangle \\ \xrightarrow{Send_1}\ & \langle (\overline{pipe_2}(message!).P_1 \mid P_2) \setminus K, s_6' \rangle \\ \xrightarrow{\tau}\ & \langle (P_1 \mid Receive_2.P_2) \setminus K, s_7' \rangle \\ \xrightarrow{Receive_2}\ & \langle (P_1 \mid P_2) \setminus K, s_8' \rangle \end{aligned}$$

Last $\tau$ action is applied using the following transitions.

$$\langle\ \overline{pipe_2}(message!).P_1, s_7 \rangle \xrightarrow{\overline{pipe_2}((1, x\uparrow, 1, \langle 1,0 \rangle))} \langle P_1, s_7 \rangle \quad (1)$$

$$\langle\ P_2, s_7 \rangle \xrightarrow{pipe_2((1, x\uparrow, 1, \langle 1,0 \rangle))} \langle Receive_2.P_2, s_7' \rangle \quad (2)$$

$$\langle\ (\overline{pipe_2}(message!).P_1 \mid P_2), s_7 \rangle \xrightarrow{\tau} \langle (P_1 \mid Receive_2.P_2), s_7' \rangle \quad (3)$$

$$\langle\ (\overline{pipe_2}(message!).P_1 \mid P_2) \setminus K, s_7 \rangle \xrightarrow{\tau} \langle (P_1 \mid Receive_2.P_2) \setminus K, s_7' \rangle \quad (4)$$

$$\frac{(1)\quad (2)}{\dfrac{(3)}{(4)}}\ (\tau \notin K)$$

The state transitions as variable components are shown as follows:

We adopt notations $\oplus$ and $\mapsto$ in Z in order to

describe how states are changed. Given two functions $f$ and $g$, $f \oplus g$ denote the relational overriding of f with g in[2]. $x \mapsto v$ denotes a mapping from a variable $x$ to a value $v$.

$s_0 = \{\}$
$s_0' = s_1 = \{pn_1? \mapsto 1\}$
$s_1' = s_2 = s_1 \oplus \{pn_1 \mapsto 1, M_1 \mapsto \langle \bot \cdots \bot \rangle,$
$t_1 \mapsto \langle 0,0 \rangle, OutQueue_1 \mapsto \langle\rangle,$
$InQueue_1 \mapsto \langle\rangle, L_1 \mapsto \langle\rangle, S_1 \mapsto \langle\rangle\}$
$s_2' = s_3 = s_2 \oplus \{pn_2? \mapsto 2\}$
$s_3' = s_4 = s_3 \oplus \{pn_2 \mapsto 2, M_2 \mapsto \langle \bot \cdots \bot \rangle,$
$t_2 \mapsto \langle 0,0 \rangle, OutQueue_2 \mapsto \langle\rangle,$
$InQueue_2 \mapsto \langle\rangle, L_2 \mapsto \langle\rangle, S_2 \mapsto \langle\rangle\}$
$s_4' = s_5 = s_4 \oplus \{x? \mapsto x\uparrow, v? \mapsto 1\}$
$s_5' = s_6 = s_5 \oplus \{M_1 \mapsto M_1 \oplus \{x\uparrow \mapsto 1\},$
$t_1 \mapsto \langle 1,0 \rangle,$
$OutQueue_1 \mapsto \langle (1, x\uparrow, 1, \langle 1,0 \rangle)\rangle,$
$L_1 \mapsto \langle w_1(x\uparrow, 1)\rangle, S_1 \mapsto \langle w_1(x\uparrow, 1)\rangle\}$
$s_6' = s_7 = s_6 \oplus \{message! \mapsto (1, x\uparrow, 1, \langle 1,0 \rangle),$
$OutQueue_1 \mapsto \langle\rangle\}$
$s_7' = s_8 = s_7 \oplus \{message? \mapsto (1, x\uparrow, 1, \langle 1,0 \rangle)\}$
$s_8' = s_9 = s_8 \oplus \{InQueue_2 \mapsto \langle (1, x\uparrow, 1, \langle 1,0 \rangle)\rangle\}$

## 6. Verification of Causal Memory

In this section, we verify that the causal memory described in section 5 meets the causal memory consistency model described in section 4 using the state-based CCS semantics.

