NAIST-IS-DT9561021

博士論文

メモリ・コンシステンシ・モデルの形式的仕様記述と検証に関する研究

高田 司郎

1999年2月8日

奈良先端科学技術大学院大学
情報科学研究科 情報システム学専攻

本論文は奈良先端科学技術大学院大学情報科学研究科に
博士 (工学) 授与の要件として提出した博士論文である。

高田 司郎

審査委員：　福田 晃 教授
鳥居 宏次 教授
関 浩之 教授

メモリ・コンシステンシ・モデルの形式的仕様記述と検証に関する研究*

高田 司郎

内容梗概

本論文では，田口と荒木が提案した形式手法を用いて，分散共有メモリシステムの振る舞いを定義したメモリ・コンシステンシ・モデルとそれらの実現の形式的な仕様記述とその検証を行う．

メモリ・コンシステンシ・モデルは，プロセスが分散共有メモリシステムからデータを読み出すとき，多くの書き込まれた値の候補のなかからどれを返すかを決定するモデルである．分散共有メモリシステムの性能は，これらモデルに強く依存するため，多くのモデルが提案されている．これらモデルは，ストア命令やロード命令などに諸々のプログラム順序を定義してメモリアクセスを制約するものと，いつどのようにこれら命令の同期を取るべきかというプログラマの指定によりメモリアクセスを制約するものに大別される．そこで，本論文では，前者の代表としてコーザル・メモリ・コンシステンシ・モデルを，後者の代表としてリリース・コンシステンシ・モデルを取り上げる．

この形式手法では，Z の表記法と value-passing CCS (Calculus of Communicating Systems) の統合，プロセスの展開と状態遷移を同時に記述できる状態遷移に基づく CCS 意味論（S-CCS），および，遷移規則などが提案されている．本論文では，この S-CCS を用いて記述された一つの操作に関する遷移関係を複数の操作を持つトレースに関する遷移関係に自然に拡張して，分散共有メモリシステムの逐次実行列上の操作に関する遷移関係を表現する．また，このトレースは

*奈良先端科学技術大学院大学 情報科学研究科 情報システム学専攻 博士論文, NAIST-IS-DT9561021, 1999年2月8日.

上記の遷移規則を適用して導出することができ，このトレースを使って，分散共有メモリシステムの仕様記述の検証を行う．

まず，上記の二つのメモリ・コンシステンシ・モデルを S-CCS を用いて記述する．次に，これらモデルの分散共有メモリシステムの機能的な仕様は Z を用い，並行性の仕様は value-passing CCS を用いて分離して記述する．また，因果先行関係の比較には弱ベクトル時計を，メモリアクセスの同期にはセマフォを用いる．次に，これら分散共有メモリシステムの仕様記述が，それぞれのメモリ・コンシステンシ・モデルの要求を満足しているかどうかを検証する．

これら代表的な二つのモデルと実現の形式的仕様記述と検証を示すことで，この形式手法のメモリ・コンシステンシ・モデルに対する有効性を確認した．

キーワード

メモリ・コンシステンシ・モデル，分散共有メモリ，形式手法，Z，CCS，ベクトル時計

Specification and Verification of Memory Consistency Models*

Shiro Takata

Abstract

In this paper, we formally specify and verify some memory consistency models and their implementations that define the behavior of multiple memory accesses on DSM (Distributed Shared Memory) systems, using a formal method proposed by Taguchi and Araki.

Memory consistency models are defined as sets of rules that specify which of the written values is read and returned to the process. For efficient use of DSM systems, various memory consistency models have been proposed. These models can be classified into two groups. One group includes systems which constrain memory accesses by specifying various program orders of write and read operations. The other includes systems which constrain memory accesses by programmers specifying how and when synchronization of write and read operations should be made. In this paper, we deal with "Causal Memory Consistency Model" as a representative of the former group and "Release Consistency Model" as a representative of the latter group.

The formal method proposed by Taguchi and Araki includes the combination of the Z notation and value-passing CCS (Calculus of Communicating Systems), the state-based CCS semantics (S-CCS) that describes the evolution of processes and the transition of states simultaneously, and a set of transition rules. We

*Doctor's Thesis, Department of Information Systems, Graduate School of Information Science, Nara Institute of Science and Technology, NAIST-IS-DT9561021, February 8, 1999.

propose that a single transition relation described in terms of S-CCS should be extended to a trace of multiple transition relations. An execution history on shared-memory systems is expressed in a trace in terms of S-CCS. The trace is a sequence of actions resulted from the application of the transition rules, and it is used for verifying DSM systems.

We define the memory consistency models in terms of S-CCS. Then, we separately describe the functional aspect and the concurrent aspect of the shared-memory systems, i.e. the former in the Z notation and the latter using value-passing CCS. We use weak vector clocks based on the causally-precedes relation and semaphore based on the synchronization of memory accesses. We also verify that the specified shared-memory systems meet the requirements of the defined memory consistency models.

We conclude that this formal method is feasible enough for the formal specification and verification of both types of memory consistency models.

Keywords:

memory consistency model, distributed shared memory, formal method, Z, CCS, vector clock

目 次

1. はじめに **1**

1.1 本研究の背景と目的 1

1.2 本論文の概要 3

2. メモリ・コンシステンシ・モデル **5**

3. 形式手法 **7**

3.1 Z 7

3.2 CCS 9

3.3 Z と value-passing CCS の統合 10

3.4 状態遷移に基づく CCS 意味論（S-CCS） 10

3.4.1 ラベル付き遷移システム 10

3.4.2 状態遷移に基づく CCS 意味論（S-CCS） 11

3.4.3 遷移規則 12

4. メモリ・コンシステンシ・モデルの記述 **15**

4.1 分散共有メモリ並列計算機モデル 15

4.2 コーザル・メモリ・コンシステンシ・モデルの記述 19

4.3 リリース・コンシステンシ・モデルの記述 23

5. メモリ・コンシステンシ・モデルの実現 **26**

5.1 コーザル・メモリ・コンシステンシ・モデルの実現 26

5.1.1 弱ベクトル時計 26

5.1.2 コーザル・メモリの機能の記述 29

5.1.3 コーザル・メモリの並行性の記述 35

5.1.4 コーザル・メモリの展開例 37

5.2 リリース・コンシステンシ・モデルの実現 39

5.2.1 リリース・メモリの実現方式 39

5.2.2 リリース・メモリの機能の記述 40

5.2.3 リリース・メモリの並行性の記述 49
5.2.4 リリース・メモリの展開例 51

6. メモリ・コンシステンシ・モデルの実現の検証 **53**
6.1 コーザル・メモリの検証 53
6.2 リリース・メモリの検証 61

7. 関連研究 **69**
7.1 Z + CCS 69
7.2 (Object-)Z + CSP 70
7.2.1 (Object-)Z + CSP の検証方法 70
7.2.2 単一イベント vs 多重イベント 71
7.2.3 プロセスのブロック方式 71
7.3 まとめ 71

8. 結論 **73**
8.1 本研究で得られた成果 73
8.2 今後の課題 74

謝辞 **76**

参考文献 **78**

付録 **83**

A. Zの構文と演算子の概要 **83**

B. CCS の構文と演算子の概要 **87**

C. 著者研究業績 **88**

図目次

1.1 分散共有メモリの概念 1
2.1 メモリ・コンシステンシ・モデルの関係 6
5.1 ベクトル時計 28

1. はじめに

1.1 本研究の背景と目的

分散共有メモリ（Distributed Shared Memory：DSM）は，物理メモリを共有しない並列計算機/分散システムアーキテクチャおよびシステムソフトウェアの分野において，プロセス間でデータ共有するための抽象概念である．プロセスは，自アドレス空間内の普通のメモリに対するのと同じように，ロード命令やストア命令を発行することによって DSM にアクセスすることができる．これは，複数のプロセスが1つの共有メモリをアクセスしているように見えるが，実際には，物理メモリは図 1.1 のように分散している．

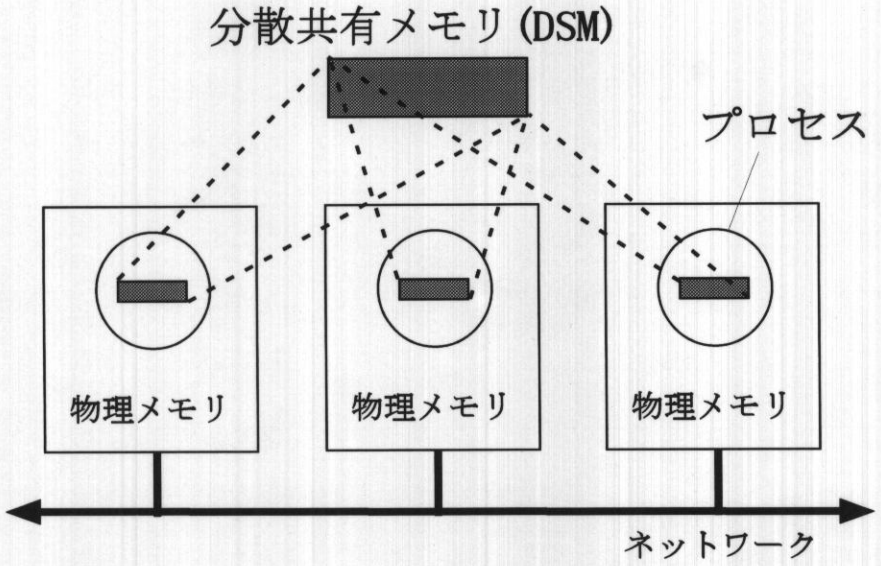

図 1.1 分散共有メモリの概念

そこで，物理的な共有メモリが無いため，DSM は実行時に並列計算機間で通信を行い更新情報を交換する必要がある．各計算機は，アクセス速度向上のためにローカルメモリを持ち，DSM システムはこれら複製データの管理をする．

DSM は 1980 年代から研究の活発な分野であり，Appolo Domain のファイルシステムに採用され成功を収めている．そこでは，個々のワークステーション上

のプロセスが，各アドレス空間のファイルを同時にマップすることにより共有している [1].

DSM は共有メモリ型マルチプロセッサの開発と並んでその重要性が増している．共有メモリ型マルチプロセッサにおける並列計算に適したアルゴリズムの研究やプロセッサ数を最大にするための研究が多く行われたが，メモリモジュールを一個所に集中させた集中共有メモリ型マルチプロセッサシステムでは，プロセッサの上限は 30 程度であり，それ以上のプロセッサを追加すると急速に性能が落ちる．一方，分散メモリ型マルチプロセッサやネットワーク分散システムでは，はるかに多くのプロセッサまたはコンピュータが接続されるようになってきており，分散共有メモリが期待されている．

DSM の実現では，既に述べたように，性能を高めるためにローカルメモリを持ち，いくつかの共有変数のコピーを保持している．共有変数は効率向上のためにローカルな値を使って読み出されるが，更新された値は他の全てのプロセスに同報通信して伝播される．そこで，並列プログラムを書くに当って，このような通信を介したメモリの更新が個々のプロセスから同じ条件で観測されているかどうかという一貫性（consistency）が問題となる．

メモリ・コンシステンシ・モデルは，そのような個々のプロセスのメモリ更新の観測に一貫性を持たすように，多重メモリアクセスの振る舞いを定義したものである．つまり，メモリ・コンシステンシ・モデルは，プロセスが DSM システムからデータを読み出すとき，多くの書き込まれた値の候補の中からどれを返すかを決定するモデルである．分散共有メモリシステムの性能はこれらモデルに強く依存するため，多くのモデルが提案されている [2, 3, 4].

一般に，メモリ・コンシステンシ・モデルの条件が弱ければ弱いほど，その DSM システムにおけるメモリ操作の振る舞いはより複雑になる．

また，メモリ・コンシステンシ・モデルを統一的に形式化する研究については，Adve [5], Ahamad [6], Kohli [7] などがある．これらは，ある視点を設定して，代数を用いて異なるモデル間の比較を目的として形式化がされている．但し，そこで定義されているプログラム順序など関係の定義は自然言語で定義されており，非形式的な記述に留まっている．また，実現に関して統一的な手法で記述・検証

まで触れている研究は見あたらない.

そこで，我々は，メモリ・コンシステンシ・モデルとそれらの実現の形式的な仕様記述と検証に関する研究を行ない，従来のモデルの正確な理解や比較を行うと共に，新しいモデルを提案することを目的とする.

また，これらモデルは，ストア命令やロード命令などに諸々のプログラム順序を定義してメモリアクセスを制約するものと，いつどのようにこれら命令の同期を取るべきかというプログラマの指定によりメモリアクセスを制約するものに大別される．そこで，本論文では，前者の代表としてコーザル・メモリ・コンシステンシ・モデルを，後者の代表としてリリース・コンシステンシ・モデルを取り上げる.

1.2 本論文の概要

本論文は，全 8 章から構成される.

第 1 章，すなわち本章では，分散共有メモリならびにメモリ・コンシステンシ・モデルを概説し，本研究の目的を述べた.

第 2 章では，主なメモリ・コンシステンシ・モデルについて概説すると共に，本論文で採用するコーザル・メモリ・コンシステンシ・モデルとリリース・コンシステンシ・モデルの位置づけを示す.

第 3 章では，本論文で用いる形式的仕様記述言語 Z とプロセス代数 CCS について概説すると共に，田口と荒木が提案した Z の表記法 [8] と value-passing CCS [9]（Calculus of Communicating Systems）を統合した形式手法 [10] について概説する．この形式手法では，Z の表記法と value-passing CCS (Calculus of Communicating Systems) の統合，プロセスの展開と状態遷移を同時に記述できる状態遷移に基づく CCS 意味論（S-CCS)，および，遷移規則などが提案されている.

第 4 章では，まず，本論文で採用する共有メモリ並列計算機モデルを定義して，その計算機モデルにおけるプログラム順序，実行履歴，逐次実行列などを S-CCS を用いて記述する．次に，コーザル・メモリ・コンシステンシ・モデルの定義を述べ，S-CCS を用いて形式的に定義する．同様に，リリース・コンシステンシ・

モデルの定義を述べ，S-CCS を用いて形式的に定義する．

第 5 章では，コーザル・メモリ・コンシステンシ・モデルとリリース・コンシステンシ・モデルの実現について，それぞれ形式的な仕様記述を行い，S-CCS の遷移規則を適用した展開例をそれぞれ示す．これらモデルの実現の仕様記述においては，状態と操作で表現される機能は Z を用い，通信による同期などの並行性の仕様記述は value-passing CCS を用いるという，システム分析に関する観点の違いを，二つの表記法を用いて分離して記述する．また，因果先行関係の比較には弱ベクトル時計を，メモリアクセスの同期についてはセマフォを用いる．

第 6 章では，コーザル・メモリ・コンシステンシ・モデルとリリース・コンシステンシ・モデルの検証をそれぞれ行う．検証に当っては，証明の対象となっている命令間のトレースを，遷移規則を適用して S-CCS を用いて記述する．次に，このプロセス展開と同時に遷移したローカルメモリ，各種キュー，弱ベクトル時計などの状態遷移を Z の操作スキーマの記述から求めて検証を行う．つまり，検証においても，プロセス展開と状態遷移に分離して行う．

第 7 章では，並列分散システムの形式手法について関連研究を述べる．

最後に，第 8 章では，本研究で得られた成果をまとめ，今後の課題について述べる．

2. メモリ・コンシステンシ・モデル

メモリ・コンシステンシ・モデルは，共有データの整合性に関して，プロセス間のプログラム順序に制約を付与して保つグループと，プログラマが指定した同期処理によって保つグループに大別される [2].

前者のグループには，ストリクト（Strict），シーケンシャル（Sequential），コーザル（Causal），プロセッサ（Processor），*PRAM* といったコンシステンシ・モデルが，後者のグループには，ウィーク（Weak），先行リリース（Eager Release），リリース（Release），遅延リリース（Lazy Release），エントリ（Entry）といったコンシステンシ・モデルがある．以下，前者と後者のグループ別に各モデルの概要について述べる．

まず，前者のグループの各モデルの概要を述べる．ストリクト・コンシステンシ・モデルでは，全てのプロセスで絶対時間をもち，いかなるロード命令も最も最新のストア命令が格納した値が返される．シーケンシャル・コンシステンシ・モデルでは，個々のプロセスが全てのプロセスのロード命令やストア命令を同じ順序で観測するという強い制約がある [11]．コーザル・メモリ・コンシステンシ・モデルでは，あるプロセスがストア命令を実行した後，その値を別のプロセスが使ってストア命令を発行するような因果関係があるような場合は，これらのストア命令はその他の全てのプロセスでも同じ順序で観測されねばならない [12]．プロセッサ・コンシステンシ・モデルでは，同じプロセスからの複数のストア命令と，異なるプロセスからの同一アドレスに対する複数のストア命令は，全てのプロセスによって同じ順序で観測されねばならないという Goodman の定義 [13] と，この定義とは微妙に異なる Gharachorloo の定義 [14] がある．*PRAM* コンシステンシ・モデルでは，あるプロセスから発行される全てのストア命令はその他の全てのプロセスで発行された順序で観測されねばならない [15]．図 2.1 の左側の図は，この前者のグループの各コンシステンシ・モデルのプログラム順序制約の強さの関係を表わしたものであり，円の中ほど強い制約を持つ．

次に，後者のグループの各モデルの概要を述べる．ウィーク・コンシステンシ・モデルでは，ロード命令やストア命令など通常の命令に対する制約は持たなく，プログラマの指定した同期命令に従って，共有データの同期を取る．但し，同期

命令は，シーケンシャル・コンシステンシ・モデルに従う [16]．リリース・コンシステンシ・モデルでは，同期変数をもつ同期命令に従って共有変数の同期を取る．但し，これら同期命令は，プロセッサ・コンシステンシ・モデルに従う [14]．先行リリース [17] と遅延リリース・モデル [18] はリリース・コンシステンシ・モデルを実現する上での改良版であり，どの時点で，ストア命令を共有メモリに反映させるかの違いである．エントリ・コンシステンシ・モデルでは，共有変数に所有者の概念を付け，クリティカル・セクションに属す共有データのみの一貫性を取る [19]．図 2.1 の右側の図は，この後者のグループの各コンシステンシ・モデルの同期命令に関する制約の強さの関係を表わしたものであり，円の中ほど強い制約を持つ．

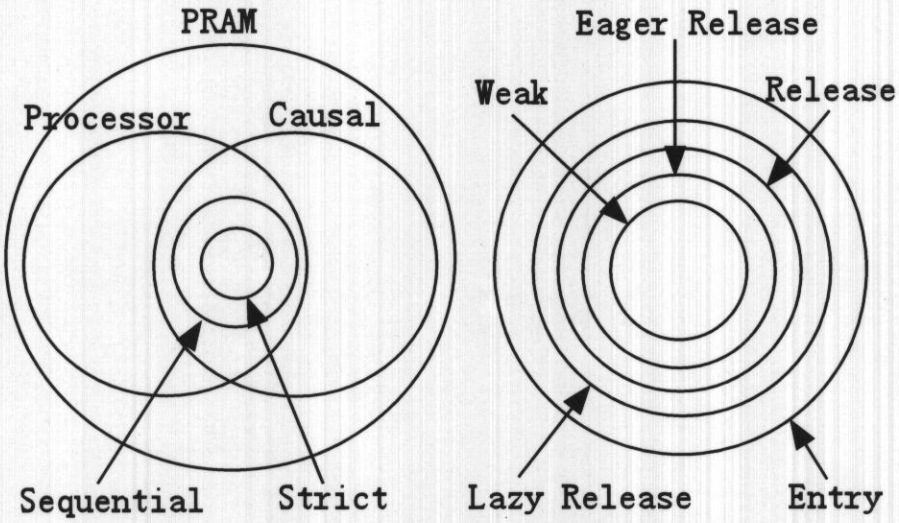

図 2.1 メモリ・コンシステンシ・モデルの関係

以下，本論文では，前者の代表としてコーザル・メモリ・コンシステンシ・モデルを，後者の代表としてリリース・コンシステンシ・モデルを取り上げる．

3. 形式手法

本章では，形式的仕様記述言語 Z とプロセス代数 CCS について概説すると共に，田口と荒木が提案した Z の表記法 [8] と value-passing CCS [9]（Calculus of Communicating Systems）を統合した形式手法 [10] について概説する．この形式手法では，Z の表記法と value-passing CCS (Calculus of Communicating Systems) の統合，プロセスの展開と状態遷移を同時に記述できる状態遷移に基づく CCS 意味論，および，遷移規則などが提案されている．

