Introduction to Distributed Algorithms 輪講
Chapter 2. The Model

米澤研修士 2 年 金田憲二

平成 14 年 4 月 8 日

概 要

本章では，計算システムを記述するモデルについて述べる．
Section 1 では遷移システム (transition system) について述べる．Section 2, Section 3 ではモデルの特性の証明方法について述べる．Section 4 では，拡張されたモデルについて説明する．

1 Transition Systems and Algorithms

ここでは，分散システムを，各々が自律して動く process の集合としてモデル化する．system 全体の状態は個々の process の状態から成り立つ．

また，ここでは，分散システムをメッセージパッシング型のシステムとしてモデル化する[1]．プロセスはメッセージを交換することによって通信し，通信の種類には同期通信，非同期通信[2]の二種類がある．

1.1 Transition Systems

まず始めに transition system の定義を行う．個々の process， system 全体，共にこの transistion system として表すこととなる．

Definition 1.1 *transition system* とは $S = (\mathcal{C}, \rightarrow, \mathcal{I})$ である．
ただし，

- $\mathcal{C}$ は *configuration* の集合

[1] 分散システムを共有変数型としてモデル化するやりかたもある．共有変数への読み書きによって process 間で通信が行われる．

[2] 同期通信，非同期通信を簡単に区別すると以下のようになる．

同期通信 相手が受信可能になるまでメッセージを送信することができない．送信されたメッセージはすぐに受信される．

非同期通信 相手が受信可能でなくてもメッセージを送信することができる．送信されたメッセージはすぐに受信されるとは限らない．

- $\rightarrow$ は $\mathcal{C}$ 上の二項遷移関係[3]
- $\mathcal{I}$ は $\mathcal{C}$ の部分集合 *(initial configuration* と呼ぶ*)*

である．

Definition 1.2 $S = (\mathcal{C}, \rightarrow, \mathcal{I})$ を *transition system* とする．このとき，S の *execution* とは以下を満たす最大列 $E = (\gamma_0, \gamma_1, \gamma_2, \ldots)$ である．

- $\gamma_0 \in \mathcal{I}$
- $\forall i \geq 0, \gamma_i \rightarrow \gamma_{i+1}$

$\gamma \rightarrow \delta$ を満たす δ が存在しない γ のことを *terminal* configuration と呼ぶ．E は最大列であるとき，無限列か，末尾が terminal configuration である列．

Definition 1.3 任意の $i(0 \leq i < k)$ に対して $\gamma_i \rightarrow \gamma_{i+1}$ を満たす列 $\gamma_0(= \gamma), \gamma_1, \gamma_2, \ldots, \gamma_k(= \delta)$ が存在するとき，δ は γ から到達可能 *(reachable)* といい，$\gamma \rightsquigarrow \delta$ と表す．

Configuration δ が到達可能であるとは，*initial configuration* から到達可能であること．

1.2 Systems with Asynchronous Message Passing

次に，非同期分散システムを定義する．非同期分散システムとは，各々自律して動く process が存在し，process 間で非同期通信が可能なシステムである．

まず個々の process についてモデル化を行い，次に system 全体をモデル化することとなる．

以後，system 全体の遷移，状態を，それぞれ transition，configuration と呼び，process の遷移，状態を，それぞれ event，state と呼ぶ．$\mathcal{M}$ をメッセージの集合とし，$\mathcal{M}$ の multiset の集合を $\mathbf{M}(\mathcal{M})$ とする．

Definition 1.4 *process* の *local algorithm* とは $(Z, I, \vdash^i, \vdash^s, \vdash^r)$ である．

ただし

- Z は *state* の集合
- I は Z の部分集合 *(initial state* と呼ぶ．*)*
- $\vdash^i$ は $Z \times Z$ 上の関係 *(process* 内部の状態遷移．*initial event* と呼ぶ．*)*
- $\vdash^s, \vdash^r$ は $Z \times \mathcal{M} \times Z$ 上の関係 *(*送信・受信による状態遷移．*send event, receive event* と呼ぶ．*)*

[3] $\gamma \rightarrow \delta$ は $(\gamma, \delta) \in \rightarrow$ を意味するとする．

とする．

また，二項関係 $\vdash$ を以下のように定義する．

$$c \vdash d \iff ((c,d) \in \vdash^i) \vee (\exists m \in \mathcal{M}, (c,m,d) \in \vdash^s \cup \vdash^r)$$

今後，$\mathbf{P}$ を process の集合とする．

Definition 1.5 *process* の集合 $\mathbf{P} = \{p_1, \ldots, p_N\}$ の分散アルゴリズムは，$\mathbf{P}$ 中の *process* の *local algorithm* の集合である．

よって非同期分散システムは以下のように定義することができる．

Definition 1.6 非同期通信下の *process* $p_1, \ldots, p_N$ の分散アルゴリズムとは $S = (\mathcal{C}, \rightarrow, \mathcal{I})$ である．

ただし，

- p_i の *local algorithm* は $(Z_{p_i}, I_{p_i}, \vdash^i_{p_i}, \vdash^s_{p_i}, \vdash^r_{p_i})$
- $\mathcal{C} = \{(c_{p_1}, \ldots, c_{p_N}, M) | (\forall p \in \mathbf{P}, c_p \in Z_p)$ *and* $M \in \mathbf{M}(\mathcal{M})\}$

