博 士 論 文

プロセス代数に基づくシステムレベル設計アーキテクチャ

指導教官　日比野 靖 教授

北陸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

岩政 幹人

2009 年 12 月

要 旨

LSI の設計フローにおいて，UML 等の上流工程の仕様書記述と，実現する LSI の設計書との間には，ギャップが存在する．特にシナリオ記述に基づく仕様書は動作の部分的な側面しか記述していない．このような断片的な仕様書から対象システムの全容を構成する手段が不足している．

本論文では，仕様書として，Message Sequence Chart(MSC) で，動作シナリオを記述し，その記述をプロセス代数に基づきモデル化することにより，プロセス代数の計算規則により，複数の動作シナリオを機械的にマージし，全体のシステム仕様を生成する手法を与える．並行プロセスの仕様記述を与える形式的手法であるプロセス代数を用い，本手法のために，プロトコル (個別の動作シナリオ) の並行結合を行う計算法を示した．

目 次

はじめに

ESL(Electric System Level) 設計における上流工程からの設計自動化の目標の 1 つとして，システムレベルの仕様書からシステムを自動的に生成することが挙げられている．例えば UML(Unified Modeling Language) では要求分析から仕様書の策定までを行うための仕様記述形式を提供している．しかし，シーケンス図等による振る舞い仕様記述はシステムの動作の一側面しか規定しないのでシステム全体としての振る舞いそのものを記述できない等のギャップがあった [19].

これらの課題に対して，動作の仕様記述としてメッセージシーケンス図 (MSC:Message Sequence Chart) を採用して，MSC で記述された仕様書のみから，最終システムを合成してシステム設計を行う手法 (eMSC システムと呼ぶ) が開発された [18]．eMSC システムでは MSC に階層性を導入し，下位の MSC(コマンド) で詳細なプロトコルを記述し，上位の MSC(シナリオ) にてシステムとしての振る舞いを記述する．シナリオ合成機能によりシナリオから，互いに連携しながら並列に動作する部分システムを状態遷移機械の集合として合成し，シナリオマージ機能にて複数のシナリオから全システム仕様を合成する．

eMSC システムでは，シナリオ合成機能，シナリオマージ機能の実現に個別対応している部分があり，形式化が十分でなく，状態遷移機械同士の連携プロトコルの正しさが検証されていなかった．

本論文では，プロセス代数に基づいたプロセスの並列結合法を導入し，シナリオ合成やシナリオマージを形式的に取り扱う手法を与え，実行順序制約の充足性を満たすという意味で正しい合成が保証されることを示す．

本論文の構成を説明する．1 章では eMSC システムを説明し，2 章にてプロセス代数を導入し，3 章にてプロセス代数による eMSC の形式化について述べる．4 章にて eMSC の合成処理の一部であるシナリオ合成の正しさをプロセス代数で検証する．5 章ではシナリオ合成結果を実装と結びつける方法について説明し，6 章ではシナリオマージについて説明する．最後に 7 章にて関連研究をレビューし 8 章にてまとめと展望について述べる．

第 1章

eMSCシステム

eMSC システム [18] は，プロトコル通信や演算処理を MSC 形式で表現したコマンド図を「部品」として，コマンド図を組み合わせてシステム動作の一局面を表現するシナリオ図，複数のシナリオ図をマージしたマージドシナリオ，マージドシナリオを組み合わせた全体システムであるトップシステムから構成される．図 1.1 は eMSC の階層を表したものである．コマンド (Command) はメッセージ (IRDY,TRDY) 通信をお行う MSC として定義され，シナリオ (Scenario) はコマンドをメッセージとして使う上位の MSC として定義される．複数のシナリオをマージしたものが Merged Scenario であり，これらを束ねた全体システムがトップシステム (Top System) と階層的に構成されている．

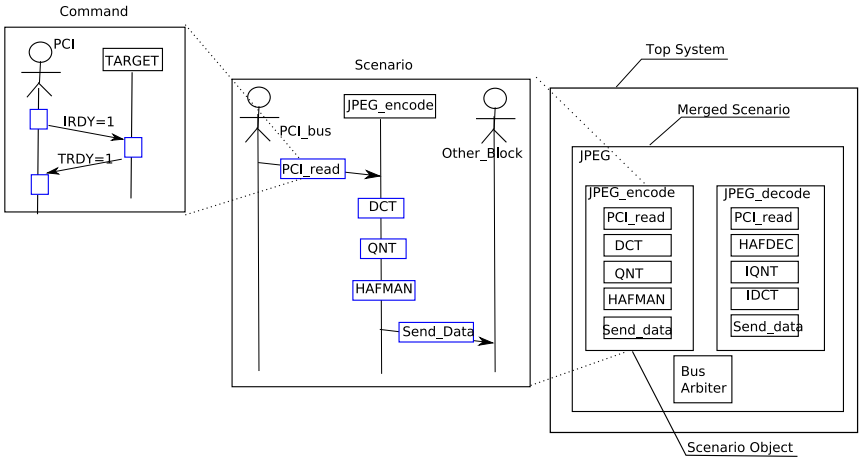


図 1.1: eMSC システム

1.1　コマンド図

コマンドは，状態遷移機械の仕様を規定する．コマンド図には，1 つの状態遷移機械の仕様を規定するプロセス内コマンドと，2 つの状態遷移機械とそれらの間の通信仕様を規定するプロセス間コマンドがある．

状態遷移機械間では，メッセージ通信によって同期をとる．すなわちクロック同期は行わないことを前提とする．メッセージ通信には，次の 2 つのタイプの通信チャネルを用意する．非同期の連携仕様を規定するための「待ちの有無 (blocking, non blocking) と「読み書きの区別 (read, write)」とを組み合わせた，非同期チャネルと，ランデブー型の同期チャネルとである．ここで blocking write とは，メッセージの送信完了まで書き込みを待つことであり，blocking read とは，メッセージの受信完了まで読み込みを待つことである．読み側が，blocking read 型で書き側が non blocking write の通信チャネル（以下 Ch_{br-nbw} と略す）は，コマンド図では，下線を付したフラグ型メッセージとして表現し，ポーリングを行わない通信チャネル（以下 $Ch_{nbr-nbw}$ と略す）は，下線のないメッセージで表す．

図 1.2 は，プロセス間コマンドの例である．コマンドの処理内容は，活性体 (Activity:図では矩形で示す) に記述する．活性体はコマンドオブジェクト (Object) に沿って配置され，活性体の間にはサイクル境界 (cycle boundary) がある．コマンドオブジェクトが状態遷移機械に対応し，活性体が状態に対応する．活性体の実行はコマンドの最上部からスタートし下方に向かって最下部に達したら再び最上部に戻って動作を繰り返す．

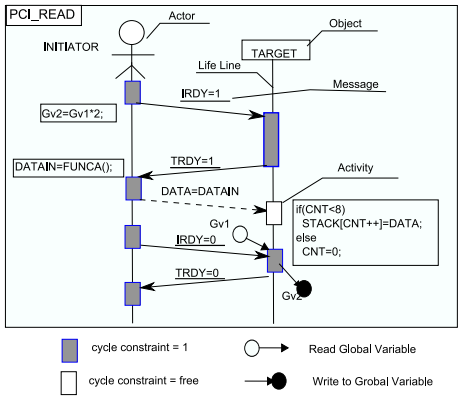


図 1.2: コマンド図

1.2　シナリオ図

シナリオは，MSC 形式で表現した動作順序に従ってコマンドが動作する仕様を規程する．シナリオ図はコマンド図を下位部品とし，シナリオオブジェクト配下に展開した上位階層の MSC として表現される．プロセス内コマンドは，1 つのコマンド活性体 (図では矩形で表現) として，プロセス間コマンドは，2 つのコマンド活性体を矢印で接続したものとして表される．コマンドは状態遷移機械の動作仕様を規定するが，シナリオ図は状態遷移機械同士を所定の順序で動作するように連係させる仕様を規定している．

個々のコマンド活性体は並列に動作する状態遷移機械であり，シナリオオブジェクト (ScenarioObject) は複数プロセスを束ねる中間階層に相当する．図 1.3 はシナリオ図の例である．

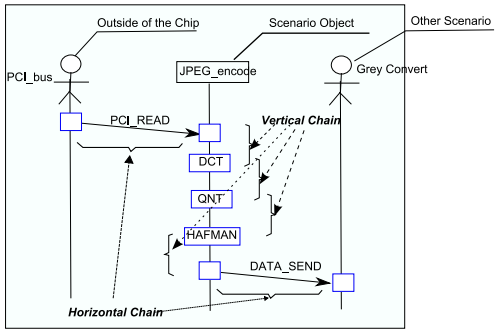


図 1.3: シナリオ図

シナリオ図が規定するコマンド活性体間の実行順序制約には 2 種類ある．同一のシナリオオブジェクト上に配置されるコマンド活性体は，配置の上下方向に沿って実行順序制約があり（縦チェーンと呼ぶ），プロセス間コマンドの送信・受信側のコマンド活性体間には，同期して動作する実行制約があるものとする（横チェーンと呼ぶ）．縦チェーン　(ψ_v)，横チェーン (ψ_h) は以下のように表される．

- ψ_v = ”Bn の実行は An の実行の後であり，An+1 の終了は Bn の実行を越えない”
- ψ_h = ”An の実行は Bn の実行を越えない，かつ Bn の実行は An の実行を越えない”

X と Y の間で横チェーン (ψ_h) を充足する順序制約は，n 回目および n+1 回目の実行における開始 (st_n, st_{n+1})，終了 (en_n, en_{n+1}) 間の制約で記述すると例えば以下にあげるようなものが挙げられる．

横チェーン 1 $st_n(X)$ は $st_n(Y)$ に先行し，$st_n(Y)$ は $st_{n+1}(X)$ に先行する

横チェーン 2 $en_n(X)$ は $st_{n+1}(Y)$ に先行し，$en_n(Y)$ は $st_{n+1}(X)$ に先行する

横チェーン 3 $st_n(X)$ は $st_n(Y)$ に先行し，$en_n(Y)$ は $en_n(Y)$ に先行する

横チェーン 4 $st_n(X)$ は $st_n(Y)$ に先行し，$en_n(Y)$ は $st_{n+1}(X)$ に先行する

一方 X と Y の間で縦チェーン (ψ_v) を充足する順序制約は，例えば以下のようなものが挙げられる．

縦チェーン 1 $en_n(X)$ は $st_n(Y)$ に先行し，$en_n(Y)$ は $en_{n+1}(X)$ に先行する

コマンド活性体 X の n 回目の繰り返し実行の最初の action を $st_n(X)$，最後の action を $en_n(X)$ と表し，“横チェーン 3” と” 縦チェーン 1” を順序制約として選択すると，コマンド活性体 A, B, C が図 1.4 のように連動するシナリオ図を，A, B, C 3 つの状態遷移機械の間に成立する順序制約は，次のようになる．

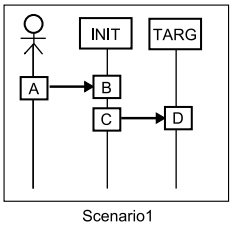


図 1.4: シナリオ図

横チェーン 3 $st_n(A)$ は $st_n(B)$ に先行し，$en_n(B)$ は $st_{n+1}(A)$ に先行する

縦チェーン 1 $en_n(B)$ は $st_n(C)$ に先行し，$en_n(C)$ は $en_{n+1}(B)$ に先行する

先行関係が半順序関係であるので推移律により，このシナリオ図の A, C の間に成立する連係制約は次のように計算できる．

- $st_n(A)$ は $st_n(C)$ に先行する．
- $st_n(C)$ は $st_{n+2}(A)$ に先行する．

シナリオを実行したイベントの軌跡を単位時間区切りを";" で表すと，図 1.4 のシナリオ図では以下の実行軌跡が可能である．

$st_1(A); st_1(B); st_2(A), en_1(B); st_2(B), st_1(C); ...$

この軌跡は 2 巡目の B の開始 $st_2(B)$ と 1 巡目の C の開始 $st_1(C)$ が同時刻に実行することが可能になることを示唆している．これは縦チェーン制約がシナリオのパイプライン動作（1 巡目と 2 巡目が同時に動作）を可能にしていることを示している．

1.3 シナリオ合成

シナリオ合成とは，シナリオ図におけるコマンド活性体をコマンド図が規定する個別の状態遷移機械として生成し，シナリオ図が規定する接続に合わせて，個々の状態遷移機械間に通信チャネルを，個々の状態遷移機械に連携プロトコル (チャネルへの読み書き) を挿入してシナリオ図が規定する仕様に沿った動作を行うひとつの状態遷移機械を生成する処理である．この状態遷移機械を、通信状態遷移機械 (CSFMs) と呼ぶ.

1.4 シナリオマージ

シナリオマージは，複数のシナリオ図からシナリオ合成して得られた CFSMs 群を，全体で 1 つの CFSMs としてマージする処理である．マージ処理は状態遷移機械の合併によって行われ，面積や通信帯幅のリソース制約を考慮した最適化を行う．マージ処理の制約は，もともとのシナリオ図で規定されている状態遷移機械間の動作連携の仕様が保存されていることである．

図 1.5 はシナリオマージの例である．

最初に 2 つのシナリオが単純にマージされる．最適化ステップにおいて CMD3 のコマンド活性体 E 以降を共有化する．

1.5 設計フロー

図 1.6 に，eMSC システムにおける設計フローを示す．eMSC システムではコマンド (Command)，シナリオ (Scenario)，マージシナリオ (Merged Scenario)，トップシステム (Top

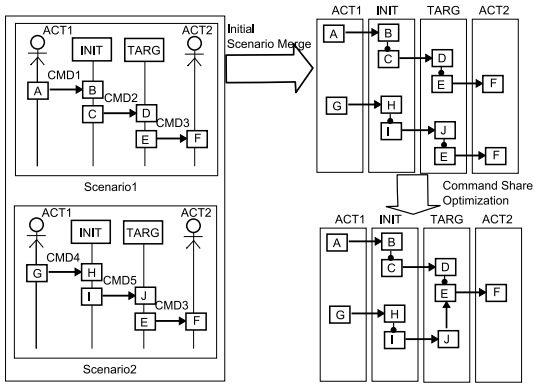


図 1.5: シナリオマージの例

System) に対応する設計エディタを備え，これらのエディタを順に用いることにより、コマンド設計 (Command Design)，シナリオ設計 (Scenario Design)，シナリオマージ (Scenario Merge)，トップ設計 (Top System Design) を行う．

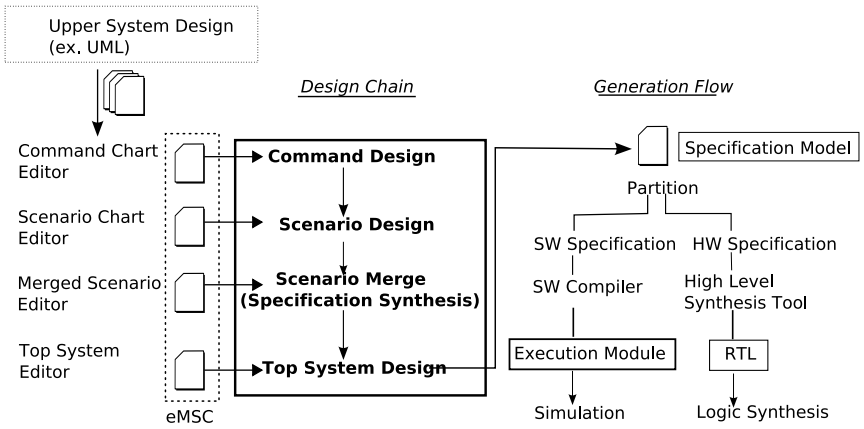


図 1.6: eMSC 設計フロー

トップ設計の結果である仕様モデル (Specification Model) は CFSM 形式でしゅつりょくされ，CFSM は LSI システム全体の仕様として ESL 言語形式に変換され，ESL 記述から高位合成ツールを利用して RTL 記述が生成される．eMSC システム [18] では，出力形式としてサイクル精度の SpecC 記述を採用している．

1.6　検証可能な設計技術

eMSC システムにおいては，シーケンス図形式の仕様書からボトムアップに LSI システムを合成している．本論文における形式化の目的は合成の正しさの検証であり，正しさを定義しこれを検証する手順について次章以降明らかにする．

一般的にモデル検査に代表される形式的手法に基づく検証においては，状態爆発という課題がある．特に互いに並列に動作する部品を組み合わせながら全体を検証する場合には，最悪構成部品の状態数の積で状態数が増え、検証処理が規模の増大に対してスケールしない．

eMSC の方式では、個々の機能部品が連動する部分制約にのみ着目して部分検証を行い、この検証結果が上位階層の設計においても保存されることを保証するだけですむという特徴がある．シーケンス図のみで仕様を記述するということで設計の自由度が限定されていると同時に検証の範囲も限定しているといえる．

すなわち eMSC の形式化は，検証可能な設計手法（設計自由度は制限されるが）を構築するアプローチの具体的な一例となる．

第 2章

プロセス代数

プロセス代数とは並列 (Parallel[1]) に動作するプロセスの動作を代数的に取り扱うための形式化の一つである [7, 8, 13, 4].

CCS(Calculus of Communicating Systems)[20] はプロセス間の同期通信は同じ名前の入出力アクションを介して行われる．同期通信の結果として 2 つのプロセスは 1 つのプロセスであるかのように振る舞う．本論文ではこれを同期合成と呼ぶ．同期合成した結果のプロセスにおいては同期したアクションは内部アクション τ に置き換えられる．一方 ACP(Algebra of Communication Proceses)[6, 8] はプロセス間では任意のアクションが同期動作可能でこれを明示的に強要することもできる，離散時間における ACP の拡張が ACP_{drt} である．

CCS,ACP,ACP_{drt} ともプロセス代数としての基本要素（同期合成，隠蔽，通信）は同じであるが，歴史的に CCS では LTS(ラベル付遷移システム) としての振る舞いのモデル化と検証に重きが置かれるのに対し，ACP では代数的な取り扱いに重きが置かれる．

本論文では，仕様としてのアクション間の順序制約 (シナリオ) に関しては CCS を用いて形式化を行い，CCS に基づくツール (`CWB-NC`)[1] を用いて検証を行う．一方より実装に近い動作仕様（コマンド）に関しては離散動作記述に ACP を拡張した ACP_{drt} を形式化に用いる．本章では ACP(2.1 節) およびその離散版である ACP_{drt}(2.2 節) について説明する．なお CCS の説明は付録 A.2 節にて行っている．

[1] ここでは「並行 (Concurrent)」を非同期な独立動作であるとして，何らかの手段（例えば同期通信）により同期して動作することを「並列 (Parallel)」と書き分けることにする

2.1 ACP

2.1.1 BPA

ACP では有限なプロセスは，atomic(原始的) なアクションの集合 A と演算子 $+$ および $\cdot$ から構成された閉じた (closed) 項により表現できる．そのような項を basic process term(プロセス項) と呼び basic process term の全ての集合を BPA(Basic Process Algebra) と呼ぶ．以下，原子アクションは子文字，それ以外のプロセス項は大文字で表す．BPA では操作的な意味が遷移システムとして定義される．atomic なアクション $a \in A$ は単独では $a \xrightarrow{a} \surd$ で表せる正常終了遷移を行う．b を別のアクションとすると,$a \cdot b$ は逐次実行, $a + b$ は選択実行を表す．BPA にデッドロック:δ を付与したものが BPA_δ である．デッドロック (δ) は空の動作を示す.

表 2.1 に BPA_δ の公理を示す.

A1	$x + y$	$=$	$y + x$
A2	$(x + y) + z$	$=$	$x + (y + z)$
A3	$x + x$	$=$	x
A4	$(x + y) \cdot z$	$=$	$x \cdot z + y \cdot z$
A5	$(x \cdot y) \cdot z$	$=$	$x \cdot (y \cdot z)$
A6	$x + \delta$	$=$	x
A7	$\delta \cdot x$	$=$	δ

表 2.1: BPA_δ の公理

通信による，状態 s, t の遷移の同時実行を定義するため通信関数 γ:$A \times A \to A$ を導入し，$\gamma(s, t)$ によりアクション s, t の同時実行を伴う通信アクションを表現する．また，この関数，すなわち通信により同時に遷移するアクションを構成する演算子を通信マージ ($|$) と呼ぶ．さらに左の遷移が先に動作する場合の演算子を左優先マージ ($\lfloor\!\lfloor$) と呼ぶ.

PAP(Process Algebra with Parallelism) は BPA にマージ演算 ($\|$)，通信マージ演算 ($|$)，左優先マージ演算 ($\lfloor\!\lfloor$) を追加して並列拡張したものである [20]．マージ演算子 ($\|$) は 2 つのプロセスの並列実行として組み合わせる．表 2.2 に PAP の公理を示す．**M1** はマージ, 通

M1	$x\|\|y$	$=$	$(x \between y + y \between x) + x\|y$
LM2	$v \between y$	$=$	$v \cdot y$
LM3	$(v \cdot x) \between y$	$=$	$v \cdot (x\|\|y)$
LM4	$(x + y) \between z$	$=$	$x \between z + y \between z$
CM5	$v\|w$	$=$	$\gamma(v, w)$
CM6	$v\|(w \cdot y)$	$=$	$\gamma(v, w) \cdot y$
CM7	$(v \cdot x)\|w$	$=$	$\gamma(v, w) \cdot x$
CM8	$(v \cdot x)\|(w \cdot y)$	$=$	$\gamma(v, w) \cdot (x\|\|y)$
CM9	$(x + y)\|z$	$=$	$x\|z + y\|z$
CM10	$x\|(y + z)$	$=$	$x\|y + x\|z$

表 2.2: PAP の公理

信マージ, 左優先マージ間の関係を表している．

PAP にさらに，encapsulation 演算子 (∂_H) を加えたものを ACP(Algebra of Communication Proceses) と呼ぶ．encapsulation 演算子 (∂_H)（ここで $H \subseteq A$）は H に含まれる全てのアクションを δ に置き換える (rename)．Encapsulation はアクションに通信を強要させる働きを持つ．例えば $\partial_{\{a,b\}}(a\|b)$ は $\partial_{\{a,b\}}(a \cdot b + b \cdot a + a \mid b)$ に展開されるが，単独の a, b 実行をそれぞれ δ に置き換えるので，結果として同時実行 $a \mid b$ しか実行できなくさせる．

2.1.2 再帰方程式

プロセス代数では再帰方程式 (Recursive Equation) によりループ構造を表現する [7].

再帰方程式は $E = \{X_1 = t_1(X_1, , , X_n), .., X_n = t_n(X_1, .., X_n)\}$ という形式で記述される．X_k を再帰変数と呼び，t_i は X_j による多項式である．再帰方程式 E の解 (Solution) であるプロセスのクラスを再帰変数 X に対応させて $\langle X|E\rangle$ と表す．例えば再帰方程式 $E = \{X = aY, Y = bX\}$ の解において $\langle X|E\rangle$ はプロセス *ababab...* に対応し，$\langle Y|E\rangle$ は *bababa...* に対応する．

再帰方程式 E の解は $\langle X_k|E\rangle$ をノードにした，グラフとして表現できる．例えば $\{s_0 \xrightarrow{a} s_0, s_0 \xrightarrow{b} s_1, s_1 \xrightarrow{c} s_0, s_1 \xrightarrow{a} s_1\}$ であるような，システムと双模倣である線形再帰方程式 E

は，$E = \{X = aX + bY, Y = cX + aY\}$ である．再帰方程式のプロセスグラフは図 2.1 に示す $\langle X|E\rangle$ と $\langle Y|E\rangle$ をノードとする遷移グラフとして表現できる．

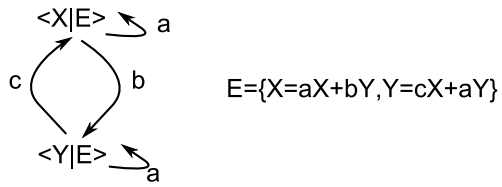


図 2.1: 再帰プロセスのグラフ

2.1.3 再帰方程式の並列結合計算

2 つの状態遷移機械 A, B が再帰変数 X, Y による再帰方程式 E で表現されるとする．

$$\begin{aligned} E &= \{ \\ &\quad X = a \cdot bX \\ &\quad Y = c \cdot dY \\ &\quad \} \end{aligned} \tag{2.1}$$

A と B を組合せ結合した状態遷移機械 C を再帰方程式の展開で計算できる．ここでは A, B が独立並列に動作するものとする．

X, Y を結合した全体のシステムの動作における組合せ状態に対応する変数を Z とすると，Z を再帰変数 X, Y の並列結合演算 $(Z = (X \parallel Y))$ で表すことができる．ここでは再帰変数の並列結合演算とプロセス項のマージ演算を同じ演算記号 $(\parallel)$ で記述している．

組合せ再帰変数 Z に関連する，新しい再帰変数と状態遷移の計算を行うため $Z = (X \parallel Y)$ に対してプロセス項の展開を行う．

式 (2.1) の再帰方程式 E から全体の動作を表す再帰変数 Z_1 は以下のように展開できる．

$$\begin{aligned} Z_1 &= X \parallel Y = X \between Y + Y \between X + X \mid Y \\ &= (a \cdot bX) \parallel Y + (c \cdot dY) \between X + (a \cdot bX) \mid (c \cdot dY) \\ &= a \cdot (bX \parallel Y) + c \cdot (dY \parallel X) + (a \mid c)(bX \parallel dY) \end{aligned}$$

ここで $Z_2 = (bX \parallel Y), Z_3 = (dY \parallel X), Z_4 = (bX \parallel dY)$ としてそれぞれ展開を行うと以下の再帰変数 (Z_1, Z_2, Z_3, Z_4) による再帰方程式を得ることができる．

$$
\begin{aligned}
Z_1 &= aZ_2 + cZ_3 + (a \mid c)Z_4 && (2.2)\\
Z_2 &= b \cdot (X \parallel Y) + c \cdot (bX \parallel dY) + (b \mid c)(X \parallel dY) \\
&= bZ_1 + cZ_4 + (b \mid c)Z_3 && (2.3)\\
Z_3 &= d \cdot (Y \parallel X) + a \cdot (dY \parallel bX) + (d \mid a)(Y \parallel bX) \\
&= dZ_1 + aZ_4 + (d \mid a)Z_2 && (2.4)\\
Z_4 &= b \cdot (X \parallel dY) + d \cdot (bX \parallel Y) + (b \mid d)(X \parallel Y) \\
&= bZ_3 + dZ_2 + (b \mid d)Z_1 && (2.5)
\end{aligned}
$$

得られた再帰方程式は並列結合演算の結果において可能な動作を全て含んでいる．

2.2 離散時間プロセス代数 ACP_{drt}

ACP_{drt}[5] は離散時間に ACP を拡張した体系である．

ACP_{drt} の拡張部分は，immediate deadlock:$\dot{\delta}$，遅延しないアクション:$cts(a)$，単位遅延演算:$\sigma_{rel(1)}$ または σ_d，により構成される．

$\dot{\delta}$ は原子項以外のプロセス項にも演算が可能で，原子項に対する δ に相当する．また $\sigma_d(X)$ は単位遅延時間後に X が実行されることを表している．

表 2.3 は ACP_{drt} の公理の一部を抜粋したもので，**DRT2** は σ_d が最初のアクションに実質的に作用することを示している．例えば下記のプロセス項の書換えが可能である．

$$e1 \cdot \sigma_d(e2 \cdot \sigma_d(e3)) = e1 \cdot \sigma_d(e2) \cdot \sigma_d(e3)$$

$\alpha \cdot \sigma_d(\beta)$ が，今の時間で α が実行された後単位時間後に β が実行されることを意味するからアクションの実行を遷移関係に読み替えると，$e1 \cdot \sigma_d(e2) \cdot \sigma_d(e3).0$ は

$$e1 \cdot \sigma_d(e2) \cdot \sigma_d(e3).0 \xrightarrow{e1} \sigma_d(e2) \cdot \sigma_d(e3).0 \xrightarrow{e2} \sigma_d(e3).0 \xrightarrow{e3} 0$$

LMID1	$\dot{\delta} \between X = \dot{\delta}$
LMID2	$X \between \dot{\delta} = \dot{\delta}$
A6ID	$X + \dot{\delta} = X$
A7ID	$\dot{\delta} \cdot X = \dot{\delta}$
CMID1	$\dot{\delta} \mid X = \dot{\delta}$
CMID2	$X \mid \dot{\delta} = \dot{\delta}$
DRT1	$\sigma_d(X) + \sigma_d(Y) = \sigma_d(X + Y)$
DRT2	$\sigma_d(X) \cdot Y = \sigma_d(X \cdot Y)$
DRT3	$\sigma_d(\dot{\delta}) = cts(\delta)$
ADRT	$a = cts(a) + \sigma_d(a)$
DRT4	$cts(a) + cts(\delta) = cts(a)$

表 2.3: ACP_{drt} の公理の一部

という遷移の連鎖であるとみなせる．このように ACP_{drt} は離散時間で遷移が進行するシステムの動作を記述できる．

以下の説明では，イベントはすべて単位時間内で実行される遅延無きアクション $cts(ev)$ であるとして記号”cts” を省略する．

第 3章

プロセス代数によるeMSCの形式化

eMSC システムにおけるプロセスの合成をプロセス代数にて取り扱うために，コマンド，シナリオの 2 階層それぞれをプロセス代数で形式化する．

コマンドは，メッセージ送信･受信をイベントとして捕らえ，MSC をイベント間の依存関係集合とする Alure ら [3] の形式化とプロセス代数体系 ACP を離散時間に拡張する ACP_{drt} の記法に従い形式化する．一方，シナリオは，CCS をベースに主に順序制約に関する性質と操作を形式化する．CCS の簡単な紹介は付録 A.2 章を参照のこと．

以下，コマンド，シナリオを異なるプロセス代数体系を用いて形式化しているが，CCS は抽象度の高い順序関係にのみ着目した部分，ACP_{drt} はサイクルの区切りを意識し，具体的な並列実行を考慮した部分に着目した形式化であり，それぞれの形式化にそった設計・検証が議論される．両者は，上位階層の仕様と下位階層の実装という関係で第 5 章のシナリオ合成で関係づけられる．なお並列結合演算は ACP では ||，CCS では | として記述される．ACP で | は同期実行演算を意味するので注意.