In[6], 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 $ts(o_1) \prec ts(o_2)$.*

*Proof* : Let $A$ be a set of operation schemas and actions as described in section 5. Let $t_i(o)$ and $t_i'(o)$ be vector timestamps of $P_i$ when an event $o$ starts and finishes, respectively. Note that the process $P_i$ has the state, operations and actions described in section 5. Consider the following three cases:

• $o_1 \underset{i}{\rightarrow} o_2$
By definition of operation schemas of $P_i$ which adopt weak vector clocks, only $Write_i$ and $Apply_i$ operation schemas update local timestamp $t_i$ as follows:

$$t_i'(Write_i)[i] = t_i(Write_i)[i] + 1$$
$$t_i'(Apply_i)[j] = t_i(Apply_i)[j] + 1$$

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 $ts(o_1) \prec ts(o_2)$.

• $o_1 \mapsto o_2$
This means that $o_1$ is a write operation, say $w_j(x_j?, v_j?)$, and $o_2$ is a corresponding read operation, say $r_i(x_i?, v_i!)$.
$\exists\, w_j(x_j?, v_j?), r_i(x_i?, v_i!) \in S \text{ of } H,$
$\exists \omega \in A^* \bullet$
$\langle w_j(x_j?, v_j?).E_j, s_j\rangle \xrightarrow{\omega} \langle r_i(x_i?, v_i!).E_i, s_i\rangle$
$\wedge\ x_i? = x_j? \wedge v_i! = v_j?$
Consider traces of $\mathcal{CM}$ in CCS. We can get the following trace $\omega$ by applying transition rules iteratively:
$\exists \omega \in A^* \bullet$
$\omega = w_j(x_j?, v_j?) \ldots Send_j \ldots \overline{pipe_i}(message!)$
$\ldots pipe_i(message?) \ldots Receive_i \ldots Apply_i \ldots$
(For example,$w_2(x,1) \rightarrow r_1(x,1)$ in Figure 1.)
By applying $o_1 \underset{i}{\rightarrow} o_2 \Rightarrow ts(o_1) \preceq ts(o_2)$ as proved above, the following holds:

$ts(w_j(x_j?, v_j?)) \preceq ts(Send_j)$
$\preceq ts(\overline{pipe_i}(message!))$
$ts(pipe_i(message?)) \preceq ts(Receive_i)$
$\prec ts(Apply_i) \preceq ts(r_i(x_i?, v_i!))$

Note that there is a timestamp $ts(w_j(x_j?, v_j?))$ in $message!$. By use of the operation schema $Apply_i$, compare $ts(w_j(x_j?, v_j?))$ with $t'(Apply_i)$.

$k : 1 \ldots n \mid k \neq j$
$\bullet\ ts(w_j(x_j?, v_j?))[k] \leq t'(Apply_i)[k]$
$ts(w_j(x_j?, v_j?))[j] = t(Apply_i)[j] + 1$
$ts(w_j(x_j?, v_j?))[j] = t'(Apply_i)[j]$

So $ts(w_j(x_j?, v_j?)) \preceq t'(Apply_i) = ts(Apply_i) \preceq ts(r_i(x_i?, v_i!))$. Thus $ts(o_1) \preceq ts(o_2)$.