  (system 全体の状態 *(configuration)* は個々の *process* の状態 *(state)* の集合と転送中のメッセージからなる．*)*
- $\rightarrow = (\cup_{p \in \mathbf{P}} \rightarrow_p)$

  ただし $\rightarrow_p$ は以下のように定義され，

  $(c_{p_1}, \ldots, c_{p_i}, \ldots, c_{p_N}, M_1) \rightarrow_{p_i} (c_{p_1}, \ldots, c'_{p_i}, \ldots, c_{p_N}, M_2)$ ならば以下の条件のうちどれか一つが成り立つ．
  - $(c_{p_i}, c_{p'_i}) \in \vdash^i_{p_i}$ *and* $M_1 = M_2$
  - $\exists m \in \mathcal{M}, (c_{p_i}, m, c'_{p_i}) \in \vdash^s_{p_i}$ *and* $M_2 = M_1 \cup \{m\}$
  - $\exists m \in \mathcal{M}, (c_{p_i}, m, c'_{p_i}) \in \vdash^r_{p_i}$ *and* $M_1 = M_2 \cup \{m\}$
- $\mathcal{I} = \{(c_{p_i}, \ldots, c_{p_N}, M) | (\forall p \in \mathbf{P}, c_p \in I_p) \wedge (M = \emptyset)\}$

とする．

process p の event e は configuration $\gamma = (c_{p_1}, \ldots, c_p, \ldots, c_{p_N}, M)$ において，以下の条件のうちいずれかを満たすとき，適応可能 (applicable) と呼ぶ．また遷移後の configuration を返す関数 $e(\gamma)$ を以下のように定義する．

- internal event $e = (c, d)$ において，$c = c_p$ ならば適応可能ある．このとき $e(\gamma) = ((c_{p_1}, \ldots, d, \ldots, c_{p_N}, M)$ と定義する．
- send event $e = (c, m, d)$ において，$c = c_p$ ならば適応可能である．このとき $e(\gamma) = ((c_{p_1}, \ldots, d, \ldots, c_{p_N}, M \cup \{m\})$ と定義する．

- receive event $e = (c, m, d)$ において，$c = c_p$ かつ $m \in M$ ならば適応可能である．このとき $e(\gamma) = ((c_{p_1}, \ldots, d, \ldots, c_{p_N}, M \setminus \{m\})$ と定義する．

また．各々のメッセージを受け取る process は unique であることを仮定している．メッセージを受け取る process を，そのメッセージの *destination* と呼ぶ．

1.3 Systems with Synchronous Message Passing

次に，同期分散システムを定義する．非同期分散システムの場合とほぼ同様に定義することができる．ただし，対応する send と receive で一つの遷移となる．

Definition 1.7 同期通信下の *process* $p_1, \ldots, p_N$ の分散アルゴリズムとは $S = (\mathcal{C}, \rightarrow, \mathcal{I})$ である．
ただし，

- p_i の *local algorithm* は $(Z_{p_i}, I_{p_i}, \vdash^i_{p_i}, \vdash^s_{p_i}, \vdash^r_{p_i})$
- $\mathcal{C} = \{(c_{p_1}, \ldots, c_{p_N}) | \forall p \in \mathbf{P}, c_p \in Z_p\}$
- $\rightarrow = (\cup_{p \in \mathbf{P}} \rightarrow_p) \cup (\cup_{p,q \in \mathbf{P}: p \neq q} \rightarrow_{pq})$
  ただし $\rightarrow_p$ ，$\rightarrow_{pq}$ は以下のように定義され，
  $(c_{p_1}, \ldots, c_{p_i}, \ldots, c_{p_N}, M_1) \rightarrow_{p_i} (c_{p_1}, \ldots, c'_{p_i}, \ldots, c_{p_N}, M_2)$ ならば以下が成り立つ．
  - $(c_{p_i}, c_{p'_i}) \in \vdash^i_{p_i}$

  $(c_{p_1}, \ldots, c_{p_i}, \ldots, c_{p_j}, \ldots, c_{p_N}) \rightarrow_{p_i p_j} (c_{p_1}, \ldots, c'_{p_i}, \ldots, c'_{p_j}, \ldots, c_{p_N})$ ならば以下が成り立つ．
  - $\exists m \in \mathcal{M}, (c_{p_i}, m, c'_{p_i}) \in \vdash^s_{p_i}$ *and* $(c_{p_j}, m, c'_{p_j}) \in \vdash^r_{p_j}$
- $\mathcal{I} = \{(c_{p_i}, \ldots, c_{p_N}) | \forall p \in \mathbf{P}, c_p \in I_p\}$

とする．

非同期通信用につくられたアルゴリズムを同期通信で用いる場合には注意が必要となる．例えば 2 つの process がお互いにメッセージを交換し合う system を考える．共に send, receive の順に event を行うとすると，非同期通信下ではメッセージは送受信されるが，同期通信下では dead lock が起こってしまう．

1.4 Fairness

「適応可能な event がいつまでたっても実行されない」ということはないことの保証をしたい場合がある．この保証のことを fair と呼ぶ．

例えば，「execution 中で，send event が無限に長い間適応可能ならば，無限回 send event が実行される」ことの保証など．

Weak fair execution 実行されないが無限に多い連続した configuration 中で適応可能という event は存在しない．(no event is applicable in infinitely many **consecutive** configurations without ocurring in the execution.) つまり一度適応可能になり，そのあとずっと適応可能な event が，一回も実行されないということはない．