図 3.1 は eMSC とプロセス代数による形式化を模式的に表した図である．

上段が，CCS による型式化であり，下段が ACP_{drt} による型式化である．前者はラベル付き遷移システム LTS(3.2.1) で、後者は、ラベル付けされた 半順序構造 (3.1.1) で形式化される．コマンドは，ACP_{drt} に従ってラベル付けされた半順序構造で形式化される．図では `CMD0` として表されいる。`CMD0` を順序関係のみに着目すると、CCS のプロセス LTS0 を得る．

一方シナリオから順序制約 ψ(3.2.4 節) および ∇ 演算 (3.4 節) が得られ，LTS0 に ψ に対する充足判定を行うとこれを充足する最大部分モデル LTS1 が得られる (3.3 節)．一方 ∇_{OW} 演

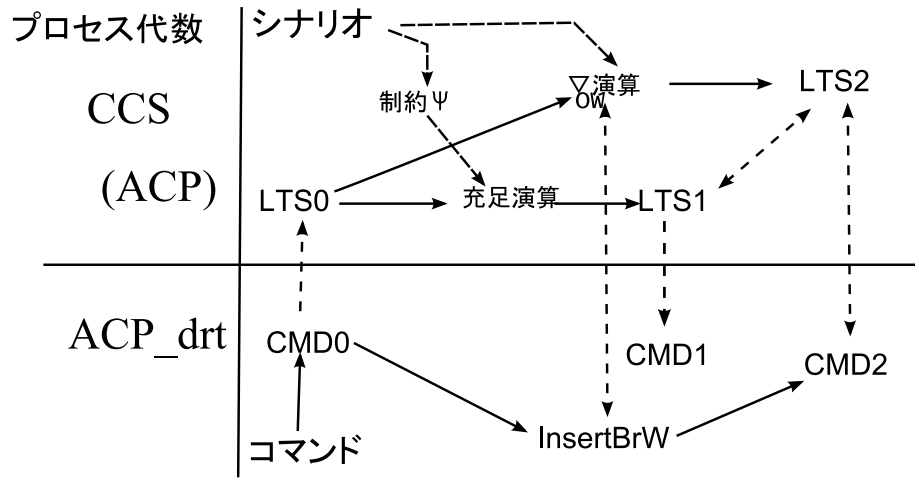


図 3.1: プロセス代数による形式化

算を LTS0 に適用すると LTS2 が得られる．LTS2 と LTS1 の関係は 3.5 節で議論する．ACP における順序制約 ψ および ∇ 演算に関する議論は 3.8 節にて行う．LTS1 を実装したものが CMD1 である (5.1 節)．また ∇_{OW} 演算は ACP_{drt} においては LTS に対する `InsertBrW` 操作を施すことに対応し，操作後の CMD2 は LTS2 の実装になっている (5.2 節)．

3.1 コマンドの形式的定義

3.1.1 コマンドの形式化

まず，eMSC を構造 $(\mathcal{P}, M, Act, \{\sigma_d\}, \Sigma)$ にて定義する．

ここで，$\mathcal{P} = P_1, .., P_n$ はプロセスの集合，M はメッセージの集合，Act はアクションの集合，Σ は文字列の集合であるとする．また σ_n は，サイクルの n 番目の切れ目であるとする．

プロセス $\mathcal{P}$ 間は，1 対 1 の通信を行う信頼できるバッファ長=1 の FIFO を介したメッセージ $(m \in M)$ を送受信する．個々のプロセスはメッセージの送受信に関わるアクションの他に，通信を伴わないアクションを内部アクションとして行う．

$\Sigma_p \subset \Sigma$ をプロセス $p \in \mathcal{P}$ により実行されるアクションや通信を表す文字列の集合とする．

Σ_p は $\langle p!q, m\rangle, \langle p?q, m\rangle, \langle p, a\rangle$ という形式で表せるアクションで構成される．$\langle p!q, m\rangle$ はプロセス p が q に対して，メッセージ $m \in M$ を送信するアクション，$\langle p?q, m\rangle$ はプロセス

p が q からメッセージ m を受信するアクション，また $\langle p, a\rangle$ は，プロセス p の内部アクションを表すものとする．

MSC である $(\mathcal{P}, M, Act, \{\sigma_d\}, \Sigma)$ を Σ でラベル付けされた半順序集合に基づく構造 $Ch = (E, \leq, \lambda)$ として形式化する．

ここで，イベント E は，$E = Act \cup \{\sigma_n\}$ ，$(E, \leq)$ は E 上の半順序関係 $\leq$ による半順序集合であり，$\lambda : E \to \Sigma$ はラベル付け関数である．

$X \subset E$ に対して，$\downarrow X = \{e' | e' \leq e, e \in X\}$ とする．

$p \in \mathcal{P}$ に対して，$E_p = \{e | \lambda(e) \in \Sigma_p\}$ と表す．同様に，

$$E_{p!q} = \{e | \exists m. e \in E_p \text{かつ} \lambda(e) = \langle p!q, m\rangle, m \in M\}$$

$$E_{p?q} = \{e | \exists m. e \in E_p \text{かつ} \lambda(e) = \langle p?q, m\rangle, m \in M\}$$

また $\lambda(e) = \langle p!q, m\rangle, \lambda(e') = \langle q?p, m\rangle$ かつ $|\downarrow(\{e\}) \cap E_{p!q}| = |\downarrow(\{e'\}) \cap E_{q?p}|$ であるときに通信関係 $(e, e') \in R$ があるとする．

コマンドは $(\mathcal{P}, M, Act)$ 上，Σ でラベル付けされた半順序集合に基づく構造 $Ch = (E, \leq, \lambda)$ であり以下を充たす

1. $\leq_p$ は単一プロセス $p \in \mathcal{P}$ の線形順序であり，$\leq$ を $E_p \times E_p$ に限定したものである
2. $\lambda(e) = \langle p?q, m\rangle$ であるとき $|\downarrow(e) \cap E_{p?q}| = |\downarrow(e) \cap E_{q!p}|$ であり，また $e' \in \downarrow(e)$ が存在し，$|\downarrow(e) \cap E_{q!p}| = |\downarrow(e') \cap E_{q!p}|$
3. $p \neq q$ に対して $|E_{p?q}| = |E_{q!p}|$ である
4. すなわち行き先，出元のないメッセージはない
5. $\lambda(\sigma_n) = \sigma_n$

従って eMSC は，半順序関係 $\leq$ により順序付けされた Σ でラベル付けされたイベントの集合 E として形式化できる．

特に eMSC においては 1 本のメッセージ線 (var=value) は，送信イベント $\langle p!q, \text{var=value}\rangle$ および受信イベント $\langle p?p_i, \text{var=value}\rangle$ に対応する．またポーリングに関する受信イベントを区別するためにイベント記号に下線を付与する ($\underline{ev}$)．

図 1.2 におけるアクター側を p，TARGET 側のプロセスを q，とすると TARGET 側 (q) に関わるイベントに関する形式化を行うと，活性体の切れ目を離散時間の切れ目 $\sigma_0, \sigma_1, \sigma_2$ とすると，以下を得る．また TARGET 側の 3 つの活性化で実行される内部アクションを上からそれぞれ a, b, c と置く．

$$
\begin{aligned}
&E_q = \{\underline{e1}, e2, e3, \underline{e4}, e5, e_a, e_b, e_c, \sigma_0, \sigma_1, \sigma_2\},\\
&\lambda(\underline{e1}) = \underline{\langle q?p, \mathrm{IRDY{=}1}\rangle},\\
&\lambda(e2) = \langle q!p, \mathrm{TRDY{=}1}\rangle,\\
&\lambda(e3) = \langle q?p, \mathrm{DATA{=}DATAIN}\rangle,\\
&\lambda(\underline{e4}) = \underline{\langle q?p, \mathrm{IRDY{=}0}\rangle},\\
&\lambda(e5) = \langle q!p, \mathrm{TRDY{=}0}\rangle,\\
&\lambda(e_a) = \langle q, a\rangle,\\
&\lambda(e_b) = \langle q, b\rangle,\\
&\lambda(e_c) = \langle q, c\rangle,\\
&\leq_q = \{(\sigma_0, \underline{e1}), (\underline{e1}, e_a), (e_a, e2), (e2, \sigma_1), (\sigma_1, e3), (e3, e_b), (e_b, \sigma_2),\\
&\quad (\sigma_2, \underline{e4}), (\underline{e4}, e_c), (e_c, e5)\}
\end{aligned}
$$

3.1.2 通信の形式化

eMSC ではメッセージ通信は，チャネルを介した読み書きのプロトコルとして実現される．ここでは，深さ 1 のバッファを持つ FIFO 型のチャネルと，このチャネルに対する読み書きのアクションを導入する．本形式化では，チャネルを介して通信されるデータに関する情報は捨象される．

定義 3.1.1 (チャネル $ch^1(Id, B)$) チャネル $ch^1(Id, B)$ は，バッファ長=1 のバッファ付きの通信チャネルで，Id はチャネル固有の番号を表し，B はバッファの初期値 $\in \{0, 1\}$ を表すものとする

読み書きのアクションは，$n = Id$ として，ブロッキング型読み込み:$br_n/1$，non ブロッキング型書き込み:$w_n/1$ の 2 種類を用意する[1]．ここで $fun/1$ は関数 fun が 1 引数関数であることを表している．直感的には $br_n(\alpha)$ はチャネル n のバッファが空でない場合に α を

[1] eMSC ではブロッキング型書き込み=$bw_n/1$ は無い

実行する関数，$w_n(\alpha)$ はチャネル n に対してバッファの内容にかかわらず α を実行しバッファ値を 1 にする働きを行う．

図 3.2 は $br_n/1, w_n/1$ の操作的意味を定義している．プロセス項とチャネル状態の組合せを状態:$\langle$ プロセス項, 現チャネル状態, 次チャネル状態の更新 $\rangle$ として表している．次チャネル状態とは離散単位時間 σ_d 経過後のチャネル状態を表している．ここで α, β は原子アクションを S, T はプロセス項であるとする．また τ は外から観測不能な内部アクションであるとする．

$$\text{ATOM: } \frac{}{\langle \sigma_d(\alpha) \cdot P, Cs, Us\rangle \xrightarrow{\alpha} \langle P, Us \to Cs, \{\}\rangle}$$

$$\text{BR1: } \frac{(i,1) \in Cs}{\langle \sigma_d(br_i(S)) \cdot P, Cs, Us\rangle \xrightarrow{\tau} \langle \sigma_d(S) \cdot P, Cs, (i,0) \to Us\rangle}$$

$$\text{W1: } \frac{}{\langle \sigma_d(w_i(S)) \cdot P, Cs, Us\rangle \xrightarrow{\tau} \langle \sigma_d(S) \cdot P, Cs, (i,1) \to Us\rangle}$$

図 3.2: チャネル $ch^1(i,j)$ と読み書き関数の動作の操作的意味定義

ATOM は $br_n/1, w_n/1$ いずれにも束縛されない原子アクションの動作を規定し，α 遷移の結果，次チャネル状態更新の値が現在チャネル状態に反映され $(Us \to Cs)$，次チャネル状態更新がクリアされる．

BR1 は $br_n/1$ が現チャネル状態 Cs においてチャネル i の値が 1 の時にのみ τ 遷移し次チャネル状態の更新 Us に新しい値 0 を登録すること，W1: はそれぞれ $w_n/1$ が次チャネル状態の更新 Us に新しい値 1 を現在チャネル値にかかわらず登録するという動作を表している．

また eMSC では，non ブロッキング型読み込み:$r_n/1$ を用いるが，以下の形式化では不要なので省略する．

3.1.3 プロセス代数へのマッピング

形式化されたコマンドを以下のルールにてプロセス項に変換する

- プロセス p に対するイベントを集める E_p
- E_p に対して半順序関係 le_p に従ってイベントを整列した列を得る

- E_p に対する Σ_p の列を得る
- $\langle p!q, m\rangle$ を $w_c/1$ に書き換える
- $\langle p?q, m\rangle$ を $br_c/1$ に書き換える
- $\langle p, x\rangle$ を x に書き換える
- 列において le_p を $(\cdot)$ に書き換えて結合する
- $\sigma_n \cdot P_t$ を $\sigma_d(P_t)$ に書き換える (P_t はプロセス項)
- $w_c/1, br_c/1$ の引数を決める

図 1.2 の例では，イベント列は

$$\{\sigma_0, \langle p, a\rangle, \langle p!q, m1\rangle, \sigma_1, \langle p, c\rangle, \sigma_2, \langle q?p, m2\rangle, \langle p, d\rangle\}$$

半順序関係 $\leq$ で整列してこれを $\lambda(x)$ にて Σ_p の列に変換すると以下の列を得る

$$\Sigma_p^+ = \sigma_0 \leq \langle p, a\rangle \leq \langle p!q, m1\rangle \leq \sigma_1 \leq \langle p, c\rangle \leq \sigma_2 \leq \langle q?p, m2\rangle \leq \langle p, d\rangle$$

$\langle p!q, m\rangle$，$\langle p?q, m\rangle$ を w_c，br_c，に書換え，$\leq_p$ を $(\cdot)$ に書き換えて結合すると以下を得る．

$$P = \sigma_0 \cdot a \cdot w_{c1} \cdot \sigma_1 \cdot c \cdot \sigma_2 \cdot br_{c2} \cdot d$$

さらに σ_i を $\sigma_d()$ に変換すると以下のプロセス項 P を得る．

$$P = \sigma_d(a \cdot w_{c1})) \cdot \sigma_d(c) \cdot \sigma_d(br_{c2} \cdot d)$$

最後に $w_c/1, br_c/1$ の引数を決めると以下のプロセス項 P を得る．

$$P = \sigma_d(w_{c1}(a)) \cdot \sigma_d(c) \cdot \sigma_d(br_{c2}(d))$$

3.1.4 並列結合の動作意味定義

eMSC では離散時間の切れ目 (σ_i) に挟まれる区間を実行単位として同期して動作する．並列結合演算 $(\|)$ は以下の制限を受ける．

$$(\sigma_d(a)X \parallel \sigma_d(b)Y) = \sigma_d(a \mid b)(X \parallel Y)$$

すなわちアクション a, b の並列実行で可能な組合せ $\{a \cdot b, b \cdot a, a \mid b\}$ のうち同時実行 $(a \mid b)$ のみが実行される．

$br_n : Proc \to Proc, w_n : Proc \to Proc$ はプロセス項 $(Proc)$ を引数にとって新たなプロセス項 $(Proc)$ を生成する関数として形式化される．br_n，w_n は双方とも，原子項 α，原子項の並列結合 $(\alpha \mid \beta)$，および $w_k(Proc), \{$ ここで $n \neq k\}$ を引数に取ることができる．

$br_n(\alpha), w_n(\beta)$ のチャネル状態 $ch^1(i,j)$ に対する動作意味をプロセス項とチャネル状態の組合せを状態:$\langle$ プロセス項, チャネル状態, 次チャネル状態の更新 $\rangle$ として表して以下の図 3.3 の様に定義する．ここでチャネル状態は $\{$(チャネル Id, バッファ値), ...$\}$ で表す．

$$\text{PAR: } \frac{}{\langle(\sigma_d(\alpha)\cdot P) \parallel (\sigma_d(\beta)\cdot Q), Cs, Us\rangle \xrightarrow{\alpha|\beta} \langle P \parallel Q, Us \to Cs, \{\}\rangle}$$

$$\text{PBR1: } \frac{(i,1) \in Cs, (i,k) \in Us}{\langle(\sigma_d(br_i(S))\cdot P \parallel Q), Cs, Us\rangle \xrightarrow{\tau} \langle(\sigma_d(S)\cdot P \parallel Q), Cs, Us\rangle}$$

$$\text{PBR2: } \frac{(i,1) \in Cs, (i,k) \notin Us}{\langle(\sigma_d(br_i(S))\cdot P \parallel Q), Cs, Us\rangle \xrightarrow{\tau} \langle(\sigma_d(S)\cdot P \parallel Q), Cs, (i,0) \to Us\rangle}$$

$$\text{PBR3: } \frac{(i,0) \in Cs}{\langle(\sigma_d(br_i(S))\cdot P \parallel \sigma_d(\alpha)\cdot Q), Cs, Us\rangle \xrightarrow{\alpha} \langle(\sigma_d(S)\cdot P \parallel Q), Us \to Cs, \{\}\rangle}$$

$$\text{PW1: } \frac{}{\langle(\sigma_d(w_i(S))\cdot P \parallel Q), Cs, Us\rangle \xrightarrow{\tau} \langle(\sigma_d(S)\cdot P \parallel Q), Cs, (i,1) \to Us\rangle}$$

図 3.3: チャネル $ch^1(i,j)$ と並列結合の動作の操作的意味定義

PAR は並列実行 $(\sigma_d(\alpha)X \parallel \sigma_d(\beta)Y)$ の動作を表している．

PBR1 は，チャネル i の現在値が 1(full であるということ) であり，状態更新 Us にチャネル i に関する登録がある場合には $br_i/1$ が実行され，τ 遷移が発生する．

PBR2 は，チャネル i の現在値が 1(full であるということ) であり，状態更新 Us にチャネル i に関する登録がない場合には $br_i/1$ が実行され，τ 遷移の結果，次チャネル状態更新の値が 0 に設定される．

PBR3 は，チャネル i の現在値が 0(empty であるということ) である場合の原子アクションの遷移に関するルールである．α 遷移の結果，次チャネル状態更新の値が現在チャネル状態に反映され $(Us \rightarrow Cs)$，次チャネル状態更新がクリアされる．

PW1 は, チャネル i の現在値にかかわらず，$w_i/1$ は次チャネル状態値を 1(full であるということ) に設定することを示している．

チャネル i に対して読み書きが同時に発生する場合は $(\langle \sigma_d(br_1(\alpha)) \| \sigma_d(w_1(\beta)), \{(1,1)\}, \{\}\rangle)$ は，

PBR2 → PW1 の順にルールを適用して得られる結果と，

$$\begin{aligned}
&\langle \sigma_d(br_1(\alpha)) \parallel \sigma_d(w_1(\beta)), \{(1,1)\}, \{\}\rangle \\
&\xrightarrow{\tau} \langle \sigma_d(\alpha) \parallel \sigma_d(w_1(\beta)), \{(1,1)\}, \{(1,0)\}\rangle \\
&\xrightarrow{\tau} \langle \sigma_d(\alpha) \parallel \sigma_d(\beta), \{(1,1)\}, \{(1,1)\}\rangle \\
&\xrightarrow{\alpha|\beta} \langle , \{(1,1)\}, \{\}\rangle
\end{aligned}$$

PW1 → PBR1 の順にルールを適用して得られる結果は同じである (合流性がある)．

$$\begin{aligned}
&\langle \sigma_d(br_1(\alpha)) \parallel \sigma_d(w_1(\beta)), \{(1,1)\}, \{\}\rangle \\
&\xrightarrow{\tau} \langle \sigma_d(br_1(\alpha)) \parallel \sigma_d(\beta), \{(1,1)\}, \{(1,1)\}\rangle \\
&\xrightarrow{\tau} \langle \sigma_d(\alpha) \parallel \sigma_d(\beta), \{(1,1)\}, \{(1,1)\}\rangle \\
&\xrightarrow{\alpha|\beta} \langle , \{(1,1)\}, \{\}\rangle
\end{aligned}$$

また 2 つのチャネルをクロスする例を以下に示す．

$$\begin{aligned}
P1 &= \sigma_d(w_1(a)) \cdot \sigma_d(c) \cdot \sigma_d(br_2(d)) \\
P2 &= \sigma_d(br_1(w_2(b)))
\end{aligned}$$

に対して $\langle P1 \parallel P2, \{(1,0),(2,0)\}, \{\}\rangle$ を計算すると

$\langle P1 \parallel P2, \{(1,0),(2,0)\}, \{\}\rangle$

$\xrightarrow{\tau} \langle \sigma_d(a)\cdot\sigma_d(c)\cdot\sigma_d(br_2(d)) \parallel \sigma_d(br_1(w_2(b)), \{(1,0),(2,0)\}, \{(1,1)\}\rangle \ldots$ (PW1 より)

$\xrightarrow{a} \langle \sigma_d(c)\cdot\sigma_d(br_2(d)) \parallel \sigma_d(br_1(w_2(b)), \{(1,1),(2,0)\}, \{\}\rangle \ldots$ (PBR3 より)

$\xrightarrow{\tau} \langle \sigma_d(c)\cdot\sigma_d(br_2(d)) \parallel \sigma_d(w_2(b)), \{(1,1),(2,0)\}, \{(1,0)\}\rangle \ldots$ (PBR2 より)

$\xrightarrow{\tau} \langle \sigma_d(c)\cdot\sigma_d(br_2(d)) \parallel \sigma_d(b), \{(1,1),(2,0)\}, \{(1,0),(2,1)\}\rangle \ldots$ (PW1 より)

$\xrightarrow{c|b} \langle \sigma_d(br_2(d)), \{(1,0),(2,1)\}, \{\}\rangle \ldots$ (PAR より)

$\xrightarrow{\tau} \langle \sigma_d(d), \{(1,0),(2,1)\}, \{(2,0)\}\rangle \ldots$ (PBR2 より)

$\xrightarrow{d} \langle , \{(1,0),(2,0)\}, \{\}\rangle \ldots$ (ATOM より)

を得る．これは，チャネル 1,2 をつかったブロック読み込み型のプロトコルにより b と c の同時実行が可能になったことを示している．

3.2　シナリオの形式的定義

ここではプロセス代数における充足性の定義を導入し，シナリオの形式化を行う．本章でのプロセス代数は CCS の記述方法をベースとして，ACP の encapsulation 演算 (∂_H)，rename 演算 (ρ_S) を組み入れた独自のものを採用している．

以下の充足性の定義とシナリオ形式化の議論においては，通信を伴わない同期実行 ($act1|act2$) を取り扱わないが，これは 3.8 節にて補足する．また前章のコマンドの形式化において導入した，バッファ長=1 の FIFO チャネルを介した通信は，同期実行を伴わない（チャネルを介した同期が離散時間の区切りをまたぐ，すなわちある離散時間区切り σ_1 で実行された書き込みは，少なくとも次の時間区切り σ_2 でないと反映されない）ので，同様に考慮しない．

3.2.1　ラベル付き遷移システム (LTS)

最初に，プロセス代数記法の一つである CCS に倣いモデルやプロパティを，ラベル付き遷移システム (LTS) を，次に示すラベル付き遷移システム LTS(Labeled Transition System) で表す。

$$(Proc, Act, \{\xrightarrow{\alpha} | \alpha \in Act\})$$

ここで $Proc$ はプロセスの集合，Act はアクションの集合，$\{\xrightarrow{\alpha}\}$ はアクション $\alpha \in Act$ に伴う遷移関係の集合であるとする．

A はチャネル名 (name) の集合で $\overline{A}$ を補名 (co-name) の集合であるとする．$\overline{\overline{a}} = a$ である．

$$\overline{A} = \{\overline{a} | a \in A\}$$

ラベル L はチャネル名と補名の和集合である．

$$L = A \cup \overline{A}$$

アクション Act は，ラベル $\mathcal{L}$ と観測不能な内部アクションである τ で構成される

$$Act = L \cup \{\tau\}$$

$$\frac{P \xrightarrow{\alpha} P'}{K \xrightarrow{\alpha} P'} \; K \stackrel{\text{def}}{=} P \qquad \frac{}{\alpha \cdot P \xrightarrow{a} P}$$

$$\frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \qquad \frac{Q \xrightarrow{\alpha} Q'}{P \mid Q \xrightarrow{\alpha} Q' \mid Q}$$

$$\frac{P \xrightarrow{\alpha} P'}{\partial_L(P) \xrightarrow{\alpha} \partial_L(P')} \; \alpha, \overline{\alpha} \notin L \qquad \frac{P \xrightarrow{\alpha} P' \; Q \xrightarrow{\overline{\alpha}} Q'}{\partial_L(P \mid Q) \xrightarrow{\alpha \mid \overline{\alpha}} \partial_L(P' \mid Q')} \; \alpha, \overline{\alpha} \in L$$

$$\frac{P \xrightarrow{\alpha} P'}{P[f] \xrightarrow{f(\alpha)} P'[f]} \qquad \frac{P_j \xrightarrow{\alpha} P'_j}{\Sigma_{i \in I} P_i \xrightarrow{\alpha} P'_j} \; j \in I$$

図 3.4: LTS の展開ルール

また 0 は特別なプロセスで，それ以上遷移がない終端プロセス (0) であるとする．

プロセス式は以下の演算要素にて構成される

$$P, Q := K \mid \alpha \cdot P \mid P \mid Q \mid P + Q \mid \partial_L(P) \mid P[f]$$

ここで K はプロセス名を表す定数である. 演算要素の操作的な意味を図 3.4 のように定義する．

ここで，$\cdot$ は逐次実行結合，$\mid$ は並列実行結合演算であり，$\Sigma_{i \in \{1,2\}} P_i$ は $P_1 + P_2$ の簡略記法であるとする．

さらに，

1. プロセス定義式は $K \stackrel{def}{=} P$ としてプロセスを（再帰的に）定義する記法である．

2. ∂_L 演算は，L に含まれるアクション α に対して単独での $\alpha, \overline{\alpha} \in L$ の実行を禁止する演算である．α と $\overline{\alpha}$ アクションの同時実行 $(\alpha \mid \overline{\alpha})$ は禁止しないので結果として $\alpha \mid \overline{\alpha}$ のみを許可する．

3. rename 演算 $(P[f])$ は P における α 遷移を $f(\alpha)$ 遷移に書換える．以降便宜的に $f = \alpha/\beta$ により α を β に置き換える関数を表すものとする．

CCS における通信を伴う同期遷移 (ランデブー) は ∂ 演算と rename 演算を組み合わせたものとして表現できる．

$$\frac{P \xrightarrow{\alpha} P' \; Q \xrightarrow{\overline{\alpha}} Q'}{\partial_{\{\alpha\}}(P \mid Q)[(\alpha \mid \overline{\alpha})/\tau] \xrightarrow{\tau} \partial_{\{\alpha\}}(P' \mid Q')[(\alpha \mid \overline{\alpha})/\tau]}$$

LTS の動作の例を以下に示す.

$$\begin{aligned} P &\stackrel{def}{=} a \cdot b \cdot P \\ Q &\stackrel{def}{=} \overline{b} \cdot c \cdot Q \\ Sys &\stackrel{def}{=} \partial_{\{b\}}(P \mid Q)[(b \mid \overline{b})/\tau] \end{aligned}$$

で定義される LTS において Sys からスタートする遷移は以下のようになる

$$\begin{aligned} & Sys \\ & \xrightarrow{a} \partial_{\{b\}}(b \cdot P \mid \overline{b} \cdot c \cdot Q)[(b \mid \overline{b})/\tau] \\ & \xrightarrow{\tau} \partial_{\{b\}}(P \mid c \cdot Q)[(b \mid \overline{b})/\tau] \\ & \xrightarrow{c} \partial_{\{b\}}(P \mid Q)[(b \mid \overline{b})/\tau] \\ & \text{または} \\ & \xrightarrow{a} \partial_{\{b\}}(b \cdot P \mid c \cdot Q)[(b \mid \overline{b})/\tau] \\ & \cdots \end{aligned}$$

3.2.2 LTS の並列結合演算

LTS 同士の並列結合演算 COMM を以下のように定義する

$LTS3$	$:= COM(LTS1, LTS2)$
$Proc_{LTS3}$	$:= Proc_{LTS1} \times Proc_{LTS2}$
Act_{LTS3}	$:= Act_{LTS1} \cup Act_{LTS2}$
S	$:= Act_{LTS1} \cap Act_{LTS2}$
$\xrightarrow{\alpha}$	$:= \partial_S(P \mid \overline{Q})[(\alpha \mid \overline{\alpha})/\alpha, \alpha \in S]$
	の遷移関係に従う

LTS3 の初期プロセスは，LTS1 の初期プロセス ×LTS2 の初期プロセスであるとする. また $P \in$ LTS1, $Q \in$ LTS2 であり，$\overline{Q}$ は LTS2 に含まれる $\beta \in Act_{LTS1} \cap Act_{LTS2}$ を co-name である $\overline{\beta}$ に置き換えたプロセスを指す物とする.