3.1 Z

本節では，Z の概要について述べる．Z [8, 20, 21, 22, 23] は，1970 年代後半から英国のオックスフォード大学の PRG（Programming Research Group）を中心に開発された集合論と 1 階述語論理に基づいた形式的仕様記述言語である．対象システムをどうやって実現するかには触れないで何をしたいのかを表現することで，そのシステムを抽象化することを目的としている．

この観点から Z は，集合論，述語論理，関係，および，関数の原理に基礎を置く数学的表現を選択して状態や操作を正確に記述し，厳密な議論検討をするのに適している．また，数学の証明技法を応用して，仕様記述の検証を行うこともできる．

一般的に，Z ではシステムを

- システム内部状態，と
- 操作

を定義することにより表現する．

システムの内部状態は，集合論，述語論理，関係，および，関数を用いて記述される．操作は，入力，出力，および，その操作によるシステム内部状態の前後関係を用いて記述される．Z の最も特徴的なものとして，スキーマによる仕様記述の図式表現が挙げられる．システムの内部状態と操作は，それぞれ，状態スキーマおよび操作スキーマを用いて記述される．

以下では，Z の仕様記述の構成について，簡単な例を用いて説明する．

ある団体の会員の名前を保持するシステムについて考える [24]．

Z では，基本的なデータの型として，システムが定義している基本型（basic type) と，ユーザが定義する型（given set）がある．まず，人の名前を表す型について given set を用いて定義する．given set では，その内部構造については言及しない．

$$[PERSON]$$

これで，*PERSON* は人の名前を表す集合として定義された．次に，会員の名前を保持するシステムの内部状態を，以下の状態スキーマを用いて定義する．

Member

$$member : \mathbb{P}\, PERSON$$

$$\#member < 100$$

この状態スキーマの名前は *Member* である．スキーマの表記は宣言部（上半分）と述語部（下半分）に分かれていて，それぞれ，宣言部ではスキーマで使用する変数の宣言を，述語部ではその変数の制約を記述する．宣言部の変数は型を持つ．$\mathbb{P}$ は巾集合を表し，*member* の型は人の名前 PERSON の巾集合となる．また，述語部では会員の数が 100 人未満であることを表現している．# は Z で定義されている演算子で，集合の要素の数を表す．

最後に，会員を登録する操作の仕様を定義する．操作スキーマの構成は状態スキーマと同じである．操作スキーマでは，宣言部で入力，出力，および，実行前後のシステム内部状態の変数を宣言し，それらの関係を述語部で定義することによりその振舞いを定義する．操作スキーマを集合の概念でとらえると，入力とシステム前状態の組と出力とシステム後状態の組との間の関係を定義している．

会員の登録を表す操作スキーマを，以下に示す．

$$
\begin{array}{|l}
\hline
AddPerson \\
member, member' : \mathbb{P}\, PERSON \\
p? : PERSON \\
\hline
member' = member \cup \{p?\} \\
\hline
\end{array}
$$

Z では，変数名の最後に ′ がついた変数は操作の実行後の変数を表し，′ がついていない変数は実行前の変数を表す．また，変数名の最後に ? がついた変数は入力であることを表し，! がついた変数は出力であることを表す．上記の例では，宣言部において，システムの前状態 $member$，後状態 $member'$，および，入力として登録する会員の名前 $p?$ を宣言している．この例では出力はない．述語部ではこれらの変数の間の関係を定義している．

3.2 CCS

CCS は，Milner によって提案されたプロセス代数である [9, 25]．プロセスをラベル付き遷移システムとして操作的意味論を与え，並行システムの数学的構造をモデル化するのに適している．CCS では代数的ステップは，全てのプロセスの構成要素を 1 つの代数理論における演算子として取り扱っている．特に，プロセス間の最も基本的な相互作用を，あるプロセスによるデータの送信は別のプロセスによるデータの受信と同期した内部隠蔽操作（τ 操作）としている．

並行システムを記述する際は，まず，CCS を使った式を定義する．この定義を行った後，その操作的意味論をラベル付き遷移システムとして示し，それらのトレースを導出木で表わすことができる．導出木は与えられたプロセスが行う一連の遷移あるいは履歴を表わしている．また，ソートの概念が導入されている．ソートは，ラベルの集合であり，プロセスの導出木において出現するラベルを L が含むとき，そのプロセスはソート L という属性を持つと言い，フローグラフを使って表現される．フローグラフは並行に動作し互いに通信を行う部分プロセスの構造を図式化するのに有効である．

また，本論文で使用する value-passing CCS は，プロセス間で値渡しを許す

CCSの変種である.

3.3 Z と value-passing CCS の統合

Zは，集合論と1階述語論理に基づく形式的仕様記述言語であり，様々な操作を定義する豊かなデータ構造と機能を備えているため状態や操作をモデル化するには適しているが，並行性を記述する十分な機能は持っていない．一方，プロセス代数である CCS [9] は，状態や操作を明示的にモデル化する機能は持っていないが，並行システムの数学的構造をモデル化するには適している．そこで，田口と荒木 [10] は，Z の表記法とプロセス間で値渡しを許す CCS の変種である value-passing CCS を統合して，プロセスの展開と状態遷移を同時に記述できる「状態遷移に基づく CCS 意味論（State-Based CCS Semantics：S-CCS）」と遷移規則を持つ形式手法を提案している．また，情報システムの設計において，状態と操作で表現される機能は Z を用い，通信による同期などの並行性の仕様記述は value-passing CCS を用いるという，システム分析に関する観点の違いを二つの表記法を用いて分離して行なうことを提案している．以下，状態遷移に基づく CCS 意味論を概説する.

3.4 状態遷移に基づく CCS 意味論（S-CCS）

本節では，本論文において用いられる形式手法の操作的意味論「状態遷移に基づく CCS 意味論（State-Based CCS Semantics：S-CCS）」とその遷移規則を説明する [10].

3.4.1 ラベル付き遷移システム

Milner は CCS の操作的意味論として，以下のラベル付き遷移システムを与えた [9].

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

このシステムは，CCSのエージェント表現の集合 $\mathcal{E}$，操作の集合 Act，および，全ての操作 $\alpha \in Act$ のための遷移関係 $\stackrel{\alpha}{\rightarrow} \subseteq \mathcal{E} \times \mathcal{E}$ から構成されている．例えば，操作 α によるプロセス E から別のプロセス E' への展開は，以下の遷移関係で示される．

$$E \stackrel{\alpha}{\rightarrow} E'$$

田口と荒木は，Zの操作スキーマを古い状態から新しい状態へのラベル付き遷移システムとして，以下の操作的意味論を提案した．

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

このシステムは，Zの状態集合 St，操作スキーマの集合 Op，および，全ての操作スキーマ $\alpha \in Op$ のための遷移関係 $\stackrel{\alpha}{\rightarrow} \subseteq St \times St$ から構成されている．例えば，操作スキーマ α による状態 s から別の状態 s' への展開は，以下の遷移関係で示される．

$$s \stackrel{\alpha}{\rightarrow} s'$$

3.4.2 状態遷移に基づくCCS意味論（S-CCS）

次に，田口と荒木は，Zの表記法とvalue-passing CCSを組み合わせた操作的意味論を，以下のラベル付き遷移システムとして与えた．

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

但し，CCSの操作とZの操作スキーマを区別するため $Act \cap Op = \emptyset$ とする．例えば，Zの操作スキーマ α による状態 s を持つプロセス $\alpha.E$ から状態 s' を持つ別のプロセス E への展開は，以下の遷移関係で示される．

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

但し，Θ を操作スキーマ α の1階述語表現とするとき，$s, s' \models [\![\Theta]\!]$ を満足する．

3.4.3 遷移規則

Z の操作スキーマと value-passing CCS の操作によるプロセスの展開と状態遷移は，以下の遷移規則に従う．

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)

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

S-CCS では，Z の入出力変数として定義された変数を value-passing CCS の入出力ポートの変数として使用することで，環境と Z の操作スキーマ間の入出力を行なう．このため，value-passing CCS の表現では，この変数の項書き換えは認めていない．また，Z の表記法では，? を持つ変数 $x?$ は入力変数，! を持つ変数 $x!$ は出力変数である．そこで，出力ポート $\overline{\alpha}$ を通して環境に出力変数 $x!$ の値を送り出す $\overline{\alpha}(x!)$，入力ポート α を通して環境から Z で定義された入力変数 $x?$ の値を受けとる $\alpha(x?)$ は，それぞれ，以下の遷移規則に従う．

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)

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

以下， prefix operator が prefix operator(2) または (3) ならば，s' は s である．

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)

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

出力値（例えば，c）が出力ポート $\overline{\alpha}$ から入力ポート α に通信される場合，以下の規則が適用される．但し，τ は，CCS の内部隠蔽操作（internal action）であり，以下 τ 操作と呼ぶ．

Concurrent Composition (2)

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

操作 α と $\overline{\alpha}$ が値を伴わない通信（通常，同期を取る通信）の場合は，以下の規則が適用される．

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 は一つの操作の遷移関係 $\xrightarrow{\alpha}$ を有限長の操作 **トレース** $\alpha_1 \ldots \alpha_n$ に対して，以下のような自然な拡張を提案した [26].

ω は，空のトレース ε を持つ列とする．表記法 $E \xrightarrow{\omega} F$ は「E はトレース ω を逐次に展開して F になる」という関係である．

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

我々は，遷移規則を有限長のトレースへ自然に拡張した [27, 28]．例えば，トレース ω が操作スキーマ $\alpha_1 \ldots \alpha_n$ を含む場合，その展開の遷移関係を，以下に示す．

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

但し，n を ω の中の操作スキーマの個数，Θ_i を α_i の 1 階述語による表現とするとき，$\forall i : 1 \ldots n \bullet s_i, s'_i \models [\![\Theta_i]\!] \wedge \forall i : 1 \ldots n-1 \bullet s_{i+1} = s'_i$ を満足する．

Trace

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

また，我々は value-passing CCS の *Conditional* 構文に対する遷移規則を，以下のように追加した [29]．

Conditional

$$\frac{\langle E, s \rangle \xrightarrow{\alpha} \langle E', s' \rangle}{\langle \mathit{if}\ b\ \mathit{then}\ E, s \rangle \xrightarrow{\alpha} \langle E', s' \rangle}\ (b\ \mathit{is\ true})$$

4. メモリ・コンシステンシ・モデルの記述

本章では，まず，本論文で採用する分散共有メモリ並列計算機モデルを定義して，その計算機モデルにおけるプログラム順序，実行履歴，逐次実行列などを，S-CCS を用いて記述する．次に，コーザル・メモリ・コンシステンシ・モデルの定義を述べ，S-CCS を用いて形式的に定義する．同様に，リリース・コンシステンシ・モデルの定義を述べ，S-CCS を用いて形式的に定義する．

4.1 分散共有メモリ並列計算機モデル

まず，Herlihy と Wing [30] および Misra [31] が定義した計算機モデルを基に，本論文で採用する分散共有メモリ並列計算機モデルを，以下のように定義する．但し，Z の表記法に従い，? を持つ変数は環境からの入力変数，! を持つ変数は環境への出力変数である．

- プロセス $\{P_1, \ldots, P_n\}$ を含む有限集合 $\mathcal{P}$ から構成され，有限なアドレス空間からなる共有メモリを通して，ロード命令，ストア命令，および，同期命令の列によって相互作用する．
- プロセス P_i が発行するロード命令 $r_i(x?, v!)$ は，不可分な命令であり，値 $v!$ がアドレス $x?$ に格納されていることを P_i に通知する．
- プロセス P_i が発行するストア命令 $w_i(x?, v?)$ は，不可分な命令であり，値 $v?$ をアドレス $x?$ に格納する．
- プロセス P_i が発行する同期命令は，同期変数を持たない $sync_i$ と，同期変数 $z?$ をもつ獲得命令 $acq_i(z?)$ および解放命令 $rel_i(z?)$ からなる．
- プロセス P_i が発行する観測（perceive）命令 $per_i(o_k)$ は，プロセス P_k が発行した命令 o_k を共有メモリを通して観測して，プロセス P_i のローカルメモリに適用する．但し，P_i が自ら発行した命令 o_i は，その命令 o_i そのものとする．

例えば，$per_i(r_i(x?,v!))$, $per_i(w_j(x?,v?))$ は，それぞれ，P_i が発行したロード命令 $r_i(x?,v!)$ そのもの，P_j が発行した $w_j(x?,v?)$ を P_i が観測して P_i のローカルメモリに適用する $Apply_i$ 命令である（5.1.2 項，5.2.2 項を参照）．以下，上記で定義した命令の実行履歴と順序関係を定義する．

プロセス P_i が逐次発行したロード命令，ストア命令，および，同期命令の実行履歴を L_i とする．例えば，$L_1 = \{w_1(x,1), r_1(y,2)\}$ のようなリストで表す．また，全てのプロセスの実行履歴の集合を $H = \langle L_1, L_2, \ldots, L_n \rangle$ とする．これら実行履歴に含まれる命令の実行順序関係は，以下の通りである．

1. $o_1 \underset{i}{\rightarrow} o_2$：実行履歴 L_i に含まれる命令 o_1,o_2 について，o_1 は o_2 より先行（先に実行）している．
2. $o_1 \rightarrow o_2$：実行履歴 H に含まれる命令 o_1,o_2 について，o_1 は o_2 より先行している．

次に，プロセス P_i が観測した命令列を逐次実行列 S_i と呼ぶ．但し，観測命令の定義より，プロセス P_i が自ら発行した命令 o_i は，その命令 o_i そのものである．S_i はプロセス P_i と他のプロセスとの共有メモリを通した相互作用を表わしている．例えば，$S_1 = \{w_1(x,1), per_1(w_2(y,2)), r_1(y,2)\}$ では，プロセス P_1 が発行した $w_1(x,1), r_1(y,2)$ とプロセス P_2 が発行した $w_2(y,2)$ とを共有メモリを通して観測した結果，$w_1(x,1)$, $w_2(y,2)$, $r_1(y,2)$ のプログラム順序でプロセス P_2 と相互作用したことが分かる．

また，全てのプロセスの逐次実行列 S_i の和集合 ΣS_i の命令を実行順序で列にしたものを逐次実行列 S と呼び，全てのプロセス間の共有メモリを通した相互作用を表わす．つまり，S はこのシステムの逐次実行列である．例えば $S = \{w_1(x,1), w_2(y,2), per_1(w_2(y,2)), r_1(y,2)\}$ のような列で，どのアドレスにどのプロセスが書き込みを行いそれをどのプロセスが観測して読み込んだかが分かる．

A を 実行履歴 H の全ての命令の集合，A^H_{i+w} を P_i が発行した全てのロード命令，ストア命令，同期命令，および，その他の全てのプロセスが発行したストア命令の和集合とする．

また，$S_i \mid z$，$S \mid z$ を，それぞれ，逐次実行列 S_i，S の中から変数 z を含む命令のみを取り出した逐次実行列とする．

逐次実行列 S_i に含まれる命令の実行順序関係は，以下の通りである．

3. $o_1 \overrightarrow{S_i} o_2$：逐次実行列 S_i に含まれる命令 o_1,o_2 に関して，o_1 は o_2 より先行している．
4. A^H_{i+w} の逐次実行列 S_i が有効：S_i は命令集合 A^H_{i+w} の全ての命令を含み，かつ，S_i 内の全てのロード命令はそのアドレスに最後にストアされた値を読み込む．但し，先行するストア命令がない場合は，初期値 $\perp$ が読み込まれる．
5. 同期変数 z に関する逐次実行列 $S_i \mid z$ が有効：$S_i \mid z$ 内の全ての獲得命令には，$S \mid z$ 上でそれぞれ異なる解放命令が先行している．但し，先行する解放命令がない場合は，$S_i \mid z$ で最初の獲得命令である．

以下，上記の順序関係 1 ～ 5 を S-CCS を用いて記述する．

$\langle \mathcal{E} \times St, Act \cup Op, \{\stackrel{\alpha}{\rightarrow} \mid \alpha \in Act \cup Op\}\rangle$ を分散共有メモリ並列計算機モデルのラベル付き遷移システム，ω を逐次実行列 S の部分列 A^* からなる有限長の命令トレース，E_0, E_1, E_2, E_i, E_j, E'_j, E_l, E'_l, E_m, E'_m, E_n, E_o, E_p を $\mathcal{E}$ の要素，s_0, s_1, s_2, s_i, s_j, s'_j, s_l, s'_l, s_m, s'_m, s_n, s_o, s_p を St の要素，$\mathcal{M}$ をこのシステムのアドレスの集合，Val をこのシステムの取り得る値の集合，Syn を同期変数の集合とするとき，上記 1 ～ 5 で定義された順序関係は，逐次実行列 S の有限長の命令トレース ω を使って，それぞれ，以下のように記述できる．

6. $o_1, o_2 \in L_i, \exists \omega \in A^* \bullet$
   $\langle o_1.E_1, s_1\rangle \xrightarrow{\omega} \langle o_2.E_2, s_2\rangle$
7. $o_1, o_2 \in H, \exists \omega \in A^* \bullet$
   $\langle o_1.E_1, s_1\rangle \xrightarrow{\omega} \langle o_2.E_2, s_2\rangle$
8. $o_1, o_2 \in S_i, \exists \omega \in A^* \bullet$
   $\langle o_1.E_1, s_1\rangle \xrightarrow{\omega} \langle o_2.E_2, s_2\rangle$
9. $(\forall o \in A^H_{i+w} \bullet \exists per_i(o) \in S_i \; of \; A^H_{i+w})$
   $\wedge$
   $x_l \in \mathcal{M}, v_l, v_m \in Val,$

$\forall\, r_i(x_l, v_l) \in S_i \text{ of } A^H_{i+w},$

$\exists\, j : 1 \ldots n, \exists\, \omega \in A^* \bullet$

$((\exists\, per_i(w_j(x_l, v_l)) \in S_i \text{ of } A^H_{i+w},$

$per_i(w_j(x_l, v_m)) \notin \omega \bullet$

$\langle per_i(w_j(x_l, v_l)).E_l, s_l\rangle$

$\xrightarrow{per_i(w_j(x_l,v_l))\,\omega\, r_i(x_l,v_l)} \langle E_m, s_m\rangle)$

$\vee\, (((\exists\, per_i(w_j(x_l, v_m)) \in S_i \text{ of } A^H_{i+w},$

$per_i(w_j(x_l, v_m)) \notin \omega \bullet$

$\langle start.E_0, s_0\rangle \xrightarrow{\omega\, r_i(x_l,v_l)} \langle E_m, s_m\rangle)$

$\vee\, per_i(w_j(x_l, v_m)) \notin S_i \text{ of } A^H_{i+w})$

$\Rightarrow v_l = \bot))$

10. $z \in Syn, \forall\, acq_i(z) \in S_i \text{ of } A^H_{i+w},$

$\exists\, j : 1 \ldots n, \exists\, \omega \in A^* \bullet$

$((\exists\, rel_j(z) \in S_j \text{ of } A^H_{j+w},$

$rel_j(z), acq_j(z) \notin \omega \bullet$

$\langle rel_j(z?).E_l, s_l\rangle \xrightarrow{rel_j(z)\,\omega\, acq_i(z)} \langle E_m, s_m\rangle)$

$\vee\, (rel_j(z) \notin \omega \bullet$

$\langle start.E_0, s_0\rangle \xrightarrow{\omega\, acq_i(z)} \langle E_m, s_m\rangle))$

4.2 コーザル・メモリ・コンシステンシ・モデルの記述

Huttoによって提案されたコーザル・メモリ・コンシステンシ・モデルは，シーケンシャル・コンシステンシ・モデル [11] の条件を少し緩めたものであり，以下の書き込み先導 (write-into order) および因果順序 (causality order) などのプログラム順序を用いて定義されている [12].

まず，書き込み先導 $\mapsto$ は次のように定義されている.

- もし $o_1 \mapsto o_2$ の関係があるなら $o_1 = w(x,v)$ かつ $o_2 = r(x,v)$ を満たすような x と v が存在する.
- 全ての操作 o_2 について少なくとも一つは $o_1 \mapsto o_2$ の関係を満たすような o_1 が存在する.
- もし，ある x に対する操作 $o_2 = r(x,v)$ に関して $o_1 \mapsto o_2$ を満足する o_1 が存在しなければ，その値は初期値として $v = \perp$ を持つ.

次に，因果順序 $o_1 \rightsquigarrow o_2$ は次のように定義されている.

- $o_1 \vec{i} o_2$; または,
- $o_1 \mapsto o_2$; または,
- $o_1 \rightsquigarrow o' \rightsquigarrow o_2$ を満たす o' が存在する.