Strong fair execution 実行されないが無限に多い configuration 中で適応可能という event は存在しない．(no event is applicable in infinitely many configurations without ocurring in the execution.) つまり無限回適応可能になる event が，一回も実行されないということはない

基本的には，この本で紹介するアルゴリズムに fairness を仮定していない．

2 Proving Properties of Transition Systems

アルゴリズムがある種の性質 (property) を持つことを証明したい場合がある (正しく動作することの証明など)．ここでは，アルゴリズムのもつ property をどうやって証明するかについて述べる．

property を以下の二種類に分類して考える．

safety property 任意の execution における任意の configuration において成り立つ property (常に成り立つ property)．

liveness property 任意の execution におけるいくつかの configuration において成り立つ property (いつかは成り立つ property)．

property は assertional verification method と呼ばれる方法を用いて証明される．configuration の集合上の単項関係 (unary relation on the set of configurations) を assertion と呼ぶ[4] [5]．

[4] すなわち，assertion P に対して，以下を満たす $C' \subseteq \mathcal{C}$ が一意に定まる．

- $\forall \gamma \in C' \Longrightarrow P(\gamma)$ は真
- $\forall \gamma \notin C' \Longrightarrow P(\gamma)$ は偽

[5] Assertion の例としては，process の local 変数の値が特定の値を取るとき真になる述語などが挙げられる．

2.1 Safety Properties

ある property が safety property であることをどうやって証明するかについて述べる．

$S = (\mathcal{C}, \to, \mathcal{I})$ を transition system とする．

任意の $\forall\gamma \to \delta$ に対して，$P(\gamma)$ が成り立つとき $Q(\delta)$ が成り立つことを，$\{P\} \to \{Q\}$ と表す．これは，P が成り立っているならば，遷移後も Q が成り立つということを意味する．

Definition 2.1 以下を条件を満たすとき，*assertion* P を S の *invariant* と呼ぶ．

- $\forall\gamma \in \mathcal{I}, P(\gamma)$
- $\{P\} \to \{P\}$

Theorem 2.2 もし P が S の *invariant* であるならば，S の任意の *execution* 中の任意の *configuration* において P は成り立つ *(P は safety property である)*．

(証明概略) S の execution を $E = (\gamma_0, \gamma_1, \gamma_2, \ldots)$ とし，帰納的に証明する．□

逆は成り立たないことに注意する．つまり，任意の execution 中の任意の confiugraion で真である assertion (safety property) が，invariant であるとは限らず，[6] 全ての safty property を Theorem 2.2 を用いて証明できるわけではない．

Theorem 2.3 Q を S の *invariant* し，任意の $\gamma \in \mathcal{C}$ において $Q \Rightarrow P$ という *assertion* が成り立っていると仮定する．このとき，S の任意の *execution* 中の任意の *configuration* において P は成り立つ．

(証明) Theorem 2.2 より，Q は任意の configuration において成り立つ．また $Q \Rightarrow P$ も任意の configuration で成り立つ．よって，P も任意の configuration において成り立つ．□

Definition 2.4 S を *transition system*，P，Q を *assertion* とする．P が以下の条件を満たすとき，*Q-derivate* と呼ぶ

- $\forall\gamma \in \mathcal{I}, Q(\gamma) \Rightarrow P(\gamma)$
- $\{Q \wedge P\} \to \{Q \Rightarrow P\}$

[6] 以下のような反例 $S = (\mathcal{C}, \to, \mathcal{I})$ を挙げることにより，証明することができる．

- $\mathcal{C} = \{a, b, c, d\}$
- $\mathcal{I} = \{a\}$
- $\to = \{(a, b), (c, d)\}$

Assertion P が $P(a), P(b), P(c)$ は真で，$P(d)$ は偽であるとき，a, b という execution しか存在しないので，P は safety property である．しかし $c \to d$ において $P(c)$ は真で，$P(d)$ は偽となるので，P は invariant ではない．

Theorem 2.5 Q が *invariant* で，P が Q*-derivate* であるとき，$Q \wedge P$ は *invariant* である．

(証明) Definition 2.1 より，

- $\forall \gamma \in \mathcal{I}, Q(\gamma) \wedge P(\gamma)$
- $\{Q \wedge P\} \rightarrow \{Q \wedge P\}$

の二つを示せばよい．

Q は invariant であるので，$\forall \gamma \in \mathcal{I}, Q(\gamma)$．また，$\forall \gamma \in \mathcal{I}, Q(\gamma) \Rightarrow P(\gamma)$ であるので，$\forall \gamma \in \mathcal{I}, P(\gamma)$．よって，前者は成り立つ．

$\gamma \rightarrow \delta$ と $Q(\gamma) \wedge P(\gamma)$ を仮定する．Q は invariant であるので，$Q(\delta)$ が成り立つ．また，P は Q-derivate であるので，$\{Q \wedge P\} \rightarrow \{Q \Rightarrow P\}$ が成り立ち，$Q(\delta) \Rightarrow P(\delta)$ が成り立つ．よって $P(\delta)$ が成り立つことから，後者も成り立つ．□

2.2 Liveness Properties

ある property が liveness property であることをどうやって証明するかについて述べる．

$S = (\mathcal{C}, \rightarrow, \mathcal{I})$ を transition system，P を述語とする．P が最終的には真になることを，*norm* 関数と *proper termination* を用いて証明する．