プロセス定義 $E = \{X \stackrel{def}{=} a{\cdot}b{\cdot}X, Y \stackrel{def}{=} c{\cdot}d{\cdot}Y, \}$ に対応する LTS を，LTS1=$\langle Proc1, Act1, \xrightarrow{\alpha} \rangle$ および LTS2=$\langle Proc2, Act2, \xrightarrow{\alpha} \rangle$ とするとき，

LTS1とLTS2の並列結合演算を行い，結合後のプロセスを`M0=(X|Y),M1=(b.X|Y),M2=(X|d.Y),M3=(b.X|c.Y)`と置くと，以下のLTS3を得る．

```
LTS3:
Proc={M0,M1,M2,M3},
Act={a,b,c,d}
-a->={(M0,M1),(M2,M3)},-c->={(M0,M2),(M1,M3)}
-b->={(M1,M0),(M3,M2)},-d->={(M2,M0),(M3,M1)}
```

eMSCにおけるコマンドは，定義式 $P \stackrel{def}{=} T \cdot P$(ここで$T$は$a \in Act, \cdot, +$により構成されるプロセス式) の形式で再帰的に定義されるプロセスPとして形式化でき，$P_i \stackrel{def}{=} T \cdot P_{i+1}$として$i$巡目の実行を区別することができる．

繰り返し実行のn巡目を区別する場合のLTSの並列結合演算は$\alpha \in Act_{LTS1}, \beta \in Act_{LTS2}$をそれぞれLTS1,LTS2の初期プロセスからの最初のアクションであるとすると，$ocr(\alpha), ocr(\beta)$を動作の巡目を表す関数として，$\langle R, ocr(\alpha) - ocr(\beta)\rangle$($R$はn巡目を区別しない並列結合$P \mid Q$) で表せるプロセスを持つLTSとして計算できる．

LTS1とLTS2の並列結合結果は，アクションa,cのk,j巡目を$ocr(a) = k, ocr(c) = j$と表して，以下の無限のLTSであるLTS4を得る．ここで初期プロセスは$ocr(a) = 0, ocr(c) = 0$より$\langle M0, 0\rangle$である．

```
LTS4:
Proc={<M0,i>,<M1,i>,
     <M2,i>,<M3,i>}
Act={a,b,c,d}
-a->={(<M0,i>,<M1,i+1>),(<M2,i>,<M3,i+1>)}
-b->={(<M1,i>,<M0,i>),(<M3,i>,<M2,i>)}
-c->={(<M0,i>,<M2,i-1>),(<M1,i>,<M3,i-1>)}
-d->={(<M2,i>,<M0,i>),(<M3,i>,<M1,i>)}
```

再帰方程式と並列演算の関係

コマンド図により互いに通信を行う最大２つの状態遷移機械が定義された．コマンド図で定義される個々の状態遷移機械は，初期状態に必ず戻るループ構造を１つだけ必ず持つ．状態遷移機械のｎ巡目の実行とは初期状態をｎ回通過した実行状態を指すものとする．

コマンドのプロセス代数による形式化においては，１つの状態遷移機械は，１つの再帰変数のみを含む１つの再帰定義式のみで構成される再帰方程式として形式化された．

並列に動作する複数の状態遷移機械を，１つのシステムとして組み合わせることは，プロセス代数では，複数の再帰定義変数を並列演算（$\|$) により結合することに対応する．

プロセス代数，特にACPにおいては，再帰定義変数の並列演算（$\|$) を，再帰定義式に従って展開し,，並列組合せを状態とする，システム全体としての，再帰方程式を計算する．再帰方程式は状態遷移系に読みかえることができ，実装や，検証が容易になる．

例えば，状態遷移機械における繰り返し実行のｎ巡目を区別しなければ，並列組合せの結果として，全体の状態遷移を以下のように計算できる．

２つの状態遷移機械 A, B が再帰変数 X, Y による再帰方程式 E で表現されるとする．

$$\begin{aligned} E &= \{ \\ &\quad X = a \cdot bX \\ &\quad Y = c \cdot dY \\ &\quad \} \end{aligned} \tag{3.1}$$

A と B を並列結合した状態遷移機械 C を再帰方程式の展開で計算できる．ここでは A, B が独立並列に動作するものとする．

再帰方程式 E は，ｎ巡目の動作状態を X_n, Y_n とすると，$X_n = a_n \cdot b_n X_{n+1}, Y_n = c_n \cdot d_n Y_{n+1}$ であることを示している．

X, Y を結合した全体のシステムの動作における組合せ状態に対応する変数を Z とすると，Z を再帰変数 X, Y の並列結合演算 $(Z = (X \parallel Y))$ で表すことができる．

A, B におけるｎ巡目の実行を区別しなければ，

$(X_0 \parallel Y_0) = (X_1 \parallel Y_1), ..., (X_n \parallel Y_n)$

となり，C を構成する，組み合わせ状態は，以下の４状態である．

$$\{(X_0 \parallel Y_0), (b_0X_0 \parallel Y_0), (X_0 \parallel d_0Y_0), (b_0X_0 \parallel d_0Y_0)\}$$

以下では，添字 (0) を省略して $(X \parallel Y)$ のように記述する．

組合せ再帰変数 $Z_0 = (X \parallel Y)$ に対して，新しい再帰変数と状態遷移の計算を行うため $Z_0 = (X \parallel Y)$ に対して E の再帰方程式に従いプロセス項の展開を行う．

再帰方程式 E から全体の動作を表す再帰変数 Z_0 は以下のように展開できる．

$$\begin{aligned} Z_0 &= X \parallel Y = X \lfloor\lfloor Y + Y \lfloor\lfloor X \\ &= (a \cdot bX) \parallel Y + (c \cdot dY) \lfloor\lfloor X \\ &= a \cdot (bX \parallel Y) + c \cdot (dY \parallel X) \end{aligned}$$

ここで $Z_1 = (bX \parallel Y), Z_2 = (dY \parallel X), Z_3 = (bX \parallel dY)$ としてそれぞれ展開を行うと以下の再帰変数 (Z_0, Z_1, Z_2, Z_3) による再帰方程式を得ることができる．

$$\begin{aligned} Z_0 &= aZ_1 + cZ_2 \\ Z_1 &= bZ_0 + cZ_3 \\ Z_2 &= dZ_0 + aZ_3 \\ Z_3 &= bZ_2 + dZ_1 \end{aligned} \tag{3.2}$$

一方，CCS' ではラベル付き遷移システム (LTS)$(Proc, Act, \{\xrightarrow{\alpha} | \alpha \in Act\})$ に当てはめると，再帰方程式 E は以下の LTS で表現できる．

```
Proc = {X,X1,Y,Y1}
Act = {a,b,c,d}
-a->={(X,X1)}
-b->={(X1,X)}
-c->={(Y,Y1)}
-d->={(Y1,Y)}
```

ここでプロセス X, Y の並列結合 (今度は $|$) を行った $X \mid Y$ は LTS の意味定義に従うと以下のように展開できる．

```
(X|Y)  -a-> (X1|Y)
(X|Y)  -c-> (X|Y1)
(X1|Y) -b-> (X|Y)
(X|Y) -c-> (X1|Y1)
(X|Y1) -a-> (X1|Y1)
(X|Y1) -d-> (X|Y)
(X1|Y1)-b-> (X|Y1)
(X1|Y1)-d-> (X1|Y)
```

ここで，Z0=(X|Y),Z1=(X1|Y),Z2=(X|Y1),Z3=(X1|Y1) と置くと X, Y の並列結合演算により得られた LTS は以下のようになる．

```
Proc={Z0,Z1,Z2,Z3},
Act={a,b,c,d}
-a-> = {(Z0,Z1),(Z2,Z3)}
-b-> = {(Z1,Z0),(Z3,Z2)}
-c-> = {(Z0,Z2),(Z1,Z3)}
-d-> = {(Z2,Z0),(Z3,Z1)}
```

このように ACP のようにプロセス項を展開しても，CCS のようにプロセスの並列結合を LTS のルールに従って，展開しても，同一の遷移系を得られることがわかる．

3.2.3 プロセス代数におけるプロパティと充足性

ここでは，プロセス代数において，プロセス M の動作にかかわる性質であるプロパティ ψ に対する充足性を定義する．

プロセス代数において，動作に関わる性質（プロパティ）を記述する代表的な手法に，1) プロセス代数自体で性質を記述する方法と，2) 様相論理を用いて性質を記述する方法の 2 種類ががある

1 ）プロセス記述で性質を記述する方法においては，動作を表すモデル M に対してプロパティ ψ の充足性（$M| = \psi$) を，ψ をプロセス記述に読みかえた P_ψ を M の関係により定義する．モデル M とプロパティ P_ψ の間の動作の等価性 (equivalence) を計算することに

より実現される．等価性の基準としては例えば模倣関係 (simulation relation) やトレース等価関係 (trace equivarence) を用いる (付録 A.2.3 参照)．

２）プロパティψを様相論理 (HML 等) を用いて定義する場合は，充足性判断はプロパティを充足する状態の集合を計算する問題に帰着される．モデル M を構成する全ての状態においてプロパティψが成立するならば，M に対してψが充足する（$M| = \psi$）という．

例えば，HML の場合（[2]）では，state formula F に対する充足集合 [[F]] を定義して，そこで，HML 式の意味定義 (参考文献 [2] のように denotational にあたえられるか，または帰納的にあたえられるか）を用いて，実行の有限トレース列に対する HML 式の充足性が定義される．

さらに，ループを含む状態遷移系における無限の実行トレース列に対して,「Fがいつかは成立する:$Pos(F)$」,「Fが常に成立する:$Inv(F)$」のような性質を定義する場合には，再帰的なプロパティ定義が行われこの再帰定義の最大あるいは最小解を求めることが充足集合を求めることすなわち充足性判断になる．最大解，最小解の計算は，最大不動点，最小不動点を求める問題に帰着される．不動点を求める手順には，有限回の繰り返し演算アルゴリズムを用いることができる (付録 A.2.4 参照)．

双模倣計算も様相論理の充足性計算も，2 人 Player の game として形式化することができることが Starling 等の研究成果として知られている [26]．また，Alfaro の “interface theory”[11] では，２つの状態遷移機械 (M_1, M_2) 間の通信が正しく（空振りしないという意味で）行われるための M_1, M_2 を取り巻く外部環境 E の制約 (A) を求めることに game を利用している，また Dimitra の，あるプロパティψを充足するような，モデル M に対する環境の制約 (A) を求める問題も，game による最大不動点計算に帰着できることがわかっている [14]．

本論文では，プロパティをプロセスで表現し，充足問題を game として形式化する (付録 A.1 参照)．

最初に，プロパティψをラベル付き遷移システム:LTS とプロセスにて記述する．

巡目を区別する並列結合について

ループする再帰変数 X で構成される再帰方程式 $X = a \cdot bX$ の実行において，ループの繰返し回数 i を区別する場合に，アクション a が i 回実行された状況を $ocr(a) = i$ と表す

と，再帰方程式に対応する，LTS の状態は，<P，ocr(a)=i>にて表すことができ，以下の遷移規則に従う．

$$\frac{}{\langle \alpha \cdot P, ocr(\alpha) = i \rangle \xrightarrow{\alpha} \langle P, ocr(\alpha) = i + 1 \rangle}$$

同様に，それぞれ独立にループする再帰変数 X, Y で構成される再帰方程式 $X = a{\cdot}bX, Y = c \cdot dY$ におけるプロセスの並列結合 $X \mid Y$ にいて，プロセスの i 番目の実行を区別する場合には

アクション a, c が何回実行されたかを $ocr(a), ocr(b)$ と表すと，状態の記述は

<P，ocr(a)=i，ocr(c)=j> として表せる．

$X \mid Y$ におけるプロセスの組み合わせは有限 $(X \mid Y, bX \mid Y, bX \mid dY, X \mid dY)$ であるが，$ocr(a) = i, ocr(b) = j$ の組み合わせは無限になるので，結果として並列システム $X \mid Y$ に対応する LTS は無限状態遷移系を構成することになる．

3.2.4 LTS 上の順序制約 ψ

LTS 上の順序制約 $\psi^n_{\alpha_i \to \beta_j}$ は，アクション $\alpha, \beta \in Act_{LTS}$ の間の実行順番に関して，“α の i 巡目は β の j 巡目より先行し，α の β に対する先行は n を越えない” という性質を表すものとする．

$P \in Proc_{LTS}$ を順序制約とは関係ない，元々のプロセス名であるとし，exocr(x) を x の先行度合いを表す整数であるとする．

$\psi^n_{a_i \to b_j}$ では exocr(a)=ocr(a)-(ocr(b)-(j-i)) になる．先行度合いを考慮したプロセスは<P,exocr(a)>と表すこととすると制約の意味を図 3.5 の遷移ルールで定義することができる．

図 3.5 は $\psi^n_{a_i \to b_j}$ が exocr(a) に制限，$0 \leq exocr(a) \leq n$ を与えていることを示している．

$0 \leq exocr(a)$ は $\psi^\infty_{a_i \to b_j}$ に対応する．一方 $exocr(a) \leq n$ は

exocr(a)=ocr(a)-(ocr(b)-(j-i)) $\leq n$ であるから

$0 \leq ocr(b) - (j - i) - ocr(a) + n$

即ち

$0 \leq ocr(b) - (ocr(a) - (i - (j - n))$

となりこれは制約 $\psi^\infty_{b_{j-n} \to a_i}$ に対応するので以下が成立する．

$$\frac{P \xrightarrow{\alpha} P'}{\langle P,k\rangle \xrightarrow{\alpha} \langle P',k+1\rangle}\ k<n \qquad \frac{P \xrightarrow{\beta} P'}{\langle P,k\rangle \xrightarrow{\beta} \langle P',k-1\rangle}\ k>0$$

$$\frac{P \xrightarrow{\alpha} P'}{\langle P,n\rangle \xrightarrow{\alpha} \delta} \qquad \frac{P \xrightarrow{\beta} P'}{\langle P,0\rangle \xrightarrow{\beta} \delta}$$

図 3.5: 順序制約 $\psi^n_{\alpha_i\to\beta_j}$ の展開ルール

$$\psi^n_{a_i\to b_j} = \psi^\infty_{a_i\to b_j} \wedge \psi^\infty_{b_{j-n}\to a_i}$$

$n=1, j=i$ であるとすると

$$\psi^1_{a_i\to b_i} = \psi^\infty_{a_i\to b_i} \wedge \psi^\infty_{b_{k-1}\to a_k} = \psi^\infty_{a_i\to b_i} \wedge \psi^\infty_{b_k\to a_{k+1}}$$

$n=1, j=i+1$ であるとすると

$$\psi^1_{a_i\to b_{i+1}} = \psi^\infty_{a_i\to b_{i+1}} \wedge \psi^\infty_{b_k\to a_k}$$

となる．

exocr(x) に着目して状態を抽象化するとルールは LTS として抽象化できる，

例えば $\psi^1_{a_i\to c_i}$ は，exocr(a)=ocr(a)-ocr(c) を状態変数とし，プロセスを”P 状態変数”と記述すると以下の初期プロセスを P0 とする LTS5 として表現できる．NG プロセスは遷移ルールの δ に対応し，順序制約が充足しない場合を表している．

```
LTS5:
 Proc = {P0,P1,NG}
 Act = {a,c}
 -a->{(P0,P1),(P1,NG)},-c->{(P1,P0),(P0,NG)}
```

同様に i 巡目の d の実行が i+1 巡目の a の実行に最大 1 巡先行する場合の制約 $\psi^1_{d_i\to a_{i+1}}$ は，

exocr(d)=ocr(d)-(ocr(a)-1) としてこれを状態変数とすると，以下の初期プロセスを P1 とする LTS6 として表現できる．

```
LTS6:
Proc={P0,P1,NG}
Act ={a,d}
-a->{(P1,P0),(P0,NG)},-d->{(P1,NG),(P0,P1)}
```

同様に i 巡目の d の実行が i+1 巡目の a の実行に最大 2 巡先行する場合の制約 $\psi^2_{d_i \to a_{i+1}}$ は，以下の LTS7 になる

```
LTS7:
Proc={P0,P1,P2,NG}
Act ={a,d}
-a->{(P1,P0),(P2,P1),(P0,NG)},
-d->{(P1,P2),(P0,P1),(P2,NG)}
```

順序制約に対応する LTS は，NG 状態を除く全ての状態において全ての入力を受理できる性質をもっている (input enabled と呼ぶ).

先行を無視する順序制約:$\psi_{\mathbf{OW}}$

LTS 上の順序制約 $\psi^n_{\mathrm{OW}:\alpha_i \to \beta_j}$ はアクション $\alpha, \beta \in Act_{LTS}$ の間の実行順番に関して，“α の i 巡目は β の j 巡目より先行し，α の β に対する先行度合いは n を越えると失われる” という性質を表すものとする.

順序制約 $\psi^n_{\alpha_i \to \beta_j}$ では，先行度合いは n を越えないが，$\psi^n_{\mathrm{OW}:\alpha_i \to \beta_j}$ では先行度合いは n を越えることができるが，n 以上の超過は記録されない.

$\psi^n_{\alpha_i \to \beta_j}$ の展開ルールに以下を加えたルールをもつ，即ち n を越えると，いくら越えたかの情報は失われてしまう.

$$\frac{P \xrightarrow{\alpha} P'}{\langle P, k\rangle \xrightarrow{\alpha} \langle P', k\rangle} \quad k = n$$

3.2.5　シナリオの順序制約

シナリオは，並列動作するコマンド起因の状態遷移機械を連係させて，データをシナリオの最初（起点）から最後（終点）に伝えることを強要する上位階層の仕様である．

個々のコマンド起因の状態遷移機械は，外部との通信によりデータを受理し，処理を行い，外部との通信により結果を出力するものとし，シナリオは状態遷移機械間でデータの受け渡しが正しく行われることを強要する．ここでは，正しく受け渡されるとは，n 巡目の実行のデータが $n+1$ 巡目の実行のデータにより上書きされないことであるとする．そこで状態遷移機械同士の連携動作に制約を設けることによりデータが正しく受け渡される範囲で，正しく n 巡目の動作が実行されることがシナリオの制約である．

eMSC においてはシナリオにおける，状態遷移機械同士の連係の種類には横チェーンと縦チェーンがあった．

横チェーンは，２つの状態遷移機械の間で n 巡目の一連の通信手順（プロトコル）を連動させることにより，データを伝えることを保証する．縦チェーンは，２つの状態遷移機械の片方（A とする）の n 番目の実行が終了するときにもう片方（B とする）の n 番目の実行を開始し，A の $n+1$ 番目の実行が B の n 巡目の実行を越えないことによりデータをリレー式に伝えることを保証する．

並列動作する状態遷移機械間の動作の性質とその実現に関して，プロパティと合成操作とを導入する．

プロパティ（動作の性質）は，２つの並列動作する状態遷移機械の連係動作を宣言的に定義するものであり，一方合成操作は，並列動作する状態遷移機械を１つの状態遷移機械に合成する操作である．

合成操作の結果，プロパティが充足するかどうかを判定したり，逆にプロパティを充足するような合成操作を見つける．以下では，アクション間の実行順序制約（→ : 先行する）をプロパティ定義に用いる．

横チェーン (ψ_h) は，並列動作する状態遷移機械同士は n 巡目の動作を連動させながら，互いにデータをやりとりする．片方 (A とする) が n 巡目のデータのやりとり中にもう片方 (B とする) が $m = n+1$ 回目のデータのやりとりが開始されないことがプロパティ（動作の性質）になる．$st_n(X)$ を X の n 巡目の開始，$en_n(X)$ を n 巡目の終了アクションであるとすると,「先行する」関係により

$$A\psi_h B = en_n(A) \text{ は } st_{n+1}(B) \text{ に先行} \wedge$$
$$en_n(B) \text{ は } st_{n+1}(A) \text{ に先行する}$$

,

という制約であると定義できる.

一方，縦チェーン (ψ_v) は，並列動作する状態遷移機械同士は，前側　(A とする) の n 回目の終了時に，n 巡目のデータの後側 (B とする) とのやりとりが行われるから，このとき B が $n-1$ 巡目の動作を終了していることがプロパティ（動作の性質）になり,「先行する」関係によりプロパティを記述すると，

$$A\psi_v B = en_n(A) \text{ は } st_n(B) \text{ に先行し} \wedge$$
$$en_n(B) \text{ は } en_{n+1}(A) \text{ に先行する}$$

3.2.6　シナリオの形式化

シナリオ図は，横チェーンと縦チェーンの連鎖をあらわしている

定義 3.2.1 (シナリオ SC) シナリオ SC は $\langle PS, RH, RV\rangle$ で定義される

1. PS はプロセスの集合
2. RH は横チェーン関係 $(X, Y), X, Y \in PS$ の集合
3. RV は縦チェーン関係 $< X, Y >, X, Y \in PS$ の集合
4. ここで $\langle\rangle$ は順序性がある $\langle X, Y\rangle \neq \langle Y, X\rangle$

プロセス代数では，プロセスは再帰変数で形式化され，縦チェーン・横チェーンは 3.4 節で導入される ∇ 演算として形式化される．縦チェーンおよび横チェーンは二項関係であるので，プロセスをリーフノード，関係をリーフノード間を結ぶノードとして関係ノードとリーフノードを線で結ぶと，グラフ構造を得ることができる．しかしこのグラフは，木構造ではない.

例えば，図 3.6 の例では,

$SC_{abcd} = \langle PS, RH, RV\rangle$ と表される.

ここで $PS = \{A, B, C, D\}, RH = \{(A, B), (C, D)\}, RV = \{< B, C >\}$ となる.

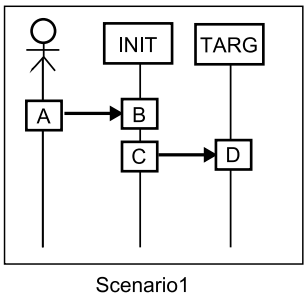


図 3.6: シナリオモデル

3.3 順序制約の充足性

プロセス M と順序制約 ψ の間の充足性を定義する．M に対応する LTS を LTS_M，ψ に対応する LTS を LTS_ψ とし，$Act_{LTS_\psi} \subseteq Act_{LTS_M}$ とする．

3.3.1 充足性の定義と充足ゲーム

プロセス M と P_ψ の間の充足性をここでは Game 理論を用いて定義する．

定義 3.3.1 (充足性) プロセス M が ψ を充足するとは，$\mathrm{COM}(LTS_M, LTS_\psi)$ の動作が初期プロセスから NG プロセス (遷移先が δ である場合を NG プロセスとする) に到達しないことである

Player1 を，アクション $Act_{LTS_M} - Act_{LTS_\psi}$，Player2 を Act_{LTS_ψ} を実行できる player であるとすると，以下のような充足 Game として，充足性の判定計算を行うことができる [11].

定義 3.3.2 (充足 Game) LTS で記述されるモデル M(Player1 とする) と，プロパティ ψ に対応する LTS で記述されるプロセス P_ψ(Player2 とする) が与えられたとき，P_ψ と M_{mon} とを並列結合 (|) した M' に対して，Player2 が NG 状態に遷移させようと，Player1 がそれを阻止しようとして互いに自分のアクションを実行する Game 問題においては，Player1 が Player2 に対して，常勝する場合に限り，M は ψ を充足する

最初に，Game 構造 (Game Structure) を導入する．Game 構造 (Game Structure) は以下の 4 要素構造で構成される，

定義 3.3.3 (Game 構造 (Game Structure))

$$G = \langle S, M, \Gamma_1, \Gamma_2, \delta \rangle$$

S は状態の集合，M は Move の集合，Move 割り当て $(\Gamma_i, i = 1, 2)$ は状態 $s \in S$ に対して，Player i の可能な Move の集合 $\Gamma_i(s) \subseteq M$ を割り当 てる関数，遷移 $\delta : S \times M \times M \mapsto S$ は，状態 s に対して Player1,Player2 の Move の後，次状態に達する関数により定義される．

また G は Turn-based であるとする．すなわち互いに素な Player1 のみの手番の状態の部分集 合 S_1 と Player2 の手番の部分集合 S_2 で S が構成される $(S = S_1 \cup S_2)$ ものとする．

3.3.2 勝利集合と充足性計算の手順

Player1 が Player2 に対して常勝であることを示すためには，まず Player1 が Player2 にたいして universal な勝利戦略を持つことを示し（空集合でない勝利集合が求まるということ），さらに Player1 の手番の状態からの Player2 の手番の状態への遷移において遷移先の Player2 の手番の状態が全て勝利集合に含まれることを示す．

ここで universal な勝利戦略を持つとは，Player2 のいかなる手に対しても Player1 が goal の範囲に遷移先が含まれるような，選択手が常に存在することを指す．

universal な勝利戦略を求める問題は safety-game とよばれ，Player1 の勝利集合 Win_1 を求めることに帰着される．ここで勝利集合 Win_1 とは Player1 の手番において次の遷移先を選択する場合，勝利集合 Win_1 に含まれる状態への遷移を選択すれば，常に Player1 が勝てる必勝戦略 π_1 が存在するような状態の集合である．さらに Player1 の手番において選択できる次遷移先が全て Win_1 に含まれる場合，Player1 は常勝であるという．

X を状態の集合であるとして，Player1 に対する $CPre_1(X)$:conditional predecessor を以下のように定義する．

定義 3.3.4 ($CPre_1(X)$:Conditional Predecessor(Controllable Predecessor))

$CPre_1(X)$ は conditional predecessor とよばれる集合で，Player1 が 1 回の turn で game の状態が X に含まれる様に，制御できるような前状態 s の集合を表す．

$$CPre_1(X) = \{s \in S | \exists a \in \Gamma_1(s). \forall b \in \Gamma_2(s). \delta(s, a, b) \subseteq X\}$$

Turn-based な Game の場合には，$E(s)$ を状態 s から可能な遷移先の状態の集合とすると，以下と同等である．

$$CPre_1(X) = \{s \in S_1 | \exists t \in E(s), t \in X\} \cup \{s \in S_2 | \forall t \in E(s), t \in X\}$$

勝利集合 Win_1 とは，$CPre_1(X)$ を施した結果の集合も，もとの集合 X の内に留まる状態の集合，すなわち Win_1 は $X = CPre_1(X)$ となる方程式の解になる．

すなわち，勝利集合とは，S の部分モデル S' で，S' の conditional predecessor が必ず S' に含まれるような，部分モデル S' である．,

$CPre_1(X)$ が単調な関数であるので，Tarski の不動点定理 [28] より，最大不動点，最小不動点が存在し，適切な初期集合 X_0 を選択して，$CPre_1(X)$ を繰り返し適用して，収束先を求めることにより，不動点を手続き的に計算できることが，不動点理論より知られている．ここで R とは NG でない状態の集合 $S - \{NG\}$ である．

$$Win_1 = \nu X.(R \cap CPre_1(X))$$

最大不動点の場合は，$X_0 = S$ として $CPre_1(X)$ を繰り返し適用し，収束する状態の集合が最大不動点である．

$$\begin{aligned}
X_0 &= S \\
X_1 &= R \cap CPre_1(X_0) \\
\cdots &= \cdots \\
X_n &= R \cap CPre_1(X_{n-1}) \\
\cdots &= \cdots \\
Win_1 &= lim_N(X_n), X_n = X_{n-1}
\end{aligned}$$

最大不動点が，Player1 の勝利集合 $X = CPre_1(X)$ のうち最大の解になる．

Player1 の勝利集合 Win_1 が空集合である場合は必ず負ける．空集合でない場合は Player1 の手番で Win_1 に含まれる次遷移を選択することにより，自手番で常に勝利戦略をたてることができる．

さらに Win_1 が Player1 の手番における次状態を全て含む場合には，Player1 が自手番にてどの次遷移を選択しても勝利するので，常勝になる．

Player1 が常勝であるときに，モデル M はプロパティ ψ を充足するという．

次にプロセス M と P_ψ の間の充足性を充足 Game を利用して計算する．ここで M に関するアクションを Act_M，P_ψ に関するアクションを Act_ψ とし，$Act_\psi \subset Act_M$ すなわち，P_ψ は M の動作のモニターとして動作するものとする．

$\alpha \in (Act_\psi \cap Act_M)$ であるとすると

M の遷移 $(P1, P2) \in \xrightarrow{\alpha}$ に対して，$(P1, P2)$ を $(P1, P2')$ に書換え，かつ $(P2', P2)$ を $\xrightarrow{\alpha_1}$ に追加して M_{mon} を得る．

一方 P_ψ の遷移集合 $\xrightarrow{\alpha}$ に対して，アクションを $\overline{\alpha_1}$ に書き換え，P'_ψ を得る

ここで，M_{mon} と P'_ψ の間の並列結合演算 $(|)$ を行い M' を得る．

$\{a1, b1, c1, ...\} = (Act'_\psi \cap Act_{M_{mon}})$ であるとすると，

$$M' = \partial_{a1,b1,..}(M_{mon} \mid P'_\psi)[a1 \mid \overline{a1}/a2, b1 \mid \overline{b1}/b2, ..]$$

定義 3.3.5 充足性 M が ψ を充足するとは，M' の動作が状態 $\langle X|NG\rangle (X$ は M から遷移可能な状態) に到達しないことである. これは即ち M のアクションによる遷移先が全て Win_1 に含まれることである．

M のアクションを全て Player1 のアクションにするために M を M_{mon} に拡張している．

充足 Game による充足性の定式化は，M の ψ に対する充足性の計算に加えて，充足しない場合には ψ を充足する最大の M の部分モデル M' を計算できる (勝利集合 Win_1 にて M の動作を ψ を充足するように制限するということ) という特徴を持っている.

例えばモデル P を再帰変数 X，プロパティ ψ を b の後に c が実行されるという順序制約 $\psi_{b\to c}$ を再帰変数 Y で表すと，X, Y は例えば以下のように再帰方程式で記述できる

$$\begin{aligned} E1 \quad &= \{ \\ &\quad X = (a \cdot b \cdot c + e \cdot b \cdot c) \cdot X \\ &\quad Y = b \cdot c \cdot Y \\ &\quad \} \end{aligned}$$

これを，M_{mon} および P_ψ に変換する

$$
\begin{aligned}
E1 \ &= \{ \\
&X = (a \cdot b \cdot b1 \cdot c \cdot c1 + e \cdot b \cdot b1 \cdot c \cdot c1) \cdot X \\
&Y = \overline{b1} \cdot Y1 + \overline{c1} \cdot NG \\
&Y1 = \overline{c1} \cdot Y1 + \overline{b1} \cdot NG \\
&\}
\end{aligned}
$$

X と Y の並列結合

$M' = \partial_{b1,c1}(X \mid Y)[b1 \mid \overline{b1}/b2, c1 \mid \overline{c1}/c2]$ 対する LTS $(Proc, Act, \{\xrightarrow{\alpha} \mid \alpha \in Act\})$ は以下のようになる (遷移関係は δ :`Proc X Act` $\rightarrow$ `Proc` にて表している)

$$
\begin{aligned}
s0 &= \partial_{b1,c1}(X \mid Y)[b1 \mid \overline{b1}/b2, c1 \mid \overline{c1}/c2], \\
s1 &= \partial_{b1,c1}(b \cdot b1 \cdot c \cdot c1 \cdot X \mid Y)[b1 \mid \overline{b1}/b2, c1 \mid \overline{c1}/c2], \\
s2 &= \partial_{b1,c1}(b1 \cdot c \cdot c1 \cdot X \mid Y1)[b1 \mid \overline{b1}/b2, c1 \mid \overline{c1}/c2], \\
s3 &= \partial_{b1,c1}(c \cdot c1 \cdot X \mid Y1)[b1 \mid \overline{b1}/b2, c1 \mid \overline{c1}/c2], \\
s4 &= \partial_{b1,c1}(c1 \cdot X \mid Y1)[b1 \mid \overline{b1}/b2, c1 \mid \overline{c1}/c2],
\end{aligned}
$$

```
Proc={s0,s1,s2,s3,s4}
Act={a,b,c,d,e}
 δ (s0,a) = s1, δ (s0,e) = s1,
 δ (s1,b) = s2,
 δ (s2,b2)= s3,
 δ (s3,c) = s4,
 δ (s4,c2)= s0,
Proc_1 = {s0,s1,s3}
Proc_2 = {s2,s4}
```

a,b,c,d,e は環境側:player1 のアクション，b2,c2 が Player2 のアクションである．また s0,s1,s3 が Player1 の手番,s2,s4 が Player2 の手番である．

直感的には，Player2 の手筋 $b3, c3$ から NG に遷移しないので，Player1 は手筋 a, b, c, d, e のどれを選択しても，勝つので常勝になる．

不動点の計算で説明する．

R は NG 以外の状態すなわち Proc そのものである．

最初に $X_0 = Proc = s_0, s_1, s_2, s_3, s_4$ として $CPre_1(X_0)$ を計算する．

$E(s0) = \{s1\}, E(s1) = \{s2\}, E(s2) = \{s3\}, E(s3) = \{s4\}, E(s4) = \{s0\}$ であるから

Player1 の手番 s0,s1,s3 は遷移先（の少なくとも 1 つ）が X_0 に含まれるので $CPre_1(X_0)$ に含まれる．一方 Player2 の手番 s2,s4 は，遷移先（の全てが）が X_0 に含まれるので $CPre_1(X_0)$ に含まれる．

$CPre_1(X_0) = \{s_0, s_1, s_3\} \cup \{s_2, s_4\} = X_0 = Proc$

となり収束し，勝利集合 $Win_1 = \{s_0, s_1, s_2, s_3, s_4\} = Proc$ を得る．

全ての Player1 の手番 s0,s1,s3 において，次の遷移先の状態が全て勝利集合 Win_1 に含まれるので Player1 は常勝となり，M は ψ を充足する ($M \models \psi$)．

一方 E1 のプロセス X に新たな動作を加えた場合を E2 として E1 の場合と同様に，充足性を計算する．

$$
\begin{aligned}
E2 \ &= \{ \\
&\quad X = (a \cdot b \cdot c + a \cdot c \cdot d) \cdot X \\
&\quad Y = b \cdot c \cdot Y \\
&\quad \}
\end{aligned}
$$

これを，M_{mon} および P_ψ に変換する

$$
\begin{aligned}
E2 \ &= \{ \\
&\quad X = (a \cdot b \cdot b1 \cdot c \cdot c1 + a \cdot c \cdot c1 \cdot d) \cdot X \\
&\quad Y = \overline{b1} \cdot Y1 + \overline{c1} \cdot NG \\
&\quad Y1 = \overline{c1} \cdot Y + \overline{b1} \cdot NG \\
&\quad \}
\end{aligned}
$$

再帰方程式 E2 における，$M' = \partial_{b1,c1}(X|Y)[b1|\overline{b1}/b2, c1|\overline{c1}/c2]$ 対する LTS $(Proc, Act, \{\xrightarrow{\alpha} | \alpha \in Act\})$ は以下のようになる (遷移関係は δ :`Proc X Act` $\rightarrow$ `Proc` にて表している)

```
Proc={s0,s1,s2,s3,s4,s5,NG}
Act={a,b,c,d,e}
 δ (s0,a) = s1, δ (s1,c) = s5,
 δ (s1,b) = s2, δ (s5,c2)= NG,
 δ (s2,b2)= s3,
 δ (s3,c) = s4,
 δ (s4,c2)= s0,
Proc_1 = {s0,s1,s3}
Proc_2 = {s2,s4,s5}
```

a,b,c,d,e は環境側:player1 のアクション，b2,c2 が Player2 のアクションである．

直感的には，Player1 が勝つためには，Player2 の手筋 $c3$ で NG に遷移する手番 s9 に遷移させないように，Player1 は手筋 a, b, c, d, e を選択しなくてはいけなくなるので常勝ではない．

