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

高田司郎[†1,†2] 田口研治[†3,☆]
城和貴[†4,☆☆] 福田晃[†1]

我々は，形式的仕様記述言語 Z の表記法とプロセス代数 value-passing CCS を統合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検証の研究を行っている．メモリ・コンシステンシ・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．我々は，以前，前者の代表としてコーザル・メモリ・コンシステンシ・モデルを取り上げ，このモデルと実現の形式的な仕様記述と検証に対するこの形式手法の有効性を報告した．そこで，本論文では，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，このモデルと実現の形式的な仕様記述と検証を行なう．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．

A Formal Specification of a Release Consistency Model and Its Implementation

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

We study on the formal specification and verification of the memory consistency models and their implementations that define the behavior of multiple memory accesses on distributed shared memory systems, using a formal method that combines the Z notation and value-passing CCS. Memory consistency 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 a separate paper, we applied this formal method to formally specify and verify "Causal Memory Consistency Model" that is a representative of the former group mentioned above. We showed in the paper that this formal method is feasible enough for the formal specification and verification of that type of memory consistency model. In this paper, we apply the same formal method to formally specify and verify "Release Consistency Model" that is a representative of the latter group. We conclude that this formal method is feasible enough for the formal specification and verification of both types of memory consistency models.

1. はじめに

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

†1 奈良先端科学技術大学院大学
Nara Institute of Science and Technology
†2 けいはんな
Keihanna Interaction Plaza Inc.
†3 九州大学大学院 システム情報科学研究科
Graduate School of Information Science and Electrical Engineering, Kyushu University
☆ 現在，筑紫女学園大学アジア文化学科
Presently with Chikushi Jogakuen University, Department of Asian Studies
†4 和歌山大学システム工学部
Faculty of Systems Engineering, Wakayama University
☆☆ 現在，奈良女子大学理学部情報科学科
Presently with Faculty of Science, Nara Women's University

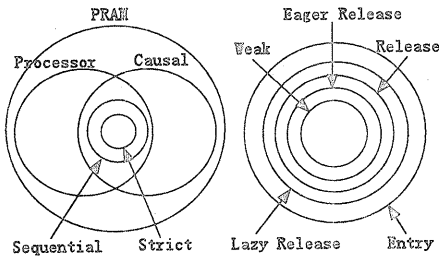

図1 メモリ・コンシステンシ・モデルの関係
Fig. 1 Relations of memory consistency models

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．

DSMの実現では，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[1)~3)]．

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．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，図1のように，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[1)]．

我々は，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[4),5)]を取り上げ，田口と荒木が提案したZの表記法[6)]とvalue-passing CCS[7)]（Calculus of Communicating Systems）を統合した形式手法[8)]をこのモデルに適用した結果，そのモデルと実現の記述，および，その実現の検証がこの形式技法で一貫して行えることを示した[9),10)]．そこで，本論文では，後者の代表としてリリース・コンシステンシ・モデル[11),12)]を取り上げ，このモデルに同様の形式手法を適用してこのモデルと実現の形式的な仕様記述と検証を行ない，この形式手法の有効性を確認する．また，コーザル・メモリ・コンシステンシ・モデルでは，弱ベクトル時計を採用して，因果先行関係などのプログラム順序の記述，および，その検証を行った[9),10)]．本論文のリリース・コンシステンシ・モデルでは，セマフォを採用して，同期処理の記述，および，その記述の検証を行なう．

ここで，本論文で用いる形式手法について簡単に触れる．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[7)]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[8)]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，2節では，本論文で用いる形式手法の操作的意味論S-CCSと遷移規則について概説する．3節では，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，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節では，まず，リリース・コンシステンシ・モデルの実現について，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を用いて分離して記述する．次に，これら仕様記述にS-CCSの遷移規則を適用した展開例を示す．5節では，リリース・コンシステンシ・モデルの実現の形式的仕様記述の検証を行なう．6節では，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，本論文のまとめと今後の課題を述べる．

2. 状態遷移に基づく CCS 意味論

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[8]．

2.1 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[7]．

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

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 $\xrightarrow{\alpha} \subseteq \mathcal{E} \times \mathcal{E}$ 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．

$E \xrightarrow{\alpha} E'$

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．

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

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 $\xrightarrow{\alpha} \subseteq St \times St$ 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．

$s \xrightarrow{\alpha} s'$

2.2 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．

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

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．

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

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．

2.3 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)

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

Concurrent Composition(1)