Definition 2.6 **term** を *termination configuration* に対して真, それ以外では偽となる述語とする．述語 (**term** $\Rightarrow P$) が S において常に成り立つとき，S は適切に終了する *(S terminates properly)* と呼ぶ．

Definition 2.7 *partial order* $(W, <)$ を，無限の続く減少列が存在しないとき，*well-founded* と呼ぶ．

Example 2.8 自然数の大小関係は *well-founded* である．

Definition 2.9 *transition system* S と *assertion* P が与えられたとする．$\mathcal{C}$ から *well-founded set* W への関数 f は，任意の $\gamma \rightarrow \delta$ に対して $f(\gamma) > f(\delta)$ が成り立つか $P(\delta)$ が成り立つとき，*(P に関する)norm* 関数と呼ばれる．

Theorem 2.10 *transition system* S と *assertion* P が与えられたとする．S が適切に終了し，ある *norm* 関数 f が存在するとき，P は，S の任意の *execution* 中のある *configuration* で真となる *(P は liveness property である)*．

(証明) S 上の execution を $E = (\gamma_0, \gamma_1, \gamma_2, \ldots)$ とする．まず E が有限長である場合を考える．exectuion の最後の configurationγ は terminal configuration であり，かつ $\mathrm{term} \Rightarrow P$ が成り立つので，$P(\gamma)$ が成り立つ．

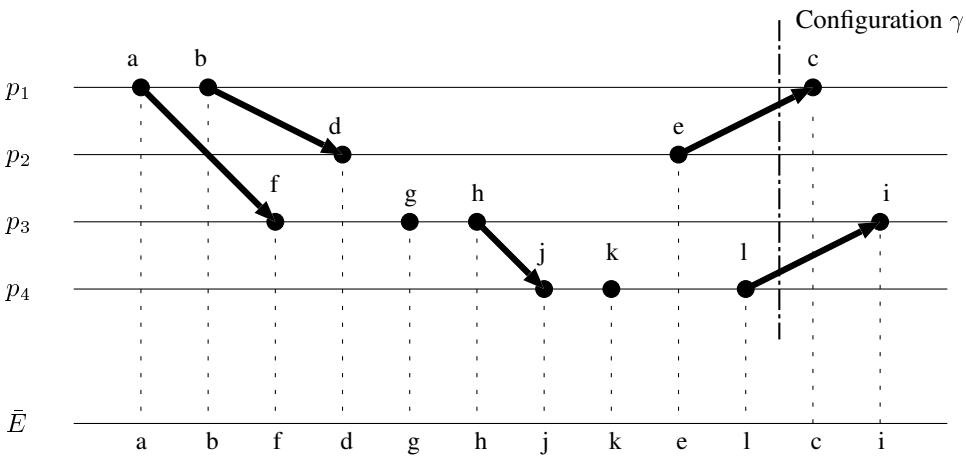

図 1: AN EXAMPLE OF A SPACE-TIME DIAGRAM

次に，E が無限長である場合を考える．E 中の，P が真である configuration を含まない最大 prefix を $E' = (\gamma_0, \gamma_1, \ldots)$ とする．E' が有限長 ($|E'| = k$) であることを証明できれば，E' の取り方より，$P(\gamma_{k+1})$ が成り立つ．

$s = (f(\gamma_0), f(\gamma_1), \ldots)$(ただし$\gamma_i \in E'$) とすると，$f$ は norm 関数であるので，s は減少列となる．W は well founded set であるので，s は有限長となる．よって $|s| = |E'|$ より E' は有限長となる．□

3 Causal Order of Events and Logical Clocks

execution を遷移の列として考えると，自然に時間というものを考えたくなる．例えば，遷移列上で event a が event b より前に位置していれば，a は b より時間的に早く起きた event とみなせる．

$E = (\gamma_0, \gamma_1, \ldots)$ を execution とし，$\bar{E} = (e_0, e_1, \ldots)$ をそれに対応する event 列とする (つまり e_i によって γ_i から γ_{i+1} に遷移)．当然，$\bar{E}$ は E から unique に定まる．この event 列を space-time diagram として表してみる (図 1 参照)．点 $a, b, \ldots$ は event を表しており，対応する send event と receive event は矢印によって結ばれている．

ある event の出現の順番を変更しても execution の最終的な configuration は変化しない場合がある．例えば，図 1 において c と i の起こる順番を変更したとしても，最終的に到達する configuration に変化はない．このとき c と i どちらが早く起こる (べき)event であるかと論ずるのは適切 (必要，本質的) ではない．

ここでは，時間を，event 列の前後の位置関係の total order として考えるのではなく，causal dependence (因果) というものを導入して考える．

3.1 Independence and Dependence of Events

連続する 2 つの event が交換可能であることは，formal には，非同期通信の分散システムにおいて以下のように表すことができる．

Theorem 3.1 γ を非同期通信分散システムにおける *configuration*，e_p と e_q を異なる *process* p，q の γ に対して適応可能な *event* とする．このとき，e_p は $e_q(\gamma)$ に適応可能, e_q は $e_p(\gamma)$ に適応可能で，$e_p(e_q(\gamma)) = e_q(e_p(\gamma))$ となる．

(証明) まず internal event，send event，receive event を一括して簡便に扱うために，(c, x, y, d) という notation を導入する．