不動点の計算で説明する．

R は NG 以外の状態，Proc-NG である．

最初に $X_0 = Proc = s_0, s_1, s_2, s_3, s_4, s_5$ として $CPre_1(X_0)$ を計算する．

$E(s0) = \{s1\}, E(s1) = \{s2, s5\}, E(s2) = \{s3\}, E(s3) = \{s4\}, E(s4) = \{s0\}, E(s5) = \{NG\}$ であるから

Player1 の手番 s0,s1,s3 は遷移先（の少なくとも 1 つ）が X_0 に含まれるので $CPre_1(X_0)$ に含まれる．一方 Player2 の手番 s2,s4 は，遷移先（の全てが）が X_0 に含まれるので $CPre_1(X_0)$ に含まれるが，s5 の遷移先 (NG) は含まれないので $CPre_1(X_0)$ に含まれない．

従って，

$X_1 = CPre_1(X_0) = \{s_0, s_1, s_2, s_3, s_4\}$

を得る．繰り返し演算を行うと $X_2 = CPre_1(X_1) = X_1$ となり収束して最大不動点を得ることができる．

勝利集合は $Win_1 = \{s_0, s_1, s_2, s_3, s_4\}$ となる．

Player1 の手番 s1 において，次の遷移先の状態 s5 が勝利集合 Win_1 に含まないの Player1 は常勝ではなくなり，M は ψ を充足しない．

逆に Player1 の手を Win_1 に含まれるように絞り込めば，ψ を充足することになる．こ

のように勝利集合を求める演算は，最大不動点計算 (ν) に帰着され，最大不動点は ψ を充足するモデル M の最大の部分モデル M'' を求めていることに相当する．

3.3.3 他の充足性判定例 (CWB-NC の利用)

順序制約 ψ_p を naive に順序強要演算である ∇_p 演算を適用したモデルが ψ_p を充足する最大のモデルであることを示すために，ここまでは充足 Game に従って充足性を説明してきたが，他の手段を用いても充足性は示すことができる．

ここでは，プロセス代数CCSの処理系である”Concurrency Work Bench New Century”[1](以下 CWB-NC と略す) を用いて，充足性を判定する例を示す．

```
*プロセスを定義します,
proc X_ok = a.b.c.X_ok + e.b.c.X_ok
proc X_ng = a.b.c.X_ng + a.c.d.X_ng + e.b.c.X_ng

* こちらが充足すべき Spec です
* (b.c)*以外 n 動作を行うものは終端 (nil) に遷移します
proc Spec0 = 'b.Spec1 + 'c.nil
proc Spec1 = 'c.Spec0 + 'b.nil

*b,c を介して同期通信を強要させます
set Internals2 = {b, c}

*並列結合を行い同期通信を強要させます
proc Prod_ng = (X_ng | Spec0) \ Internals2
proc Prod_ok = (X_ok | Spec0) \ Internals2
```

ここで，nil に遷移して（遷移先が無いから）deadlock になるような性質を様相論理を用いて以下のように定義する．

```
prop can_deadlock =
```

```
min X = [-]ff \/ <->X
```

ここで [-]ff は全てのアクションがまったく実行できない状況を表し，この状態に到達するか可能性を再帰的に X は定義している．min は最小不動点演算に関係している [2](詳しくは付録 A.2.4 参照)．

deadlock になるかどうかを判定することにより，nil に遷移するかどうか，すなわち性質 Spec0 を充足するかどうかを判定する．以下は can_deadlock を”dead.mu” というファイルから読み，プロセス (Sys_ok,Sys_ng) が仕様 (Spec0) を充足するかどうかを，判定したログである．CWB-NC を用いた充足性判定の例は，付録 A.3 章を参照のこと．

```
*プロパティ (nil 遷移付き) と掛け合わせて deadlock しないか (nil 遷移がないか)
*    を見る方法
*以下のように nil に遷移するのをみつけます
load dead.mu
cwb-nc> search Prod_ok can_deadlock
States explored: 4
No state found satisfying can_deadlock.

cwb-nc> search Prod_ng can_deadlock
State found satisfying can_deadlock.
Path to state contains 1 states, invoking simulator.
1: Prod_ng
*cwb-nc-sim>
```

Spec0 を Sys_ok は充足し，Sys_ng は充足しないことがわかる．

3.4 ∇ 演算の導入

LTS に対して $\psi^n_{\alpha_i \to \beta_j}$ を充足するような動作を強要する演算 $\nabla_{\psi^n_{\alpha_i \to \beta_j}}$ を導入する．

3.4.1 $\nabla_{\psi^n_{\alpha_i \to \beta_j}}$

$\psi^n_{\alpha_i \to \beta_j}$ に対応する LTS(LTS$_{\psi^n_{\alpha_i \to \beta_j}}$ とする) において，
以下順序制約 $\psi^n_{\alpha_i \to \beta_j}$ に対応する ∇ 演算 $\nabla_{\psi^n_{\alpha_i \to \beta_j}}$ を以下のように定義する．

$$\nabla_{\psi^n_{\alpha_i \to \beta_j}} : Proc \times int \times Act \times int \times Act \times int \mapsto Proc$$

- 任意の LTS に対して,COM(LTS, LTS1'$_{\psi^n_{\alpha_i \to \beta_j}}$) によって定義される LTS' を得る
- LTS' において初期プロセス s_0 から到達可能な LTS の部分モデル LTS" を得る

到達可能な部分 LTS を得るための演算は，以下に示す $CPre$ 演算を用いる．

$$CPre_1(X) = \{s \in S_1 | \exists t \in E(s), t \in X\}$$

$CPre_1(X)$ が単調な関数であるので，Tarski の不動点定理 [28] より，最大不動点，最小不動点が存在し，適切な初期集合 X_0 を選択して，$CPre_1(X)$ を繰り返し適用して，収束先を求めることにより，不動点を手続き的に計算できることが，不動点理論より知られている．

$$Win_1 = \nu X.(R \cap CPre_1(X))$$

ここで R とは NG でない状態の集合 $S - \{NG\}$ である．最大不動点の場合は，$X_0 = S$ として $CPre_1(X)$ を繰り返し適用し，収束する状態の集合が最大不動点 FIX である．

$$\begin{aligned}
X_0 &= S \\
X_1 &= R \cap CPre_1(X_0) \\
\cdots &= \cdots \\
X_n &= R \cap CPre_1(X_{n-1}) \\
\cdots &= \cdots \\
FIX &= lim_N(X_n), X_n = X_{n-1}
\end{aligned}$$

最大不動点が，到達可能な部分 LTS のプロセスの集合になる.

$\nabla_{\psi^n_{\alpha_i \to \beta_j}}(P)$ は順序制約 $\psi^n_{\alpha_i \to \beta_j}$ に従い，P を起点とする LTS の中で $\psi^n_{\alpha_i \to \beta_j}$ に矛盾する遷移を抑制する意味を持つ．

例えば，プロセス $PT_1 \stackrel{def}{=} a \cdot b \cdot 0, PT_2 \stackrel{def}{=} c \cdot d \cdot 0$ の並列結合に対して制約 $\nabla_{\psi^1_{b_k \to d_k}}$ を挿入するとする．

最初に，$PT_1 \mid PT_2$ は以下の LTS を構成している，

```
LTS7:
 Proc={s0=a.b.0|c.d.0, s1=b.0|c.d.0,  s3=0|c.d.0, s6=0|d.0, s8=0|0,
       s2=a.b.0|d.0, s5=a.b.0|0, s4=b.0|d.0, s7=b.0|0}
 Act={a,b,c,d}
 -a-> = {(s0,s1),(s2,s4),(s5,s7)}
 -b-> = {(s1,s3),(s4,s6),(s7,s8)}
 -c-> = {(s0,s2),(s1,s4),(s3,s6)}
 -d-> = {(s2,s5),(s4,s7),(s6,s8)}
```

$\nabla_{\psi^1_{b_k \to d_k}}$ の LTS から NG 状態及び NG への遷移を削除すると以下の LTS8 を得る

```
LTS8:
 Proc = {P0,P1}
 Act = {b,d}
 -b->{(P0,P1)},-d->{(P1,P0)}
```

ここで LTS8 の b,d を $\overline{b}, \overline{d}$ に置き換え

$M' = \partial_{b,d}(s0 \mid P0)[b \mid \overline{b}/b, d \mid \overline{d}/d]$

を計算し，s0’=s0|P0，s1’=s1|P0,s2’=s2|0,s3’=s3|P1,s4’=s4|P0，s6’=s6|P1,s8’=s8|P0，とすると結果は，以下のようになる

```
 Proc={s0',s1',s2',s3',s4',s6',s8'}
 Act={a,b,c,d}
 -a-> = {(s0',s1'),(s2',s4')}
 -b-> = {(s1',s3'),(s4',s6')}
 -c-> = {(s0',s2'),(s1',s4'),(s3',s6')}
 -d-> = {(s6',s8')}
```

即ちもとの LTS に比べて遷移が制限された LTS が得られた．

3.4.2 $\nabla_{\psi^n_{\mathbf{OW}:\alpha_i\to\beta_j}}$

$\nabla_{\psi^n_{\mathrm{OW}:\alpha_i\to\beta_j}}$ 演算は同様に $\psi^n_{\mathrm{OW}:\alpha_i\to\beta_j}$ に対応する順序制約挿入演算である．

3.4.3 巡目を考慮した並列結合における ∇ 演算

$X = a \cdot bX, Y = c \cdot dY$ として

$X \mid Y$ における巡目（ループの実行回数）を無視した，正味の状態は $X \mid Y, bX \mid Y, X \mid dY, bX \mid dY$ の 4 状態であり，

`X|Y:M0`，`bX|Y:M1`，`X|dY:M2`，`bX|dY:M3` として，`exocr(a)=ocr(a)-ocr(c)` の値との組合せで状態（正味の状態，`exocr(a)` の値）を表現する事にする．

何も制約がなければ，$exocr(a)$ は無限に増えるので，下図 3.7 の (M0,0) を初期プロセスとする無限 LTS を得る．

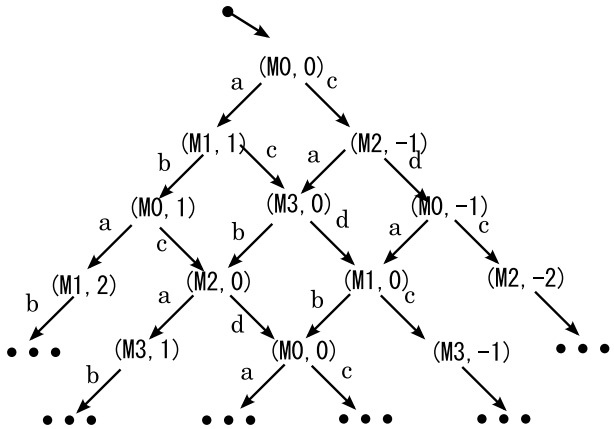


図 3.7: 無限状態 LTS

図 3.7 の LTS に $\nabla_{\psi^1_{a_i\to d_i}}$ を適用する．

$\nabla_{\psi^1_{a_i\to d_i}}$ に対応する LTS は，以下の P0 を初期プロセスとする LTS になるから

```
Proc={P0,P1,P2,NG}
Act ={a,d},
-a->{(P0,P1),(P1,NG))},
-d->{(P1,P0),(P0,NG)},
```

これらの並列同期結合演算を計算すると，状態を”(正味の状態,exocr(a),$\nabla_{\psi^1_{a_i \to d_i}}$ の状態)“で表すと以下の図 3.8 に表される LTS をえる．

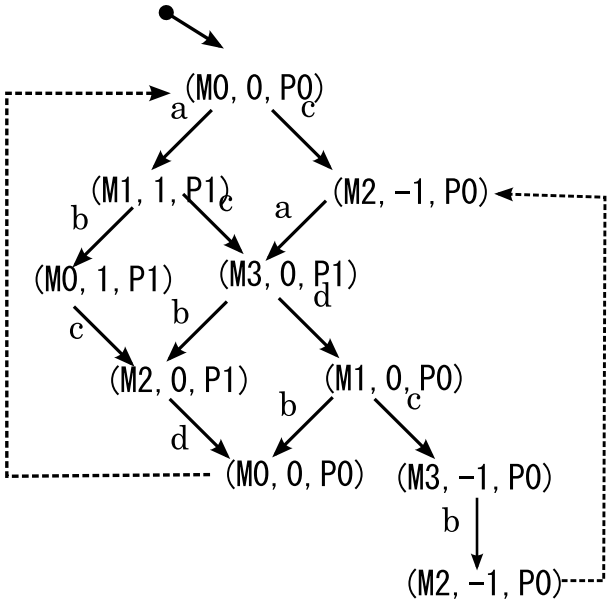


図 3.8: $\nabla_{\psi^1_{a_i \to d_i}}$ 適用後の LTS

この例では，∇ 演算の結果，有限状態の LTS(図 3.8) が得られたことになる．このように巡目を考慮した無限状態の LTS に対しても ∇ 演算を行うことができる．

3.5 ∇ 演算と充足性

順序制約 $\psi^n_{a \to b}$ と $\nabla_{\psi^n_{a \to b}}$ 演算の関係についてここでは述べる．

定理 3.5.1 (∇_ψ 演算と順序制約 ψ の充足性) $M = \nabla_{\psi^n_{x \to y}}(P)$ は制約 $\psi^n_{x \to y}$ を充たす P の最大の部分モデル $M \subset P$ である．

$\psi^1_{a \to d}$ の充足性を判定する
まず M を拡張して M_{mon} を得る．

```
Proc={s0,s1,s2,s3,s4,s5,s6,s7,s8}
Act={a,a1,b,c,d,d1}
-a-> = {(s0,s01),(s2,s24),(s5,s57)}
-a1-> = {(s01,s1),(s24,s4),(s57,s7)}
-b-> = {(s1,s3),(s4,s6),(s7,s8)}
-c-> = {(s0,s2),(s1,s4),(s3,s6)}
-d-> = {(s2,s25),(s4,s47),(s6,s68)}
-d1-> = {(s25,s5),(s47,s7),(s68,s8)}
```

一方 P_ψ は以下のようにあらわせ

```
Proc={ps0,ps1}
Act={a,d}
-a-> = {(ps0,ps1)}
-d-> = {(ps2,ps0)}
```

P_ψ における a を $\overline{a1}$,d を $\overline{d1}$ と置き換え P'_ψ とする.

$$M' = \partial_{a1,d1}(M_{mon} \mid P'_\psi)[a1 \mid \overline{a1}/a2, d1 \mid \overline{d1}/d2]$$

を求めると M' の LTS は

```
Proc={s0,s1,s2,s3,s4,s5,s6,s7,s8}
Act={a,a2,b,c,d,d2}
-a-> = {(s0,s01),(s2,s24)}
-a2-> = {(s01,s1),(s24,s4)}
-b-> = {(s1,s3),(s4,s6),(s7,s8)}
-c-> = {(s0,s2),(s1,s4),(s3,s6)}
-d-> = {(s4,s47),(s6,s68)}
-d2-> = {((s47,s7),(s68,s8)}
```

$a2, d2$ に関する a, d 遷移の拡張を削除すると，M に対して $\nabla_{\psi^1_{a\to d}}$ を行ったものと同一の LTS である.

充足性の計算は性質 ψ を充足する M の最大の部分モデルをもとめているから ∇^1_ψ 演算を施して得られた結果は，ψ を充足する M の最大の部分モデルになっていることになる．

最大であるということは ψ を充足するモデルで，∇^1_ψ 演算を施して得られた結果の部分モデルであるものが存在する場合があることを示唆している．

$\nabla_{\psi^n_{x\to y}}(P)$ ならば，展開ルールに従って展開すると順序制約 $x \to y$ と矛盾する項は δ に書き換えられて消されるので，結果が制約 $\psi^n_{x\to y}$ を充足するのは自明である．

逆は成立しない，例えば $PT_1 = a \cdot b, PT_2 = c \cdot d$ とし，$\psi_{a\to d}$ であるとする．

$\nabla_{\psi^1 a\to d}(PT_1|PT_2)$ の結果は $\psi^1_{a\to b}$ を充足するが，$\nabla_{\psi^1_{b\to d}}(PT_1|PT_2)$ の結果も $(a \to (b) \to d$ という意味で$)\psi^1_{a\to b}$ を充足する，また $\nabla_{\psi^1_{a\to c}}(PT_1 \mid PT_2)$ も同様に $(a \to (c) \to d$ という意味で$)\psi^1_{a\to b}$ を充足する．

$\psi^1_{a\to b}$ を充足する ∇ 演算の集合の中で，順序制約 $\psi^1_{a\to b}$ と直接対応する $\nabla_{\psi^1_{a\to b}}$ ものは，含まれる半順序関係 (partial order) としては最大（これより順序制約が寛大な LTS を生成する ∇ 演算がないという意味で）である．

以上の直感的な議論を，形式的に展開すると以下になる

- モデル M に対しする ∇_ψ 適用結果を $M' = \nabla_\psi(M)$ とし，ψ に対応する LTS を P_ψ，M と P_ψ の共通アクションを S_{shared} とすると以下のように表せる

$$M' = \partial_{S_{shared}}(M \mid \overline{P_\psi})[\alpha \mid \overline{\alpha_1}/\alpha_2, \alpha \in S_{shared}]$$

  さらに $CPre$ 演算を用いて初期プロセスから到達可能な部分 LTS を求めたものを LTS1 とする．

- モデル M に対する ψ を充足するを計算するための LTS を M'' とする，M の $\alpha \in S_{shared}$ に関する遷移関係 $(s \xrightarrow{\alpha} t)$ の拡張 $(s \xrightarrow{\alpha} st \xrightarrow{\alpha_1} t)$ を M_{mon} とし，P_ψ に対してアクション α を $\overline{\alpha_1}$ に rename した LTS を $\overline{P'_\psi}$ とし，$\alpha \in S_{shared}$ を α_1 に rename した集合を S'_{shared} とすると，M'' は，

$$M'' = \partial_{S'_{shared}}(M_{mon} \mid \overline{P'_\psi})[\alpha \mid \overline{\alpha_1}/\alpha_2, \alpha \in S_{shared}]$$

  M'' において M を Playre1,P_ψ を Player2 としたときの，Player1 側の勝利集合を Win_1 とし，勝利集合に含まれるように Player1 側のアクション選択を制限した LTS を LTS2 とする

- LTS1 と LTS2 は，M_{mon} に対する拡張を除けば同一の状態（プロセス）で構成される

- さらに LTS1 の M_{mon} に対する拡張を観測不能な τ に rename すると，LTS1 と LTS2 の遷移の間に弱双模倣性 (week bisimlation) または観測等価性 (observation equivalence) が成立する

- すなわち M に対する ∇_ψ 演算の結果 M' はプロパティ ψ を充足する最大の M の部分モデルであるといえる

- よって，$M' = \nabla_\psi(M)$ は ψ を充足するが，逆は成立しない．すなわち ψ を充足する部分モデルは M' のみとは限らない．

以上の議論は，$\psi^n_{x_i \to y_j}$ と $\psi^n_{\mathrm{OW}:x_i \to y_j}$ に対して成り立つ．

定理 3.5.2 (∇ 演算と順序制約の充足性 2) [32] $M = \nabla_{\psi^n_{x_i \to y_j}}(M)$ は制約 $\psi^n_{x_i \to y_j}$ を充たす M の最大の部分モデル $M' \subset M$ である．

3.5.1 ∇ 演算と順序制約 ψ の関係

LTS が有限 (finite) であるとは，LTS が，有限の状態で構成されることである．

例えば，ループする LTS はループの n 番目の実行を区別しなければ有限の状態で構成されるが，n 番目の実行を区別すれば無限の状態をもつ遷移系になる．

無限の状態を持つ LTS 同士の並列結合も，同様に無限の状態を持つ．

無限の状態をもつ LTS の並列結合に ∇ 演算を行った結果の LTS では，順序制約があるので，２つの LTS 間のループの実行回数の差により，演算後の LTS の n 回目のループを区別することができる．

すなわち，∇ に対しては，最大 $N_{LTS1} \times N_{LTS2} \times n$ の状態が考えられる．ここで N_{LTS} は LTS の n 番目の実行を区別しない正味の状態の数であるとする．n が有限である場合には，∇ を適用した LTS は有限である．

$\nabla_{\psi^n_{\mathrm{OW}:x_i \to y_j}}(P|Q)$ は $x \in Act_P$ と $y \in Act_Q$ の実行回数の差が n を越えると上書きされるが，$\nabla_{\psi^n_{x_i \to y_j}}(P|Q)$ は x と y の実行回数の数が n を越える遷移を禁止している．並列結合する P, Q 間でデータをやりとりしている場合には，n をバッファの数と思えば，前者はバッファ長を越えて X が過剰実行する場合データが失われることを，後者は，過剰実行自体がない，すなわちデータが失われない並列結合を強制していることになる．ここで” 上書きされない”，とは，$\nabla_{\psi^n_{x \to y}}$ 演算において ocr(x)=n の場合に，アクション x が実行されないことである．

∇ 演算と ψ の充足性に関しては以下のように整理できる．

- $\nabla_{\psi^n_{x \to y}}(P|Q)$ 演算は $\psi^n_{x \to y}$ を充足する
- $\nabla_{\psi^n_{\mathrm{OW}:x \to y}}(P|Q)$ 演算は $\psi^n_{\mathrm{OW}:x \to y}$ を充足する
- $\nabla_{\psi^n_{\mathrm{OW}:x \to y}}(P|Q)$ 演算は $\psi^n_{x \to y}$ を充足しない．

CCS による確認 (CWB-NC の利用)

図 3.7 の LTS に $\nabla_{\psi^1_{a_i \to d_i}}$ を適用した結果 (図 3.8) に対応する LTS は以下である

```
*図 4.3 相当の LTS
proc s000 = a.s111 + c.s290
proc s111 = b.s011 + c.s301
proc s290 = a.s301
proc s011 = c.s201
proc s301 = b.s201 + d.s100
proc s201 = d.s000
proc s100 = b.s000 + c.s390
proc s390 = b.s290
```

この LTS に対して $P0 = \psi^1_{a_i \to d_i}$, $Q0 = \psi^2_{a_i \to d_i}$, $R0 = \psi^1_{d_i \to a_i}$, の充足性を計算する

```
*隠蔽用の定義
set Internals_ad = {a, d}

*性質 φ^1_{a_i->d_i}
proc P0 = 'a.P1 + 'd.nil
proc P1 = 'd.P0 + 'a.nil

*性質 φ^2_{a_i->d_i}
proc Q0 = 'a.Q1 + 'd.nil
proc Q1 = 'd.Q0 + 'a.Q2
proc Q2 = 'd.Q1 + 'a.nil

*性質 φ^1_{d_i->a_i}
proc R0 = 'd.R1 + 'a.nil
proc R1 = 'a.R0 + 'd.nil

*性質1の調査のためのプロダクト　かつ 動作制限(a,dに関する同期通信を強要)
proc Prod_P0 = (s000 | P0) \ Internals_ad
proc Prod_Q0 = (s000 | Q0) \ Internals_ad
proc Prod_R0 = (s000 | R0) \ Internals_ad
```

以下のように ∇ 演算後の LTS において性質 $P0, P1$ は充足し $R0$ は充足しないとも充足することを確認できる.

```
cwb-nc>load dead.mu
cwb-nc>search Prod_P0 can_deadlock
..略..
No state found satisfying can_deadlock.

cwb-nc>search Prod_Q0 can_deadlock
..略..
No state found satisfying can_deadlock.

cwb-nc>search Prod_R0 can_deadlock
..略..
States explored: 1
State found satisfying can_deadlock.
Path to state contains 1 states, invoking simulator.
```

3.6 ∇ 演算の連動性

A, B の並列結合演算 (|) に関する $\nabla(A \mid B)$ 演算の連動性を以下に定義する．

定義 3.6.1 (連動性) $\nabla(A \mid B)$ 演算結果が有限であるとき，以下の弱合同性 (weak congruence) 関係が成り立つときに，A と B は連動する

- $\tau_{[A]}\nabla(A \mid B) \simeq B$ かつ
- $\tau_{[B]}\nabla(A \mid B) \simeq C$

有限性は，連動性の必要条件である．

例えば並列動作仕様 $E = \{X = a \cdot bX, Y = c \cdot dY\}$ に対する $Z = (X \mid Y) = (a \cdot b \mid c \cdot d)Z$ は，$a \cdot b$ 系列あるいは $c \cdot d$ 系列が単独で無限実行する動作パターンを含むので，上記の意味での弱後合同性が成立しないので，連動性を持たない

一方 $\nabla_{\psi^1_{a \to d}}(X \mid Y)$ の適用結果である LTS(図 3.8) は，無限の $a \cdot b$ 系列も $c \cdot d$ 系列も含まないので連動性持つ．

このように連動性をもつことは並列結合 (|) した LTS が，元の構成要素の LTS の動作を保存することを示している．

連動性は，推移的 (transitive) な性質を持つことは自明である．すなわち A と B が連動し，かつ，B と C が連動するならば，A と C は連動する．

3.6.1 連動性の検証 (CWB-NC による)

連動性を CWB-NC を用いて検証する．連動性の検証は，対象とするプロセスに対して，注目するアクション以外を内部アクション τ に変換した射影演算結果に対して，livelock が有るかどうかを検証することにより行う．

ここで livelock の有無を判断するプロパティは can_livelock を様相論理を用いて以下のように定義する．

```
prop can_livelock =
   min X = livelock_now \/ <->X
prop livelock_now =
   max X = <t>X
```

ここで<t>は内部遷移 τ を表している．

livelock_now は内部遷移による無限ループに陥る可能性のある状態に対応し，can_livelock は Pos(livelock_now) に対応し，無限ループに陥る可能性のある状態に到達する可能性があることを示している．

can_livelock は最大不動点と最小不動点を組み合わせることにより定義されていることに注意．(詳しくは Aceto らの定義を参照 [2](p.136))

最初に連動しない例を示す．

```
proc X = a.b.X
proc Y = c.d.Y
*Z は直積です
proc Z = (X | Y)

set Internals_ab = {a, b}
set Internals_cd = {c, d}

*ab,cd をそれぞれ隠蔽するためのものです
proc Hide_ab = 'a.Hide_ab + 'b.Hide_ab
proc Hide_cd = 'c.Hide_cd + 'd.Hide_cd

*Z から cd を隠蔽した遷移系を得ます
proc Proj_ab = (Z | Hide_cd) \ Internals_cd
*Z から ad を隠蔽した遷移系を得ます
proc Proj_cd = (Z | Hide_ab) \ Internals_ab
```

ここで Proj_ab は c, d を τ に書き換えた射影結果である．Proj_ad に対して livelock をチェックする

```
cwb-nc> search Proj_ab can_livelock
Building automaton...
States: 5
Transitions: 10
Done building automaton.

States explored: 1
State found satisfying can_livelock.
Path to state contains 1 states, invoking simulator.
1: Proj_ab
cwb-nc-sim>
```

can_livelock 性質を充足する状態が発見された.