但し，これらの関係はサイクルを持たないものとする.

これらのプログラム順序の定義を用いて，コーザル・メモリ・コンシステンシ・モデル（CMモデル）は次のように定義されている.

CM: 全てのプロセッサ P_i において $\rightsquigarrow$ を満足する A^H_{i+w} に関する有効な逐次実行列 S_i が存在する.

以下，上記の定義をS-CCSを用いて記述する準備として，A^H_{i+w} の有効な逐次実行列 S_i 上の命令 o_1，o_2 が因果順序 $o_1 \rightsquigarrow o_2$ を満足する3つの条件をS-CCSを用いてそれぞれ記述する.

まず，同じプロセスから発行された命令 o_1，o_2 について，o_1 が o_2 より先行する場合を述べる．プロセス P_i からこれら命令を観測する場合は，これら命令が同じプロセスから発行されたものか，異なるプロセスから発行されたものかに分けられる．これら命令が同じプロセス P_i から発行された $o_1 \underset{i}{\rightarrow} o_2$ の場合は，逐次実行列 S_i の定義より，プロセス P_i は，次のように常に o_1 が o_2 より先行していると観測する．

$$o_1, o_2 \in L_i, \exists \omega \in A^* \bullet$$
$$\langle E_1, s_1 \rangle \xrightarrow{o_1 \omega o_2} \langle E_2, s_2 \rangle \Rightarrow \langle E_1, s_1 \rangle \xrightarrow{per_i(o_1)\omega\, per_i(o_2)} \langle E_2, s_2 \rangle$$

上の条件は常に成立するため CM モデルに関する S-CCS を用いた記述においては省略できる．

次に，異なるプロセス P_j から発行された $o_1 \underset{j}{\rightarrow} o_2 (j \neq i)$ の場合は，プロセス P_i で観測するのは，プロセス P_j から発行されたストア命令 o_1，o_2 のみである．そこで，これらストア命令を，プロセス P_i で同じ順序で観測する条件は，下記のように記述できる．

$$j : 1 \ldots n, x_l, x_m \in \mathcal{M}, v_l, v_m \in Val \bullet$$
$$(\forall\, w_j(x_l, v_l), \forall\, w_j(x_m, v_m) \in A^H_{i+w}, \exists \omega_1, \omega_2 \in A^* \bullet$$
$$(\langle w_j(x?, v?).E_l, s_l \rangle \xrightarrow{w_j(x_l,v_l)\,\omega_1\, w_j(x_m,v_m)} \langle E_m, s_m \rangle \Rightarrow$$
$$\langle per_i(w_j(x_l, v_l)).E'_l, s'_l \rangle \xrightarrow{per_i(w_j(x_l,v_l))\,\omega_2\, per_i(w_j(x_m,v_m))} \langle E'_m, s'_m \rangle))$$

次に，書き込み先導関係 $\mapsto$ を満足する命令を，プロセス P_i で同じ順序で観測する条件は，以下のように記述できる．

$$
\begin{aligned}
&j:1\ldots n, x_l \in \mathcal{M}, v_l, v_m \in Val,\\
&\forall r_i(x_l, v_l) \in S_i \; of \; A^H_{i+w} \bullet\\
&((\exists w_j(x_l, v_l) \in A^H_{i+w}, \exists \omega_1, \omega_2 \in A^* \bullet\\
&\quad (\langle w_j(x?, v?).E_j, s_j\rangle \xrightarrow{w_j(x_l,v_l)\,\omega_1\, r_i(x_l,v_l)} \langle E_i, s_i\rangle \Rightarrow\\
&\quad\ \langle per_i(w_j(x_l, v_l)).E'_j, s'_j\rangle \xrightarrow{per_i(w_j(x_l,v_l))\,\omega_2\, r_i(x_l,v_l)} \langle E_i, s_i\rangle))\\
&\vee (((\exists per_i(w_j(x_l, v_m)) \in S_i \; of \; A^H_{i+w}, \exists \omega \in A^*,\\
&per_i(w_j(x_l, v_m)) \notin \omega \bullet\\
&\langle start.E_0, s_0\rangle \xrightarrow{\omega\; r_i(x_l,v_l)} \langle E_i, s_i\rangle)\\
&\vee per_i(w_j(x_l, v_m)) \notin S_i \; of \; A^H_{i+w})\\
&\quad \Rightarrow v_l = \bot))
\end{aligned}
$$

但し，上記の条件で初期値 $\bot$ を与える条件は，逐次実行列が有効であるという条件にも含まれるため，CM モデルに関する S-CCS を用いた記述においては省略できる．

次に，$o_1 \rightsquigarrow o' \rightsquigarrow o_2$ を満足する命令の場合は，上記のように $\underset{i}{\rightarrow}$ かつ $\mapsto$ を満たす ω_1, ω_2 を持つ逐次実行列を持てば，3.4.3 項で述べた **trace** 遷移規則を繰り返し適用することにより，以下のような遷移規則を得ることができ，常に S_i は，これら命令を同じ順序で観測する．

$$
\frac{\langle o_1.E_1, s_1\rangle \xrightarrow{\omega_1} \langle o'.E', s'_1\rangle \quad \langle o'.E', s'_1\rangle \xrightarrow{\omega_2} \langle o_2.E_2, s'_2\rangle}{\langle o_1.E_1, s_1\rangle \xrightarrow{\omega_1\omega_2} \langle o_2.E_2, s'_2\rangle}
$$

よって CM モデルに関する S-CCS を用いた記述においては，S-CCS を用いて記述したプログラム順序と書き込み先導関係の全ての条件を and 条件で記述することで，この条件は省略できる．

また，S-CCS にて使用している因果先行関係 $\rightarrow$ は半順序を持っておりベクトル時計を持ったトレース上の先行順序 $\longrightarrow$ 関係に閉路はない．

以下，S-CCS を用いてコーザル・メモリ・コンシステンシ・モデルを記述する．

$$
\begin{array}{l}
j : 1 \ldots n, x_l, x_m \in \mathcal{M}, v_l, v_m \in Val, \forall\, i : 1 \ldots n \bullet \\
(((\forall\, o \in A^H_{i+w} \bullet \exists\, per_i(o) \in S_i \; of \; A^H_{i+w}) \\
\wedge \\
(\forall\, r_i(x_l, v_l) \in S_i \; of \; A^H_{i+w}, \exists\, \omega \in A^* \bullet \\
\quad ((\exists\, per_i(w_j(x_l, v_l)) \in S_i \; of \; A^H_{i+w}, \\
\quad per_i(w_j(x_l, v_m)) \notin \omega \bullet \\
\qquad \langle per_i(w_j(x_l, v_l)).E_j, s_j \rangle \\
\qquad\quad \xrightarrow{per_i(w_j(x_l,v_l))\,\omega\, r_i(x_l,v_l)} \langle E_i, s_i \rangle) \\
\quad \vee (((\exists\, per_i(w_j(x_l, v_m)) \in S_i \; of \; A^H_{i+w}, \\
\quad per_i(w_j(x_l, v_m)) \notin \omega \bullet \\
\qquad \langle start.E_0, s_0 \rangle \xrightarrow{\omega\, r_i(x_l,v_l)} \langle E_i, s_i \rangle) \\
\quad \vee per_i(w_j(x_l, v_m)) \notin S_i \; of \; A^H_{i+w}) \\
\qquad \Rightarrow v_l = \bot)))) \\
\wedge \\
(\forall\, w_j(x_l, v_l), \forall\, w_j(x_m, v_m) \in A^H_{i+w}, \exists\, \omega_1, \omega_2 \in A^* \bullet \\
\quad (\langle w_j(x?, v?).E_l, s_l \rangle \xrightarrow{w_j(x_l,v_l)\,\omega_1\, w_j(x_m,v_m)} \langle E_m, s_m \rangle \Rightarrow \\
\qquad \langle per_i(w_j(x_l, v_l)).E'_l, s'_l \rangle \xrightarrow{per_i(w_j(x_l,v_l))\,\omega_2\, per_i(w_j(x_m,v_m))} \langle E'_m, s'_m \rangle)) \\
\wedge \\
(\forall\, r_i(x_l, v_l) \in S_i \; of \; A^H_{i+w}, \exists\, w_j(x_l, v_l) \in A^H_{i+w}, \\
\exists\, \omega_1, \omega_2 \in A^* \bullet \\
\quad (\langle w_j(x?, v?).E_j, s_j \rangle \xrightarrow{w_j(x_l,v_l)\,\omega_1\, r_i(x_l,v_l)} \langle E_i, s_i \rangle \Rightarrow \\
\qquad \langle per_i(w_j(x_l, v_l)).E'_j, s'_j \rangle \xrightarrow{per_i(w_j(x_l,v_l))\,\omega_2\, r_i(x_l,v_l)} \langle E_i, s_i \rangle)))
\end{array}
$$

4.3 リリース・コンシステンシ・モデルの記述

リリース・コンシステンシ・モデルの基本的な考え方は，プログラマが同期命令を使ってプログラムの同期を取ることで，DSM のオーバヘッドを軽減することである．プログラム中の同期命令には，獲得命令（ロック）と解放命令（アンロック）があり，これら同期命令によってストア命令の遅延が隠蔽される．このモデルは，以下のような制約条件で定義されている [14, 18, 2, 3].

- 他の全てのプロセスに関して通常のロード命令やストア命令の実行が許される前に，それ以前のすべての獲得命令が完了していなければならない．
- 他の全てのプロセスに関して解放命令の実行が許される前に，それ以前の全てのロード命令やストア命令が完了していなければならない．
- 獲得命令と解放命令はプロセッサ・コンシステンシ・モデル [14] に従う．

以下，形式的に定義する．但し，最後の条件の中で，獲得命令と解放命令が異なる同期変数を持つ場合，獲得命令が解放命令を飛び越すことがあるという条件については，本節の目的とする同期に関する仕様記述の本質的な問題から離れるため省略する．そこで，最後の条件は，獲得命令が解放命令と同じ同期変数をもつ場合はその同期変数に関する逐次実行列が有効であればよいという条件となり最初の条件に含めて記述する．

[リリース・コンシステンシ・モデルの形式的定義]

11. 各同期変数 z に関する有効な逐次実行列 $S_i \mid z$ が存在して，任意の獲得命令 $acq_i(z)$ と $acq_i(z) \underset{i}{\rightarrow} o_i$ を満足するプロセス P_i が発行する任意の命令 o_i は，全てのプロセス P_k において $acq_i(z) \xrightarrow{\omega} per_k(o_i)$ を満足する．
12. 各プロセス P_i の任意の解放命令 $rel_i(z)$ と $o_i \underset{i}{\rightarrow} rel_i(z)$ を満足する P_i が発行する任意の命令 o_i は，全てのプロセス P_j において $per_j(o_i) \xrightarrow{\omega} rel_i(z)$ を満足する．

但し，当モデルにおける観測命令は，異なるプロセスで発行された命令に関してはストア命令のみを対象とする．例えば，$per_1(w_2(x,1))$，$per_1(r_1(x,1))$ などは有効であるが，$per_2(r_1(x,1))$ は無効な観測命令であり，以下の式では，対象外とする．

以下，11，12 の条件，および，逐次実行列 S_i は有効であるというリリース・コンシステンシ・モデルの条件を，S-CCS を用いて記述する．

$x_l \in \mathcal{M}, v_l, v_m \in Val, \forall\, i : 1 \ldots n \bullet$

$(((\forall\, o \in A^H_{i+w} \bullet \exists\, per_i(o) \in S_i\ of\ A^H_{i+w})$

$\wedge$

$(\forall\, r_i(x_l, v_l) \in S_i\ of\ A^H_{i+w},$

$\exists\, j : 1 \ldots n, \exists\, \omega \in A^* \bullet$

$((\exists\, per_i(w_j(x_l, v_l)) \in S_i\ of\ A^H_{i+w},$

$per_i(w_j(x_l, v_m)) \notin \omega \bullet$

$\langle per_i(w_j(x_l, v_l)).E_l, s_l\rangle$

$\stackrel{per_i(w_j(x_l,v_l))\,\omega\, r_i(x_l,v_l)}{\longrightarrow} \langle E_m, s_m\rangle)$

$\vee\, (((\exists\, per_i(w_j(x_l, v_m)) \in S_i\ of\ A^H_{i+w},$

$per_i(w_j(x_l, v_m)) \notin \omega \bullet$

$\langle start.E_0, s_0\rangle \stackrel{\omega\, r_i(x_l,v_l)}{\longrightarrow} \langle E_m, s_m\rangle)$

$\vee\, per_i(w_j(x_l, v_m)) \notin S_i\ of\ A^H_{i+w})$

$\Rightarrow v_l = \bot))))$

$\wedge$

$(\forall\, z \in Syn \bullet$

$\forall\, acq_i(z) \in S_i\ of\ A^H_{i+w},$

$\exists\, j : 1 \ldots n, \exists\, \omega \in A^* \bullet$

$(((\exists\, rel_j(z) \in S_j\ of\ A^H_{j+w},$

$rel_j(z), acq_j(z) \notin \omega \bullet$

$\langle rel_j(z?).E_l, s_l\rangle \stackrel{rel_j(z)\,\omega\, acq_i(z)}{\longrightarrow} \langle E_m, s_m\rangle)$

$\vee\, (rel_j(z) \notin \omega \bullet$

$\langle start.E_0, s_0\rangle \stackrel{\omega\, acq_i(z)}{\longrightarrow} \langle E_m, s_m\rangle))$

$\wedge$

$(\forall\, o_i \in L_i, \forall\, k : 1 \ldots n, \exists\, \omega_1, \omega_2 \in A^* \bullet$

$(\langle acq_i(z?).E_l, s_l\rangle \stackrel{acq_i(z)\,\omega_1\, o_i}{\longrightarrow} \langle E_m, s_m\rangle \Rightarrow$

$\langle acq_i(z?).E_l, s_l\rangle \stackrel{acq_i(z)\,\omega_2\, per_k(o_i)}{\longrightarrow} \langle E_n, s_n\rangle)))))$

$$
\begin{aligned}
&\wedge \\
&(z \in Syn, \forall\, rel_i(z), \forall\, o_i \in L_i, \\
&\forall\, j : 1 \ldots n, \exists\, \omega_1, \omega_2 \in A^* \bullet \\
&\quad (\langle E_l, s_l \rangle \overset{o_i\, \omega_1\, rel_i(z)}{\longrightarrow} \langle E_n, s_n \rangle \Rightarrow \\
&\qquad \langle per_j(o_i).E_m, s_m \rangle \overset{per_j(o_i)\, \omega_2\, rel_i(z)}{\longrightarrow} \langle E_n, s_n \rangle))\,)
\end{aligned}
$$

5. メモリ・コンシステンシ・モデルの実現

本章では，コーザル・メモリ・コンシステンシ・モデルとリリース・コンシステンシ・モデルの実現について，それぞれ形式的な仕様記述を行い，S-CCS の遷移規則を適用した展開例をそれぞれ示す．

これらメモリシステムの実現において，状態と操作で表現される機能は Z を用い，通信による同期などの並行性の仕様記述は value-passing CCS を用いるという，システム分析に関する観点の違いを，二つの表記法を用いて分離して記述する．

また，因果先行関係の比較には弱ベクトル時計を，メモリアクセスの同期についてはセマフォを用いる．

5.1 コーザル・メモリ・コンシステンシ・モデルの実現

本節では，コーザル・メモリ・コンシステンシ・モデルの実現であるコーザル・メモリ $\mathcal{CM}$ を Ahamad 等が提案したコーザル・メモリ [6] を参考にして，Z の表記法と value-passing CCS を統合して形式的に仕様記述する．

まず，分散処理における因果先行関係の記述として使用する**弱ベクトル時計**について説明する．次に，コーザル・メモリの機能と並行性について，それぞれ分離して記述する．最後に，S-CCS の遷移規則を適用した展開例を挙げる．

5.1.1 弱ベクトル時計

ベクトル時計 [32] は，分散処理システムにおいて Lamport [11] が定義した事象間の因果先行関係 $\rightarrow$ の決定に使用される．

全ての事象の発行単位でベクトル・タイムスタンプを記録すると，任意のイベント間の因果先行関係は，それらベクトル・タイムスタンプを比較することで決定される．

ベクトル・タイムスタンプは，プロセス数が n のとき n 個の整数の列で表現される．

2つの事象 e_i, e_j とそれらのベクトル・タイムスタンプ $t(e_i)$, $t(e_j)$ が与えられた場合，以下の順序関係が定義されている．

$$
\begin{aligned}
t(e_i) \prec t(e_j) &\stackrel{\text{def}}{=} (\forall k : 1 \ldots n \bullet t(e_i)[k] \leq t(e_j)[k]) \\
&\quad \wedge (\exists l : 1 \ldots n \bullet t(e_i)[l] < t(e_j)[l]) \\
t(e_i) \preceq t(e_j) &\stackrel{\text{def}}{=} (t(e_i) \prec t(e_j)) \vee (t(e_i) = t(e_j)) \\
t(e_i) \preceq t(e_j) &\Leftrightarrow e_i \rightarrow e_j
\end{aligned}
$$

従来のベクトル時計では，プロセス P_i の各事象が実行される毎に，ローカルカウンタ $t[i]$ は増加する．対照的に，弱ベクトル時計 [33] は，P_i が，いくつかの状態変数などで表現されているシステム特性を変えるような事象が実行されたときのみ $t[i]$ を増加させる．

いずれの場合も，P_j はそのベクトル時計が変化したときは，そのベクトル・タイムスタンプ t_j の付いた P_j の更新情報を含むメッセージを，他のすべてのプロセスに同報通信する．そのようなメッセージを受信した各々のプロセス P_i は，そのプロセスの状態とそのベクトル・タイムスタンプ t_i を更新する．弱ベクトル時計の場合は，P_i は，ベクトル・タイムスタンプ t_i を以下のように更新する．

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

図 5.1 は，弱ベクトル時計を採用した 5.1.2 項，5.1.3 項で記述するコーザル・メモリシステムの逐次実行列の例である．プロセス P_j は，$w_j(x?, v?)$ 命令を実行してローカルメモリを更新すると，そのベクトル時計のカウンタ $t_j[j]$ を増加させ，$Send_j$ 命令により，ローカルメモリの更新情報とベクトル・タイムスタンプ t_j を含んだ更新メッセージをその他の全てのプロセスに同報通信する．他のプロセス P_i は，その更新メッセージを $Receive_i$ により受信して入力キューに一次保管する．そして，$Apply_i$ 命令により，自分のベクトル・タイムスタンプ t_i と入力キューの更新メッセージのベクトル・タイムスタンプ t_j を比較することで，コーザル・メモリ・コンシステンシの条件に合う更新メッセージを選択して，自らのローカルメモリとベクトル・タイムスタンプ t_i を更新する．このようにお

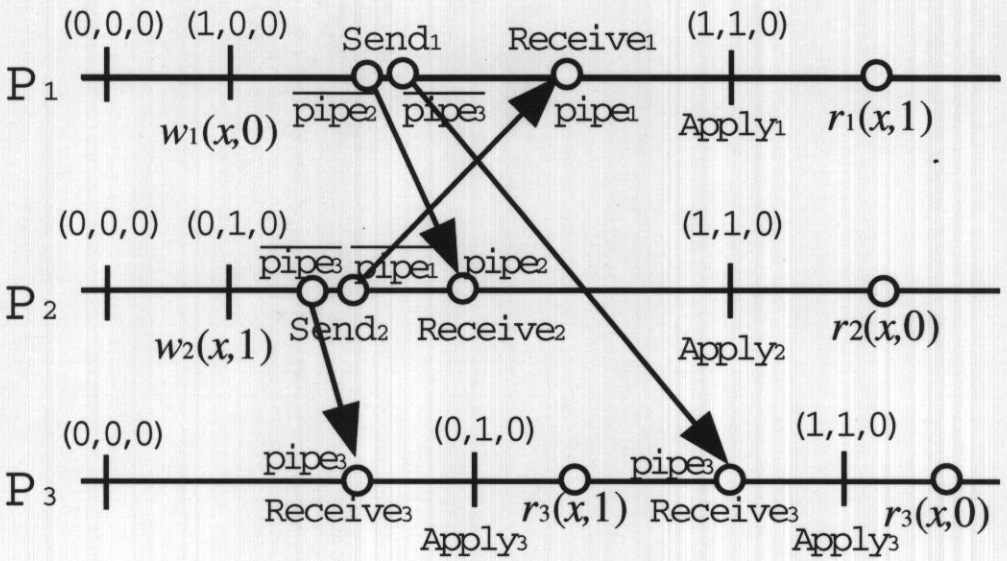

図 5.1 コーザル・メモリシステムの実行履歴

互いに更新メッセージを交換して，ベクトル・タイムスタンプを比較することで，プログラム実行順序に関係する共有メモリの一貫性を保っている．図 5.1 のベクトル・タイムスタンプとプログラム順序には以下の関係がある．