- c, d は event が起こる前後の process の state
- x, y はこの event で受信・送信するメッセージの集合

とする．例えば，internal event (c, d) は $(c, \emptyset, \emptyset, d)$ と表される．send event (c, m, d) は $(c, \emptyset, \{m\}, d)$ と表される．receive event (c, m, d) は $(c, \{m\}, \emptyset, d)$ と表される．

この notation を用いると，process p の event $e = (c, x, y, d)$ は以下の条件を満たす時，configuration $\gamma = (c_{p_1}, \ldots, c_p, \ldots c_{p_N}, M)$ に適応可能となる．

- $c_p = c$
- $x \subseteq M$

また，このとき

$$e(\gamma) \quad = \quad (c_{p_1}, \ldots, d, \ldots, (M \setminus x) \cup y)$$

となる[7]．

$e_p = (b_p, x_p, y_p, d_p)$，$e_q = (b_q, x_q, y_q, d_q)$，$\gamma = (\ldots, c_p, \ldots, c_q, \ldots, M)$ とする．このとき $c_p = b_p$，$c_q = b_q$，$x_p \subseteq M$，$x_q \subseteq M$ となる．

ここで $x_p \cap x_q = \emptyset$ であることに注意する[8]．

e_p は γ に適応可能であるので，$\gamma_p = e_p(\gamma)$ とすると，

$$\gamma_p \quad = \quad (\ldots, d_p, \ldots, c_q, \ldots, (M \setminus x_p) \cup y_p)$$

となる．

$x_q \subseteq M$ と $x_p \cap x_q = \emptyset$ より，$x_q \subseteq (M \setminus x_p) \cup y_p$ となるので，e_q が $\gamma_p = e_p(\gamma)$ に適応可能であることが証明できる．対称性により e_p が $\gamma_q = e_q(\gamma)$ に適応可能であることも証明できる．

[7] この event によって state が c から d に変化する．転送中のメッセージの集合 M から受信したメッセージ x が取り除かれ，送信したメッセージ y が加えられる．

[8] $x_p = x_q$ を仮定する．するとメッセージの destination は unique であるので，p と q は等しくなければならない．これは前提に反する．

また，$\gamma_{pq} = e_q(\gamma_p) = e_q(e_p(\gamma))$ とすると，

$$\gamma_{pq} \quad = \quad (\ldots, d_p, \ldots, d_q, \ldots, ((M \setminus x_p \cup y_p) \setminus x_q) \cup y_q)$$

となる．対称性より，e_p は $\gamma_{qp} = e_p(\gamma_q) = e_p(e_q(\gamma))$ とすると，

$$\gamma_{qp} \quad = \quad (\ldots, d_p, \ldots, d_q, \ldots, ((M \setminus x_q \cup y_q) \setminus x_p x) \cup y_p)$$

となる．

ここで M は multiset であり，$x_p \subseteq M$ と $x_q \subseteq M$ より，

$$((M \setminus x_p \cup y_p) \setminus x_q) \cup y_q) \quad = \quad ((M \setminus x_q \cup y_q) \setminus x_p) \cup y_p)$$

となり，$\gamma_{pq} = \gamma_{qp}$ となる．□

連続する event e_p と e_q が Theorem3.1 の前提を満たさない場合がある．このとき event の順番は交換可能であるとは限らない．

$$\ldots, e_p(\gamma), e_q(e_p(\gamma)), \ldots$$

とすると，以下の条件のうちいずれかが成り立つ時，Theorem3.1 の前提は満たされるとは限らない．

- $p = q$ [9]
- e_p が send event で，e_q がそれに対応する receive event [10]

つまり，同一 process 中の event の順番と，対応する send・receive event の順番は交換できるとは限らない．このような順序変更不可能な event の関係を *causal relation* と呼ぶ．causal relation を拡張し，*causal order* という execution の event の集合上の partial order を定義する．

Definition 3.2 *execution* E の *event* 上の *causal order* $\prec$ は以下を満たす最小の関係である．

1. もし e と f が同一 *process* 中の異なる *event* で，e が f の前に起きたならば，$e \prec f$．
2. もし s が *send event* で r がそれに対応する *receive event* ならば，$s \prec r$．
3. $\prec$ は *transitive.*

$a \preceq b \Longleftrightarrow (a \prec b \vee a = b)$ とする．

$\preceq$ は partial order なので，event a,b に対して $a \preceq b$ も $b \preceq a$ も成り立たないことがある．このような a, b を *concurernt* と呼び，$a \parallel b$ と書く．また，$a \prec b$ において，event 列 $e_0(= a), e_1, \ldots, e_k(= b)$ (ただし任意の i において，e_i と e_{i+1} は Definition 3.2 の 1，2 を満たす) を causal chain と呼ぶ．

例えば，図 1 においては $b \parallel f$，$d \parallel i$ となり，a からの l の causal chain は a, f, g, h, j, k, l である．

