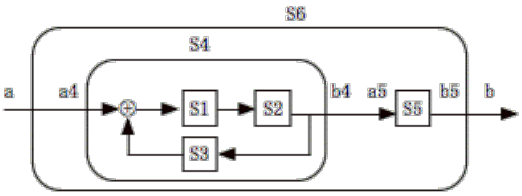
モデル理論アプローチによるシミュレーション
：**Simcast09**，開発方法論，モデル

Model Theory Approach for Simulation
: **Simcast09**, Development Method, Models

旭貴朗，高原康彦，齋藤敏雄，柴直樹
Takao Asahi, Yasuhiko Takahara, Toshio Saito, Naoki Shiba



2011 年 2 月 1 日

ふつうの数式でシステムモデルを書けば，それがそのままプログラムとなり，シミュレーションが動き出す．数学はコンピュータ言語でもある．

【用語】
実行環境　Simcast（シミュレーション開発・実行環境）
配布場所　http://www2.toyo.ac.jp/~asahi/reseach/simulation/　（2011.03.20）
配布物　　simcast.zip (バージョン 090921) + extProlog.zip (バージョン 090821)
モデル記述言語　CAST (Computer-Acceptable Set Theory)

【特徴】
３つのモデル(状態機械、Moore 型オートマトン、Solver) を混在して動かせる。
Solver を関数化できる（オートマトンに Solver を埋め込める)。
オートマトンを多層モデリングできる。-> Solver 埋込オートマトンも多層に。

序文

本書の目的は，モデル理論アプローチによるシミュレーション開発実行環境 Simcast の概要を説明し，Simcast におけるシミュレーション開発方法論を提案し，さらにその適用範囲の広さを実証することである．

高原康彦らは，モデル理論アプローチによる経営情報システム開発の方法論を提案している[1]．このアプローチにおいては，CAST と呼ばれるモデル記述言語を用いて，開発したいシステムのモデルを定式化する[2]．そのモデルを情報システム開発環境 MTA-SDK でコンパイルすると，UNIX 上で稼働する経営情報システムが出来上がる[3]．筆者が着目するモデル理論アプローチの特徴は次の２点である．１）経営情報システム(業務処理と問題解決)の一般モデルと背景理論をもつ．２）各種モデルは，数式を用いて定式化し，言語 CAST によってファイルに記述する．

第１の特徴は一般モデルと背景理論である．モデル理論アプローチでは，経営情報システムの一般モデルを構築し，理論展開を行っている．経営情報システムは業務処理（ファイル書き換え）と問題解決の組み合わせから成るシステムである．今のところ，Web ベースの業務処理システム（Transaction Processing System：TPS）の定式化と実現性理論を得ている[4]．また，問題解決システム（Solver）の定式化と解の導出理論を展開している[5, 6]．これらの理論に基づいて，システム開発環境 MTA-SDK の実装がなされている．すなわち，「モデル理論アプローチは背景理論をもっている」．一般モデルの枠組みに整合するモデルならどのような個別モデルでも動かすことが可能である．一般モデルの適用範囲の広さが重要なポイントである．

第２の特徴は，モデル記述言語 CAST の，数式との親和性である．モデル理論アプローチでは，個別に開発するシステムを数理モデルとして定式化する．ここでいう数理モデルとは，論理学におけるモデル理論の意味のモデルである．それは，台集合と関係（関数），すなわち集合と述語と関数，を具体的に指定することである．そして，それをファイルに記述するときは，数式と親和性のある，理解しやすい言語 CAST に従って記述する．例えば，集合 A に要素 b を加えて新しい集合 C を作ることは，集合論の表記では $C = A \cup \{b\}$ である．これを言語 CAST では，C := union(A,{"b"})と記述する．実際に，言語 CAST の構文は等号を含む一階述語論理であるので，数式との親和性が高い[2]．すなわち，開発したいシステムのモデルを，（頭の中では）数式で定式化し，（コンピュータ上では）数式と親和性のある言語 CAST で記述する．「モデル理論アプローチは数理的にモデル化することを強制する」と言っても良い．

しかしながら，これまでの研究では，複雑なシステムのシミュレーションに関しては触れてこなかった．また，シミュレーションモデルを動かすエンジン（ソフトウエア）もなかった．

そこで，筆者らはモデル理論アプローチによるシミュレーション開発実行環境 Simcast を開発し，公表した[7]．以下に概要を述べる．「適用システム」は，オートマトンや Solver を要素とする複合システムである．正確に言えば，Simcast が動かすことができるシステムは状態機械（state machine）やムーア型オートマトン（Moore-type automaton）と Solver である．よく知られているように，オートマトンは機械や定型的活動の一般的なモデルであることから，Simcast の適用範囲は非常に広いといえる（第 4 章，第 5 章）．すでにオー

トマトンを学習した読者は，これらの章から読み始めることができるだろう．

「使用する言語」はすでに開発されている CAST である．オートマトンモデルは状態遷移関数と出力関数の 2 本の数式を記述すればよいので，言語 CAST で記述することが可能である．これにより，ひとつの言語で業務処理システムと問題解決システムとシミュレーションの開発が可能となる．しかし，シミュレーションでは結果の分析のために「データ表示」が必要である．これに関する述語を追加したので第 2 章に掲載した．

Simcast の「背景理論」はよく知られたオートマトンの標準的な理論と問題解決の理論である．たとえば，基本的な原理として，「複数の状態機械（オートマトン）が結合したシステムは，全体として一つの状態機械（オートマトン）である」がある．これが，モデルを動かすための実行環境 Simcast の動作原理である．つまり，一つのオートマトンを動かすことができるエンジンがあれば，いかなる複合システムも動かすことができる．

また，「開発方法論」も同じ原理に従う．複数個の要素オートマトンを組み合わせて中規模の複合モデルを構成し，複数個の中規模モデルを再び組み合わせて多層的なシステムを組み立てていく．そのような「段階的なモデル化」が，同じ原理で一貫してできるようになる．その結果，複雑に結合したシステムのシミュレーションが可能となる（第 3 章）．

この報告書は，前提条件として教科書[2]の少なくとも第 3 章を読んでいれば分かるように書いた．理想的には第 3 章から第 5 章までを読んでいることを前提とする．

注

[1] Y. Takahara and Y. Liu, *Foundation and Applications of MIS：A Model Theory Approach* , Springer (2006) を参照のこと．

[2] 旭,高原,中野,斎藤,柴,竹田「経営情報システム開発のためのモデル記述言語：CAST」経営情報学会誌, Vol. 16, No. 4 (2008) pp.19-30 を参照のこと．また，次のサイトのダウンロードページの最終行から関連資料をダウンロードできる．

http://www.geocities.jp/kisoten/ （2008.08.16）

[3] 高原康彦ほか『形式手法 モデル理論アプローチ：情報システム開発の基礎』日科技連 (2007)では開発環境 MTA-SDK の操作方法と経営情報システム開発方法の基礎的な部分が詳述されている．また，第 4 章は言語 CAST の解説となっている．

[4] Y. Takahara, R. Banerji, T. Asahi and N. Shiba, "Theoretical foundation for browser-based management information system development", International Journal of General Systems, Vol. 37, No. 3, (2008) pp.275-304.

[5] 高原康彦，柴直樹，高木徹，矢野吉雄「システム論的アプローチによるソルバの設計と実装—問題解決型経営情報システムのソルバの定式化」経営情報学会誌 Vol.12, No.4 (2004)

[6] Yoshio Yano, *A Model Theory Approach to Problem Solving Systems Development*, Doctorial Dissertation, Chiba Institute of Technology (2006)

[7] Simcast はインターネットの Web サイトからダウンロード可能である．

http://www2.toyo.ac.jp/~asahi/reseach/simulation/ （2011.03.20）

目次

第 1 章 開発・実行環境 Simcast

Simcast はモデル記述言語 CAST を使ってシミュレーションをするための開発・実行環境である.

1.1 操作マニュアル

ここでは, 巻末付録 1「ダウンロードとインストール」にしたがって開発実行環境 Simcast がインストールされていることを前提に, Simcast の操作方法について述べる. 教科書[1]とは異なる部分を重点的に述べる.

システム開発手順

システムのイメージ → システムモデル → シミュレーションシステム
（モデル化） （自動生成）

Simcast は, 状態機械, Moore 型オートマトンおよび問題解決システム（Solver）のモデルをコンパイルし動かすことが可能である.

モデルの実装例

具体例として, 定価 100 円と 200 円の 2 種類の商品を販売する自動販売機を考えてみる. ただし, この機械は 100 円硬貨のみを 2 枚まで受け付け, 選択ボタンで商品選択するが, 金額が不足するときは投入金を全額返却するものとする. この自販機の Moore 型オートマトンモデル 〈A, B, C, δ , λ , c0〉 は次のようになる.

$$
\begin{aligned}
&A = \{ (a1,a2) \mid a1 \in \{0,100,200\},\ a2 \in \{100,200\} \} \\
&B = \{ (b1,b2) \mid b1 \in \{0,100,200\},\ b2 \in \{0,100\} \} \\
&C = A \\
&c0 = (0,100) \\
&\delta(c,(a1,a2)) = c' \leftrightarrow c' = (a1,a2) \\
&\lambda(c) = b \quad \leftrightarrow \\
&\qquad b = \begin{cases} (0,a1) & \text{if} \quad a1 < a2,\ (a1,a2) \in A \\ (a2,a1-a2) & \text{if} \quad a1 \geqq a2,\ (a1,a2) \in A \end{cases}
\end{aligned}
$$

図表1-1 自販機のオートマトンモデル（数式による表現）

ただし, A は入力集合で, a1 は投入金額を, a2 は選択した商品の種類を表す. また, B は出力集合で, b1 は排出された商品の種類を, b2 は釣銭を表す. C は状態集合である. c0 は初期状態を表し, δ は状態遷移関数で, λ は出力関数である. もちろん, 記号↔は必

[1] 高原康彦ほか『形式手法 モデル理論アプローチ：情報システム開発の基礎』日科技連 (2007)

要十分条件を意味する．このモデルを CAST 言語で記述すると次のようになる．

```
//automaton01.set
initialstate( )=c0 <-> c0:=[0,100];
delta(c,[a1,a2]) = c2 <-> c2:= [a1,a2];
lambda(c) = b <->
        [a1,a2]:=c,
        (a1 < a2 , p(a1,a2)) -> (b:= [0,a1]),
        (a1 >= a2 , p(a1,a2)) -> (b:= [a2,a1-a2]);
p(a1,a2) <-> member(a1,{0,100,200}), member(a2,{100,200});
inputsequence( )=c <-> c = [[100,200], [100,100]];
```

図表 1-2 言語 CAST による自販機モデルの表現

第 1 行目はコメント文で，ファイル名を記述する．第 2 行目は初期状態（定数）を無変数関数 initialstate で定義している．第 3 行目は状態遷移関数 delta を定義している．第 4 行目は出力関数 lambda を定義している. p(a1,a2)は論理学の意味の述語を定義している．(a1,a2)が入力集合 A の要素であることを意味する．ここまでがオートマトンの定義であるが，これが動くためにはなんらかの入力が必要である．最終行の inputsequence は入力系列を定義している．これは開発者が自由に決めることができる．

図表 1-2 をテキストエディタで作成し，ファイル名 automaton01.set としてディレクトリ simcast 内に保存する．同じ simcast ディレクトリで extProlog（実行ファイル xsheet）を起動し，automaton01.set を実行するとコンパイルされ，オートマトンシステム automaton01.p が作成される．また，自動的に実行される．

モデルの実装構造

個々のシステムモデル（ユーザモデルという）はユーザがテキストエディタで作成し，ディレクトリ simcast 内に保存する（拡張子は.set）．

ディレクトリ simcast 内でのオートマトンの実装構造【最小構成】

```
//filename.set
inputsequence()=As <-> As:=[...];
initialstate()=c0 <-> c0:=(...);
delta(c,a)=cc <-> cc:=(...);
```

これは出力関数 lambda のないオートマトン（つまり状態機械）のユーザモデルの構造である．1 行目はコメント文，2 行目は入力系列，3 行目は初期状態，4 行目は状態遷移関数であり，これが最小限必要なオートマトンモデルの実装構造である．上記の(...)のところに言語 CAST で式を記述し，それぞれの関数を定義する．言語 CAST そのものの解説は教科書[1]第 4 章または MTA サイト[2]「ダウンロード」ページの最終行から入手できる．

[2] MTA サイト http://www.geocities.jp/kisoten/ （2008.08.16）

もしも必要なら，次のようにオプションを加えることが可能である【最大構成】

```
//filename.set
func([...]);
ユーザ定義関数の定義;
preprocess() <-> (...);
inputsequence()=As <-> As:=[...];
times()=数値;
initialstate()=c0 <-> c0:=(...);
delta(c,a)=cc <-> cc:=(...);
lambda(c)=b <-> b:=(...);
ユーザ定義述語の定義;
postprocess() <-> (...);
```

それぞれ，func はユーザ定義関数の宣言，times は繰り返し入力の回数，lambda は出力関数を表わしている．出力関数 lambda の引数はひとつだけである．preprocess と postprocess は事前処理と事後処理を述語で定義する．これら 5 つは必要なときだけ記述すればよい．この順序で書く必要はないが，「できるだけ，この順序で記述すること」を推奨したい．また，ユーザ定義の関数や述語はいくつでも加えることが可能である．

さらに，複数のオートマトンが結合したシステムについては「5.1 基本：複合システムの分類とモデル」を参照のこと．ここでは述べていない問題解決システムの作成方法については教科書[1]第 5 章を参照のこと．

関数 times( )を使ったモデル例

Simcast09 では新しい述語 times( )が利用可能である．まずは従来のように time( )を利用しない場合を考える．例えば，

```
//simple1.set
inputsequence( )=[1,1,1,1,1,1];
initialstate( )=c0 <-> c0:=0;
delta(c,a)=cc <-> cc:=c+a;
```

を実行すると，設定された入力列[1,1,1,1,1,1]のとおりに，a=1 が 6 回入力され，状態が 0 から 1 ずつ加算されていき，6 回の処理を繰り返し，最終的には c=6 に達する．すべての変化は Dialog ウインドウに表示される（下図 1-6）．つまり times( )が定義されてなければ教科書[1]どおりに処理される．

一方，新しい述語 times( )を利用する場合は，例えば，

```
//simple2.set
inputsequence( )=1;
times( )=500;
initialstate( )=c0 <-> c0:=0;
delta(c,a)=cc <-> cc:=c+a;
```

を実行すると，a=1 が 500 回入力され，状態が 0 から 500 まで遷移する．つまり，「もし

も times()が定義されていれば，inputsequence()の右辺の内容そのままを，times()で定義された回数だけ」繰り返して入力する．その他，本書にはいろいろなモデル例があるので，参照されたい．

実行方法

1）extProlog 起動後の操作方法は，基本的に教科書[1]と同じである．
　ディレクトリ simcast の中で，上記のようなシミュレーションモデルを作成し保存する（拡張子は.set）．extProlog の実行ファイル xsheet を起動する（図 1-3）と開発実行環境になるので，Dialog ウインドウにモデルのファイル名を入力する（図 1-4）．するとモデルがコンパイルされ実行される（図 1-5, 1-6）．作業はすべてディレクトリ simcast で行なうこと．
2）出力表示は，最後に一括表示ではなく，実行中にそのときの状態や出力などが表示される．また Simcast09 からは，教科書[1]とは異なり，MTABLE ウインドウを表示しないことにした．
3）操作上のヒント
　最新 9 個のユーザ入力は記憶されている．矢印キー「↑」「↓」で入力ライン上に過去の入力が現れる．直前に入力したファイル名を再表示し，それを修正し実行することなどができるので，便利になったと思われる．

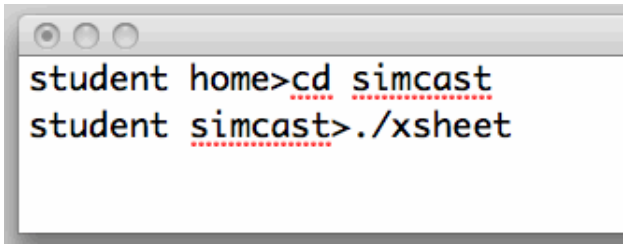


図 1-3　起動
Unix のターミナルウインドウを開き，
simcast ディレクトリに移動し，xsheet を起動する．

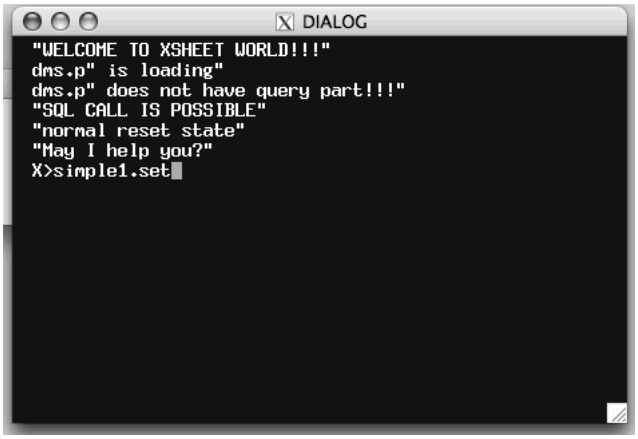


図 1-4　ファイル名入力
Dialog ウインドウにモデルのファイル名(simple1.set)を入力する．

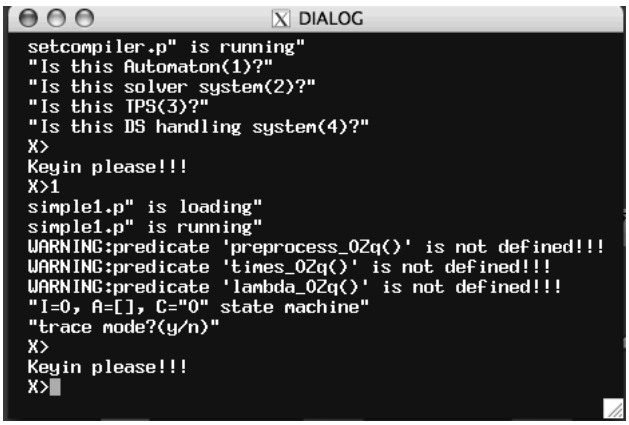


図 1-5　コンパイル

まずモデルの種類を聞かれる．コンパイルに成功すれば，
システム(simple1.p)が作成されるので，これを実行しようとする．

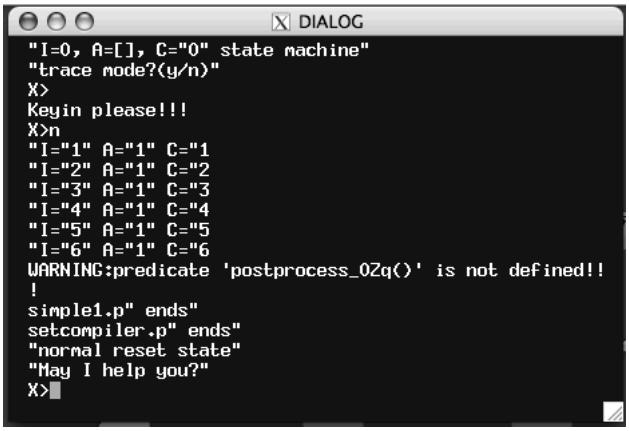


図 1-6　システム実行

システム実行開始時にトレースするかを聞かれるので，応じると
システム実行途中の様子（入力 A，状態 C，もしあれば出力 B）が表示される．
トレースしない（n）と応じた場合は，最後まで状態遷移が繰り返されて終了する．
トレースする（y）と応じた場合は，状態遷移 1 回ごとに停止するので，
Unix ターミナルウインドウ(図 1-3)をクリックしてからリターンキーを押すと，次の状態遷移に移る．

1.2 事前処理と事後処理

新しいウインドウを開いたり，多数のグローバル変数を定義したり，シミュレーションの事前準備としての初期設定が複雑なときは，事前処理を述語 preprocess にまとめて書いておくことができる．また，シミュレーション終了後の処理は postprocess に書ける．

【書式】

```
[1] preprocess() <-> (...);      (...)に事前処理の述語を記述する
[2] postprocess() <-> (...);     (...)に事後処理の述語を記述する
```

【モデル例】

```
//testpreprocess.set
preprocess() <->
       M.g:=10,
       getWp("text",Wp1),  //text ウインドウを開く
       (Wp1>0)->(Wp:=Wp1) otherwise (wopen(Wp,"text")), Wp.g:=Wp,
       xwriteln(Wp.g,"--- start ---"); //text ウインドウに書く
inputsequence()=As <-> As:=genIndex(1,M.g);
initialstate()=c <-> c:=0;
delta(c,a)=cc <-> cc:=c+a, xwriteln(Wp.g,cc);
postprocess() <-> xwriteln(Wp.g,"--- end ---");
```

【動作】

処理前に, preprocess( )でテキストウインドウを開き, Wp.g にウインドウ番号を保持する．
処理中は，テキストウインドウ Wp.g に各時点の状態 cc を表示する．
処理終了時に，postprocess( )でテキスト"--- end ---"をテキストウインドウに表示する．

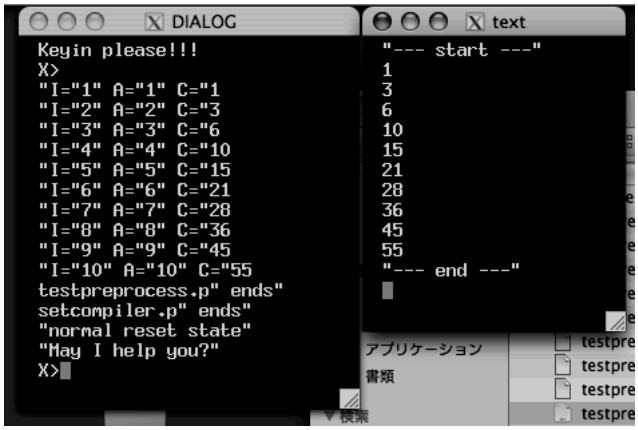


図 1-7　事前処理と事後処理

【注意】

preprocess( )のなかで，グローバル変数を記述するときは，「:=」を使うこと．

第 2 章 言語 CAST の拡張

システムシミュレーションを行なうには，まず対象システムのモデルを作成しなければならない．通常のシステムモデルは数式を使って定義されているので，その数理的思考の流れをアルゴリズム思考に転換することなく，それらの数式をできるだけそのままの形でコーディングできることが望まれる．つまり，ふつうの数式を書くようにシステムモデルを書けば，それがそのままプログラムとなり，シミュレーションが動き出すようにしたいのである．CAST はそのために生まれた言語である [1]．本節では，すでにある言語 CAST を拡張し，シミュレーション向けに新たに追加した関数や述語やそれらの使い方について述べる．

2.1 言語 CAST の概要

CAST (Computer Acceptable Set Theory) はシステムの数理モデルを記述するための言語である．通常の数理モデルは数式を使って定義される．数式は論理学では well formed formula（以下 wff と書く）と呼ばれる．CAST における wff とは，数値，文字列，集合，述語，関数を正しく構成した数式のことである．通常の加減乗除を使う算術式、あるいは、∩、∪、⊂、∈ などを使う集合の演算式は wff である。対応する記号 +, -, *, /, union, intersection, subset, member は予約語としてあらかじめ開発環境に用意されている．例えば，和集合 A∪{b}を，言語 CAST では union(A,{"b"})と書く．これは要素式(atomic formula)であり，wff である．ただし，簡単のために，等号を表す述語は，=(a,b)でなく a=b と記述し，代入も assign(x,1)でなく x:=1 と記述する．また，これらを使う演算式も wff である．従って数値，文字列，集合，述語，関数を正しく構成したものが wff である．

特に注意すべきは，順序対(x,y,z)や集合{x,y,z}である．これらはすべて内部的にリスト[x,y,z]に変換し処理している．例えば，第 1 章図表 1-2 の member(a2,{100,200})は member(a2,[100,200])と書いても同じである．正確に言えば CAST はリスト処理言語である．しかし，リスト[x,y,z]は 3 項対でもあり，3 要素の集合でもあると解釈でき，場合に応じて異なる処理ができる．リストや「リストのリスト」を集合や行列と見なせることができて処理できるので，強引ではあるが『CAST は「集合」処理言語である』と言っている．

CAST におけるモデルとは，モデル理論の意味の台集合と関係，すなわち集合と述語と関数を具体的に指定した（記述した）ものである．集合の定義，述語の定義，関数の定義は wff だから，形式的には，モデルは上記 1 階述語論理の wff の有限個の集まりである．第 1 章図表 1-2 のユーザモデル　< initialstate, delta, lambda, inputsequence > は 5 個の wff の集まりである（コメント文を除く）.

直感的に言えば, CAST は等号と有理数を含む 1 階述語論理の「限定的な」言語である。素朴集合論および代数学の範囲での高校数学の記述方法に近いと言える。これが CAST (Computer Acceptable Set Theory ：コンピュータ可読集合論) という名称の由来である。ただし、無限集合と限定子(∀、∃)を表わす記号はない．しかし，有限個の範囲なら，述語によって集合を生成する関数 defSet によって表現できるので回避できる. 詳細は文献[1]を参照されたい.

[1] 高原康彦ほか『形式手法 モデル理論アプローチ：情報システム開発の基礎』日科技連（2007）

2.2 追加した関数と述語

数値演算(+, -, *, /, %)や集合演算（∩、∪, ×）など通常の簡単な関数は, すでに MTA-SDK に組込み済みであった．それを拡張し，教科書未記載のものや Simcast09 で新たに使えるようになった関数や述語を記載する．

組込み関数

平方根は sqrt(m)，m の n 乗は pow(m,n) 指数関数 exp(x)と対数関数 log(x) 整数部分を抜き出す関数は floor(x) 三角関数 sin(θ), cos(θ), tan(θ)

【更新】　一様分布の乱数を発生させる myrandom(r)

説明（これだけは特殊である）
　myrandom(r)で，変数 r に 0 以上 1 未満の数がランダムに代入される．一様分布になる．

使い方
　小数点以下を切り捨てる関数 floor とかけ算を使うと，たとえば,
　　myrandom(r),
　　x:=floor(7*r)+1,
で，変数 x に 1 から 7 までの整数がランダムに代入される．

###　関数にしたければ，ユーザ関数 rand を作ればよい．
func([rand]);
rand()=r <-> myrandom(r);
こうしておけば，r:=rand()で r に乱数が代入される．

組込み述語

また，数値比較（<, >, =,≠）や集合の包含関係（⊂,∈）など通常の簡単な述語は，すでに MTA-SDK に組込み済みであった．今回は新たに，ファイル操作とグラフやスプレッドシートによるデータ表示に必要な述語を確認した．

xread("temp.lib", [A])	ファイル temp.lib の内容を変数 A に代入する．
xread("temp.lib", K, L, [X])	ファイルの K 行から L 行までを変数 X に代入する．
appendf("temp.lib","a", [Y])	ファイルの最終行にデータ Y を追加する．
appendf("temp.lib","w", [Y])	ファイルの内容全部をデータ Y に置き換える．
cardinalityf("temp.lib",N)	ファイルの行数を N に代入する

リスト処理関数

行列の操作のために，次の関数を追加した．

【追加】　部分行列を抜き出す projectMatrix
書式　N:=projectMatrix(M,x1,y1,x2,y2) 意味　行列 M のセル(x1,y1)からセル(x2,y2)までの長方形の部分行列 N を作成する． 例　　N:=projectMatrix([[1,2,3],[4,5,6],[7,8,9]],2,2,3,3) のとき，N=[[5,6],[8,9]] となる．

【追加】　行列の値を変更する replaceMatrix
書式　N:=replaceMatrix(M,[[x1,y1,v1],[x2,y2,v2],...]) 意味　行列 M の xi 行 yi 列の値を vi に変更した行列 N を作成する． 例　　N:=replaceMatrix[[1,1,1],[2,2,2],[3,3,3]],[[1,2,3],[2,3,4],[3,1,7]] のとき， N=[[1,3,1],[2,2,4],[7,3,3]] となる．

範囲指定は project(x, ["r", m, n])

m 番目から n 番目まで範囲(range)を指定して射影をとるのは, project(x, ["r",m,n])である．"r"が必要である．したがって，教科書の記述は誤りである．

例："r"があると，m 番目から n 番目まで抽出される．
　　y:=project([5,6,7,8,9],["r",2,4]) とすると，y=[6,7,8]となる．
例："r"がないときは，m 番目と n 番目が抽出される．
　　y:=project([5,6,7,8,9],[2,4]) とすると，y=[6,8]となる

その他，グラフウインドウやスプレッドシートを開き，データを表示するための新しい述語を追加しているが，節を改めて述べる（第 2.5 節から第 2.9 節まで）．

2.3 defList による漸化式の求解

リストを作る関数 defList を利用して漸化式を解くことができる．例題として，漸化式
　　f(0)=100,
　　f(n)=0.9×f(n-1)　(n≧1)
で定義される数列 f(1),f(2),f(3),... を求めるための CAST 言語のユーザ関数を示す．

1）初等再帰的関数

教科書[1] p.161 の初等再帰的関数として定義すると，

```
func([f]);
f(n)=y <-> (n=0) -> (y:=100) otherwise ( y:=0.9*f(n-1) );
```

となる．実は，同じ関数を次のようにも記述できる．

```
func([f]);
f(0)=100;
f(n)=0.9*f(n-1);
```

数式をそのまま書いているので分かりやすい．しかし，これで求められるのは f(28)までである．n > 28 に対してはシステムがフリーズする（Segmentation fault）．原因は不明である．

2）関数 defList による解法

関数 defList を使って数列のリスト L(n) = [f(1), f(2),..., f(n)]を求める方法を示す．

```
func([L]);
L(n)=Ps <-> y.g:=100, Ps:=defList(p(z,x,[]), member(genIndex(1,n)));
p(z,x,[]) <-> z:=0.9*y.g, y.g:=z;
```

したがって，リストの最後の要素をとりだせば f(n) が得られる．

```
fn:=project(L(n),0);
```

これなら，n が大きくても計算できる．f(10000)が計算できることを確認した．
文献 1 の非再帰的な方法がこれである．

【解説】defList ----------

defList(p(z,x,[補助変数]),member(x,Xs)) は，述語 p を満足する z を並べてリストを作成する関数である．Xs をインデックス集合と呼ぶ．述語 p(z,x,[補助変数])はインデックス集合の要素 x と並べるべき要素 z を関係づける述語である．たとえば，文献 1 の p. 156 のように，

```
Ps:= defList(peven(z,x,[]),member(x,[1,2,3,4,5,6]))
peven(z,x,[]) <-> x%2=0, z:=x;
```

と定義すれば，偶数のリスト Ps=[2,4,6]が得られる．システム内部では，

peven(z,1,[])を満足する z ＝ [],
peven(z,2,[])を満足する z ＝ [2],
peven(z,3,[])を満足する z ＝ [],
peven(z,4,[])を満足する z ＝ [4],
peven(z,5,[])を満足する z ＝ [],
peven(z,6,[])を満足する z ＝ [6],

を連接(append)してリスト Ps=[2,4,6]を得ている．集合 Xs の要素 x を「先頭から順に代入して」性質 peven が成り立つ z を求めているのである．

以下では，教科書には未記載の「裏技」を紹介する．ここで注目したいことは，「defLIst はインデックス集合の要素の数｜Xs｜だけ仕事をする」という点である．したがって，極端に言えば, x と z が無関係であっても｜Xs｜回の仕事をさせることができる. たとえば,

```
Qs:=defList(p1(z,x,[]),member(x,[6,7,8]))
p1(z,x,[]) <-> xwriteln(0,"Hallo");
```

と定義した場合, Dialog ウインドウに"Hallo"が 3 回表示される. 述語 xwriteln(0,"Hallo") は, x=6,7,8 とまったく無関係であるうえに, 変数 z の定義さえしていない. しかしながら, 3 回の仕事「真偽チェック」をしているのである. (xwriteln(0,"Hallo")のような入出力命令は, それが成功する限り常に「真」である.) したがって, 手続型言語の「繰り返し命令」のような仕事をさせることができる.

その他, 裏技としてグローバル変数を使う例を紹介する. 例えば,

```
y.g:=100,
Qs:=defList(p1(z,x,[]),member(x,[6,7,8]))
p1(z,x,[]) <-> z:=0.9*y.g, y.g:=z;
```

と定義した場合, z は y.g のみに依存して決定されるので, x=6,7,8 と z は互いに無関係である. しかしながら, 3 回の真偽チェックを行ない,

初期値は y.g=100,
p1(z,6,[])を満足する z ＝ [90], そのとき y.g=90,
p1(z,7,[])を満足する z ＝ [81], そのとき y.g=81,
p1(z,8,[])を満足する z ＝ [72.9]

であるから, Qs=[90,81,72.9]が得られる.

述語 p1 は並べるべき値 z:=0.9*y.g を決定し, ついでにグローバル変数を y.g:=z に書き換える仕事をしている. したがって,

f(0)=100,
f(n)=0.9*f(n-1)

という漸化式を初期値から順番に計算し, リスト Qs=[f(1),f(2),f(3)]を作成していると考えることができる.