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

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 $\stackrel{\alpha}{\rightarrow}$ 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[13]．

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[9]．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，$\forall i : 1 \ldots n \circ s_i, s_i' \models [\![\Theta_i]\!] \wedge \forall i : 1 \ldots n-1 \circ s_{i+1} = s_i'$ を満足する．

Trace

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

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

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

Conditional

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

3. メモリ・コンシステンシ・モデルの記述

本節では，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，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．

3.1 分散共有メモリ並列計算機モデル

まず，Herlihy と Wing[14] および Misra[15] 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（4.2 節を参照）．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．

$\langle \mathcal{E} \times St, Act \cup Op, \{\xrightarrow{\alpha} \mid \alpha \in Act \cup Op\}\rangle$ 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_l, 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_l, 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) $o_1, o_2 \in L_i, \exists \omega \in A^* \circ$
$\langle o_1.E_1, s_1\rangle \xrightarrow{\omega} \langle o_2.E_2, s_2\rangle$

(7) $o_1, o_2 \in H, \exists \omega \in A^* \circ$
$\langle o_1.E_1, s_1\rangle \xrightarrow{\omega} \langle o_2.E_2, s_2\rangle$

(8) $o_1, o_2 \in S_i, \exists \omega \in A^* \circ$
$\langle o_1.E_1, s_1\rangle \xrightarrow{\omega} \langle o_2.E_2, s_2\rangle$

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

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

3.2 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 [1),2),11),12)]．

- 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 [11)] 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 $acq_i(z) \vec{i} o_i$ 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 $o_i \vec{i} rel_i(z)$ 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．

$$
\begin{aligned}
&x_l \in \mathcal{M}, v_l, v_m \in Val, \forall i : 1 \ldots n \circledcirc \\
&(((\forall o \in A^H_{i+w} \circledcirc \exists per_i(o) \in S_i \; of \; A^H_{i+w}) \\
&\wedge \\
&(\forall r_i(x_l, v_l) \in S_i \; of \; A^H_{i+w}, \\
&\exists j : 1 \ldots n, \exists \omega \in A^* \circledcirc \\
&\quad ((\exists per_i(w_j(x_l, v_l)) \in S_i \; of \; A^H_{i+w}, \\
&\quad per_i(w_j(x_l, v_m)) \notin \omega \circledcirc \\
&\qquad \langle per_i(w_j(x_l, v_l)).E_l, s_l \rangle \\
&\qquad \xrightarrow{per_i(w_j(x_l,v_l)) \; \omega \; r_i(x_l,v_l)} \langle E_m, s_m \rangle) \\
&\quad \vee (((\exists per_i(w_j(x_l, v_m)) \in S_i \; of \; A^H_{i+w}, \\
&\quad per_i(w_j(x_l, v_m)) \notin \omega \circledcirc \\
&\qquad \langle start.E_0, s_0 \rangle \xrightarrow{\omega \; r_i(x_l,v_l)} \langle E_m, s_m \rangle) \\
&\quad \vee per_i(w_j(x_l, v_m)) \notin S_i \; of \; A^H_{i+w}) \\
&\qquad \Rightarrow v_l = \perp)))) \\
&\wedge \\
&(\forall z \in Syn \circledcirc \\
&\quad \forall acq_i(z) \in S_i \; of \; A^H_{i+w}, \\
&\quad \exists j : 1 \ldots n, \exists \omega \in A^* \circledcirc \\
&\quad ((\exists rel_j(z) \in S_j \; of \; A^H_{j+w}, \\
&\quad rel_j(z), acq_j(z) \notin \omega \circledcirc \\
&\qquad \langle rel_j(z?).E_l, s_l \rangle \xrightarrow{rel_j(z) \; \omega \; acq_i(z)} \langle E_m, s_m \rangle) \\
&\quad \vee (rel_j(z) \notin \omega \circledcirc \\
&\quad \langle start.E_0, s_0 \rangle \xrightarrow{\omega \; acq_i(z)} \langle E_m, s_m \rangle)))
\end{aligned}
$$

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

4. 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．

4.1 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 [2),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 ☆