一方図 3.8 に対応するプロセスは以下の s000 になる

```
proc s000 = a.s111 + c.s290
proc s111 = b.s011 + c.s301
proc s290 = a.s301
proc s011 = c.s201
proc s301 = b.s201 + d.s100
proc s201 = d.s000
proc s100 = b.s000 + c.s390
proc s390 = b.s290
```

a, b および c, d に射影したプロセスを作ります

```
*s000 から cd を隠蔽した遷移系を得ます
proc Proj2_ab = (s000 | Hide_cd) \ Internals_cd

*s000 から ad を隠蔽した遷移系を得ます
proc Proj2_cd = (s000 | Hide_ab) \ Internals_ab
```

Proj2_ab に対して livelock を検査する

```
cwb-nc> search Proj2_ab can_livelock
Building automaton...
States: 9
Transitions: 14
Done building automaton.
Building automaton...
States: 8
Transitions: 12
Done building automaton.
Building automaton...
States: 8
Transitions: 12
Done building automaton.
Building automaton...
States: 8
Transitions: 12
Done building automaton.
Building automaton...
..略..
States: 8
Transitions: 12
Done building automaton.

States explored: 9
No state found satisfying can_livelock.
```

今度は livelock しないことが確認できた

3.7 ∇ 演算の交換性

∇ 演算を複数適用する場合において以下の性質を持つものを並列演算 (|) に関して，交換可能であると定義する．

定義 3.7.1 (∇ 演算の交換可能性) プロセス A, B, C を含む LTS に対する，∇ 演算，∇_1, ∇_2 において，$X = \nabla_1(A \mid \nabla_2(B \mid C))$ と $Y = \nabla_2(\nabla_1(A \mid B) \mid C)$ 各々に対して，$\nabla_2(B \mid C)$ が連動し，かつ $\nabla_1(A \mid B)$ が連動する時に，∇_1, ∇_2 は交換可能であると定義する．

交換可能であるとは，もともとの 2 つの ∇ 演算（$\nabla_2(B \mid C)$，$\nabla_1(A \mid B)$）に関わる性質 ψ_1, ψ_2 が保存される範囲では，∇_1, ∇_2 どちらを先に適用しても結果は同じという意味である．

定理 3.7.1 (交換可能性の定理) ∇_1, ∇_2 がそれぞれ連動性を持つならば交換可能である

以下の手順により示される．

- ∇_2 が連動性をもつならば，$\nabla_2(B \mid C)$ の演算結果から $act \in Act_C$ の要素を隠蔽すると B と弱合同等価になる，$\tau_{[C]}\nabla_2(B \mid C) \simeq B$
- 弱合同等価は合同性 (congruence) を持つから $\nabla_1(A \mid B) \simeq \tau_{[C]}\nabla_1(A \mid \nabla_2(B \mid C))$
- 一方 ∇_2 の連動性より，$\tau_{[C]}\nabla_2(\nabla_1(A \mid B) \mid C) \simeq \nabla_1(A \mid B)$ であり $\nabla_1(A \mid B)$ が保存される，
- 同様に $\nabla_2(B \mid C)$ も保存されるので，∇_1, ∇_2 は交換可能である

3.8 ACP における順序制約 ψ および ∇ 演算

本節では CCS における順序制約 ψ および ∇ 演算を ACP に適用する場合の差分について検証する．

ACP では通信を伴わない同期並列実行 (|) が CCS に加えて許されているから，3.2.4 節における図 3.5 に以下のルール (図 3.9) を追加する．

$$\frac{P \xrightarrow{\alpha} P', Q \xrightarrow{\beta} Q'}{\langle P \mid Q, k\rangle \xrightarrow{\alpha|\beta} \langle P' \mid Q', k\rangle} \quad 0 < k < n$$

$$\frac{P \xrightarrow{\alpha} P', Q \xrightarrow{\beta} Q'}{\langle P \mid Q, n\rangle \xrightarrow{\alpha|\beta} \delta} \qquad \frac{P \xrightarrow{\alpha} P', Q \xrightarrow{\beta} Q'}{\langle P \mid Q, 0\rangle \xrightarrow{\alpha|\beta} \delta}$$

図 3.9: ACP における順序制約 $\psi^n_{\alpha_i \to \beta_j}$ の追加ルール

∇ 演算は ψ の展開ルールに従った LTS との同期並列結合演算に基づいて定義されるので，∇ 演算でも ACP においては図 3.9 に従う．

たとえば，再帰方程式 $E = \{X = a \cdot X, Y = b \cdot Y\}$ に対して，CCS として $X \mid Y$ を展開すると以下の初期状態<P,0>の LTS を得る．

```
LTS13:
 Proc = {<P,i>}
 Act = {a,b}
 -a-> {(<P,i>,<P,i+1>)}
 -b-> {(<P,i>,<P,i-1>)}
```

一方 ACP として $X \mid Y$ を展開すると以下の初期状態<Q,0>の LTS を得る

```
LTS14:
 Proc = {<Q,i>}
 Act = {a,b}
 -a-> {(<Q,i>,<Q,i+1>)}
 -b-> {(<Q,i>,<Q,i-1>)}
 -a|b->{(<Q,i>,<Q,i>)}
```

ACP における $\psi^1_{a_i \to b_i}$ に対応する LTS は，$0 < k < 1$ を充たす k が存在しないので，CCS の LTS と同じになる．

$$\frac{P \xrightarrow{\alpha} P', Q \xrightarrow{\beta} Q'}{\langle P \mid Q, n\rangle \xrightarrow{\alpha|\beta} \langle P' \mid Q', n\rangle}$$

図 3.10: ACP における ∇_{OW} 演算のルール

```
LTS_ψ^1_ai->bi
 Proc = {P0,P1,NG}
 Act = {a,b}
 -a->{(P0,P1),(P1,NG)},-b->{(P1,P0),(P0,NG)}
 -a|b->{(P0,NG),(P1,NG)}
```

ACP における $\nabla_{\psi^1_{a_i \to b_i}}$ 演算も同様に CCS の LTS と同じになる．

一方 ACP における $\nabla_{\psi^1_{\mathrm{OW}:a_i \to b_i}}$ 演算は，$k = n$ の場合に図 3.10 のルールに従うものとする．

対応する LTS は同時並列実行 a|b に伴う遷移を追加された以下の LTS になる．

```
LTS_▽ψ^1_ai->bi
 Proc = {P0,P1,NG}
 Act = {a,b}
 -a->{(P0,P1),(P1,NG)},-b->{(P1,P0),(P0,NG)}
 -a|b->{(P1,P1),(P0,NG)}
```

ここで展開ルール (図 3.10) により a|b は P1 すなわち $k = 1$ の時のみ許されることに注意．

<P,0>に $\nabla_{\psi^1_{a_i \to b_i}}$ を適用すると，以下の LTS を得ることができ，これは $\psi^1_{a_i \to b_i}$ を充足する最大の部分モデルである．

```
LTS15:
 Proc = {<P,0>,<P,1>}
 Act = {a,b}
 -a-> {(<P,0>,<P,1>)}
 -b-> {(<P,1>,<P,0>)}
```

一方<Q,0>に $\nabla_{\psi^1_{\mathrm{OW}:a_i \to b_i}}$ を適用すると，図3.10に従いa|bに関する遷移は$exocr(a) > 0$の場合のみ可能であることを考慮すると，以下のLTSを得る．ここでa, bの同時並列実行a|bは<Q1,1>のときのみの自己遷移しか許されなくなることに注意．

```
LTS16:
 Proc = {<Q,0>,<Q,1>}
 Act = {a,b}
 -a-> {(<Q,0>,<Q,1>)}
 -b-> {(<Q,1>,<Q,0>)}
 -a|b->{(<Q,1>,<Q,1>)}
```

以上の議論を図示したものが以下の図3.11である．

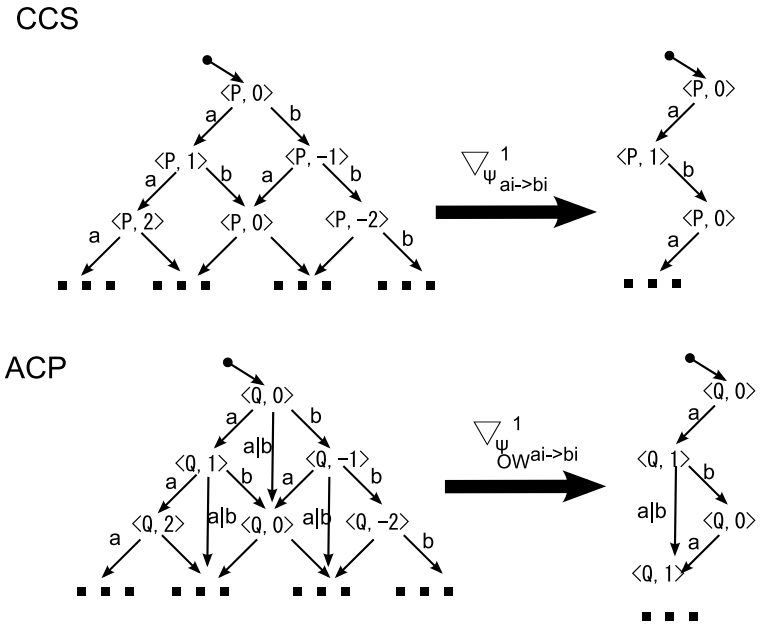


図 3.11: ACP と CCS の関係

第 4章

シナリオ合成の正当性の検証

本章では，eMSC で用いられる non blocking 型の書き込みと blocking 型の読み込みを組み合わせた通信プロトコル挿入により，横・縦チェーンが実現できるかを，∇^1_{OW} 演算の適用結果が制約 ψ_h や ψ_v を充足するかどうかを計算することにより検証する．CWB-NC を検証に用いるためプロセス代数として CCS を以下では用いている．

4.1 縦・横チェーンの形式化

横チェーン (ψ_h)，縦チェーン (ψ_v) の制約は，以下であった．

$A\psi_h B = en_n(A)$ は $st_{n+1}(B)$ に先行し $\wedge$ $en_n(B)$ は $st_{n+1}(A)$ に先行する

$A\psi_v B = en_n(A)$ は $st_n(B)$ に先行し $\wedge$ $en_n(B)$ は $en_{n+1}(A)$ に先行する

$E = \{X = a \cdot bX, Y = c \cdot dY\}$ であるとする

横チェーン ($X\psi_h Y$) は，$\psi^n_{d_k \to a_{k+1}} \wedge \psi^n_{b_k \to c_{k+1}}$ であり，縦チェーン ($X\psi_v Y$) は，$\psi^n_{b_k \to c_k} \wedge \psi^n_{d_k \to b_{k+1}}$ となる．

$\psi^n_{x_i \to y_j}$ 型の制約として形式化を行うには，x_i の先行の上限 n が必要である．n は横チェーン，縦チェーン個別に検討する．

4.2 横チェーンの検証

$E = \{X = a \cdot bX, Y = c \cdot dY\}$ においては，横チェーン制約は

$\psi^n_{b_i\to c_{i+1}} \wedge \psi^n_{d_j\to a_{j+1}}$ であり，これは

$$(\psi^\infty_{b_i\to c_{i+1}} \wedge \psi^\infty_{c_{k+1-n}\to b_k})\wedge$$
$$(\psi^\infty_{d_j\to a_{j+1}} \wedge \psi^\infty_{a_{l+1-n}\to d_l})$$

に展開される．

$\psi^\infty_{c_{k+1-n}\to b_k}$ は c に対する b の先行を押さえるが，$n=1$ であるとすると $\psi^\infty_{c_i\to b_i}$ となり，同一巡目 (例えば i) においては c_i, b_i の間に順序関係は不要であるから $n=1$ は過剰制約である．巡目の逆転が起きないことが要求されるから $c_k \to b_{k+1}$ すなわち $c_{k-1} \to b_k$ すなわち $n=2$ が最小の制限になる．$\psi^\infty_{a_{j+1-n}\to d_j}$ も同様である．

従って a,d 間，b,c 間の個別の順序制約はそれぞれ $\psi^2_{b_i\to c_{i+1}}, \psi^2_{d_j\to a_{j+1}}$ となる．

次に a,d 間の制約と b,c 間の制約を合わせた考察をする

$n=2$ の場合には，$\psi^\infty_{a_{j+1-n}\to d_j}$ は $\psi^\infty_{a_{i-1}\to d_i}$ すなわち $\psi^\infty_{a_i\to d_{i+1}}$ を規定することになる．しかし

X,Y が元来備える順序性および，$\psi^\infty_{b_i\to c_{i+1}}$ より，$a_i \to b_i$ かつ $b_i \to c_{i+1}$ かつ $c_{i+1} \to d_{i+1}$ であることを考えると．$a_i \to d_{i+1}$ は b,c 間の順序制約および X,Y がもつ元々の順序制約から導出できる性質であり新たな動作制約にならない．b,c 間も同様である．

すなわち a,d 間の制約と b,c 間の制約を合わせた場合には $n \geq 2$ であればよいことになる．

最初に $E = \{X = a\cdot bX, Y = c\cdot dY\}$ の X,Y 間において ψ_h を充足する最大の部分モデルを計算する．つぎに，ψ_h に対応する ∇^1_{OW} 型の演算を X,Y 間に施した結果が ψ_h を充足するかどうかを検証し，最後に実際の eMSC で採用された連係プロトコルに対応する ∇^1_{OW} 演算による充足性を検証する．

横チェーンの制約 $\psi_h := \psi^2_{d_n\to a_{n+1}} \wedge \psi^2_{b_n\to c_{n+1}}$ は，以下の 2 つの LTS(それぞれ初期プロセスは P1,Q1) で表すことができる．

```
LTS9:                              LTS10:
Proc={P0,P1,P2,NG},                Proc={Q0,Q1,Q2,NG}
Act ={a,d},                        Act ={c,b}
-a->{(P1,P0),(P0,NG),(P2,P1)},-c->{(Q1,Q0),(Q0,NG),(Q2,Q1)},
-d->{(P0,P1),(P1,P2),(P2,NG)},-b->{(Q0,Q1),(Q1,Q2),(Q2,NG)}
```

前章より，順序制約の LTS を ∇ 演算の LTS に読み替えて，ACP の同期実行を考慮した

LTS4 のプロセス=$\langle M0, 0\rangle$ に対して同期並列結合することにより，横チェーン制約 ψ_h を充足する最大の解 (図 4.1) が得られる．

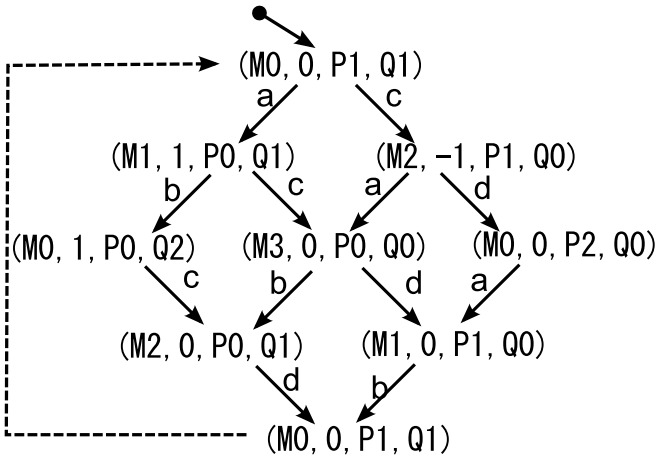


図 4.1: 横チェーン制約を充足する最大の部分モデル

LTS9ow および LTS10ow がそれぞれ $\nabla_{\psi^1_{\mathrm{OW}:d_k \to a_{k+1}}}$, $\nabla_{\psi^1_{\mathrm{OW}:b_k \to a_{k+1}}}$ に対応する LTS である．

```
LTS9ow:                 LTS10ow:
Proc={P0,P1,NG},        Proc={Q0,Q1,NG}
Act ={a,d},             Act ={c,b}
-a->{(P1,P0),(P0,NG)},-c->{(Q1,Q0),(Q0,NG)},
-d->{(P0,P1),(P1,P1)},-b->{(Q0,Q1),(Q1,Q1)}
```

図 4.2 は ∇ 演算適用後の LTS である．

図 4.2 が横チェーン制約を充足しているかどうかを検証する．図 4.2 を LTS と CWB のプロセスに書き下すと以下の s0011 になる

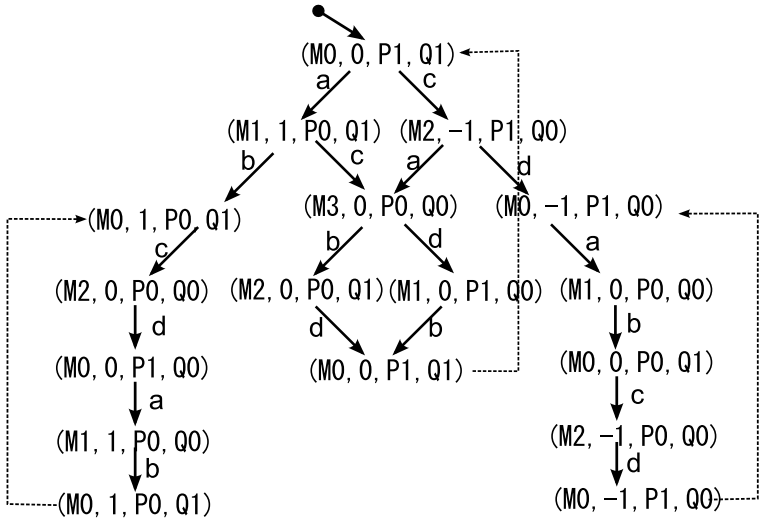


図 4.2: 横チェーン挿入後の LTS

```
*LTS_yoko に相当するプロセスを定義します
proc s0011 = a.s1101 + c.s2910
proc s1101 = b.s0101 + c.s3000
proc s2910 = a.s3000 + d.s0910
proc s0101 = c.s2000
proc s3000 = d.s1010 + b.s2001
proc s0910 = a.s1000
proc s2000 = d.s0010
proc s1010 = b.s0011
proc s1000 = b.s0001
proc s2001 = d.s0011
proc s0010 = a.s1100
proc s0001 = c.s2900
proc s1100 = b.s0101
proc s2900 = d.s0910
```

ここで LTS9,LTS10 に対応するプロセスをそれぞれ P1,Q1 と定義し，プロセス s0011 との同期通信を伴う並列結合をとりプロセス Prod_ad,Prod_bc を得る.

```
*隠蔽用の定義
set Internals_bc = {b, c}
set Internals_ad = {a, d}
set Internals_bd = {b, d}

*上書きされない性質その 1 for ad
proc P1 = 'a.P0 + 'd.P2
proc P2 = 'a.P1 + 'd.nil
proc P0 = 'a.nil + 'd.P1

*上書きされない性質その 1 for bc
proc Q1 = 'c.Q0 + 'b.Q2
proc Q2 = 'c.Q1 + 'b.nil
proc Q0 = 'c.nil + 'b.Q1

*性質 1 の調査のためのプロダクト　かつ 動作制限 (a,d に関する同期通信を強要)
proc Prod_ad = (s0011 | P1) \ Internals_ad

*性質 2 の調査のためのプロダクト かつ 動作制限 (b,c に関する同期通信を強要)
proc Prod_bc = (s0011 | Q1) \ Internals_bc
```

∇演算適用後の遷移系（図4.2）の性質P1,Q1の充足性を検証するためにProd_ad,Prod_bcに対してそれぞれnilに遷移しないことを，deadlockにならない性質で検証する.

```
cwb-nc> search Prod_ad can_deadlock
..略..
States explored: 15
No state found satisfying can_deadlock.
cwb-nc> search Prod_bc can_deadlock
..略..
States explored: 15
No state found satisfying can_deadlock.
```

このように図 4.2 は横チェーンの制約 ψ_h を充足する.

次に連動性の検証を行う，ここでは，$\nabla_{\psi^1_{\mathrm{OW}:d_k \to a_{k+1}}} \nabla_{\psi^1_{\mathrm{OW}:b_k \to a_{k+1}}} (X \,|\, Y)$ の結果である LTS が，もとの X, Y に対して連動性を持つことを示す.

以下のプロセスを追加し s0011 の動作をそれぞれ a,b および c,d の信号に射影 (Proj_ab,Proj_cd) したときに，元々の X の動作の仕様 (Spec_ab)，Y の動作の仕様 (Spec_cd) と観測同値関係があるかどうかを検証する.

```
*ab,cd をそれぞれ隠蔽するためのものです
proc Hide_ab = 'a.Hide_ab + 'b.Hide_ab
proc Hide_cd = 'c.Hide_cd + 'd.Hide_cd

set Internals_ab = {a, b}
set Internals_cd = {c, d}

*s011 から cd を隠蔽した遷移系を得ます
proc Proj_ab = (s0011 | Hide_cd) \ Internals_cd

*s011 から ad を隠蔽した遷移系を得ます
proc Proj_cd = (s0011 | Hide_ab) \ Internals_ab
```

隠蔽 (射影) したものが，can_livelock の性質を持たないことを確かめる．

```
cwb-nc> search Proj_ab can_livelock
..略..
States: 14
Transitions: 18
Done building automaton.

States explored: 15
No state found satisfying can_livelock.

cwb-nc> search Proj_cd can_livelock
..略..
States: 14
Transitions: 18
Done building automaton.

States explored: 15
No state found satisfying can_livelock.
```

このように，∇ 演算により得られた横チェーン挿入後の LTS は，ψ_h を充足するとおもに，もともとのプロセス X, Y に対して連動性の性質を持つことが確認された．

4.3 縦チェーンの検証

$E = \{X = a \cdot bX, Y = c \cdot dY\}$ においては，縦チェーン制約は
$\psi^{n_0}_{b_i \to c_i} \wedge \psi^{n_1}_{d_j \to b_{j+1}}$ であり，これは

$$(\psi^{\infty}_{b_i \to c_i} \wedge \psi^{\infty}_{d_j \to b_{j+1}}) \wedge$$
$$(\psi^{\infty}_{c_{k-n_0} \to b_k} \wedge \psi^{\infty}_{b_{l+1-n_1} \to d_l})$$

に展開される．

$\psi^{\infty}_{c_{k-n_0}\to b_k}$ は c に対する b の先行を押さえる制約であるが，b による上書きを禁止する性質より $c_k \to b_{k+1}$ すなわち $c_{k-1} \to b_k$ となり $n_0 = 1$ が上限値になる．

一方，$\psi^{\infty}_{b_{l+1-n_1}\to d_l}$ は b に対する d の先行を制約するが，b_l の実行以前に d_l が実行されることはないから $d_l \to b_l$ となり $n_1 = 1$ となる．

横チェーンと同様に順序制約同士のつながりを考慮すると，

$c_i \to d_i \to b_{i+1}$ であるので，$\psi^{\infty}_{c_k\to b_{k+1}}$ は自明となり $n_0 \geq 1$ でよいことになる．

また $b_l \to c_l \to d_l$ であるので同様に $n_1 \geq 1$ でよい．

最初に縦チェーン制約を充たす最大の部分モデルをもとめる．ここでは $n_0 = n_1 = 1$ とすると，以下の 2 つの LTS(初期プロセスは P0,Q1) で表せる

```
LTS11:                          LTS12:
Proc={P0,P1,NG},            Proc={Q0,Q1,NG}
Act ={b,c},                 Act ={b,d}
-b->{(P0,P1),(P1,NG)},      -b->{(Q1,Q0),(Q0,NG)},
-c->{(P1,P0),(P0,NG)},      -d->{(Q0,Q1),(Q1,NG)}
```

LTS は以下のように展開されるここで状態=(正味の状態,，exocr(b), exocr(d)) とし，初期状態を (0,0,1) であるとする．

横チェーンの場合と同様に，LTS4,LTS11,LTS12 の同期並列結合を行い下図 4.3 の ψ_v を充足する最大の LTS を得る．

LTS11ow および LTS12ow がそれぞれ $\nabla_{\psi^1_{\mathrm{OW}:b_k\to c_k}}$,$\nabla_{\psi^1_{\mathrm{OW}:d_k\to b_{k+1}}}$ に対応する LTS である．

```
LTS11ow:                    LTS12ow:
Proc={P0,P1,NG},            Proc={Q0,Q1,NG}
Act ={b,c},                 Act ={b,d}
-b->{(P0,P1),(P1,P1)},      -b->{(Q1,Q0),(Q0,NG)},
-c->{(P1,P0),(P0,NG)},      -d->{(Q0,Q1),(Q1,Q1)}
```

LTS4 と LTS11ow,LTS12ow の同期並列結合演算結果は，図 4.3 と同一の構造をもつ LTS を得る．即ち ∇_{OW} 型の演算を施した物は，縦チェーン制約 ψ_v を充足する．

縦チェーンの連動性を，横チェーンと同様に検証する．図 4.3 を CWB のプロセスとし

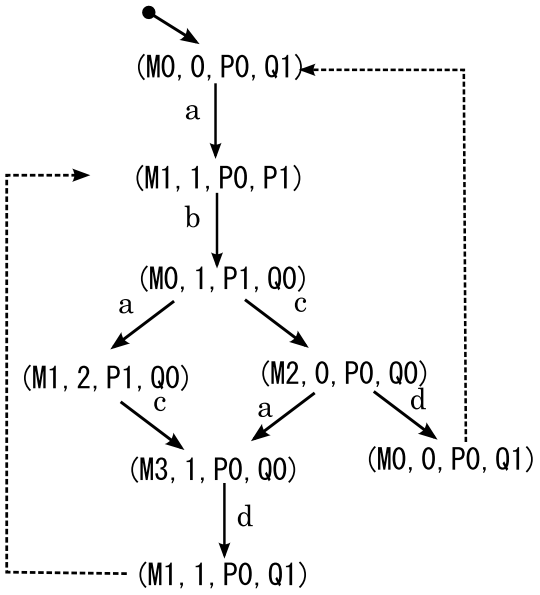


図 4.3: 縦チェーン制約を充足する最大の部分モデル

て表現すると以下の s0001 になる

```
*LTS_tate に相当するプロセスを定義します
proc s0001 = a.s1101
proc s1101 = b.s0110
proc s0110 = a.s1210 + c.s2000
proc s1210 = c.s3100
proc s2000 = a.s3100 + d.s0001
proc s3100 = d.s1101
```

a, b および c, d を捨象するために以下を用意します.

```
*ab,cd をそれぞれ隠蔽するためのものです
proc Hide_ab = 'a.Hide_ab + 'b.Hide_ab
proc Hide_cd = 'c.Hide_cd + 'd.Hide_cd

set Internals_ab = {a, b}
set Internals_cd = {c, d}

*s0001 から cd を隠蔽した遷移系を得ます
proc Proj_ab = (s0001 | Hide_cd) \ Internals_cd

*これはもとの X=a.b.X の動作仕様
proc Spec_ab = a.b.Spec_ab

*s0001 から ab を隠蔽した遷移系を得ます
proc Proj_cd = (s0001 | Hide_ab) \ Internals_ab

*これはもとの Y=c.dY の動作仕様
proc Spec_cd = c.d.Spec_cd
```

隠蔽 (射影) したものが，can_livelock の性質を持たないことを確かめる.

```
cwb-nc> search Proj_ab can_livelock
..略..
States: 6
Transitions: 8
Done building automaton.

States explored: 7
No state found satisfying can_livelock.

cwb-nc> search Proj_cd can_livelock
..略..
States: 6
Transitions: 8
Done building automaton.

States explored: 7
No state found satisfying can_livelock.
```

第 5章

シナリオ合成結果の実装

シナリオ合成は，コマンド図で定義された互いに並列に動作する個別の状態遷移機械が，シナリオ図で定義された実行順序制約（のシステム）に沿った動作をするように，状態遷移機械間に適切な動作連係の通信プロトコルを挿入した状態遷移システムを作成する．

これまで (3 章,4 章) は，主に CCS を用いてコマンド図で定義された互いに並列に動作するプロセスに対して，シナリオ図で定義された順序制約 ψ と，ψ を充足する 1 プロセスに合成するための ∇ 演算について議論した．

ここでは，シナリオ合成結果の実装[1]，すなわち ACP_{drt} に従った離散化について議論する．実装の方法として，1)∇ 演算の結果を 1 プロセスとして離散化する，2)ACP_{drt} において ∇ 演算と同等の効果を持つチャネルとチャネルに対する読み書きのプロトコルを挿入した 2 プロセスとして離散化する，を述べる．

最初に ACP で記述されたプロセスを 1 プロセスとしての離散化について述べる．次にチャネルとチャネルに対する読み書きプロトコルの導入による離散化を述べる．

5.1　プロセス代数から実装へ

ACP や CCS のプロセス代数では順序制約のみを規定しているので，本質的に可能な動作列を複数含有する（非決定性があるということ）．ところが実装においては，この複数の可能性すなわち非決定性は排除される．すなわち可能な動作列のうち一つが選択されて

[1]ここでは，いわゆる物理的な実装 physical implementation ではなく、広義の実現 implementation を指す

実現される．実装においては，この複数の可能性，すなわち非決定性は削除されなければならない．すなわち可能が動作のうち一つを選択することが必要となる．

例えば横チェーンの順序制約を充足する最大の LTS は 4.1 図の LTS で表す事ができた．

コマンド活性体の合併は，最大の LTS の選択実行 (+) を決定化して 1 つの逐次的に連続する LTS を求める操作である．このとき，コマンドは離散かつ並列に動作可能であるから，前節のように $\sigma_d(a \mid c) \cdot \sigma_d(b \mid d)$ のような，並列行実行も得られることを考慮すると，例えば，1 階のループで，a〜d の一巡をする LTS に限れば，以下の 13 種類の可能性のある LTS を得る．

$$
\begin{aligned}
X &= \sigma_d(a \mid c) \cdot \sigma_d(b \mid d)X \\
X &= \sigma_d(a) \cdot \sigma_d(c) \cdot \sigma_d(b) \cdot \sigma_d(d)X \\
X &= \sigma_d(a) \cdot \sigma_d(c) \cdot \sigma_d(d) \cdot \sigma_d(b)X \\
X &= \sigma_d(a) \cdot \sigma_d(c) \cdot \sigma_d(d \mid b)X \\
X &= \sigma_d(a \mid c) \cdot \sigma_d(b) \cdot \sigma_d(d)X \\
X &= \sigma_d(a \mid c) \cdot \sigma_d(d) \cdot \sigma_d(b)X \\
X &= \sigma_d(c) \cdot \sigma_d(a) \cdot \sigma_d(d) \cdot \sigma_d(b)X \\
X &= \sigma_d(c) \cdot \sigma_d(a) \cdot \sigma_d(b) \cdot \sigma_d(d)X \\
X &= \sigma_d(c) \cdot \sigma_d(a) \cdot \sigma_d(b \mid d)X \\
X &= \sigma_d(a) \cdot \sigma_d(b \mid c) \cdot \sigma_d(d)X \\
X &= \sigma_d(c) \cdot \sigma_d(a \mid d) \cdot \sigma_d(b)X \\
X &= \sigma_d(a) \cdot \sigma_d(b) \cdot \sigma_d(c) \cdot \sigma_d(d)X \\
X &= \sigma_d(c) \cdot \sigma_d(d) \cdot \sigma_d(a) \cdot \sigma_d(b)X
\end{aligned}
$$

このように，シナリオ合成結果をプロセス代数における再帰方程式で形式化することにより，可能なマージの組合せの全体を再帰方程式として計算でき (網羅的)，また得られた再帰方程式から面積や性能の要求仕様を考慮して任意の選択結合 (+) された部分項を選択し，シナリオ図が規定する連係制約を充足する 1 つの状態遷移機械として合併することが可能になることがわかる．

上記例では $\sigma_d(a \mid c) \cdot \sigma_d(b \mid d)$ は時間性能は高いが，アクションを同時に実行しなくてならないので面積としては他の候補に比べて不利である．

5.2　チャネルへの書込みプロトコルの挿入操作の導入

前節ではシナリオ合成の結果を 1 つのプロセスとして合成した．ここでは元々のコマンド起因のプロセスを維持したまま，シナリオが規定する順序制約を充足する手段としてチャネルとチャネルに対する読み書きプロトコルの挿入を導入する

操作 InsertBrW(E,a,b,i,B):E は再帰方程式名，a,b はアクション名，i はチャネルの Id 名，B はバッファの初期値を，以下のように定義する．ここでチャネルは 3.1.4 節で導入されたものを用いる．

定義 5.2.1 (操作 InsertBrW(E,a,b,i,B)) 操作 InsertBrW(E,a,b,i,B):E は再帰方程式名，a,b はアクション名，i はチャネルの Id 名，B はバッファの初期値である．再帰方程式 E に対して，チャネル $ch^1(i, buff)$ を導入し，アクション $br(0)$ を b の前に，$w(0)$ を a の後にそれぞれ a および b と同一離散時間区切り $(\sigma_d())$ 内に挿入する操作である

5.2.1　∇ 演算と InsertBrW の関係

再帰方程式 $E = \{X = \sigma_d(a)X, Y = \sigma_d(b)Y\}$ に対して，InsertBrW(E,a,b,0,0) を操作させると，E' を得る．

$E' = \{X = \sigma_d(w_0(a))X, Y = \sigma_d(br_0(b))Y\}$

チャネルのバッファへデータの有無で状態が区別できるので，以下では，再帰方程式の n 番目の実行を特に $X_n, Y_n(n \in N)$ のように区別しない．

状態を，$\langle$ プロセス項, チャネル状態, 次チャネル状態$\rangle$，チャネル状態は {(チャネル Id, バッファの値), により表すとすると，

X, Y の並列結合 $\langle X \parallel Y, \{(0,0)\}, \{\}\rangle$ は，図 3.2，3.3 のチャネルの操作的意味に従うと，以下のように展開できる．

$$
\begin{aligned}
&\langle X \parallel Y, \{(0,0)\}, \{\}\rangle \\
&\xrightarrow{\tau} \langle (\sigma_d(a_0)X \parallel \sigma_d(br_0(b_0))Y), \{(0,0)\}, \{(0,1)\}\rangle \\
&\xrightarrow{a_0} \langle (\sigma_d(w_0(a_1))X \parallel \sigma_d(br_0(b_0))Y), \{(0,1)\}, \{\}\rangle \\
&\xrightarrow{\tau} \langle (\sigma_d(w_0(a_1))X \parallel \sigma_d(b_0)Y), \{(0,1)\}, \{(0,0)\}\rangle \\
&\xrightarrow{\tau} \langle (\sigma_d(a_1)X \parallel \sigma_d(b_0)Y), \{(0,1)\}, \{(0,1)\}\rangle \\
&\xrightarrow{a_1|b_0} \langle (X \parallel Y), \{(0,1)\}, \{\}\rangle \\
&\cdots \\
&\xrightarrow{a_2|b_1} \langle (X \parallel Y), \{(0,1)\}, \{\}\rangle \\
&\cdots
\end{aligned}
$$

このように展開される．一方，X, Y を CCS に対応させた X_{CCS}, Y_{CCS} に対して $\nabla_{\psi^1_{a_k \to b_k}}(X_{CCS} \| Y_{CCS})$ 演算を適用すると，$a_0 \cdot b_0 \cdot a_1 \cdot b_1 \cdots$ の動作を行う LTS として並列合成される．

n 番目の実行を a_n, b_n と区別すれば，

$$\sigma(a_0) \cdot \sigma_d(b_0 \mid a_1) \cdots$$

と展開されるので，$\psi^1 a_k \to b_k$ は充足される．なお，この展開は前節 (3.8 節) における LTS16 の可能な動作の 1 つである．

このように，$\nabla_{\psi^1_{x_k \to y_k}}$ 演算は，離散プロセス (ACP_{drt}) では，操作 InsertBrW(E,x,y,0,0) により，実現（実装）することができる．

別の例として，$\nabla_{\psi^1_{x_k \to y_{k+1}}}$ 演算は，例えば y_0 は，制限なしで実行され，y_n から x_{n-1}（ここで $n \geq 1$）の実行に順序制約される．これは，離散プロセスでは，挿入するチャネルのバッファ初期値を 1 にすることにより，最初の y_0 が制限なしに実行されることを表すことができるので，操作 InsertBrW(E,x,y,0,1) により，実現（実装）することができる．