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

また，ベクトル・タイムスタンプ t_i は 0 ベクトルに初期化されるので，$t_i[j]$ は，P_j で発行されたストア命令の数を示す．

5.1.2 コーザル・メモリの機能の記述

本項では，各プロセス P_i の機能を，Z 記法を用いて状態スキーマと操作スキーマにて表現する．

各プロセス P_i は，以下の状態スキーマ s_i を持つ．各状態スキーマ s_i は，プロセス番号 pn_i，抽象的なコーザル・メモリ $\mathcal{M}$ を構成するローカルメモリ M_i，ベクトル・タイムスタンプ t_i，二つのメッセージキュー $OutQueue_i$ と $InQueue_i$，プロセス P_i が発行したロード命令とストア命令の実行履歴 L_i，および，プロセス P_i が発行したロード命令，ストア命令，観測命令，および，全プロセスが発行した全てのストア命令の集合 A^H_{i+w} に関する逐次実行列 S_i から構成される．

メッセージキュー $OutQueue_i$ は先入れ先出しキューであり，プロセス P_i で発行されたストア命令によるローカルメモリの更新情報の内，他のプロセスに伝達されていない更新情報が含まれている．メッセージキュー $InQueue_i$ は，他のプロセスが発行したストア命令に関して未だローカルメモリに反映されていない更新情報が含まれていて，ベクトル・タイムスタンプ順に並べられている．

まず，以下に，コーザル・メモリシステムの基本型宣言，略記定義，型，および，操作スキーマ s_i を定義する．

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

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

$$\begin{array}{|l} NumberOfProcesses : \mathbb{N}_1 \\ MaxOutQueue, MaxInQueue : \mathbb{N}_1 \\ MaxSerial, MaxLocalHis : \mathbb{N}_1 \\ priority\_queue : (\mathrm{seq}\ write\_tuple) \times write\_tuple \\ \qquad\qquad \nrightarrow \mathrm{seq}\ write\_tuple \end{array}$$

s_i

$pn_i : \mathbb{N}_1$

$M_i : \mathcal{M} \pfun (Val \cup \{\bot\})$

$t_i : \mathrm{seq}\, \mathbb{N}$

$OutQueue_i : \mathrm{seq}\ write\_tuple$

$InQueue_i : \mathrm{seq}\ write\_tuple$

$L_i : \mathrm{seq}\ A$

$S_i : \mathrm{seq}\ A$

$\#t_i = NumberOfProcesses$

$\#OutQueue_i \leq MaxOutQueue$

$\#InQueue_i \leq MaxInQueue$

$\#L_i \leq MaxLocalHis$

$\#S_i \leq MaxSerial$

プロセス P_i は，以下の操作スキーマ $InitP_i$, $Read_i$, $Write_i$, $Send_i$, $Receive_i$, および，$Apply_i$ などの操作スキーマを持つ．

各プロセス P_i は，後述する入力ポート id を通してプロセス番号 $pn_i?$ を受け取ると，以下の操作スキーマ $InitP_i$ を発行して状態スキーマの初期化を行う．

$InitP_i$

s_i'

$pn_i? : \mathbb{N}_1$

$pn_i' = pn_i?$

$M_i' = \lambda\, x : \mathcal{M} \bullet \bot$

$t_i' = \lambda\, n : 1\,..\,NumberOfProcesses \bullet 0$

$OutQueue_i' = \langle\rangle$

$InQueue_i' = \langle\rangle$

$L_i' = \langle\rangle$

$S_i' = \langle\rangle$

各プロセス P_i は，後述する入力ポート loc_i を通して環境からアドレス $x?$ を受け取ると，以下の操作スキーマ $Read_i$ を発行してローカルメモリ $M_i(x?)$ の値 $v!$ を受理する．また，実行履歴 L_i と逐次実行列 S_i にロード命令のラベル $r_i(x?, v!)$ を追加する．

$Read_i$

Δs_i

$x? : \mathcal{M}$

$v! : Value$

$v! = M_i x?$

$pn_i' = pn_i$

$M_i' = M_i$

$t_i' = t_i$

$OutQueue_i' = OutQueue_i$

$InQueue_i' = InQueue_i$

$L_i' = L_i \frown \langle r_i(x?, v!) \rangle$

$S_i' = S_i \frown \langle r_i(x?, v!) \rangle$

各プロセス P_i は，後述する入力ポート $heap_i$ を通して環境からアドレス $x?$ と値 $v?$ を受け取ると，以下の操作スキーマ $Write_i$ を発行してローカルなベクトル時計 $t[i]$ を 1 進め，ローカルメモリ $M_i(x?)$ の値を $v?$ に更新する．また，出力キュー $OutQueue_i$ に更新メッセージ (プロセス番号 i, 更新アドレス $x?$, 更新値 $v?$, ベクトル・タイムスタンプ t_i) を追加する．これら更新メッセージは，後述する操作スキーマ $Send_i$ により他の全てのプロセスに同報通信される．

同様に，実行履歴 L_i と逐次実行列 S_i にストア命令のラベル $w_i(x?, v?)$ を追加する．

$Write_i$

Δs_i

$x? : \mathcal{M}$

$v? : Value$

$pn_i' = pn_i$

$M_i' = M_i \oplus \{x? \mapsto v?\}$

$t_i'\, pn_i = t_i\, pn_i + 1$

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

$OutQueue_i' = OutQueue_i \frown \langle (pn_i, x?, v?, t_i') \rangle$

$InQueue_i' = InQueue_i$

$L_i' = L_i \frown \langle w_i(x?, v?) \rangle$

$S_i' = S_i \frown \langle w_i(x?, v?) \rangle$

各プロセス P_i は，以下の操作スキーマ $Send_i$ を適時発行して，出力キュー $OutQueue_i$ 内の更新情報を，後述する $\overline{pipe_j}$ を通して他の全てのプロセスに同報通信する．

$Send_i$

Δs_i

$message! : write\_tuple$

$OutQueue_i \neq \langle\rangle$

$pn_i' = pn_i$

$M_i' = M_i$

$t_i' = t_i$

$message! = head\ OutQueue_i$

$OutQueue_i' = tail\ OutQueue_i$

$InQueue_i' = InQueue_i$

$L_i' = L_i$

$S_i' = S_i$

各プロセス P_i は，後述する $pipe_i$ を通して通信されたメモリ更新メッセージを受け取ると，以下の操作スキーマ $Receive_i$ を発行して入力キュー $InQueue_i$ に追加する．但し，入力キューの要素はベクトル・タイムスタンプの昇順に並べる．関数 $priority\_queue$ は，受け取ったメッセージ中のベクトル・タイムスタンプと入力キューの要素中のベクトル・タイムスタンプを入力キューの先頭から順次比較する．もし受け取ったメッセージのベクトル・タイムスタンプの方が小さければ，比較した要素の直前にそのメッセージを追加する．但し，順次比較した末に入力キューの最後の要素のベクトル・タイムスタンプの方が小さい，または，大小関係が求められない場合は，受け取ったメッセージは入力キューの末尾に追加する．但し，本論文では，この関数の記述は省略する．

$Receive_i$

Δs_i

$message? : write\_tuple$

$pn_i' = pn_i$

$M_i' = M_i$

$t_i' = t_i$

$OutQueue_i' = OutQueue_i$

$InQueue_i' = priority\_queue(InQueue_i,$
$\quad message?)$

$L_i' = L_i$

$S_i' = S_i$

各プロセス P_i は，以下の操作スキーマ $Apply_i$ を適時発行して，$InQueue_i$ に格納された更新メッセージから，未だプロセス P_i のローカルメモリに反映されていない因果先行関係で最も先行しているメッセージを選択して，ローカルメモリに反映させる．この選択すべきメッセージとは，入力キューの先頭にある更新メッセージで，そのベクトル・タイムスタンプ $t_j[j]$ が，プロセス P_i のローカルなベクトル・タイムスタンプ $t_i[j]$ より 1 だけ進んでいて，その他のベクトル・タイムスタンプの要素はすべて小さいか等しい更新メッセージである．つまり，既

にプロセス P_i のローカルメモリに反映されたプロセス P_j が発行したストア命令の次にプロセス P_j が発行したストア命令であり，しかも，既にプロセス P_i に反映されたストア命令より因果先行関係で後に発行されたストア命令の更新メッセージである．

この更新メッセージの更新値 v_j をローカルメモリ $M_i(x_j)$ に適用して，$InQueue_i$ からこのメッセージを削除し，ベクトル・タイムスタンプの $t_i[j]$ のみを 1 だけ増加させる．

また，このストア命令 $w_j(x_j, v_j)$ のベクトル・タイムスタンプは，その後に，プロセス P_i で発行されるロード命令 $r_i(x_j, v_j)$ のベクトル・タイムスタンプより小さいか等しくなるため $w_j(x_j, v_j) \rightarrow r_i(x_j, v_j)$ を満足する．つまり，このロード命令は書き込み先導順序を満足している．

同様に，逐次実行列 S_i に観測した観測命令のラベル $per_i(w_j(x_j, v_j))$ を追加する．

$Apply_i$

Δs_i

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

$InQueue_i \neq \langle\rangle$

$pn_i' = pn_i$

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

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

$\quad \wedge\ t_j\, j = t_i\, j + 1$

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

$t_i' j = t_j\, j$

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

$OutQueue_i' = OutQueue_i$

$InQueue_i' = tail\ InQueue_i$

$L_i' = L_i$

$S_i' = S_i \frown \langle per_i(w_j(x_j, v_j))\rangle$

5.1.3 コーザル・メモリの並行性の記述

本項では，value-passing CCS を用いて，コーザル・メモリ $\mathcal{CM}$ の並行性に関する形式的仕様記述を行う．

コーザル・メモリ $\mathcal{CM}$ は，n 個のプロセスを持つ．各プロセスは，前項にて Z を用いて記述した操作スキーマと以下の入出力ポート：

$$P_i : \{id_i, loc_i, \overline{val_i}, heap_i, pipe_i, \overline{pipe_1} \cdots \overline{pipe_{i-1}}, \overline{pipe_{i+1}} \cdots \overline{pipe_n}\}$$

を持ち，$pipe_i, \overline{pipe_1} \cdots \overline{pipe_{i-1}}, \overline{pipe_{i+1}} \cdots \overline{pipe_n}$ を通して他のプロセスと通信を行う．

入力ポート id_i の処理では，プロセス番号を環境から入力する．入出力ポート $pipe_i$ は，プロセス間の通信ポートである．その他のポートは，以下のロード命令 $r_i(x?, v!)$，ストア命令 $w_i(x?, v?)$，同報通信命令 $broadcast_i(message!)$，受信命令 $receive_i$ $(message?)$ 内で使用される．

ラベル $r_i(x?, v!)$ は，以下の不可分なロード命令であり，入力ポート loc_i を通して環境からアドレス $x?$ を受け取り，操作スキーマ $Read_i$ を実行した後，ローカルメモリ $M_i(x?)$ の値 $v!$ を出力ポート $\overline{val_i}$ を通して環境に出力する．

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

ラベル $w_i(x?, v?)$ は，以下の不可分なストア命令であり，入力ポート $heap_i$ を通して環境からアドレス $x?$ と値 $v?$ を受け取り，操作スキーマ $Write_i$ を実行する．

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

ラベル $broadcast_i(message!)$ は，以下の不可分な同報通信命令であり，操作スキーマ $Send_i$ を実行後，$\overline{pipe_i}$ 以外の出力ポート $\overline{pipe}$ を通して他の全てのプロセスと通信する．

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

ラベル $receice_i(message?)$ は，以下の不可分な受信命令であり，入力ポート $pipe_i$ から，上記の同報通信 $broadcast_i(message!)$ の出力ポート $\overline{pipe_i}$ との通信を通して $message?$ を受け取る．CCS では，このような通信を，3.4.3 項の **Concurrent Composition (2)** を用いて記述する．その後，操作スキーマ $Receive_i$ を実行する．

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

以上の定義より，以下のプロセス P_i が定義される．

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

各プロセス P_i は，以下の $\mathcal{CM}$ のように並行に実行する．そして，$pipe_i, \overline{pipe_1} \cdots \overline{pipe_{i-1}}, \overline{pipe_{i+1}} \cdots \overline{pipe_n}$ を通して他のプロセスと内部プロセスの τ 操作として通信を行なうことでローカルメモリの更新情報を交換して，コーザル・メモリ・コンシステンシ・モデルを実現する．

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

5.1.4 コーザル・メモリの展開例

前項で定義したコーザルメモリ $\mathcal{CM}$ に S-CCS の遷移規則を適用した展開例を以下に示す．但し，簡略化のため，プロセスは， P_1, P_2 の二個とする．ここで，x はアドレスを示す定数である．

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

以下，上記の τ 操作についての遷移規則の適用を記述する．

$$
\langle \overline{pipe_2}(message!).P_1, s_7 \rangle \xrightarrow{\overline{pipe_2}((1,x,1,\langle 1,0\rangle))} \langle P_1, s_7 \rangle \tag{5.1}
$$

$$
\langle P_2, s_7 \rangle \xrightarrow{pipe_2((1,x,1,\langle 1,0\rangle))} \langle Receive_2.P_2, s_7' \rangle \tag{5.2}
$$

$$
\langle (\overline{pipe_2}(message!).\ P_1 \mid P_2), s_7 \rangle \xrightarrow{\tau} \langle (P_1 \mid Receive_2.P_2), s_7' \rangle \tag{5.3}
$$

$$
\langle (\overline{pipe_2}(message!).\ P_1 \mid P_2) \setminus K, s_7 \rangle \xrightarrow{\tau} \langle (P_1 \mid Receive_2.P_2) \setminus K, s_7' \rangle \tag{5.4}
$$

$$
\frac{\dfrac{(5.1)\quad (5.2)}{(5.3)}}{(5.4)}\ (\tau \notin K)
$$

次に各展開における状態遷移を下記に記述する．尚，$\oplus$ と $\mapsto$ は，Z の記号であり，f と g をそれぞれ関数とする時，$f \oplus g$ は f に g を上書きする，$x \mapsto v$ は，変数 x に値 v をマッピングするという意味である [8].

$$
\begin{aligned}
&s_0 = \{\} \\
&s'_0 = s_1 = \{pn_1? \mapsto 1\} \\
&s'_1 = s_2 = s_1 \oplus \{pn_1 \mapsto 1, M_1 \mapsto \langle \bot \cdots \bot \rangle, \\
&\quad t_1 \mapsto \langle 0, 0 \rangle, OutQueue_1 \mapsto \langle \rangle, \\
&\quad InQueue_1 \mapsto \langle \rangle, L_1 \mapsto \langle \rangle, S_1 \mapsto \langle \rangle\} \\
&s'_2 = s_3 = s_2 \oplus \{pn_2? \mapsto 2\} \\
&s'_3 = s_4 = s_3 \oplus \{pn_2 \mapsto 2, M_2 \mapsto \langle \bot \cdots \bot \rangle, \\
&\quad t_2 \mapsto \langle 0, 0 \rangle, OutQueue_2 \mapsto \langle \rangle, \\
&\quad InQueue_2 \mapsto \langle \rangle, L_2 \mapsto \langle \rangle, S_2 \mapsto \langle \rangle\} \\
&s'_4 = s_5 = s_4 \oplus \{x? \mapsto x, v? \mapsto 1\} \\
&s'_5 = s_6 = s_5 \oplus \{M_1 \mapsto M_1 \oplus \{x \mapsto 1\}, \\
&\quad t_1 \mapsto \langle 1, 0 \rangle, \\
&\quad OutQueue_1 \mapsto \langle (1, x, 1, \langle 1, 0 \rangle) \rangle, \\
&\quad L_1 \mapsto \langle w_1(x, 1) \rangle, S_1 \mapsto \langle w_1(x, 1) \rangle\} \\
&s'_6 = s_7 = s_6 \oplus \{message! \mapsto (1, x, 1, \langle 1, 0 \rangle), \\
&OutQueue_1 \mapsto \langle \rangle\} \\
&s'_7 = s_8 = s_7 \oplus \{message? \mapsto (1, x, 1, \langle 1, 0 \rangle)\} \\
&s'_8 = s_9 = s_8 \oplus \{InQueue_2 \mapsto \langle (1, x, 1, \langle 1, 0 \rangle) \rangle\}
\end{aligned}
$$

5.2 リリース・コンシステンシ・モデルの実現

本節では，リリース・コンシステンシ・モデルの実現であるリリース・メモリ $\mathcal{RM}$ の同期に関する仕様を Z の表記法と value-passing CCS を統合して記述する．まず，本論文で採用する $\mathcal{RM}$ の実現方式について説明する．次に，Z の表記法と value-passing CCS を用いて $\mathcal{RM}$ の機能と並行性について，それぞれ，分離して記述する．最後に，S-CCS の遷移規則を適用した $\mathcal{RM}$ の展開例を挙げる．

5.2.1 リリース・メモリの実現方式

$\mathcal{RM}$ で採用するリリース・コンシステンシ・モデルの実現方式は，以下のものである [4, 3].

- 同期モデル：セマフォを使った分散同期サービス
- 更新オプション：書き込み時更新方式
- 粒度：変数単位 [1]

書き込み時更新方式とは，あるプロセスが発行したストア命令が，そのローカルメモリの共有変数を更新してその共有変数のコピーを他の全てのプロセスに同報通信した後，他の全てのプロセスのローカルメモリの同じ共有変数が更新されるまで待つ方式である．但し，リリース・コンシステンシ・モデルでは，ある解放命令より先行して発行された全てのストア命令は，それぞれの共有変数のコピーを他の全てのプロセスに同報通信した後，終了する．そして，個々のストア命令の代わりに，その解放命令が他の全てのプロセスのローカルメモリの同じ共有変数が更新されるまで待つ方式である．

[1] 通常，物理的なページ単位を粒度とするが，仮想メモリの問題を含む複雑な問題となるため本論文では変数単位とした．

5.2.2 リリース・メモリの機能の記述

各プロセス P_i は，以下の状態スキーマ s_i を持つ．各 s_i は，プロセス番号 pn_i，ローカルメモリ M_i，同期変数 z_i，ストア命令番号 t_i，他プロセスとの通信キュー $OutQueue_i$ と $InQueue_i$，受け取り確認関数 $ConfirmF_i$，実行履歴 L_i，および，命令集合 A^H_{i+w} の逐次実行列 S_i を持つ．

まず，以下に，リリース・メモリシステムの基本型宣言，略記定義，型，および，操作スキーマ s_i を定義する．

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

$$write\_tuple == \mathbb{N}_1 \times (\mathbb{N}_1 \cup \{\top\}) \times \mathbb{N}_1 \times \mathcal{M} \times Val$$
$$confirmed\_process == \mathbb{N}_1 \pfun \operatorname{seq} \mathbb{N}_1$$
$$Bool == \{true, false\}$$

$$MaxOutQueue, MaxInQueue : \mathbb{N}_1$$
$$MaxSerial, MaxLocalHis : \mathbb{N}_1$$
$$update\_confirmation : confirmed\_process \pfun (confirmed\_process \times Bool)$$

s_i

$pn_i : \mathbb{N}_1$

$M_i : \mathcal{M} \pfun (Val \cup \{\bot\})$

$z_i : Syn \cup \{\bot\}$

$t_i : \mathbb{N}$

$OutQueue_i : \mathrm{seq}\ write\_tuple$

$InQueue_i : \mathrm{seq}\ write\_tuple$

$ConfirmF_i : confirmed\_process$

$L_i : \mathrm{seq}\ A$

$S_i : \mathrm{seq}\ A$

$\#OutQueue_i \leq MaxOutQueue$

$\#InQueue_i \leq MaxInQueue$

$\#L_i \leq MaxLocalHis$

$\#S_i \leq MaxSerial$

各プロセス P_i は，以下の操作スキーマ $InitP_i$ で初期化される．

$InitP_i$

s_i'