• There is some operation $o_1, o', o_2$ such that $o_1 \rightsquigarrow o' \rightsquigarrow o_2$. Let $\omega_1, \omega_2$ be traces that respect $\underset{i}{\rightarrow}$ or $\mapsto$. By iteratively applying the **trace** transition rule:

$$\frac{\langle o_1.E_1, s_1\rangle \xrightarrow{\omega_1} \langle o'.E', s'\rangle \quad \langle o'.E', s'\rangle \xrightarrow{\omega_2} \langle o_2.E_2, s_2\rangle}{\langle o_1.E, s_1\rangle \xrightarrow{\omega_1\omega_2} \langle o_2.E_2, s_2\rangle}$$

By the transitivity of $\preceq$, so $ts(o_1) \preceq ts(o_2)$.
Furthermore, if $o_2$ is a write operation by $P_i$, $ts(o')[i] \prec ts(o_2)[i]$, so $ts(o_1) \preceq ts(o') \prec ts(o_2)$.
□

**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.*

*Proof* : If $j = i$, an inspection of operation

schema $Write_i$ shows that the write is applied to its memory by $M_i' = M_i \oplus \{x? \mapsto v?\}$; for the remainder of the proof, assume that $j \neq i$. Since each $OutQueue_i$ is a finite first-in-first-out queue, an inspection of $\mathcal{CM}$ in CCS shows that, once $P_i$ has executed $w_j(x, v)$, the following trace transition exists by iteratively applying transition rules.

$\forall i : 1 \ldots n (i \neq j), \exists Receive_i \in A, \exists \omega \in A^* \bullet$
$\langle w_j(x_j?, v_j?).E_j, s_j\rangle \xrightarrow{\omega} \langle Receive_i.E_i, s_i\rangle$

Next the message! of the $w_j(x_j?, v_j?)$ is received by $P_i$ and is appended to $InputQueue_i$ which is a finite priority queue sorted by vector timestamps. Recall that $ts[j]$ is the number of write operations by $P_j$. Let $m$ be the sum of all the components in the vector clock of the message!. $\mathcal{CM}$ creates at most $m$ messages such that its vector timestamps $\preceq ts(w_j(x_j?, v_j?))$. Then the message! is inserted no deeper than the $m$th element from the head of $InputQueue_i$.

After that the following trace transition exists:

$\forall i : 1 \ldots n (i \neq j), \exists Apply_i \in A, \exists \omega \in A^* \bullet$
$\langle Receive_i.E_i, s_i\rangle \xrightarrow{\omega} \langle Apply_i.E_k, s_k\rangle$

Let $t$ be $P_i$'s vector clock in the state $s_k$ before $Apply_i$ applies the message! to its local memory. We must show that $ts(w_j(x_j?, v_j?))[k] \leq t[k]$ for all $k \neq j$ and that $ts(w_j(x_j?, v_j?))[j] = t[j] + 1$. Let $P_l$ be any process other than $P_j$ and $w'$ be the $(ts(w_j(x_j?, v_j?))[k])$th write by $P_l$. This means $P_j$ has applied $w'$ before $P_j$ executes $w_j(x_j?, v_j?)$. Therefore $ts(w') \prec ts(w_j(x_j?, v_j?))$. Thus, by induction, $P_i$ has already applied $w'$ before the state $s_k$. Once $P_i$ applies $w'$, $ts(w_j(x_j?, v_j?))[k] \leq t[k]$. Similarly the $(ts(w_j(x_j?, v_j?))[j] - 1)$th write by $P_j$ has been applied by $P_i$ before the state $s_k$. After that $ts(w_j(x_j?, v_j?))[j] = t[j] + 1$. Since $P_i$ executed operation schema $Apply_i$ infinitely often, $P_i$ eventually applies $w_j(x_j?, v_j?)$.

□

**Theorem 3:** *Let $H$ be a history of the implementation. 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,

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

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 sends the value that the most recent write operation schema has written. Thus, the following transitions exist:

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

The state $s_j'$ has a value $x_l \mapsto v_k$ in the local memory $M_i$. The value $x_l \mapsto v_k$ 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, $ts(o_1) \preceq ts(o_2)$. One of the following cases must hold. We assume that $j \neq i$.