さらに, もしも Rs=[f(11),f(12),f(13)] を計算したいときは, 次のように定義すればよい.

```
y.g:=100,
Rs:=defList(p2(z,x,[]),member(x,genIndex(1,13)))
p2(z,x,[]) <-> z:=0.9*y.g, y.g:=z, x>10;
```

述語 p2 ではインデックスを使って x>10 と指定しているので,
11 回目の計算結果から Rs の要素に採用されていく.
しかし, 10 回目まで計算は行なわれていることに注意したい.
10 回目まで「計算は行なわれている」が Rs の要素として採用されないことに特徴がある.
###
ただし x>10 は右端に記述する必要がある.
例えば x>10, z:=0.9*y.g, y.g:=z と書くと, 10 回までの計算は実行されない.
その理由は, システムは, 偽である x>10 を見つけると, それより右の項の真偽チェックをしないからである.
記述の順序に気をつける必要がある.
###

その他,
defList の結果にリスト演算子 sum を適用すると, 数列の和 (シグマ : Σ) を計算できる.

defList を 2 つ使って，行列（リストのリスト）を作成することもできる．
　また，リストではなく集合を作る関数として defSet があるが，これは defList を行なったあとに「集合」化したもの，つまり，defSet(...) = distinct(defList(...))と考えることができる．

2.4 ファイル操作

2.4.1　基本

　temp.lib:=["a","b","c"] はファイル temp.lib に値 ["a","b","c"] を書込め（上書きせよ）という述語である．
1）ファイルの中では，"a"改行"b"改行"c"改行となっているので，
このファイルをテキストエディタで開くと，
　　"a"
　　"b"
　　"c"
と表示される．
2）逆に，A:=temp.lib とすると，A=["a","b","c"] に戻る．
3）project(temp.lib,3)="c" となる．
4）教科書未記載だが，
　　述語 xread("temp.lib", [A]) で，temp.lib の内容が変数 A に代入されます．
　　この場合は，A:=xread("temp.lib")と書けない．

2.4.2　ファイルを空にできない．
　temp.lib:=[ ]; とすると，ファイル temp.lib に"nil"が代入される．
　今の所，これが仕様である．ファイルを空にするには，テキストエディタで手作業で内容を削除するか，または OS に戻って，ファイルを削除し，新規にファイルを作成する方法しかない．

2.4.3　読んでから書くか，書いてから読むか．

1）教科書[1]の p.274 の getOrderId() では，
「ファイルを読み込んで処理して書き込む」ことをしている．書いて読むのでなく，読んで書いているだけなので成功するのである．

2）一方，書いた直後の「読み」は無効となる．
　　temp.lib:=[1,2,3], N:=cardinality(temp.lib),
のとき，N=3 のはずだが，実際にはならない．その理由は書き込み速度が遅いからかも知れないが，実際のところ分からない．上記の例では，
　　P:=[1,2,3], N:=cardinality(P), temp.lib:=P,
とすれば回避できる．
###
　書いた直後にそれを読むことは，実際にはほとんどありえないレアケースである．にもかかわらず，無理にそれをしたいときは，ワンテンポ遅らせて，状態遷移後に読めば正しく N=3 になる．または，資料「Solver の関数化」の out のように，別途関数を作れば成

功する．
　　func([out]);
　　out("temp.lib")=N <-> N:=cardinality(temp.lib);
と定義しておいて，モデルのなかで，
　　temp.lib:=[1,2,3], N:=out("temp.lib"),
とすれば成功するが推奨しない．

2.4.4　巨大なファイルは「教科書のやり方」では扱えない．
　例えば，ファイル temp.lib の内容が 1 万行を超えると，
　　B:=temp.lib や
　　Bs.g:=temp.lib は無効であり．
　　N:=cardinality(temp.lib)は無効である．
ファイルサイズが大きすぎると変数に代入できないからである．しかし，代入しないで直接的にファイルを触ることはできる．次の 5 や 6 を参照せよ．

2.4.5　1 万行以上のファイル temp.lib でも，次の述語が有効です．
xread("temp.lib", K, L, [X]）ファイルの K 行から L 行までを変数 X に代入する．
appendf("temp.lib","a", [Y]) ファイルの最終行にデータ Y を追加する．
appendf("temp.lib","w", [Y]) ファイルの内容全部をデータ Y に置き換える．
変数に[ ]がついていることに注意すること．

2.4.6　1 万行以上のファイル temp.lib に対しては，行数を数える述語
　　cardinalityf("temp.lib",N)
を用意した．この形で N に temp.lib の行数が代入される．しかし 10 万行のとき 20 秒以上かかり，処理が遅い．どうしても必要なら使用可能である．また simcast.zip の中の stdAutomatonEngine.p に直接追加しているので，コードを修正することは可能である．

2.5 グラフによるデータ表示

モデル理論アプローチによるシミュレーション開発実行環境 Simcast において，簡便にグラフ表示を行なうことができる述語 show1, show2, show3 の使用方法について述べる．これらを show 系の述語と呼ぶことにする．（show 系以外にも，座標軸を表示しない wopen 系のグラフ表示の方法があるが，それについては次節 2.6 を参照されたい.）

show 系の述語は次の 3 つの特徴を持っている．
1）グラフウインドウを開くための事前操作が不要である．
2）グラフの種類に合わせた座標軸が表示される．
3）数値の大小に応じて座標軸のスケール（刻み幅）が変化する．
スケールの変化については体験してもらいたい.

2.5.1 show 系のデータ表示述語

show1 と show2 の述語の代表的な書式は次のようになる（show3 については 2.5.6 を参照せよ）.

【書式】
(1) show2(番号,データ，グラフの種類)
(2) show1(データ，グラフの種類)

【例】
(1) show2(3,[10, 20,30],"plot")
(2) show1([10,20,30],"plot")

【意味】
(1) out_3.g という名前のグラフウインドウを開き，座標平面上に３点(0,10),(1,20),(2,30)を結んだ折れ線を描く．
(2) out_0.g という名前のグラフウインドウを開き，折れ線を描く．show2(0, [10,20,30],"plot") と同じ.

show2 は複数のグラフウインドウを番号つきの名称（out_番号.g）で区別しているが，show1 ではひとつのグラフウインドウ out_0.g だけを使う．
＃＃＃
ウインドウの種類は，グラフ，スプレッドシート，テキストがある．すべてのウインドウはウインドウ番号 Wp とウインドウ名称 Wname をもつ．show 系のグラフ表示の場合にはウインドウ名称（out_n.g）で区別する．（一方，wopen 系のグラフ表示の場合は，ウインドウ番号で区別する.）末尾が.g ではあるがグローバル変数とは無関係である．一方，スプレッドシートの場合はウインドウ番号で区別する．
＃＃＃
以下では，各種のグラフを描くオートマトンモデルを例示する．

2.5.2 plot グラフ（折れ線）

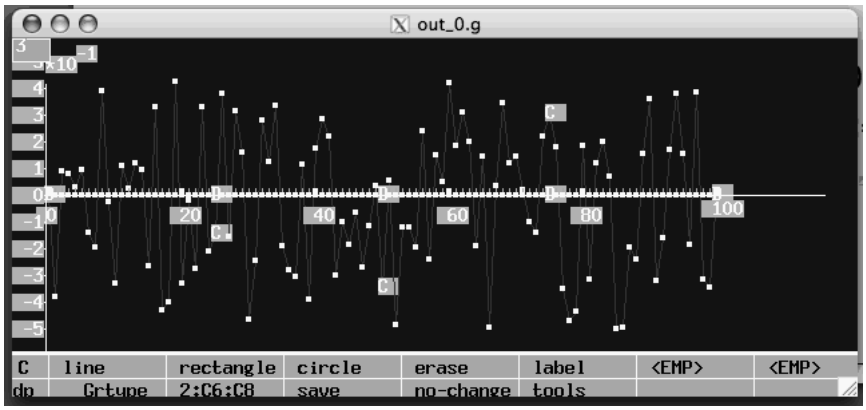
座標平面に点を打って，直線でつないでゆく（折れ線を描く）.

例

show1([20,30,50], "plot"),　　// ３点(0,20),(1,30),(2,50)を結ぶ折れ線をひとつ描く．
show1([[1,2,3],[5,3,4]], "plot"),　// ２つの折れ線を描く．

演習問題

乱数を 100 個発生させて，折れ線を作れ．Hint：オートマトンの状態遷移のときに乱数を発生させ，それを次の状態とせよ．出力関数にグラフ表示を定義せよ．ダミー入力を数 100 回入力せよ．



図表 2-5-1　乱数をグラフ表示する

ヒント：関数 myrandom( )で 0 以上 1 以下の乱数を発生させることができる．次のモデルは Dialog ウインドウに乱数を数値として表示していくものである．

```
inputsequence()="d";
times()=100;
initialstate()=c0 <-> c0:=0;
delta(c,a)=cc <-> cc:= myrandom()-0.5;
```

このモデルでは，状態 c は -0.5 から +0.5 までの間の実数となる．演習問題の主旨は，状態 c の列を蓄積しながらグラフを表示せよ，ということである．

（答）

```
//show01.set
inputsequence()="d";
times()=100;
initialstate()=c0 <-> c0:=0, hist.g:=[];
delta(c,a)=cc <-> cc:= myrandom()-0.5;
lambda(c)=b <-> b:=c,
        hist:=hist.g,
        hist1:=append(hist,[c]),
        show1(hist1,"plot"),
        hist.g:=hist1;
```

図表 2-5-2　乱数をグラフ表示するユーザモデル

ここでは，初期状態 initialstate の設定のときに，グローバル変数 hist.g を用意しておき，そこに状態の履歴を保存するようにした．また，出力関数 lambda がグラフ表示を行なうことにした．発生した乱数 c は hist1:=append(hist,[c])で変数 hist1 に蓄積され，show1 で表示

されている．結果として，履歴は hist.g:=[0, 0.32, -0.41, ...]のようなリストになっていく．それを show1 でグラフ表示している．強制的に時間を進めるため，inputsequence()="d"; times()=100; で，（状態遷移に影響しない）ダミーの"d"を 100 回入力している．

しかし，入力回数を times()=1000 に変更したい（1000 回処理を行ないたい）場合は，ひとつの画面に多数の点を表示すると，見て理解することが困難になるので，適当に削りたい．

演習問題

乱数を 1000 個発生させて，折れ線を作れ．ただし，常に最新の 200 データだけを表示せよ．Hint：リスト z の i 番目から j 番目までを抽出する述語 project(z,["r", i, j])を使え，

（答）

```
//show01a.set
inputsequence()="d";
times()=1000;
initialstate()=c0 <-> c0:=0, hist.g:=[];
delta(c,a)=cc <-> cc:=myrandom()-0.5;
lambda(c)=b <-> b:=c,
        hist:=hist.g,
        hist1:=append(hist,[c]),
        len:=cardinality(hist1),
        hist2:=project(hist1,["r",len-199,len]),
        show1(hist2,"plot"),
        hist.g:=hist2;
```

図 2-5-3

上記モデルでは，len:=cardinality(hist1)で履歴の長さを求め，project(hist1,["r",len-199,len])で，最新 200 回分のデータを抽出している．len-199＜0 のときは，0 から len までの要素が抽出されるので問題ない．

演習問題

関数 sin(θ)のグラフを描け．ただし，刻み幅は $2\pi/100$ ラジアンとする．また，常に最新の 200 データを表示せよ．

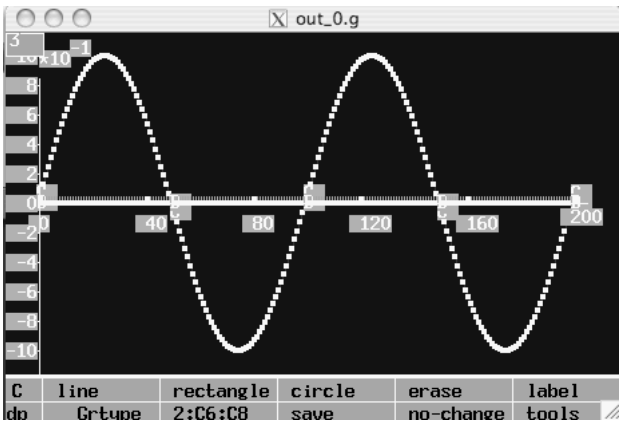


図 2-5-4

（答）　数値として sin(θ)を計算するだけなら，次のようになる．

```
//show04.set
inputsequence()="d";
times()=1000;
initialstate()=c0 <-> c0:=0;
delta(c,a)=cc<-> cc:=c+2*.pi/100;
lambda(c)=b <-> b:=sin(c);
```

図 2-5-5

このモデルでは，状態 c は角度 θ に相当する．状態遷移関数 delta は変化していく θ の数値を求め，出力関数 lambda の中の sin(c)で，sin(θ)をグラフ表示している．ただし，.pi は定項で，.pi = 3.14159284 である．

図形として描画するには，ウインドウを開き，そこにデータを表示する必要がある．ここでは述語 show1 を利用する．

```
//show05.set
inputsequence()="d";
times()=1000;
initialstate()=c0 <-> c0:=0, hist.g:=[];
delta(c,a)=cc<-> cc:=c+2*.pi/100;
lambda(c)=b <-> b:= sin(c),
        hist:=hist.g,
        hist1:=append(hist,[b]),
        len:=cardinality(hist1),
        hist2:=project(hist1,["r",len-199,len]),
        show1(hist2,"plot"),
        hist.g:=hist2;
```

図 2-5-6

例えば，述語 show1(hist2,"plot")を使わず，show2(0,hist2,"bar"), show2(1,hist2,"plot")に変更すると，２つのグラフウインドウが開き，同時に２つのグラフが変化していく（読者はそれを試みよ）．

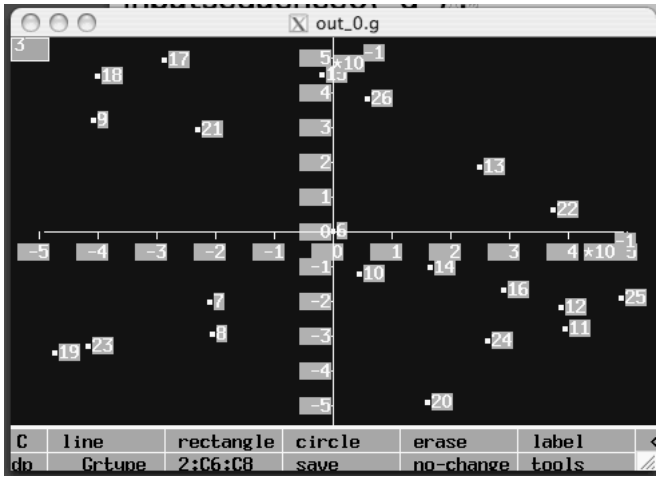
2.5.３　xyplot グラフ

xy 平面座標の上に，点(x, y)を打ってゆく．

書式　show1([xList,yList], "xyplot")　　//データ=[x 座標のリスト, y 座標のリスト]
例　　show1([[3,4,5],[10,20,30]], "xyplot")　　//３点(3,10), (4,20), (5,30)を打つ．

演習問題

２つづつ乱数を発生させて，平面上に 1000 個の点(c1,c2)を打て．ただし常に最新の 200 個を表示せよ．



（答）

```
//show02.set
inputsequence()="d";
times()=1000;
initialstate()=c0 <-> c0:=[0.5,-0.5], hist.g:=[];
delta([x,y],a)=[xx,yy] <-> [xx,yy]:= [myrandom()-0.5, myrandom()-0.5];
lambda(c)=b <-> b:=c,
        hist:=hist.g,
        hist1:=append(hist,[c]),
        len:=cardinality(hist1),
        hist2:=project(hist1,["r",len-199,len]),
        Ds:=transpose(hist2),
        show1(Ds,"xyplot"),
        hist.g:=hist2;
```

上記モデルの状態遷移関数 delta の定義は 1 行で書かれており，2 変数[xx, yy]に乱数を「同時に」代入している．2 行で書くなら，

```
        x2:= myrandom()-0.5,
        y2:= myrandom()-0.5;
```

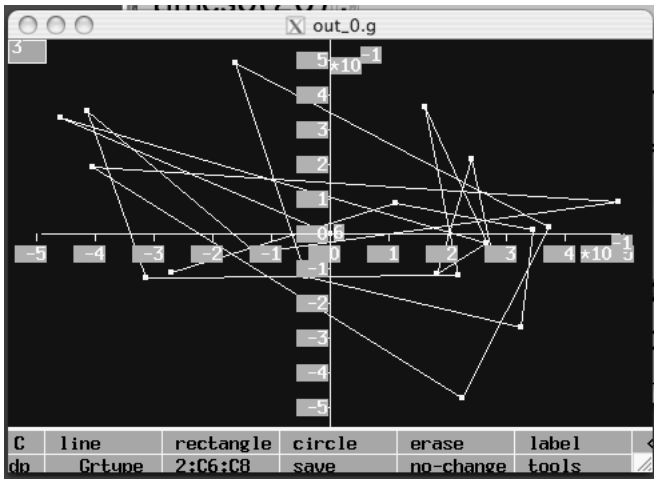
となり，同じ意味である．

出力関数 lambda でグラフ表示を行なうが，履歴は hist.g=[[x1,y1],[x2,y2],[x3,y3],.....]という形で蓄積されていく．ここでは，**リストのリストが「行列」の表現でもある**ことを考慮し，転置行列を求める関数 tranpose を使って Ds = [[x1,x2,x3,...], [y1,y2,y3,...]] の形に変換している．本稿では，この方法を推奨する．

2.5.４　trajectory グラフ

show1(データ,"trajectory")とすると，xyplot した各点がつながれてゆく.

例　show1([[3,4,5],[10,20,30]], "trajectory"),　　// ３点(3,10), (4,20), (5,30)を結ぶ.



（答）

```
//show02c.set
inputsequence()="d";
times()=1000;
initialstate()=c0 <-> c0:=[0.5,-0.5], hist.g:=[];
delta([x,y],a)=[xx,yy] <-> [xx,yy]:= [myrandom()-0.5, myrandom()-0.5];
lambda(c)=b <-> b:=c,
        hist:=hist.g,
        hist1:=append(hist,[c]),
        len:=cardinality(hist1),
        hist2:=project(hist1,["r",len-199,len]),
        Ds:=transpose(hist2),
        show1(Ds,"trajectory"),
        hist.g:=hist2;
```

2.5.6 その他のグラフ

グラフの種類には，円グラフ（"pie"），棒グラフ（"bar"），レーダグラフ（"radar"），折れ線グラフ（"plot"），平面座標上の点（"xyplot"），軌跡の表示（"trajectory"）がある.

円グラフのデータは，総和が１であること，また，レーダグラフのデータは正の数でなくてはならない．一つのウインドウに複数の折れ線（"plot"）を描けるが，線の数に限界がある.

1000 回くらいオートマトンが状態遷移を行なうと，パソコンの CPU が発熱し，換気扇が回り始める．ウインドウの数が増えると処理速度は遅くなる．グラフ表示の処理負荷は相当に高い.

2.5.6　色付き xyplot グラフ(show3 の使用方法)

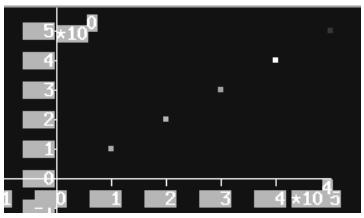
まだ制限が多く，洗練されていないが，座標(i,j)に色付きの点を描くことができる．色は数値 v で５種類を指定することができる．

【書式】show3(n, [IList, JList, VList], "xyplot","color")

ただし，n はウインドウ名称（out_n.g）を指定する自然数である．また，IList と JList はそれぞれ自然数を要素とするリストであり，VList は（色を指定するための）0 以上の実数を要素とするリストである．３つのリスト IList, JList, VList は，要素数が同じでなければならない．色指定は５種類あり，1 (orange), 2 (green), 3 (purple), 4 (white), 5 (red)である．これ以上の数値に対しても同じ順序で繰り返される．したがって, v=6 のときは oranege で, v=10 のときは red である．特別に v=0 だけは無視される（あるいは背景色と同じになる）ことに注意する．この述語 show3 は，既存の show1(L,Gr,"color")を筆者が修正したものである．

【例１】

```
//showColor00.set
inputsequence()=As <-> As:=[0];
initialstate()=c <-> c:=0;
delta(c,a)=cc <-> cc:=c,
        Data:=[[1,1,1],[2,2,2],[3,3,3],[4,4,4],[5,5,5]],
        Ds:=transpose(Data),
        show3(2,Ds,"xyplot","color");
```



【意味】out_2.g という名前のグラフウインドウを開き，座標平面上に５点(1,1), (2,2), (3,3), (4,4), (5,5)を描き，それぞれの点に数値 1, 2, 3, 4, 5 に対応する色をつける．

この例では，観測データ Data:=[[1,1,1],[2,2,2],[3,3,3],[4,4,4],[5,5,5]]をグラフ表示するために，まず転置（transpose）をしている．したがって，Ds=[IList, JList, VList]は，実際には，IList=[1,2,3,4,5], JList=[1,2,3,4,5], VList=[1,2,3,4,5] となっている．[i,j,v]の形をしたデータのリストは，転置をしてから show3 に代入することがポイントである．

【例２】

```
//showColor08.set
Ps.g=product(genIndex(1, 30),genIndex(1, 30));
inputsequence()=As <-> As:=genIndex(1,100);
initialstate()=c <-> c:=[[3,3,3]];
delta(c,a)=cc <-> cc:=defSet( p(z,[i,j],[a]),member([i,j],Ps.g) );
p(z,[i,j],[a]) <-> z:=[i, j, a+i];
lambda(c)=b <->    b:=project(c,0),
                   L:=transpose(c), show3(2,L,"xyplot","color");
```

この例では，まず座標の範囲 Ps.g={(i,j)｜i∈{1,2,...,30}, j∈{1,2,...,30}}を指定している．つまり一度に 30×30 = 900 個の点に対して色をつけることができる．入力 a が引き起こす次状態は述語 defSet で定義されており, cc={(i,j,v)｜v = a+i , (i,j)∈Ps.g}である．したがって，点 (i,j) の色は v=a+i となる．入力は a=1,2,3,...,100 と変化するので，グラフは動的に変化していき，右から左へ縞模様が流れてゆくことになる．

【制限】　表示範囲については，Mac OS X の場合，30×30 = 900 個までは正常に表示された．これ以上になると，縦方向に関して間引き現象が起こる．40×40 のときは開発環境がダウンした．また，常に縦軸は j を示さず，v の最大値を示すので，縦軸の目盛は無視せざるを得ない．なぜか「ディレクトリ usr/local/oopara を作成せよ」との警告が出たことを記しておく．これらの制限の原因は究明していない．

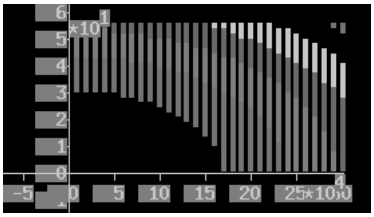
【演習】

　上の例２の述語 p を変更し，

p(v,[i,j],[a]) <-> v:= [i,j,a/2 +(i*i+j*j)*7/2000];

にすると，右のような図ができ，動的に変化していく．

やってみよう．



2.6 座標軸の非表示と多数データの描画

前節のグラフ表示述語 show1()や show2()は，ユーザがウインドウ設計をする必要がなく，簡単に使用できて便利である．少数の（400 個以内の）データが動的に変化する様子を描くのに適しており，またデータの最大値に合わせて座標軸を動的に変化させていく．

しかしながら，「座標軸を固定したまま多数のデータを表示する」のには不適である．ここではグラフの座標軸を表示しないで，最大 2 万個までのデータを描く方法について述べる．

【書式】

(1) wopen(wp, "xyplot")
(2) plaingraph(wp)
(3) xwrite(wp, "q", [x1,y1,x2,y2])
(4) xwrite(wp, "c", [x1,y1,x2,y2])

【意味】

(1) 新規 xyplot ウインドウを開く．
　自動的に変数 wp にそのウインドウ番号（整数）が代入される．
(2) 番号 wp のウインドウの座標軸を消去する．新しい座標系は，
　ウインドウ左上端を原点(0,0)とし，右方向に横軸 x，下方向に縦軸 y である．
(3) 左上端の座標が(x1,y1)，右下端の座標が(x2,y2)であるような長方形を描く．
　(x1,y1) = (x2,y2)のときは点の代用として利用できる．
(4) 中心が(x1,y1)で，円周が点 (x2,y2)を通る，オレンジ色の円盤を描く．
　円の半径は $\sqrt{(x2-x1)^2 + (y2-y1)^2}$ となる．

【制限】

述語 wopen( )は，なるべく事前準備として preprocess0( )内で使用すること．種類が xyplot グラフ以外の場合，例えば，pie グラフや radar グラフを表示するためのウインドウを開いても座標軸は消えない．述語 xwrite()により，ディスプレー上に最大 2 万個まで図形（点の代用としての長方形）を描くことができる．show1()や show2()とは異なり，以前に描いた点は消えない．どんどん点が追加されて描かれて行く．点の座標は $x > 0, y > 0$ のとき（下向きの第 1 象限）だけ見ることができる．したがって，x や y の値が負になるときは，図形を移動させるなどの工夫が必要である．

【例】

```
//testplaingraph.set
preprocess() <->
        wopen(Wp1,"xyplot"), plaingraph(Wp1), Wp1.g:=Wp1,
        wopen(Wp2,"xyplot"), plaingraph(Wp2), Wp2.g:=Wp2;
inputsequence()="d"; times()=300;
initialstate()=c0 <-> c0:=0;
delta(c,a)=cc <-> cc:=c+1;
lambda(c)=b <->
        b:=0.01*c*(300 -c),
        show2(2,[[0,150,c],[0,150,b]],"trajectory"),
        xwrite(Wp1.g,"q",[c,b,c+2,b+2]),
        xwrite(Wp2.g,"c",[c,b,c+10,b+10]);
```

図 2-6-1

この例では，まず事前準備として座標軸のない新規ウインドウを開き，オートマトンの実行過程で y=0.03x(300-x)のグラフを描いている．

次図はこの例のモデルを実行した様子である．述語 show2 により座標軸をもつウインドウも作成されるが，座標軸のないウインドウと共存することができる．

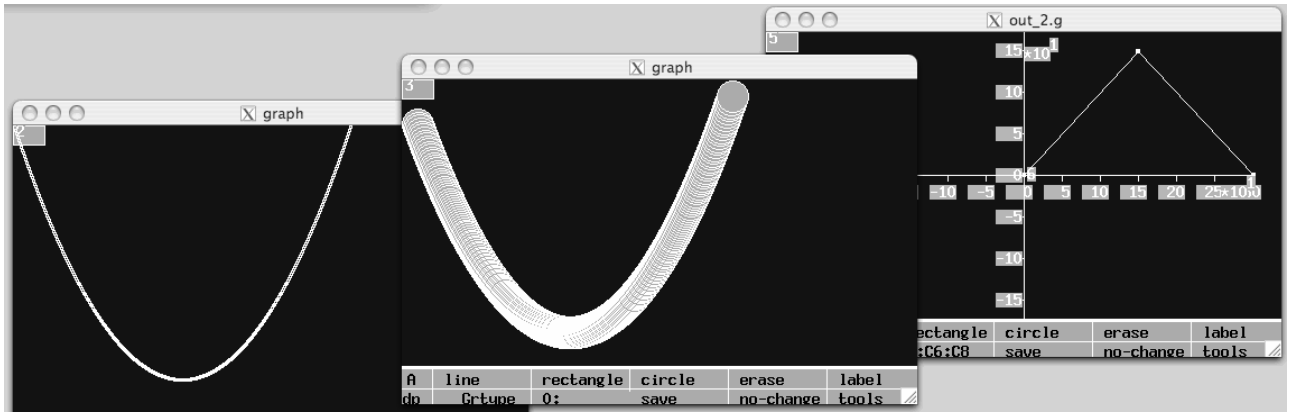


図 2-6-2

2.7 問題解決実行中のグラフ表示

本節では，微分方程式システムを近似的にオートマトンとしてモデル化する方法について紹介し，あわせて，グローバル変数を利用した，問題解決システム実行途中でのグラフ表示の方法も例示する．

2.7.1 微分方程式系の離散近似

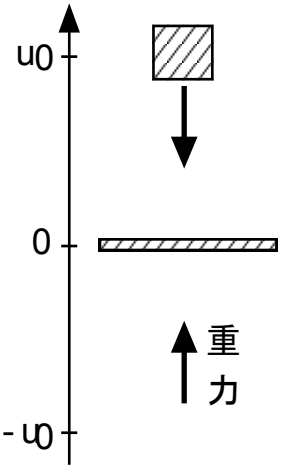


図 2-7-1 は，位置 $u=u_0$ から惑星（位置 u=0）に向かって落下しつつある物体 A である．そのままでは速度を増しながら惑星に向かって落ちていき衝突してしまう．しかし，ここでは仮想的に，位置 0 に重力が集まっており，そこを通過できる（衝突しない）ものとする．すると，惑星を通過した後，重力は逆向きになり，この物体 A は速度を減じていくだろう．やがて $u=-u_0$ に達した物体 A は，元の位置に向かって逆進することになるだろう．この運動は振動である．

図 2-7-1 物体の落下運動

以下では，とりあえず $u \geqq 0$ の場合を定式化する．ニュートン力学の運動方程式は次のようになる．

$u \geqq 0$ のとき，$d^2u(t)/dt^2 = -\alpha$ --- (1)

ただし，$\alpha > 0$ は惑星の重力加速度である．速度 v は位置 u の時間微分だから，

$$\begin{cases} dv(t)/dt = -\alpha \\ du(t)/dt = v(t) \end{cases} \quad \text{--- (2)}$$

が成り立つ．この**系（システム）の状態**は $(u(t), v(t))$ である．式(2)は，状態の変化を表現したモデルであるので，システムの（連続時間軸の）状態遷移関数である．しかし，一般にコンピュータシミュレーションでは，離散時間軸をもつ系しか扱えない（計算できない）ので，式(2)を，適当に定めた微小な時間間隔 h を使って離散近似することになる[ﾇ]．逐次，$(u(t+h), v(t+h))$を$(u(t), v(t))$から計算することになる．

微分に関する中間値の定理により，適当な $0 \leqq \theta_i \leqq 1$ によって，

$$\begin{cases} v(t+h) = v(t) + hv'(t + h\,\theta_1) \\ u(t+h) = u(t) + hu'(t + h\,\theta_2) \end{cases} \quad \text{--- (3)}$$

と書けるが，できれば計算誤差を小さくするように θ_i を決めて固定したい．逐次計算では $0 < \theta_i < 1$ に対する$(u(t + h\,\theta_1), v(t + h\,\theta_2))$ を計算することはなく，$\theta_i = 0$ または 1 の場合しか扱わない．しかも，$(\theta_1, \theta_2) = (1,1)$ では次の時刻の値を計算するのに次の時刻の値が必要となり計算不能であり，$(\theta_1, \theta_2) = (0,0)$では誤差が大きくなることは容易に分かる．した

ﾇ 離散近似の標準的な方法として，フーリエ展開を利用する方法があるが，刻み幅 h を十分に小さくする必要があるので，ここでは採用しない．

がって, (θ_1, θ_2) =(0,1)または(1,0)の２つの場合だけが考えられる. ここでは, (θ_1, θ_2) =(0,1)を採用する．すると，式(3)は次のようになる．

$$\begin{cases} v(t+h) = v(t) + hv'(t) = v(t) + h(-\alpha) \\ u(t+h) = u(t) + hu'(t+h) = u(t) + hv(t+h) \end{cases} \quad \text{--- (4)}$$

この式をオートマトン（正確には状態機械モデル）の**状態遷移関数**として CAST 言語で記述すると，次のようになる．

$$\begin{cases} v2 = v + h*(-\alpha) \\ u2 = u + h*v2 \end{cases} \quad \text{--- (5)}$$

式(5)は，時刻 t での状態が(u,v)であるとき，次の時刻(t+h)での状態が(u2,v2)になることを意味している．CAST 言語によるモデルは図 2-7-2 である．以下では，h=0.1, α =2 とする．

```
//gravity10.set
preprocess() <-> h.g:=0.1;
inputsequence()=c <->c:="d";
times()=t <-> t:=500;
initialstate()=c <-> c:=[10, 0];
delta([u,v],a)=[u2,v2] <->
        v2:=v + h.g*(-2)*u/abs(u),
        u2:=u + h.g*v2;
```

図 2-7-2　落下物体の CAST 言語によるモデル

速度 v2 の定義に u/abs(u)が付加されているのは，u＜0 のときに重力の向きが逆転することを表している．また，時間を進ませるために，ダミー入力"d"を 500 回入力するように定式化している．これをそのまま Simcast でコンパイルして実行することができる．

2.7.3 グローバル変数によるグラフ表示

しかし，ここでは実行途中の様子をグラフ表示したい．

```
//gravity11.set
preprocess() <-> h.g:=0.1, hist.g:=[], show2(7,[[5],[5]],"trajectory");
inputsequence()=a <->a:="d";
times()=t <-> t:=500;
initialstate()=c <-> c:=[10, 0];
delta([u,v],a)=[u2,v2] <->
        v2:=v + h.g*(-2)*u/abs(u),
        u2:=u + h.g*v2,
        showHistory([u2,v2]);

showHistory([u,v]) <->
        hist1:=hist.g,
        hist2:=append(hist1,[[u,v]]), len:=cardinality(hist2),
        hist3:=project(hist2,["r", len-199,len]),
        show2(7,transpose(hist3),"trajectory"),
        hist.g:=hist3;
```

図 2-7-3 グローバル変数によるグラフ表示

図 2-7-3 では，まず事前準備 preprocess として運動の履歴を保存するための hist.g:=[]を用意しておき，show2(2,[[5],[5]],"trajectory") でウインドウ out_7.g を開き，グラフをクリアしている．また，状態遷移のときにグラフ表示の仕事をさせるため，述語 showHistory を追加した．グローバル変数 hist.g に最新 200 個の状態を蓄積しておき，述語 show2 を使って状態の軌跡（trajectory）を描くためである．関数ではないので，ユーザ定義述語の宣言 func([]) は不要である．

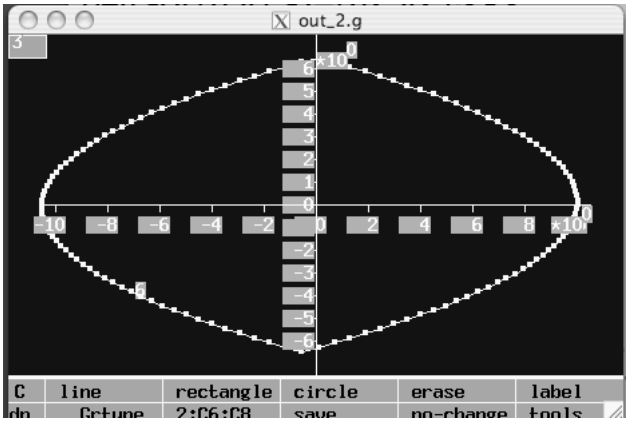


図 2-7-４ 振動系の位相図

これを実行すると図４のようになる．位置を横軸に，速度を縦軸にとり，点(10,0)から出発した状態(u,v)の軌跡が描かれている．これを位相図という．グラフは円ではなく，２次曲線をつないだものである．原点を中心とした円環状のグラフは振動を表している．

近似モデルなので，グラフはわずかに歪んでいる．しかし，大まかな刻み幅(h=0.1)と誤差があるにもかかわらず，もとの位置に戻る運動が繰り返されている．また，正確に u=0 にならないおかげで，u/abs(u)が無限大になることはない[3]．