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．

4.2 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

☆ 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．

逐次実行列 S_i を持つ．

以下に，リリース・メモリシステムの基本型宣言，略記定義，および，型を定義する．

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

$$write\_tuple == \mathrm{N}_1 \times (\mathrm{N}_1 \cup \{\top\}) \times \mathrm{N}_1 \times \mathcal{M} \times Val$$

$$confirmed\_process == \mathrm{N}_1 \pfun \mathrm{seq}\,\mathrm{N}_1$$

$$Bool == \{true, false\}$$

$$MaxOutQueue, MaxInQueue : \mathrm{N}_1$$

$$MaxSerial, MaxLocalHis : \mathrm{N}_1$$

$$update\_confirmation : confirmed\_process \pfun (confirmed\_process \times Bool)$$

各プロセス P_i は，以下の状態スキーマ s_i を持ち，操作スキーマ $InitP_i$ で初期化される．

s_i

$$pn_i : \mathrm{N}_1$$
$$M_i : \mathcal{M} \pfun (Val \cup \{\bot\})$$
$$z_i : Syn \cup \{\bot\}$$
$$t_i : \mathrm{N}$$
$$OutQueue_i : \mathrm{seq}\, write\_tuple$$
$$InQueue_i : \mathrm{seq}\, write\_tuple$$
$$ConfirmF_i : confirmed\_process$$
$$L_i : \mathrm{seq}\, A$$
$$S_i : \mathrm{seq}\, A$$

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

$InitP_i$

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

$$pn_i' = pn_i?$$
$$M_i' = \lambda x : \mathcal{M} \odot \bot$$
$$z_i = \bot$$
$$t_i = 0$$
$$OutQueue_i' = \langle\rangle$$
$$InQueue_i' = \langle\rangle$$
$$ConfirmF_i' = \langle\rangle$$
$$L_i' = \langle\rangle$$
$$S_i' = \langle\rangle$$

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．

$Read_i$

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

$$v! = M_i x?$$
$$pn_i' = pn_i$$
$$M_i' = M_i$$
$$z_i' = z_i$$
$$t_i' = t_i$$
$$OutQueue_i' = OutQueue_i$$
$$InQueue_i' = InQueue_i$$
$$ConfirmF_i' = ConfirmF_i$$
$$L_i' = L_i \frown \langle r_i(x?, v!)\rangle$$
$$S_i' = S_i \frown \langle r_i(x?, v!)\rangle$$

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．

$Write_i$

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

$$pn_i' = pn_i$$
$$M_i' = M_i \oplus \{x? \mapsto v?\}$$
$$z_i' = z_i$$
$$t_i' = t_i + 1$$
$$OutQueue_i' = OutQueue_i \frown \langle(pn_i, \top, t_i', x?, v?)\rangle$$
$$InQueue_i' = InQueue_i$$
$$ConfirmF_i' = ConfirmF_i \oplus \{t_i' \mapsto \langle\rangle\}$$
$$L_i' = L_i \frown \langle w_i(x?, v?)\rangle$$
$$S_i' = S_i \frown \langle w_i(x?, v?)\rangle$$

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．

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

$message? : write\_tuple$

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

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

$tar_j = \top \vee tar_j = pn_i$

$pn_i' = pn_i$

$M_i' = M_i$

$z_i' = z_i$

$t_i' = t_i$

$OutQueue_i' = OutQueue_i$

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

$ConfirmF_i' = ConfirmF_i$

$L_i' = L_i$

$S_i' = S_i$

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

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

$InQueue_i \neq \langle\rangle$

$pn_i' = pn_i$

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

$tar_j = \top$

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

$z_i' = z_i$

$t_i' = t_i$

$OutQueue_i' =$

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

$InQueue_i' = tail\ InQueue_i$

$ConfirmF_i' = ConfirmF_i$

$L_i' = L_i$

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

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

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

$z? : Syn$

$update! : Bool$

$InQueue_i \neq \langle\rangle$

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

$tar_j = pn_i$

$InQueue_i' = tail\ InQueue_i$

$pn_i' = pn_i$

$M_i' = M_i$

$z_i' = z?$

$t_i' = t_i$

$OutQueue_i' = OutQueue_i$

$(ConfirmF_i', update!) =$

$\quad update\_confirmation($

$\quad ConfirmF_i \oplus$

$\quad \{t_j \mapsto (ConfirmF_i\ t_j \frown \langle src_j \rangle)\})$

$update! = true \Rightarrow z_i' = \perp$

$\quad \wedge\ L_i' = L_i \frown \langle rel_i(z?) \rangle$

$\quad \wedge\ S_i' = S_i \frown \langle rel_i(z?) \rangle$

$update! = false \Rightarrow z_i' = z_i$

$\quad \wedge\ L_i' = L_i \wedge S_i' = S_i$

4.3 リリース・メモリの並行性の記述

本節では，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，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 $\overline{pipe_j}$ を通して他の全てのプロセスと通信する．