$pn_i? : \mathbb{N}_1$

$pn_i' = pn_i?$

$M_i' = \lambda\, x : \mathcal{M} \bullet \bot$

$z_i = \bot$

$t_i = 0$

$OutQueue_i' = \langle\rangle$

$InQueue_i' = \langle\rangle$

$ConfirmF_i' = \langle\rangle$

$L_i' = \langle\rangle$

$S_i' = \langle\rangle$

各プロセス P_i は，後述する入力ポート loc_i を通して環境からアドレス $x?$ を受け取ると，以下の操作スキーマ $Read_i$ を発行してローカルメモリ $M_i(x?)$ の値 $v!$ を受理する．また，実行履歴 L_i と逐次実行列 S_i にロード命令のラベル $r_i(x?, v!)$ を追加する．

$Read_i$

Δs_i

$x? : \mathcal{M}$

$v! : Value$

$v! = M_i x?$

$pn_i' = pn_i$

$M_i' = M_i$

$z_i' = z_i$

$t_i' = t_i$

$OutQueue_i' = OutQueue_i$

$InQueue_i' = InQueue_i$

$ConfirmF_i' = ConfirmF_i$

$L_i' = L_i \frown \langle r_i(x?, v!) \rangle$

$S_i' = S_i \frown \langle r_i(x?, v!) \rangle$

各プロセス P_i は，後述する入力ポート $heap_i$ を通して環境からアドレス $x?$ と値 $v?$ を受け取ると，以下の操作スキーマ $Write_i$ を発行してローカルメモリ $M_i(x?)$ の値を $v?$ に更新する．また，出力キュー $OutQueue_i$ に更新メッセージ(起点プロセス番号 pn_i, 終点プロセス番号 $\top$, ストア命令番号 t_i',$x?$,$v?$)を追加する．尚，終点プロセス番号が $\top$（上限）とは全てのプロセスを示す．また，全てのストア命令は，各プロセス内のストア命令番号 t_i' で区別できる．そこで，ストア命令番号 t_i' を定義域，書き込み更新メッセージを送ってきたプロセスのプロセス番号列を値域とする受け取り確認関数 $ConfirmF_i$ のストア命令番号 t_i' に対する値を $\langle\rangle$ に初期化する．同様に，実行履歴 L_i と逐次実行列 S_i にストア命令のラベル $w_i(x?, v?)$ を追加する．

$Write_i$

Δs_i

$x? : \mathcal{M}$

$v? : Value$

$pn_i' = pn_i$

$M_i' = M_i \oplus \{x? \mapsto v?\}$

$z_i' = z_i$

$t_i' = t_i + 1$

$OutQueue_i' = OutQueue_i \frown \langle (pn_i, \top, t_i', x?, v?) \rangle$

$InQueue_i' = InQueue_i$

$ConfirmF_i' = ConfirmF_i \oplus \{t_i' \mapsto \langle\rangle\}$

$L_i' = L_i \frown \langle w_i(x?, v?) \rangle$

$S_i' = S_i \frown \langle w_i(x?, v?) \rangle$

各プロセス P_i は，以下の操作スキーマ $Send_i$ を適時発行して，出力キュー $OutQueue_i$ 内の更新情報を，後述する $\overline{pipe_j}$ を通して他の全てのプロセスに同報通信する．

$Send_i$

Δs_i

$message! : write\_tuple$

$OutQueue_i \neq \langle\rangle$

$pn_i' = pn_i$

$M_i' = M_i$

$z_i' = z_i$

$t_i' = t_i$

$message! = head\ OutQueue_i$

$OutQueue_i' = tail\ OutQueue_i$

$InQueue_i' = InQueue_i$

$ConfirmF_i' = ConfirmF_i$

$L_i' = L_i$

$S_i' = S_i$

各プロセス P_i は，後述する $pipe_j$ を通して通信されたメモリ更新の情報を受け取ると，以下の操作スキーマ $Receive_i$ を発行する．$Receive_i$ は，受け取ったメモリ更新の中の終点プロセス番号が $\top$（上限）またはプロセス P_i 自身宛の時のみ，このメモリ更新情報を入力キュー $InQueue_i$ に追加する．

$Receive_i$

Δs_i

$message? : write\_tuple$

$(src_j, tar_j, t_j, x_j, v_j) : write\_tuple$

$(src_j, tar_j, t_j, x_j, v_j) = message?$

$tar_j = \top \lor tar_j = pn_i$

$pn_i' = pn_i$

$M_i' = M_i$

$z_i' = z_i$

$t_i' = t_i$

$OutQueue_i' = OutQueue_i$

$InQueue_i' = InQueue_i \frown \langle message? \rangle$

$ConfirmF_i' = ConfirmF_i$

$L_i' = L_i$

$S_i' = S_i$

各プロセス P_i は，以下の操作スキーマ $Apply_i$ を適時発行して，$InQueue_i$ に格納された更新メッセージの中から終点プロセス番号が $\top$（上限）の更新メッセージを選択して，ローカルメモリに逐次反映させる．また，この起点プロセス番号 src_j のプロセスにストア命令書き込み確認メッセージを送るために，書き込み更新メッセージを $OutQueue_i$ に追加する．

$Apply_i$

Δs_i

$(src_j, tar_j, t_j, x_j, v_j) : write\_tuple$

$InQueue_i \neq \langle\rangle$

$pn_i' = pn_i$

$(src_j, tar_j, t_j, x_j, v_j) = head\ InQueue_i$

$tar_j = \top$

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

$z_i' = z_i$

$t_i' = t_i$

$OutQueue_i' = OutQueue_i \frown \langle(pn_i, src_j, t_j, x_j, v_j)\rangle$

$InQueue_i' = tail\ InQueue_i$

$ConfirmF_i' = ConfirmF_i$

$L_i' = L_i$

$S_i' = S_i \frown \langle per_i(w_j(x_j, v_j))\rangle$

各プロセス P_i は，後述する入力ポート syn_a_i を通して環境から同期変数 $z?$ を受け取ると，操作スキーマ Acq_i を発行して，$z?$ に対応したクリティカル・セクションに入る．同様に，実行履歴 L_i と逐次実行列 S_i に獲得命令のラベル $acq_i(z?)$ を追加する．

Acq_i

Δs_i

$z? : Syn$

$pn_i' = pn_i$

$M_i' = M_i$

$z_i' = z?$

$t_i' = t_i$

$OutQueue_i' = OutQueue_i$

$InQueue_i' = InQueue_i$

$ConfirmF_i' = ConfirmF_i$

$L_i' = L_i \frown \langle acq_i(z?) \rangle$

$S_i' = S_i \frown \langle acq_i(z?) \rangle$

各プロセス P_i は，後述する入力ポート syn_r_i を通して環境から同期変数 $z?$ を受け取ると，操作スキーマ Rel_i を発行して，$z?$ に対応した書き込み更新メッセージを他の全てのプロセスから受けたか否かを確認する．P_i は，まず，$InQueue_i$ から自分宛に来た書き込み更新メッセージを選択すると，このメッセージを先頭から削除する．次に，更新メッセージ内のストア命令番号 t_j に対応する $ConfirmF_i$ の値に更新メッセージ内のプロセス番号 src_j を追加した $ConfirmF_i$ を入力として関数 *update_confirmation* を実行する．関数 *update_confirmation* は，列の要素のプロセス番号列が，他の全てのプロセス番号を持てば，このストア命令番号とプロセス番号列の対を $ConfirmF_i$ から削除する．さらに，*update_confirmation* は，$ConfirmF_i$ が $\langle\rangle$ になっていれば，今までにプロセス P_i が発行した全てのストア命令に対する他の全てのプロセスからの更新メッセージを受けたことが確認されたため，*update!* に *true* を，そうでなければ，*false* を返す関数である．この

関数の記述は，本論文では省略する．次に，P_i は，$update!$ が $true$ の時は，同期変数を $\perp$ に初期化し，実行履歴 L_i と逐次実行列 S_i に解放命令のラベル $rel_i(z?)$ を追加する．

Rel_i

Δs_i

$(src_j, tar_j, t_j, x_j, v_j) : write\_tuple$

$z? : Syn$

$update! : Bool$

$InQueue_i \neq \langle\rangle$

$(src_j, tar_j, t_j, x_j, v_j) = head\ InQueue_i$

$tar_j = pn_i$

$InQueue_i' = tail\ InQueue_i$

$pn_i' = pn_i$

$M_i' = M_i$

$z_i' = z?$

$t_i' = t_i$

$OutQueue_i' = OutQueue_i$

$(ConfirmF_i', update!) = update\_confirmation($
$\quad ConfirmF_i \oplus \{t_j \mapsto (ConfirmF_i\ t_j \frown \langle src_j \rangle)\})$

$update! = true \Rightarrow z_i' = \perp \wedge\ L_i' = L_i \frown \langle rel_i(z?) \rangle$
$\quad \wedge\ S_i' = S_i \frown \langle rel_i(z?) \rangle$

$update! = false \Rightarrow z_i' = z_i \wedge L_i' = L_i \wedge S_i' = S_i$

5.2.3 リリース・メモリの並行性の記述

本項では，value-passing CCS を用いて，$\mathcal{RM}$ の並行性および同期に関する形式的仕様記述を行う．

各プロセス P_i は，入出力ポートとして，

$$P_i : \{id_i, loc_i, \overline{val_i}, heap_i, pipe_i, \overline{pipe_i}, syn\_a_i, acq, \overline{rel_i}, syn\_r_i, confirm_i\}$$

を持つ．入力ポート id_i は，プロセス番号を環境から入力する．

ラベル $r_i(x?, v!)$ は，$\mathcal{CM}$ 同様，以下の不可分なロード命令であり，入力ポート loc_i を通して環境からアドレス $x?$ を受け取り，操作スキーマ $Read_i$ を実行した後，ローカルメモリ $M_i(x?)$ の値 $v!$ を出力ポート $\overline{val_i}$ を通して環境に出力する．

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

ラベル $w_i(x?, v?)$ は，$\mathcal{CM}$ 同様，以下の不可分なストア命令であり，入力ポート $heap_i$ を通して環境からアドレス $x?$ と値 $v?$ を受け取り，操作スキーマ $Write_i$ を実行する．

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

ラベル $broadcast_i(message!)$ は，$\mathcal{CM}$ 同様，以下の不可分な同報通信命令であり，操作スキーマ $Send_i$ を実行後，$\overline{pipe_i}$ 以外の出力ポート $\overline{pipe_j}$ を通して他の全てのプロセスと通信する．

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

ラベル $receice_i(message?)$ は，$\mathcal{CM}$ 同様，以下の不可分な受信命令であり，入力ポート $pipe_i$ から，上記の同報通信 $broadcast_i(message!)$ の出力スキーマ $\overline{pipe_i}$ との通信を通して $message?$ を受け取る．CCS では，このような同期通信を，3.4.3

項の **Concurrent Composition (2)** を用いて記述する．その後，操作スキーマ $Receive_i$ を実行する．

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

ラベル $acq_i(z?)$ は，以下の同期変数 $z?$ を持つ不可分な獲得命令であり，入力ポート syn_a_i を通して環境から $z?$ を受け取り，操作スキーマ Acq_i を実行する．次に，後述するセマフォ Sem から入力ポート acq を通して $z?$ を受け取り，$z?$ に対応したクリティカル・セクションに入る．

$$acq_i(z?) \equiv syn\_a_i(z?).Acq_i.acq(z?).$$

ラベル $rel_i(z?)$ は，以下の同期変数 $z?$ を持つ不可分な解放命令であり，入力ポート syn_r_i を通して環境から $z?$ を受け取り，エージェント $check_i(z?)$ を起動する．そして，入力ポート $confirm_i$ を通して操作スキーマ Rel_i から $update!$ を受け取り，$z?$ に対応したクリティカル・セクション内の書き込み時更新が終了するまで受信と確認を繰り返す．終了確認後，出力ポート $\overline{rel}$ を通してセマフォ Sem に $z?$ を送り，$z?$ に対応したクリティカル・セクションから出る．

$$\begin{aligned}
rel_i(z?) &\equiv syn_r_i(z?).check_i(z?).\overline{rel}(z?) \\
check_i(z?) &\stackrel{\text{def}}{=} Rel_i.confirm_i(update!). \\
&\quad ((if \quad update! = true \; then \; \mathbf{0}) + \\
&\quad (if \quad update! = false \; then \; receive_i(message?).check_i(z?)))
\end{aligned}$$

以上の定義により，以下のプロセス P_i が定義される．

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

$$receive_i(message?).P_i +$$
$$Apply_i.P_i +$$
$$rel_i(z?).P_i$$

各プロセス P_i は以下のセマフォで同期を取る．但し，同期変数の集合 $Syn = \{a, \cdots, z\}$ に対するセマフォは，それぞれ，以下の $Sem_a, \cdots, Sem_z$ とする．

$$\begin{aligned}
Sem_a &\stackrel{\text{def}}{=} \overline{acq}(a).rel(a).Sem_a \\
&\cdots \\
Sem_z &\stackrel{\text{def}}{=} \overline{acq}(z).rel(z).Sem_z \\
\mathcal{RM} &\stackrel{\text{def}}{=} id(pn_1?).InitP_1.\cdots.id(pn_n?).InitP_n. \\
&\quad (Sem_a \mid \cdots \mid Sem_z \mid P_1 \mid \cdots \mid P_n) \setminus L \\
&\quad where\ L = \{acq(a), \cdots, acq(z), rel(a), \cdots, rel(z), pipe_1, \cdots, pipe_n\}
\end{aligned}$$

5.2.4 リリース・メモリの展開例

上記で定義した $\mathcal{RM}$ の同期に関係する部分の展開例を以下に示す．但し，簡略化のため，プロセスは P_1, P_2 の 2 個，同期変数は y, z の 2 個とする．

$$\begin{aligned}
& \langle \mathcal{RM}, s_0 \rangle \\
\xrightarrow{\omega syn_a_1(z)} & \langle (Sem_y \mid Sem_z \mid Acq_1.acq(z).P_1 \mid P_2) \setminus L, s_0' \rangle \\
\xrightarrow{Acq_1} & \langle (Sem_y \mid \overline{acq}(z).rel(z).Sem_z \mid acq(z).P_1 \mid P_2) \setminus L, s_1' \rangle \\
\xrightarrow{\tau} & \langle (Sem_y \mid rel(z).Sem_z \mid w_1(x?, v?).P_1 \mid P_2) \setminus L, s_2' \rangle \\
\xrightarrow{w_1(x_1,1)} & \langle (Sem_y \mid rel(z).Sem_z \mid broadcast_1(message!).P_1 \mid P_2) \setminus L, s_3' \rangle \\
\xrightarrow{broadcast_1(z,1,\top,1,x_1,1)} & \langle (Sem_y \mid rel(z).Sem_z \mid P_1 \mid receive_2(message?).P_2) \setminus L, s_4' \rangle \\
\xrightarrow{receive_2(z,1,\top,1,x_1,1)} & \langle (Sem_y \mid rel(z).Sem_z \mid P_1 \mid Apply_2.P_2) \setminus L, s_5' \rangle \\
\xrightarrow{Apply_2} & \langle (Sem_y \mid rel(z).Sem_z \mid receive_1(message?).P_1 \mid P_2) \setminus L, s_6' \rangle \\
\xrightarrow{receive_1(z,2,1,1,x_1,1)} & \langle (Sem_y \mid rel(z).Sem_z \mid rel_1(z?).P_1 \mid P_2) \setminus L, s_7' \rangle
\end{aligned}$$

$$
\begin{aligned}
&\xrightarrow{syn\_r_1(z)} && \langle (Sem_y \mid rel(z).Sem_z \mid check_1(z).\overline{rel}(z).P_1 \mid P_2) \setminus L, s'_8 \rangle \\
&\xrightarrow{check_1(z)} && \langle (Sem_y \mid rel(z).Sem_z \mid \overline{rel}(z).P_1 \mid P_2) \setminus L, s'_9 \rangle \\
&\xrightarrow{\tau} && \langle (Sem_y \mid Sem_z \mid P_1 \mid acq_2(z?).P_2) \setminus L, s'_{10} \rangle \\
&\xrightarrow{syn\_a_2(z)} && \langle (Sem_y \mid Sem_z \mid P_1 \mid Acq_2.acq(z).P_2) \setminus L, s'_{11} \rangle \\
&\xrightarrow{Acq_2} && \langle (Sem_y \mid \overline{acq}(z).rel(z).Sem_z \mid P_1 \mid acq(z).P_2) \setminus L, s'_{12} \rangle \\
&\xrightarrow{\tau} && \langle (Sem_y \mid rel(z).Sem_z \mid P_1 \mid P_2) \setminus L, s'_{13} \rangle \\
&\cdots
\end{aligned}
$$

また，上記のリリース・メモリの展開例のトレース上の最初の τ 操作に適用された遷移規則の適用例を示す．この例のように，1 ステップの τ 操作は，遷移規則を複数回適用して導出する．

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\langle \overline{acq}(z).rel(z).Sem_z, s'_1 \rangle \xrightarrow{\overline{acq}(z)} \langle rel(z).Sem_z, s'_1 \rangle \quad \langle acq(z).P_1, s'_1 \rangle \xrightarrow{acq(z)} \langle P_1, s'_2 \rangle}{\langle (\overline{acq}(z).rel(z).Sem_z \mid acq(z).P_1), s'_1 \rangle \xrightarrow{\tau} \langle (rel(z).Sem_z \mid P_1), s'_2 \rangle}}{\langle (Sem_y \mid \overline{acq}(z).rel(z).Sem_z \mid acq(z).P_1), s'_1 \rangle \xrightarrow{\tau} \langle (Sem_y \mid rel(z).Sem_z \mid P_1), s'_2 \rangle}}{\langle (Sem_y \mid \overline{acq}(z).rel(z).Sem_z \mid acq(z).P_1 \mid P_2), s'_1 \rangle \xrightarrow{\tau} \langle (Sem_y \mid rel(z).Sem_z \mid P_1 \mid P_2), s'_2 \rangle}}{\langle (Sem_y \mid \overline{acq}(z).rel(z).Sem_z \mid acq(z).P_1 \mid P_2) \setminus L, s'_1 \rangle \xrightarrow{\tau} \langle (Sem_y \mid rel(z).Sem_z \mid P_1 \mid P_2) \setminus L, s'_2 \rangle} \ (\tau \notin L)
$$

6. メモリ・コンシステンシ・モデルの実現の検証

本章では，コーザル・メモリ・コンシステンシ・モデルの実現であるコーザル・メモリ$\mathcal{CM}$とリリース・コンシステンシ・モデルの実現であるリリース・メモリ$\mathcal{RM}$の仕様記述が，それぞれのモデルの仕様記述を満足しているかどうかを検証する．

6.1 コーザル・メモリの検証

本節では，5.1節にて記述されたコーザル・メモリが4.2節にて定義されたコーザル・メモリ・コンシステンシ・モデルの要求を満足しているかどうか検証する．

5.1節では，コーザル・メモリ$\mathcal{CM}$の状態と操作で表現される機能はZを用い，操作の実行順序，通信，および並行性などはvalue-passing CCSを用いて分離して記述した．そこで，まず，証明の対象となっている命令間のトレースを，value-passing CCSを用いた仕様記述に3.4節の遷移規則を適用して，S-CCSを用いて記述する．次に，このプロセス展開と同時に遷移したローカルメモリ，各種キュー，弱ベクトル時計などの状態遷移をZの操作スキーマの記述から求めて検証を行う．この様に，検証においても，プロセス展開と状態遷移に分離して行う．

本節では，Ahamad等[6]と同様に，下記の[補題1], [補題2]および[定理1]を証明する．

[補題1] Hはコーザル・メモリ$\mathcal{CM}$の実行履歴，$ts(o)$は操作oのタイムスタンプとする．o_1とo_2が$o_1 \rightsquigarrow o_2$を満たす命令であれば$ts(o_1) \preceq ts(o_2)$が成り立つ．更にo_2がP_iのストア命令である場合は$ts(o_1)[i] < ts(o_2)[i]$，すなわち，$ts(o_1) \prec ts(o_2)$が成り立つ．

（証明） Aをシステムの操作スキーマと操作の集合，$t_i(o)$と$t'_i(o)$をそれぞれ操作oの開始時と終了時のプロセスP_iのベクトル・タイムスタンプとする．

関係$\rightsquigarrow$の定義に従って以下の3つの場合について証明する．

1. $o_1 \vec{i} o_2$の場合

プロセスP_iが，ロード命令またはストア命令であるo_1とo_2をこのプログラ

ム順序で発行したとき，value-passing CCS の記述と S-CCS の遷移規則を用いて下記のようなトレース ω を得ることができる．

$$o_1, o_2 \in L_i, \exists \omega \in A^* \bullet$$
$$\langle o_1.E_1, s_1\rangle \xrightarrow{\omega} \langle o_2.E_2, s_2\rangle$$