実行時間に関しては，図３のモデルは 20 秒かかった．グラフ表示しない図２のモデル（1 秒）に比べて極端に遅い．

[3] 近似精度の分析は行なっていない．

2.7.3　ロケットの着陸制御

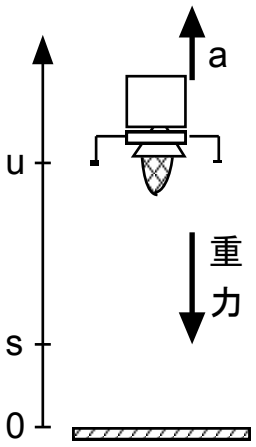


図 2-7-５　火星探査船

図 2-7-5 は，火星表面に着陸しようとする着陸船である．当然ながら，着陸船は火星の重力に引っ張られ，そのままでは速度を増しながら地表に向かって落ちていき，衝突してしまう．そこで，ロケットエンジンを逆噴射して落下速度を調節しながら，降下していくことになる．ただし，火星の大気は重く，高速で動く着陸船に対して，速度と逆向きに摩擦力が発生することを考慮する．

このシステムの状態は (u, v) である．また，ここでは着陸船に推力 a を加えて（エンジンを点火して）上下に動かすことができるものとする．運動途中で力 a が加わったとき，もちろん状態は影響を受ける．エンジンの推力 a がシステムへの入力である．しかし，力を加えないとき（a=0 のとき）も自律的に状態(u,v)が変化していく．a=0 も入力と考える．

着陸船の運動方程式は次のようになる．ここでは，u≧0 の場合のみ考える．

$$d^2u(t)/dt^2 = -\alpha - \beta v(t) + a(t) \qquad \text{--- (6)}$$

ただし，α>0 は火星の重力加速度，β ≧0 は大気との摩擦係数を着陸船の質量で割ったもの，また，$a(t)$はロケットの推力を着陸船の質量で割ったものである．離散近似した状態遷移関数は次のようになる．

$$\begin{cases} v2 = v + h(-\alpha - \beta v + a) \\ u2 = u + hv2 \end{cases} \qquad \text{--- (7)}$$

2.7.5 空中停止問題

さて，ここで問題である．ロケットが火星表面に衝突しないように，エンジンを逆噴射しつつ（推力 a を加えつつ），徐々に地表に近づいていく必要がある．どのように推力 a を加えていくべきだろうか？ ここでは，いま高さ 100 の位置で，すでに 80 の早さで地表に近づきつつあるものとし，いきなり着地しないで，高さ s.g=10 で速度が 0 になるようにしたいものとする．つまり，初期状態を(100,-80)とし，目標状態を(s.g,0)とする．

モデル理論アプローチでは，このような問題に対して，問題解決システムの開発環境を用意している[1]．図６がロケットの空中停止問題を解くための問題解決ユーザモデルである．ここに pow(x,y)は x の y 乗を表す述語である．

状態を評価する関数 goal([u,v])は加重和となっており，目標状態(s.g, 0)から離れることに対するペナルティーである．ペナルティーを小さくするように代替案(action)が As の中から選ばれていくのである．これを simcast でコンパイルし実行すると，取りあえずの解(action 列）が得られる．

[1] 高原康彦ほか『形式手法 モデル理論アプローチ：情報システム開発の基礎』日科技連 (2007)

```
//gravity40.set
preprocess() <-> s.g:=10, h.g:=1/10;
initialstate()=c <-> c:=[100,-80];
finalstate([u,v]) <-> [u,v]=[s.g,0];
delta([u,v],a)=[u2,v2] <->
        v2:=v+h.g*(-2 - 1*v + a),
        u2:=u+h.g*v2;
genA([u,v])=As <->
        As:=[-80,-40,-20,-10,-5,-3,-2,-1,0,1,2,3,5,10,20,40,80];
goal([u,v])=r <-> r:=30*pow(u - s.g,2) + 1*pow(v,2);
st([u,v]) <-> abs(u-s.g)+abs(v) <= 0.3;
```

図 2-7-6　空中停止問題を解くための問題解決ユーザモデル

文献[2]では実行後にまとめてグラフ化しているが，第３項のように，実行途中にグラフ表示するには，図 2-7-7 のようにモデル化すればよい．太字の部分が追加したものである．

```
//gravity41.set
func([_Solf]);
preprocess() <-> s.g:=10, h.g:=1/10, hist.g:=[], show2(2,[[5],[5]],"plot");
initialstate()=c <-> c:=[100,-80];
finalstate([u,v]) <-> [u,v]=[s.g,0];
delta([u,v],a)=[u2,v2] <->
        v2:=v+h.g*(-2 - 1*v + a),
        u2:=u+h.g*v2;
genA([u,v])=As <-> showHistory([u,v]),
        As:=[-80,-40,-20,-10,-5,-3,-2,-1,0,1,2,3,5,10,20,40,80];
goal([u,v])=r <-> r:=30*pow(u - s.g,2) + 1*pow(v,2);
st([u,v]) <-> abs(u-s.g)+abs(v) <= 0.3, st.g:=[u,v];

showHistory([u,v]) <->
        hist1:=hist.g,
        hist2:=append(hist1,[[u,v]]), len:=cardinality(hist2),
        hist3:=project(hist2,["r", len-199,len]),
        show2(2,transpose(hist3),"plot"),
        hist.g:=hist3;
postprocess()<->
        xwriteln(0,"hist.g=",hist.g),
        xwriteln(0,"Solf2=",_Solf()),
        xwriteln(0,"st=",st.g);
```

図 2-7-7　グローバル変数によるグラフ表示

Solver では，preprocess()と postprocess()が使用できること，状態の取得は genA()の副作用として行なえばよいことがポイントである．また，_Solf()を関数宣言すると，「最後に」解（action 列）を取得することができる．次の図 2-7-8 が実行結果である．

[2] Y. Takahara and Y. Liu, *Foundation and Applications of MIS : A Model Theory Approach* , Springer (2006)

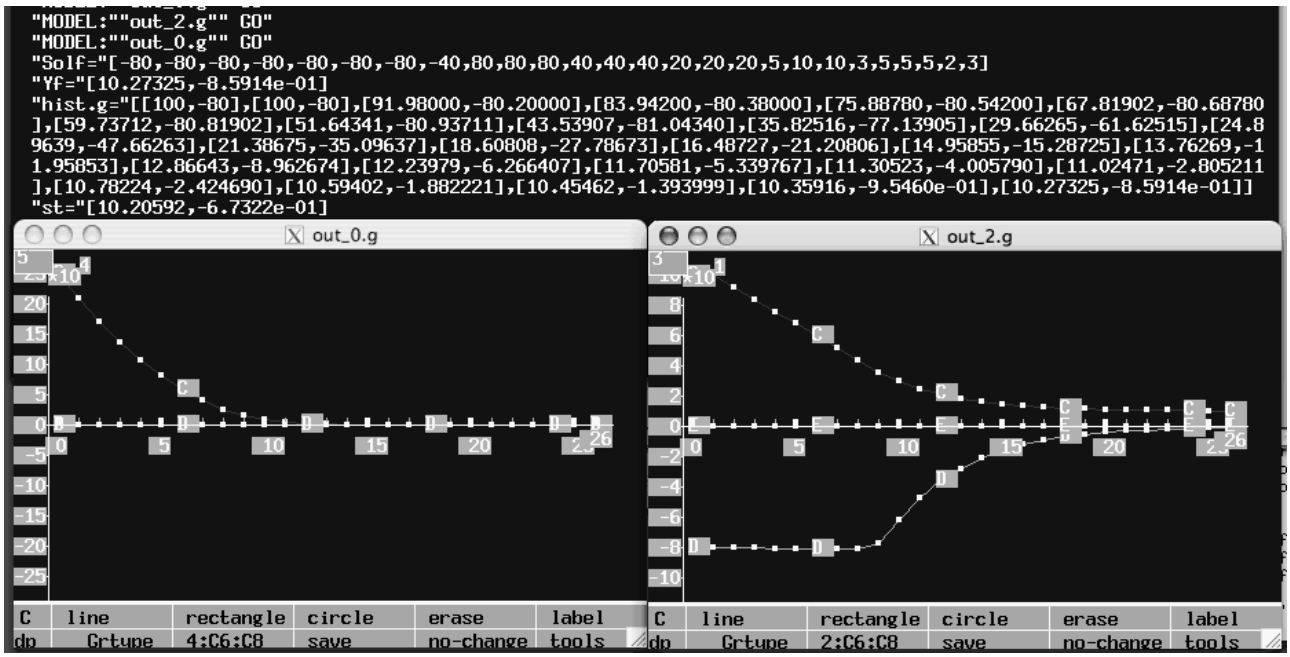


図 2-7-8　Solver 実行中のグラフ表示

図 2-7-8 の左下ウインドウは関数 goal()の値の変化を表示している. 右下ウインドウでは, 横軸の上に位置 u の変化を，下に速度 v の変化を表示している．位置の変化は滑らかであるが，速度の変化は急激である．文献℘によると，制約条件 constraint を付加して，速度の変化も滑らかにするような解を導くことができる．バックトラックが頻繁に起きることが確認できる．他のグラフ表示法としては，経過した状態をファイルにすべて記録し，後でグラフ化する方法もあろう.

2.7.6 最適制御問題

前項 4.1 の空中停止により，安全性は確保された．しかし実際には，燃料も無駄にすることはできない．つまり，安全性に加えて経済性をも考慮しなければならない．その場合には，関数 goal([u,v])に入力 a の積分項を追加する必要があるが，変数が[u,v]では率直に定式化できない．状態を[u,v,g,t]に拡張するなど，モデル化の工夫が必要である[3].

℘

[3] 前掲書 2 の第 8 章に詳しい

2.8 スプレッドシートによるデータ表示

本節では，スプレッドシートの表示と書き込み，読み込みの方法について紹介する．以下では，SS ウインドウを開閉し，値を代入，取得するオートマトンモデルを例示する．

2.8.1 SS ウインドウの開閉

スプレッドシートウインドウ（以下 SS ウインドウと略す）の開閉に関する述語は次の 4 つである．

【書式】
(1) makesheet(X,Y,ウインドウ番号)
(2) makesheet (ウインドウ名称,X,Y,ウインドウ番号)
(3) getWp(ウインドウ名称,ウインドウ番号)
(4) myclearwindow(ウインドウ番号)
変数 X,Y は表示するセルの行数，列数を表す．

【例】
(1) makesheet(11,4,Wp1)
(1') makesheet(X,Y,Wp)
(2) makesheet("mySheet",11,4,Wp)
(3) getWp("mySheet",Wp)
(4) myclearwindow(3)

【意味】
述語 makesheet により常に 600 行×200 列のスプレッドシートが作成される．ただし，それ全体をウインドウに表示することはできないので，表示の範囲を指定することになる．
(1) "sheet"という名称の SS ウインドウを開き，11 行 4 列分のセルを表示する．システムが自動的にウインドウ番号を付与し，変数 Wp に代入する．
(1') "sheet"という名称のスプレッドシートウインドウ（以下 SS ウインドウ）を開く．システムが自動的に行数 X 列数 Y とウインドウ番号を付与し，変数 X,Y,Wp に代入する．
(2) "mySheet"という名称の SS ウインドウを開き，11 行 4 列のセルを表示する．システムが自動的にウインドウ番号を付与し，変数 Wp に代入する．
(3) "mySheet"という名称の SS のウインドウ番号を取得し，変数 Wp に代入する．その名称のウインドウが存在しないときは，Wp に -1 が代入される．
(4) ウインドウ番号 3 の SS 内すべてのセルを空にする．

通常使用する述語はこれだけであるが，セル幅を指定したいときは次の特別な述語を使用する．

【特別な述語】
書式(5) openlocalsheet(Name,WinY,WinX,Y,X,Y0,X0,NotMove,Map,Len_cell,Wp)
例(5)　openlocalsheet("mySheet",0,1,150,3,0,0,1,1,1,Wp)
意味
名称 mySheet の SS ウインドウを開き，Wp にそのウインドウ番号を代入する．
ただし，
1）(WinY,WinX)=(0,1)だから，ウインドウの位置はディスプレーの左中央である．

2）(Y0,X0)=(0,0)だから，ウインドウ内の左上端に表示されているのはセル(1,1)である．
3）(Y,X)=(150,3)だから，セル(1,1)からセル(3,150)までの領域が表示される．
4）NotMove は実装されていないので，NotMove=1 は無関係である．
5）Map=1 なので，ウインドウ mySheet は他のウインドウより前面に置かれる．
6）Len_cell=1 なので，セル幅は 1 である．

述語 openlocalsheet は特別に詳細に SS を設定したいときに利用するだけなので，昔の書式を改めていない．適当に数値を変更してどうなるかを体感することにより理解してもらいたい．第 3.4 節「シミュレーションの高速化」に使用例がある．本書では詳述しない．以下では，簡単に使える makesheet に関して説明する．

開き方

述語 makesheet の問題点は，一度開いたシートをうまく消さないと，シミュレーションを繰り返すうちに，どんどんシートの数が増えてゆくことである．ここでは回避の工夫として，すでに開いている SS ウインドウを再利用することにする．

SS ウインドウはウインドウ番号 Wp によって区別する．したがって，ウインドウの再利用をするには, ウインドウ名称からウインドウ番号を知るための述語 getWp()が必要である．新しく用意したので利用されたい．

さて，ウインドウを開くといった,「シミュレーション開始前の準備作業」は，事前準備 preprocess の定義中に記述することができる．方針としては，もしも"sheet"という名前の SS がすでに開いていればすべてのセルの内容を空白にし利用可能にする．しかし，まだ開いてなければ新規に SS ウインドウを作成する，というものである．例えば，次のオートマトンモデルのようになる．

```
//ss10.set
preprocess() <->
        getWp("sheet",Wp),
        (Wp >0) ->
                ( myclearwindow(Wp),Wp.g:=Wp )
        otherwise
                (makesheet(11,4,Wp1), Wp.g:=Wp1);
inputsequence()="d";
times()=1;
initialstate()=c0 <-> c0:=0;
delta(c,a)=cc <-> cc:=c;
```

図 2.8.1 スプレッドシートを表示する

これにより，SS ウインドウの再利用ができる．ウインドウだけでなくセル内の値までそのまま再利用したい時（セルを空白にする必要がない場合）は，myclearwindow(Wp)は不要である．オートマトン実行中に使用する予定であるので，すぐにグローバル変数 Wp.g にウインドウ番号を代入している．オートマトンモデル ss10.set を実行すると，図 2.8.2 のようになる．

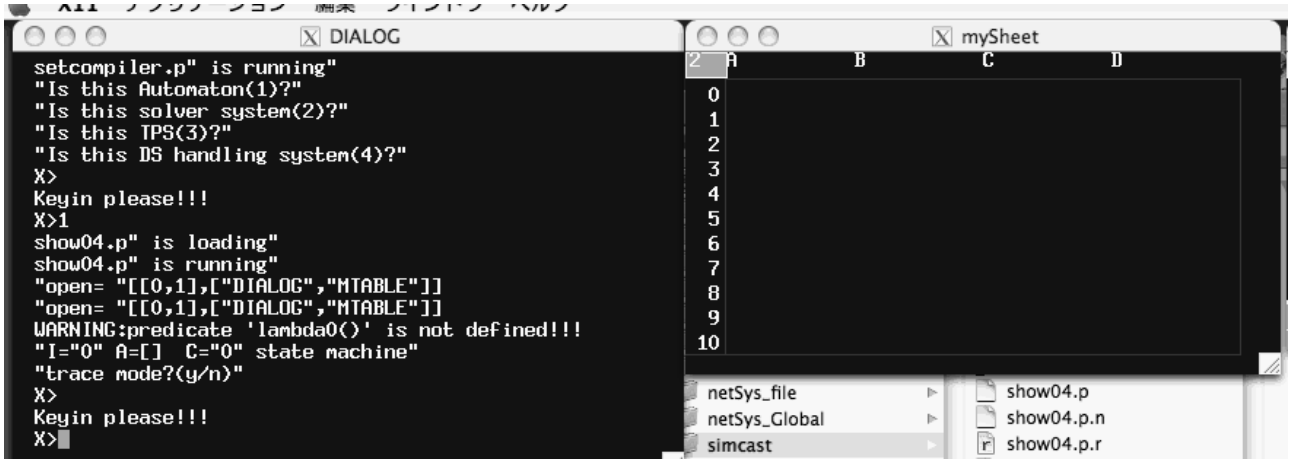


図 2.8.2　スプレッドシートの表示

閉じ方

ウインドウ右上端の×印をポインターでクリックすると，すべてのウインドウが閉じ，実行中のシステムも終了する．SS ウインドウだけを閉じる方法が判明しない．

2.8.2 SS との入出力

すでに開いている SS にオートマトンから値を書き込んだり，SS から読み込むための述語には次のようなものがある．

【書式】
(1) Z:=ssr(Wp,X,Y)
(2) ssw(Wp,X,Y,3)
(3) Z:=projcetss(Wp,X1,Y1,X2,Y2)
(4) writess(Wp, "r", X, Y, M)
(5) replacess(Wp,[[X1,Y1,V1],[X2,Y2,V2]])
変数 Wp はウインドウ番号，変数 X,Y はセル位置，変数 M は行列を表す．

【例】
(1) Z:=ssr(Wp,3,1)
(2) ssw(Wp,1,2,3)
(3) D:=projcetss(Wp,2,1,3,3)
(4) writess(Wp, "r", 7, 8, [[1,2,3],[3,4,5]])
(5) replacess(Wp,[[4,3,1],[4,4,"bbb"]])
【意味】
(1) ウインドウ番号 Wp の SS のセル(3,1)から値を取得し，変数 Z に代入する．
(2) ウインドウ番号 Wp の SS のセル(1,2)に 3 を書き込む．
(3) ウインドウ番号 Wp の SS 内の，セル(2,1)を左上端としセル(3,3)を右下端とする長方形の領域内の値をまとめて取得し，行列として変数 Z に代入する．
(4) ウインドウ番号 Wp の SS のセル(7,8)を基準として，2 行 3 列のデータ[[1,2,3],[3,4,5]]をまとめて書き込む．オプションが"r"でなく"c"のときは，データを転置して書き込む．
(5) ウインドウ番号 Wp の SS のセル(4,3)の値を 1 に，セル(4,4)の値を bbb に書き直す．

文字は引用記号で囲み"aaa"などとする．オートマトン内で２つの引用記号を並べた""（長さ 0 の文字列）がセル内では空（empty）として表示される．たとえば，ssw(Wp,1,3,"")と

したとき，1 行 3 列目のセルに空白が書き込まれる．一方，5 行 7 列目のセルが空のとき，z:=ssr(Wp,5,7)で，変数 z に""が代入される．

【モデル例】

```
//ss12.set
preprocess() <->
        getWp("mySheet",Wp),
        (Wp >0) -> (Wp.g:=Wp)
        otherwise
                (
                makesheet("mySheet",11,4,Wp1), Wp.g:=Wp1,
                Ps:=[[1,2,3,4],[5,6,7,8],[9,10,"",12],[13,14,15,"aaa"]],
                writess(Wp.g,"r",1,1,Ps)
                );
inputsequence()="d";
times()=10000;
initialstate()=c0 <-> c0:=0;
delta(c,a)=cc <-> cc:=c,
        x:=ssr(Wp.g,2,4)+4,   //x は数値
        ssw(Wp.g,2,4,x+1),
        y:=projectss(Wp.g,1,1,1,3),   //z は行列
        writess(Wp.g,"r",1,1,y+1),
        replacess(Wp.g,[[4,3,ssr(Wp.g,4,3)+1],[4,4,"bbb"]]);
```

図 2.8.3　スプレッドシートと値を授受するオートマトン

このユーザモデルを実行すると次のようになる．1) Simcast を起動した直後，トレースモードにするかどうかの選択の前に，"sheet"という名称の SS ウインドウが開き， SS にデータ[[1,2,3,4],[5,6,7,8],[9,10,"",12],[13,14,15,"aaa"]] が書き込まれる． 2) 続いて，トレースモードにしない選択をする（n を入力する）と，SS の 2 行 4 列目のセルに 5 を加え，次に第 1 行目を読み込み，行列 y として 1 を加えてから，SS の第 1 行目に書き込む（図 3）．最後に，replacess で 2 つのセルの値を書き直す．

要するに SS から値を取得し，数値を加えてから，SS に書き込んでいる．回数が times() = 10000; のとき，実行時間は 5 秒であった．

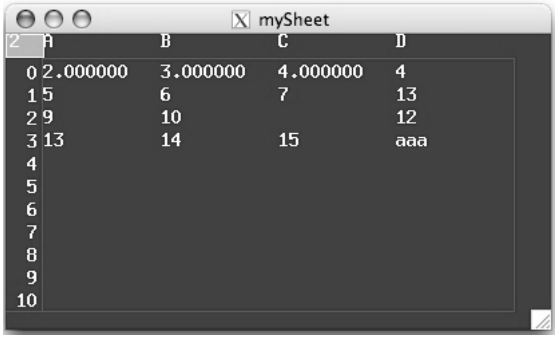


図 2.8.4　セル内の値が変化していく SS

2.8.3 CSV ファイルの読み書き

第 2.4 節によると，オートマトン実行中の処理途中や最終データのファイル保存が可能である．述語 appendf( )でデータをファイルに書き込め，xread( )で読み込むことができる．シミュレーションに関しては，これだけで問題はない．しかしながら，それらのデータを市販の表計算ソフト（例：MS-Excel）に読み込んで分析をしたり，報告書を書きたいこともあるだろう．そのような場合は，保存したファイルのデータを CSV 形式（各値をコンマで区切る形式）に書き換えて，取り込めばよい．

例えば，データ[[1,2,"",4],[5,6,7,8],[9,10,11,12],[13,14,15,16]]を，オートマトンからファイルに書き込むと，ファイルの中では改行され，

```
[1,2,"",4]
[5,6,7,8]
[9,10,11,12]
[13,14,15,"aaa"]
```

となって保存されている．適当なテキストエディタでこれを開き，置換機能を使い，各行の両端にある括弧[ ]を取り除けば CSV 形式（各値をコンマで区切る形式）となる．必要なら改行を CR+LF にして保存する．ここで，市販の表計算ソフトを起動すれば，CSV 形式として読みことができる．

逆に，すでに存在する CSV ファイルは，事前に各行の両端に括弧を挿入し（必要なら改行を LF にし）ておけば，述語 xread でオートマトンに取り込むことができ，前節で述べたように，それを Simcast（正確には actDSS）の SS に表示することができる．

2.8.4　SS を操作する述語の定義（Prolog.ctl に追加記述したもの）

```
getWp(Wname,Wp):-
        procC("getWp",["openwin"],[X,Y]),
        xwriteln(0,"win opened= ", [X,Y]),
        Z:=transpose([X,Y]),
        if member([Wp,Wname],Z) then xwriteln(0,"Wp=",Wp) else Wp:=-1;
makesheet(X,Y,Wp):-openlocalsheet(1,1,Y,X,Wp);
makesheet(Name,X,Y,Wp):-openlocalsheet(Name,1,1,Y,X,0,0,1,Wp);
ssr(Wp,X,Y,N):-(N:=#(Wp,Y-1,X-1));
ssw(Wp,X,Y,N):-(#(Wp,Y-1,X-1):=N);
projectss(Wp,X1,Y1,X2,Y2,N):-xread(Wp,Y1-1,X1-1,Y2-1,X2-1,N);
writess(Wp,T,X1,Y1,D):-xwrite(Wp,T,Y1-1,X1-1,D);
replacess(Wp,D):-    // D must be a list of [i,j,v]
        if D=nil then ( ) else
                if exptype(D,"matrix") then
                        strlen(D,Nmax),
                        for (N:=1; N<=Nmax; N++)
                        begin
                                project(D,N,U), [I,J,V]:=U,
                                #(Wp,J-1,I-1):=V
                        end
                else
                        xwriteln(0,"replacess : D is not [[i,j,v]]"), fail
                end
        end;
```

2.9 テキストによるデータ表示

ここでは，新しい text ウインドウを開いて，そこにデータを表示する方法を述べる．しかし，もともと Dialog ウインドウに文字を表示することができていたし，text ウインドウの使い勝手はよくないので，今の所，スプレッドシートによるデータ表示の方法を推奨する．

【書式】
[1] wopen(Wp,Wtype)「ウインドウを作成する」Wp：ウインドウ番号，Wtype：ウインドウタイプ
[2] getWp(Wname,Wp)「ウインドウ番号を取得する」Wname：ウインドウ名称，Wp：ウインドウ番号

【使用例】
[1] wopen(Wp,"text")
text という名前（固定）の”text"型ウインドウを開き，ウインドウ番号を自動的に割り振って変数 Wp に代入する．
[2] getWp("myWindow",Wp)
myWindow という名前のウインドウを探し, そのウインドウ番号を変数 Wp に代入する．存在しないときは 0 を Wp に代入する．

【モデル例】
```
//testtextw.set
preprocess() <->
        getWp("text",Wp),
        (Wp>0)->(Wp1:=Wp) otherwise (wopen(Wp1,"text")),
        Wp.g:=Wp1;

inputsequence()=As <-> As:=[1,2,3,4];
initialstate()=c <-> c:=[0];
delta(c,a)=cc <->
        cc:=append(c,[a]),
        xwriteln(Wp.g,"c=", c);
```

【動作】
初期状態（initialstate)の設定のとき，
すでに text という名称のウインドウがあれば，それを使う．
なければ text という名称のウインドウを作る（wopen).
ウインドウ番号はグローバル変数(Wp.g)に代入しておく．
この例ではテキストの書き込みに述語 xwriteln を使った．

【注意】
テキストウインドウをリフレッシュする述語とテキストウインドウに空行（改行のみ）を表示する述語はない．リフレッシュしたい場合，便宜的に，ウインドウの行数だけ xwriteln(Wp.g,"")を実行すればよい．

第 3 章 開発方法論

Simcast におけるモデルの中心的なものはオートマトンである．ここではオートマトンを結合して複雑なモデルを作成する方法について述べる．

3.1 複合システムの分類とモデル

3.1.1 状態機械の複合システム

本節では，状態機械（state machine）が結合したシステムをモデル化する方法や考え方を紹介する．あわせて，それらの CAST 言語によるモデルを紹介する．基本的な原理は，「複数の状態機械が結合したシステムは，各要素の状態を並べた組を状態とする，全体として一つの状態機械である」ということである．

以下，第２項では，要素システムである状態機械を定義する．第３項では，逐次処理システムとして，直列結合，並列結合，フィードバック結合を紹介する．第４項では，並列処理システムとして，ネットワーク，固定結合網，可変結合網，同期型，非同期型を紹介する．

最も注意すべきは, 逐次処理システムと並列処理システムは, たとえ同じ図であっても，全く異なるシステムであり，異なるモデルを作る必要がある点である．それぞれの項では CAST 言語によるモデルも紹介する．

3.1.２ 状態機械と複合システム

まず要素システムである状態機械を定義し，その CAST 言語によるモデルを紹介する．

(1) 状態機械

まずオートマトンについて考える．オートマトンは，入力に応じて状態が遷移し，遷移後の状態に対して出力が決まるようなシステムのモデルである．

ある時刻でのシステム S への入力を a，出力を b，状態を c とする（図 3-1-1）．また，次の時刻での状態を cc と書くことにする（cc は２つの c を乗じたものではなく，ひとつの記号として扱う）．これらの変数の間に，

$\delta(c,a)=cc$ かつ $\lambda(c)=b$

という関係があるとき，S は（Moore 型）オートマトンであるといい，$<\delta, \lambda, c0>$を S の Moore 型オートマトンモデルと呼ぶ．また，δ を状態遷移関数，λ を出力関数， c0 を初期状態と呼ぶ.

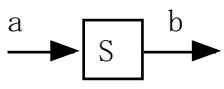


図 3-1-1　要素システム S1

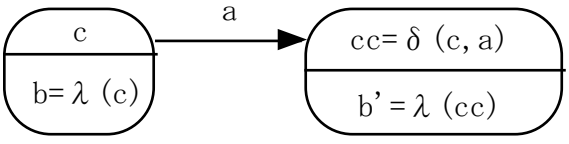


図 3-1-2　状態遷移図

状態遷移図は図 3-1-2 のようになる．時刻 t における状態が c のとき，その時刻での出力は $\lambda(c)$ である. そこに a が入力されると, 次の時刻 t+1 における状態は $cc=\delta(c,a)$ となり，$b'=\lambda(\delta(c,a))$ が出力される．逆に言えば，現在の状態や出力が過去の状態と過去の入力に

よって決定されるシステムがオートマトンである．

一般に，ある時刻の出力が過去に依存せず，その時刻の入力 a のみに依存して定まるようなシステムを**静的システム**（static system）と呼び，それ以外を動的システム（dynamic system）と呼ぶ．直感的に言えば，静的システムは何ものも蓄積せず入力の値だけに依存して瞬間的に反応するシステムのことである．同じ入力なら同じ出力を同時刻に返すことになる．式で書くなら f(a) = b となる．これに対して，動的システムは，何かを蓄積していきながら，それを参照しつつ入力に反応するシステムのことである．したがって，同じ入力でありながら状態が異なるので出力も異なる場合がある．したがってオートマトンは動的システムである．

さて，通常は，オートマトンは同じ出力 b でありながら内部の状態が異なる場合も想定されている（λ(c) = λ(c')）．しかしながら，本節では特別に，出力関数が恒等関数であるような場合（λ(c) = c）を考える．つまり，そのときの出力を見れば内部の状態が一意的に推定できるシステムであり，「出力＝状態」と見なすことのできるオートマトンである．出力関数が恒等関数であるようなオートマトンを**状態機械**（state machine）と呼ぶ．もはや出力関数について考える必要はないので，状態遷移図も簡略化して書くことができる（図 3-1-3）

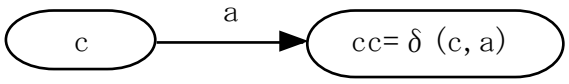


図 3-1-3　状態機械の状態遷移図

(2) 状態機械の CAST 言語によるモデル

具体的な例として，任意の時刻の入力 a をそのまま次の時刻に出力するようなシステム S1 を考える．逆に考えると，入力を一時的に状態として保持する（記憶する）ようなシステムである．これを状態機械としてモデル化すると，

δ 1(c,a) = cc <-> cc = a

となる．たとえば，初期状態を c0=0 とすると，このモデル ＜δ 1, 0＞ は，a が入力されたとき，次の状態 cc が a と等しくなるようなモデルである．このシステム S1 の状態機械モデル ＜δ 1, 0＞ を CAST 言語で書くと図 3-1-4 のようになる．

```
//sim10.set
inputsequence()=As <-> As:=[1,0,0];
initialstate()=c0 <-> c0:=0;
delta(c,a)= cc <-> cc:=a;
```

図 3-1-4　要素システム S1 のコンピュータ実現モデル

図 3-1-3 の initialstate()=c0 <-> c0:=0; は初期状態を定義している．状態遷移関数の定義は，delta(c,a)= cc <-> cc:=a; である．このシステムを動かすには入力が必要であるので，inputsequence()=As <-> As:=[1,0,0]; で事前に具体的な入力列を指定している．リスト [1,0,0] は 1 回目に 1 を入力し，2 回目に 0 を入力し，3 回目も 0 を入力することを意味している．入力が終わればシミュレーションは終了する．

テキストエディタでこれを作成し，例えば sim10.set というファイル名で保存する．その後，開発環境 Simcast の実行可能ファイル xsheet を起動し，ファイル名 sim10.set を入力すると，コンパイルされたのち実行され，図 3-1-5 の最終３行のような結果が得られる．

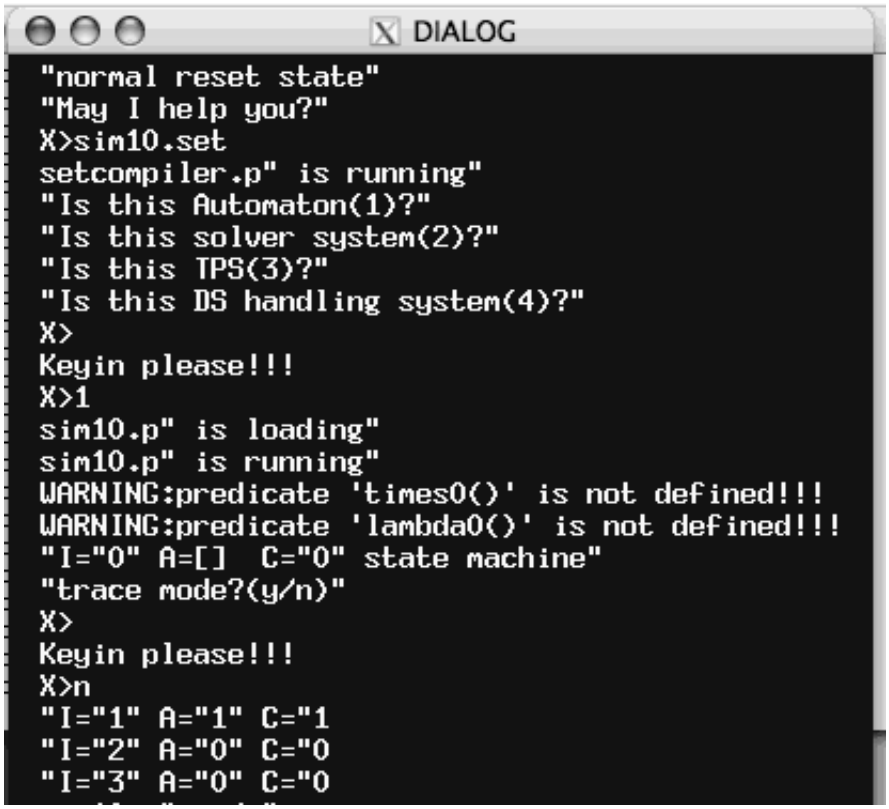


図 3-1-5　モデル sim10.set のコンパイルと実行

しかしながら，本稿では，さらに複数のオートマトンを追加し，組み合わせて複雑なモデルを構築する予定なので，図 3-1-6 のように inputsequence()，initialstate()，delta(c,a)，を最後に記述することにしておく．これは作法である．

```
//sim11.set
func([delta1]);
delta1(c,a)=cc <-> cc:=a;

inputsequence()=[1,0,0];
initialstate()=0;
delta(c,a)=delta1(c,a);
```

図 3-1-6　シミュレーションのための S1 のモデル

このモデルでは，まず delta1 というユーザ関数を定義しておき，あとで（システム全体の状態遷移関数）delta を delta1 と等しいものとして定義している．図 3-1-6 は図 3-1-4 と同等であるから，このモデルを Simcast でコンパイルし，実行すると，図 3-1-5 と同じ結果が得られるはずである．そして実際にそうなる