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

ラベル $receive_i(message?)$ 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，2.3 節の **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．

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

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，$z?$

$$
\begin{aligned}
&\langle \mathcal{RM}, s_0 \rangle \\
\xrightarrow{\omega syn\_a_1(z)} \quad &\langle (Sem_y \mid Sem_z \mid Acq_1.acq(z).P_1 \mid P_2) \setminus L, s'_0 \rangle \\
\xrightarrow{Acq_1} \quad &\langle (Sem_y \mid \overline{acq}(z).rel(z).Sem_z \mid acq(z).P_1 \mid P_2) \setminus L, s'_1 \rangle \\
\xrightarrow{\tau} \quad &\langle (Sem_y \mid rel(z).Sem_z \mid w_1(x?, v?).P_1 \mid P_2) \setminus L, s'_2 \rangle \\
\xrightarrow{w_1(x_1,1)} \quad &\langle (Sem_y \mid rel(z).Sem_z \mid broadcast_1(message!).P_1 \mid P_2) \setminus L, s'_3 \rangle \\
\xrightarrow{broadcast_1((z,1,\top,1,x_1,1))} \quad &\langle (Sem_y \mid rel(z).Sem_z \mid P_1 \mid receive_2(message?).P_2) \setminus L, s'_4 \rangle \\
\xrightarrow{receive_2((z,1,\top,1,x_1,1))} \quad &\langle (Sem_y \mid rel(z).Sem_z \mid P_1 \mid Apply_2.P_2) \setminus L, s'_5 \rangle \\
\xrightarrow{Apply_2} \quad &\langle (Sem_y \mid rel(z).Sem_z \mid receive_1(message?).P_1 \mid P_2) \setminus L, s'_6 \rangle \\
\xrightarrow{receive_1((z,2,1,1,x_1,1))} \quad &\langle (Sem_y \mid rel(z).Sem_z \mid rel_1(z?).P_1 \mid P_2) \setminus L, s'_7 \rangle \\
\xrightarrow{syn\_r_1(z)} \quad &\langle (Sem_y \mid rel(z).Sem_z \mid check_1(z).\overline{rel}(z).P_1 \mid P_2) \setminus L, s'_8 \rangle \\
\xrightarrow{check_1(z)} \quad &\langle (Sem_y \mid rel(z).Sem_z \mid \overline{rel}(z).P_1 \mid P_2) \setminus L, s'_9 \rangle \\
\xrightarrow{\tau} \quad &\langle (Sem_y \mid Sem_z \mid P_1 \mid acq_2(z?).P_2) \setminus L, s'_{10} \rangle \\
\xrightarrow{syn\_a_2(z)} \quad &\langle (Sem_y \mid Sem_z \mid P_1 \mid Acq_2.acq(z).P_2) \setminus L, s'_{11} \rangle \\
\xrightarrow{Acq_2} \quad &\langle (Sem_y \mid \overline{acq}(z).rel(z).Sem_z \mid P_1 \mid acq(z).P_2) \setminus L, s'_{12} \rangle \\
\xrightarrow{\tau} \quad &\langle (Sem_y \mid rel(z).Sem_z \mid P_1 \mid P_2) \setminus L, s'_{13} \rangle \\
&\cdots
\end{aligned}
$$

図 2 リリース・メモリの展開例
Fig. 2 Example of $\mathcal{RM}$ transitions

に対応したクリティカル・セクションから出る．

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

以上の定義により，以下のプロセス P_i が定義される．

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

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．

$Sem_a \stackrel{\text{def}}{=} \overline{acq}(a).rel(a).Sem_a$

$\cdots$

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

$\mathcal{RM} \stackrel{\text{def}}{=} id(pn_1?).InitP_1.\cdots.id(pn_n?).InitP_n.$

$(Sem_a \mid \cdots \mid Sem_z \mid P_1 \mid \cdots \mid P_n) \setminus L$

$where\ L = \{acq(a), \cdots, acq(z),$

$rel(a), \cdots, rel(z), pipe_1, \cdots, pipe_n\}$

4.4 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図 2 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．

また，図 3 に，図 2 のトレースの最初の 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．

5. リリース・コンシステンシ・モデルの実現の検証

本節では，4 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，3.2 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 2.3 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