弱ベクトル時計を採用した P_i の操作スキーマの定義では，操作スキーマ $Write_i$ と $Apply_i$ のみが下記のようにローカルなタイムスタンプ t_i を更新する:

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

上記の定義より，o_1 から o_2 へのプロセス展開においてタイムスタンプの値を小さくする操作スキーマは存在しないため $ts(o_1) \preceq ts(o_2)$ が成立する．更に，o_2 が P_i のストア命令の場合は，o_2 において $Write_i$ を実行するため $ts(o_1)[i] < ts(o_2)[i]$ となり $ts(o_1) \prec ts(o_2)$ が成立する．

2. $o_1 \mapsto o_2$ の場合

$\mapsto$ の定義より o_1 はストア命令，例えば $w_j(x_l, v_l)$，o_2 は対応するロード命令，例えば $r_i(x_l, v_l)$ とする．同様に遷移規則を用いて下記のような逐次実行列を作ることができる (図 5.1 の $w_2(x,1) \rightarrow r_1(x,1)$ を参照)．

$$\exists\, w_j(x_l, v_l), \exists\, r_i(x_l, v_l) \in A^H_{i+w}, \exists \omega \in A^* \bullet$$
$$\langle w_j(x?, v?).E_j, s_j\rangle \xrightarrow{\omega} \langle E_i, s_i\rangle$$
$$\wedge\ \omega = w_j(x_l, v_l) \ldots Send_j \ldots$$
$$\tau\,^2 \ldots$$
$$Receive_i \ldots Apply_i \ldots r_i(x_l, v_l)$$

ここで，上記で証明した $o_1 \xrightarrow[i]{} o_2 \Rightarrow ts(o_1) \preceq ts(o_2)$ を適用することによって以下の関係が成立する:

$$ts(w_j(x_l, v_l)) \preceq ts(Send_j) \preceq ts(\overline{pipe_i}(message_l))$$
$$ts(pipe_i(message_l)) \preceq ts(Receive_i) \prec ts(Apply_i) \preceq ts(r_i(x_l, v_l))$$

[2] $\overline{pipe_i}(message!)$ と $pipe_i(message?)$ との内部隠蔽操作

$InQueue_i$ から取り出す $w_j(x_l, v_l)$ に対応する更新メッセージ $message_l$ 中のタイムスタンプ $ts(w_j(x_l, v_l))$ と操作スキーマ $Apply_i$ のタイムスタンプの関係は，操作スキーマ $Apply_i$ の記述より，

$$k : 1 \ldots n \mid k \neq j \bullet ts(w_j(x_l, v_l))[k] \leq t_i'(Apply_i)[k]$$
$$ts(w_j(x_l, v_l))[j] = t_i(Apply_i)[j] + 1 = t_i'(Apply_i)[j]$$

である．よって，

$$ts(w_j(x_l, v_l)) \preceq t_i'(Apply_i) = ts(Apply_i) \preceq ts(r_i(x_l, v_l))$$

ここに $ts(o_1) \preceq ts(o_2)$ が成立する．

3. $o_1 \rightsquigarrow o' \rightsquigarrow o_2$ の場合

o_1 と o' および o' と o_2 が，それぞれ，上記のように $\underset{i}{\rightarrow}$ または $\mapsto$ を満たす ω_1, ω_2 を持つ逐次実行列で表現されるとき，**trace** 遷移規則を繰り返し適用することにより以下のような遷移規則を得ることができる．

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

o_1 から o_2 へのトレース $\omega_1\omega_2$ においてタイムスタンプの値を小さくする操作スキーマは存在しないため $ts(o_1) \preceq ts(o_2)$ となる．更に，o_2 がプロセス P_i のストア命令の場合は，o_2 において $Write_i$ を実行するため $ts(o_1)[i] < ts(o_2)[i]$ となり $ts(o_1) \prec ts(o_2)$ が成立する．

□

[補題 2] H をコーザル・メモリ $\mathcal{CM}$ の実行履歴，$w_j(x_l, v_l)$ を任意のプロセス P_j の任意のストア命令とする．このとき，全てのプロセスは，必ずいつかは $w_j(x_l, v_l)$ をそれぞれのローカルメモリに適用できる．

（証明） 任意のプロセス P_i は，自ら発行した $w_i(x_l, v_l)$ に関しては，操作スキーマ $Write_i$ 中の $M_i' = M_i \oplus \{x_l \mapsto v_l\}$ にてローカルメモリに更新するため，必ず，ローカルメモリに適用できる．そこで，以下では，任意のプロセス $P_j(j \neq i)$ が発行した $w_j(x_l, v_l)$ を，任意のプロセス P_i が観測する場合について証明する．

プロセス P_j が $w_j(x_l, v_l)$ の更新メッセージを一次的に蓄積する $OutQueue_j$ は先入先出の有限キューであるため，$broadcast_j$ によって $w_j(x_l, v_l)$ の更新メッセージ $message_l$ は，いつかは，その他の全てのプロセスに同報通信される．任意のプロセス P_i は $receive_i(message_l)$ を発行して，そのメッセージを $\overline{pipe_i}(message_l)$ と $pipe_i(message_l)$ との τ 操作によって受信する．ここに，value-passing CCS の記述と遷移規則から，以下のようなトレース ω を得ることができる．

$$\exists \omega \in A^* \bullet$$
$$\langle w_j(x?, v?).E_j, s_j\rangle \xrightarrow{\omega} \langle Receive_i.E_i, s_i\rangle$$
$$\wedge\ \omega = w_j(x_l, v_l) \ldots Send_j \ldots \tau \ldots$$

次に，その更新メッセージ $message_l$ は P_i の操作スキーマ $Receive_i$ によってベクトル・タイムスタンプをソートキーとして持つ有限優先順位付き待ち行列 $InQueue_i$ に追加される．ところで，ベクトル・タイムスタンプ $ts[j]$ は P_j でのストア命令の回数を示している．今，メッセージ $message_l$ のベクトル・タイムスタンプのすべての要素の合計を m とすると，$\mathcal{CM}$ は $w_j(x_l, v_l)$ を含めてベクトル・タイムスタンプが $ts(w_j(x_l, v_l))$ 以下の更新メッセージを高々 m 個だけ発行している．そこで，送られてきた更新メッセージは必ず $InQueue_i$ の先頭から m 番目以内の要素として挿入され，取り出されるまで m 番目以内であり続ける．その後，P_i の非決定性より以下のトレース ω が存在する．

$$\exists \omega \in A^* \bullet$$
$$\langle Receive_i.E_i, s_i\rangle \xrightarrow{\omega} \langle Apply_i.E_l, s_l\rangle$$

$Apply_i$ は，$InQueue_i$ の先頭の更新メッセージをローカルメモリに適用しては

先頭から削除する．また，この更新メッセージを受信した後，このベクトル・タイムスタンプより小さい更新メッセージを受信する可能性がある．しかし，このようなメッセージは，高々$(m-1)$ 個である．そこで，P_i の非決定性より $Apply_i$ は，有限回実行して，この受信した更新メッセージは，いつかは $InQieue_i$ の先頭に来る．

t_i を P_i が上記のように $Apply_i$ を発行する直前の状態 s_l 内のタイムスタンプとする．

以下，この先頭に来た更新メッセージのベクトル・タイムスタンプが，操作スキーマ $Apply_i$ の定義より，$k : 1..\#t_i \mid k \neq j$ について $ts(w_j(x_l, v_l))[k] \leq t_i[k]$ かつ $ts(w_j(x_l, v_l))[j] = t_i[j] + 1$ の選択条件を満足して選択されるかどうかを調べる．

P_k を P_j 以外のプロセス，w' を P_k による $ts(w_j(x_l, v_l))[k]$ 番目のストア命令とする．これは $Apply_i$ の定義と弱ベクトル時計の定義より P_k が，$w_j(x_l, v_l)$ を実行する以前に w' を P_k ローカルメモリに適用していることを示している．つまり，$ts(w') \prec ts(w_j(x_l, v_l))$ である．従って，状態 s_l 以前に，P_i は既に $InQueue_i$ より w' の更新メッセージを選択してそのローカルメモリに適用しているか，または，この後で，P_k より w' の更新メッセージを受信するかである．後者の場合もいずれ $InQueue_i$ の先頭に来て P_i のローカルメモリに適用される．一旦，P_i が w' を適用すると選択条件 $ts(w_j(x_l, v_l))[k] \leq t_i[k]$ を満足する．同様に，P_j の $(ts(w_j(x_l, v_l))[j] - 1)$ 番目のストア命令も状態が s_l になる以前に適用されている．その後は，弱ベクトル時計の定義より選択条件 $ts(w_j(x_l, v_l))[j] = t[j] + 1$ を満たす．

よって，P_i の非決定性より P_i は操作スキーマ $Apply_i$ を有限回実行して，ついには，$Apply_i$ で記述されている上記の選択条件を満たした $w_j(x_l, v_l)$ の更新メッセージを $InQueue_i$ から選択してローカルメモリに適用できる．

□

[定理 1] H を コーザル・メモリ $\mathcal{CM}$ の実行履歴とすると，H はコーザル・メモリ・コンシステンシ・モデルの条件を満たしている．

(証明) 操作スキーマ $Read_i$ より，プロセス P_i の逐次実行列 S_i には P_i が発行した全てのロード命令が含まれる．また，[補題 2] より全てのプロセスから発行されたストア命令はプロセス P_i の逐次実行列 S_i に含まれる．よって，

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

となる．

操作スキーマ $Read_i$, $Write_i$, および，$Apply_i$ の記述より，プロセス P_i はローカルメモリ M_i を $M_i' = M_i \oplus \{x? \mapsto v?\}$ によって更新して，$v! = M_i x?$ により報告する．よって，常に，ロード命令はローカルメモリ内に最新のストア命令が書いた値を読みプロセスに通知する．したがって，以下の遷移が存在する:

$$x_l \in \mathcal{M}, v_k, v_l, v_m \in Val, \forall i : 1 \ldots n \bullet$$
$$(\forall\, r_i(x_l, v_l) \in S_i \; of \; A^H_{i+w},$$
$$\exists j : 1 \ldots n, \exists \omega \in A^* \bullet$$
$$((\exists\, per_i(w_j(x_l, v_k)) \in S_i \; of \; A^H_{i+w},$$
$$per_i(w_j(x_l, v_m)) \notin \omega \bullet$$
$$\langle per_i(w_j(x_l, v_k)).E_l, s_l\rangle$$
$$\xrightarrow{per_i(w_j(x_l,v_k))\, \omega\, r_i(x_l,v_l)} \langle E_m, s_m\rangle)$$
$$\vee\, (((\exists\, per_i(w_j(x_l, v_m)) \in S_i \; of \; A^H_{i+w},$$
$$per_i(w_j(x_l, v_m)) \notin \omega \bullet$$
$$\langle start.E_0, s_0\rangle \xrightarrow{\omega\, r_i(x_l,v_l)} \langle E_m, s_m\rangle)$$
$$\vee\, per_i(w_j(x_l, v_m)) \notin S_i \; of \; A^H_{i+w})$$
$$\Rightarrow v_l = \bot)))$$

$per_i(w_j(x_l, v_k))$ (操作スキーマ $Apply_i$) の展開による状態 s_l の次の状態 s_l' のローカルメモリ M_i は，値 $x_l \mapsto v_k$ を持つ．状態 s_l' からラベル $r_i(x_l, v_l)$ の展開による状態 s_m までのトレース ω 中には，$per_i(w_j(x_l, v_m))$ は含まれないためローカ

ルメモリ M_i 中のロケーション x_l の値は上書きされることはなく $v_l = v_k(= M_i\, x_l)$ である．よって，逐次実行列 S_i は有効である．

次に，o_1 と o_2 を $o_1 \rightsquigarrow o_2$ の関係を満たす A^H_{i+w} に含まれる命令とする．[補題 1] により $ts(o_1) \preceq ts(o_2)$ が成立する．$\rightsquigarrow$ の定義より以下のいずれの場合でも，逐次実行列 S_i 上で o_1 が o_2 より先行していることを示す．以下では $j \neq i$ とする．

1. $o_1 \underset{i}{\rightarrow} o_2$ by P_i の場合
   操作スキーマ $Read_i$ と $Write_i$ の記述より，プロセス P_i でこのプログラム順序で発行された命令は，逐次実行列 S_i の定義から無条件に o_1 は o_2 より先行している．
2. $o_1 \underset{j}{\rightarrow} o_2$ の場合
   これは o_1 と o_2 の両方がストア命令の場合であり，[補題 1] より $ts(o_1) \prec ts(o_2)$ となる．また，操作スキーマ $Receive_i$ と $Apply_i$ の記述より，o_1 は o_2 より先行して P_i のローカルメモリ M_i に適用されている．よって S_i 上で o_1 は o_2 より先行している．
3. $o_1 \mapsto o_2$ の場合
   これは $\mapsto$ の定義より o_1 はストア命令，o_2 はロード命令である．[補題 1] より $ts(o_1) \prec ts(o_2)$ となる．上記 2. と同様に操作スキーマ $Receive_i$, $Apply_i$ および $Read_i$ より，o_1 は o_2 より先行して $Apply_i$ により P_i のローカルメモリ M_i に適用されその後，o_2 が発行されている．よって S_i 上で o_1 は o_2 より先行している．
4. $o_1 \rightsquigarrow o' \rightsquigarrow o_2$ の場合
   上記のように $\underset{i}{\rightarrow}$ または $\mapsto$ を満たす ω_1, ω_2 を持つ逐次実行列のもつ場合は，3.4.3 項の **trace** 遷移規則を繰り返し適用することにより以下のような遷移規則を得ることができ，常に S_i 上で o_1 は o_2 より先行している．

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

上記の 1. の条件は無条件に成立している．また，4. の条件は，1., 2., 3. の条件が成立していれば常に成立する．よって 2., 3. の条件を示すだけで良い．つ

まり下記の条件が成立している.

$$
\begin{aligned}
&j : 1 \ldots n, x_l, x_m \in \mathcal{M}, v_l, v_m \in Val, \forall\, i : 1 \ldots n \bullet \\
&((\forall\, w_j(x_l, v_l), \forall\, w_j(x_m, v_m) \in A^H_{i+w}, \exists \omega_1, \omega_2 \in A^* \bullet \\
&\quad (\langle w_j(x?, v?).E_l, s_l\rangle \xrightarrow{w_j(x_l,v_l)\,\omega_1\, w_j(x_m,v_m)} \langle E_m, s_m\rangle \Rightarrow \\
&\qquad \langle per_i(w_j(x_l, v_l)).E'_l, s'_l\rangle \xrightarrow{per_i(w_j(x_l,v_l))\,\omega_2\, per_i(w_j(x_m,v_m))} \langle E'_m, s'_m\rangle)) \\
&\quad \wedge \\
&\quad (\forall\, r_i(x_l, v_l) \in S_i \; of \; A^H_{i+w}, \exists\, w_j(x_l, v_l) \in A^H_{i+w}, \\
&\quad \exists \omega_1, \omega_2 \in A^* \bullet \\
&\qquad (\langle w_j(x?, v?).E_j, s_j\rangle \xrightarrow{w_j(x_l,v_l)\,\omega_1\, r_i(x_l,v_l)} \langle E_i, s_i\rangle \Rightarrow \\
&\qquad\quad \langle per_i(w_j(x_l, v_l)).E'_j, s'_j\rangle \xrightarrow{per_i(w_j(x_l,v_l))\,\omega_2\, r_i(x_l,v_l)} \langle E_i, s_i\rangle)))
\end{aligned}
$$

以上より，コーザル・メモリ $\mathcal{CM}$ の記述は，コーザル・メモリ・コンシステンシ・モデルの要求を満たしている.

□

6.2 リリース・メモリの検証

本節では，5.2 節にて記述されたリリース・コンシステンシ・モデルの実現が，4.3 節にて記述されたリリース・コンシステンシ・モデルを満足するかどうか検証する．検証に当っては，まず，証明の対象となっている命令間のトレースを，value-passing CCS を用いた仕様記述に 3.4.3 項の遷移規則を適用して S-CCS を用いて記述する．次に，このプロセス展開と同時に遷移したローカルメモリ，各種キューなどの状態遷移を Z の操作スキーマの記述から求めて検証を行う．この様に，モデルの実現の形式的仕様記述をプロセスの展開と状態遷移に分離して記述しているため，検証においてもプロセスの展開と状態遷移の視点を明確に分けて証明を行うことができる．

[補題 3] $w_j(x_l, v_l)$ を任意のプロセス P_j の任意のストア命令とする．このとき，全てのプロセスは，必ずいつかは $w_j(x_l, v_l)$ をそれぞれのローカルメモリに適用できる．さらに，全てのプロセスで更新されたかどうかを P_j にて確認することができる．

(証明) 任意のプロセス P_i は，自ら発行した $w_i(x_l, v_l)$ に関しては，操作スキーマ $Write_i$ の定義より，$M_i' = M_i \oplus \{x_l \mapsto v_l\}$ にてローカルメモリを更新するため，必ず，ローカルメモリに適用できる．そこで，以下では，任意のプロセス $P_j(j \neq i)$ が発行した $w_j(x_l, v_l)$ を，任意のプロセス P_i が観測する場合について証明する．

プロセス P_j が $w_j(x_l, v_l)$ の更新メッセージ $message_l$ を一次的に蓄積する $OutQueue_j$ は先入先出の有限キューであるため，$broadcast_j$ によって $message_l$ は，いつかは，その他の全てのプロセスに，宛先を $\top$（上限）にして同報通信される．任意のプロセス P_i は $receive_i(message_l)$ を発行して，そのメッセージを $\overline{pipe_i}(message_l)$ と $pipe_i(message_l)$ との τ 操作によって受信する．ここに，value-passing CCS の記述と遷移規則から，以下のようなトレース ω を得ることができる．

$$
\begin{aligned}
&\exists \omega \in A^* \bullet \\
&\quad \langle w_j(x?, v?).E_j, s_j\rangle \xrightarrow{\omega} \langle Receive_i.E_i, s_i\rangle \\
&\quad \wedge\ \omega = w_j(x_l, v_l) \ldots Send_j \ldots \tau \ldots
\end{aligned}
$$

次に，その更新メッセージ $message_l$ は P_i の操作スキーマ $Receive_i$ によって先入先出の有限キューである $InQueue_i$ に追加される．その後，P_i の非決定性より，以下のトレース ω が存在することになる．

$$\exists \omega \in A^* \bullet$$
$$\langle Receive_i.E_i, s_i \rangle \xrightarrow{\omega} \langle Apply_i.E_l, s_l \rangle$$

操作スキーマ $Apply_i$ は，$InQueue_i$ の先頭の要素の宛先が T（上限）であれば，この更新メッセージをローカルメモリに適用しては先頭から削除する．また，操作スキーマ Rel_i は，$InQueue_i$ の先頭の要素の宛先が P_i であれば，この先頭の確認メッセージを取り出しては先頭から削除する．そこで，P_i の非決定性より $Apply_i$, Rel_i を有限回実行して，この受信した宛先が T（上限）である更新メッセージは，ついには，先入先出有限キュー $InQueue_i$ の先頭に来て $Apply_i$ に選択されることになる．そこで，$Apply_i$ は，$w_j(x_l, v_l)$ を，そのローカルメモリに適用することができる．さらに，$Apply_i$ は，この更新メッセージが送られた P_j を宛先にして書き込み確認メッセージ $message_m$ を $OutQueue_i$ に追加する．

$OutQueue_i$ も先入先出の有限キューであるため，$broadcast_i$ によってこのプロセス P_j を宛先にした書き込み確認メッセージ $message_m$ は，いつかは，その他の全てのプロセスに同報通信されることになる．プロセス P_j は $receive_j(message_m)$ を発行して，そのメッセージを $\overline{pipe_j}(message_m)$ と $pipe_j(message_m)$ との τ 操作によって受信する．ここに，value-passing CCS の記述と遷移規則から，以下のようなトレース ω を得ることができる．

$$\exists \omega \in A^* \bullet$$
$$\langle Apply_i.E_l, s_l \rangle \xrightarrow{\omega} \langle Receive_j.E_o, s_o \rangle$$
$$\wedge\ \omega = Apply_i \ldots Send_i \ldots \tau \ldots$$

次に，このプロセス P_j 宛の確認メッセージ $message_m$ は P_j の操作スキーマ $Receive_j$ によって選択され，先入先出の有限キューである $InQueue_j$ に追加される．但し，P_j 宛以外の確認メッセージを受信した場合は，何もしないで，この確認メッセージを破棄する（但し，宛先が T（上限）のメッセージは $InQueue_j$ に追加する）．その後，P_j の非決定性より，以下のトレース ω が存在する．

$$\exists \omega \in A^* \bullet$$
$$\langle Receive_j.E_o, s_o\rangle \xrightarrow{\omega} \langle Rel_j.E_p, s_p\rangle$$