練習問題 1 ：図 3-1-6 のモデルを実際に作成し，実行してみよ．

(3) 複合システム

複数の要素システムが結合してできる全体のシステムを**複合システム**と呼ぶ．結合形態の視点から分類すると，各要素（ここでは状態機械）が結合する仕方には，**フィードバック結合**，**直列結合**，**並列結合**の 3 種類がある．

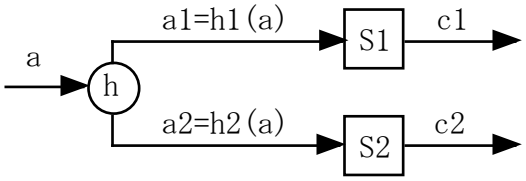


図 3-1-7　並列結合

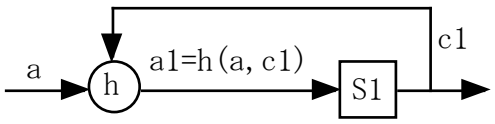


図 3-1-8　フィードバック結合

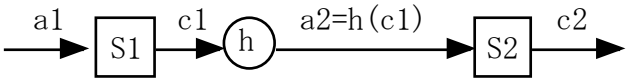


図 3-1-9　直列結合

丸印で表わす**結合関数** h は，２つ以上の要素システムを結合するための「接着剤」に相当する．これは静的システムである．一般に，どのような複雑な結合形態でも，これら３種類（図７～図９）を組合わせたものである．例えば，次の図 3-1-10 は並列結合とフィードバック結合を組合わせた複合システムである．

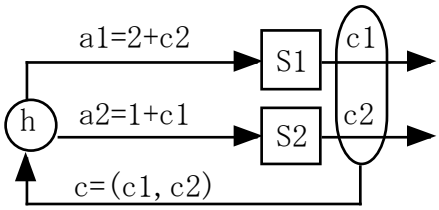


図 3-1-10　並列結合＋フィードバック結合

以下，要素システム S*i* のモデルを ＜δ *i*, 0＞ とする（*i*=1,2,3)．また h, h1, h2 は関数（静的システム）であるとする．

さて，結合の形態が同じ（結合図式が同じ）であっても，各要素の動作の種類によって複合システムの行動は異なってくる. 要素の動作には逐次処理と並列処理の 2 種類がある．

複合システムが**逐次処理システム**（sequential processing system）であるとは，「その**各要素が１度にひとつしか動作しない**」システムのことである．つまり，「要素 S1 が処理を実行しているときに S2, S3 は動いておらず, 要素 S2 が処理を実行しているときは S1, S3 が動いていない」ようなシステムであるとするのである．このように，結合の並びの順に（図の左から右に向かって）各々の要素システムだけが動作するような複合システムを逐次処理システムと呼ぶ．例えば，基板の上に複数の電子的な部品が焼き込まれた電子回路は，（多くは）逐次処理システムである．

また，複合システムが**並列処理システム**（concurrent processing system）であるとは，結合の形態にかかわりなく，「**すべての要素がいつでも自律的に動いている**」ような複合システムのことを言う．例えば，自律的に動いている多くのパソコンが結合しているインターネットは並列処理システムである．

以下の節では要素システムの結合の仕方に応じて，複合システムをどのようにモデル化できるかを紹介する．最も注意すべきは，逐次処理システムと並列処理システムは，たとえ同じ結合図式であっても，全く異なるシステムであり，異なるモデルを作る必要がある点である．

3.1.3 逐次処理システムのモデル

まず，基本的な結合形態（直列，並列，フィードバック）の逐次処理システムの言語 CAST によるモデルを紹介する．

(1) （逐次処理）直列結合のモデル

例として，次図 11 のように要素 S1，S2，S3 が直列結合したシステム S5 を考える．

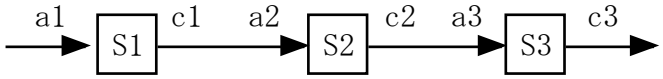


図 3-1-11　複合システム S5（結合関数が a2=c1, a3=c2 の場合）

これは，図 3-1-8 中の結合関数 h が恒等関数 h(c)=c である複合システムであり，まず要素 S1 が入力を処理し，次に要素 S2 が S1 の出力を受け取って処理するものである．この逐次処理の動作を図で表現すると，次のようになる．

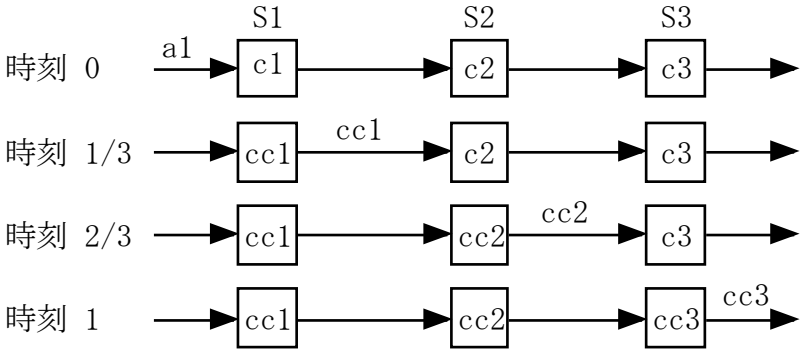


図 3-1-12　逐次処理の動作

このとき，全体システム S5 は，(c1,c2,c3)を状態とする状態機械である．なぜなら，S5 の次状態 cc は，関数

δ5((c1,c2,c3),a1)=cc <->

cc1= δ1(c1,a1) かつ cc2= δ2(c2,cc1) かつ δ3(c3,cc2) かつ

cc = (cc1 , cc1 , cc1).

で計算できるからである．したがって，全体システム S5 は δ5 を状態遷移関数とするひとつの状態機械であり，その数理的なモデルは S5 = < δ5, (0,0,0) > である．全体を一つと見たときの初期状態は(0,0,0)とした．

例えば要素システムが入力の２倍を次状態とするものであるとして，この逐次処理直列結合システム S5 を具体的に CAST 言語でモデル化すると図 3-1-13 のようになる．

```
//sim14.set
func([delta1,delta2,delta3,delta5]);
delta1(c,a)=cc <-> cc:=2*a;
delta2(c,a)=cc <-> cc:=2*a;
delta3(c,a)=cc <-> cc:=2*a;
delta5([c1,c2,c3],a)=cc <->
        cc1:=delta1(c1,a),
        cc2:=delta2(c2,cc1),
        cc3:=delta3(c3,cc2),
        cc:=[cc1,cc2,cc3];
inputsequence()=[1,2,0];
initialstate()=[0,0,0];
delta(c,a)=delta5(c,a);
```

```
"I="1" A="1" C="[2,4,8]
"I="2" A="2" C="[4,8,16]
"I="3" A="0" C="[0,0,0]
```

（CAST 言語によるモデル）　　（実行結果）

図 3-1-13　逐次処理直列結合システム S5 の CAST 言語によるモデル

モデル理論アプローチによるシミュレーション開発実行環境 Simcast では，ひとつの状態機械（オートマトン）しか実行できないが，全体システム S5 = < δ 5, (0,0,0) >がひとつの状態機械である（とみなせる）ので，Simcast で動かすことができる．Simcast でコンパイルし実行した結果は図 3-1-13 の右側に示した．確かに，時刻 1 で 1 が入力されたときは，その倍の 2，その倍の 4，その倍の 8 が各要素の状態になっている．

ここで，時間の進み方について考える．逐次処理システムでは，要素をひとつずつ実行するけれど，最後の要素が実行を終えたときに全体として時刻が 1 つ進むと考えている．「複合システムの中にある要素すべてが実行を終えたときに，全体として時刻がひとつ進む」と考えればよい．

(2) 並列結合

次に，並列結合について考える．複合システム（図 3-1-7）を S01 とするとすると，全体としての状態は要素システムの状態の組(c1,c2)である．また，次の時刻の状態は

δ 01((c1,c2),a) =cc <-> a1=h1(a) かつ a2=h2(a) かつ cc= (δ 1(c1,a1), δ 2(c2,a2))

で計算することができる．したがって，S01 の状態機械モデルは < δ 01,(0,0)> である．

ここで例えば，結合関数が h1(a)=a, h2(a)=a で，要素システムが入力の２倍を次状態とするものであるとすると，CAST 言語によるモデルは次のようになる（図 3-1-14）．

```
//sim01.set
func([delta1,delta2,delta01]);
delta1(c1,a1)=cc1 <-> cc1:=2*a1;
delta2(c2,a2)=cc2 <-> cc2:=2*a2;
delta01([c1,c2],a)=cc <->
        a1:=a,   //結合関数 h に相当
        a2:=a,
        cc:=[delta1(c1,a1),delta2(c2,a2)];
inputsequence()=[2,0,0];
initialstate()=[0,0];
delta(c,a)=delta01(c,a);
```

```
"I="1" A="2" C="[4,4]
"I="2" A="0" C="[0,0]
"I="3" A="0" C="[0,0]
```

（CAST 言語によるモデル）　　（実行結果）

図 3-1-14 並列処理直列結合システム S01 のモデル

複合システム S01 の状態は c=(c1,c2)であるため，初期状態は initialstate() = [0,0] となっている．実行結果は図 3-1-14 右に示した．時刻 1 で S01 に 2 が入力され次状態が[4,4]と計算され，次の時刻に 0 が入力されたときには次状態が[0,0]になっている．

さて，モデル sim01.set を次のように書き直すこともできる．

```
//sim02.set
func([delta1,delta2,delta02]);
delta1(c,a)=cc <-> cc:=2*a;
delta2(c,a)=cc <-> cc:=2*a;
delta02([c1,c2],a)=cc <->
        [a1,a2]:=[a,a],
        cc1:=delta1(c1,a1),
        cc2:=delta2(c2,a2),
        cc:=[cc1,cc2];
inputsequence()=[2,0,0];
initialstate()=[0,0];
delta(c,a)=delta02(c,a);
```

図 3-1-15　変数を使った表記法

図 3−1−14 と図 3−1−15 の主な違いは状態遷移関数の定義のしかたである．図 3-1-15 では変数 cc1, cc2 を使っている．どちらでも読者好みの流儀でかまわない．

練習問題２：このモデル sim02.set を simcast でコンパイルし，実行すると，sim01.set と同じ結果が得られるはずである．実際にやってみよ．

(3) フィードバック結合のモデル

次に，フィードバック結合について考える．複合システム（図 3-1-8）を S03 と呼ぶことにすると，全体システムの状態は要素システムの状態 c そのものである．また，次の時刻の状態は

δ03(c1,a)=cc <-> a1= h(a,c1) かつ δ1(c,a1)

で計算することができる．したがって，S の状態機械モデルは ＜δ03,0＞ である．

ここで例えば，結合関数が h(a,c1)=a+c1 で，要素システムが入力の２倍を次状態とするものであるとき，CAST 言語によるモデルは次のようになる（図 3-1-16）．

```
//sim03.set
func([delta1, delta03]);
delta1(c,a)=cc <-> cc:=2*a;
delta03(c1,a)=cc <->
        a1:=a+c1,    //結合関数 h に相当
        cc:=delta1(c1,a1);

inputsequence()=[2,0,0];
initialstate()=0;
delta(c,a)=delta03(c,a);
```

（CAST 言語によるモデル）

```
"I="1" A="2" C="4
"I="2" A="0" C="8
"I="3" A="0" C="16
```

（実行結果）

図 3-1-16 フィードバック結合システム S03 の CAST 言語によるモデル

複合システム S の状態は c であるため，初期状態は initialstate() = 0 となっている．実行結果は，時刻 1 で S1 に 2+0 が入力され次状態が 4 と計算され，次の時刻に 0+4 が入力されたときには次状態が 8 になっている．

練習問題 3　並列＋フィードバック結合（図 3-1-10）のモデル sim04.set を作成し，実行せよ．

（答）

```
//sim04.set
func([delta1,delta2,delta04]);
delta1(c,a)=cc <-> cc:=2*a;
delta2(c,a)=cc <-> cc:=2*a;
delta04([c1,c2],a)=cc <->
        [a1,a2]:=[2+c2,1+c1],    //結合関数 h の具体的表現
        cc:=[delta1(c1,a1), delta2(c2,a2)];
inputsequence()=[0,0,0];
initialstate()=[0,0];
delta(c,a)=delta04(c,a);
```

注意：入力 a はどこへも影響を与えないダミーの変数となっているが，時刻を刻む目的で，（今の所）Simcast では書く必要がある．

練習問題４：下図 3-1-17 のシステムで，結合関数 h が a2=c1, a3=c1 でああるとして，モデルを作成し，実行せよ．また，時間の進み方について考察せよ．

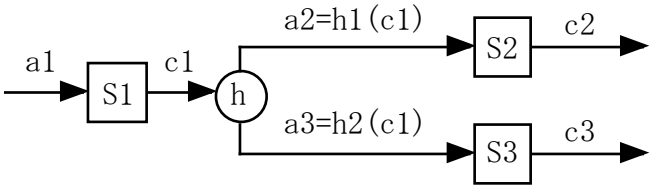


図 3-1-17 直列結合と並列結合を組合わせた複合システム

（答）

```
//sim17.set
func([delta1,delta2,delta3,delta6]);
delta1(c,a)=cc <-> cc:=2*a;
delta2(c,a)=cc <-> cc:=2*a;
delta3(c,a)=cc <-> cc:=2*a;
delta6([c1,c2,c3],a)=cc <->
        cc1:=delta1(c1,a),
        cc2:=delta2(c2,cc1),
        cc3:=delta3(c3,cc1),
        cc:=[cc1,cc2,cc3];
inputsequence()=[1,2,0];
initialstate()=[0,0,0];
delta(c,a)=delta6(c,a);
```

以上，逐次処理システムのモデルについて紹介してきた．次節では並列処理について述べる．ただし，並列結合の場合だけは逐次処理でも並列処理でもモデルは同じとしている（本稿では厳密に区別しないこととする）．

3.1.４　ネットワーク

次に，並列処理システム（concurrent processing system）について述べる．並列処理システムとは，結合の形態にかかわりなく，「**すべての要素がいつでも自律的に動いている**」ような複合システムのことを言う．例えばインターネットには無数のコンピュータが結合している．各コンピュータは自律的に活動しているので，それら自律的要素がつながったネットワークは並列処理システムである．

ただし，本書では，フィードバック結合，直列結合，並列結合，あるいはそれらの組合わせのどの結合形態であっても，並列処理を行なう複合システムを**ネットワークシステム（あるいは単に，ネットワーク**）とも呼んでいる．

(1)　（並列処理）直列結合のモデル

まず例として，3 つの要素が直列に結合した複合システムを考え，そのすべての要素がいつでも自律的に動いているものとする（並列処理システム）．ただしここでは特別に，各要素が「**同じタイミングで**」動いている（同期型）とすると，このシステムの動作は次の図のようになる（図 3-1-18）．前項の逐次処理（図 3-1-12）とは，まったく異なる動作をしていることに注意されたい．

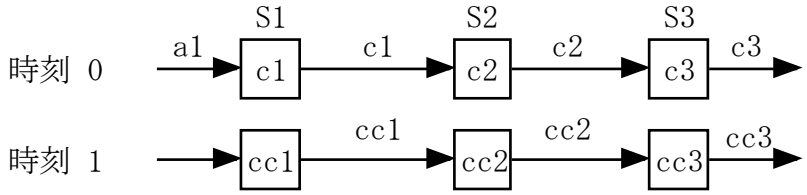


図 3-1-18　並列処理の動作

このとき，全体システム S4 は，(c1,c2,c3)を状態とする，ひとつの状態機械である．なぜなら，S4 の次状態 cc は，関数

δ4((c1,c2,c3),a) = cc <->

a1=a かつ a2=c1 かつ a3=c2 かつ

cc = (δ1(c1,a1), δ2(c2,a2), δ3(c3,a3)))

で計算できるからである．単に，それぞれの状態遷移関数を並べればよい．したがって，S4 は状態機械であり，そのモデルは< δ4, (0,0,0) > である．そこで，CAST 言語によるモデルは次のようになる（図 3-1-19）．

```
//sim13.set
func([delta1,delta2,delta3,delta4]);
delta1(c,a)=cc <-> cc:=2*a;
delta2(c,a)=cc <-> cc:=2*a;
delta3(c,a)=cc <-> cc:=2*a;
delta4([c1,c2,c3],a)=cc <->
        [a1,a2,a3]:=[a,c1,c2],
        cc:=[delta1(c1,a1), delta2(c2,a2), delta3(c3,a3)];
inputsequence()=[1,0,0];
initialstate()=[0,0,0];
delta(c,a)=delta4(c,a);
```

```
"I="1" A="1" C="[2,0,0]
"I="2" A="0" C="[0,4,0]
"I="3" A="0" C="[0,0,8]
```

（CAST 言語によるモデル）　　（実行結果）

図 3-1-19　並列処理直列結合システム S4 の CAST 言語によるモデル

全体システム S4 の状態は c=(c1,c2,c3)であるため，初期状態は initialstate() = [0,0,0] とした．実行結果をみると，時刻 1 で S4 に 1 が入力され次状態が[2,0,0]と計算され，次の時刻に 0 が入力されたときには次状態が[0,4,0]になっている．全体としては時刻が 3 進んでから最終出力 8 を得る．ここが逐次処理と異なる点である．逐次処理システム（図 3-1-13）では，時刻が 1 進めばすでに出力が得られていた．

練習問題５：各要素システムが自律システムであるとして，図 3-1-17 の複合システムのモデルを作成し，実行せよ．ただし，h1(c1)=c1, h2(c1)=c1 とする．

（答）

```
  //sim195.set
  func([h,delta1,delta2,delta3,delta6]);
  h(c1)=[a2,a3] <-> [a2,a3]:=[c1,c1];
  delta1(c,a)=cc <-> cc:=2*c;
  delta2(c,a)=cc <-> cc:=2*c;
  delta3(c,a)=cc <-> cc:=2*c;
  delta6([c1,c2,c3],a)=cc <->
          [a2,a3]:=h(c1),
          cc:=[delta1(c1,a), delta2(c2,a2), delta3(c3,a3)];
  inputsequence()="d";
  times()=9;
  initialstate()=[0,0,0];
delta(c,a)=delta6(c,a);
```

(2) ネットワークの一般的構造

実は，状態機械を要素とするすべてのネットワーク（並列処理システム）は，図 3-1-20 のように，並列結合とフィードバック結合の組み合わせで図式化することができる．たとえば，直列結合システム S4（図 3-1-18）でさえも，横に並んでいる要素システムを縦にならべて，図 3-1-20 のように書き直すことができる（詳細は次項 3.2 で述べる）．

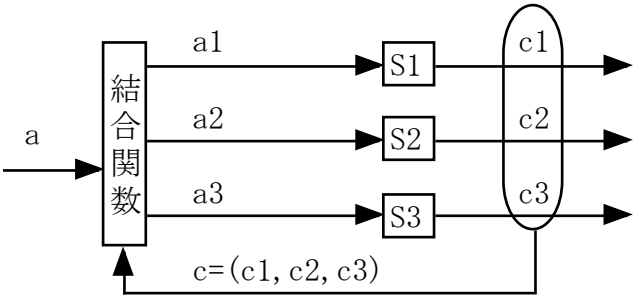


図 3-1-20　ネットワークシステムの一般的構造
（結合関数は h(a,c)=(a1,a2,a3)である）

結合の形態（直列，並列，フィードバック）は結合関数 h で記述できる．前項の直列結合 S4 の場合には，図 3-1-19 中の式 [a1,a2,a3]:=[a,c1,c2] が結合関数 (a1,a2,a3) = h(a,c) に相当するものである（具体的な形である）．また，静的システムも結合関数に吸収することにする．

もちろん，逐次処理と並列処理が混合しているシステムもある．そのときは，まず逐次処理システムを「ひとつ」にまとめ，自律的な要素として扱えば，全体として並列処理システムとして見なすことも可能である．

さて，ネットワークの種類に関しては，結合の形態よりも，むしろデータの流れや時間の観点から，固定結合網—可変結合網，同期型—非同期型の 4 種類に分類することができる．固定結合網とはデータの流れが固定的であること，同期型とはすべての要素が同じタイミングで動作しているネットワークのことである．同期型固定結合網に関しては，練習問題 3 を拡張していけばよく，次節「状態機械の多層ネットワーク」に詳述してあるので，そちらを参照されたい．ここでは，それ以外の種類について述べる．

(3) 固定結合網と可変結合網

例えば，テレビ受像機の中の電気回路群は結合形態が固定されたものであって，データ（信号）の流れが常に同じ結線上を流れている．図 3-1-20 の結合関数 h がすべての要素を互いに結合させているとは限らず，しかも固定的な場合に相当する．これを**固定結合網**と呼ぶ．固定結合網のモデルは，並列結合とフィードバック結合の組み合わせであるから図 3-1-10 のモデルを拡張していけばよい. 例えば図 3-1-18 の並列処理直列結合システムは(同期型）固定結合網のモデルである．

一方，インターネットでは，有線や無線でコンピュータが繋がっており，すべてのコンピュータがコミュニケーションできる可能性があるとしても，実際にデータをやりとりするコンピュータは変化している．例えば，すべての国民が常に総理官邸のホームページを見ているとは限らない．図 3-1-20 の結合関数 h がすべての要素を互いに結合させており，しかしデータの宛先が変化するような場合に相当する．これを**可変結合網**と呼ぶ.

可変結合網では，インターネットの ip アドレスのように，出力（データ）に宛先を付加して考えることになる．したがって，モデルとしては，各時刻ごとにすべての要素の出力データを集め，各データの宛先の要素システムへそのデータを入力するように結合関数 h を構成すればよい.

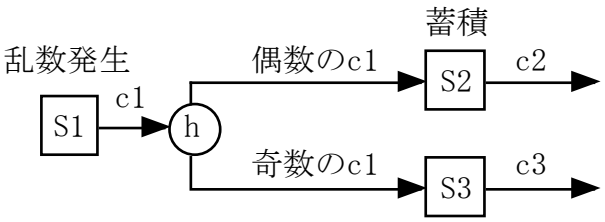


図 3-1-21　可変結合網の例 S6

可変結合網の例として，上のようなシステム S6 を考える．要素システム S1 は 1 から 10 までの乱数 c1 を発生させるシステムで，S2, S3 は受け取った c1 を 1 時刻だけ蓄積するシステムである．しかし，c1 が偶数のときだけ S2 に入力され，c1 が奇数のときには S3 に入力される．入力を受けなかった要素システムの状態は変化しないものとする．データの流れが変化するので S2 は可変結合網である．

これをモデル化したい．まず，入力がなくても時間は進むので，0 を特別な値とし，入力がないことを表すとする．結合関数 h を定義すると，

$$h(c1) = (a2,a3) \quad <-> \quad (a2,a3) = \begin{cases} (c1,0) & \text{if} \quad c1 \text{ が偶数} \\ (0,c1) & \text{if} \quad c1 \text{ が奇数} \end{cases}$$

となる．ここでは，データ c1 が偶数か奇数かが宛先アドレスに相当し，偶数奇数に対応して c1 が入力される要素システムが決まっている．もちろん，要素システム S2, S3 に入力がないとき（入力=0 のとき）の状態遷移を記述しておく必要があることに注意する．すると結局，複合システム S6 は練習問題 5 と同じタイプになる．その理由は，入力がないことに特別な記号を割り当てたことにある．CAST 言語によるモデルは次のようになる（図 3-1-22）．

```
//sim15.set
func([h,delta1,delta2,delta3,delta6]);
h(c1)=[a2,a3] <->
        ((c1%2)=0)->([a2,a3]:=[c1,0]) otherwise ([a2,a3]:=[0,c1]);
delta1(c,a)=cc <-> myrandom(r), cc:=floor(10*r +1);
delta2(c,a)=cc <-> (a=0)->(cc:=c) otherwise (cc:=a);
delta3(c,a)=cc <-> (a=0)->(cc:=c) otherwise (cc:=a);
delta6([c1,c2,c3],a)=cc <->
        [a2,a3]:=h(c1),
        cc:=[delta1(c1,a), delta2(c2,a2), delta3(c3,a3)];
inputsequence()="d";
times()=9;
initialstate()=[0,0,0];
delta(c,a)=delta6(c,a);
```

図 3-1-22　可変結合網 S6 の CAST 言語によるモデル

ここでは，結合関数 h を明示的に定義したので，ユーザ関数宣言 func([h])をしている．1 から 10 までの一様乱数を発生させるために，述語 myrandom と関数 floor を使用した．要素システム S1 への入力はないはずだが，時間を進ませるためにダミーの"d"を delta1(c1,a) の a に 9 回入力している．実行結果は図 3-1-23 に示す．

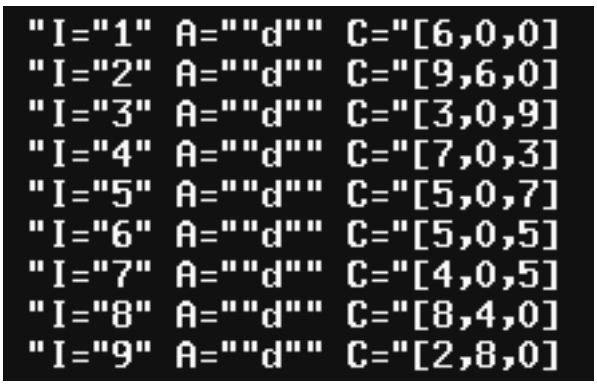


図 3-1-23　sim15.set の実行結果

確かに，状態 c1 が偶数のとき次の時刻に c1 が c2 に代入され，状態 c1 が奇数のときに次の時刻に c1 が c3 に代入されている．

繰り返すが，入力がないことに特別な記号を割り当てることがアイデアである（他のアイデアもあるかもしれない）．その特別な記号を受け取ったら，入力がないときの処理を行なえばよい．モデルの中では入力はあるのである．その結果，上記の結合関数 h のように工夫すれば，可変結合網は普通の並列処理固定結合網と同じようにモデル化することができる．

(4) 同期型と非同期型

全体システムが同期型であるとは，各要素システムが「常に同じタイミング」で動作していることをいう．毎回の単位時刻ごとにすべての要素が何らかの処理を行なうのである．そうでないシステムを非同期型という．例えば，第 4.1 項の直列結合システムは同期型固定結合網を扱っていた．言語 CAST は，基本的に同期型ネットワークだけをモデル化することができる．

したがって，非同期ネットワークを構成するためには，基本的には同期型であるが見かけ上は非同期型のように動作しているように工夫が必要である．通常は，

1）シミュレーションの単位時間を短く設定する．

2）入力がないときの処理（状態が不変）を記述しておく．

が考えられる．単位時間を短く設定しておけば，要素 1 が入力 1 を受け取り処理を開始するが，要素 2 は処理を開始しないことにできる（しかし，実は要素 2 は無入力処理を行なっている）．したがって，CPU を無駄に使用していることになる．CPU の浪費が過酷なものになる場合は，Simcast をあきらめて別の言語を使用する必要がある．

以上，Simcast において，状態機械（state machine）が結合したシステムをモデル化する方法や考え方を紹介した．あわせて，それらの CAST 言語によるモデルを紹介した．非同期可変ネットワークのモデリングには工夫が必要であるが，Simcast は状態機械を要素とするネットワークをそれなりにモデル化できる．

3.2 状態機械の多層ネットワーク

ここでは，複数の単層ネットワークを組合せて多層的にネットワークを構成するときのモデルを例示する．特に，単層ネットワーク（同期型固定結合網）の一般構造は重要である．

3.2.1　背景となる原理

背景となる基本的な原理は,「複数の状態機械が結合したシステムは，全体として一つの状態機械である」ということである．この原理により，重層的なネットワークの段階的なモデル化が，同じ単純な方法で一貫してできるようになる．複数のシステムが結合したネットワークを多層的に組合わせた複雑なシステムのシミュレーションを行なう方法を概観する．

これが理解できれば，次の段階に簡単に進むことができるだろう．次の段階とは，状態機械ではなく, 複数のオートマトンが結合したネットワークをモデル化することである（第3.5 節）．したがって，本稿は，そのための準備であると位置づけることができる．

3.2.2　単層ネットワーク

単層ネットワーク（同期型固定結合網）の一般的構造と CAST 言語によるモデルの実装構造，および例として循環するネットワークのモデルを示す.

(1) 循環するネットワークのモデル

単層ネットワークの例として，図 3-2-1 のような循環するネットワークを考える．このシステム S4 は，単純なフィードバックシステムではない．S4 は，３つの動的システム（要素システム S1, S2, S3）と１つの静的システム（加算要素(+)）を含む，ネットワークシステムである．以下では，Si のモデルを ＜$\delta\, i$, 0＞ とする（i=1,2,3).

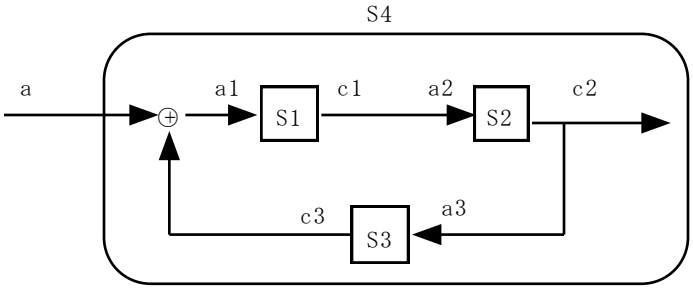


図 3-2-1　循環するネットワーク S4

次の図 3-2-2 は，図 3-2-1 の配置を換え，動的システム S1, S2, S3 が並列に並ぶように描き直したものである．このように配置すると，要素システムの結合が図の左側にひとつにまとまり，後述するような一貫した理解ができるようになる．ただし，出力は状態の組 (c1,c2,c3) であるとしている点に注意されたい.

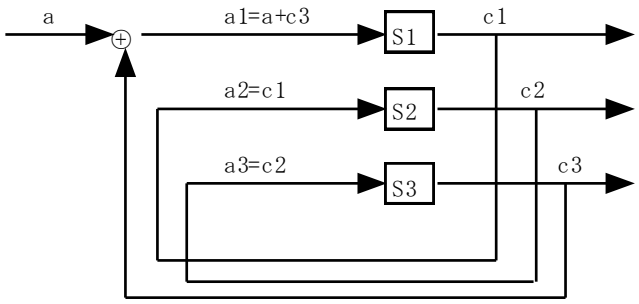


図 3-2-2　循環するネットワーク S4 の構造

このとき，全体システム S4 は，(c1,c2,c3)を状態とする，ひとつの状態機械である．なぜなら，S4 の次状態 cc は，関数

δ4((c1,c2,c3),a)=cc <->
(a1,a2,a3) = (a+c3,c1,c2) かつ
cc = (δ1(c1,a1), δ2(c2,a2), δ3(c3,a3)).

で計算できるからである．したがって，S4 は状態機械であり，そのモデルは< δ4, (0,0,0) > である．

この循環するネットワーク S4 を，具体的に CAST 言語でモデル化すると，次のようになる．

```
//sim41.set
func([delta1 ,delta2 ,delta3 ,delta4]);
delta1(c,a)=cc <-> cc:=a;
delta2(c,a)=delta1(c,a);
delta3(c,a)=delta1(c,a);
delta4(c,a)= cc <->
        [c1,c2,c3]:=c,
        [a1,a2,a3]:=[a+c3,c1,c2], //  入力結合 in(a,c)
        cc:=[delta1(c1,a1),delta2(c2,a2),delta3(c3,a3)];

inputsequence()=[1,0,0,0,0];
initialstate()=[0,0,0];
delta(c,a)=delta4(c,a);
```

図 3-2-3　単層ネットワーク S4 の CAST 言語によるモデル

このネットワークは３つの要素をもつシステムなので，状態は c=(c1,c2,c3)である．そのため，初期状態は initialstate() = [0,0,0] となっている．モデル理論アプローチによるシミュレーション実行環境 simcast では，ひとつの状態機械モデルしか実行できないが，全体システム S4 = < δ4, (0,0,0) > がひとつの状態機械であるので，これで動かすことができる．

```
"I="1" A="1" C="[1,0,0]
"I="2" A="0" C="[0,1,0]
"I="3" A="0" C="[0,0,1]
"I="4" A="0" C="[1,0,0]
"I="5" A="0" C="[0,1,0]
```

図 3-2-4　モデル sim41.set の実行結果

実行結果は図 3-2-4 に示した．確かに，時刻 1 に入力された 1 がいつまでも S4 内を循環している様子が分かる．

(2) 単層ネットワークの実装構造

一般的には，結合の仕方として，直列結合，並列結合，静的システムによるフィードバック結合の３種類があるが，複数の状態機械をどのように結合しても，全体としてはやはり状態機械となることが知られている．また，そのときの全体システムの状態は，各要素の状態を並べた組（c1,c2,c3）であることも重要である．

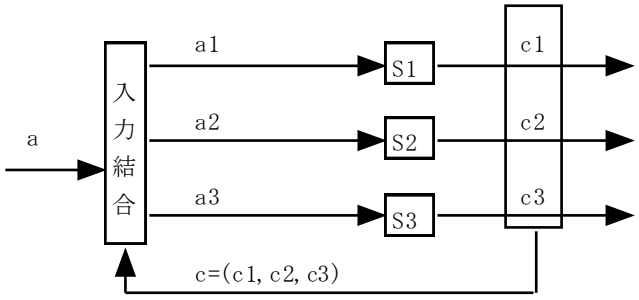


図 3-2-5　状態機械ネットワークの一般的構造

上の図 3-2-5 は，状態機械を要素とする単層ネットワークの一般的構造である．外部からの入力 a と状態 c と要素システムへの入力 (a1,a2,a3) の間の関係を「入力結合」と呼び，関数で in(a,c) = (a1,a2,a3)と書くことにしよう．このとき，全体システムの状態遷移関数は，次の式で表せる．

δ (c,a)=cc <->
　　(c1,c2,c3)=c かつ
　　(a1,a2,a3) = in(a,(c1,c2,c3)) かつ
　　cc = (δ 1(c1,a1), δ 2(c2,a2), δ 3(c3,a3)).

したがって，単層ネットワークの CAST 言語によるモデルの一般的な形は次のようになる．

```
delta(c,a)= cc <->
        [c1,c2,c3]:=c,
        [a1,a2,a3]:=in(a,c1,c2,c3),
        cc:=[delta1(c1,a1),delta2(c2,a2),delta3(c3,a3)];
```

図 3-2-6　CAST 言語による単層ネットワークの実装構造

図 3-2-6 は３つの状態機械を要素にもつネットワークの場合のモデルであり，テンプレートとして使用することができる．要素システムの数が変われば delta*i* を増減させ，入力結合 in()を状況に応じて変えれば良い．循環するネットワーク S4 の例では，

in(a,c)= (a1,a2,a3) <-> (a1,a2,a3) =(a+c3,c1,c2)

であった.