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\langle \overline{acq}(z).rel(z).Sem_z, s_1'\rangle \xrightarrow{\overline{acq}(z)} \langle rel(z).Sem_z, s_1'\rangle \quad \langle acq(z).P_1, s_1'\rangle \xrightarrow{acq(z)} \langle P_1, s_2'\rangle}{\langle (\overline{acq}(z).rel(z).Sem_z \mid acq(z).P_1), s_1'\rangle \xrightarrow{\tau} \langle (rel(z).Sem_z \mid P_1), s_2'\rangle}}{\langle (Sem_y \mid \overline{acq}(z).rel(z).Sem_z \mid acq(z).P_1), s_1'\rangle \xrightarrow{\tau} \langle (Sem_y \mid rel(z).Sem_z \mid P_1), s_2'\rangle}}{\langle (Sem_y \mid \overline{acq}(z).rel(z).Sem_z \mid acq(z).P_1 \mid P_2), s_1'\rangle \xrightarrow{\tau} \langle (Sem_y \mid rel(z).Sem_z \mid P_1 \mid P_2), s_2'\rangle}}{\langle (Sem_y \mid \overline{acq}(z).rel(z).Sem_z \mid acq(z).P_1 \mid P_2) \setminus L, s_1'\rangle \xrightarrow{\tau} \langle (Sem_y \mid rel(z).Sem_z \mid P_1 \mid P_2) \setminus L, s_2'\rangle} \ (\tau \notin L)$$

図 3 τ 操作の遷移規則の適用例
Fig. 3 Example of application of transition rules to τ action

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 1] $w_j(x?, v?)$ は任意のプロセス 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 $w_j(x?, v?)$ 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 $Write_i$ 中の $M_i' = M_i \oplus \{x_l \mapsto v_l\}$ 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 ⊤（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．

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

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．

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

操作スキーマ $Apply_i$ は，$InQueue_i$ の先頭の要素の宛先が ⊤（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 ⊤（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．

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

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 ⊤（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．

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

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 2] 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，4.3 節の定義より，

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

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 2] は成立する．

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

□

[定理 1] 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 1] 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，

$$\begin{aligned}
&\forall i : 1 \ldots n \circ \\
&\quad (\forall\, o \in A^H_{i+w} \circledcirc \exists\, per_i(o) \in S_i \; of \; A^H_{i+w})
\end{aligned}$$

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:

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

s_l の状態のローカルメモリ M_i には，値 $x_l \mapsto v_k$ を持つ．状態 s_l から s_m のトレース ω 中には，$w_j(x_l, v_m)$ 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 2] 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 2] 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，

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

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，

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

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 1] 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．

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

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 1] は証明された．

□

6. 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[16]，Ahamad[5]，Kohli[17] 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，本節では，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[18] 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．

6.1 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[19] 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[20),21)]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．

6.2 (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[22)] や Z と CSP の結合である CSP-Z[23)]，Smith[24)]，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[25)] 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．

6.2.1 (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[18)] 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[26)]．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[27)]．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．

6.2.2 単一イベント vs 多重イベント

Fischer[18)] 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．

6.2.3 プロセスのブロック方式

Fischer[18)] 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．

6.3 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．

7. おわりに

本論文では，まず，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であるコーザル・メモリ・コンシステンシ・モデル[4),5),9),10)]に続いて，リリース・コンシステンシ・モデル[11),12)]について田口と荒木が提案した形式手法[8)]を用いて形式的に記述した．

我々は，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した[9),10)]．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した．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．

次に，リリース・コンシステンシ・モデル[11),12)]を

もつ分散共有メモリの実現について，コーザル・メモリ・コンシステンシ・モデルの実現[4),5),9),10)]と同様，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，分散共有メモリのトレースを例示した．

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を弱ベクトル時計を用いて Z にて記述することで，入力待ち行列内で因果先行順に並べるとともに，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した．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[9),10)]．

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 にて分けて記述することで，value-passing CCS にて同期変数に対応する複数のセマフォを容易に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 のループ構造にて記述した．

最後に，コーザル・メモリ・コンシステンシ・モデル[4),5)]と同様に，リリース・コンシステンシ・モデル[11),12)]について，その分散共有メモリシステムが，そのメモリ・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．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 による記述と「状態遷移に基づく意味論」の遷移規則を用いて，この意味論の式で記述する．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証明する．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というこの形式手法の有効性を示した．