[9] p と q が異なるという前提に反する

[10] e_p と e_p が共に γ に適応可能である，という前提に反する (場合がある)．

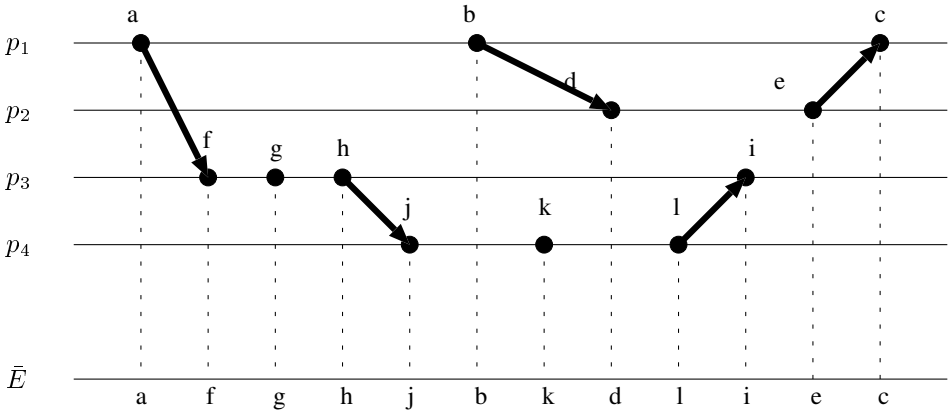

図 2: A SPACE-TIME DIAGRAM EQUIVALENT TO FIGURE 1

3.2 Equivalence of Executions: Computations

causal order に従って execution の event を並べ直しても， execution が最終的に取りうる configuration は変化しない．

event の causal order を保存する execution E と F を，*equivalent* execution と呼び $E \sim F$ と表す．

正式には以下の通り．

Theorem 3.3 $E = (\gamma_0, \gamma_1, \ldots)$ を *execution*，$\bar{E} = (e_0, e_1, \ldots)$ を E に対応する *event* 列， $f = (f_0, f_1, \ldots)$ を $\bar{E}$ の順列とする．$f_i \preceq f_j$ ならば $i \leq j$ である *(causal order* が保存される*)* とすると，f によって F が一意に定まり以下の条件を満たす．

- E と F の *initial configuration* は等しい．
- F は E は同数の *event* をもつ．
- もし E が有限ならば E と F の最後の *configuration* は等しい．このような E と F を *equivalent execution* と呼び，$E \sim F$ と表す．

例えば，図 2 と図 1 は同値な execution である．

Definition 3.4 分散アルゴリズムの *computation* は，アルゴリズムの *execution* の $\sim$ の元での同値類である．

3.2.1 Synchronous message passing

定理 3.1 は synchronous message passing にも当てはめることができる．

Theorem 3.5 γ を同期通信分散システムにおける *configuration*，e_1 を *process* p と q による *transition*, e_2 を *process* r と s による *transition* とする．r，s は p と q とは異なるとする．e_1 と e_2 共に γ に適応可能であるとき，e_1 は $e_2(\gamma)$ に適応可能, e_2 は $e_1(\gamma)$ に適応可能で，$e_1(e_2(\gamma)) = e_2(e_1(\gamma))$ となる．

3.3 Logical Clocks

clock を定義し，causality を表す．

Definition 3.6 *clock* とは，以下を満たす *event* の集合から *ordered set* $(X, <)$ への関数 Θ である．

$$a \prec b \implies \Theta(a) < \Theta(b)$$

以後，clock の例をいくつか挙げる．

Order in sequence event 列 $(e_0, e_1, e_n, \ldots)$ に対して $\Theta_g(e_i) = i$ とする．

ただし，Θ_g を求める分散アルゴリズムは存在しない[11]．

Real-time clocks 各々の process が hardware clock をもつようにモデルを拡張する．

Lamport's logical clock event a に おいて $\Theta_L(a) = k$ とする．ただし，k は以下を満たす最長 event 列 $(e_1, \ldots, e_k)$ の長さとする．

$$e_1 \prec e_2 \prec \ldots \prec e_k (= a)$$

明らかに $a \prec b$ ならば $\Theta_L(a) < \Theta_L(b)$ となる．

例えば，図 1 における Lamport's logical clock は図 3 のようになる．

Θ_L は分散アルゴリズムによって求めることができる (図 4 参照)．

- a が internal event か send event ならば，$\Theta_L(a) = \Theta_L(a') + 1$ とする．(ただし，a' は同一 process 中の一つ前の event)
- a が receive event ならば，$\Theta_L(a) = max(\Theta_L(a'), \Theta_L(b)) + 1$ とする．(ただし，a' は同一 process 中の一つ前の event，b は a に対応する send event)

[11] 任意の execution において 値が clock と等しい変数を仮定する．つまり，event 列を $(e_1, e_2, \ldots)$ すると，event e_i において変数の値は $\Theta_g(e_i) = i$ となる．これと同値な execution を考え，その event 列を $e_1, e_2, \ldots, e_{i+1}, e_i$ とすると．$\Theta_g(e_i) = i + 1$ だが，e_i における変数の値は i である．

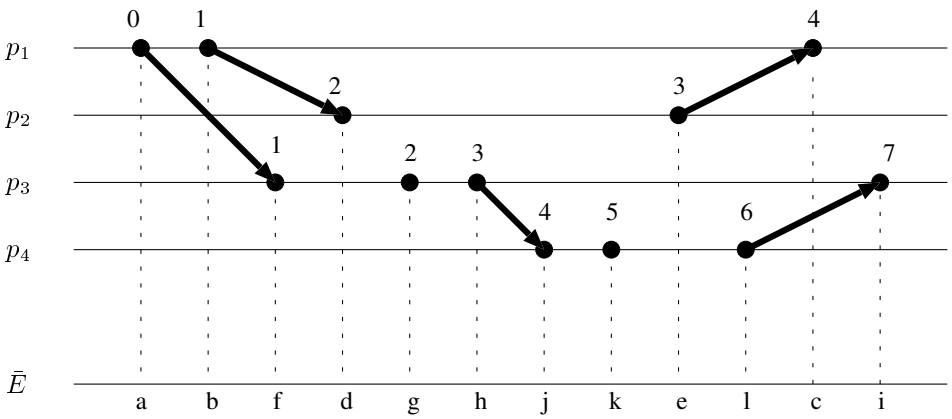

図 3: LAMPORT'S LOGICAL CLOCK

var θ_p : interger **init** 0;