3.2.3　多層ネットワーク

前節で提案した方法で，多層ネットワークをモデル化する．例として，次のネットワーク S6 を CAST 言語でモデル化する．

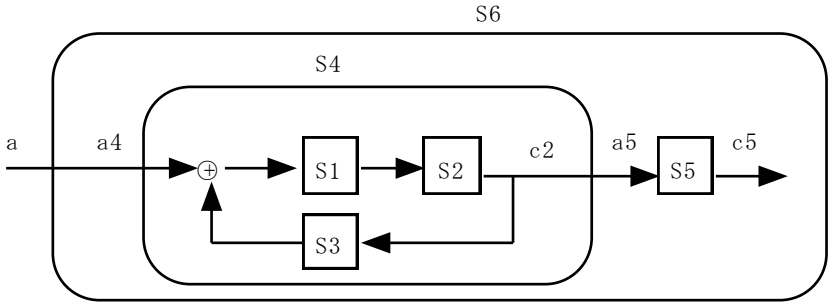


図 3-2-7　多層ネットワーク S6

システム S6 は，単層ネットワーク S4 とひとつの状態機械 S5 が直列結合した多層ネットワークである．しかし，いまや S4 もひとつの状態機械であるとみなせるので，結果として，システム S6 を２つの状態機械が結合している単層ネットワークとみなすことができる（図 3-2-7）．

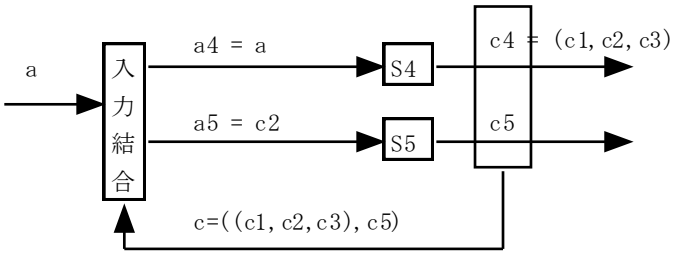


図 3-2-7　多層ネットワーク S6 の構造

したがって，前節の単層ネットワークと同じように，モデル化することができる．

```
//sim47.set
func([delta1,delta2,delta3,delta4,delta5,delta6]);
delta1(c,a)=cc <-> cc:=a;
delta2(c,a)=delta1(c,a);
delta3(c,a)=delta1(c,a);
delta4(c,a)= cc <->
        [c1,c2,c3]:=c,
        [a1,a2,a3]:=[a+c3,c1,c2],
        cc:=[delta1(c1,a1),delta2(c2,a2),delta3(c3,a3)];
delta5(c,a)=delta1(c,a);
delta6(c,a)= cc <->
        [c4,c5]:=c,
        [a4,a5]:=[a,project(c4,2)],   //入力結合 in(a,c)
        cc:=[delta4(c4,a4),delta5(c5,a5)];

inputsequence()=[1,0,0,0,0];
initialstate()=[[0,0,0],0];
delta(c,a)=delta6(c,a);
```

図 3-2-8　多層ネットワーク S6 の CAST 言語によるモデル

このネットワークは２つの要素 S4, S5 をもつシステムだが, S4 の状態が c=(c1,c2,c3)で, S5 の状態が c5 であるため，全体としての状態は，c=((c1,c2,c3),c5)となる．そのため，初期状態は initialstate()=[[0,0,0],0]; となっている．また，要素システム S2 が S5 に直列結合しているので，a5=c2 でなければならない．そこで，入力結合 in(a,c)は [a4,a5]:=[a, project(c4,2)] となっている．実行結果は図 3-2-9 に示した．

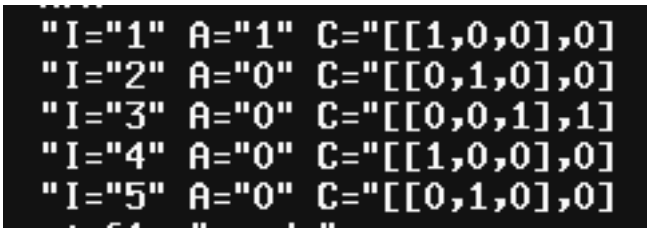


図 3-2-9　モデル sim47.set の実行結果

（宿題２：次の図 3-2-10 のように，外部からの入力を受け付けない閉じたシステム S6 を作成し，実行せよ．ただし，システム S1 の初期状態だけ 1 であるとする．Hint どんな入力に対しても同じように動くことを確認せよ.）

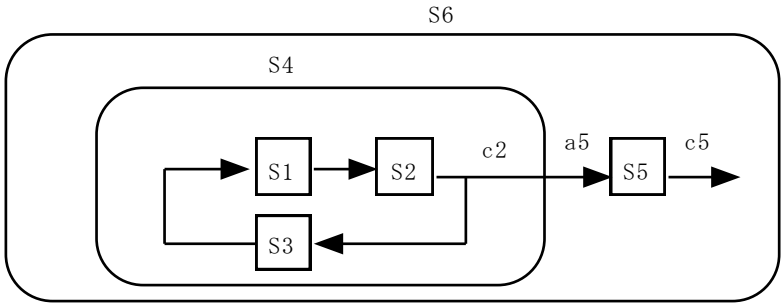


図 3-2-10 閉じたネットワーク

3.2.4　開発手順

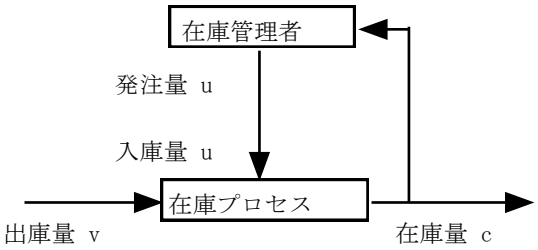
以上，モデル理論アプローチによるシステム開発環境 Simcast において，状態機械を要素とする単層ネットワークを積み重ねて，段階的に多層ネットワークをモデル化することができることを紹介した．ただし，同期型固定結合網に限っている．

段階的モデル化における設計技法の主要点をまとめると，
１）単層ネットワークに対しては，要素の状態機械（state machine）を並列に並べた図 3-2-5 を考え，図 3-2-6 のようにモデル化する（第 2 項）.
２）要素システムの結合関係と静的システムは，入力結合 in( )に含める.
３）多層ネットワークに対しては，要素がネットワークであっても，それをひとつの動的システム（状態機械）とみなして，単層ネットワークと同様にモデル化する（第 3 項）.
４）以上を繰り返して，複雑な多層ネットワークのモデルを構成していく.
となる.
この方法でモデル化すれば，一貫した視点で多層ネットワークのモデルを構成していくことが可能である.

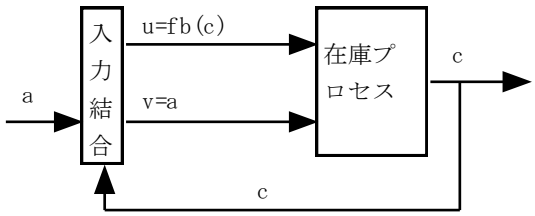
3.2.5 在庫管理のモデル

静的システムは入力結合 in( )で処理する．実際，システム S4（図 3-2-7）中の静的システム（加算要素(+)）は，モデル（図 3-2-9）の中では入力結合 [a1,a2,a3]:=[a+c3,c1,c2] として処理している．

ここでは，もうひとつの例として，即納の在庫管理システムを考える（図 3-2-11）．



付図 3-2-11　在庫管理モデル



付図 3-2-12　自己フィードバック

在庫プロセス S1 のモデルは，

delta1(c,[u,v])=cc <-> cc:=c+u-v;

である．状態 c はその日の朝の在庫量，入力 [u,v] はその日の夕刻までの入庫量と出庫量を表す．したがって，翌日の朝の在庫量 cc は cc:=c+u-v である．

在庫管理者は毎朝在庫量をチェックする，在庫管理者の管理ルール fb を，ここでは，

fb(c)=u <-> (c<=3) -> (u:=6) otherwise (u:=0);

とする．つまり，在庫量が３個以下であったら，６個の商品を発注する．即納であると仮定しているので，商品は発注と同時に納品（入庫）される．遅れがないので，静的システムによる自己フィードバック（図 3-2-12）である．

この在庫管理システムをモデル化してみよう．いま，u=fb(c)を delta1 に代入し，全体システムの状態遷移関数を求めると，

delta(c,v)=cc <-> cc:=c+ fb(c)-v;

となるので，単純に考えれば，図 3-2-13 のようになる．実際に Simcast で実行することができる．

```
//sim44.set　フィードバック（即納の在庫管理システム）
func([fb]);
fb(c)=u <-> (c<=3) -> (u:=6) otherwise (u:=0);

inputsequence()=[1,1,1,1,2,2,2,2,2,2,2,2,2,2,];
initialstate()=6;
delta(c,a)=cc <-> cc:= c+fb(c)-a;
```

図 3-2-13　CAST 言語による即納の在庫管理モデル

しかしながら，本稿で紹介した方法（第 3.2.4 項）では，まず在庫プロセス S1 をモデル化しておき，次に自己フィードバックを付加した「新しい」システム S2 を構成するという手順で行なうことになる．CAST 言語によるモデルは図 3-2-14 のようになる．在庫管理ルールは静的システムなので，入力結合 [u,v]:=[fb(c),a2] の中に fb(c)を記述している．

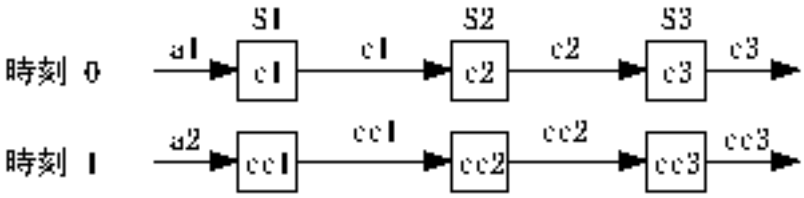
```
//sim45.set　フィードバック（即納の在庫管理システム）
func([delta1,fb,delta2]);
delta1(c,[u,v])=cc <-> cc:=c+u-v;
fb(c)=u <-> (c<=3) -> (u:=6) otherwise (u:=0);　//在庫管理ルール

delta2(c,a)= cc <->
      [u,v]:=[fb(c),a],                    //入力結合
      cc:=delta1(c,[u,v]);

inputsequence()=[1,1,1,1,2,2,2,2,2,2,2,2,2,];
initialstate()=6;
delta(c,a)=delta2(c,a);
```

図 3-2-14　シミュレーションのための在庫管理モデル

このように記述する理由は，常に同じパターンでモデル作成を行なうほうが覚えやすいし，さらに新しい要素システムを付加するときに簡便だからである．例えば，納品に遅れをともなうシステムなら，発注から入庫までの間に問屋のシステムを付加しなければならない．問屋に相当する遅れシステム S2 を作成し，S1 と S2 からなる単層ネットワークを構成することになる．

（宿題３：即納でなく１日遅れの在庫管理システムの図を描き，CAST 言語によるモデルを作成し，動かしてみよ．Hint 図 7,8,10）

3.3 Hint: 多数のオートマトンを結合する



例題として，1,000 個の状態機械 S1, S2,..., S1000 を直列結合し並列処理するシステムをモデル化する。

(1) アイデア

変数 n を加えて関数の番号を表す. 状態機械 Sn の状態遷移関数を delta(n,c,a)で定義する.
関数 defList で 1000 個の関数の計算を行なう（次状態を計算する）.

(2) 番号リスト genIndex(1,1000)をインデックスに使用する方法

```
//manyfunc5a.set
func([delta1,deltaA]);
delta1(n,c,a)=cc <-> cc:=a;
deltaA(c,a)=cc <->
       Ps:=append([a],c),
       cc:=defList(p(y,n,[Ps]),
       member(n,genIndex(1,M.g)) );
p(y,n,[Ps]) <->
[an,cn]:=project(Ps,["r",n,n+1]),y:=delta1(n,cn,an);
preprocess() <-> M.g:=1000;
inputsequence()=0; times()=10;
initialstate()=c <-> c:=append( [1],constantlist(0,M.g-1) );
delta(c,a)=deltaA(c,a);
```

上記ユーザモデルの 3 行目の delta1(n,c,a) の部分が複数の関数（状態遷移関数）を定義している．変数 n は状態機械の番号である．ひとつの式であるが，1,000 個の関数が定義されていると思っておけばよい．その次の，deltaA(c,a)が 1000 個の状態機械を直列結合した全体システム S の状態遷移関数である. ただし, その状態変数は c=[c1,c2,...,c1000] である. 次状態 cc を計算するために，まずリスト Ps=[a,c1,c2,...,c1000]を作成する．状態機械 Sn の状態は cn, 入力は an=cn-1 であるから，関数 project()を使って Ps から 2 つづつ要素を抽出すれば delta(n,cn,an)を計算することができる.

全体システム S はひとつの状態機械であるから Simcast で実行することができる．コン

パイルして実行すると10回の状態遷移で15秒かかった. 上図はその途中を表示している. 全体システムの初期状態が c=[1,0,0,...] だったので，1 回目の状態遷移で c=[0,1,0,...] と変化している．1 が隣の状態機械に移動しているようすがわかる.

(3) 高速化の方法

上記のモデル manyfunc5a.set において，times() =1000 に変更すれば 1000 回の状態遷移を行なうはずであるが，推定 25 分（1500 秒）の実行時間がかかるだろう．計算が遅い理由は，関数 defList の中で project()を使用しているからである．そのため関数 project は，状態遷移 1000 回×計算 1000 回＝100 万回使用 されている. 以下では，関数 project を使わないで高速化する方法を示す.

1) 代入すべき値のリストを作成する方法

defList を使う前に，あらかじめ代入すべき値のリストを作成しておき，それをインデックスとして defList を定義する方法がある.

```
deltaA(c,a)=cc <->
      Ls:=transpose([genIndex(1,M.g),c,deleteList(append([a],c),M.
g+1)]),      cc:=defList(p(y,z,[]),member(z,Ls));  p(y,z,[]) <->
[n,cn,an]:=z, y:=delta(n,cn,an);
```

この場合，Ls=[[1,c1,a], [2,c2,c1],[3,c3,c2],...] というリストを作成しておき，要素[n,cn,an]を取り出しながら delta(n,cn,an)を計算し，並べて，次状態 cc を得ている. 実行時間は，1000 回の状態遷移（times()=1000） で 60 秒であった．耐えられる実行速度になったが快適とは言えない．その理由は多数の要素をもつリストがあるからである．実際，defList のインデッ クスとなる Ls は 3 項目×1000 組＝3000 個の要素をもつリストである. 長いリストの処理には時間がかかる.

2) グローバル変数を利用する方法

第 2.3 節「漸化式を解く」を参考に，グローバル変数を利用すれば，さらに早くすることができる.

```
deltaA(c,a)=cc <->
      n.g:=1, a.g:=a,
      cc:=defList(p(y,cn,[]),member(cn,c) );
      p(y,cn,[]) <-> n:=n.g, y:=delta(n,cn,a.g), n.g:=n+1, a.g:=cn;
```

実行時間は，1000 回の状態遷移で 27 秒であった.

3) スプレッドシートを利用する方法

第 3.4 節「シミュレーションの高速化：グラフ表示の制約」を参考に，スプレッドシートを状態の保存場所にする方法が考えられるが，やらなかった.

(4) 逐次処理の場合

これまでは並列処理の場合を述べてきた．逐次処理の場合は次のようにすればよい.

```
deltaA(c,a)=cc <->
      n.g:=1, a.g:=a,
      cc:=defList(p(y,cn,[]),member(cn,c) );
      p(y,cn,[]) <-> n:=n.g, y:=delta(n,cn,a.g), n.g:=n+1, a.g:=y;
```

変更点は a.g:=y だけである．ひとつの入力 5 に対して，瞬時に 1000 個の要素の状態が 5 になるようすが分かる.

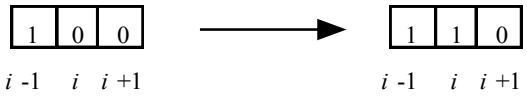
3.4 Hint:シミュレーションの高速化

本節では，例題として，1 次元細胞オートマトン（Cellular Automaton）のシミュレーションモデルを，モデル理論アプローチによるモデル記述言語 CAST を用いて構成し実行する．1 次元細胞オートマトンのシミュレーションでは，全体システムの状態を数値が並んだリストで表し，状態遷移をリストの要素を書き換える計算とする．ここでは，オートマトン 1 回の状態遷移で，200 個の細胞の 1 サイクル（1 世代）の計算を行なう．この例題を通じて，開発実行環境 Simcast におけるシミュレーション高速化ための工夫やグラフ表示の制約を述べることが目的である．

3.4.1 例題：1 次元細胞オートマトン

直線上に並んだ複数の 2 状態オートマトン（正確には 2 状態の状態機械）を考える．各細胞の状態は 0 または 1 である．左から順番に番号をつけ，*i* 番目のオートマトンを細胞 *i*（cell *i*）と呼ぶ．

【システム要素】　各細胞の状態は，左右に隣り合う 2 つの細胞の状態の影響を受けて変化していく．



図表 3-4-1　細胞 *i* の状態遷移

例えば，細胞 *i* の近傍が図 1 左のようであったとき，次の時刻の細胞 *i* の状態は 1 であるなどと遷移するのである．このような規則を，以下では次のように書くことにする．

[[1,0,0]],1]

これは，自分が健康(0)であっても隣の細胞が病気(1)になったら自分も感染して病気(1)になる，などと適当に解釈する．自身を含めた近傍の様子は可能性として 2^3=8 パターンあるので，それぞれの場合に応じて次の時刻の状態を定めることになる．一つの細胞は複数の規則からなる状態遷移関数をもつ．以下では，次の規則を考える．

Rule.g=[[[1,0,0],1], [[0,0,1],1], [[1,1,0],0], [[0,1,1],0]];

【全体システム】　以下では 200 個の細胞を考え，すべての細胞の状態遷移関数は同じであるとする．一般に，状態機械の複合システムは全体としてひとつの状態機械であるから，全体をひとつの状態機械と考えることができる．すると，全体システムの状態は，値が 0 か 1 であるような 200 個の要素からなるリストとなる．状態遷移につれて，そのリストの中の数値が書き換えられていくのである．1 回の状態遷移で 200 個すべての細胞の状態が変化する可能性もある．

上記の全体システムの CAST 言語によるモデルは次のように構成できる．

```
/*cellular02.set ( 1 次元細胞オートマトン）*/
preprocess() <->
        M.g:=200,
        Rule.g:=[ [[1,0,0],1], [[0,0,1],1], [[0,1,1],0], [[1,1,0],0] ];
inputsequence()="d";
times()=1000;
initialstate()=c0 <->
        L:=constantlist(0,M.g),
        [c0]:=replaceMatrix([L],[[1,7,1],[1,140,1]]);
delta(c,a)=cc <->
        Ls:=defList(p(z,i,[c]),member(i,genIndex(2,M.g-1))), //変更チェック
        [cc]:=replaceMatrix([c],Ls);   //変更実施
p(z,i,[c]) <-> N:=project(c,["r",i-1,i+1]), member([N,v],Rule.g), z:=[1,i,v];
```

図表 3-4-2　細胞オートマトンのモデル

全体システムの実行手順は簡単である．まず初期状態 c0 を定めるために，constantlist 関数で 0 が M.g 個（200 個）並んだリスト L を作り，それを行列[L]に変えてから，replaceMatrix 関数で 7 番目と 140 番目だけを 1 に書き換えている．その結果のリスト c0 が初期状態である．実際の全体システムの状態遷移は delta で定義し，変化すべき細胞のリストアップ（変更チェック）と状態（リスト）の変更を実施する．そのため，defList 関数と述語 p を使って，近傍 N と変更規則 Rule.g に従って書き換えるべき細胞の番号と次の状態値の組（[i,v]）をリストアップし，再び replaceMatrix 関数でリストの要素の書き換えを実行する．その結果のリストが全体システムの次状態 cc である．（defList は i を動かして述語 p(z,i,[c])を満足する要素 z を集めてリストを作る関数である．）全体は自律的に動くシステムであるが，強制的に時間を進めるためにダミーの"d"を 1000 回入力している（inputsequence()="d"; times()=1000;）．

このモデル（図表 3-4-2）をモデル理論アプローチによるシミュレーション開発実行環境 Simcast でコンパイルし実行した結果が図表 3-4-3 である．Dialog ウインドウに次々とリストが表示され，状態遷移するようすが分かる．

図表 3-4-3　実行結果

3.4.2 計算処理の制限

細胞数を増やすには，各種の制限に注意する必要がある．

1）変数の制約

Simcast における（正確に言えば，extProlog が処理できる）リストの長さは 4,000 までである．したがって，グローバル変数を M.g=4000 に修正すれば 4 千個の細胞オートマトンを扱うことができるはずであるが，実際には関数 constantlist()がエラーとなる．正常に動くことは M.g=1900 まで確認した．1 次元の細胞オートマトンなら通常 1,900 個を扱う必要はないのでこれで十分である．しかし実行速度は遅くなる．

2）ファイル保存

遷移したすべての状態を，シミュレーション実行中にファイルに書き込むことができる．述語 xwrite()を使用すると，例題では 200 要素のリストが 1000 行並んだファイルが出来上がる．容易に CSV 形式に変換できるので，シミュレーション終了後に，そのデータを何らかの市販の表計算ソフトで読み込んでグラフ化することは可能である．

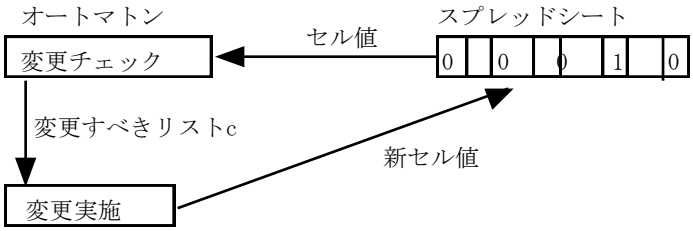
3）実行速度

次にシミュレーションの速度について考える．これらのモデルでは，近傍（左右の細胞）の状態をチェックして変更すべき細胞をリストアップしてから，状態 c（リスト）を書き換えていた．しかしながら，細胞数が増加すると近傍チェックの回数も増加していく．例えば，細胞数 200 個の場合は，（リストの両端はチェックしないので）1 回の状態遷移のために 198 回のチェックを行うことになる．状態遷移を 1,000 回繰り返すなら，198,000 回のチェックを行うことになる．実行時間は 64 秒であった．

実は，背景で動いている extProlog はリストの書き換え速度が遅い．例題では，リストや行列の要素を（project 関数で）読んだり，（replaceList や replaceMany 関数で）書き換えたりしている．そのたびごとにリストや行列を作成するから実行速度は遅くなる．細胞数が増加するとシミュレーション速度が遅くなり，多数の繰り返しには不向きである．その理由は「ひとつの変数 c にすべての細胞の状態を記憶させている」からである．そこで，次節のようにモデル化の大胆な方針変更が必要となる．

3.4.3　シミュレーションの高速化

シミュレーション実行速度を向上させるため，ここでは「ひとつのスプレッドシート（以下 SS）に全細胞の状態を保存する」方法を紹介する．理由は，Simcast では SS の読み書き速度が速いからである．



図表 3-4-4　SS を利用した高速化

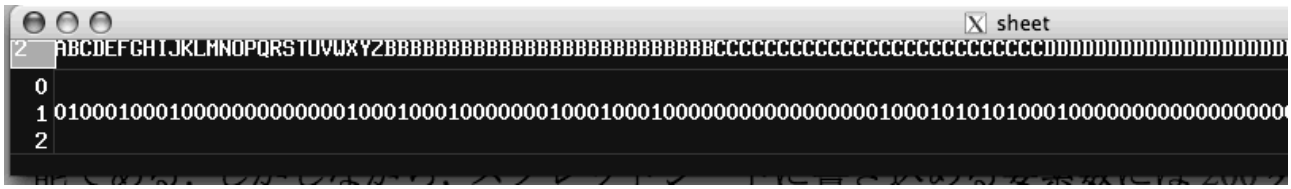
まず，ひとつの変数 c にすべての細胞の状態を保存することをあきらめる．その代わり SS にすべての状態を書き込んでゆく．したがって，オートマトンの仕事は，SS を書き換えることとする（図表 3-4-4）．この方針に従って，例題の細胞オートマトンを，書き直したものを図表 5 に掲載する．

```
//cellular06.set
preprocess() <-> //SS ウインドウを開き，初期値を書き込む
        M.g:=200,
        Rule.g:=[ [[1,0,0],1], [[0,0,1],1], [[1,1,0],0],[[0,1,1],0] ],
        getWp("sheet",Wp),
        (Wp>0)->(Wp.g:=Wp)
        otherwise (
                openlocalsheet("",0,1,150,3,x1,y1,1,1,1,Wp1), Wp.g:=Wp1,
                writess(Wp.g,"r", 1,1, constantlist(0,M.g)),
                ssw(Wp.g,1,70,1), ssw(Wp.g,1,140,1)     );
inputsequence()="d";
times()=1000;
initialstate()=c <-> c:=[];
delta(c,a)=cc <->
        cc:=defList(p1(z,i,[]),member(i,genIndex(2,M.g-1))),  //変更チェック
        replacess(Wp.g,cc);  //SS に書き込み
p1(z,i,[]) <-> [N]:=projectss(Wp.g,1,i-1,1,i+1), member([N,v],Rule.g), z:=[1,i,v];
```

図表 3-4-5　スプレッドシートを「状態」の記憶場所として利用するモデル

図表 3-4-5 では, 実行前の準備 proprocess として 3 行 150 列表示の SS ウインドウを開き，SS の第 1 行に初期値としてセル(1,70)とセル(1,140)に 1 を書き込んでいる．これが全体システムの初期状態に相当する．状態遷移の定義 delta では，まず SS から近傍の様子 N を読み込み，規則 Rule.g に従ってチェックし，変化すべき細胞をリストアップしている（変数 cc）．次に変更（SS に書き込み）を実施している．

これを実行すると，図表 3-4-5 の 1000 世代分の実行時間は 22 秒であった．前回の 3 分の 1 である．また，遷移する各時点の状態がスプレッドシートに書き込まれるので，シミュレーションの状況がいくぶん分かるようになった（図表 3-4-6）．



図表 3-4-6　セル幅 1 のスプレッドシートを「状態」として利用する

さらに高速化を望むなら，述語 replacess(Wp.g, cc)を使わず，関数 defList を使えばよい．

Ls:=defList(p2(z,[r,i,v],[]),member([r,i,v],cc));　//SS に書き込み

p2(z,[r,i,v],[]) <-> ssw(Wp.g,r,i,v);

この場合は，1000 回で 15 秒であった（付録 3-4-1 も参照のこと）．

このモデルでは，変化すべき細胞のセル位置(1,i)と値 v のリスト[1,i,v]を記憶する場所としてオートマトンの状態変数 c を利用している．もしもこれをファイルに保存しておけば，初期状態からの再生も可能である．一方，細胞オートマトン全体の真の状態は SS そのものであり，initialstate や delta の仕事は SS を規則に基づいて書き換えることである．これは典型的な手続き型のプログラミングスタイルであり，もはやモデル理論アプローチの範囲を逸脱しているとも言える．その意味でこれを裏技というべきかも知れない．ただし，CAST 言語の文法（構文）に従ってモデルを作成していることは確かである．

3.4.4　データ表示の限界

研究や教育のためには処理データのビジュアルな表示が必要である，ここではデータ表示の限界を考察する．

1）スプレッドシート

処理できるスプレッドシートのサイズは「600 行×200 列まで」である．しかしサイズを大きくすると狭いパソコン画面からウインドウがはみ出すことになる．述語

openlocalsheet(Name,WinY,WinX,Y,X,Y0,X0,NotMove,Map,Len_cell,Wp)

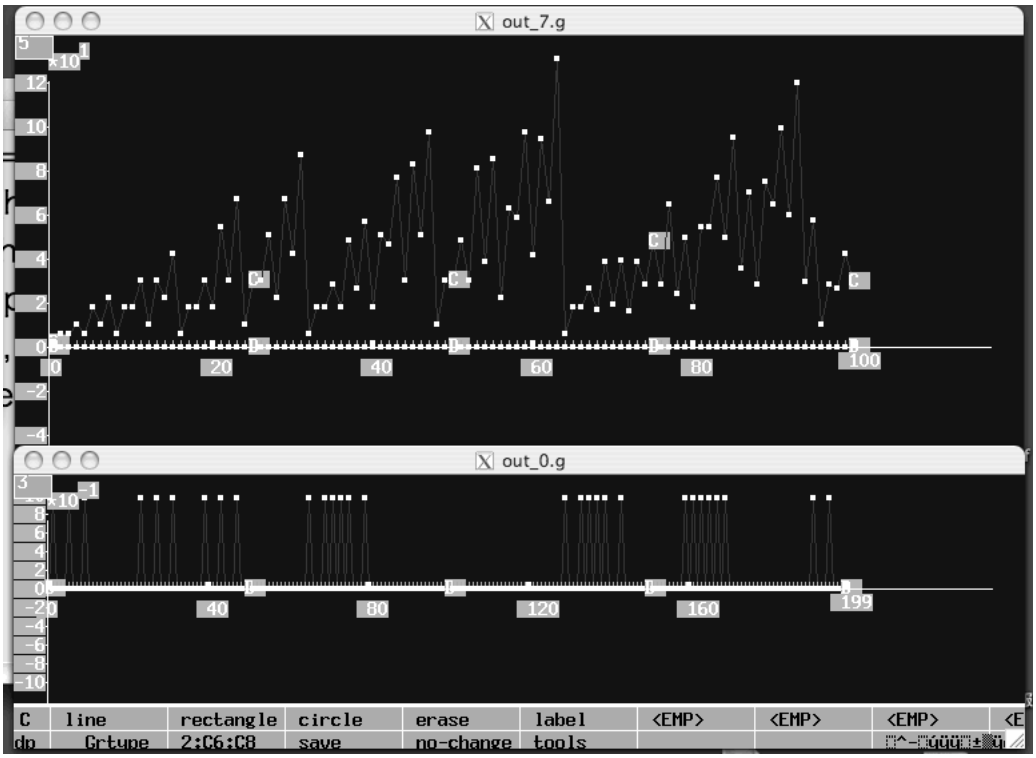
でセル幅 Len_cell を指定できるので，セル幅を 1 にすれば，12 インチディスプレーでも 40 行×150 列程度はリアルタイムで表示できる．

例題の場合，細胞数が 200 を超えると，折り畳んで次の行に書き込む必要がある．番号 i を $200k+l$ に変換すればよい．600×200 = 12 万であるから，SS は十分な広さを持っている．

2）グラフとスプレッドシートの混在

付録 3-4-1 に show 系の述語でグラフ表示を行うユーザモデルを掲載した．図表 3-4-5 に述語 showHistory()を加えたものである（ただし，高速化のため replacess を使わず defList を使っている）．ここでは，SS は状態の表示を担わず，単に状態の保存場所として確保しているにすぎない．その代わり，述語 show1 によって状態をグラフ表示することとした．1000 回の状態遷移を実行するのに 33 秒かかった．グラフ表示には多少の負荷がかかる．

【制限】データをグラフ表示すると，グラフウインドウとスプレッドシートが混在することになる．Simcast では混在は可能である．しかし，混在させると SS のセル幅指定がデフォールトのセル幅 11 に戻るのである（改善は今後の課題である）．



図表 3-4-7　グラフによるデータ表示

付録 3-4-1 のモデルをコンパイルして実行すると，図表 3-4-7 のように，シミュレーション実行中に刻々と状態が変化していく様子が観察できる．下のウインドウ out_0.g には 200 個の細胞のその時点の状態が表示されている．横軸は細胞番号 *i*=1,2,3,...,200 が，縦軸に状態 0,1 がグラフ化されている（関数 show1 による）．上のウインドウ out_7.g には，各時点で状態 1 の細胞数が表示されている．横軸は時間で，縦軸は状態 1 の細胞数である．

グラフウインドウに表示できる点の個数に制限がある．簡便な show 系のグラフ表示の述語 show1 や show2 では，400 個の点までは正常に表示できるが，それ以上は無理である．細胞数が多いときや繰り返しの回数が多いときは，最新の 200 個だけを表示するなど工夫が必要である（計算そのものは細胞数 1,900 個まで正常に実行される）．

一方，座標軸を非表示にし xwrite を使って描画する wopen 系のグラフ表示の述語では，多数の点が打てるので問題はない．横軸をセル番号，縦軸を時刻にして下に伸びていくようにモデル化することも可能である．

以上，計算はできるがデータ表示の制限があることを見た．したがって，200 以上の細胞数のシミュレーションモデルを作成するときの方針としては，最初に細胞数 M.g = 200 でモデルを作成し，グラフ表示をさせながらバグのチェックなどを行い，次にグラフ表示を外し，M.g を 200 以上に設定し，実行することになる．その際，ファイルに「変化した状態 cc」のみを保存しておき，シミュレーション実行後に市販ソフトでグラフ化することになろう．

以上，1 次元細胞オートマトンのモデルを CAST 言語を用いて構成し，モデル理論アプローチによるシミュレーション開発実行環境 Simcast で実際に実行した．また，処理限界

やその回避方法としてスプレッドシートを利用した高速化の工夫について紹介した.
　一応, グラフ表示しなければ多数の細胞数でもシミュレーションは可能である. しかし, もしもグラフ表示するならば，その表示限界はある．グラフ表示しつつモデルを作成し, 実際のシミュレーションではグラフ表示をはずすことになろう.