これらの代表的な二つのモデルとそれら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．

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，実際にその他の従来モデルや新しいモデルについて記述・検証することが挙げられる．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，全てのプロセス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，全てのプロセス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，前述のように，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，渡邊勝正教授，関浩之教授，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，深く感謝の意を表します．また，最後に，貴重なコメントを頂いた査読者，編集委員に深く感謝の意を表します．

参考文献

1) 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).
2) 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).
3) Coulouris, G., Dollimore, J. and Kindberg, T.: *Distributed Systems*, Addison-Wesley, second edition (1994).
4) 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).
5) 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).
6) Spivey, J.: *The Z Notation*, Prentice Hall, second edition (1992).
7) Milner, R.: *Communication and Concurrency*, Prentice Hall (1989).
8) 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).
9) 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).
10) 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).
11) 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).
12) 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).
13) Stirling, C.: Modal and temporal logic for processes, *Logic for Concurrency*, Lecture Notes in Computer Science, Vol. 1043, pp. 149–237 (1996).
14) 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).
15) 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).
16) 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).
17) 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).
18) 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).
19) 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).
20) McMillan, K.L.: *Symbolic Model Checking: An Approach to the State Explosion Problem*, PhD Thesis, Carnegie Mellon University (1992).
21) Clarke, E.M., Grumberg, O., Hiraishi, H., Jha, S., Long, D. E. and McMillan, K. L.: Verification of the futurebus+ cache coherence protocol, *Formal Methods in System Design*, Kluwer Academic Publishers, pp. 217–232 (1995).
22) 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).
23) Fischer, C.: Combining CSP and Object-Z, Technical Report TRCF-97-1, University of Oldenburg (1997).
24) 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).
25) 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).
26) 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).
27) 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).

(平成 10 年 10 月 30 日受付)
(平成 10 年 12 月 21 日再受付)
(平成 11 年 1 月 12 日採録)

高田 司郎（正会員）

1953年生．1979年大阪大学基礎工学部情報工学科卒業．1993年奈良先端科学技術大学院大学情報科学研究科博士前期課程入学，1999年同研究科後期課程修了．1979年（株）CSK入社．1993年より（株）けいはんな交流部課長．博士（工学）．エキスパートシステム，オブジェクト指向，形式手法，分散共有メモリ，インターネットの利用技術などの研究に従事．人工知能学会，ソフトウェア科学会，IEEE各会員．

田口 研治（正会員）

1956年生．1981年東洋大学文学部哲学科卒業．1984年ビクトリア大学哲学部B.A (Hons) 卒業．1987年同大学院M.A修了．1995年奈良先端科学技術大学院大学情報科学研究科博士後期課程入学，1997年中退．1985年CSK総合研究所入所．1987年関西新技術研究所．1990年株式会社CSK．1997年九州大学大学院システム情報科学研究科情報工学専攻助手．1999年より筑紫女学園大学アジア文化学科助教授．エキスパートシステム構築用ツール，形式手法，モーバイル計算，論理プログラミングのモーバイルエージェントへの拡張などの研究に従事．ソフトウェア科学会，IEEE各会員．

城 和貴（正会員）

阪大·理·数学卒．日本DEC，ATR視聴覚研究所（日本DECより出向），（株）クボタ・コンピュータ事業推進室で勤務．1993年奈良先端科学技術大学院大学情報科学研究科博士前期課程入学，1996年同研究科後期課程修了，同年同研究科助手．1997年和歌山大学システム工学部情報通信システム学科講師，1998年同学科助教授．1999年より奈良女子大学理学部情報科学科教授．工博．画像処理，文字認識，ニューラルネットワーク，並列計算機アーキテクチャ，自動並列化コンパイラ，並列計算機の解析モデル，視覚化などの研究に従事．IEEE会員．

福田 晃（正会員）

1954年生．1977年九州大学工学部情報工学科卒業．1979年同大学院修士課程修了．同年NTT研究所入所．1983年九州大学大学院総合理工学研究科助手．1989年同大学助教授．1994年より奈良先端科学技術大学院大学情報科学研究科教授．工学博士．オペレーティング・システム，並列化コンパイラ，計算機アーキテクチャ，並列／分散処理，性能評価などの研究に従事．本学会平成2年度研究賞，平成5年度Best Author賞受賞．著書「並列オペレーティングシステム」（コロナ社）訳書「オペレーティングシステムの概念」（共訳，培風館）．ACM，IEEE，電子情報通信学会，日本OR学会各会員．