- $o_1 \xrightarrow[i]{} o_2$ by $P_i$. An inspection of operation schemas $Read_i$ and $Write_i$ shows that the labels of these operations are concatenated to $L_i$ and $S_i$ in the same order in which these operation are executed by $P_i$. Therefore $o_1$ precedes $o_2$ in $S_i$.
- $o_1 \xrightarrow[j]{} o_2$ 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 $o_2$. Therefore $o_1$ precedes $o_2$ in $S_i$.
- $o_1 \rightsquigarrow o' \rightsquigarrow o_2$

$$\frac{\langle o_1.E_1, s_1\rangle \xrightarrow{\omega_1} \langle o'.E', s'\rangle \quad \langle o'.E', s'\rangle \xrightarrow{\omega_2} \langle o_2.E_2, s_2'\rangle}{\langle o_1.E, s_1\rangle \xrightarrow{\omega_1\omega_2} \langle o_2.E_2, s_2\rangle}$$

Thus, the following condition holds.

$j : 1 \ldots n, \forall i : 1 \ldots n \bullet$
$((\forall o_1, o_2 \in L_j \text{ of } A^H_{i+w}, \exists \omega_1, \omega_2 \in A^* \bullet$
$\langle o_1.E_1, s_1\rangle \xrightarrow{\omega_1} \langle o_2.E_2, s_2\rangle \Rightarrow$

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

Thus, the proof is complete.

□

## 7. Conclusion

In this paper we described a causal memory using the formal method proposed by Taguchi and Araki[1]. First, we described functional aspects using the Z notation so that we could define the state variables and operation schemas separately. Second, we described concurrency aspects using value-passing CCS so that we could define the processes, communications, and concurrency of processes. Third, we verified that the causal memory meets the causal memory consistency model.

As compared with only an algebra or a process algebra, this combination language is useful to us while we are designing causal memory, because we can describe the operation schemas in detail and independently and we could also express the concurrency of the processes. In verifying the system, we could consider the evolution of processes and the state transition separately. For example, it is easy to suppose that the system in which the following condition is removed from the $Receive_i$ operation schema meets PRAM consistency model[12].

$$k : 1 \,..\, \#t_i \mid k \neq j \bullet t_j\, k \leq t_i\, k$$

Much future work remains to be done. First, we should adopt this technique for other memory consistency models, especially release consistency models, and 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 the 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 formal techniques we propose.

DSM systems are the hot research field and a lot of implementations have been reported. We believe that our formal techniques can validate those numerous DSM systems from a formal viewpoint.

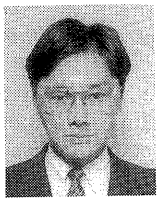
**Acnowledgements** We would like to thank Keijiro Araki at Kyushu University.

## References

1) Taguchi, K. and Araki, K.: The State-based CCS Semantics for Concurrent Z Specification, *Proc. of the 1st Int'l Conf. of Formal Engineering Methods*, IEEE, pp. 283–292 (1997).
2) J.Spivey: *The Z notation*, Prentice Hall, second edition (1992).
3) R.Milner: *Communication and Concurrency*, Prentice Hall (1989).
4) Galloway, A.J. and Stoddart, W.J.: An operational semantics for ZCCS, *Proc. of the 1st Int'l Conf. of Formal Engineering Methods*, IEEE, pp. 272–282 (1997).
5) Stirling, C.: Modal and Temporal Logic for Processes, *Logic for Concurrency*, LNCS, No. 1043, Springer-Verlag, pp. 149–237 (1996).
6) Ahamad, M., Neiger, G., Kohli, P., Burns, J. E. and Hutto, P. W.: Causal Memory: Definitions, Implementation and Programming, Technical Report 93/55, College of Computing, Georgia Institute of Tecnology (1993).
7) Marzullo, K. and Sabel, L.: Using Consistent Subcuts for Detecting Stable Properties, *Proc. of the 5th Workshop on Distributed Algorithms and Graphs* (1991).
8) Marzullo, K. and Neiger, G.: Detection of Global State Predicates, Technical Report 91/39, College of Computing, Georgia Institute of Tecnology (1991).
9) Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System, *Communications of the ACM*, Vol. 21, No. 7, pp. 558–565 (1978).
10) Mattern, F.: Virtual Time and Global States of Distributed Systems, *Proc. of the 10th Int'l Workshop on Parallel and Distributed Algorithms* (Cosnard, M., Quinton, P., Robert, Y. and Raynal, M.(eds.)), North-Holland, pp.215–226 (1988).
11) Hutto, P. W. and Ahamad, M.: Slow Memory: Weakening Consistency to Enhance Concurrency in Distributed Shared Memories, *Proc. of the 10th Int'l Conf. on Distributed Computing Systems*, pp. 302–311 (1990).
12) R.J.Lipton and J.S.Sandberg: PRAM:A Scalable Shared Memory, Technical Report CS-TR-180-88, Dept. of Computer Science, Princeton University (1989).