付録 3-2-1　グラフによるデータ表示

```
//cellular07.set
preprocess() <-> //SS ウインドウを開き，初期値を書き込む
        M.g:=200,hist.g:=[],
        Rule.g:=[ [[1,0,0],1], [[0,0,1],1], [[1,1,0],0],[[0,1,1],0] ],
        getWp("sheet",Wp),
        (Wp>0)->(Wp.g:=Wp)
        otherwise (
                makesheet(3,4,Wp1), Wp.g:=Wp1,
                writess(Wp.g,"r", 1,1, constantlist(0,M.g)),
                ssw(Wp.g,1,70,1), ssw(Wp.g,1,140,1)      );
inputsequence()="d";
times()=1000;
initialstate()=c <-> c:=[];
delta(c,a)=cc <->
        cc:=defList(p1(z,i,[]),member(i,genIndex(2,M.g-1))),   //変更チェック
        Ls:=defList(p2(z,[r,i,v],[]),member([r,i,v],cc));   //SS に書き込み
lambda(c)=b <-> b:=0, showHistory();

p1(z,i,[]) <-> [N]:=projectss(Wp.g,1,i-1,1,i+1), member([N,v],Rule.g), z:=[1,i,v];
p2(z,[r,i,v],[]) <-> ssw(Wp.g,r,i,v);
showHistory() <->
        [c]:=projectss(Wp.g,1,1,1,M.g),
        show1(c,"plot"),
        hist1:=hist.g, x:=sum(c),
        hist2:=append(hist1,[x]), len:=cardinality(hist2),
        hist3:=project(hist2,["r", len-199,len]),
        show2(7,hist3,"plot"),
        //y:=invproject(c,1,"l"),(x <7)->(xwriteln(0,"place=",floor(y-1)),.stop),
        hist.g:=hist3;
```

3.5 応用：オートマトンの多層ネットワーク

本節の目的は，モデル理論アプローチによるシミュレーション開発実行環境 Simcast を使用して，複数の「Moore 型オートマトン」を要素とするネットワークを多層的に組合わせた複雑なシステムのシミュレーションを行なう方法を提案することである．

3.5.1 要素システム

ここでは，要素システムである Moore 型オートマトンを解説し，その CAST 言語によるモデルを紹介する．

(1) Moore 型オートマトン

Moore 型オートマトンは，入力に応じて状態が遷移し，遷移後の状態に対して出力が決まるようなシステムのモデルである．

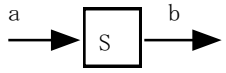


図 3-5-1　要素システム S1

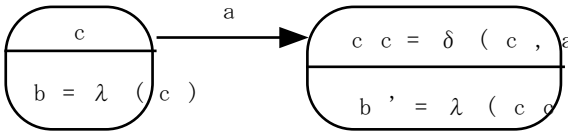


図 3-5-2　状態遷移図

ある時刻でのシステム S への入力を a，出力を b，状態を c とする（図 3-5-1）．また，次の時刻での状態を cc と書くことにする（cc は２つの c を乗じたものではなく，ひとつの記号として扱う）．これらの変数の間に，

$\delta(c,a)=cc$ かつ $\lambda(c)=b$

の関係があるとき，S はオートマトンであるといい，$<\delta, \lambda, c0>$ を S1 の Moore 型オートマトンモデルと呼ぶ．ただし，δ を状態遷移関数，λ を出力関数，c0 を初期状態と呼ぶ．

状態遷移図は図 3-5-2 のようになる．時刻 t における状態が c のとき，出力は $\lambda(c)$ である．そこに a が入力されると，次の時刻 t+1 における状態は $cc=\delta(c,a)$ となり，$b'=\lambda(\delta(c,a))$ が出力される．

一般に，出力がその時刻の入力のみに依存して定まるようなシステムを静的システムと呼び，それ以外を動的システムと呼ぶ．この言い方からすると，現在の出力が過去の入力と過去の状態によって定まるオートマトンは動的システムである．

(2) Moore 型オートマトンの CAST 言語によるモデル

具体的な例として，入力をそのまま出力するが，その入力を状態として一時的に保持するようなシステム S1 を考える．これを Moore 型オートマトンとしてモデル化すると，

$$\begin{cases} \delta 1(c,a) = cc <-> cc=a \\ \lambda 1(c) = b <-> b=c \end{cases}$$

となる．ただし，初期状態を c0=0 とする．このモデル $<\delta 1, \lambda 1, 0>$ は，a が入力されたとき，次の状態 cc が a と等しくなり，出力 b も a と等しくなるようなモデルである．結果として，S1 のオートマトンモデルとなっている．

さて，上記のオートマトンモデルを CAST 言語で書くと図 3-5-3 のようになる．テキストエディタでこれを作成し，例えば sim10.set というファイル名で保存する．ここで，開発環境 MTA-SDK を起動し，ファイル sim10.set をコンパイルすると自動的に実行され，図 3-5-4 の最終３行のような結果が得られる．

```
//sim10.set　要素システム S1 のモデル
func([delta1,lambda1]);
delta1(c,a)=cc <-> cc:=a;
lambda1(c)=b <-> b:=c;

inputsequence()=[1,0,0,0,0,0,0,0,0,0,0,0,0];
initialstate()=0;
delta(c,a)=delta1(c,a);
lambda(c)=lambda1(c);
```

図 3-5-3　要素システム S1 の CAST 言語によるモデル

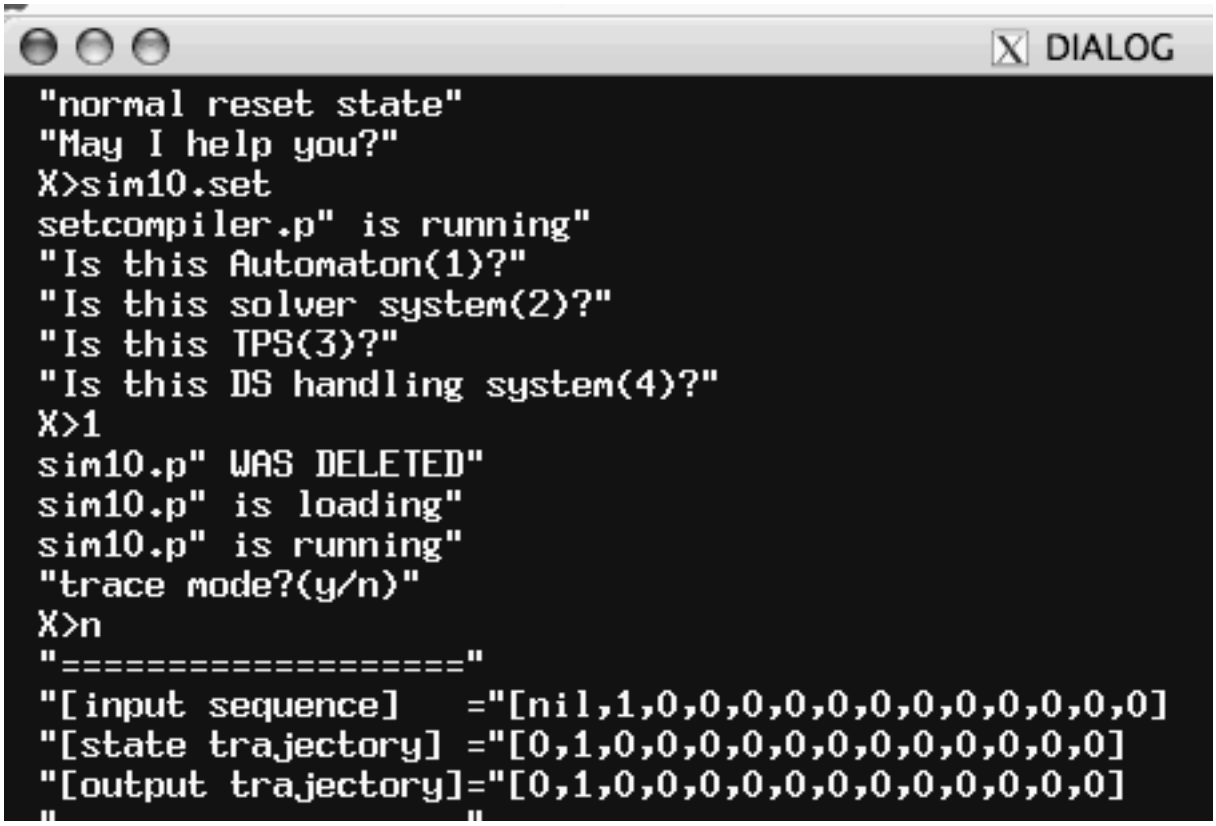


図 3-5-4　モデル（図 3-5-3）のコンパイルと実行

下から３行目の inputsequence の右辺の中の nil は，初期入力（時刻 0 での入力）がないことを表している．nil の下に，初期状態 0 と初期出力 0 が示されている．

２項目の１が時刻１での入力である．その１の下に，それに対する遷移後の状態が１，出力が１であることが示されている．

オートマトンを実行するエンジン stdAutomatonEngine.p の動作について述べる．まず，initialstate()から初期状態を取り出し，lambda(c)を使って，初期出力を計算する．次に，入力系列 inputsequence()から順次入力を取り出して，delta(c,a)と lambda(c)を繰り返して計算する．本質的に，エンジンは，delta(c,a)と lambda(c)の右辺で定義されているものを，交互に，実行する（計算する）だけのものである．

3.5.2 単層ネットワーク

単層ネットワーク（同期型固定結合網）の一般的構造と CAST 言語によるモデルの実装構造，および例として循環するネットワークのモデルを示す.

(1) 循環するネットワークのモデル

単層ネットワークの例として，図 3-5-5 のような循環するネットワークを考える.

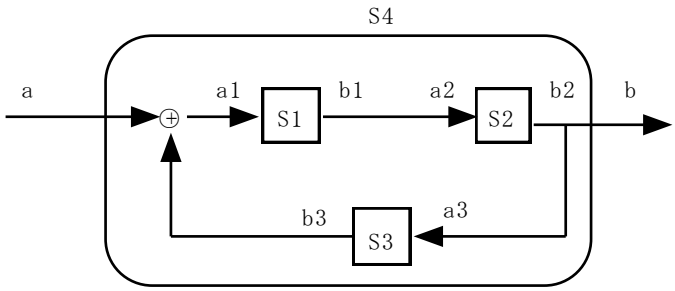


図 3-5-5　循環するネットワーク S4

このシステム S4 は，３つの動的システム（要素システム S1,S2,S3）と静的システム（加算要素(+)）を含む, ネットワークシステムである. 次の図 3-5-6 は, 図 3-5-5 の配置を換え, 動的システム S1, S2, S3 が並列に並ぶように描き直したものである．このように配置すると，要素システムの結合が図の左側にひとつにまとまり，後述するような一貫した理解ができるようになる.

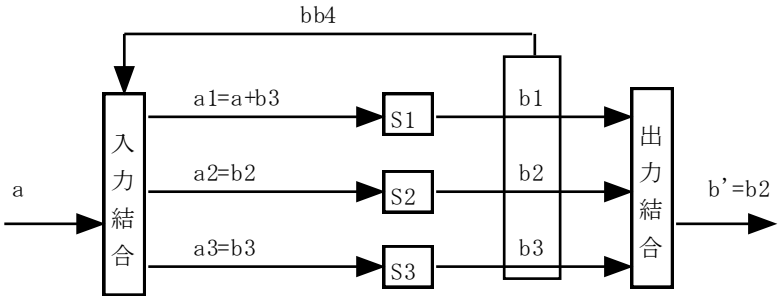


図 3-5-6　循環するネットワーク S4 の構造

外部からの入力 a と出力 bb4 と要素システムへの入力 aa=(a1,a2,a3) の間の関係を入力結合と呼ぶ．それを関数として表現するなら，aa=in(a,bb4)と書くことができる．循環するネットワーク S4 の例では，(a1,a2,a3)=(a+b3,b1,b2)に対応する．また，出力 bb4 と外部への出力の関係は，b=out(bb4)と書くことができる．S4 の例では，b=b2 となっている．ただし，出力 bb4=(b1,b2,b3) は, 次の時刻にどれかの要素システムへの入力として使用されるので, 入力側に帰還（フィードバック）している.

これらを使えば，つぎのように全体システムの状態遷移関数と出力関数を定義することができる.

$$
(1)\quad \begin{cases} \delta 4(c,a)=cc <-> \\ \quad bb4=(\lambda 1(c1),\lambda 2(c2),\lambda 1(c3)), \\ \quad (a1,a2,a3)=in(a,bb4), \\ \quad cc=(\delta 1(c1,a1),\delta 1(c1,a1),\delta 1(c1,a1)). \\ \lambda 4(c)=b <-> \\ \quad bb4=(\lambda 1(c1),\lambda 2(c2),\lambda 1(c3)), \\ \quad b=out(bb4). \end{cases}
$$

したがって，S4 は Moore 型オートマトンであり，$<\delta 4,\lambda 4,(0,0,0)>$がそのオートマトンモデルである．これを CAST 言語でモデル化すると，次のようになる．

```
//sim40.set　循環するネットワーク
func([delta1,lambda1,delta2,lambda2,delta3,lambda3,delta4,lambda4]);
delta1(c,a)=cc <-> cc:=a;
lambda1(c)=b <-> b:=c;
delta2(c,a)=delta1(c,a);
lambda2(c)=lambda1(c);
delta3(c,a)=delta1(c,a);
lambda3(c)=lambda1(c);
delta4(c,a)= cc <->
        [c1,c2,c3]:=c,
        [b1,b2,b3]:=bb4.g,              //前時刻の出力を取得
        [a1,a2,a3]:=[a+b3,b1,b2],       //入力結合
        cc:=[delta1(c1,a1),delta2(c2,a2),delta3(c3,a3)];    //次の状態
lambda4(c)= b <->
        [c1,c2,c3]:=c,
        bb4:=[lambda1(c1),lambda2(c2),lambda3(c3)],
        bb4.g:=bb4,                     //この時刻の出力を保存
        b:=project(bb4,2);              //出力結合

inputsequence()=[1,0,0,0,0,0,0,0,0,0,0,0,0];
initialstate()=[0,0,0];
delta(c,a)=delta4(c,a);
lambda(c)=lambda4(c);
```

図 3-5-7　ネットワーク S4 の CAST 言語によるモデル

ただし，オートマトンを実行するエンジン stdAutomatonEngine.p は２変数関数 delta(c,a)と１変数関数 lambda(c)を交互に実行するが，lambda4(c)を先に計算するので，bb4=(λ1(c1), λ2(c2), λ1(c3)) はすでに計算されている．これをグローバル変数 bb4.g に代入しておき，あとで delta4(c,a)を計算するときに再利用している．

また，このネットワークは３つの要素をもつシステムなので，状態は c=(c1,c2,c3)である．そのため，初期状態 initialstate()は３項組 [0,0,0] となっている．

要素システムが円状に結合しているので，入力結合は [a1,a2,a3]:=[a+b3,b1,b2] である．これが式１の入力結合(a1,a2,a3)=in(a,bb4)の具体的な形である．外部からの入力 a と内部要素の出力(b1,b2,b3)がここで混合され，各要素 S*i* への入力 a*i* となる．

出力結合は b:=project(bb4,2) となっている．これが式１の b=out(bb) の具体的な形である．bb4:=[b1,b2,b3] の中から b2 を抽出している．実行結果は図 3-5-8 に示した．

```
"==================="
"[input sequence]    ="[nil,1,0,0,0,0,0,0,0,0,0,0,0,0,0]
"[state trajectory] ="[[0,0,0],[1,0,0],[0,1,0],[0,0,1],[1,0,0],[0,1,0],[0,0,1]
,[1,0,0],[0,1,0],[0,0,1],[1,0,0],[0,1,0],[0,0,1],[1,0,0]]
"[output trajectory]="[0,0,1,0,0,1,0,0,1,0,0,1,0,0,1]
"==================="
```

図 3-5-8　モデル（図 3-5-7）の実行結果

(2) 単層ネットワークの実装構造

一般に，どのように結合した単層ネットワーク全体でも，式(1)のように，各要素システムの状態の組を状態とする，ひとつの Moore 型オートマトンとしてモデル化することができる．したがって，単層ネットワークの CAST 言語によるモデルの一般的な形は図 3-5-9 のようになる．

```
delta(c,a)= cc <->                                  //f(c,a)=cc のこと
        [c1,c2,c3]:=c,
        [b1,b2,b3]:=bb.g,                           //前時刻の出力を取得
        [a1,a2,a3]:=in(a,b1,b2,b3),                 //in( )は入力結合
        cc:=[delta1(c1,a1),delta2(c2,a2),delta3(c3,a3)];     //次の状態
lambda(c)= b <->                                    //g(c)=b のこと
        [c1,c2,c3]:=c,
        bb:=[lambda1(c1),lambda2(c2),lambda3(c3)],
        bb.g:=bb,                                   //この時刻の出力を保存
        b:=out(bb);                                 //out( )は出力結合
```

図 3-5-9　CAST 言語による単層ネットワークの実装構造

図 3-5-9 は３つの要素をもつネットワークのモデルであり，テンプレートとして使用することができる．要素の数が変われば delta*i* と lambda*i* を増減させ，入力結合 in()や出力結合 out()を状況に応じて変えれば良い．例えば，前項のシステム S4（図 3-5-7）では，入力結合 in(a,bb)は，[a1,a2,a3]:=[a+b3,b1,b2] ，出力結合 out(bb)は，b:=project(bb4,2) であった．

ただし，静的システムによる自己フィードバックについては第 3.2 節「状態機械の多層ネットワーク」を参照されたい．また，状況に応じて，結合そのものが変化したり，同じ入力に同時に複数の入力があるようなシステムの場合は，出力記憶 bb.g を，{["b1" v1],["b2" v2],["b3" v3]}のように，出力名とその値の組の集合で表記し，それを操作するような入力結合関数 in( ) を作成すればよい．例えば，h(x,bb)=y <-> member([x,y],bb); と定義しておけば，入力結合 in(a,bb.g)=aa <-> aa:=[h(bb.g,"a1"), h(bb.g,"a2"),h(bb.g,"a3")]; と bb.g:=[["a1",5],["a2",6],["a3",7]]に対して，aa=[a1,a2,a3]=[5,6,7]が得られる．関数 h を工夫して，複数入力の設定も可能である．

3.5.3　多層ネットワーク

前項で紹介した方法で，多層ネットワークをモデル化する．例として，次のネットワーク S6（図 3-5-10）を CAST 言語でモデル化する．

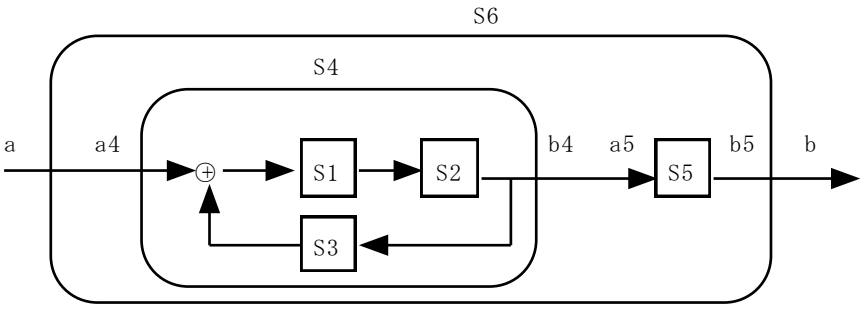


図 3-5-10　多層ネットワーク S6

システム S6 は，単層ネットワーク S4 とひとつのオートマトン S5 が結合した多層ネットワークである．しかし，いまや S4 もひとつのオートマトンであるとみなせるので，結果として, システム S6 を２つのオートマトンが直列結合している単層ネットワークとみなすことができる．したがって, 前節の単層ネットワークを実装構造（図 3-5-9）と同じように，モデル化することができる．モデルを図 3-5-11 に，実行結果は図 3-5-12 に示した．

```
〜〜〜〜〜　中略　〜〜〜〜〜〜
delta6(c,a)= cc <->
        [c4,c5]:=c,
        [b4,b5]:=bb6.g,
        [a4,a5]:=[a,b4],      //in(a,bb6)に相当
        cc:=[delta4(c4,a4),delta5(c5,a5)];
lambda6(c)= b <->
        [c4,c5]:=c,
        bb6:=[lambda4(c4),lambda5(c5)],
        bb6.g:=bb6,
        b:=project(bb6,2);

inputsequence()=[1,0,0,0,0,0,0,0,0,0,0,0,0];
initialstate()=[[0,0,0],0];
delta(c,a)=delta6(c,a);
lambda(c)=lambda6(c);
```

図 3-5-11　多層ネットワーク S6 の CAST 言語によるモデルの一部（詳細は付録 3-5-2）

このネットワークは２つの要素をもつシステムだが，S4 の状態が c=(c1,c2,c3)で，S5 の状態が c5 であるため，全体としての状態は，c=((c1,c2,c3),c5)となる．そのため，初期状態 initialstate()は [[0,0,0],0] となっている．

要素システムが直列に結合しているので，入力結合 in(a,bb6)は [a4,a5]:=[a,b4] である．出力結合 out(bb6)は b:=project(bb6,2) となっている．実行結果は図 3-5-12 に示した．

```
"==================="
"[input sequence]    ="[nil,1,0,0,0,0,0,0,0,0,0,0,0,0]
"[state trajectory] ="[[[0,0,0],0],[[1,0,0],0],[[0,1,0],0],[[0,0,1],1],[[1,0,
0],0],[[0,1,0],0],[[0,0,1],1],[[1,0,0],0],[[0,1,0],0],[[0,0,1],1],[[1,0,0],0]
,[[0,1,0],0],[[0,0,1],1],[[1,0,0],0]]
"[output trajectory]="[0,0,0,1,0,0,1,0,0,1,0,0,1,0]
"==================="
```

図 3-5-12　モデル（図 3-5-11）の実行結果

3.5.4 開発方法

モデル理論アプローチによるシミュレーション開発実行環境 Simcast において，単層ネットワークを積み重ねて，段階的に多層ネットワーク（同期型固定結合網）をモデル化することができる．

段階的モデル化における設計技法の主要点をまとめると，
１）単層ネットワークに対しては，要素の動的システム（オートマトン）を並列に並べた図 3-5-6 を考え，図 3-5-9 のようにモデル化する（第 2 項）
２）その時，すべての要素の出力をグローバル変数 bb.g に代入したり，参照したりする．
３）多層ネットワークに対しては，要素がネットワークであっても，それをひとつの動的システム（オートマトン）とみなして，単層ネットワークと同様にモデル化する（第 3 項）．
４）以上を繰り返して，複雑な多層ネットワークのモデルを構成していく．
となる．

この方法でモデル化すれば，記述も単純になり，一貫した視点で多層ネットワークのモデルを構成していくことが可能である．

付録 3.5.1（多層ネットワーク S6 の CAST 言語によるモデル）

```
//sim60.set
func([delta1,lambda1,delta2,lambda2,delta3,lambda3,
delta4,lambda4,delta5,lambda5,delta6,lambda6]);
delta1(c,a)=cc <-> cc:=a;
lambda1(c)=b <-> b:=c;
delta2(c,a)=delta1(c,a);
lambda2(c)=lambda1(c);
delta3(c,a)=delta1(c,a);
lambda3(c)=lambda1(c);
delta4(c,a)= cc <->
        [c1,c2,c3]:=c,
        [b1,b2,b3]:=bb4.g,
        [a1,a2,a3]:=[a+b3,b1,b2],
        cc:=[delta1(c1,a1),delta2(c2,a2),delta3(c3,a3)];
lambda4(c)= b <->
        [c1,c2,c3]:=c,
        bb4:=[lambda1(c1),lambda2(c2),lambda3(c3)],
        bb4.g:=bb4,
        b:=project(bb4,2);
delta5(c,a)=delta1(c,a);
lambda5(c)=lambda1(c);
delta6(c,a)= cc <->
        [c4,c5]:=c,
        [b4,b5]:=bb6.g,
        [a4,a5]:=[a,b4],
        cc:=[delta4(c4,a4),delta5(c5,a5)];
lambda6(c)= b <->
        [c4,c5]:=c,
        bb6:=[lambda4(c4),lambda5(c5)],
        bb6.g:=bb6,
        b:=project(bb6,2);

inputsequence()=[1,0,0,0,0,0,0,0,0,0,0,0,0];
initialstate()=[[0,0,0],0];
delta(c,a)=delta6(c,a);
lambda(c)=lambda6(c);
```

3.6 問題解決システム（Solver）の関数化

Simcast は MTA-SDK の段階ですでに提供されていた Solver（問題解決システム）も実行することができる．Solver とは，問題対象にアクションを加えながら，現在の状態から望みの状態に変化させてゆくシステムのことである．人工知能の分野では一般問題解決器 GPS と呼ばれるものに相当する．

Simcast では，この Solver を関数化して利用することが可能になった．Solver を関数化するとは，直感的に言えば，外側から Solver の設定値を変更して，Solver を実行できるようにすることである. Solver を関数化することができれば, さまざまなシステム要素に, Solver を埋め込むことができるので，複雑なシステムシミュレーションを行なうことが可能となる．

3.6.1 Solver の呼び出し

本稿では，オートマトンから問題解決システム（Solver）を呼び出して，値の授受を行ないながら利用する方法を解説する．すなわち，Solver は，外部から各種の設定値を変更されうるし，問題解決の実行結果を外部に返すことができる．そのような Solver をここでは「関数化した Solver」と称している（図 3-6-1）．また，関数化した Solver を外部から利用することを「Solver を呼び出す（call する）」ということにする．

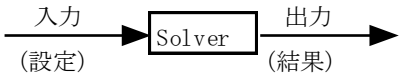


図 3-6-1　関数化した問題解決システム

オートマトンから Solver を呼び出すことができれば,「Solver の定義ファイルを変更することなく」異なる設定値に対する解を次々と求めることが可能になるので，各変数と解の間の関係を知るのに便利である．また，関数化した Solver を状態遷移関数とするオートマトンを作成することができる．複数のオートマトンを結合させることができるので，複雑なシステムシミュレーションを行なうことが可能となる．

3.6.2 利用の概要と書式

次の手順に従えば，Solver を関数化して利用することができる．

（1）まず，**入力出力値を授受できる**問題解決ユーザモデル（.set ファイル）を作成しコンパイルすることにより，関数化した Solver（.p ファイル）を作成しておく．

【書式】
1）solverInput.g(x)
2）solverOutput.g:=y

【意味】
1）外部からの入力をローカル変数 x に代入する．
2）外部への出力（y の値）を規定のグローバル変数 solverOutput.g に代入する．

（2）オートマトンから Solver を呼び出して利用する．

【書式】y:=call_solver("ファイル名", 入力)
【例】　y:=call_solver("opt1.p", x)
【意味】関数化されたシステム opt11.p に x の値を入力し，opt1.p を実行する．結果を y の値に取得する[8].

これらの述語や関数の利用に関する詳細は以下で述べていく．

3.6.3 Solver の関数化（1 変数）

例として極値を求める Solver（付録 3-6-1）を取り上げる．これは，関数 f(x) = x*(x - r.g) の変数 x を変動させたときに極大を与える x を求める問題解決システムである．問題解決システムの設計法については文献を参照されたい[3]．問題解決活動の最中では，定数 r.g の値は固定である．ここでは，定数 r.g を外部（別ファイルに定義したオートマトン）から自由に設定変更できるようにしたい．定数 r.g の値が変われば，最終状態（関数 f の極大を与える x）も変わるので，これを出力として取得するつもりである．Simcast（バージョン 0908xx 以降）では，外部からの入力と外部への出力は，図 3-6-2 のように，規定のグローバル変数 solverInput.g と solverOutput.g を媒介して授受する．

```
preprocess() <-> (solverInput.g(v)) -> (r.g:=v) otherwise (r.g:=4);
   ---
st(x) <-> finalstate(x), solverOutput.g:=x;
```

図 3-6-2　入力出力値を授受する Solver（付録 3-6-1 の抜粋）

外部入力の受け取りに関しては，図 3-6-2 のように**条件つき**で定義する．

(solverInput.g(v)) -> (r.g:=v) otherwise (r.g:=4)

これは作法である．solverInput.g(v)は，「**もしも外部入力があれば**，その値を変数 v に代入し，真理値が真となる」述語である．もしも外部入力がなければ solverInput.g(v)は偽となる．したがって，事前準備 preprocess()の行は「もしも外部入力があるならば，定数 r.g の値を外部で設定されている値とせよ．しかし外部入力がないならば，r.g:=4 とせよ．」と理解することができる．ただし，条件判定部分に v:=solverInput.g や solverInput.g(r.g)と記述してもエラーとなるので注意すること．

また，もしも事前準備 preprocess()を使用したくない場合には，初期値設定の副作用として外部入力を取得することもできる．例えば，

initialstate()=x0 <-> x0:=0, e.g:=0.01, (solverInput.g(v)) -> (r.g:=v) otherwise (r.g:=4);

とすれば良い．

このように，記述する場所がどこであれ，条件つきで定義する利点は，**外部入力が無い時でも Solver 単体でモデル作成，実行，修正が可能**だからである．

一方，外部への出力に関しては，停止条件の「副作用」として定義する．図２のモデルでは，停止条件 st(x)の行で，グローバル変数 solverOut.g に最終状態 x の値を代入している．

このように入力出力値を授受できる問題解決ユーザモデル（.set ファイル）を作成し，コ

---

[8]　関数 call_solver は開発環境 Simcast に新たに追加した組込みの関数である．その定義はファイル stdAutomatonEngine.p にある．

[3]　高原康彦ほか『形式手法 モデル理論アプローチ：情報システム開発の基礎』日科技連 (2007)

ンパイルすると，関数化した問題解決システム（.p ファイル）を作ることができる．重要なことは，関数化した問題解決システム（今の場合は，opt11.p）を必ず作成し，削除しないことである．図 3-6-3 は，opt11.set を単体で動作確認したときの goal(x)=-f(x)の推移を表わすグラフである．原点から関数 f(x)を下って極小に達している様子が分かる．

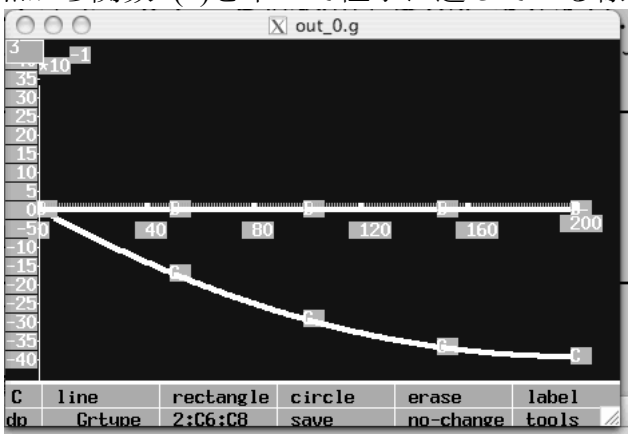


図 3-6-3　問題解決の実行

3.6.4 関数化した Solver の呼び出し

次に，別ファイルに定義したオートマトンから，Solver を呼び出して実行させたい．図 3-6-4 は上記の関数化した Solver "opt11.p"を子プロセスとして実行させるオートマトンモデルである．

```
//opt12.set
inputsequence0()=Rs <-> Rs:=[1,2,3,4];
initialstate0()=c <-> c:=0;
delta0(c,r)=cc <-> cc:=call_solver("opt11.p",r);
```

図 3-6-4　Solver を呼び出すオートマトンモデル

述語 cc:=call_solver("opt11.p", r) で，変数 r の値を Solver に渡して実行させている．返ってきた実行結果（解）は変数 cc に代入される．以上のように定義しておけば，Solver を関数として扱うことができる．

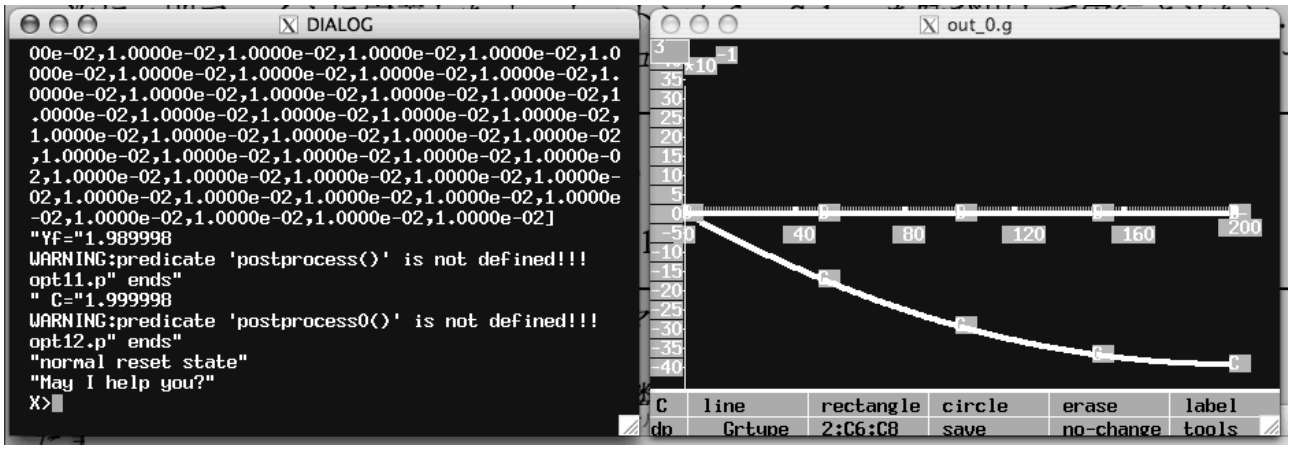


図 3-6-5　オートマトン opt12.set の実行結果

図 3-6-4 を実行すると上図のように結果が表示される．右側は通常 Solver 実行中に出現

するグラフウインドウであり，関数 goal(x) = -f(x)の値の変化が表示されている．4 回の入力がある（r=1,2,3,4)ので，4 回このウインドウは書き換えられる．

左側は Solver とオートマトンの実行処理中の入力と出力と状態の変化の様子が表示されている．最終的な入力 r = 4 のときの極値を与える x は，本来は 2.0 であるはずだが，このウインドウでは, C=1.999998 と表示されていることに注意する. その理由は, 付録 1 の Solver が誤差 0.01 以内で停止するように定義されているからである．以上により,「Solver の定義ファイルを変更することなく」, 異なる設定値に対する解を次々と求めることが可能になった．

しかしながら，図 3-6-5 のままでは直感的に理解することが困難である．そこで第 2. 3 節を参考に, グラフに点(r,cc)をプロットするように変更したものを付録 3-6-2 に収録した．そのユーザモデル"opt13.set"をコンパイルし実行した結果を図 3-6-6 に示す．問題解決システムの各変数と解の間の関係を知るのに便利である．

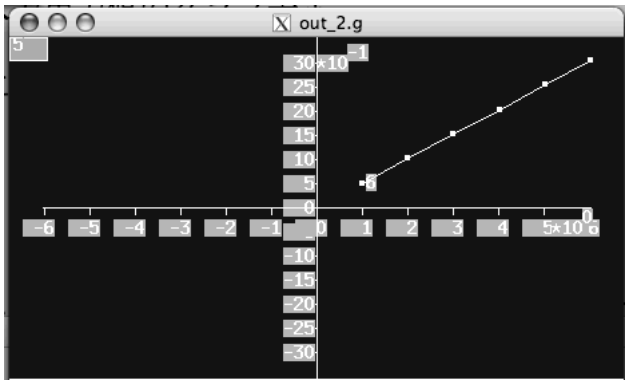


図 3-6-6　入力出力値のグラフ表示

さて，処理時間を見てみたい．ここでオートマトンへの入力系列 Rs（つまり Solver への外部入力）を，inputsequence0()=Rs <-> Rs:=[10,11,12,13]; に変更し実行すれば，極端に遅いことが分かる．その理由は，問題解決のサイクル

r → r.g → x0=0 → 問題解決 → 解 x → cc

毎に，いつも x0 = 0 から問題解決活動を開始するからである．しかしながら，r.g と r.g’が近ければ解 x と x’も近いので,「次のサイクルの開始点 x0 = 前回の結果 x」のほうが問題解決の時間が少なくてすむはずである．したがって「前回の解 x を今回の初期状態 x0 として再利用」しながら問題解決活動を連続して実行できれば処理時間が早くなるはずである．次項でこれを扱う．

3.6.5　Solver のオートマトン化（2 変数）

本節では，2 変数の値を Solver に渡すことを考える．例として，図 3-6-7 のように，オートマトンの状態遷移関数 δ そのものを Solver にしてみる．これにより,「前回の解を記憶しておき今回の初期状態として再利用し」ながら問題解決活動を連続して実行することが可能である．

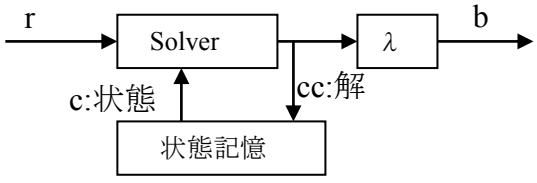


図 3-6-7　状態遷移関数が Solver であるオートマトン

付録 3-6-3 に 2 変数の外部入力を取得する Solver の例（opt21.set)を収録した．opt21.set は，外部入力があれば初期状態 x0 と定数 r.g を設定変更でき，結果（関数 f の極値を与える x）を外部に出力するモデルである．また付録 3-6-4 はそれを状態遷移関数とする状態機械（出力のないオートマトン）のモデルである．オートマトンからの入力(c,r)の値は Solver の(x0, r.g) に渡され，問題解決の結果（関数 f の極値を与える x）は，オートマトンの変数 cc に返される．

それら付録の方法の本質的な特徴は，

オートマトンの状態 c の値を Solver の初期状態 x0 に代入し，
Solver の解 x の値をオートマトンの次の状態 cc に代入する

ことにある．要するに Solver の解を使い回すのである．何度も繰り返す問題解決では，まず解を得た時点で立ち止まり，設定を変えて，再びその時点から問題解決活動を行なうことになろう．この方法は，そのような連続した**繰り返し問題解決のモデル**となるので，この方法を「Solver のオートマトン化」と呼んでいる．

さて，関数が f(x) = x*(r.g -x)であるので，定数 r.g が決まれば，どんな初期状態 x0 から出発しても同じ極値を与える x が得られる．したがって，付録 3, 4 をコンパイルし実行させても，前節の図 3 や図 4 と同じ計算結果を得るのは当然である．しかしながら，入力系列を inputsequence()=Rs <-> Rs:=[10,11,12,13]; に変更し実行すれば，本節の opt22.set の実行時間は前節の opt12.set より処理時間が短くなっていることが分かる（一回目の状態遷移を除く）.