操作スキーマ Rel_j は，$InQueue_j$ の先頭の要素の宛先が P_j であれば，この先頭の確認メッセージを取り出しては先頭から削除する．また，操作スキーマ $Apply_j$ は，$InQueue_j$ の先頭の要素の宛先が $\top$（上限）であれば，この更新メッセージを取り出しては先頭から削除する．

そこで，P_j の非決定性より Rel_j, $Apply_j$ を有限回実行して，この受信した宛先が P_j である確認メッセージは，ついには，先入先出キュー $InQueue_j$ の先頭に来て選択される．そこで，Rel_j は，$w_j(x_l, v_l)$ がプロセス P_i のローカルメモリに適用されたことを確認する．このような確認メッセージは，P_j 以外の全てのプロセスから P_j に送信され，ついには，Rel_j 内の関数 *update_confirmation* にて，全ての他のプロセスから確認メッセージが送られたことを確認することができる．

□

[補題 4] 全てのプロセス P_i において，任意の同期変数 z に関する $S_i \mid z$ 上の全ての獲得命令には，それぞれ異なる解放命令が先行している．但し，先行する解放命令がない場合は，$S_i \mid z$ 上で最初の獲得命令である．

（証明） セマフォエージェント Sem_z は，5.2.3 項の定義より，

$$Sem_z \stackrel{\text{def}}{=} \overline{acq}(z).rel(z).Sem_z$$

Sem_z の Prefix operator は，常に，$\overline{acq}(z)$ と $rel(z)$ のみであり，しかも必ず交互に Prefix operator になるように定義されている．

そこで，今，プロセス P_i の任意の同期変数 z を持つ任意の獲得命令は，下記の定義

$$acq_i(z?) \equiv syn\_a_i(z?).Acq_i.acq(z?).$$

より，入力ポート syn_a_i を通して環境から z を受け取り，操作スキーマ Acq_i を実行する．次に，$acq(z)$ は，制限集合 L に含まれているためセマフォ Sem_z の出力ポート $\overline{acq}(z)$ との τ 操作を通じてのみ展開することができる．

今，セマフォ Sem_z の Prefix operator が，$\overline{acq}(z)$ であれば，遷移規則 Concurrent Composition(2) の τ 操作により内部隠蔽されて，プロセスは次の操作に進む．但し，セマフォ Sem_z の Prefix operator が $\overline{acq}(z)$ となるのは，そのセマフォ Sem_z と内部隠蔽した任意のプロセスの入力ポートが今までないか，または，セマフォ Sem_z の入力ポート $rel(z)$ と任意のプロセスの出力ポート $\overline{rel}(z)$ とが τ 操作を $S_i \mid z$ 上の直前で展開したかのいずれかである．$\overline{rel}(z)$ は，下記のラベル $rel_i(z)$ で発行される．

$$rel_i(z?) \equiv syn\_r_i(z?).check_i(z?).\overline{rel}(z?)$$

従って，以下のようになり，[補題 4] は成立する．

$$
\begin{aligned}
&\forall\, i : 1 \ldots n \bullet \\
&\quad \forall\, z \in Syn, \forall\, acq_i(z) \in S_i \; of \; A^H_{i+w}, \\
&\quad \exists\, j : 1 \ldots n, \exists\, \omega \in A^* \bullet \\
&\qquad ((\exists\, rel_j(z) \in S_j \; of \; A^H_{j+w}, \\
&\qquad rel_j(z), acq_j(z) \notin \omega \bullet \\
&\qquad\quad \langle rel_j(z?).E_l, s_l \rangle \xrightarrow{rel_j(z)\,\omega\, acq_i(z)} \langle E_m, s_m \rangle) \\
&\qquad \vee\, (rel_j(z) \notin \omega \bullet \\
&\qquad\quad \langle start.E_0, s_0 \rangle \xrightarrow{\omega\, acq_i(z)} \langle E_m, s_m \rangle))
\end{aligned}
$$

□

[定理 2] H を リリース・メモリ $\mathcal{RM}$ の実行履歴とすると，H はリリース・コンシステンシ・モデルの条件を満足している．

(証明) 操作スキーマ $Read_i$，$Write_i$，Acq_i，および，Rel_i より，プロセス P_i の逐次実行列 S_i には P_i が発行した全ての命令が含まれる．また，[補題 3] より全ての他のプロセスから発行されたストア命令はプロセス P_i のローカルメモリに適用される．よって，

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

操作スキーマ $Read_i$, $Write_i$，および，$Apply_i$ の記述より，プロセス P_i はローカルメモリ M_i を $M_i' = M_i \oplus \{x? \mapsto v?\}$ によって更新して，$v! = M_i x?$ により報告する．よって，常に，ロード命令はローカルメモリ内に最新のストア命令が書いた値を読みプロセスに通知する．したがって，以下の遷移が存在する:

$$x_l \in \mathcal{M}, v_k, v_l, v_m \in Val, \forall\, i : 1 \ldots n \bullet$$
$$(\forall\, r_i(x_l, v_l) \in S_i \; of \; A^H_{i+w},$$
$$\exists j : 1 \ldots n, \exists \omega \in A^* \bullet$$
$$((\exists\, per_i(w_j(x_l, v_k)) \in S_i \; of \; A^H_{i+w},$$
$$per_i(w_j(x_l, v_m)) \notin \omega \bullet$$
$$\langle per_i(w_j(x_l, v_k)).E_l, s_l\rangle \xrightarrow{per_i(w_j(x_l,v_k))\, \omega\, r_i(x_l,v_l)} \langle E_m, s_m\rangle)$$
$$\vee\, (((\exists\, per_i(w_j(x_l, v_m)) \in S_i \; of \; A^H_{i+w},$$
$$per_i(w_j(x_l, v_m)) \notin \omega \bullet$$
$$\langle start.E_0, s_0\rangle \xrightarrow{\omega\, r_i(x_l,v_l)} \langle E_m, s_m\rangle)$$
$$\vee\, per_i(w_j(x_l, v_m)) \notin S_i \; of \; A^H_{i+w})$$
$$\Rightarrow v_l = \bot)))$$

$per_i(w_j(x_l, v_k))$（操作スキーマ $Apply_i$）の展開による状態 s_l の次の状態 s_l' のローカルメモリ M_i は，値 $x_l \mapsto v_k$ を持つ．状態 s_l' からラベル $r_i(x_l, v_l)$ の展開による状態 s_m までのトレース ω 中には，$per_i(w_j(x_l, v_m))$ は含まれないためローカ

ルメモリ M_i 中のロケーション x_l の値は上書きされることはなく $v_l = v_k(= M_i\, x_l)$ である．よって，逐次実行列 S_i は有効である．

次に，[補題 4] より，各同期変数 z に関する有効な逐次実行列 $S_i \mid z$ が存在する．この任意の獲得命令 $acq_i(z)$ に引き続くプロセス P_i が発行する任意の命令は，[補題 4] のように $rel_j(z)$ との同期が完了するまでは，発行されない．よって，

$$
\begin{array}{l}
\forall\, i : 1 \ldots n \bullet \forall\, z \in Syn \bullet \\
\quad \forall\, acq_i(z) \in S_i \; of \; A^H_{i+w}, \\
\quad \exists\, j : 1 \ldots n, \exists\, \omega \in A^* \bullet \\
\quad\quad (((\exists\, rel_j(z) \in S_j \; of \; A^H_{j+w}, \\
\quad\quad rel_j(z), acq_j(z) \notin \omega \bullet \\
\quad\quad\quad \langle rel_j(z?).E_l, s_l\rangle \overset{rel_j(z)\,\omega\, acq_i(z)}{\longrightarrow} \langle E_m, s_m\rangle) \\
\quad\quad \vee\, (rel_j(z) \notin \omega \bullet \\
\quad\quad\quad \langle start.E_0, s_0\rangle \overset{\omega\; acq_i(z)}{\longrightarrow} \langle E_m, s_m\rangle)) \\
\quad\quad \wedge \\
\quad\quad (\forall\, o_i \in L_i, \forall\, k : 1 \ldots n, \exists\, \omega_1, \omega_2 \in A^* \bullet \\
\quad\quad\quad (\langle acq_i(z?).E_l, s_l\rangle \overset{acq_i(z)\,\omega_1\; o_i}{\longrightarrow} \langle E_m, s_m\rangle \Rightarrow \\
\quad\quad\quad\quad \langle acq_i(z?).E_l, s_l\rangle \overset{acq_i(z)\,\omega_2\; per_k(o_i)}{\longrightarrow} \langle E_n, s_n\rangle)))
\end{array}
$$

次に，$rel_i(z)$ より先行してプロセス P_i で発行された任意の命令は，逐次実行列 S_i を対象として，

$$
\begin{array}{l}
z \in Syn, \forall\, rel_i(z), \forall\, o_i \in L_i, \forall\, j : 1 \ldots n, \exists\, \omega_1, \omega_2 \in A^* \bullet \\
\quad (\langle E_l, s_l\rangle \overset{o_i\,\omega_1\; rel_i(z)}{\longrightarrow} \langle E_n, s_n\rangle \Rightarrow \\
\quad\quad \langle per_j(o_i).E_m, s_m\rangle \overset{per_j(o_i)\,\omega_2\; rel_i(z)}{\longrightarrow} \langle E_n, s_n\rangle)
\end{array}
$$

が，成立するかどうか検証する．

まず，$j = i$ の時は，プロセス P_i で発行された命令に関しては，$acq_i(z)$ と $rel_j(z)$ との同期が取られるまで待たされる以外は，実行履歴と同様の実行順序で，逐次実行される．そこで，残りの証明では，$j \neq i$ とする．

プロセス P_i で発行された命令の内，プロセス P_i 以外の任意のプロセス P_j で観測される命令は，ストア命令だけである．そこで，任意のプロセス P_i で発行さ

れた任意の $rel_i(z)$ より先行して発行された任意のストア命令を $w_i(x_l, v_l)$ とすると，これは，[補題 3] より，他の全てのプロセスで観測されて P_i の下記の $rel_i(z)$ にて確認される．

$$
\begin{aligned}
rel_i(z?) &\equiv syn\_r_i(z?).check_i(z?).\overline{rel}(z?) \\
check_i(z?) &\stackrel{\text{def}}{=} Rel_i.confirm_i(update!). \\
&\quad ((if \quad update! = true\ then\ \mathbf{0}) + \\
&\quad (if \quad update! = false\ then \\
&\quad receive_i(message?).check_i(z?)))
\end{aligned}
$$

上記の記述より，全ての確認が終了して初めて $rel_i(z)$ は終了する．よって，他のプロセスのこのストア命令の観測は，この $rel_i(z)$ より先行して終了している．

以上より，[定理 2] は証明された．

□

7. 関連研究

メモリ・コンシステンシ・モデルを統一的に形式化する研究については，Adve [5], Ahamad [6], Kohli [7] などがある．これらは，ある視点を設定して，代数を用いて異なるモデル間の比較を目的として形式化がされている．但し，そこで定義されているプログラム順序など関係の定義は自然言語で定義されており，非形式的な記述に留まっている．また，実現に関して統一的な手法で記述・検証まで触れている研究は見あたらない．

そこで，本章では，形式手法に関する関連研究について述べる．形式手法の関連研究として仕様記述言語 Z または Object-Z とプロセス代数 CCS または CSP を結合した他研究との比較を行う．これらの研究に対しては，Fischer [34] による比較研究があるので，その成果を参照しつつ比較を行うことにする．

7.1 Z + CCS

Z と CCS を結合した形式的手法の提案としては，他には Galloway と Stoddart [35] による研究がある．彼らの提案手法と我々が本論文において用いている Z と CCS を結合した形式的手法の大きな違いは，彼らは Z を value-passing CCS の値計算システムとして用いているのに対して，我々の手法は， CCS のアクションとして Z の操作スキーマを用いる点である．value-passing CCS では，出力ポートにおいて関数記号を用いて計算が行われる．この出力ポートにおける関数の代りに Z の操作スキーマを用いるのが Galloway らによる手法である．

Galloway らの仕様記述方法は CCS を用いて仕様記述する立場から見た場合，Z は単にデータの宣言と，データ処理の形式的理論として採用されただけであり，CCS の仕様を変更する必要はまったく無い．それに対して，我々の仕様記述方法は CCS 本来の仕様記述方法から離れており，Z の仕様記述に対して，その制御を CCS の演算子を用いて行う，という大きな違いがある．しかし，これらは観点の違いであり，表現能力においては，基になる理論が同一であるので，同等であると言える．

本研究の特徴は，与えられた仕様により定義されるラベル付き状態遷移システ

ム上の実行遷移関係を用いて検証を行っている点である．CCS における検証方法には，CCS のラベル付き状態遷移システム上の様相論理（様相 μ 計算）を性質の記述に用い，その式を検証する方式が知られている．検証方式としては，モデルチェッキング [36, 37] を用いるものが主であり，検証ツールとしては Concurrency Workbench が開発されている．現在，両手法ともこれらの既存のツールを用いて検証する手法を確立しておらず，今後の研究課題である．

7.2 (Object-)Z + CSP

Z（もしくは Object-Z）と CSP を結合した形式的手法の研究には，Fischer による CSP と Object-Z の結合である CSP-OZ [38] や Z と CSP の結合である CSP-Z [39]，Smith [40]，および，Mahony と Dong による Object-Z と CSP をリアルタイムに拡張した Timed-CSP との結合による TCOZ [41] などが知られている．Fischer と Smith は，Object-Z のクラスを CSP の意味論を用いて解釈することにより，結合を行っている．それに対して，TCOZ は本論文で採用した方式と同様な方法を採用している．すなわち，CSP におけるイベント（CCS におけるアクション）として Z の操作スキーマを用いている．以下，検証方法，イベントの解釈，プロセスのブロック方式に関して，Z + CSP と (Object-)Z + CSP の比較を行う．

7.2.1 (Object-)Z + CSP の検証方法

Concurrency Workbench が CCS の検証ツールであったように，CSP には FDR という検証ツールが開発されている．Fischer [34] が指摘しているように，CSP + Z (Object-Z) の組み合わせは，ツールの再利用が容易であり，FDR を用いた CSP-Z の仕様の検証が行われている [42]．また，リファインメント計算を用いて仕様の詳細化も可能である [43]．これは，Object-Z のクラスの解釈を CSP の意味論を使うという枠組みにより，CSP におけるリファインメント計算を用いることができる，という利点による．これらの点から，現在のところ Z + CCS の組み合わせより，(Object-)Z + CSP の組み合わせの方が，検証に関しては一

歩進んでいる状況である．

7.2.2 単一イベント vs 多重イベント

Fischer [34] は，Zとプロセス代数との組み合わせにおいて，Zの操作がプロセス代数においてどのような粒度で用いられているかで分類している．(Object-)Z + CSP においては，Z の操作を CSP におけるイベントとして解釈する点において単一イベント型と言える．それに対して，Z + CCS と TCOZ においては，入出力アクションと Z の操作は別々に解釈されている．尚，TCOZ が他の (Object-)Z + CSP と異なるのは，時間の概念を入れることにより，入出力イベントと操作を区別する必要性があったためであると考えられる．

7.2.3 プロセスのブロック方式

Fischer [34] は，さらにガードの有無について分析している．$\{CSP, CCS\} \times \{Z, Object\text{-}Z\}$ の組み合わせにおいては，ほとんどの方式が Z における前条件をガードとして用いている．これは，Z を用いて仕様記述する場合には，当然の方法であるが，並列システムの記述においては，プロセス代数のレベルでの仕様記述において明示的に示す必要がある．我々が本論文で採用した仕様記述方式も，同じ問題点を持っており，この問題を解決するためには，何らかの構文的な拡張が必要になる．

7.3 まとめ

上記のように $\{CSP, CCS\} \times \{Z, Object\text{-}Z\}$ の組み合わせにおいては，様々な提案がなされており，意味論的基礎づけ，長所や短所の分析もほぼ終了している段階であると言える．

我々の採用している手法においても，プロセスのブロック方式の導入を行い，プロセス代数レベルでガードの記述を行い，ブロック内の入力に関する同期を明示する必要がある．この記述に当って，どのような新しい構文を採用するか，ま

た，現在の記述方式から，新しい構文へ変換が容易であるか，といった点については，今後の研究課題としたい．

8. 結論

本章では，本研究で得られた成果をまとめ，今後の課題について述べる．

8.1 本研究で得られた成果

本研究で得られた成果は以下の通りである．

メモリ・コンシステンシ・モデルの形式的仕様記述

本論文では，メモリ・コンシステンシ・モデルを二分する代表的なモデルとしてコーザル・メモリ・コンシステンシ・モデル [12, 6] とリリース・コンシステンシ・モデル [14, 18] について田口と荒木が提案した形式手法を用いて形式的に記述した．

この形式手法で提案された「状態遷移に基づく CCS 意味論」を用いて記述された式をトレースに自然に拡張することで，分散共有メモリの逐次実行列を表現した．

コーザル・メモリ・コンシステンシ・モデルでは，このトレースにおいて，ロード命令とストア命令の間のプログラム順序関係や共有変数の値との関係を記述することでこのモデルを定義できた．

リリース・コンシステンシ・モデルでは，このトレースにおいて，同期命令，ロード命令，および，ストア命令などのプログラム順序関係を記述することでこのモデルを定義できた．但し，リリース・コンシステンシ・モデルにおける異なる同期変数を持つ獲得命令は，解放命令を飛び越すことがある，という条件は本論文では省略した．今後の課題としたい．

分散共有メモリシステムの実現の形式的仕様記述

同様に，メモリ・コンシステンシ・モデルを二分する代表的なモデルとしてコーザル・メモリ・コンシステンシ・モデル [12, 6] とリリース・コンシステンシ・モデル [14, 18] をもつ分散共有メモリの実現について，それぞれに，田口と荒木が提案した同様の形式手法を用いて形式的に仕様記述した．そして，それぞれの実

行トレースを遷移規則による遷移例として示した．

分散共有メモリの記述において，ロード命令，ストア命令，同期命令，および観測命令などによるローカル・メモリの更新・参照および他のプロセスとの通信に使用される入出力待ち行列の更新・参照，および，確認情報の更新・参照などに関する操作や状態遷移は Z の状態スキーマおよび操作スキーマを用いて記述した．また，非決定的な操作の並列性やプロセス間の通信やセマフォによる同期などは value-passing CCS を用いて，分離して記述した．

特に，コーザル・メモリ・コンシステンシ・モデルの実現では，プログラムの先行関係に弱ベクトル時計を採用して，Z を用いて記述することで，入力待ち行列内で因果先行順に並べると共に，操作スキーマ $Apply_i$ において，その入力待ち行列の内からそのプロセスに対する最も小さい因果先行関係を満足するものを選択するように記述することができた．また，検証においては実現にて記述された弱ベクトル時計の大小関係を基にして証明した．

また，リリース・コンシステンシ・モデルの実現では，関数で表現した確認情報の状態遷移や同期変数の状態遷移などを Z を用いて，同期変数に対応する複数のセマフォなどは，value-passing CCS を用いて，分離して記述することができた．また，全て確認情報が返答されたかどうかを確認する操作のラベル Rel_i の動作は，value-passing CCS を用いて，操作スキーマ Rel_i のループ構造を記述した．

分散共有メモリシステムの実現の検証

同様に，コーザル・メモリ・コンシステンシ・モデル [12, 6] とリリース・コンシステンシ・モデル [14, 18] について，それぞれの分散共有メモリシステムが，それぞれのメモリ・コンシステンシ・モデルを満足しているかどうか検証した．

8.2 今後の課題

今後の課題としては，次の 2 点が挙げられる．まず，二つの代表的なモデルでこの形式手法で記述・検証を行い，その他のモデルについても同様の形式手法で記述・検証できる見通しはできたが，実際に，第 2 章で紹介したその他のモデル

について記述・検証をすることがある．次に，これらの経験を踏まえて，この形式手法を拡張することなどが挙げられる．

例えば，リリース・コンシステンシ・モデルの操作スキーマ $Apply_i$ において，ストア命令の更新確認メッセージは，本来は，そのストア命令を発行したプロセスにのみ送ればいいのであるが，Z の操作スキーマ内では次の操作選択を記述することが出来ないため，全てのプロセッサへの同報通信の記述とした．これは value-passing CCS の Conditional 構文を用いて，全てのプロセッサに場合分けをすれば，個別にメッセージを送るように記述できるが，仕様記述としては非常に煩雑なものになる．このように Z の操作スキーマの実行後の次の操作に場合分けが多い時に柔軟に記述できるように，Z と value-passing CCS との統合した操作意味論を拡張することなどである．また，7.2 節で述べたように，プロセスのブロック方式の導入，および，モデルチェッキングなどの検証の自動化なども挙げられる．