(* An internal event *)
$\theta_p := \theta_p + 1$;
Change state

(* A send event *)
$\theta_p := \theta_p + 1$;
send $\langle \mathbf{messg}, \theta_\mathbf{p} \rangle$; Change state

(* An receive event *)
receive $\langle \mathbf{messg}, \theta_\mathbf{p} \rangle$; $\theta_p := max(\theta_p, \theta) + 1$
Change state

図 4: Lamport's Logical Clock

Vector clocks Lamport's logical clock の値を見るだけでは event が concurrent であるかどうか分からない．

以下を満たすように clock を定義することによって，clock の値をみるだけで，concurrency かどうか判断できるようにする．

$$a \prec b \iff \Theta(a) < \Theta(b)$$

すなわち，$\Theta(x) < \Theta(y)$ も $\Theta(y) < \Theta(x)$ も成り立たなければ，event x と y は concurrent．

Θ_v を event の集合から $(\mathbf{N}^N, <_v)$ への関数とし，$\Theta_v(a) = (a_1, \ldots, a_N)$ と定義する．ただし

$$(a_1, \ldots, a_n) \leq_v (b_1, \ldots, b_n) \Longleftrightarrow \forall i (1 \leq i \leq n) : a_i \leq b_i$$

であり，a_i を $e \preceq a$ を満たす process p_i の event e の数とする．

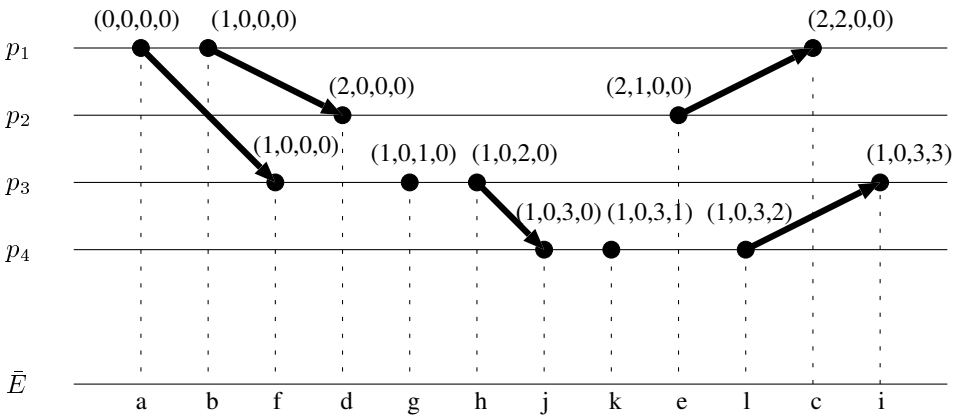

図 5: VECTOR CLOCK

図 1 における vector clock は図 5 のようになる．図 3 の lamport の clock の場合と異なり，conncurrent な event 間では order は定まらない．

また，vector clock も lamport の clock と同様，分散アルゴリズムによって計算可能である．

4 Additional Assumptions, Compelexity

いままででアルゴリズムを表し，検証を行うモデルについて述べた．
以後の章で必要の応じて使われる仮定や notation について述べる．

4.1 The Network Topology

実際には任意の process がそのほかの全ての process に対してメッセージを送信可能であるとは限らない．

process p がメッセージを送信可能な process の集合を *neighbors* と呼ぶ．また，p が q へメッセージを送信可能なとき，p，q 間に *channel* が存在するという．相互に通信可能な channel を *bidirectional* と呼び，片方の process からしか通信可能な channel を *unidirectional (directed)* と呼ぶ．

以上のことは *graph* $G = (V, E)$ と表すことができる．ただし V は process の集合，E は channel の集合である．このグラフを *network topology* と呼ぶこともある．大概の場合，グラフは *connected* であると仮定する．

Rings ノード数 N の ring では，ノードを $v_0, v_1, \ldots, v_{N-1}$ としたとき，エッジは $v_i v_{i_1}$ 間のみ存在 (ただし，添字は mod N)．

Trees ノード数 N の tree とは，$N-1$ 個のエッジが存在する連結グラフ．

Stars ノード数 N の star では，中央ノードと他の $N-1$ 個のノードとの間のみエッジが存在．

Cliques 任意の 2 ノード間に edge が存在するグラフ．

Hypercubes ノード数 $N = 2^n$ のとき，$HC_N = (V, E)$ は以下のようになる．

$$V \quad = \quad \{(b_0, \ldots, b_{n-1}) | b_i \in \{0, 1\}\}$$

というように個々のノードに bit を割り当てられており，bit 数が 1 違うノード間のみエッジが存在．

また，topology が *static* であるとは，計算中に topology が変化しないことを意味する．*dynamic* であるとは，計算中に channel，process が追加・削除されることを意味する．