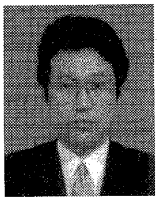
(Received July 2, 1998)
(Revised August 14, 1998)
(Accepted September 4, 1998)

**Shiro Takata** received his B.E., degree from Osaka University in 1979, and M.E., degree from Nara Institute of Science and Technology in 1995, respectively. From 1979 to 1993, he worked for CSK Corporation, where he engaged in development of application systems and a tool, named XPT-II, for developing knowledge engineering systems. From 1993, he has been working for Keihanna Interaction Plaza Inc., where he engaged in computer and network system management and in the interaction of culture, science and technology in Kansai Science City and he has been a student at Nara Institute of Science and Technology. His research interests include formal method, temporal logic, and distributed shared memory. He is a member of the IEEE Computer Society, IPSJ, JSSST, and JSAI.



**Kenji Taguchi** received his B.A., degree from Toyo University in 1981, B.A (Hons)., degree in 1984 and M.A., degree in 1987 from Victoria University of Wellington. From 1985 to 1987, he worked for CSK Research Corp., where he engaged in the development of expert systems shells and expert systems. From 1987 to 1990, he worked for Kansai Research Institute International., where he engaged in consulting for the research and development of AI. From 1990 to 1997, he worked for CSK Corp., where he involved in several R & D projects on AI systems and the formal methods. From 1997 he has been working as a research associate at Kyushu University, Graduate School of Information Science and Electrical Engineering, Department of Computer Science and Communication Engineering. His research interests include the formal methods, mobile calculi and mobile extensions of logic programming. He is a member of IPSJ, JSSST.

**Kazuki Joe** received the B.S. degree in mathematics from Osaka University in 1984, M.S. and PhD degree in information science from Nara Institute of Science and Technology in 1995 and 1996 respectively. He is currently an associate professor at Wakayama University. From 1984 to 1986, he was a software engineer of Japan DEC. From 1986 to 1990, he was a researcher of ATR Auditory and Visual Perception Research Lab. From 1991 to 1993, he was a senior researcher of Kubota corporation. From 1996 to 1997, he was an assistant professor at Nara Institute of Science and Technology. His research interests include parallel computer architectures, analytic modeling for parallel computers, parallelizing compilers, neural networks and image processing. He is a member of the IEEE Computer Society.



**Akira Fukuda** received the B.E., M.E., and Dr.Eng. degrees in computer science and communication engineering from Kyushu University in 1977, 1979, and 1985, respectively. From 1977 to 1981, he worked for the Nippon Telegraph and Telephone Corporation, where he engaged in research on performance evaluation of computer systems and queueing theory. From 1981 to 1991 and from 1991 to 1993, he worked for the Department of Information Systems and Department of Computer Science and Communication Engineering of Kyushu University, respectively. From 1994 he has been a Professor of Graduate School of Information Science of Nara Institute of Science and Technology. His research interests include parallel and distributed operating systems, parallelizing compilers, and performance evaluation of computer systems. He received 1990 IPSJ Research Award and 1993 IPSJ Best Author Award. He is a member of the ACM, the IEEE Computer Society, the IEICE, and the Operations Research Society of Japan.