先ほどの例と同様に，E に操作 InsertBrW(E,x,y,0,1) を適用した場合の展開を示す．

$$
\begin{aligned}
&\langle X \parallel Y, \{(0,1)\}\,\{\}rangle \\
&\xrightarrow{\tau} \langle (\sigma_d(a_0)X \parallel \sigma_d(br_0(b_0))Y), \{(0,1)\}, \{(0,1)\}\rangle \\
&\xrightarrow{\tau} \langle (\sigma_d(a_0)X \parallel \sigma_d(b_0)Y), \{(0,1)\}, \{(0,1)\}\rangle \\
&\xrightarrow{a_0|b_0} \langle (X \parallel Y), \{(0,1)\}, \{\}\rangle \\
&\cdots \\
&\xrightarrow{a_1|b_1} \langle (X \parallel Y), \{(0,1)\}, \{\}\rangle \\
&\cdots
\end{aligned}
$$

n 番目の実行を a_n, b_n と区別すれば，

$$\sigma_d(a_0 \mid b_0) \cdot \sigma_d(a_1 \mid b_1) \cdots$$

となり $\psi^1_{a_k \to b_{k+1}}$ は充足される．

5.3 横チェーン用プロトコルの挿入

横チェーンは，2 つのプロセスが連動して動作し，かつデータが上書きされないという性質 $\psi^1_{d_k \to a_{k+1}}, \psi^1_{b_k \to c_{k+1}}$ 両方が成立することが必要であった．LTS の場合には，$\nabla^1_{d_k \to a_{k+1}}, \nabla^1_{b_k \to c_{k+1}}$ を適用することにより，これらを充足する LTS が得られることがわかっていた．ここでは，離散実行するプロセスでの実現について InsertBrW を用いる実現を検討する．

再帰方程式 (3.1) において，$\nabla^1_{d_k \to a_{k+1}}, \nabla^1_{b_k \to c_{k+1}}$ を適用することは，

InsertBrW(E,d,a,1,1) かつ InsertBrW(E,b,c,2,1) を適用することに対応する．

$$
\begin{aligned}
X &= \sigma_d(br_1(a)) \cdot \sigma_d(w_2(b))X \\
Y &= \sigma_d(br_2(c)) \cdot \sigma_d(w_1(d))Y
\end{aligned}
$$

$\langle X \parallel Y, \{(1,1),(2,1)\}, \{\}\rangle$ は

$\langle X \parallel Y, \{(1,1),(2,1)\}\rangle$
$\xrightarrow{\tau} \langle (\sigma_d(a)\sigma_d(w_2(b))X \parallel \sigma_d(br_2(c))\sigma_d(w_1(d))Y), \{(1,1),(2,1)\}, \{(1,0)\}\rangle$
$\xrightarrow{\tau} \langle (\sigma_d(a)\sigma_d(w_2(b))X \parallel \sigma_d(c)\sigma_d(w_1(d))Y), \{(1,1),(2,1)\}, \{(1,0),(2,0)\}\rangle$
$\xrightarrow{a|c} \langle (\sigma_d(w_2(b))X \parallel \sigma_d(w_1(d))Y), \{(1,0),(2,0)\}, \{\}\rangle$
$\xrightarrow{\tau} \langle (\sigma_d(b)X \parallel \sigma_d(w_1(d))Y), \{(1,0),(2,0)\}, \{(2,1)\}\rangle$
$\xrightarrow{\tau} \langle (\sigma_d(b)X \parallel \sigma_d(d)Y), \{(1,0),(2,0)\}, \{(1,1),(2,1)\}\rangle$
$\xrightarrow{b|d} \langle (X \parallel Y), \{(1,1),(2,1)\}, \{\}\rangle$
$\ldots$

と展開され，n 番目の実行を a_n, b_n, c_n, d_n と区別すると，

$\sigma_d(a_0 \mid c_0) \cdot \sigma_d(b_0 \mid d_0) \cdot \sigma_d(a_1 \mid c_1) \cdots$

と続く動作が得られ，$\psi^1_{d_k \to a_{k+1}}, \psi^1_{b_k \to c_{k+1}}$ 両方が成立することがわかる．

これは，シナリオの形式化における横チェーン挿入後の LTS(図 4.1) において，ACP との違い (3.8 節参照) を考慮して (M0,0,P1,Q1),(M3,0,P0,Q0) 間に $a|c$ のパス，(M3,0,P0,Q0),(M0,0,P1,Q1) 間に $b \mid d$ のパスを追加した LTS 上で以下の，部分パスを選択し，非決定性を削除したことに相当する．

$$(M0,0,P1,Q1) \xrightarrow{a|c} (M3,0,P0,Q0) \xrightarrow{b|d} (M0,0,P1,Q1)\ldots$$

5.4 縦チェーン用プロトコルの挿入

縦チェーンは，2 つのプロセスが連動して動作し，かつデータが上書きされないという性質 $\psi^1_{b_k \to c_k}, \psi^1_{d_k \to b_{k+1}}$ 両方が成立することが必要であった．LTS の場合には，$\nabla^1_{b_k \to c_k}, \nabla^1_{d_k \to b_{k+1}}$ を適用することにより，これらを充足する LTS が得られることがわかっていた．ここでは，離散実行するプロセスでの実現について InsertBrW を用いる実現を検討する．

再帰方程式 (3.1) において，$\nabla^1_{b_k \to c_k}, \nabla^1_{d_k \to b_{k+1}}$ を適用することは，

InsertBrW(E,b,c,1,0) かつ InsertBrW(E,d,b,2,1) を適用することに対応する．

$$X = \sigma_d(a) \cdot \sigma_d(br_2(w_1(b)))X$$
$$Y = \sigma_d(br_1(c)) \cdot \sigma_d(w_2(d))Y$$

$\langle X \parallel Y, \{(1,0),(2,1)\}\{\}\rangle$ は，

$\langle X \parallel Y, \{(1,0),(2,1)\}\rangle$
$\xrightarrow{a_0} \langle \sigma_d(br_2(w_1(b_0)))X \parallel \sigma_d(br_1(c_0)) \cdot \sigma_d(w_2(d_0))Y), \{(1,0),(2,1)\}, \{\}\rangle$
$\xrightarrow{\tau} \langle \sigma_d(w_1(b_0))X \parallel \sigma_d(br_1(c_0)) \cdot \sigma_d(w_2(d_0))Y), \{(1,0),(2,1)\}, \{(2,0)\}\rangle$
$\xrightarrow{\tau} \langle \sigma_d(b_0)X \parallel \sigma_d(br_1(c_0)) \cdot \sigma_d(w_2(d_0))Y), \{(1,0),(2,1)\}, \{(1,1),(2,0)\}\rangle$
$\xrightarrow{b_0} \langle \sigma_d(a_1) \cdot \sigma_d(br_2(w_1(b_1)))X \parallel \sigma_d(br_1(c_0)) \cdot \sigma_d(w_2(d_0))Y), \{(1,1),(2,0)\}, \{\}\rangle$
$\xrightarrow{\tau} \langle \sigma_d(a_1) \cdot \sigma_d(br_2(w_1(b_1)))X \parallel \sigma_d(c_0) \cdot \sigma_d(w_2(d_0))Y), \{(1,1),(2,0)\}, \{(1,0)\}\rangle$
$\xrightarrow{a_1|c_0} \langle \sigma_d(br_2(w_1(b_1)))X \parallel \sigma_d(w_2(d_0))Y), \{(1,0),(2,0)\}, \{\}\rangle$
$\xrightarrow{\tau} \langle \sigma_d(br_2(w_1(b_1)))X \parallel \sigma_d(d_0)Y), \{(1,0),(2,0)\}, \{(2,1)\}\rangle$
$\xrightarrow{d_0} \langle \sigma_d(br_2(w_1(b_1)))X \parallel \sigma_d(br_1(c_1)) \cdot \sigma_d(w_2(d_1))Y), \{(1,0),(2,1)\}, \{\}\rangle$
$\xrightarrow{\tau} \langle \sigma_d(w_1(b_1))X \parallel \sigma_d(br_1(c_1)) \cdot \sigma_d(w_2(d_1))Y), \{(1,0),(2,1)\}, \{(2,0)\}\rangle$
$\cdots$

と展開され，n 番目の実行を a_n, b_n, c_n, d_n と区別すると，

$\sigma_d(a_0) \cdot (\sigma_d(b_0) \cdot \sigma_d(a_1 \mid c_0) \cdot \sigma_d(d_0)) \cdots (\sigma_d(b_i) \cdot \sigma_d(a_{i+1} \mid c_i) \cdot \sigma_d(d_i)) \cdots$

と続く動作が得られ，$\psi^1_{b_k \to c_k}, \psi^1_{d_k \to b_{k+1}}$ 両方が成立することがわかる．

これは，シナリオの形式化における縦チェーン挿入後の LTS(図 4.3) において，ACP との違い (3.8 節参照) を考慮して `(M0,1,P1,Q0),(M3,1,P0,Q0)` 間に $a \mid c$ のパスを追加した上で以下の $a \mid c$ のパスを追加した上で以下の，部分パスが選択されることに相当する．

$(M0,0,P0,Q1) \xrightarrow{a} (M1,1,P0,Q1) \xrightarrow{b} (M0,1,P1,Q0) \xrightarrow{a|c} (M3,1,P0,Q0) \xrightarrow{d} (M1,1,P0,Q1) \ldots$

5.5 シナリオ合成の手順

これまで，シナリオで定義された順序制約 ψ を充足するようにコマンドの LTS を合成して離散時間の動作仕様 (ACP_{drt} のプロセス) として実装する手段として，∇ 演算結果の LTS をリソースや性能を加味して 1 つのプロセスとして決定化する方法と，複数プロセスを保持したままプロセス間の通信チャネルとチャネルに対する読み書きプロトコルの挿入 (InsertBrW) により実現する方法の 2 種類があることがわかった．

この節ではシナリオ合成において，これらの 2 つの手段を組み合わせて，シナリオ・コマンドから適切な合成結果を得る手順を示す．

シナリオ合成 ($Synteisis(SC, CMD, MC)$) は以下の手順で実現される．

1. シナリオ図から制約 ψ の集合 SC(ψ システムと呼ぶ) を得る
2. SC(ψ システム) を，∇ 演算で結合されたコマンド $cmd \in CMD$ からなる式 (∇ システムと呼ぶ) を得る
3. ∇ 演算に従い，1)2 つの LTS を 1 つに合成するあるいは，2) プロトコル挿入 (Insert-BrW) を順次行う．
4. ∇ 演算の適用が全て行われたとき，結果として得られたコマンドの LTS の集合がシナリオ合成の結果となる．各々の LTS が，HW のモジュール (VHDL なら entity) に対応する．

ψ システムから ∇ システムを得るアルゴリズムを以下では述べる．

シナリオ $SC = \langle PS, RH, RV \rangle$ に対して，個々のプロセスに適宜 ∇ 演算を適用して合成結果である ∇ システムを得る以下の手順に従う．ここでは 3.7.1 節における ∇ 演算の交換可能性の説明で用いた書換え規則をつかって，任意のシナリオが ∇ システムに書き換えられるることを示す．

R を RH あるいは RV とし，i,j を制約を区別する番号であるとすると，書き換え規則は，以下のように書ける．

```
-[書換規則]--------
PS={X,Y,Z}
R={(X,Y,i),(Y,Z,j)}
↓
PS={▽ i(X|Y),Z}
R={(▽ i(X|Y),Z,j)}
----------------
```

例えば，図 3.6 の例では,

シナリオは $SC_{abcd} = \langle PS, RH, RV \rangle$ と表される．ここで $PS = \{A, B, C, D\}, RH = \{(A, B), (C, D)\}, RV = \{< B, C >\}$ である．

ステップ１ :書き換え規則を用いた ∇ 演算の適用

シナリオ SC_{abcd} に順次書き換え規則を適用すると例えば以下の書き換えシーケンスが得られる.

```
---------------------------
PS={A,B,C,D}
RH={(A,B,1),(C,D,3)}
RV={<B,C,2>}
  ↓
PS={▽ 1(A|B),C,D}
RH={(C,D,3)}
RV={<▽ 1(A|B),C,2>}
  ↓
PS={▽ 2(▽ 1(A|B)|C),D}
RH={(▽ 2(▽ 1(A|B)|C),D,3)}
RV={}
  ↓
PS={▽ 3(▽ 2(▽ 1(A|B)|C)|D)}
RH={}
RV={}
---------------------------
```

書換は停止し，以下の ∇ システムを得る.

$$\nabla_{sc1}(A, B, C, D) = \nabla_3(\nabla_2(\nabla_1(A \mid B) \mid C) \mid D)$$

書換は一意ではなく，R から取り出す関係の順番の組合せにより，同様の計算を繰り返して，結果として以下のヴァリエーションを得る.

- $\nabla_3(\nabla_2(\nabla_1(A \mid B) \mid C) \mid D)$
- $\nabla_1(A \mid \nabla_3(\nabla_2(B \mid C) \mid D))$

- $\nabla_1(A \mid \nabla_2(B \mid \nabla_3(C \mid D)))$
- $\nabla_3(\nabla_1(A \mid \nabla_2(B \mid C))) \mid D)$

これらは，交換可能であるという意味で等しい（$=$）．

ステップ 2 :合成

一旦 ∇ システムが得られたら，これに従い (ステップ 2a):2 つの LTS を 1 つに合成するあるいは，(ステップ 2b):プロトコル挿入 (InsertBrW) を順次行う．2 つの LTS を 1 つにまとめることを以下では畳み込むと表現する．

ステップ 2a :畳み込みによる合成

例えば図 5.1 では，先に A, B を 1 つに畳み込み，次に C を続けて畳み込む例 (Scenario1 → Scenario1B → Scenario1C) と，先に B, C を 1 つに畳み込み，続けて A を畳み込む例 (Scenario1 → Scenario1A → Scenario1C) が合流することを示している．前者は $\nabla_3(\nabla_2(\nabla_1(A \mid B) \mid C) \mid D)$ に，後者は $\nabla_1(A \mid \nabla_3(\nabla_2(B \mid C) \mid D))$ に相当する．

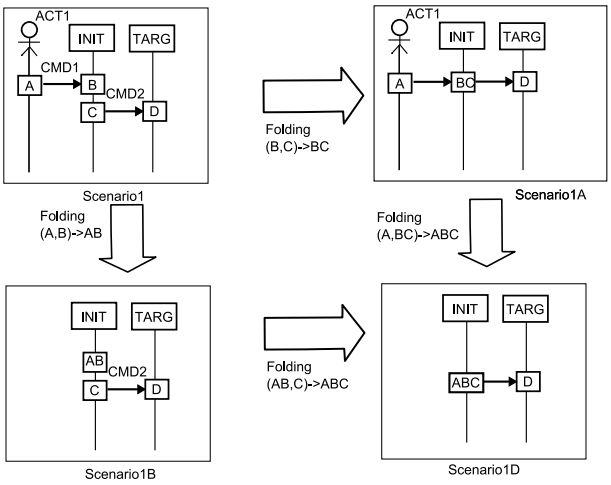


図 5.1: シナリオ畳み込み

どの畳み込みを選択するか，どこまで畳み込みを行い，どこから InsertBrW によるプロトコル挿入を行うかは，実装したときの性能や面積等の非機能要件の評価結果を用いる．

以上のように，シナリオ定義，コマンド定義から合成手順に従って実装である LTS のシステムを得る手順が明らかになった．

第 6章

全システムの統合

シナリオマージとは，複数のシナリオ合成の結果を，1つのシステムとしてまとめる操作である．最初は最大並列に動作する状態遷移機械群としてマージされ，性能やリソース制約を考慮して，適切なコマンド活性体（状態遷移機械）を選択して逐次合併することで，システムを最適化する．ここでは，プロセス代数にてマージ操作とマージ操作によっても元々のシナリオ動作が保持されることを説明する.

6.1　シナリオマージ

シナリオの合併 (マージ) とは，複数のシナリオを 1 つのシナリオにマージする操作である．

シナリオはコマンド起因の LTS が互いに順序制約に従って動作するための，順序制約のシステムであったから，シナリオのマージは，制約同士の合成操作になる．シナリオのマージでは，シナリオ間に跨る順序制約を新たに考慮する．代表的なシナリオ間の順序制約は排他関係である．排他関係はシナリオ同士がリソースを共有する部分を含む場合に，新たに挿入される．

マージされたシナリオが得られた後に，前章までで説明したシナリオ合成にて，コマンドの LTS に適切な，プロトコルを挿入したり，コマンドの合併を行って，最適化な実装解を得る．

6.1.1 排他関係

$E = \{X = a \cdot bX, Y = c \cdot dY\}$ における a, b と c, d 間の排他関係 $\psi_{\{a,b,c,d\}}$ は以下の初期プロセスを P0 とする以下の LTS として表現できる．

```
LTS20:
 Proc = {P0,P1,P2}
 Act = {a,b,c,d}
 -a->{(P0,P1),(P1,NG),(P2,NG)}
 -b->{(P1,P0),(P0,NG),(P2,NG)}
 -c->{(P0,P2),(P1,NG),(P2,NG)}
 -d->{(P2,P0),(P0,NG),(P1,NG)}
```

順序制約と同様に，$\psi_{\{a,b,c,d\}}$ を強制する $\nabla_{\psi_{\{a,b,c,d\}}}$ 演算を E に施すと以下の初期プロセス (M0,P0) の LTS を得る．

```
LTS21:
 Proc = {(M0,P0),(M1,P1),(M2,P2)}
 Act = {a,b,c,d}
 -a->{<(M0,P0),(M1,P1)>}
 -b->{<(M1,P1),(M0,P0)>}
 -c->{<(M0,P0),(M2,P2)>}
 -d->{<(M2,P2),(M0,P0)>}
```

6.1.2 アービターの挿入

eMSC では排他制約に対して，特別なアービター LTS を導入し，アービターとコマンドの LTS 間に InsertBrW によりプロトコルを挿入して，排他動作を実装していた．

$$
\begin{aligned}
X = \quad & \sigma_d(w_1(\tau)) \cdot \sigma_d(br_2(a)) \cdot \sigma_d(w_3(b)) \cdot X \\
Y = \quad & \sigma_d(w_4(\tau)) \cdot \sigma_d(br_5(c)) \cdot \sigma_d(w_6(d)) \cdot Y \\
ABT = \quad & \sigma_d(br_1(w_2(\tau)) \cdot \sigma_d(br_3(\tau)) \cdot ABT + \\
& \sigma_d(br_3(w_4(\tau)) \cdot \sigma_d(br_5(\tau)) \cdot ABT
\end{aligned}
$$

ここでτは無名アクションであり ABT はアービターのプロセスである．ABT 自体の合成は，本論文の範囲外であるが，プロパティからコントローラ本体を合成する技術として例えば Ramadge と Wonham の研究 [24] がある．

6.2 シナリオマージによるシステムの統合例

図 6.1 はシナリオマージの例を図示したものである．マージ後にC, D, I, Jが同一のバスを共有して通信するというリソース制約がある場合には，D, IおよびC, J間に排他制約が挿入される．

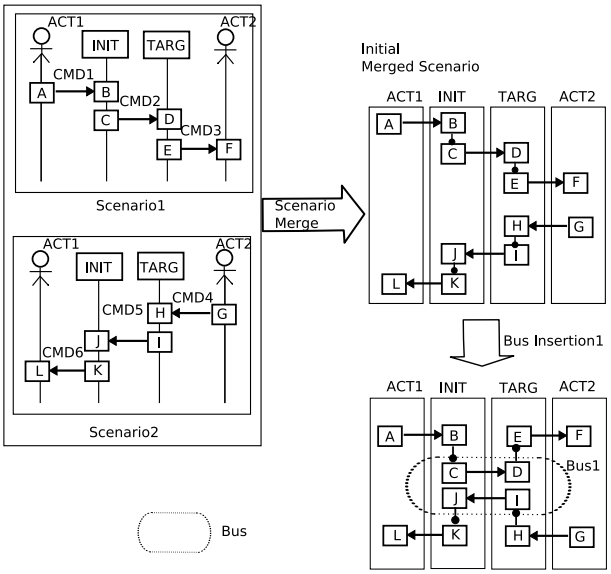


図 6.1: シナリオマージの例

排他制約が守られれば，元々のシナリオは遅延は生じるが実行される（図 6.1 の例ではシナリオの実行はクロスしている）．

このようにシナリオマージにおいても，制約ψとこれを強制する∇演算を導入し，これを実装するInsertBrWプロトコル挿入を割り当てるという枠組みに沿って説明できた．

第 7章

関連研究

7.1 プロパティの形式化と検証手法

本論文ではプロセス代数で記述された検証対象の LTS に対して、検証したいプロパティを LTS として表現し，Game に基づく解法により充足性を検証した．プロパティの検証方法の代表的な手法であるモデル検査と Game に基づく手法の関係についてここでは述べる．

モデル検査 [10] ではモデルを記述する手段を Promela 等の仕様記述言語として提供し，LTL(Linear Temporal Logic) や CTL(Computational Tree Logic) に代表される時相論理によって検証したいプロパティを記述する．検証手順としては，automaton 理論に基づき，モデルを automata に内部で変換し，また一方プロパティも automata に変換し，モデルの automata が受理する言語が，プロパティの automata が受理する言語に包含されるか（モデルの振る舞いが，プロパティで許容されている範囲に収まっているかどうか）を判定する．具体的な計算手順は例えば記号モデル検査では、モデルの automaton とプロパティの否定型の automaton の直積に対して非空性（受理言語があること）を計算するという手順をとってプロパティの充足性を判定している [10].

一方，計算機科学における様々な検証・合成問題は Game 理論により形式化されることがわかっている [30]．例えば状態遷移系の検証技術の基盤となる，LTS（ラベル付き遷移システム）間 (ここでは LTS1 と LTS2 とする) の双模倣性 (bisimimular) の判定を，LTS1 と LTS2 の間の Game（この場合はまねっこ game）と見なすことで形式化することができる．また，例えば reactive システム（環境からの刺激に対してシステムが応答する）においては，性質 ψ に対する検証問題は，性質 ψ が成立する範囲に動作が留まることをシステ

ム側の目的とし、範囲から外すことを環境側の目的とするシステムと環境の間の game と見なすことができる．システム側に必勝戦略が存在するならば，必勝戦略に従ってシステムが動く限りにおいて ψ が正しいことが検証される．また，性質 ψ に対してこれを充足するシステムを合成する問題を，合成問題 [9, 16, 23] と呼ぶ．合成問題においてはシステムの外部を環境として，環境とシステムの間の相互作用を game として形式化し，システム側が環境側に勝てる戦略を持ちうるかを計算し，システム側に必勝戦略がある場合に，性質が合成可能であるとする．そして勝利戦略から性質を充足するシステムの動作仕様を合成する [24, 17, 22].

一方，インターフェイス理論 [11] では，互いに通信する LTS1，LTS2 が与えられたとき，これらが矛盾（通信の空振りが起きない）無く連携して動作するための外部 (LTS1 および LTS2 の外) 入力の制約を計算する．ここでは LTS1 $\otimes$ LTS2 が矛盾に至らない安全 (safe) な領域へ滞留するための条件を計算するために”Safety-game” と呼ばれる Game 理論の手法を適用している．インターフェイス理論は，２つのインターフェイス (LTS1,LTS2) が連携するための前提条件を計算するという意味でインターフェイスの摺り合わせ検証を行っていることに相当し，従来の組み合わせ検証による状態爆発を回避する技術として着目されている [31].

また本論文における性質の形式化 (3.2.4 章) のように、性質 ψ をラベル付遷移システム (LTS) として形式化すれば，インターフェイス理論は性質 ψ を充足するための環境側の動作制約（仮定=Assume) を計算していることになる [14]．また到達可能性（いつかは ψ が成り立つ）を検査するためには Reachablity-Game の考え方が適用される．

このように Game 理論による形式化は，同じ性質 ψ に関する検証と合成を一つのフレームワークで取り扱えるので問題とその解法の見通しを良くする．また，以下の実用的なメリットを持っている．まず勝利戦略の計算は，適切な関数の最大不動点・最小不動点を求める問題に帰着できることが知られており，これにより有限の時間で記号的に解（正しいかどうかの判定，双模倣であるかどうかの判定，合成可能であるかどうかの判定）を求めることができる．さらに不動点計算の記号的な計算においては，モデル検査等で使われる効率化の手段（BDD 等）を用いることができる．例えば Stevens と Staring[25] によると、プロセス代数系の検証環境 CWB(The Edinburgh Concurrency Workbench) では v7.0 より従来のタブロー法に基づくモデル検査から game ベースのモデル検査に切り替えることにより、スケーラビリティが格段に向上したことを報告している．

また Staring[27, 25] によると，μ 計算 (様相 μ 計算) は時相を含む論理に基づく検証や合成の基本的な形式化理論であり、モデル検査などで用いられている CTL(計算木論理) を包含する体系である。様相 μ 計算におけるモデル検査は、parity-game に線形時間でエンコード可能 (Emerson 他 [12]) であり，モデル検査は、モデル (M) とプロパティ (P) の間の充足性 Game(P に対して M を充足することを示そうとする PlayerI と、逆の Player2 の間の Game) として形式化できる。M, P がラベル付遷移システム (LTS) としてモデル化できる場合は、直積 ($M \otimes P$) を Game の Arena として計算して M と P の間の Game として形式化できることが知られている [26].

本論文における検証という目的においては従来のモデル検査手法以外に Game に基づく手法を用いる理由はない．本論文において Game に基づく手法を採用した理由は，まずプロパティである順序制約を形式化する方法が時相論理による記述よりも LTS として形式化するほうが自然であった点，さらには，将来的な発展方向としてインタフェース理論との親和性（組合せの検証から、摺り合わせの検証へ）および、合成問題との親和性があるからである．

また Game 問題に帰着できれば、検証を専用のソルバを用いて行うこともできる．PGSolver[21] はこのような観点に基づき、種々の parity game の解法アルゴリズムを集めた汎用の Game エンジンである．PGSolver を用いた game の解法の具体例は付録 A.1.6 にて説明する．

7.2 MSC からの合成手法

7.2.1 MTS(Model Transition System) のマージによる方法

Uchitel らの MTS[29] は，プロセスの部分仕様を MTS(Modal Transition System) を用いて記述し，複数の MTS 部分仕様を 1 つの MTS としてマージすることを行う．MTS は，required,posible,maybe という 3 種のトランジションを持つことにより必須の仕様 (required) と可能性のある仕様 (possible,maybe) を区別することを特徴としている．マージ対象の複数の MTS に成立するトランジションを比較して可能遷移 (maybe) を必須遷移 (required) に変更するあるいは不要ならば取り除く修正 (refinement) を行い MTS の共通解を得ることにより MTS のマージを定義している．マージ処理は状態遷移機械に対して実行順序の部分的な仕様が複数与えられたときに，仕様をマージして全てを充足する単一の状態遷移

機械の実体を計算することに相当する．

7.2.2 MSC(Message Sequence Chart) から不足の仕様を導出する方法

Alure ら [3] は，メッセージシーケンス図 (MSC) において，MSC の集合から imply される MSC を導出する方法 (weak realizability) により，明示的に表現されない意図しない動きや dead lock を起こす可能性のある MSC 例を生成提示する．この imply する MSC が，dead lock しない場合，順次もとの MSC の集合に追加して，最後に何も imply されなくなる時が，MSC が safe realizable の場合で，このとき，この MSC の集合から，並列動作する automaton である状態遷移機械を求める．Alur らの手法は，目的もスコープも Uchitel らの手法と同じであるが，部分仕様である MSC を足しこむことにより，足らない仕様 (imply される MSC) を生成できるところが特徴である．

7.2.3 提案手法との比較

Uchitel や Alure の提唱する手法では，MSC を制約として捕らえて，制約を実現するシステムを合成する，いわゆる「合成問題」の入力として MSC を捕らえている．

我々のアプローチでは，対象システムをコマンドとシナリオの 2 階層の MSC としてモデル化し，コマンドは直接離散プロセスとして変換し，またシナリオで定義される順序制約に関しては，これを充足する最大の部分モデルを得るような合成コマンド (∇ 演算のこと) を個別に導入して，これらを組み合わせて全体のシナリオ動作を担保した．

第 8章

まとめと展望

本研究では，eMSC システム (1 章) における仕様記述と合成手順をプロセス代数 (2 章) によって形式化し，正しさを保証する枠組みについて検討した．

3 章では eMSC システムにおける下位階層プロトコルであるコマンド記述を，離散時間拡張されたプロセス代数 ACP_{drt} を用いて形式化することができた．プロセス代数における並行結合 (||) を拡張した COMM 演算を導入し，シナリオ動作を LTS 上の順序制約 $\psi^n_{x_i \to y_j}$ として形式化することができた．また順序制約 $\psi^n_{x_i \to y_j}$ の充足性を定義し，LTS に動作制限を与える ∇ 演算を導入し順序制約 ψ との間の関係 (充足性) を明らかにした．

4 章ではシナリオ合成に対する性質の充足性の判定および，正しいプロセスの合併の計算が保障されることが判った．5 章ではシナリオ合成結果を実装と結びつける 2 つの方法について明らかになり，6 章ではシナリオマージを考察し排他動作の例において 5 章までの枠組みで取り扱えることを示した．

本研究によりシナリオ形式で記述された仕様に従ってプロセス間連携のための部分プロトコルを逐次挿入してシナリオを合成して，得られたシナリオを複数マージすることによりシステム全体の仕様を構成的に構築する手法が確立された．

今後の展望としては，シナリオマージにおける合併操作を設計探索の最適化問題として形式化することや，マージにおける連携制約自体をシステム要求仕様から導出することが考えられる．

謝辞

本論文を執筆するにあたり，４年強の長きに亘ってご指導をいただいた日比野教授に感謝をいたします．また本論文の予備審査時に，本論文の草稿に対して，精密な検証をしていただき，改善の提案までしていただいた産業技術総合研究所 (AIST) の磯部博士に感謝します．

また，社会人ドクターコースへのチャレンジを快く許可していただいた，(株）インターデザインテクノロジーの山本社長（当時），本研究のきっかけとなった eMSC システムの実現に情熱をもって取り組まれた鈴木氏を代表とするエンジニアの方々，またドクターコースに最初に誘っていただいた落水教授に感謝します．また落水教授の言葉を信じて，一度も大学本部に物理的に登校することなしに博士後期課程を終えることができましたことは，田町サテライトオフィスのスタッフの協力なしには達成できなかったと思っております，ここに感謝します．

最後に，土日ろくに家族サービスできない父親を，文句も言わず癒してくれた萌子，知歩，日に影に支援をしてくれた智子に最大の感謝をするとともに本論文を捧げます．

第 A 章

付録

A.1 Gameの理論の概要

Gameとは，2人のPlayerが，互いに自分の手番を打ち，どちらかが勝利するまでこれを続ける問題であるとする．

Gameの理論では，片方が必ず勝つ必勝戦略を求めることが，目的とされる．

計算機科学における様々な問題はGameの理論により形式化されることがわかっている．

例えば状態遷移系の検証技術の基盤となる，LTS（ラベル付き遷移システム）間 (ここではLTS1とLTS2とする) の双模倣性 (bisimimular) の判定を，LTS1とLTS2の間のGame（この場合はまねっこゲーム）と見なすことで説明することができる．