3.6.6 注意点

1）Solver のコンパイル

先に Solver をコンパイルして実行しておくことが必要である．例題の場合は，問題解決システム opt11.p が存在しなければならない．その後，オートマトン opt12.set をコンパイルし実行する．

2）ユーザ定義の述語や関数の衝突

Solver を他のファイルから呼び出すときの留意点は，親プロセスで定義された述語や関数が子プロセスに引き継がれることである．したがって，オートマトンで定義された「**ユーザ定義の**述語や関数」は，Solver で定義された「**ユーザ定義の**述語や関数」と衝突しないように，異なる名称をつける必要がある．

3）トレースモード

オートマトンの実行時と Solver の実行時に，トレースモードにするか否かを聞かれるので，n を入力すること．システム開発時には役立つが，シミュレーション開発の後半や，他人に見せるとき（プレゼン用）には煩わしい．不要と感じる場合には，
stdAutomatonEngine.p と stdPDsolver122.p の中の「trace mode?」関連部分をコメントアウトせよ．

4）Solver から Solver を呼び出して実行することはできない．

もしそうしたければ，二つの Solver を第 5 項の方法でオートマトン化し，オートマトンとして結合させること．

古いバージョンの Simcast では，述語 initialstate()がオートマトンと Solver で衝突することを回避するためにゼロ(0)をつけていたが，新しい Simcast09（バージョン 0908 以降）ではゼロが不要となった．また，使いやすい関数 **call_solver**("opt11.p",r)を新規に導入した．

その結果，すべての Solver を簡単にオートマトンに埋め込み，オートマトンとして扱う

ことができるようになった．すでに確立した方法（第 3.2, 3.4 節）により，それらを組合わせた複雑なシステムを構築することができるだろう. それが利点である. 応用例として「4.3 価格調整システム」を紹介しているので参照されたい.

付録 3-6-1　値を授受する（関数 f の極値を求める）Solver

```
//opt11.set
func([f]); f(x)=y <-> y:=x*(x-r.g);
preprocess() <-> (solverInput.g(v)) -> (r.g:=v) otherwise (r.g:=4);

initialstate()=x0 <-> x0:=0, e.g:=0.01;
finalstate(x) <-> f(x)<f(x-e.g), f(x)<f(x+e.g);
genA(x)=Es <-> Es:=[-e.g, e.g];
delta(x,e)=x2 <-> x2:=x+e, constraint(x2);
constraint(x) <-> x >= 0, x<=20;
goal(x) = f(x);
st(x) <-> finalstate(x), solverOutput.g:=x;
```

付録 3-6-2　入力出力のグラフ表示

```
//opt13.set
inputsequence0()=Rs <-> Rs:=[1,2,3,4,5,6];
initialstate0()=c <-> c:=0, hist.g:=[];
delta0(c,r)=cc <->
        cc:=call_solver("opt11.p",r),
        showHistory(r,cc);

showHistory(r,cc) <->
        hist1:=hist.g,
        hist2:=append(hist1,[[r,cc]]),
        show2(2,transpose(hist2),"trajectory"),
        hist.g:=hist2;
```

付録 3-6-3　2 変数の値を取得する Solver

```
//opt21.set
func([f]); f(x)=y <-> y:=x*(x-r.g);
preprocess() <->
    (solverInput.g([v1,v2])) -> (x0.g:=v1,r.g:=v2) otherwise (x0.g:=0, r.g:=4);

initialstate()=x0 <-> x0:=x0.g, e.g:=0.01;
finalstate(x) <-> f(x)<f(x-e.g), f(x)<f(x+e.g);
genA(x)=Es <-> Es:=[-e.g, e.g];
delta(x,e)=x2 <-> x2:=x+e, constraint(x2);
constraint(x) <-> x >= 0, x<=20;
goal(x) = f(x);
st(x) <-> finalstate(x), solverOutput.g:=x;
```

付録 3-6-4　Solver を状態遷移関数とするオートマトン

```
//opt22.set
inputsequence0()=Rs <-> Rs:=[1,2,3,4,5,6];
initialstate0()=c <-> c:=0;
delta0(c,r)=cc <-> cc:=call_solver("opt21.p", [c,r]);
```

第 4 章 最適化のモデル

ここでは最適化を目的としたシミュレーションの方法やモデル例を示す.

4.1 最大・最小・最適化

言語 CAST において，最大，最小，最適化に関連する述語や方法について例を示す.

(1) 最大，最小

リスト L=[30,40,40,20]が与えられているとき,

最大値	max(L)=40
最小値	min(L)=20
４番目	project(L,4)=20
一致するものの位置（最初）	invproject(L,40)=2
一致するものの位置（全部）	invproject(L,40,"l")=[2,3]
最大値を与える位置（全部）	invproject(L,max(L),"l")=[2,3]

となる．最後の式 invproject(L,max(L),"l")は，これ 1 行で最大値 max(L)と一致する位置の全部をリストアップする関数である．モデルの中では

```
L:=[30,40,40,20],
L2:=invproject(L,max(L),"l")
```

と定義しておくと，変数 L2 に[2,3]が代入される．したがって，リスト L の最大値は 2 番目と 3 番目にあることが分かる.

(2) 最適化

最適化問題 (Optimization Problem) の簡単な例を考える.

問：関数 f が，f("a")=30; f("b")=40; f("c")=40; f("d")=20;
で与えられているとき，f(x)を最大にする x を求めよ.
答：x="b"または x="c"のとき f(x)は最大値 40 をとる.

関数 f は集合 X={"a","b","c","d"} から集合 Y={実数} への関数である．X は定義域で，{20,30,40}が値域です．{"b","c"} を最適解集合といい，40 を最適値という.

１）関数が有限集合 F:=[["a",30],["b",40],["c",40],["d",20]] で定義されているとき

【リスト演算】

リストのリストは行列として扱えるので，行列 F の転置をとり，[F1,F2]:=transpose(F) とすると，F1= ["a","b","c","d"], F2= [30,40,40,20]となる．F2 の最大値を与える位置をすべて求め(L2=[2,3])，対応する F1 の位置の値のリスト L1=["b","c"]を求めればよい.

```
[F1,F2]:=transpose(F),
L2:=invprojcet(F2, max(F2),"l"),
L1:=project(F1,L2),
```

【defSet の利用】

別の方法として，関数 defSet を使うことができる．集合 F の要素[u,v]をひとつずつ順にチェックしながら，第２項 v が最大値であるような要素の第１項 u を集めて集合にすれば最適解集合が得られる．

```
func([opt]);
opt(F)=defSet(p(y,[u,v],[F]),member([u,v],F));
p(y,[u,v],[F]) <->
        R=project(transpose(F),2),//値 f(x)のリスト
        v=max(R),//の最大値が第２項 v と等しいような
        y=u; //第１項 u を取り出す
```

とユーザ定義しておけば，opt(F)は関数 F の最適解集合になる．述語 p(y,[u,v],[F])の第３項の[F]は必要である．忘れがちなので要注意せよ．たとえば，
F:=[["a",30],["b",40],["c",40],["d",20]] のとき，opt(F)=["b","c"] となる．

２）関数が写像(mapping)で定義されている場合

```
func([f]);
f("a")=30; f("b")=40; f("c")=40; f("d")=20;
```

関数が上のように定義されているとき，

```
f.g=defSet(pf(z,x,[]),member(x,["a","b","c","d"]));
pf(z,x,[]) <-> z:=[x,f(x)];
```

で，関数 f を集合表現 f.g に変換できる（MTA 教科書[1]第 4.5 節）．上記１）のリスト演算を行なうと L1=["b","c"] が得られ，またはユーザ関数 opt を使うと，opt(f.g)=["b","c"] となる．

３）初等関数の場合

```
func([f]);
f(x)=y <-> y:=-x*x+5x-6;
```

関数が上のように数式で定義されている場合でも，有限個の f(x)を比較するだけなら，
集合表現 f.g に変換するやり方２）が使える．x=0,1,2,3,4,5 に対する最適解は，opt(f.g)=[2,3] となる．

[1] 高原康彦ほか『形式手法 モデル理論アプローチ：情報システム開発の基礎』日科技連（2007）

４）連続区間上の関数の場合

しかし，連続区間[0,5]上で連続関数（例えば，y=-x*x+5x-6）の最大値を与える実数 x を求めたいときは，連続区間[0,5]上の実数の数は 無限個あるので，defSet()も使えず，何か別の方法が必要である．関数のグラフの山登りをする必要があるので，問題解決システム（MTA 教科書[1]第５章）を作成することになる．

```
//opt01.set       極値を求める問題解決ユーザモデル
preprocess() <-> e.g:=0.001; //刻み幅
func([f]);
   f(x)=y <-> y:=-x*x+5*x-6;
initialstate() = 2;                                    //状態 x の初期値
finalstate(x) <-> f(x)>=f(x-e.g), f(x)>=f(x+e.g) ;  //極値の定義
delta(x,a) = x2 <-> x2:=x+a, constraint(x2);          //状態遷移関数
genA(x) = [-e.g,e.g];                                  //アクション集合（代替案集合）
constraint(x) <-> x >= 0, x<=5;/                       /状態 x の制限
st(x) <-> finalstate(x);                               //停止条件
goal(x) = -f(x);                                       //状態を評価する関数
```

上記のユーザモデル opt01.set をテキストエディタで作成し, 開発環境 Simcast を起動し，「Solver として」コンパイル／実行すれば，刻み幅 e.g づつ山登りをして，極値に達したら停止する．極値を与える近似 x=2.499 が得られる．

まとめると，リストや有限集合を対象とするときは，（多くは）CAST 言語で「式」を書けば答えが得らるが，しかし無限集合を相手にするときは，第 2 項の 4)のように問題解決をするために，CAST 言語で「ユーザモデル」を書く必要がある．

4.2 ナッシュ均衡を求める

4.2.1 ナッシュ均衡

例として，次の非零和二人ゲームを考える．

$$G = \begin{pmatrix} (1,1) & (4,4) & (1,1) & (4,4) & (1,1) \\ (6,2) & (5,3) & (6,2) & (5,3) & (6,2) \\ (1,1) & (4,4) & (1,1) & (4,4) & (1,1) \\ (6,2) & (5,3) & (6,2) & (5,3) & (6,2) \\ (1,1) & (4,4) & (1,1) & (4,4) & (1,1) \end{pmatrix}$$

この双行列 G の縦軸はプレーヤ 1 の戦略 1〜5 に対応し，横軸はプレーヤ 2 の戦略 1〜5 に対応している．たとえば，第 2 行第 3 列の(6,2)は，プレーヤ 1 が戦略 2 を実行し，プレーヤ 2 が戦略 3 を実行したとき，プレーヤ 1 は利得 6 を獲得し，プレーヤ 2 は利得 2 を獲得することを表わしている．つまり，各要素のコンマの左側がプレーヤ 1 の利得で，コンマの右側がプレーヤ 2 の利得である．双行列 G は，次の 2 つの行列を重ねたものである．G1 がプレーヤ 1 の利得行列，G2 がプレーヤ 2 の利得行列である．

$$G1 = \begin{pmatrix} 1 & 4 & 1 & 4 & 1 \\ 6 & 5 & 6 & 5 & 6 \\ 1 & 4 & 1 & 4 & 1 \\ 6 & 5 & 6 & 5 & 6 \\ 1 & 4 & 1 & 4 & 1 \end{pmatrix}, \qquad G2 = \begin{pmatrix} 1 & 4 & 1 & 4 & 1 \\ 2 & 3 & 2 & 3 & 2 \\ 1 & 4 & 1 & 4 & 1 \\ 2 & 3 & 2 & 3 & 2 \\ 1 & 4 & 1 & 4 & 1 \end{pmatrix}$$

ナッシュ均衡とは，「他のプレーヤがその戦略をとる限り，今の自分の戦略は最適である」とすべてのプレーヤが思う状態（戦略の組）のことである．ゲーム理論によるとこのゲームのナッシュ均衡解は 4 つあり，A(2,2), B(2,4), C(4,2), D(4,4)である．例えば，解 A(2,2)はプレーヤ 1 が戦略 2 をプレーヤ 2 が戦略 2 を実行することである．プレーヤ 1 から見れば，行列 G1 の第 2 行第 2 列の値 5 は，第 2 列のなかで最大値である．また，プレーヤ 2 から見れば行列 G2 の第 2 行第 2 列の値は，第 2 行のなかで最大値である．つまり，第 2 行第 2 列は，他のプレーヤがその戦略をとる限り，両プレーヤは最適となっている．

したがって，**各戦略の組(i,j)に対して，G1 の第 j 列のなかで第 i 番目が最大値であり，かつ，G2 の第 i 行のなかで第 j 番目が最大値であるとき，その戦略の組がナッシュ均衡解である．**

4.2.2 言語 CAST によるモデル

モデル理論アプローチでは，このような計算を行なうために，言語 CAST を使ってオートマトンを作成する．モデル化は次の方針で行なうこととする．

・オートマトンへの「入力 a」はチェックすべき戦略の組(i,j)とする．
・与えられた入力 a=(i,j)に対して，まず G1 の j 列 F1，G2 の i 行 F2 を切り出す．
・F1 の第 i 番目と，F2 の第 j 番目が，それぞれ最大値であるかをチェックする．
・もしもともに最大値なら，(i,j)がナッシュ均衡であることを「状態」に記録する．

今回はすべての戦略の組をチェックすることにする．1 回の状態遷移で 1 組の(i,j)のチェックを行なうので，オートマトンは 5×5＝25 回の状態遷移を行なうことになる．上記の計算を行なうオートマトン（実際には状態機械）の CAST 言語によるモデルは，次のように構成できる．

```
//nash17.set
inputsequence()=As <-> As:=product(genIndex(1,5),genIndex(1,5));
initialstate()=c0 <-> c0:=[];
delta(c,[i,j])=cc <->
        G1:=[[1,4,1,4,1],[6,5,6,5,6],[1,4,1,4,1],[6,5,6,5,6],[1,4,1,4,1]],
        G2:=[[1,4,1,4,1],[2,3,2,3,2],[1,4,1,4,1],[2,3,2,3,2],[1,4,1,4,1]],
        F1:=project(transpose(G1),j),
        F2:=project(G2,i),
        ( project(F1,i)=max(F1), project(F2,j)=max(F2) ) ->
        ( cc:=[i,j] ) otherwise ( cc:=[] );
```

図表 4−2−1　ナッシュ均衡を求める状態機械のモデル

言語 CAST では行列をリストのリストで表わすので，G1, G2 は次のように書く．

```
G1:=[[1,4,1,4,1],[6,5,6,5,6],[1,4,1,4,1],[6,5,6,5,6],[1,4,1,4,1]],
G2:=[[1,4,1,4,1],[2,3,2,3,2],[1,4,1,4,1],[2,3,2,3,2],[1,4,1,4,1]],
```

また，transpose(G1)は転置行列を表わしている．

図表 4-2-1 のモデルでは，入力系列 inputsequence として，まずチェックすべき戦略の組(i,j)のリスト As = [[1,1],[1,2],[1,3],...,[5,5]] を定めている．状態遷移 delta の定義では，行列 G1 の第 j 列を F1:=project(transpose(G1),j)によって定め, G2 の第 i 列を F2:=project(G2,i)によって定めている．また，project(F1,i)=max(F1), project(F2,j)=max(F2)は第 i 行 j 列が最大値であることを確認する述語である．このモデルでは「状態 c」は単に確認の結果を格納する場所 cc:=[i,j]であるにすぎず，それ以上の意味はない．全体として非常に簡単なモデルである．

図表 4-2-2　実行結果

このモデルをモデル理論アプローチによるシミュレーション開発実行環境 Simcast でコンパイルし実行した結果が図表 2 である．確かに C = [2,2], [2,4], [4,2], [4,4] がナッシュ均衡解であることが分かる．

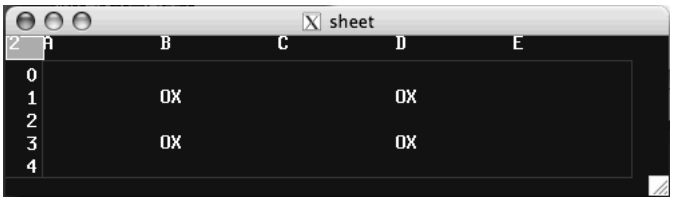
4.2.3　データの表示

以上が計算の本質的なところであるが，Simcast ではスプレッドシートによるデータ表示ができる（第 2.8 節）ので，若干の述語を追加したものを図表 4-2-3 に掲げた．

```
//nash18.set ナッシュ均衡を求めるオートマトン with SS
preprocess() <-> makesheet(5,5,Wp), Wp.g:=Wp;
inputsequence()=As <-> As:=product(genIndex(1,5),genIndex(1,5));
initialstate()=c0 <-> c0:=[];
delta(c,[i,j])=cc <->
      G1:=[[1,4,1,4,1],[6,5,6,5,6],[1,4,1,4,1],[6,5,6,5,6],[1,4,1,4,1]],
      G2:=[[1,4,1,4,1],[2,3,2,3,2],[1,4,1,4,1],[2,3,2,3,2],[1,4,1,4,1]],
      F1:=project(transpose(G1),j),
      F2:=project(G2,i),
      ( project(F1,i)=max(F1), project(F2,j)=max(F2) ) ->
      ( cc:=[i,j], ssw(Wp.g,i,j,"OX") ) otherwise ( cc:=[] );
```

図表 4-2-3　スプレッドシート表示を加えたモデル

まず，事前準備 preprocess として起動直後に 5 行 5 列のスプレッドシートウインドウを開いている．また，(i,j)がナッシュ均衡解であった場合に，述語 ssw(Wp.g,i,j,"OX")でスプレッドシートのセル(i,j)に文字"OX"を代入している．



図表 4-2-4　グラフ表示の結果

これを実行すると，図表 4-2-4 のようになる．確かに，セル(2,2), (2,4), (4,2), (4,4)に文字 OX が代入されている．

本項では，モデル記述言語 CAST を用いて，ゲーム理論における均衡解（具体的には，非零和ゲームにおけるナッシュ均衡解）を求めるオートマトンモデルを作成し実行してみた．モデルは 5×5＝25 個の入力(i,j)をチェックするだけのものである．

本節では，ゲーム理論の定理を使って筆算でできることを，わざわざオートマトンにやらせている．オートマトンにする必要性はないが，とりあえず Simcast でこのようなことができることを示した．

4.3 価格調整システム

本節では，ミクロ経済学における均衡価格を求めるシステム（価格調整システム）のシミュレーションを行なう．生産の意思決定と消費の意思決定はどちらも問題解決システム（以下，Solver）であるが，価格調整はフィードバックメカニズムである．モデル理論アプローチでは，Solver とオートマトンを混在させたシミュレーションを行なうことになる．

シミュレーション開発実行環境 Simcast では, Solver を関数としてオートマトンに組み込むことができる（第 3.6 節）ので，消費者，生産者，調整者をオートマトン（状態機械）としてモデル化できる．それらを結合した複合システムとして価格調整システムの CAST 言語によるモデルを構築したい.

4.3.1 価格調整システム

まず経済学における古典的な価格調整システムを簡単に紹介する.

(1) 消費者の意思決定

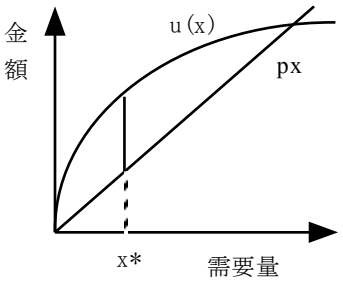


図 4-3-1　効用関数と費用関数

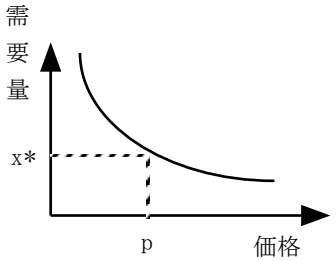


図 4-3-2　需要曲線 S1

ある財を量 x だけ購入したときの消費者にとっての効用を u(x)とする（単位は金額）．グラフは逓減しつつ右上がり，つまり上に凸である（図 4-3-1）．価格が p のとき，費用は px だから，純効用は f(x) = u(x) - px である．消費者は，価格 p が決まっているなら，純効用 f(x)を最大にするように購入量 x*を決めるだろう(f(x*) = max f(x))．最適購入量 x*は，図１のように，u(x)と px の差が最大の所である.

価格 p に対して最適購入量（需要量）x*が決まるので，関数で表すことができる(x* = S1(p))．関数 S1 を需要関数という．需要関数のグラフは右下がり．価格が上がれば需要は減少する（図 4-3-2）.

(2) 生産者の意思決定

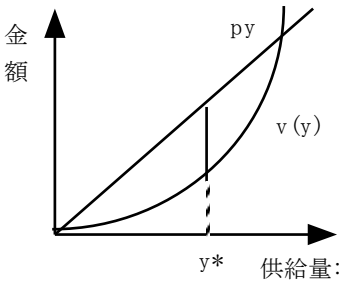


図 4-3-3　収入関数と費用関数

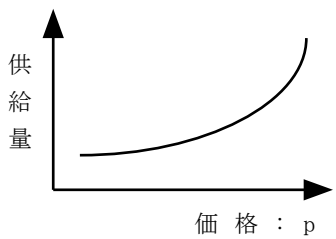


図 4-3-4　供給関数 S2

ある財を量 x だけ生産したときの生産者にとっての費用を v(y)とする（単位は金額）．グラフは逓増しつつ右上がり，つまり下に凸である（図 4-3-3）．価格が p のとき，すべて売れれば収入は py だから，純利益は g(y) = py - v(y)である．生産者は，価格 p でもすべて売れると分かっているなら, 純利益 g(y)を最大にするように生産量 y*を決めるだろう(g(y*) = max g(x))．最適生産量 y*は，図３のように，py と v(y)の差が最大の所である．

価格 p に対して最適生産量（供給量）y*が決まるので，関数で表すことができる(y* = S2(p))．関数 S2 を供給関数という．供給関数のグラフは右上がり．価格が上がれば供給量は増大する（図 4-3-4）．

(3) 価格調整メカニズム

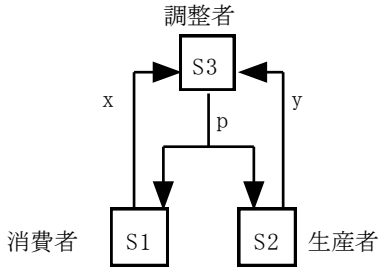


図 4-3-5　価格調整システム

今，調整者 S3 がおり，価格 p を修正していくことができるとする．その調整規則を，
１）需要量が供給量より大きい(S1(p) > S2(p))ならば，少し価格 p を上げる．
２）供給量が需要量より大きい(S1(p) < S2(p))ならば，少し価格 p を下げる．
とする．

図 4-3-5 のように，S1, S2, S3 が結合した複合システムを価格調整システムとよぶ．何らかの初期状態 (p,x,y) から始まっても，S3 の調整規則により，いずれは均衡（ S1(p) = S2(p) ）に至るだろう．

本稿では，このような動作を行なうシステムのシミュレーションを行ないたい．

4.3.2 問題解決システム

ここでは経済学のように微分を使わず，消費者と生産者の意思決定をモデル理論アプローチの意味の問題解決システム（Solver）としてモデル化する．

(1) 消費者の問題解決ユーザモデル

消費者 S1 は，与えられた価格 p に対して，最適購入量 x*を求めるシステムである．ここでは，

$$f(x)=3\sqrt{x} - px$$

としてみた．単に p を固定したときに極値を与える x を求めるにすぎない．購入量の初期値 x0 から出発し，少しずつ変化させながら関数 f(x)を最大にする x（最適購入量 x*）を求めるシステムである．

第 1 回目の問題解決活動中，価格 p は一定であるが，購入量は x0 から x*まで変化してゆく．重要なことは，価格が変化した後の第 2 回目の初期設定である．変化した価格は外部から与えられる（外生変数である）し，1 回目の最適購入量 x*を 2 回目の初期値 x0 にし

たいのである．そこで第 3.6 節の「Solver のオートマトン化」の部分を参考に，消費者 S1 の問題解決活動を表す CAST 言語によるユーザモデルを作成した．

```
//price1.set      極値を求める Solver（消費者）
preprocess() <->
   (solverInput.g([x,p])) -> (x0.g:=x, p.g:=p) otherwise (x0.g:=6, p.g:=0.6);

func([f]); f(x)=y <-> y:=3*sqrt(x)-p.g*x;
initialstate()=x0 <-> x0:=x0.g, e.g:=0.01;
finalstate(x) <-> f(x)>=f(x-e.g), f(x)>=f(x+e.g);
genA(x)=Es <-> Es:=[-e.g, e.g];
delta(x,e)=x2 <-> x2:=x+e, constraint(x2);
constraint(x) <-> x >= 0, x<=20;
goal(x) = -f(x);
st(x) <-> finalstate(x), solverOutput.g:=x;
```

図 4-3-6　最適購入量を求める Solver のユーザモデル

図 4-3-6 は「外部と入力出力値を授受できる問題解決ユーザモデル」である．事前準備 preprocess の定義

(solverInput.g([x,p])) -> (x0.g:=x, p.g:=p) otherwise (x0.g:=6, p.g:=0.6)

が，外部から x0.g, p.g の値を受け取るための記述である．外部から呼び出されたときの問題解決の初期状態 x0. g と価格 p.g は，グローバル変数 solverInput.g の値が代入される．一方，単体で実行するときの問題解決の初期状態は x0.g=6, 価格は p.g=0.6 となる．これら初期状態から徐々に x を増減して最適解 x*に移動させてゆくモデルとなっている．

また，

st([p,x]) <-> finalstate([p,x]), solverOutput.g:=x;

は，最適購入量 x*の値を外部に渡すための記述である．その他は教科書[ﾇ]第 5 章を参照されたい．

[ﾇ] 高原康彦ほか『形式手法 モデル理論アプローチ：情報システム開発の基礎』日科技連 (2007)

(2) 生産者の問題解決ユーザモデル

図 4-3-7 が生産者 S2 の意思決定を表す CAST 言語によるユーザモデルである．消費者のモデルと同じく，「入力出力値を授受できる問題解決ユーザモデル」である．ここでは，

$$g(x) = py - (1/10)y^2$$

としてみた．単に価格 p を固定したときに極値を与える y を求めているにすぎない．

```
//price2.set      極値を求める Solver（生産者）
preprocess() <->
    (solverInput.g([y,p])) -> (y0.g:=y, p.g:=p) otherwise (y0.g:=6, p.g:=0.6);

func([g]); g(y)=s <-> s:=p.g*y -(y*y/10);
initialstate()=y0 <-> y0:=y0.g, e.g:=0.01;
finalstate(y) <-> g(y)>=g(y-e.g), g(y)>=g(y+e.g);
genA(y)=Es <-> Es:=[-e.g, e.g];
delta(y,e)=y2 <-> y2:=y+e, constraint(y2);
constraint(y) <-> y >= 0, y<=20;
goal(y) = -g(y);
st(y) <-> finalstate(y), solverOutput.g:=y;
```

図 4-3-7　最適生産量を求める Solver のユーザモデル

(3)　問題解決システムの作成

これらのファイル price1.set, price2.set を，それぞれ単体でコンパイルして実行することができる．するとコンパイルされたファイル price1.p と price2.p ができる．これが関数化された問題解決システム（Solver）の実体である．次節では，これらの Solver をオートマトンから呼び出し，実行する予定である．

ところで，調整者 S3 については Solver を作成しない．オートマトンとしてモデル化するからである．以下の項で扱う．

4.3.3 オートマトン（状態機械）モデル

最終的な全体システムS4は図4.3.8のように３つの要素システムからなる複合システムとする．消費者 S1 は price1.p を関数化した状態機械とし，生産者 S2 は price2.p を関数化した状態機械とする．価格 p が状態機械 S1, S2 に入力されると，新しい需要量 x*と供給量 y*が計算され，次の状態（出力）となる．調整者 S3 は，2.3 項で述べたように，入力(x,y)に対して少しだけ修正した価格 p を次の状態（出力）とする状態機械とする．

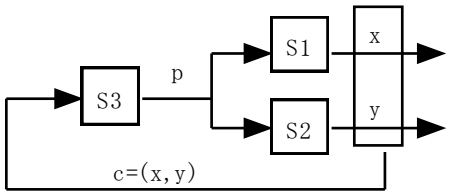


図 4-3-8　価格調整システム S4 のブロック線図

また, S1 と S2 は並列処理並列結合しており, S3 はこれらと逐次処理直列結合している．したがって，その構造は，第 3.1 節の練習問題４にフィードバックがかかったものに等しい．よって，価格調整システムの CAST 言語によるモデルは次のようになる．

```
//price4.set      均衡価格を求めるオートマトン
func([delta1,delta2,delta3,delta4]);
delta1(x,p)=xx <-> xx:=call_solver("price1.p",[x,p]);
delta2(y,p)=yy <-> yy:=call_solver("price2.p",[y,p]);
delta3(p,[x,y])=pp <->
      (x -y > 0.5) ->(pp:=p +0.05),
      (y -x > 0.5) ->(pp:=p -0.05),
      (abs(x -y)<= 0.5) -> (pp:=p);
delta4([p,x,y],a)= cc <->
      fb:=[x,y],
      pp:=delta3(p,fb),
      cc:=[pp,delta1(x,pp),delta2(y,pp)];

inputsequence()="d";
times()=5;
initialstate()=c <-> c:=[0.6,6,3];
delta(c,a)=delta4(c,a);
```

図 4-3-9　価格調整システムの CAST 言語によるモデル

このモデルの中で，delta1 と delta2 が，それぞれ消費者 S1 と生産者 S2 の状態遷移関数であり，

```
delta1(x,p)=xx <-> xx:=call_solver("price1.p",[x,p]);
delta2(y,p)=yy <-> yy:=call_solver("price2.p",[y,p]);
```

が，それぞれ price1.p と price2.p を関数化している．ただし，S1, S2 にとっては，x と y が現在の状態で，価格 p が（外部からの）入力である．次の時刻の状態 xx と yy は，それぞれ p に対する最適購入量（需要量）と最適生産量（供給量）である．delta3 は調整者の状

態遷移関数である．現在の価格（状態）p に対する需要 x と供給 y（入力）を比較して，次の時刻での価格 pp（出力）を計算している．

これら S1, S2, S3 を要素とする全体システム S4 の状態は３項組 (p,x,y) である．したがって，初期状態を[0.6,6,3]とした．全体システムの状態遷移関数は delta4 である．フィードバックは fb:=[x,y]であり，これがまず delta3 に代入され，得られる価格 pp が次に delta1 と delta2 に代入されている．ただし，delta4([p,x,y],a)= cc の中の入力 a はどこへも影響を与えない変数だが，時間を進ませるために必要である．ダミーの入力として"d"を５回入力するように設定してみた．