この topology の変化も configuration の transition としてモデル化することができる (Chapter 4 参照)．

4.2 Properties of Channels

Reliability 送信されたメッセージが一度だけ受信される channel を *reliable* と呼ぶ．reliable でないときは，以下のような *communication failure* が起こりうる．

loss 送信したメッセージが受信されないこと．M からメッセージが取り除かれる transition としてモデル化される．

garbling 送信したのと異なるメッセージを受信すること．M 中のメッセージを変化させる transition としてモデル化される．

duplication 送信した以上の個数のメッセージを受信すること．M 中のメッセージを複製する transition としてモデル化される．

creation 送信していないメッセージを受信すること．M 中にメッセージを追加する transition としてモデル化される．

この本では，基本的には channel は reliable であるとする．

The fifo property メッセージが送信された順に受信される channel を *fifo* と呼ぶ．この本では，基本的には channel が fifo であることを仮定しない．M を queue としてモデル化すればよい．

Channel capacity *capacity* とは，channel が同時に転送することの出来るメッセージの数．

4.3 Real-time Assumptions

物理的な clock のようなものを仮定する．新たに real-time clock 用の変数を個々の process に追加し，その変数を変更する transition も追加することによってモデル化される．ただしこのような global time frame を導入することによって，Theorem 3.1 などは成り立たなくなってしまう．

4.4 Process Knowledge

process にあらかじめ与えられている情報．

Topological information process の数，グラフの直径など

Process identity 個々の process もつ unique な名前

Neighbor identites 隣接 process の名前

4.5 The Complexity of Distributed Algorithms

アルゴリズムを以下の様な基準で評価する．

Message complexity アルゴリズム中に交換されたメッセージの総数

Bit complexity メッセージの集合 $\mathcal{M}$ の大きさ

Time complexity アルゴリズム中に起きた遷移数

Space complexity process が計算を行うのに必要とするメモリ量

A Pseudocode Conventions

この本でアルゴリズムを記述するのに用いられる疑似コードについて．

A.1 Pascal-like pseudo code

Pascal に似た疑似コードを用いる．多くの場合，個々の process の local algorithm を図示する．プログラム中に現れる変数は個々の process で独自のものである (例えば $Neigh_p$ は process p のもつ変数で，その値は p に隣接するノードの集合)．

以下のものは Pascal と同様の意味で使われる．

- *a := expression*
- **if** *condition* **then** *statement*

- **while** *condition* **do** *statement*

forall $x \in X$ **do** *statement* は個々の X の要素 x に対して statement が実行されることを意味する．

statement のリストは ; で区切られ，**begin** と **end** で囲まれる．

A.2 Message passing

“send” と “receive” が疑似コード上でメッセージの送受信を意味する．

一つのメッセージは *message type* と 0 個以上の *data field* からなる．例えば，**info** という message type が 2 つの data field をもつときは $\langle$**info**$, a, b\rangle$ というように表される．

“send” はメッセージとその宛先からなる．例えば以下のよう．

$$\text{send}\langle\textbf{info}, id_p, |Neigh_p|\rangle \text{ to } q$$

“shout” は隣接ノード全てに送信することを指す．つまり以下と等しい．

$$\textbf{forall} q \in Neigh_p \textbf{ do send } \langle\textbf{info}, id_p, |Neigh_p|\rangle \text{ to } q$$

“receive” は message type と data field と送信者からなる．例えば，

$$\text{receive}\langle\textbf{info}, a, b\rangle \text{ from } q$$

ならば，メッセージが受信されると，変数 a ，b ，q にそれぞれ値が代入される．

特定の process からメッセージを受け取りたいときは，以下のように記述する．

$$\text{receive}\langle\textbf{info}, a, b\rangle \text{ fromthis } q_0$$

A.3 Control-oriented vesus event-driven notation

アルゴリズムには以下の 2 通りの記述がある．

control-oriented 変数宣言と statement のリストからなる．local algorithm の execution は，この statement リストを一度実行すること (図 6 参照)．

event-driven *action* のリストからなる．action は *guard* と *body* からなり，guard が真ならば，body は実行されうる．gurad が真である限り，何度も body は実行することが可能 (図 7 参照)．

var $rec_p[q]$ for each $q \in Neigh_p$: boolean **init** false;

begin while # $\{q|rec_p[q]$ is false$\} > 1$ **do**
begin recevice $\langle$**tok**$\rangle$ from q; $rec_p[q] := true$ **end**;
send $\langle$**tok**$\rangle$ to q_0 with $rec_p[q_0]$is false;
receive $\langle$**tok**$\rangle$ from q_0 ; $rec_p[q_0] := true$;
decide
end

図 6: CONTROL-ORIENTED NOTATION (TREE ALGORITHM)

var $rec_p[q]$ for each $q \in Neigh_p$: boolean **init** false;
$sent_p, dec_p$: boolean **init** false;

$\mathbf{R}_p$: {A message $\langle$**tok**$\rangle$ has arrived }
begin receive $\langle$**tok**$\rangle$ from q; $rec_p[q] := true$ **end**

$\mathbf{S}_p$: { # $\{q|rec_p[q]$ is false$\} = 1$ and $sent_p = false$ }
begin send $\langle$**tok**$\rangle$ to q_0 with $rec_p[q_0]$ is false;
$sent_p := true$
end

$\mathbf{D}_p$: { # $\{q|rec_p[q]$is false$\} = 0$ and $dec_p = false$ }
begin $decide$; $dec_p := true$ **end**

図 7: EVENT-DRIVEN NOTATION (TREE ALGORITHM)