また，検証問題も，例えばリアクティブシステム（環境からの刺激に対してシステムが応答する）においては，検証問題は，プロパティψが正しいと定義される範囲に留まることを目的関数とする，環境とシステムの間のゲームと見なすことができる．システムに必勝戦略が存在するならば，ψが正しいことが検証される．また到達可能性（いつかはψが成り立つ）を検査するためにはReachablity-Gameの考え方が適用される．

また，合成問題も，リアクティブシステムを例にとると，環境とシステムの間のゲームにおいて，環境の動作のプロパティ（動作制約）が与えられているときに，システムが環境に勝てる戦略を持ちうるか？について計算することにより，システムの動作仕様を合成する．

Gameによる形式化は，問題とその解法の見通しを良くする以外に，以下の実用的なメリットを持っている．まず勝利戦略の計算は，適切な関数の最大不動点・最小不動点を求

める問題に帰着できることが知られており，これにより，有限の時間で記号的に解（正しいかどうかの判定，双模倣であるかどうかの判定，合成可能であるかどうかの判定）を求めることができる．さらに不動点計算の記号的な計算においては，モデル検査等で使われる効率化の手段（BDD 等）を用いることができる．

A.1.1 Game の構造と勝利集合

まず最初に Game 構造 (Game Structure) を以下のように定義する．

定義 A.1.1 (Game 構造 (Game Structure))

$$G = \langle S, M, \Gamma_1, \Gamma_2, \delta \rangle$$

S は状態の集合，M は Move の集合，Move 割り当て $(\Gamma_i : S \times 2^M/\emptyset, i = 1, 2)$ は状態 $s \in S$ に対して，Player i の可能な Move の集合 $\Gamma_i(s) \subseteq M$ を割り当てる関数，遷移 $\delta : S \times M \times M \to S$ は，状態 s に対して Player1,Player2 の Move の後，到達する次状態の集合を得る関数 $\delta(s, a, b) \in S$ により定義される．

また Game 構造 G は，以下の性質を充たすときに Turn-based Game と呼ぶ．

定義 A.1.2 (Turn-based Game)
Game 構造 $G = \langle S, M, \Gamma_1, \Gamma_2, \delta \rangle$ は，S 上の写像 $\gamma : S \mapsto \{1, 2\}$ が存在し，$s \in S$ に対する Player の順番を規定し，全ての $s \in S$ に対して，$\gamma(s) = i$ ならば，$|\Gamma_{-i}| = 0$ すなわち，一方 (i) の順番である状態 s においては他方 $(-i)$ の可能な Move が必ずない．

Tunr-based Game においては状態 S は，Player1 の手番である S_1，Player2 の手番である S_2 のどちらかに分類される，すなわち $S = S_1 \cup S_2$ である．

Game 構造における戦略は，状態列（打つ手の履歴）$S+$ に対して，次の Move $m \in M$ を決める関数として定義される．

定義 A.1.3 (戦略 (strategy))
Player i (i=1,2) の戦略は $\pi_i : S+ \to M$ で表される写像である．

また π のクラス（π の集合ということ）を Π とする．

定義 A.1.4 (Outcome)
また状態 $s \in S$ と戦略 π_1, π_2 があたえられたとき，s から開始して戦略を適用して得られる，状態の列を $s_0, s_1, s_2, .. = outcoume(s, \pi_1, \pi_2) \in S^\omega$ と呼ぶ．ここで S^ω は状態 S の無限列 (トレースと呼ぶ) の集合全体である．

- $s_0 = s$
- $\forall k, s_{k+1} = \pi_i(s_0, ..., s_k)...s_k \in S_i$

Outcome の定義をみてもわかるように，必ずしも Player1,Player2 は交互に手番がくるわけではない．

Player 1 と Player2 が $R \subseteq S$ をゴール (goal) とし，s_k をトレース $s \in S^\omega$ の k 番目の要素だとすると，R に対する勝利条件 (Winning Condition) は以下のような S^ω の部分集合 (状態の無限列の集合) として定義する．

定義 A.1.5 (勝利条件:Winning Condition)

Reachablity Game

$$\Diamond R = \{\sigma \in S^\omega | \exists k.\sigma_k \in R\}$$

トレースの k 番目の要素 s_k が少なくとも 1 つは R に含まれている

Safety Game

$$\bigcirc R = \{\sigma \in S^\omega | \forall k.\sigma_k \in R\}$$

トレースの全ての要素が R に含まれている．

定義 A.1.6 (勝利集合:Winning states)
勝利条件に対して，勝利集合を定義する．Player1 が勝利条件を満たすような戦略を（Player2 の戦略にかかわらず）常に持つことができる，状態の集合を勝利集合と定義する．$\langle 1 \rangle \Box R, \langle 1 \rangle \Diamond R$ と表す．

$$\langle 1\rangle \Diamond R = \{s \in S | \exists \pi_1 \in \Pi_1. \forall \pi_2 \in \Pi_2. outcomes(s, \pi_1, \pi_2) \subseteq \Diamond R\}$$
$$\langle 1\rangle \Box R = \{s \in S | \exists \pi_1 \in \Pi_1. \forall \pi_2 \in \Pi_2. outcomes(s, \pi_1, \pi_2) \subseteq \Box R\}$$

すなわち，勝利集合は全ての Player2 の戦略に対して，勝利条件に滞留できるような戦略を Player1 が一つでも持ちうるような状態の集合である．

勝利集合 Win_1 が求まれば，Player1 の Game に対する勝ち負けを以下のように判定できる．

任意の状態 s からの Player1 の Move 割り当てが $\Gamma_1(s) \cap Win_1 \neq \phi$ である場合には，Player1 は遷移先状態が Win_1 に含まれるように自分の次のアクションを選択すれば勝利できる．常に $\Gamma_1(s) \subset Win_1$ である場合には，Player1 は何を選択しても勝利するので常勝である．$\Gamma_1(s) \cap Win_1 = \phi$ となるような状態 s が一つでも存在するならば，Player1 は負けである．

勝利集合の求め方，

まずゲーム構造を，$G = (S, M, \Gamma_1, \Gamma_2, \delta)$ とする

$X \subset S$ に対して $Pre(X)$ を以下のように定義する．

定義 A.1.7 (Pre:Conditional Predecessor(Controllable Predecessor))
$Pre(X)$ は conditional predecessor とよばれる集合で，Player1 が 1 回の turn で game の状態が X に含まれる様に，制御できるような前状態 s の集合を表す．

$$Pre_1(X) = \{s \in S | \exists a \in \Gamma_1(s). \forall b \in \Gamma_2(s). \delta(s, a, b) \subseteq X\}$$

これは，Turn based Game ならば，$E(s)$ を状態 s から可能な遷移先の状態の集合とすると，以下と同等である．

$$Pre_1(X) = \{s \in S_1 | \exists t \in E(s), t \in X\} \cup \{s \in S_2 | \forall t \in E(s), t \in X\}$$

A.1.2 Game の逐次解法

Reachablity ゲームの解法 (だんだん広がって境界に到達するイメージ) $\langle 1\rangle \Diamond R$:Reachablity ゲームにおける Player1 の勝利集合，を求めるには 以下の X_nX に関するイタレーションが収束するまで繰り返すことにより求まる．

$$
\begin{aligned}
X_0 = \quad & \phi \\
\ldots & \\
X_{k+1} = \quad & R \cup Pre_1(X_k), \forall k \geq 0 \\
\ldots & \\
\langle 1 \rangle \Diamond R = \quad & \lim_{k \to \infty} X_k
\end{aligned}
$$

すなわち，R に到達できる全ての前状態を順に追加していって（∪演算）それが収束するところが，勝利集合

Safety ゲームの解法 (だんだん狭まって境界に到達するイメージ)　一方 $\langle 1 \rangle \Box R$:Safety ゲームにおける Player1 の勝利集合を求めるには，　以下の X_n に関するイタレーションを収束するまで繰り返す.

$$
\begin{aligned}
X_0 = \quad & S \\
\ldots & \\
X_{k+1} = \quad & R \cap Pre_1(X_k), \forall k \geq 0 \\
\ldots & \\
\langle 1 \rangle \Box R = \quad & \lim_{k \to \infty} X_k
\end{aligned}
$$

すなわち，R に留まることができる全ての前状態を順に削除していって（∩演算）それが収束するところが，勝利集合

A.1.3　不動点による形式化

勝利集合は，最小不動点 (μ)，最大不動点 (ν) の計算として形式化される.

$$\langle 1 \rangle \Diamond R = \mu X.(R \cup Pre_1(X))$$

$$\langle 1 \rangle \Box R = \nu X.(R \cap Pre_1(X))$$

また，不動点演算の双対性より，Safety-Game の Player1 のゴール (R) に対する勝利集合の補集合は，Player2 の Reachablity-Game におけるゴールの否定 ($\neg$ R) に対する勝利集合と同じことがわかる（Player1 の勝利は Player2 の負け）

$$\neg\nu X.(R \cup Pre_1(X)) = \mu X.(\neg R \cap Pre_2(X))$$

計算の複雑さ

最大不動点計算は $Pre_I(X)$ を適用するに従って状態の数が単調減少するので，最大 $|S|$ ステップで収束する．しかし，Pre_i を求める計算は，観測可能な変数の数 n に対して EXPTIME(すなわち) であることが知られており，BDD による効率的な計算が行われる．

”A Lattice Theory for Solving Games of Imperfect Information“

A.1.4 Game による検証 (μ 計算の場合)

μ 計算 (様相 μ 計算) は時相を含む論理に基づく検証や合成の基本的な形式化理論であり、モデル検査などでもちられている CTL(計算木論理) を包含する体系である。モデル検査は、モデル (M) とプロパティ (P) の間の充足性 Game(P に対して M を充足することを示そうとする PlayerI と、逆の Player2 の間の Game) として形式化できる。M, P がラベル付遷移システム (LTS) としてモデル化できる場合は、直積 ($M \otimes P$) を Game の Arena として計算して M と P の間の Game として形式化できることが知られている。

ここではプロパティ (P) が様相論理で記述される場合のモデル検査、特に μ 計算におけるモデル検査の Game の理論による解法の概要を Staring[27, 25] による形式化に沿って示す。

最初に μ 計算の文法と意味を定義する。

μ 計算における式とは、$Prop$ を命題定数、Act をアクション、Var を変数の集合であるとするとき以下の文法にて定義される、

$$F := tt \mid ff \mid Prop \mid \neg Prop \mid Var \mid F \vee F \mid F \wedge F \mid \langle Act \rangle F \mid [Act]\, F \mid \mu Var.F \mid \nu Var.F$$

式は遷移システムにより解釈される。ここで遷移システムとは $M = \langle S, \{R_a\}_{a \in Act}, \sigma \rangle$ により構成され、S は状態の集合、$R_a = S \times S$ は遷移関係 $\sigma : Prop \to \mathcal{P}(S)$ は命題定数に対してこれが成立する状態の集合を割り当てる関数であるとする (ここで $\mathcal{P}(S)$ は S の集合の集合=べき集合 2^S を意味する)。

値割り当て $V : Var \to \mathcal{P}(S)$ は変数に対して状態の集合を割り当てる関数である。

遷移システム M と変数への値割り当て $V : Var \to \mathcal{P}(S)$ に対して式 ψ が真となる状態の集合を $\|\psi\|_V^M$ を以下のように定義する

$$\begin{aligned}
\|tt\|_V^M = S \quad , \quad \|ff\|_V^M &= \{\} \\
\|p\|_V^M = \sigma(p) \quad , \quad \|\neg p\|_V^M &= S - \sigma(p) \\
\|X\|_V^M &= V(X) \\
\|\langle a\rangle\alpha\|_V^M &= \{s : \exists s'.R_a(s,s') \wedge s' \in \|\alpha\|_V^M\} \\
\|\,[a]\,\alpha\|_V^M &= \{s : \forall s'.R_a(s,s') \wedge s' \in \|\alpha\|_V^M\} \\
\|\mu X.\alpha(X)\|_V^M &= \bigcap\{S' \subset S : \|\alpha\|_{V[S'/X]}^M \subset S'\} \\
\|\nu X.\alpha(X)\|_V^M &= \bigcup\{S' \subset S : S' \subset \|\alpha\|_{V[S'/X]}^M\}
\end{aligned}$$

ここで $V\,[S/X]$ とは値割付 V において $q \neq X$ の場合は $V\,[S/X]\,(q) = V(q)$ $q = X$ の場合は $V\,[S/X]\,(X) = S$ であるものを示す。

そして $s \in \|\alpha\|^M$(s においては M を鑑みると α は真であるというほどの意味) の代わりに $M, s \models \alpha$ と記述することとする

μ 計算におけるモデル検査問題とは、与えられた α と有限遷移システム M および状態 s_0 において、$M, s_0 \models \alpha$ であるかどうかを決定する処理である。

また以降では演算 $[]\,, \langle\rangle$ に現れる Act の部分集合の簡易表現として、$-K$ により $Act - K$ (K の補集合) を表すものとする。特別なケースとして $-$ は $Act - \emptyset = Act$ を表す。

μ 計算によるプロパティ例

safety 悪い状態に陥らないことを示す。悪い状態の complement が Φ であるとすると

$$\nu Z.\Phi \wedge [-]\,Z$$

としてあらわせる。悪いアクションが発生しないという見方をするならば、

$$\nu Z.\,[K]\,\mathrm{ff} \wedge [-]\,Z$$

としてあらわせる。

liveness ある良い性質がいつかは成立することを示す。Φ を良い良い状態を表すとすると

$$\mu Z.\Phi \vee (\langle -\rangle \mathrm{tt} \wedge [-]\,Z)$$

またアクションに着目すると

$$\mu Z.\langle -\rangle \mathrm{tt} \wedge [-K]\, Z$$

組み合わせ safety と liveness を組み合わせたプロパティもある、たとえば a が発生する場合は、いつかは b が発生する

$$\nu Z.\, [a]\, (\mu Y.\langle -\rangle \mathrm{tt} \wedge [-b]\, Y) \wedge [-]\, Z$$

としてあらわせる

モデル検査ゲーム

μ 計算のモデル検査を Game で解く手法を紹介する。

V を値割付、E をプロセス、Φ を μ 計算の式であるとしたとき、$\mathcal{G}_v(E, \Phi)$ をモデル検査ゲーム[1]と定義する。

$\mathcal{G}_v(E, \Phi)$ では、PlayerI が、E が値付け V に対して Φ を充足しないことを証明しようとし、PlayerII は逆を行う Game を実行する。

$\mathcal{G}_v(E, \Phi)$ の play は有限あるいは無限の $(E_0, \Phi_0)...(E_n, \Phi_n)..$ 列であり、$\Phi_i \in Sub(\Phi), E_i \in \mathcal{P}(E_0)$ である。

いま仮に play 列が $(E_0, \Phi_0)...(E_j, \Phi_j)$ であるとして次の可能な手番 move は Φ_j の形式に依存して決定される。

Game のルール

- $\Phi_j = \Psi_1 \wedge \Psi_2$ ならば、PlayerI が Ψ_i を選択する。すなわち $E_{j+1} = E_j, \Phi_{j+1} = Psi_i$
- $\Phi_j = \Psi_1 \vee \Psi_2$ ならば、PlayerII が Ψ_i を選択する。すなわち $E_{j+1} = E_j, \Phi_{j+1} = \Psi_i$
- $\Phi_j = [K]\, \Psi$ ならば、PlayerI が $E_j \xrightarrow{a} E_{j+1}$ を選択し、$\Phi_{j+1} = \Psi$ とする
- $Phi_j = \langle K \rangle \Psi$ ならば、PlayerII が $E_j \xrightarrow{a} E_{j+1}$ を選択し、$\Phi_{j+1} = \Psi$ とする
- $\Phi_j = \sigma Z$ ならば $\Phi_{j+1} = Z, E_{j+1} = E_j$
- $\Phi_j = Z$ かつ Φ_0 において Z に関連する部分式が $\sigma Z.\Psi$ ならば $\Phi_{j+1} = \Psi, E_{j+1} = E_j$

[1] 元論文では property checking game

Player I の勝利条件

1. 有限の play が $(E_0, \Phi_0)...(E_n, \Phi_n)$ であり、$\Phi_n = \mathrm{ff}$ あるいは $\Phi_n = Z$(Z は Φ_0 の中で束縛されないかつ $E_n \notin V(Z)$)
2. 有限の play が $(E_0, \Phi_0)...(E_n, \Phi_n)$ であり、$\Phi_n = \langle K \rangle \Psi$ かつ $\{F : E \xrightarrow{a} F \wedge a \in K\} = \emptyset$
3. 無限の play が $(E_0, \Phi_0)...(E_n, \Phi_n)...$ であり、infinitely often に展開される変数 χ_i の中で最外郭のものが $\nu\chi.\Psi$ という形式をとる場合

Player II の勝利条件

1. 有限の play が $(E_0, \Phi_0)...(E_n, \Phi_n)$ であり、$\Phi_n = \mathrm{tt}$ あるいは $\Phi_n = Z$(Z は Φ_0 の中で束縛されないかつ $E_n \in V(Z)$)
2. 有限の play が $(E_0, \Phi_0)...(E_n, \Phi_n)$ であり、$\Phi_n = [K]\,\Psi$ かつ $\{F : E \xrightarrow{a} F \wedge a \in K\} = \emptyset$
3. 無限の play が $(E_0, \Phi_0)...(E_n, \Phi_n)...$ であり、infinitely often に展開される変数 χ_i の中で最外郭のものが $\mu\chi.\Psi$ という形式をとる場合

play が無限の場合は infinitely often に展開される「最外郭の不動点」により勝者が決まる。最小不動点の場合は PlayerI が、最大不動点の場合には PlayerII が勝利する。最外郭とは式の subsumption 関係（部分式の関係）により定義される。たとえば $\sigma\chi_1\sigma\chi_2..\sigma\chi_n\Phi(\chi_1, .., \chi_n)$ であるとすると infinitely often に展開される χ_a の中で他の chi_i を subsume するもっとも大きい chi_j が存在するはずで、これを最外郭と定義する。

例

$D \xrightarrow{a} D', D' \xrightarrow{a} D', D \xrightarrow{b} D''$ であり、Ψ が

$$\mu Y.\nu Z.\,[a]\,((\langle b \rangle \mathrm{tt} \vee Y) \wedge Z)$$

であるとすると、D と D' は性質 Ψ を充足しない、つまり PlayerI は $\mathcal{G}(D', \Psi)$ において、勝利戦略を持つことになる、これを以下に示す。

$$(D', \Psi)(D', \nu Z.\,[a]\,((\langle b\rangle \mathrm{tt} \vee Y) \wedge Z))(D', [a]\,((\langle b\rangle \mathrm{tt} \vee Y) \wedge Z))$$
$$(D, (\langle b\rangle \mathrm{tt} \vee Y) \wedge Z)$$

ここで PlayerI は $(\langle b\rangle \mathrm{tt} \vee Y)$ を選択する。

$$...(D, \langle b\rangle \mathrm{tt} \vee Y)$$

ここで PlayerII が $(D, \langle b\rangle \mathrm{tt})$ を選択すれば即自負けになるので、(D, Y) を選択する、play は続き、

$$(D, Y)(D, \nu Z.\,[a]\,((\langle b\rangle \mathrm{tt} \vee Y) \wedge Z))(D, [a]\,((\langle b\rangle \mathrm{tt} \vee Y) \wedge Z))$$
$$(D', (\langle b\rangle \mathrm{tt} \vee Y) \wedge Z)$$

ここで PlayerI は (D', Z) を選択する

$$...(D', Z)(D', [a]\,((\langle b\rangle \mathrm{tt} \vee Y) \wedge Z))...$$

以上のように、繰り返しが得られて、Z、Y はそれぞれ infinitely often に出現するが、Y は Z を subsume し、Y は最小不動点と関連するから PlayerI が勝つ。すなわち PlayerI は以下のメモリなしの必勝戦略を持ちえる。

- $(D, (\langle b\rangle \mathrm{tt} \vee Y) \wedge Z)$ の場合には $(D, \langle b\rangle \mathrm{tt} \vee Y)$ を選択する
- $(D', (\langle b\rangle \mathrm{tt} \vee Y) \wedge Z)$ の場合には (D', Y) を選択する

A.1.5 Parity Game

まず一般的な (Turn-based でない)Game G を以下のタプルにより定義する。

$$\langle V_E, V_A, T \subset (V_E \cup V_A)^2, A_{cc} \subset V^\omega \rangle$$

また $V = V_E \cup V_A$ として、$v \in V$ に対する successors を $vT = \{v' \in V | (v, v') \in T\}$ と記述する。

無限の play の結果である無限の $v \in V$ の状態列をパス (path) と呼ぶ ($\psi = v_0 v_1 v_2 ... \in V^\omega$)。

ここで A_{cc} は勝利条件 (winning condition) を定義する path の集合で、$\langle V_E \cap V_A, T\rangle$ は Eve と Adam[2]の手番のノードで分割された有向グラフであるといえる。$T(v, v')$ が成立するとき v' を v の successor であるという。V_E, V_A はそれぞれ Eve,Adam の手番である。

無限の play の結果である無限の状態列 $v_0v_1v_2v_3...$ を path と呼ぶ。path が Eve に対して勝利している (winning path for Eve) であるとはパスが A_{cc} に属していることであるとする。

Eve の戦略 σ とは $v \in V_E$ に対して succesor を得る関数 $(\sigma : V_E \to \mathcal{P}(V))$ である。起点 v からの戦略 σ が勝利戦略であるとは、v 起点で σ を施しつつ得られるパスがすべて勝利パスであることである。

メモリ付の戦略とは M をメモリとして、以下の c,up により定義される戦略である

$c : M \times V_E \to \mathcal{P}(V), up : M \times V \to M, m_0 \in M$

ここで up は選択した手に依存してメモリ要素を更新する関数 m_0 は初期メモリであるとする。一般的にはメモリはパス履歴を用いることが多い。メモリが不要な戦略を memory less strategy あるいは history free strategy と呼ぶ。

C を有限の Color の集合、χ を状態から C への写像 $\chi : V \to C$、$\mathcal{F} \subset \mathcal{P}(C)$ とすると勝利条件 (Muller, Rabin, Street, Parity, Buchi condition) は以下のように定義される。

Muller condition $(A_{cc} = \mathcal{F})$:$\pi \in W_\chi(A_{cc})$ iff $Inf(\chi(\pi)) \in \mathcal{F}$

Rabin condition $(A_{cc} = \{(E_0, F_0), (E_1, F_1), ..., (E_{m-1}, F_{m-1})\})$:$\pi \in W_\chi(A_{cc})$ iff $\exists k \in [m]$ s.t. $Inf(\chi(\pi)) \cap E_k = \emptyset \vee Inf(\chi(\pi)) \cap F_k \neq \emptyset$

Streett condition $(A_{cc} = \{(E_0, F_0), (E_1, F_1), ..., (E_{m-1}, F_{m-1})\})$:$\pi \in W_\chi(A_{cc})$ iff $\exists k \in [m]$ s.t. $Inf(\chi(\pi)) \cap E_k \neq \emptyset \wedge Inf(\chi(\pi)) \cap F_k = \emptyset$

Parity condition (C は有限な整数の集合)

- max parity condition:$\pi \in W_\chi(A_{cc})$ iff $max(Inf(\chi(\pi)))$ が偶数
- min parity condition:$\pi \in W_\chi(A_{cc})$ iff $min(Inf(\chi(\pi)))$ が偶数

Buchi condition $(A_{cc} = F \in C)$:$\pi \in W_\chi(A_{cc})$ iff $Inf(\chi(\pi)) \cap F \neq \emptyset$

ここで $Inf(\chi(\pi))$ は $\chi(\pi)$ に無限の頻度で (infinitely often) 出現する色の集合を指す。"Parity condition"は Muller condition の特殊ケースである。

[2]Player1 と Player2 とも呼ばれる

Muller ゲーム、Buchi ゲーム、Rabin ゲーム、Street ゲームはすべて Parity ゲームに帰着することができることが知られており ([15])、これらを regular game と呼ぶ。

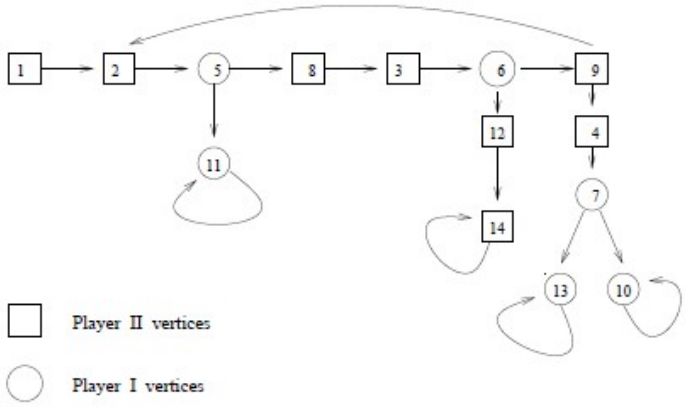
X condition を勝利条件とするゲームを X ゲーム (例 parity ゲーム) と呼ぶ。

A.1.6 PGSolve による充足判定例

Game による μ 計算のモデル検査の解法は parity game として読み替えることができる。すなわち play における PlayerI,II の手番 (E_i, Φ_j) に対して $\{1, 2\}$ をそれぞれ割り当てるラベルを用意すればよい。

parity game の定義より、play の中で infinitely often に現れるラベルが 1 の場合は PlayerI が、そうでない場合には PlayerII が勝利がわかる。

$D \xrightarrow{a} D', D' \xrightarrow{a} D', D \xrightarrow{b} D''$ であるようなモデルに対して $\mathcal{G}(D, \nu Z.\langle b\rangle \mathrm{tt} \wedge \langle -\rangle Z)$ を展開すると以下のグラフ A.1 を得る。



1	$(\mathtt{D}, \nu Z.\langle \mathrm{b}\rangle \mathtt{tt} \wedge \langle -\rangle Z)$	8	$(\mathtt{D}, \langle -\rangle Z)$
2	$(\mathtt{D}, Z)$	9	$(\mathtt{D}', \langle -\rangle Z)$
3	$(\mathtt{D}', Z)$	10	$(\mathtt{D}'', \langle -\rangle Z)$
4	$(\mathtt{D}'', Z)$	11	$(\mathtt{D}, \langle \mathrm{b}\rangle \mathtt{tt})$
5	$(\mathtt{D}, \langle \mathrm{b}\rangle \mathtt{tt} \wedge \langle -\rangle Z)$	12	$(\mathtt{D}', \langle \mathrm{b}\rangle \mathtt{tt})$
6	$(\mathtt{D}', \langle \mathrm{b}\rangle \mathtt{tt} \wedge \langle -\rangle Z)$	13	$(\mathtt{D}'', \langle \mathrm{b}\rangle \mathtt{tt})$
7	$(\mathtt{D}'', \langle \mathrm{b}\rangle \mathtt{tt} \wedge \langle -\rangle Z)$	14	$(\mathtt{D}'', \mathtt{tt})$

図 A.1: Game

s1:$(D, \nu Z.\langle b\rangle \mathrm{tt} \wedge \langle -\rangle Z)$	s8:$(D, \langle -\rangle Z)$
s2:(D, Z)	s9:$(D', \langle -\rangle Z)$
s3:(D', Z)	s10:$(D'', \langle -\rangle Z)$
s4:(D'', Z)	s11:$(D, \langle b\rangle \mathrm{tt})$
s5:$(D, \langle b\rangle \mathrm{tt} \wedge \langle -\rangle Z)$	s12:$(D', \langle b\rangle \mathrm{tt})$
s6:$(D' \langle b\rangle \mathrm{tt} \wedge \langle -\rangle Z)$	s13:$(D'', \langle b\rangle \mathrm{tt})$
s7:$(D'', \langle b\rangle \mathrm{tt} \wedge \langle -\rangle Z)$	s14:(D'', tt)

とおくと、PlayerI,II が player 0,1 に対応することに注意すると以下の PGSolver 向けの問題にエンコードできる。

```
parity 14;
1 2 1 2 "s1";
2 2 1 5 "s2";
5 1 0 8,11 "s5";
11 1 0 11 "s11";
8 2 1 3 "s8";
3 2 1 6 "s3";
6 1 0 9,12 "s6";
12 2 1 14 "s12";
14 2 1 14 "s14";
9 2 1 2,4 "s9";
4 2 1 7 "s4";
7 1 0 13,10 "s7";
13 1 0 13 "s13";
10 1 0 10 "s10";
```

PGSolver により以下の解を得る。

```
$ bin/pgsolver.exe --recursive -d staring.dot sample/starling.gm
Player 0 wins from nodes:
  {1,2,3,5,6,8,12,14}
with strategy
  [5->8,6->12]

Player 1 wins from nodes:
  {4,7,9,10,11,13}
with strategy
  [4->7,9->4]
```

すなわち $\mathcal{G}(D, \nu Z.\langle b\rangle \mathrm{tt} \wedge \langle -\rangle Z)$ は s1 であり s1 は player 0(PlayerI) の勝利ノードであるから、$D \models \nu Z.\langle b\rangle \mathrm{tt} \wedge \langle -\rangle Z$ は成立しない、すなわち充足しない。

以上を図示したものが図 A.2 である。

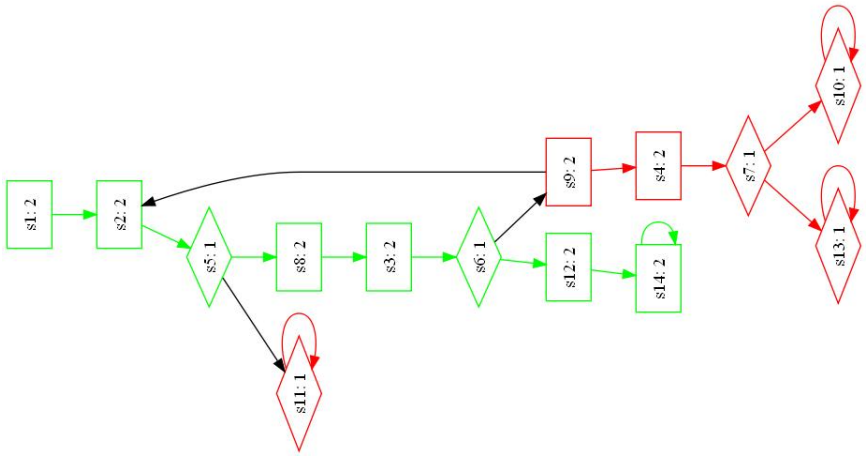


図 A.2: PGSolver の解

A.2 CCS(Calculus of Communicating Systems)

CCS[20] は

定義 A.2.1 Labelled Transition System LTS(Labelled Transition System) は，$(Proc, Act, \{\xrightarrow{a} | a \in Act\})$ であり

- $Proc$ は状態 s の集合
- Act はアクション a の集合

以下は CCS の逐次部分である.

- Nil (or 0) プロセス (無動作)
- アクション・プレフィックス (action prefixing) $(a.P)$
  アクション a を行って P になる
- プロセス名 (names) と 再帰定義 $(\stackrel{\text{def}}{=})$
- 非決定的選択演算 $(+)$

有限の LTS は，以上の構成により表す事ができる.
以下は CCS の並列およびリネームに関する部分である

- 並列結合演算:parallel composition $(|)$
  ２つの LTS 間の同期通信により実現
- 動作制限:restriction $(P \setminus L)$
  L に含まれているアクションは行わない
- リネーム:relabelling $(P[f])$

CCS 基本要素の定義

- $\mathcal{A}$: チャネル名 (channel names) の集合 (e.g. チャネル名の例：tea, $coffee$)

- $\mathcal{L} = \mathcal{A} \cup \overline{\mathcal{A}}$ は labels の集合であり以下を充たす
  - $\overline{\mathcal{A}} = \{\overline{a} \mid a \in \mathcal{A}\}$
    ($\mathcal{A}$ は name と呼ばれ $\overline{\mathcal{A}}$ は co-names と呼ばれる)
  - co-name の co-name は name である $\overline{\overline{a}} = a$
- $Act = \mathcal{L} \cup \{\tau\}$ は アクション::actions であり，
  - τ は内部 あるいは 無言 アクションである

  (e.g. アクションの例：τ, tea, $\overline{coffee}$)
- $\mathcal{K}$ は プロセス名 (定数) (e.g. CM).