謝辞

本研究を進めるにあたり，終始温かい御指導を賜わりました奈良先端科学技術大学院大学情報科学研究科の福田晃教授に心より感謝の意を表します．また，本論文をまとめるにあたり，奈良先端科学技術大学院大学情報科学研究科の鳥居宏次教授ならびに関浩之教授の貴重なる御助言と御指導を賜りましたことをここに述べ，両教授に深く感謝の意を表します．

九州大学大学院システム情報科学研究科の荒木啓二郎教授には，著者在学の間，福田教授とともに御指導を賜り，また対外発表の機会とその際の御支援を頂きましたことをここに述べ，心より感謝の意を表します．

和歌山大学システム工学部の城和貴助教授には，本研究全般に渡って御支援ならびに御指導を賜り，またしばしば本研究の細部にまで及ぶ議論に貴重な時間を割いて頂きました．また城助教授には内外の諸分野の研究者との議論の機会を多く設けて頂き，これにより諸先生方の様々の視点からの助言が得られ，また著者の知見を広めることができましたことをここに述べ，心より感謝の意を表します．

九州大学大学院システム情報科学研究科の田口研治助手には，本研究に関する基本的なアイデアである Z + CCS やその他の形式手法の統合に関する関連研究の情報を提供して頂き，また，議論のみならず論文の校正にまで貴重な時間を割いて頂きましたことをここに述べ，心より感謝の意を表します．

奈良先端科学技術大学院大学情報科学研究科の渡邊勝正教授には，議論のみならず論文の校正に貴重な助言を頂きましたことをここに述べ，心より感謝の意を表します．

奈良先端科学技術大学院大学情報科学研究科の中西恒夫助手には，共有メモリシステムに関して貴重な助言を頂きましたことをここに述べ，心より感謝の意を表します．

財団法人九州システム情報技術研究所/奈良先端科学技術大学院大学情報科学研究科の張漢明氏には，Z の記述に関して貴重な助言を頂きましたことをここに述べ，心より感謝の意を表します．

最後に今日まで筆者の研究活動に対する理解と協力を頂いた株式会社けいはんなの江川寿夫副社長，足立一郎元代表専務取締役，吉田正行前代表専務取締役，

田中宏信取締役，鈴木孝雄元取締役，田敏機前取締役，柳澤卓取締役，加藤裕勝部長，福西国男次長，松浦輝昭元課長，岡本圭司前課長，沢尾俊和課長，内藤貴江さん，小倉潤子さん，藤原有希子さん，酒井美穂さんおよび妻千春に感謝します．

参考文献

[1] Leach, P. J., Levin, P. H., Douros, B. P., Hamilton, J. A., Nelson, D. L., and Stumpf, B. L.: The Architecture of an Integrated Local Network, *J. Selected Areas in Communications*, Vol. SAC-1, No. 5, pp. 842–856 (1983).

[2] 城和貴: メモリ・コンシステンシ・モデルの諸定義と解釈例, 並列処理シンポジウム JSPP'97（チュートリアル講演）, 情報処理学会, pp. 149–163 (1997).

[3] 福田晃: 並列オペレーティングシステム, 並列処理シリーズ, Vol.7 , コロナ社 (1997).

[4] Coulouris, G., Dollimore, J., and Kindberg, T.: *Distributed Systems*, Addison-Wesley, second edition (1994).

[5] Adve, S. V. and Hill, M. D.: A Unified Formalization of Four Shared-Memory Models, *IEEE Trans. on Paralled and Distributed Systems*, Vol. 4, No. 6, pp. 613–624 (1993).

[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] Kohli, P., Neiger, G., and Ahamad, M.: A Characterization of Scalable Shared Memories, Technical Report GIT-CC-93-04, College of Computing, Georgia Institute of Tecnology (1993).

[8] Spivey, J.: *The Z Notation*, Prentice Hall, second edition (1992).

[9] Milner, R.: *Communication and Concurrency*, Prentice Hall (1989).

[10] 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*, pp. 283–292 (1997).

[11] 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).

[12] 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).

[13] Goodman, J. R.: Cache Consistency and Sequential Consistency, Technical Report No. 61, IEEE Scalable Coherent Interface Working Group (1989).

[14] Gharachorloo, K., Lenoski, D., Laudon, J., Gibbson, P., Gupta, A., and Hennessy, J.: Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors, *Proc. of the 17th Ann. Int'l Symp. Computer Architecture*, pp. 15–26 (1990).

[15] Lipton, R. J. and Sandberg, J. S.: PRAM:A Scalable Shared Memory, Technical Report CS-TR-180-88, Dept. of Computer Science, Princeton University (1989).

[16] Dubois, M., Scheurich, C., and Briggs, F. A.: Memory Access Buffering in Multiprocessors, *Proc. of the 13th Ann. Int'l Symp. Computer Architecture*, pp. 434–442 (1986).

[17] Carter, J. B., Bennett, J. K., and Zwaenepoel, W.: Implementation and Performance of Munin, *Proc. of the 13th Symp. Operating Systems Principles*, pp. 152–164 (1991).

[18] Keleher, P., Cox, A. L., and Zwaenepoel, W.: Lazy Release Consistency for Software Distributed Shared Memory, *Proc. of the 19th Ann. Int'l Symp. Computer Architecture*, pp. 13–21 (1992).

[19] Bershad, B. N. and Zekauskas, M. J.: Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors, Technical Report CMU-CS-91-170, Carnegie-Mellon University (1991).

[20] Wordsworth, J. B.: *Software Development with Z: A Practical Approach to Formal Methods in Software Engineering*, Addison-Wesley (1992).

[21] Potter, B., Sinclair, J., and Till, D.: *An Introduction to Formal Specification and Z*, Prentice Hall (1991).

[22] Diller, A.: *Z: An Introduction to Formal Methods*, John Wkley & Sons (1990).

[23] Hayes, I. J.(ed.): *Specification Case Studies*, Prentice Hall, second edition (1993).

[24] 張漢明: Z における仕様記述変換に関する研究 ～変換によるモデル形成支援～, 修士論文 NAIST-IS-MT351066, 奈良先端科学技術大学院大学 (1995).

[25] Milner, R.: Operational and Algebraic Semantics of Concurrent Processes, *Formal Models and Semantics*, Handbook of Theoretical Computer Science, Vol. B, The MIT Press, pp. 1201–1242 (1996).

[26] Stirling, C.: Modal and Temporal Logic for Processes, *Logic for Concurrency*, Lecture Notes in Computer Science, Vol. 1043, Springer-Verlag, pp. 149–237 (1996).

[27] Takata, S., Taguchi, K., Joe, K., and Fukuda, A.: Specification and Verification of Memory Consistency Models for Shared-Memory Multiprocessor Systems, *Proc. of Int'l Conf. on Parallel and Distributed Processing Techniques and Applications*, Vol. 2, pp. 923–930 (1998).

[28] Takata, S., Taguchi, K., Joe, K., and Fukuda, A.: Specification and Verification of Memory Consistency Models for Shared-Memory Multiprocessor Systems, 情報処理学会論文誌：数理モデル化と応用, Vol. 40, No. SIG2, pp. 33–44 (1999).

[29] 高田司郎, 田口研治, 城和貴, 福田晃: リリース・コンシステンシ・モデルとその実現の 形式的仕様記述について, 情報処理学会論文誌：数理モデル化と応用 (1999). 掲載予定.

[30] Herlihy, M. P. and Wing, J. M.: Linearizability: A Correctness Condition for Concurrent Objects, *ACM Trans. on Programming Languages and Systems*, Vol. 12, No. 3, pp. 463–492 (1990).

[31] Misra, J.: Axioms for Memory Access in Asynchronous Hardware Systems, *ACM Trans. on Programming Languages and Systems*, Vol. 8, No. 1, pp. 142–153 (1986).

[32] 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).

[33] Marzullo, K. and Neiger, G.: Detection of Global State Predicates, Technical Report 91/39, College of Computing, Georgia Institute of Tecnology (1991).

[34] Fischer, C.: How to Combine Z with a Process Algebra, *Proc. of Int. Conf. of Z Users '98 The Z Formal Specification Notation*, Lecture Notes in Computer Science, Vol. 1493, pp. 5–23 (1998).

[35] Galloway, A. J. and Stoddart, W. J.: An Operational Semantics for ZCCS, *Proc. of the 1st Int'l Conf. of Formal Engineering Methods*, pp. 272–282 (1997).

[36] McMillan, K. L.: *Symbolic Model Checking: An Approach to the State Explosion Problem*, PhD Thesis, Carnegie Mellon University (1992).

[37] Clarke, E. M., Grumberg, O., Hiraishi, H., Jha, S., Long, D. E., and McMillan, K. L.: Verification of the Futurebus+ Cache Coherence Protocol, *For-*

mal Methods in System Design, Kluwer Academic Publishers, pp. 217–232 (1995).

[38] Fischer, C.: CSP-OZ: A Combination of Object-Z and CSP, *Proc. of Int. Conf. on Formal Methods for Open Object-based Distributed Systems '97*, Vol. 2, Chapman and Hall, pp. 423–438 (1997).

[39] Fischer, C.: Combining CSP and Object-Z, Technical Report TRCF-97-1, University of Oldenburg (1997).

[40] Smith, G.: A Semantic Integration of Object-Z and CSP for the Specification of Concurrent System, *Formal Methods Europe'97*, Lecture Notes in Computer Science, Vol. 1313, pp. 62–81 (1997).

[41] Mahony, B. and Dong, J.-S.: Blending Object-Z and Timed CSP: An introduction to TCOZ, *Proc. of Int. Conf. on Software Engineering '98*, pp. 95–104 (1998).

[42] Mota, A. and Sampaio, A.: Model-Checking CSP-Z, *Proc. of the European Joint Conference on Theory and Practice of Software*, Lecture Notes in Computer Science, Vol. 1382, pp. 205–220 (1998).

[43] Smith, G. and Derrick, J.: Refinement and Verification of Concurrent Systems Specified in Object-Z and CSP, *Proc. of the 1st Int'l Conf. of Formal Engineering Methods*, pp. 293–302 (1997).

付録

A. Zの構文と演算子の概要

ここでは，Zの構文と演算子の簡単な説明を示す．厳密な定義は，文献 [8] を参照されたい．

スキーマ定義 (Schema box)

Name[パラメータ]
宣言部

述語部

大域的変数定義 (Axiomatic description)

宣言部

述語部

パラメータによる定義 (Generic definition)

[パラメータ]
宣言部

述語部

関係 (Relations)

$X \leftrightarrow Y$	2 項関係 (*Binary relations*)
$x \mapsto y$	対応 (*Maplet*)
$\mathrm{dom}\, R$	定義域 (*Domain*)
$\mathrm{ran}\, R$	値域 (*Range*)
$S \lhd R$	定義域制限 (*Domain restriction*)
$S \lhd\!\!\!- R$	定義域反制限 ($Domain\ anti - restriction$)
$R(\!\mid S \mid\!)$	関係の像 (*Relational image*)
$Q \frown R$	列の連接
$Q \oplus R$	オーバライド (*Overriding*)

関数 (Functions)

$f(x)$	関数適用 (*Function application*)
$X \nrightarrow Y$	部分関数 (*Partial functions*)
$X \rightarrow Y$	全域関数 (*Total functions*)
$X \rightarrowtail Y$	単射 (*Total injections*)
λ	ラムダ式

基本式 (Basic expressions)

$x = y$	等式 (*Equality*)
$x \neq y$	不等式 (*Inequality*)
θS	シータ式 ($Theta - expression$)
$E.x$	選択 (*Selection*)

集合 (Sets)

$x \in S$	集合に属する (*Membership*)
$x \notin S$	集合に属さない ($Non-membership$)
$\{x_1, ..., x_n\}$	集合の列挙 (*Set display*)
$\{x : T \mid P \bullet E\}$	集合の内包表現 (*Set comprehension*)
$\emptyset$	空集合 (*Empty set*)
$\mathbb{P}\, S$	巾集合 (*Power set*)
$S \times T$	直積 (*Cartesian product*)
(x, y, z)	組 (*Tuple*)
$S \cup T$	和集合 (*Set union*)
$S \cap T$	積集合 (*Set intersection*)
$S \setminus T$	差集合 (*Set difference*)
$\bigcup A$	A の和集合 (*Generalized union*)

数と算術 (Numbers and arithmetic)

$\mathbb{N}$	自然数 (*Natural numbers*)
$\mathbb{N}_1$	正整数
$\mathbb{Z}$	整数 (*Integers*)
$+ \; - \;$ div mod	算術演算子 (*Arithmetic operations*)
$< \leq \geq >$	算術比較演算子 (*Arithmetic comparisons*)
$m \mathinner{..} n$	数の領域 (*Number range*)
$\#S$	集合の大きさ (*Size of a set*)

基本型宣言 (Basic type definition) の例

$[NAME, DATE]$

略記定義 (Abbreviation definition) の例

$DOC == \mathbb{N}$

自由型定義 (Free type definition) の例

$Ans ::= ok\langle\langle \mathbb{Z} \rangle\rangle \mid error$

論理，スキーマ計算 (Logic and schema calculus)

$true, false$	論理定数 (*Logical constants*)
$\neg P$	否定 (*Negation*)
$P \wedge Q$	連言 (*Conjunction*)
$P \vee Q$	宣言 (*Disjunction*)
$P \Rightarrow Q$	含意 (*Implication*)
$P \Leftrightarrow Q$	同値 (*Equivalence*)
$\forall x : T \mid P \bullet Q$	全称記号 (*Universal quantifier*)
$\exists x : T \mid P \bullet Q$	存在記号 (*Existential quantifier*)

特別スキーマ演算子 (Special schema operators)

$S[y_1/x_1, y_2/x_2]$	変数の名前換え (*Renaming*)
$S\ (x_1, x_2)$	変数の隠蔽 (*Hiding*)
$Op1 \gg Op2$	パイプ (*Piping*)

B. CCS の構文と演算子の概要

ここでは，CCS の構文と演算子の簡単な説明を示す．厳密な定義は，文献 [9] を参照されたい．

Set Constructions

$\tilde{z}$	*indexed set* $\{z_i : i \in I\}$ *(I understood)*

Basic Agent Constructions

$\alpha.E$	*Prefix*
$\mathbf{0}$	*inactive agent*
$E + F$	*Summation*
$\sum_{i \in I} E_i$	*Summation over indexing set*
$E \mid F$	*Composition*
$\prod_{i \in I} E_i$	*Composition over indexing set*
$E \setminus L$	*Restriction*
$E[f]$	*Relabelling*
$\{\tilde{E}/\tilde{X}\}$	*simultaneous substitution*
$\mathbf{fix}(\tilde{X} = \tilde{E})$	*Recursion*

Value-passing Agent Constructions

$a(x).E$	*Prefix(input of values)*
$\overline{a}(e).E$	*Prefix(output of values)*
if b **then** E	*Conditional*
$A(\tilde{x}) \stackrel{\text{def}}{=} E$	*parametric agent definition*
$\hat{E}$	*translation to basic calculus*

Agent Equivalence relations

$E \equiv F$	*syntactic identity*

C. 著者研究業績

本論文に関連する研究業績

学術論文

[1] 高田 司郎, 田口 研治, 城 和貴, 福田 晃: リリース・コンシステンシ・モデルとその実現の形式的仕様記述について, 情報処理学会論文誌：数理モデル化と応用, 採録決定（1999 年 9 月掲載予定）, 本論文リリース・コンシステンシ・モデルに関連する内容.

[2] Takata, S., Taguchi, K., Joe, K., and Fukuda, A.: Specification and Verification of Memory Consistency Models for Shared-Memory Multiprocessor Systems, 情報処理学会論文誌：数理モデル化と応用, Vol. 40, No. SIG2, pp. 33–44(1999), 本論文コーザル・メモリ・コンシステンシ・モデルに関連する内容.

国際会議

[1] Takata, S., Taguchi, K., Joe, K., and Fukuda, A.: Specification and Verification of Memory Consistency Models for Shared-Memory Multiprocessor Systems, *Proc. of Int'l Conf. on Parallel and Distributed Processing Techniques and Applications*, Vol. II, pp. 923–930(1998), 本論文コーザル・メモリ・コンシステンシ・モデルに関連する内容.

口頭発表

[1] 高田 司郎, 田口 研治, 城 和貴, 福田 晃: リリース・コンシステンシ・モデルとその実現の形式的仕様記述について, 情報処理学会数理モデル化と問題解決研究会, MPS-22-14, pp. 81–86 (1998), 本論文リリース・コンシステンシ・モデルに関連する内容.

[2] Takata, S., Taguchi, K., Joe, K., and Fukuda, A.: Specification and Veri-

fication of Memory Consistency Models for Shared-Memory Multiprocessor Systems, 情報処理学会数理モデル化と問題解決研究会, MPS-20-5, pp. 25–30 (1998), 本論文コーザル・メモリ・コンシステンシ・モデルに関連する内容.

その他の研究業績

著書

[1] 丸尾雅一, 高田司郎, 中村弘成: 第 5 部 システムの効率化と管理 ソフトウェアの生産性向上策, 実例コンピュータバンキング, No. 9, 石崎純夫編集, 近代セールス社, pp. 174–192 (1983).

招待講演

[1] 知識情報システム構築支援ツール XPT-II, 土木学会関西支部 (1989).

[2] 知識情報システム構築支援ツール XPT-II–オブジェクト指向の視点から–, 情報処理学会関西支部ソフトウェア研究会 (1989).

口頭発表

[1] 高田 司郎: 自動並列化コンパイラの正当性と性能評価への形式的仕様記述言語の適用について, 日本ソフトウェア科学会第 11 回大会論文集, pp. 349–352 (1994).

[2] 高田 司郎: 継続時間の扱いを考慮した並列性の形式化について, 情報処理学会ソフトウェア工学研究会, SE-99-4, pp. 25–32 (1994).

[3] 朴 炳植, 小田孝和, 高田司郎, 鈴木　胖: 異分野研究のための知的オリエンテーション・データベースシステムの構築, システム制御情報学会第 38 回研究発表講演会, pp. 247–248 (1994).

[4] 高田 司郎, 下地 俊一, 藤本 久志, 田口 研治, 山本 博史: エキスパートシステム構築支援ツール XPT(1)–概要–, 情報処理学会第 35 回全国大会, 1N-4 (1987).

[5] 田口 研治, 土居 誉生, 高田 司郎: エキスパートシステム構築支援ツール XPT(2)–推論方式–, 情報処理学会第 35 回全国大会, 1N-5 (1987).

[6] 杉本直樹, 武田博隆, 高田司郎, 松浦敏雄, 吉岡信夫: X.Toolkit 上でのアプリケーションプログラム作成支援ツール Weiss-XT, 情報処理学会第 39 回全国大会, 7N-3, pp. 1151–1152 (1987).

[7] 花崎賢一, 松本一夫, 土居誉生, 高田司郎, 上原邦昭, 豊田順一: PROLOG 処理系におけるコンパイル方式の改良, 日本ソフトウェア科学会第 4 回大会論文集, A-1-1, pp. 23–26 (1987).

[8] 高田 司郎, 米山 寛二, 野田 昭司, 鳥居 宏次: 実行時メッセージに基づくデバッグモデルとその適用例, 情報処理学会第 33 回全国大会, 4G-8 (1986).

[9] 的場 裕司, 工藤 英男, 吉岡 信夫, 高田 司郎: プログラミング教育用疑似計算機システムについて, 情報処理学会第 20 回全国大会, 4H-2 (1979).

雑誌

[1] 高田 司郎 他: 知識情報処理システム構築支援ツール「XPT-II」, CSK 技術通信, No. 20, pp. 10–21 (1990).

[2] 瀧本 幸雄, 高田 司郎: オブジェクト指向に基づく上流から下流まで一貫支援した CASE について, CSK 技術通信, No. 20, pp. 72–74 (1990).

[3] 白石 滋美, 米山 寛二, 岡本 憲治, 高田 司郎: 知識処理と情報処理の融合による知的 OA システム, オフィス・オートメーション, Vol. 8, No. 1, pp. 33–38 (1987).

[4] 高田司郎: 知識を利用したプログラム自動生成について, CSK 技術通信, No. 10, pp. 14–18 (1985).