以下は CCS における式 (expression) の定義である

$P :=$	K	|	プロセス定数 ($K \in \mathcal{K}$)
	$\alpha.P$	|	プレフィックス ($\alpha \in Act$)
	$\sum_{i \in I} P_i$	|	総和 (I is an arbitrary index set)
	$P_1 \mid P_2$	|	並列結合
	$P \setminus L$	|	動作制限 ($L \subseteq \mathcal{A}$)
	$P[f]$	|	リラベリング ($f : Act \to Act$) であり，以下の性質を持つ • $f(\tau) = \tau$ • $f(\overline{a}) = \overline{f(a)}$

上記，抽象構文から生成された全ての項を，“CCS のプロセス式”($\mathcal{P}$ と記す) と呼ぶ
以下の省略をすることがある．

$$P_1 + P_2 = \sum_{i \in \{1,2\}} P_i$$
$$Nil = 0 = \sum_{i \in \emptyset} P_i$$

演算子の強さは以下の順である．
選択演算 < 並列結合 < 動作プレフィックス < 動作制限，理ラベリング

A.2.1 CCS における定義 (定義方程式)

CCS プログラムは，以下の形式の定義方程式:defining equations の集合である.

$$K \stackrel{\text{def}}{=} P$$

ここで $K \in \mathcal{K}$ はプロセス定数であり $P \in \mathcal{P}$ は CCS プロセス式である

- 一つのプロセス定数に対しては，1 つの定義方程式のみ許される
- 再帰定義は可能である: e.g. $A \stackrel{\text{def}}{=} \overline{a}.A \mid A.$

A.2.2 SOS(構造的操作意味論) による CCS の意味定義

CCS 定義方程式の集合に対して，以下の LTS を定義する $(Proc, Act, \{\stackrel{a}{\longrightarrow} \mid a \in Act\})$:

- $Proc = \mathcal{P}$ (CCS のプロセス式の全集合)
- $Act = \mathcal{L} \cup \{\tau\}$ (τ を含む CCS 動作の全集合)
- 遷移関係は以下の SOS rules 形式で表現される:

$$\text{RULE}\ \frac{\text{前提}}{\text{帰結}}\ \text{条件}$$

$$\text{ACT}\ \frac{}{\alpha.P \stackrel{\alpha}{\longrightarrow} P} \qquad \text{SUM}_j\ \frac{P_j \stackrel{\alpha}{\longrightarrow} P'_j}{\sum_{i\in I} P_i \stackrel{\alpha}{\longrightarrow} P'_j}\ j \in I$$

$$\text{COM1}\ \frac{P \stackrel{\alpha}{\longrightarrow} P'}{P|Q \stackrel{\alpha}{\longrightarrow} P'|Q} \qquad \text{COM2}\ \frac{Q \stackrel{\alpha}{\longrightarrow} Q'}{P|Q \stackrel{\alpha}{\longrightarrow} P|Q'}$$

$$\text{COM3}\ \frac{P \stackrel{a}{\longrightarrow} P' \quad Q \stackrel{\overline{a}}{\longrightarrow} Q'}{P|Q \stackrel{\tau}{\longrightarrow} P'|Q'}$$

$$\text{RES}\ \frac{P \stackrel{\alpha}{\longrightarrow} P'}{P \setminus L \stackrel{\alpha}{\longrightarrow} P' \setminus L}\ \alpha, \overline{\alpha} \notin L \qquad \text{REL}\ \frac{P \stackrel{\alpha}{\longrightarrow} P'}{P[f] \stackrel{f(\alpha)}{\longrightarrow} P'[f]}$$

$$\text{CON}\ \frac{P \stackrel{\alpha}{\longrightarrow} P'}{K \stackrel{\alpha}{\longrightarrow} P'}\ K \stackrel{\text{def}}{=} P$$

A.2.3 等価性

等価性に関する共通な性質

プロセスが等価であることに関する性質を以下に挙げる，

- 反射性:reflexivity 全ての P に対して $P \equiv P$
- 推移性:transitivity $Spec_0 \equiv Spec_1 \equiv Spec_2 \equiv \cdots \equiv Impl$ ならば
$$Spec_0 \equiv Impl$$
- 対象性:symmetry $P \equiv Q \Leftrightarrow Q \equiv P$
- 合同性:congruence $P \equiv Q$ ならば $C(P) \equiv C(Q)$

トレース等価性

トレース等価性とは，以下を示すことである．

$(Proc, , \{\xrightarrow{a} | \ a \in \})$ を LTS として，まず $s \in Proc$ に対する Trace 集合を以下のように定義する，

$$Traces(s) = \{w \in Act^* \mid \exists s' \in Proc.\ s \xrightarrow{w} s'\}$$

$s \in Proc$ および $t \in Proc$ に対して，s と t がトレース等価:trace equivalent $(s \equiv_t t)$ であることの必要十分条件は，以下である．

$$Traces(s) = Traces(t)$$

強双模倣等価性

$(Proc, Act, \{\xrightarrow{a} | \ a \in Act\})$ を LTS であるとすると，

2 項関係 $R \subseteq Proc \times Proc$ は強双模倣:strong bisimulation であることの必要十分条件は，あらゆる $(s,t) \in R$ において，$a \in Act$ に対し，

- $s \xrightarrow{a} s'$ ならば，$t \xrightarrow{a} t'$ となる t' で $(s', t') \in R$ が成り立つものが存在する

- $t \xrightarrow{a} t'$ ならば， $s \xrightarrow{a} s'$ となる s' で $(s', t') \in R$ が成り立つものが存在する

定義 A.2.2 Strong Bisimilarity 2つのプロセス $p_1, p_2 \in Proc$ が強双模倣:strongly bisimilar $(p_1 \sim p_2)$ であることの必要十分条件は，強双模倣関係 R が存在し，$(p_1, p_2) \in R$ に対して以下が成り立つことである．

$$\sim = \cup\{R \mid R \text{ is a strong bisimulation}\}$$

強双模倣関係は以下の性質を充たす，

定理 A.2.1 その1 $\sim$ は，反射的，対称的，推移的 (reflexive, symmetric and transitive) な等価性である

定理 A.2.2 その2 $\sim$ は最大の強双模倣関係である

定理 A.2.3 その3 $s \sim t$ であることの必要十分条件は，各々の $a \in Act$ に対して，

- $s \xrightarrow{a} s'$ ならば $t \xrightarrow{a} t'$ となる t' で，$s' \sim t'$ が成立するものが存在する
- $t \xrightarrow{a} t'$ ならば $s \xrightarrow{a} s'$ となる s' で， $s' \sim t'$ が成立するものが存在する

強双模倣性は，CCSの演算に対して，合同性 (Congruence) を持つ．

定理 A.2.4 その4 P と Q を CCS プロセスで，強双模倣関係:$P \sim Q$ であるとすると，CCSの演算に対して，以下 が成り立つ

- $\alpha.P \sim \alpha.Q$ ：各々の動作 $\alpha \in Act$ に対して，
- $P + R \sim Q + R$,$R + P \sim R + Q$：各々のCCS プロセス R に対して，
- $P \mid R \sim Q \mid R$, $R \mid P \sim R \mid Q$：各々のCCSプロセス R に対して，
- $P[f] \sim Q[f]$：各々のrelabelling 関数 f に対して，
- $P \setminus L \sim Q \setminus L$：各々のラベルの集合 L に対して，

また以下の性質がCCSプロセス P, Q, R の間で成り立つ

- $P + Q \sim Q + P$
- $P \mid Q \sim Q \mid P$
- $P + Nil \sim P$
- $P \mid Nil \sim P$
- $(P + Q) + R \sim P + (Q + R)$
- $(P \mid Q) \mid R \sim P \mid (Q \mid R)$

弱双模倣性

$(Proc, , \{\stackrel{a}{\longrightarrow} | \ a \in \})$ である LTS において $\tau \in Act$ とすると，

定義 A.2.3 弱遷移関係弱遷移関係 $\stackrel{a}{\Longrightarrow}$ は以下のように定義される

$$\stackrel{a}{\Longrightarrow} = \begin{cases} (\stackrel{\tau}{\longrightarrow})^* \circ \stackrel{a}{\longrightarrow} \circ (\stackrel{\tau}{\longrightarrow})^* & \text{if } a \neq \tau \\ (\stackrel{\tau}{\longrightarrow})^* & \text{if } a = \tau \end{cases}$$

$s \stackrel{a}{\Longrightarrow} t$ は直感的には以下の意味を持つ

- $a \neq \tau$ ならば $s \stackrel{a}{\Longrightarrow} t$ とは
  s から t にゼロ以上の τ 動作，動作 a，そしてゼロ以上の τ 動作で到達できる
- $a = \tau$ ならば $s \stackrel{\tau}{\Longrightarrow} t$ とは
  s から t にゼロ以上の τ 動作で到達できる

$s \stackrel{\hat{a}}{\Longrightarrow} t$ は，$a \neq \tau$ の特別な場合であって，$\hat{a}$ は，$\epsilon (a = \tau$ の場合)，それ以外の場合は a という意味である．すなわち $s \stackrel{\hat{a}}{\Longrightarrow} t$ は ϵ 遷移を含むが $s \stackrel{a}{\Longrightarrow} t$ は ϵ 遷移を含まない．

$(Proc, Act, \{\stackrel{a}{\longrightarrow} | \ a \in Act\})$ を $\tau \in Act$ であるような LTS であるとすると，

定義 A.2.4 弱双模倣関係:Weak Bisimulation 二項関係 $R \subseteq Proc \times Proc$ は，弱双模倣:weak bisimulation であることの必要十分条件は，

$(s, t) \in R$ ならば常に，$a \in Act$(含む τ) に対して

- $s \stackrel{a}{\longrightarrow} s'$ ならば， $t \stackrel{\hat{a}}{\Longrightarrow} t'$ であり $(s', t') \in R$ であるような t' が存在する

- もし $t \xrightarrow{a} t'$ ならば $s \stackrel{\hat{a}}{\Longrightarrow} s'$ であり $(s', t') \in R$ であるような s' が存在する

定義 A.2.5 弱双模倣性：Weak Bisimilarity 2 つのプロセス $p_1, p_2 \in Proc$ が弱双模倣である：weakly bisimilar $(p_1 \approx p_2)$ ための必要十分条件は，弱双模倣関係 R が存在し，$(p_1, p_2) \in R$ であることである

$$\approx = \cup\{R \mid R \text{ は弱双模倣関係 }\}$$

弱双模倣性は観測等価性 (Observation Equivalence) とも呼ばれる．
弱双模倣性の性質は以下である，

- 等価性関係 (equivalent relation) である
- 最大の双模倣関係である
- 多くの自然法則を充たす，
  - $a.\tau.P \approx a.P$
  - $P + \tau.P \approx \tau.P$
  - $a.(P + \tau.Q) \approx a.(P + \tau.Q) + a.Q$
  - $P + Q \approx Q + P \qquad P|Q \approx Q|P \qquad P + Nil \approx P \quad \ldots$
- 強双模倣性は弱双模倣性に含まれる $(\sim \subseteq \approx)$
- τ 動作ループを抽象化する

弱双模倣性は，CCS において合同性を持つか？

定理 A.2.5 弱双模倣性の合同性 P と Q を CCS プロセスとして，$P \approx Q$ ならば

- 全ての $\alpha \in Act$ 動作に対して，$\alpha.P \approx \alpha.Q$，
- 全ての CCS プロセス R に対して，$P \mid R \approx Q \mid R$ また $R \mid P \approx R \mid Q$
- 全てのリラベリング関数 f に対して，$P[f] \approx Q[f]$

- 全てのラベルの集合 L に対して $P \setminus L \approx Q \setminus L$

しかし，和演算に対しては，

$\tau.a.Nil \approx a.Nil$　　であるが，しかし　　$\tau.a.Nil+b.Nil \not\approx a.Nil+b.Nil$

すなわち，弱双模倣性は，CCS に対して合同性を持たない．これは弱双模倣性が Or 分岐に対して，fairness を仮定しているからである（無限に τ 遷移を繰り返すループを含んでも，他に等価なパスが存在すればかまわないということ）．

弱合同性

定義 A.2.6 弱合同関係:Weak Congruence 二項関係 $R \subseteq Proc \times Proc$ は，弱合同:weak congruent であることの必要十分条件は，

$(s,t) \in R$ ならば常に，$a \in Act$(含む τ) に対して

- $s \xrightarrow{a} s'$ ならば，$t \stackrel{a}{\Longrightarrow} t'$ であり $(s',t') \in R$ であるような t' が存在する
- もし $t \xrightarrow{a} t'$ ならば $s \stackrel{a}{\Longrightarrow} s'$ であり $(s',t') \in R$ であるような s' が存在する

弱合同性は観測合同性 (Observation Congruence) とも呼ばれる．

$s \stackrel{a}{\Longrightarrow} t$ は ϵ 遷移を含まないので，τ 遷移には少なくとも 1 つの τ 遷移が対応するという特徴がある．

弱合同性は，CCS に対して合同性をもつ．

A.2.4　様相論理 (HML とその拡張)

様相論理は,「〜でなければならない」「〜でありうる」「〜べきである」といった可能性や必然性に関わる命題を扱う論理である．

特に safety property(悪いことは決して起こらない)，liveness property(何か良いことがいつか起きる) が代表的である．CCS における様相論理として Hennessy-Milner Logic(以下 HML) をここでは採用する．

HML 式の文法

Act をアクションの集合，$a \in Act$ であるとすると HML 式 F, G は以下の要素で構成される，

$$F, G \ ::= \ tt \ \mid \ ff \ \mid \ F \wedge G \ \mid \ F \vee G \ \mid \ \langle a \rangle F \ \mid \ [a]F$$

各々の式要素の直感的な意味は以下のとおり，

tt 全てのプロセスがこのプロパティを充足する

ff どのプロセスもこのプロパティを充足しない

$\wedge, \vee$ 論理積と論理和

$\langle a \rangle F$ a 遷移後の状態で F を充足するものが少なくとも 1 つある

$[a]F$ 全ての a 遷移後の状態は F を充足する

HML 式の表示的意味

HML の表示的意味 (denotational semantics) は以下のように定義できる．

HML 式 F に対して $[\![F]\!] \subseteq Proc$ は F を充足するすべての状態であるとすると HML 式の構成要素の意味は $[\![F]\!]$ を用いて以下のように定義される．

- $[\![tt]\!] = Proc$
- $[\![ff]\!] = \emptyset$
- $[\![F \vee G]\!] = [\![F]\!] \cup [\![G]\!]$
- $[\![F \wedge G]\!] = [\![F]\!] \cap [\![G]\!]$
- $[\![\langle a \rangle F]\!] = \langle \cdot a \cdot \rangle [\![F]\!]$
- $[\![[a]F]\!] = [\cdot a \cdot][\![F]\!]$

ここで $\langle\cdot a\cdot\rangle, [\cdot a\cdot] : 2^{(Proc)} \to 2^{(Proc)}$ は,

$$\langle\cdot a\cdot\rangle S = \{p \in Proc \mid \exists p'.\ p \xrightarrow{a} p' \text{ and } p' \in S\}$$

$$[\cdot a\cdot] S = \{p \in Proc \mid \forall p'.\ p \xrightarrow{a} p' \implies p' \in S\}.$$

である.

HML で直接表せない性質

HML では以下のような性質は直接表現できない,

- $s \models Inv(F)$:s から到達可能な全ての状態が F を充足すること
- $s \models Pos(F)$:s から到達可能な状態に少なくとも 1 つは F を充足すること

なぜならば以下のような無限の式になるからである.

$$Inv(F) \equiv F \wedge [Act]F \wedge [Act][Act]F \wedge [Act][Act][Act]F \wedge \ldots$$
$$Pos(F) \equiv F \vee \langle Act\rangle F \vee \langle Act\rangle\langle Act\rangle F \vee \langle Act\rangle\langle Act\rangle\langle Act\rangle F \vee \ldots$$

一般的にこのような問題の典型的な解決方法は再帰的な定義を用いることである.

- $Inv(F)$ は $X \stackrel{\text{def}}{=} F \wedge [Act]X$ により表現され
- $Pos(F)$ は $X \stackrel{\text{def}}{=} F \vee \langle Act\rangle X$ により表現される

これらの表現の課題は再帰変数の意味の定義であり，ここでは状態の集合で定義する
$X \stackrel{\text{def}}{=} [a]ff \vee \langle a\rangle X \quad \Rightarrow \quad S = [\cdot a\cdot]\emptyset \cup \langle\cdot a\cdot\rangle S$ であるような $S \subseteq 2^{Proc}$ を見つける.
このような状態の集合を求めることは $x = f(x)$ の解を求めていることになり，不動点理論を適用することができる

不動点理論

$(D, \sqsubseteq)$ を半順序集合と半順序関係であるとする．X を D の部分集合であるとするとき X の上界，下界，上限，下限を以下のように定義する.

- $d \in D$ は X の**上界 (upper bound)**($X \sqsubseteq d$ と書く)
  $\Leftrightarrow$ 全ての $x \in X$ に対して $x \sqsubseteq d$
- $d \in D$ は X の**下界 (lower bound)**($d \sqsubseteq X$ と書く)
  $\Leftrightarrow$ 全ての $x \in X$ に対し $d \sqsubseteq x$
- $d \in D$ は X の**上限 (least upper bound または supremum)** ($\sqcup X$ と書く) $\Leftrightarrow$
  1. $X \sqsubseteq d$
  2. $\forall d' \in D.\ X \sqsubseteq d' \ \Rightarrow\ d \sqsubseteq d'$
- $d \in D$ は X の**下限 (greatest lower bound または infimum)** ($\sqcap X$ と書く) $\Leftrightarrow$
  1. $d \sqsubseteq X$
  2. $\forall d' \in D.\ d' \sqsubseteq X \ \Rightarrow\ d' \sqsubseteq d$

半順序集合 $(D, \sqsubseteq)$ はいかなる D の部分集合 X においても $\sqcup X$ と $\sqcap X$ が存在するときそのときに限り，**"complete lattice"** であるという.

Tarski の不動点定理

定理 A.2.6 (Tarski の不動点定理) $(D, \sqsubseteq)$ を"complete lattice" であり，$f : D \to D$ を 単調関数であるとする.

ことき f は以下で定義される，唯一の最大不動点 (largest fixed point) z_{max} ，および唯一の 最小不動点 (least fixed point) z_{min} を持つ.

$$z_{max} \stackrel{\text{def}}{=} \sqcup\{x \in D \mid x \sqsubseteq f(x)\}$$
$$z_{min} \stackrel{\text{def}}{=} \sqcap\{x \in D \mid f(x) \sqsubseteq x\}$$

最大（小）不動点の求め方（有限集合の場合）

D が有限集合ならば整数 $M, m > 0$ が存在し，以下のように $\top$(全体集合)，$\bot$(空集合) から逐次繰り返し (picard iteration と呼ぶ) により最大（小）不動点を求めることができる.

- $z_{max} = f^M(\top)$
- $z_{min} = f^m(\bot)$

再帰変数を含む HML の意味定義

以下の 2 つの再帰変数の定義式の意味を導入する

- $X \stackrel{\min}{=} F_X$, $X \stackrel{\max}{=} F_X$

最初に全ての式 F に対して以下の性質をもつ関数 $O_F : 2^{Proc} \to 2^{Proc}$ を導入する.

- もし S が X を充足するプロセスの集合ならば
- $O_F(S)$ は F を充足するプロセスの集合である

HML 式 X に対して以下の意味を持つことがわかる.

$$
\begin{aligned}
O_X(S) &= S \\
O_{tt}(S) &= Proc \\
O_{ff}(S) &= \emptyset \\
O_{F_1 \wedge F_2}(S) &= O_{F_1}(S) \cap O_{F_2}(S) \\
O_{F_1 \vee F_2}(S) &= O_{F_1}(S) \cup O_{F_2}(S) \\
O_{\langle a \rangle F}(S) &= \langle \cdot a \cdot \rangle O_F(S) \\
O_{[a]F}(S) &= [\cdot a \cdot] O_F(S)
\end{aligned}
$$

さらに O_F は単調関数であることより，$(2^{Proc}, \subseteq)$ が"complete lattice" であり O_F が単調であるから, 不動点定理より O_F は唯一の最大（小）不動点を持つ.

再帰変数 X の意味

- $X \stackrel{\max}{=} F_X$ の意味→

$$[\![X]\!] = \bigcup \{S \subseteq Proc \mid S \subseteq O_{F_X}(S)\}.$$

- $X \stackrel{\min}{=} F_X$ の意味→

$$[\![X]\!] = \bigcap \{S \subseteq Proc \mid O_{F_X}(S) \subseteq S\}.$$

直感的には最大解 (maximal solution) は，良い性質を阻害する有限遷移列が一つも存在しないことを，最小解 (minimal solution) は，逆に有限遷移列の一つでも良い性質を充たす場合に使われる．

$Inv(F), Pos(F)$ の正しい定義は以下になる．他に代表的なプロパティ式の意味も併記する．

- $Inv(F)$: $X \stackrel{\max}{=} F \wedge [Act]X$
- $Pos(F)$: $X \stackrel{\min}{=} F \vee \langle Act\rangle X$
- $Safe(F)$: $X \stackrel{\max}{=} F \wedge ([Act]ff \vee \langle Act\rangle X)$
- $Even(F)$: $X \stackrel{\min}{=} F \vee (\langle Act\rangle tt \wedge [Act]X)$
- $F\ \mathcal{U}^w\ G$: $X \stackrel{\max}{=} G \vee (F \wedge [Act]X)$
- $F\ \mathcal{U}^s\ G$: $X \stackrel{\min}{=} G \vee (F \wedge \langle Act\rangle tt \wedge [Act]X)$

特に $Safe(F)$ は F を充たし続ける遷移列が 1 つでも存在すること，$Even(F)$ は F を充たす状態を含むことという性質でありよく使われる．

これにより deadlock 検知で用いる以下の性質は $Pos([Act]ff)$ に対応していることがわかる．

```
prop can_deadlock =
   min X = [-]ff \/ <->X
```

A.3 CWB-NC について

CWB-NC(Concurrent Work Bench of the New Century)[1] は，CCS(Calculus of Communicating Systems) で仕様を書いて，検証等を行うツールである．

A.3.1 プロセス定義

プロセスは以下の形式 (proc 文) で記述する

```
proc Spec = pub.Spec
proc CM = 'coin.coffee.CM
proc CTM = 'coin.coffee.CTM +'coin.tee.CTM
proc CS = pub.coin.'coffee.CS
```

pub(論文を出版する)，coin(コインを入れる)，coffee(コーヒーを飲む) は出力アクションで，アクションに' が付いたものが入力アクションを示す．

CM はコインを入力としてコーヒーを出力する自販機，CTM はコインを入力して，非決定的にコーヒーを出力するか紅茶を出力する自販機，CS はコインを入れてコーヒーを受け取って，論文を出力する計算機科学の学生の仕様を表している．

CCS における，並列結合および動作制限は同様に記述される

例えば $Sys \stackrel{\text{def}}{=} (\texttt{CTM}|\texttt{CS}) \setminus \{coin, coffee, tee\}$ は以下のように定義される，

```
proc Sys = (CTM | CS) \ {coin, coffee, tee}
```

A.3.2 等価性の判定

また，CWB-NC は，以下の等価性を検査するコマンドを備えている

eq -S 等価性指示子 プロセス 1 プロセス 2

ここで，可能な等価性指示子を示す

強双模倣性 A と B の双模倣性を以下のコマンドで確認できる

```
eq -S bsim A B
```

trace 等価性 A と B の trace 等価性を以下のコマンドで確認できる

```
eq -S trace A B
```

弱双模倣性 A と B の弱双模倣性（あるいは観測等価性) を以下のコマンドで確認できる

```
eq -S obseq A B
```

default 等価性指示子を省略するとデフォルトは，弱双模倣性（観測等価性）が選択される．

例えば，以下のセッションのように，Spec と Sys の間の観測透過性を検査すると，Sys が [pub]<pub>tt を充足しないと表示される，これは CTM が tea を選択した場合に，デッドロックが発生するからである．

一方 trace 等価性ならば，受理するトレース（デッドロックは含まない）は同じなので，等価であると判定される．

```
cwb-nc> eq Spec Sys
Building automaton...
States: 6
Transitions: 6
Done building automaton.
Transforming automaton...
Done transforming automaton.
FALSE...
Spec satisfies:
        [[pub]]<<pub>>tt
Sys does not.
Execution time (user,system,gc,real):(0.047,0.000,0.000,0.047)

cwb-nc> eq -S trace Spec Sys
Building automaton...
States: 6
Transitions: 6
Done building automaton.
Transforming automaton...
Done transforming automaton.
TRUE
Execution time (user,system,gc,real):(0.000,0.000,0.000,0.000)
```

A.3.3 様相論理と到達可能性判定

様相論理をもちいた，到達可能性の判定もできる．CWB では HML 式にて様相論理を表現する．

例えば，いかなるアクションをも実行不能である状況を以下の `can_deadlock` という命題で表現することができる．`[-]ff` は全てのアクションがまったく実行できない状況を表

し，この状態に到達するかどうかを再帰的に X は定義している．min は最小不動点演算に関係する．

```
prop can_deadlock =
   min X = [-]ff \/ <->X
```

命題が定義できれば，プロセスに対して命題が成立する場合に到達可能であるかを判定できる．いま can_deadlock をファイル”dead.mu” に保存してあるものとして，上記 Sys がデッドロック状況に到達可能であるかどうかを判定する．

```
cwb-nc> load dead.mu
cwb-nc> search Sys can_deadlock
Building automaton...
States: 5
Transitions: 5
Done building automaton.

States explored: 1
State found satisfying can_deadlock.
Path to state contains 1 states, invoking simulator.
1: Sys
cwb-nc-sim>
```

充足可能であること，すなわちデッドロックに到達可能であることがわかる．

今度は，tee を出力しない自販機 CM との組合せに対してデッドロックを判定する

```
proc SmUni = (CM | CS) \ {coin, coffee}
```

```
cwb-nc> search SmUni can_deadlock
Building automaton...
States: 4
Transitions: 4
Done building automaton.
Building automaton...
States: 3
...略...


States explored: 4
No state found satisfying can_deadlock.
```

すなわち，SmUni に対しては deadlock に到達可能でないことがわかる．

このように，CCS に従ってプロセスを定義し，透過性を検証し，また様相論理式をもちいて性質を定義して，充足性を判定できる．

参考文献

[1] The concurrency workbench of the new century. http://www.cs.sunysb.edu/c̃wb.

[2] L. Aceto, A. Ingolfsdottir, and J. Srba. *Reactive Systems: Modelling, Specification and Verification*. Cambridge University Press, 2007.

[3] R. Alur, K. Etessami, and M. Yannakakis. Inference of message sequence charts. *Software Engineering, IEEE Transactions on*, 29(7):623–633, 2003.

[4] J. C. M. Baeten. A brief history of process algebra. *Theor. Comput. Sci.*, 335(2-3):131–146, 2005.

[5] JCM Baeten and JA Bergstra. Discrete time process algebra. *Formal Aspects of Computing*, 8(2):188–208, 1996.

[6] J.C.M. Baeten and W.P. Weijland. *Process algebra*. Cambridge University Press Cambridge, 1990.

[7] J.A. Bergstra and J.W. Klop. The Algebra of Recursively Defined Processes and the Algebra of Regular Processes. 1983.

[8] J.A. Bergstra and J.W. Klop. Process Algebra for Synchronous Communication. *Information and Control*, 60(1-3):109–137, 1984.

[9] A. Church. Logic, arithmetic and automata. In *Proc. Internat. Congr. on Mathematics, Moscow*, 1963.

[10] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model checking*. Springer, 1999.

[11] L. de Alfaro, L.D. da Silva, M. Faella, A. Legay, P. Roy, M. Sorea, et al. Sociable interfaces. *Procedings of FROCOS*, 5, 2005.

[12] EA Emerson, CS Jutla, and AP Sistla. On model-checking for fragments of t-calculus. In *CAV'93*, volume 697, pages 385–396. Springer, 1993.

[13] W. Fokkink. *Introduction to Process Algebra*. Springer, 2000.

[14] D. Giannakopoulou, C.S. Pasareanu, and H. Barringer. Assumption generation for software component verification. In *Proceedings of the 17th IEEE international conference on Automated software engineering*, page 3. IEEE Computer Society Washington, DC, USA, 2002.

[15] E. Gradel, W. Thomas, and T. Wilke. *Automata, logics, and infinite games*. Springer, 2002.

[16] Y. Gurevich and L. Harrington. Trees, automata, and games. In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pages 60–65. ACM New York, NY, USA, 1982.

[17] B. Jobstmann, S. Galler, M. Weiglhofer, and R. Bloem. Anzu: A tool for property synthesis. *Lecture Notes in Computer Science*, 4590:258, 2007.

[18] M.Iwamasa K.Yamamoto, T.Ishii. System Level Specification Synthesis-An Extended Message Sequence Chart based Approach-. *The 13rd Workshop on Synthesis And System Integration of Mixed Information technologies(SASIMI)*, 2006.

[19] H. Liang, J. Dingel, and Z. Diskin. A comparative survey of scenario-based to state-based model synthesis approaches. *Proceedings of the 2006 international workshop on Scenarios and state machines: models, algorithms, and tools*, pages 5–12, 2006.

[20] R. Milner. A Calculus of Communicating Systems. *LNCS 92 Springer-Verlag*, 1980.

[21] M. Lange O. Friedmann. The pgsolver collection of parity game solvers. Technical report, Ludwig-Maximilians-Universit ¨ at M ¨ unchen,, 2009.

[22] N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive (1) designs. *Lecture notes in computer science*, 3855:364, 2006.

[23] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 179–190. ACM New York, NY, USA, 1989.

[24] PJG Ramadge and WM Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, 1989.

[25] P. Stevens and C. Stirling. Practical model checking using games. *Lecture Notes in Computer Science*, 1384:85–101, 1998.

[26] C. Stirling. Games and Modal Mu-Calculus. *LECTURE NOTES IN COMPUTER SCIENCE*, pages 298–312, 1996.

[27] C. Stirling. Bisimulation, modal logic and model checking games. *Logic Journal of IGPL*, 7(1):103–124, 1999.

[28] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

[29] S. Uchitel and M. Chechik. Merging partial behavioural models. *SIGSOFT Softw. Eng. Notes*, 29(6):43–52, 2004.

[30] I. Walukiewicz. A landscape with games in the background. In *Proceedings of LICS*, volume 4, pages 356–366, 2004.

[31] 岩政幹人、渡邊竜明. ツンターフデツベ理論に基づく分割統治型検証の試作と評価. **情報処理学会組込みシステムシンポジウム** *2009(ESS2009)*, Oct.2009.

[32] 岩政幹人、日比野靖. プロセス代数に基づくシステムレベル仕様合成. **情報処理学会誌**, 50(11):2633–2642, 2009.