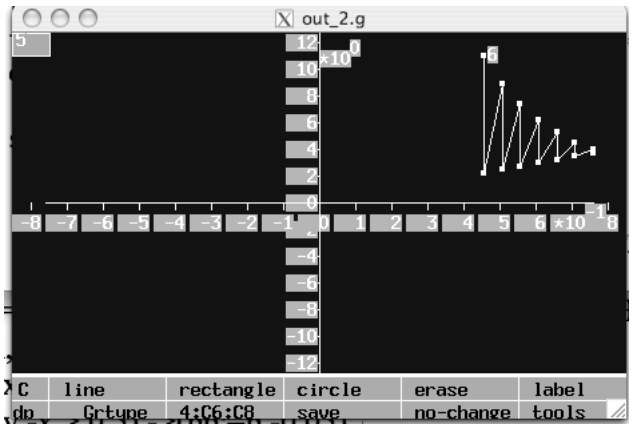
問題解決システム price1.p と price2.p が存在すれば，price4.set（図 4-3-9）をコンパイルして動かすことができる．毎回トレースモードにするかを聞かれるので n と答える．それが煩雑に感じるなら，拡張子.p のファイル中の対応部分をコメントアウトすればよい．

4.3.4 グラフ表示を加えたユーザモデル

参考のため，これまでのモデルを拡張し，処理途中のグラフ表示を行なうモデルを例示する．

```
//price5.set      均衡価格を求めるオートマトン（グラフ表示）
func([delta1,delta2,delta3,delta4]);
delta1(x,p)=xx <-> xx:=call_solver("price1.p",[x,p]),showHistory(p,xx);
delta2(y,p)=yy <-> yy:=call_solver("price2.p",[y,p]),showHistory(p,yy);
delta3(p,[x,y])=pp <->
        (x -y > 0.5) ->(pp:=p +0.05),
        (y -x > 0.5) ->(pp:=p -0.05),
        (abs(x -y)<= 0.5) -> (pp:=p);
delta4([p,x,y],a)= cc <->
        fb:=[x,y],
        pp:=delta3(p,fb),
        cc:=[pp,delta1(x,pp),delta2(y,pp)];

inputsequence()="d";
times()=9;
initialstate()=c0 <-> c0:=[ 0.4, 9, 3], hist.g:=[];
delta(c,a)=delta4(c,a);

showHistory(u,v) <->
        hist1:=hist.g,
        hist2:=append(hist1,[[u,v]]),
        show2(2,transpose(hist2),"trajectory"),
        hist.g:=hist2;
```

このモデルを実行した結果は図表 4-3-10 に示す．



図表 4-3-10

本節では，もともと Simcast が複雑なシステムのシミュレーションができることを例示したかった．そこで価格調整システムを題材に，Solver とオートマトンを混在できることを示した．正確には，Solver をオートマトンに組み込んで，オートマトンとして扱うことができる．そして，それらを結合して複合システムを構成することができるのである．

【補足】

しかし，もしも関数 f, g が微分可能なら，微分により極値を与える関数を求め，それを使用すれば Solver を利用する必要はない．

練習問題１：微分を使って，f, g の極値を与える関数 S1(p)=x*, S2(p)=y*を求め，さらに均衡価格 p*の理論値を求めよ．シミュレーションの途中の結果（最終結果）と理論値を比較せよ．

練習問題２：消費者の効用関数に変数 w を加え f(p,x,w)とする（適当な関数を作る）と，全体システムの入力 inputsequence によって w を変化させることができるはずである．消費者の効用が時間の経過とともに変化するようなモデルを作成し，実行してみよ．

4.4 遺伝アルゴリズム

本稿では，遺伝アルゴリズム（Genetic Algorithm：GA）を行なうオートマトンを CAST 言語を用いて構成する．オートマトン 1 回の状態遷移で，遺伝アルゴリズムの 1 サイクル（1 世代）の計算を行なう．本稿では，遺伝子プールの行列を状態とし，状態遷移は行列計算で行なう．

4.4.1 遺伝アルゴリズム

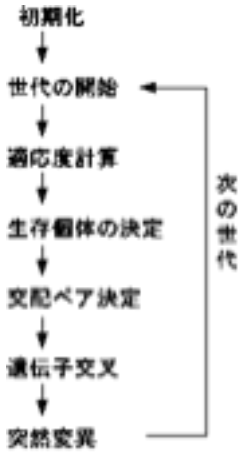


図 4-4-1　遺伝アルゴリズム

遺伝アルゴリズム（図 4-4-1）の手順と Excel 上の VBA 言語によるプログラムは，Web 資料[ﾇ]を参照されたい．本稿では，そのページの「自治体税収の近似式」を求める例題を，CAST 言語でモデル化する．

4.4.2 CAST 言語によるモデル

近似式を求める遺伝アルゴリズムの CAST 言語によるオートマトンモデルを構成したい．上記 Web ページの図表３「近似式の推定(Excel 画面)」を見ると，右側に 20 個の遺伝子と適応度（fitness)を表示した表（20 行×3 列の行列）がある．その表中の右 2 列が遺伝子を表わしている．この 20 行×2 列の行列が遺伝子プールである．これをオートマトンの状態とする．

つまり，ひとつの個体（直線）の遺伝子は直線の切片と傾きの組[u,v]である．状態はそれら 20 組をリストにして並べたもの c=[[u1,v1], [u2,v2], ... , [u20,v20]] である. 言語 CAST では，リストのリストは「行列」と考えて処理できる．

単に状態遷移だけを考えるので, 出力は不要である. したがって, 正確には状態機械（state machine）モデルを構成する．遺伝アルゴリズムを行なう状態機械の CAST 言語によるモデルは図２のようになる

[ﾇ] 旭貴朗「Excel による進化シミュレーション」 (2008.03.29)
http://www2.toyo.ac.jp/~asahi/education/soron/shiryo/programing/language/excel/genetic/

```
//ga05b.set 遺伝アルゴリズム（defList 計算）
func([y,f,fit,rand,h,g]);
        y(1)=0.602; y(2)=1.346; y(3)=1.247; y(4)=2.350; y(5)=2.597;
        f(u,v,x)=abs(u+v*x - y(x));
        fit(u,v)=f(u,v,1)+f(u,v,2)+f(u,v,3)+f(u,v,4)+f(u,v,5);
        rand(n,m)=r <-> myrandom(r1), r:=n+(m-n)*r1;
        h()=rand(-5,5)*pow(0.1,rand(1,10));
        g(L,i,j)=M <->
                [[u1,v1],[u2,v2]]:=project(L,[i,j]),
                M:=[[u1+h(),v2+h()],[u2+h(),v1+h()]]; //交叉+突然変異
inputsequence()="d";
times()=1000;
initialstate()=c0 <-> c0:=defList(p1(z,x,[]),member(x,genIndex(1,20)));
delta(c,a)=cc <->
        //世代の開始
        Ws:=defList(p2(z,[u,v],[]),member([u,v],c)), //適応度計算
        c1:=sort(Ws), //適応度順の個体リスト
        Ps:=projectMatrix(c1,1,2,10,3), //親を選択
        L:=[g(Ps,1,2),g(Ps,3,4),g(Ps,5,6),g(Ps,7,8),g(Ps,9,10)],
        Qs:=append(L), //子を確定
        cc:=append(Ps,Qs); //次世代が確定
p1(z,x,[]) <-> z:=[rand(-10,10),rand(-10,10)];
p2(z,[u,v],[]) <-> z:=[fit(u,v),u,v];
```

図 4-4-2　遺伝アルゴリズムを行なう状態機械の CAST 言語によるモデル

図 4-4-2 の初期状態 initialstate の定義では一様乱数を発生させて 20 個の初期遺伝子を設定している．つまり，状態 c は，各行が個体の遺伝子であり，それが 20 行並んだ行列である．

状態遷移 delta(c,a)=cc は次のように計算される．まず適応度 fit(u,v)を計算し，それを付加した個体[fit(u,v),u,v]のリスト Ws を作成する．これをソートすれば，適応度順に並んだ個体リスト c1 ができる．

次の Ps は行列 c1 の第 1 行第 2 列から第 10 行第 3 列までを抜き出しており，適応度の高い親グループを作成している．関数 g(Ps,1,2)は，適応度が 1 位の親と 2 位の親から，遺伝子交叉により２子を作成するものである．そのとき（小さな数字の乱数 h()を加えて）突然変異も行なっている．ちなみに，rand(n,m)は数値 n から m までの間で乱数を取得する関数である．したがって，Qs は子グループである．ここでは Web ページと同じように親グループと子グループを合わせて次世代 cc としている．

入力系列の定義 inputsequence()="d"; times()=1000;ではダミーの"d"を a に 1000 回代入して時間を進める（1000 世代の進化を計算する）．シミュレーション開発実行環境 Simcast で図 2 のモデルをコンパイルし実行した結果を図 4-4-3 に示す．実行時間は 4 秒であった．

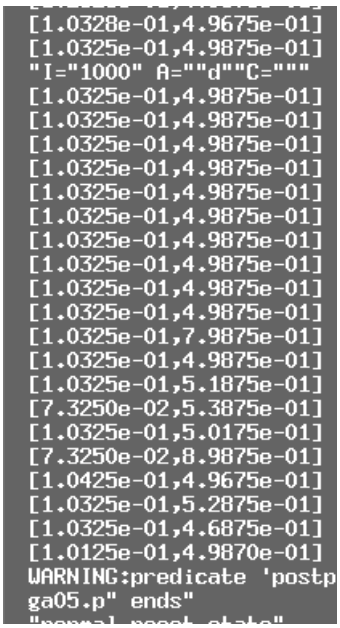


図 4-4-3　実行結果

また，図 4-4-2 では遺伝アルゴリズムの計算部分だけを示してあるが，データのグラフ表示やスプレッドシート表示の機能はない．第 2.5, 2.8 節を参考にして，データ表示とファイル保存も加えると次のようになる．実行時間は 18 秒であった．

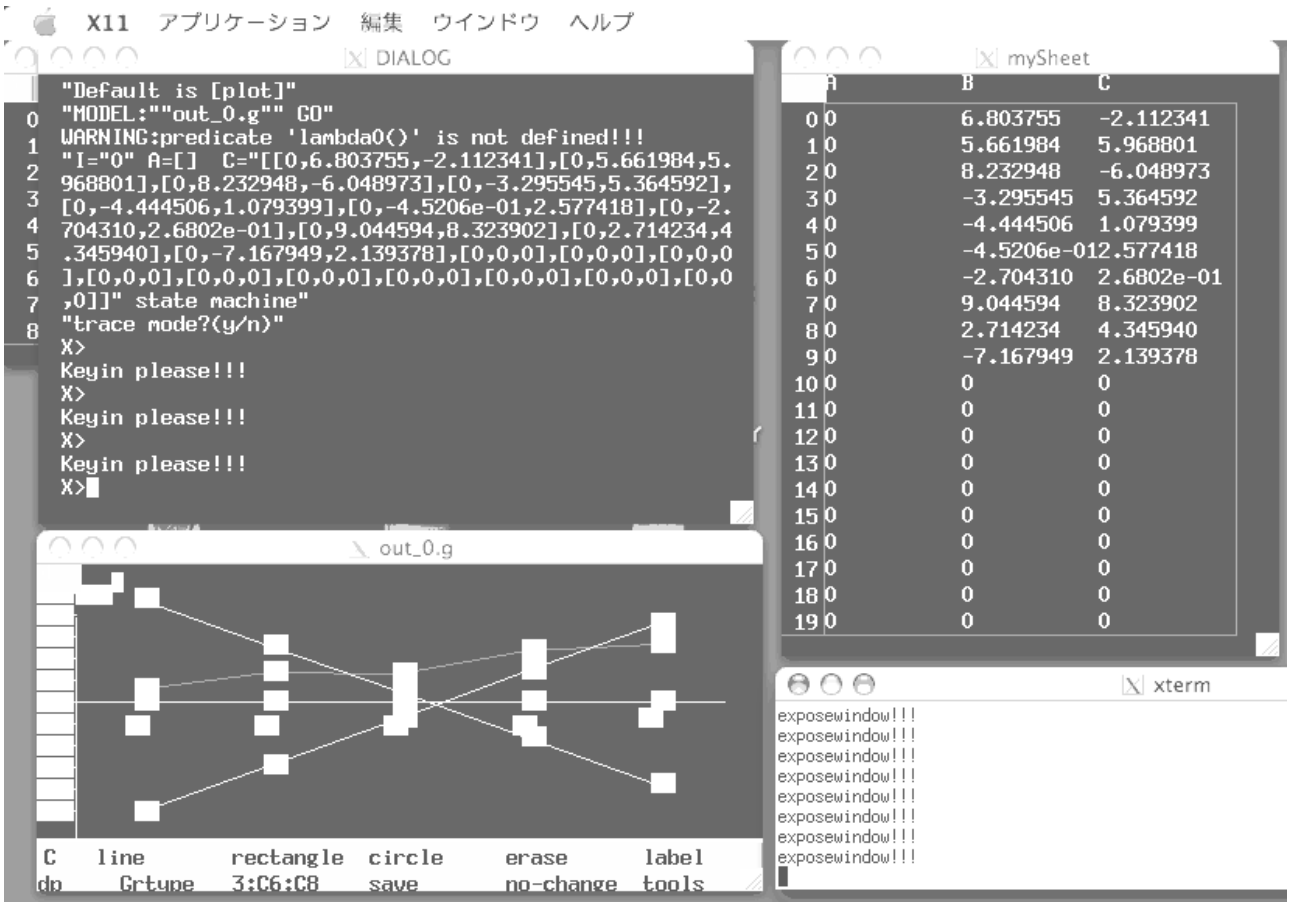


図 4-4-4 実行中の様子

```
//ga11.set 遺伝アルゴリズム（全データ表示）
func([y,f,fit,rand,h,g]);
        y(1)=0.602; y(2)=1.346; y(3)=1.247; y(4)=2.350; y(5)=2.597;
        f(u,v,x)=abs(u+v*x - y(x));
        fit(u,v)=f(u,v,1)+f(u,v,2)+f(u,v,3)+f(u,v,4)+f(u,v,5);
        rand(n,m)=r <-> myrandom(r1), r:=n+(m-n)*r1;
        h()=rand(-5,5)*pow(0.1,rand(1,10));
        g(L,i,j)=M <->
                [[u1,v1],[u2,v2]]:=project(L,[i,j]),
                M:=[[u1+h(),v2+h()],[u2+h(),v1+h()]];
preprocess() <->
        getWp("sheet",Wp), //スプレッドシート表示
        (Wp >0) -> (Wp.g:=Wp)
        otherwise (
                L:=defList(p1(z,x,[]),member(x,genIndex(1,20))),
                makesheet(20,3,Wp1), Wp.g:=Wp1,
                showGeneration(L) );

inputsequence()="d";
times()=1000;
initialstate()=c0 <-> c0:=projectss(Wp.g,1,2,20,3);
delta(c,a)=cc <->
        //世代の開始
        Ws:=defList(p2(z,[u,v],[]),member([u,v],c)), //適応度計算
        c1:=sort(Ws), //適応度順の個体リスト
        showGeneration(c1),
        Ps:=projectMatrix(c1,1,2,10,3), //親を選択
        xwriteln(0,"Ps="),xwriteln(0,Ps),
        L:=[g(Ps,1,2),g(Ps,3,4),g(Ps,5,6),g(Ps,7,8),g(Ps,9,10)],
        Qs:=append(L), //子を確定
        cc:=append(Ps,Qs); //次世代が確定

p1(z,x,[]) <-> z:=[0, rand(-10,10), rand(-10,10)];
p2(z,[u,v],[]) <-> z:=[fit(u,v),u,v];
showGeneration(c1) <->
        appendf("ga11.lib","w", c1), //ファイルに書き込み
        writess(Wp.g,"r",1,1,c1),   //スプレッドシートに書き込み
        [[w,u,v],[z,x,y]]:=project(c1,[1,10]),
        d1:=[y(1),y(2),y(3),y(4),y(5)],
        d2:=[u+v*1, u+v*2, u+v*3, u+v*4, u+v*5],
        d3:=[x+y*1, x+y*2, x+y*3, x+y*4, x+y*5],
        data:=[d1,d2,d3],
        show1(data,plot); //グラフに表示
```

図 4-4-5

本項では，参考文献と同じ遺伝アルゴリズムを別の言語 CAST で書き直し，実行した．実行結果は参考文献の結果と一致している．しかしながら，手続型言語とはまったく異なる形式のユーザモデルになっている．

CAST 言語によるシミュレーション開発のポイントは，何をオートマトンの状態にするかである．今回は，遺伝子プールの行列（20 行×2 列）を状態としたので，ベクトル計算と転置行列をとることにより，見た目は簡単に（文字数は少なく）記述できている．

モデルの開発は，関数 delta(c,a)の定義を 1 行作成するごとにコンパイル・実行し，確認しながら行なえる．例えば，まず，

```
Ws:=defList(p2(z,[u,v],[]),member([u,v],c)),
cc:=Ws;
p2(z,[u,v],[]) <-> z:=[fit(u,v),u,v];
```

で実行し確認する．次に c1:=sort(Ws)を加えて再実行するなどである．

したがって，行列（リストのリスト）の処理に慣れないユーザにとっては，行列がどのように変化していくかを体験しながら開発することができる．もちろん行列処理に慣れたユーザなら，(頭の中の）行列イメージがモデルのコードに正確に対応しているのでモデリングは容易だろう．

第 5 章 複雑系のモデル

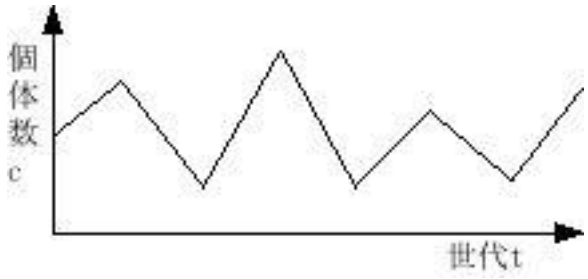
ここでは複雑系に関連したシミュレーションの方法やモデル例を示す.

5.1 一次元カオス

本稿では，1 次元カオス（chaos）のシミュレーションモデルを CAST 言語を用いて構成し実行する．具体的には，係数 c の値を徐々に変化させながら，ロジスティック写像と呼ばれる漸化式 x(t+1) = c*x(t)*(1-x(t)) の計算を行う（t は時刻を表す）．係数が c=2.7 付近では時間が経つと x(t)は 1 点に収束する．c=3.0 付近では時間が経つと x(t)の値は 2 点を行ったり来たりする．c=3.3 あたり以上から何だか分からなくなる．これが**カオス挙動**である．また値 x(t)の変化をグラフ表示し，その表示制限を調査する．

5.1.1 一次元カオス

限られた地域に生息するある種の昆虫の個体数 x(t)は，次の式「ロジスティク写像」にしたがって変化するという．x(t+1) = c*x(t)*(1-x(t))．ただし，t は世代を表し，係数 c は繁殖力の強さを表し，0＜c＜4 である．この式は，ある世代で個体数が増えすぎると，次の世代では減少するという傾向を表している（図表 5-1-1）.



図表 1　個体数の変化

さて初期値 x(0)を適当に定めて計算を行なう．すると，係数 c が小さいときは時間が経過すると個体数 x(t)は 1 点や 2 点に収束する．しかし，係数 c が大きいときは時間が経過しても安定にはならず，つねに「ふらふら」と個体数が変化するようになる．本稿ではその様子をグラフ表示したい.

さて，このような計算と描画を行なうために，モデル理論アプローチではオートマトンを作成する. オートマトンの状態は係数 c とする. ただし, 係数を c = 2.5 から 4.0 まで 0.01 刻みで変化させつつ，500 世代づつを計算する．したがって，オートマトンは 150 回の状態遷移を行なうことになる.

モデル化の方針としては，まず各係数（すなわち状態）c に対して，漸化式 x(t+1) = c*x(t)*(1-x(t))で定義される数列 x(t)のリスト[x(1),x(2),x(3),...,x(500)] を計算する．次に，そこから最後の 100 個を抜き出して「集合」化し b={x(401),x(402),x(403),...,x(500)}を作成する．ここでは，第 2.3 節にもとづき，そのような計算を関数 defSet を用いておこなう．b は集合であるから，もしもすべての要素が同じなら，ひとつの要素をもつ集合となる．そのことは 400 世代以降の個体数が一定である（1 点に収束した）ことを表す．これがモデル化の主たるアイデアである.

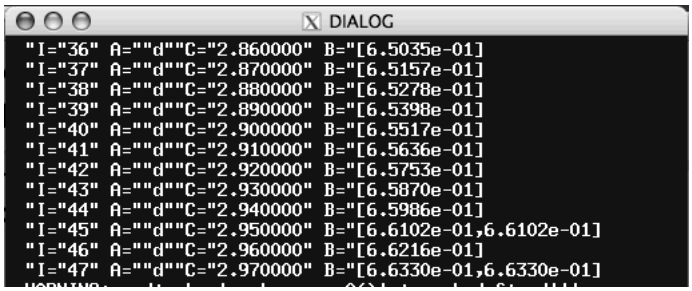
5.1.2　言語 CAST による表現

上記の計算を行なうオートマトンの CAST 言語によるモデルを次のように構成した.

```
//chaos07.set
inputsequence()="d"; times()=150;
initialstate()=2.5;
delta(c,a)=cc <-> cc:=c+0.01;
lambda(c)=b <->
        x.g:=0.5,
        b:=defSet(p(y,t,[c]), member(t, genIndex(1,500)));
p(y,t,[c]) <-> y:=c*x.g*(1-x.g), x.g:=y, t > 400;
```

図表 5−1−2　一次元カオスのモデル

図表 5-1-2 のモデルでは，まず係数 c が 2.5 から始まり（initialstate()=2.5），0.01 づつ大きくなる（delta(c,a)=cc <-> cc:=c+0.01）．主たる計算は出力関数 lambda で行ない，それぞれの c に対する，個体数の収束の様子を見ることになる．性質 p を満足する集合を作成する関数 defSet を使用して, 400 世代以降の個体数の集合 b={x(401),x(402),x(403),...,x(500)}を求めている．また，強制的に時間を進めるためにダミーの"d"を 150 回入力している（inputsequence()="d"; times()=150;）．非常に簡単なモデルである．



図表 5-1-3　実行結果

このモデル（図表 5-1-2）をモデル理論アプローチによるシミュレーション開発実行環境 Simcast でコンパイルし実行したところ，実行時間は 3 秒であった．実行途中の様子が図表 5-1-3 である．係数が c=2.95, 2.97 のとき，収束先が 2 点であることが分かる．

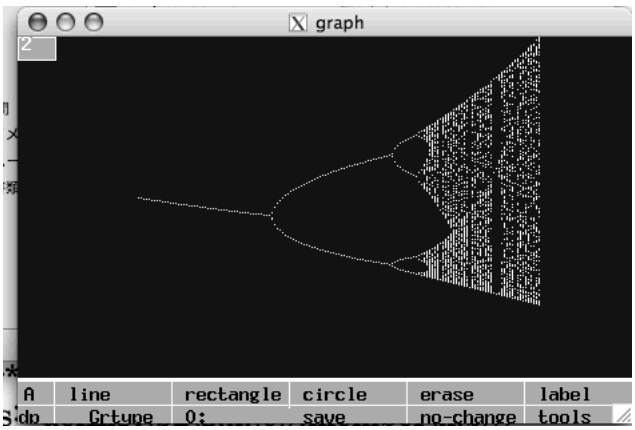
5.1.3　グラフ表示と制限

以上が計算の本質的なところであるが，グラフ表示のために，若干の述語を追加したものが図表 5-1-4 である．

```
//chaos09.set
preprocess() <-> wopen(Wp,"xyplot"),plaingraph(Wp), Wp.g:=Wp;

inputsequence()="d"; times()=150;
initialstate()=c0 <-> c0:=2.5;
delta(c,a)=cc <-> cc:=c+0.01;
lambda(c)=b <->
        x.g:=0.5,
        b:=defSet(p(y,t,[c]), member(t, genIndex(1,500))),
        showPoint(c,b);

p(y,t,[c]) <-> y:=c*x.g*(1-x.g), x.g:=y, t > 400;
showPoint(c,b) <-> Qs:=defList(p2(z,x,[c]),member(x,b));
p2(z,x,[c]) <-> [x1,y1]:=[90,0]+200*[c-2.5,x], xwrite(Wp.g,"r",[x1,y1,x1,y1]);
```

図表 5-1-4　グラフ表示を加えたモデル

まず，事前準備 preprocess として起動直後に xyplot グラフのウインドウを開き，座標軸を非表示にしている．また，述語 showPoint(c,b)でグラフを描いている．
xwrite(Wp.g,"r",[u,v,u,v])は点を描く述語である．ちなみに，Qs はダミーであり常に空である．これを実行すると，次の図表 5-1-5 のようになる．



図表 5-1-5　グラフ表示の結果

この図の原点はウインドウの左上端であり，横軸（右向き）が係数 c で縦軸（下向き）が収束点（400 世代以降の個体数）を示している．図を見ると，c=2.7 付近では時間が経つと 1 点に収束する．c=3.0 付近では時間が経つと 2 点を行ったり来たりする．c=3.3 あたり以上から何だか分からなくなる．このカオス挙動が描かれている．グラフ表示を含めた実行時間は 59 秒であった．

【制限】
使用した xwrite(Wp.g,"r",[x,y,xx,yy])は，ウインドウ左上端を原点として座標(x,y)と(xx,yy)を端点とする長方形 rectangle を描く述語である．**グラフオブジェクト数の制限**に注意する

必要がある．グラフウインドウに描画できるオブジェクト数は，ソースコード extProlog.zip の中のヘッダーファイル Xgraphdefines.h のなかに変数 GRDATA_MAX で規定している．バージョン 0902xx での設定値は 20,000 にした．したがって 2 万個以上の点を描画することはできない．2 万個以上のオブジェクトを描画するには，ソースコードを修正し再コンパイルする必要がある．

本節では，1 次元カオスのモデルを CAST 言語を用いて構成し，モデル理論アプローチによるシミュレーション開発実行環境 Simcast で実際に実行し，収束点の様子を描画した．その結果，それなりの計算はできるし一応のグラフ表示が可能であることが分かった．表示できる点の数（グラフオブジェクト数）に限界があることはそれほど大きな問題ではなかった．

さて，手続き型の通常の計算機言語と CAST 言語との比較を考えよう．漸化式で定義された数列 x(t)の計算そのものは，通常の言語で作成してもそれほど大きなプログラムではない．2 重の繰り返し計算（t の変化と c の変化）を行なえばよい．大きく異なる部分はその後の処理の仕方である．通常の言語では収束点の個数などを求める手続き型のプログラムが別途必要である．しかし，モデル理論アプローチでは，数列を集合 b として作成できるので，図表 5-1-2 のように簡単なモデルとなっている．これは CAST 言語（モデル理論アプローチ）の利点である．

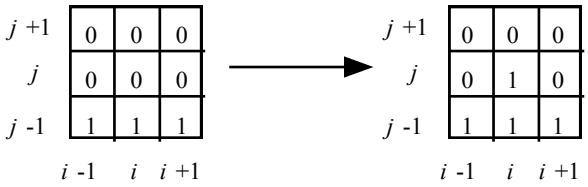
5.2 二次元細胞オートマトン

本節では，二次元細胞オートマトン（Cellular Automaton）を言語 CAST を用いて構成し，モデル理論アプローチによるシミュレーション開発実行環境 Simcast で実際に実行した結果を報告する．本稿では，全体システムの状態を表（行列）で表し，状態遷移は行列計算で行なう．オートマトン 1 回の状態遷移で，30×30=900 個の細胞の 1 サイクル（1 世代）の計算を行なう．

5.2.1 ２次元細胞オートマトン

2 次元平面を考える．各座標 (i, j) に 2 状態オートマトン（正確には 2 状態の状態機械）を配置する．ただし，座標 i, j は整数で状態は 0, 1 を考える．

【システム要素】　それぞれのオートマトンを細胞（cell）と呼ぶ．各細胞の状態は，上下左右に隣り合う 8 つの細胞の状態の影響を受けて変化していく．



図表 5-2-1　細胞(i,j)の状態遷移

例えば，細胞(i,j)の近傍が図 5-2-1 左のようであったとき，次の時刻の細胞(i,j)の状態は 1 であるなどと遷移するのである. このような規則を, 以下では次のように書くことにする.

[[[1,1,1],[0,0,0],[0,0,0]],1]

本稿では，直感的なイメージとして，木の枝が横に 3 伸びたら芽がでる（今後はそこから上方向に伸びていくだろう）という意味の解釈を行う．自身を含めた近傍の様子は可能性として 2^9=512 パターンあるので, それぞれの場合に応じて次の時刻の状態を定めることになる．一つの細胞は複数の規則からなる状態遷移関数をもつ．以下では次の規則群を考える．

Rule.g=[[[[1,1,1],[0,0,0],[0,0,0]],1], [[[0,1,0],[0,0,0],[0,0,0]],1], //縦に伸びる
[[[1,0,0],[1,0,0],[1,0,0]],1], [[[0,0,0],[1,0,0],[0,0,0]],1], //右に伸びる
[[[0,0,1],[0,0,1],[0,0,1]],1], [[[0,0,0],[0,0,1],[0,0,0]],1]];//左に伸びる

ここでは，自身の状態が 0 で，近接する 3 つの細胞の状態が 1 なら，自身も 1 となるように設定した.

【全体システム】　以下では 30×30=900 個の細胞を考え，すべての細胞の状態遷移関数は同じであるとする. 状態機械の複合システムは, 全体としてひとつの状態機械であるから, 全体をひとつの状態機械と考えることができる．全体システムの状態は，値が 0 か 1 であるような，30×30 の行列となる．状態遷移につれて，その表の中の数値が書き換えられていくのである．1 回の状態遷移で 900 個すべての細胞の状態を書き換える可能性もある.

5.2.2　CAST 言語による表現

上記の全体システムの CAST 言語によるモデルは次のように構成できる.

```
/*cellular20.set ( 2 次元細胞オートマトン）　*/
preprocess() <->
   M.g:=30, N.g:=30,
   MN.g:=product(genIndex(2,M.g-2),genIndex(2,N.g-2)),
   Rule.g:=[
        [[[1,1,1],[0,0,0],[0,0,0]],1], [[[0,1,0],[0,0,0],[0,0,0]],1], //縦に伸びる
        [[[1,0,0],[1,0,0],[1,0,0]],1], [[[0,0,0],[1,0,0],[0,0,0]],1], //右に伸びる
        [[[0,0,1],[0,0,1],[0,0,1]],1], [[[0,0,0],[0,0,1],[0,0,0]],1] ];//左に伸びる
inputsequence()="d";
times()=30;
initialstate()=c0 <->
        L:=constantlist(0,[M.g,N.g]),
        c0:=replaceMatrix(L,[[1,5,1],[1,6,1],[1,7,1]]);
delta(c,a)=cc <->
        Ls:=defList(p(z,[i,j],c),member([i,j],MN.g)), //変更チェック
        xwriteln(0,"Ls=",Ls),
        cc:=replaceMatrix(c,Ls);   //変更実施
p(z,[i,j],c) <->
        N:=projectMatrix(c,i-1,j-1,i+1,j+1),
        member([N,v],Rule.g), z:=[i,j,v];
```

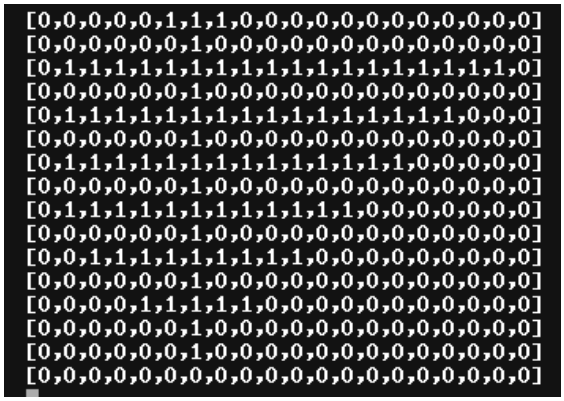
図表 5-2-2　細胞オートマトンの 2 次元モデル

全体システムの実行手順は簡単である．まず initialstate で初期状態 c0 を定め，次に delta で次の状態 cc を計算する. 強制的に時間を進めるためにダミーの"d"を 30 回入力している.

初期状態を定める initialstate の定義では，constanlist 関数で 0 が M.g×N.g 個（30×30 個）並んだ行列 L を作り, replaceMatrix 関数で 7 列目の 3 つの要素だけを 1 に書き換えている. その結果の行列 c0 が初期状態である.

実際の全体システムの状態遷移を定義している delta では，変化すべき細胞のリストアップ（変更チェック）と状態（リスト）の変更を実施する．各細胞の状態遷移関数は，グローバル変数 Rule.g で与えている. この規則に従って, 細胞(i,j)の状態を書き換えるのが delta である．近傍の状況がどれかの規則に一致するかどうかを述語 member([N,value],Rule.g)で判定し，もしも一致するものがあれば変数 v に値が取得できるので，z:=[i,j,v]を変更リスト Ls に並べている．最後に，関数 replaceMatrix が変更リスト Ls をもとに新しい 30×30 の行列を作成している．これにより，900 個の細胞の次状態 cc が定まる.

このモデル（図表 5-2-2）をモデル理論アプローチによるシミュレーション開発実行環境 Simcast で実際に実行した結果が図表 5-2-3 である.



図表 5-2-3　実行結果

5.2.3 高速化とグラフ表示

1）スプレッドシートによる高速化

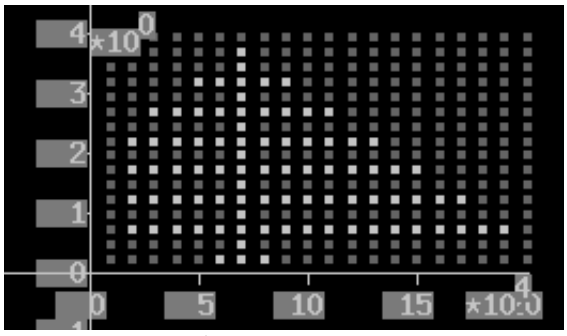
実行時間が遅いことは，入門者・初心者には問題ではないが，熟練した研究者にとっては欠点となる．前節では，細胞数 30×30=900 の状態遷移を 30 回繰り返すのに 90 秒かかっていた（状態遷移 1 回あたり 3 秒）．高速化をはかるために，第 3.4 節を参考に，スプレッドシートを状態の保存場所とする方法でモデルを書き直したものが付録 5.2.1 である．付録 5.2.1 を実行すると 30 回繰り返すのに 4 秒（一回あたり 0.13 秒）であった．

2）色付き xyplot グラフによる表示

また，次の出力関数 lambda を加えて色付き xyplot グラフで表示したものが図表 5-2-4 である．

```
lambda(c)=b <-> b:=0,
      S:=defList(p3(z,[i,j],[]),member([i,j],MN.g)),
      L:=transpose(S),
      show3(7,L,"xyplot","color");
p3(z,[i,j],[]) <-> z:=[j,i,ssr(Wp.g,i,j)+3];
```

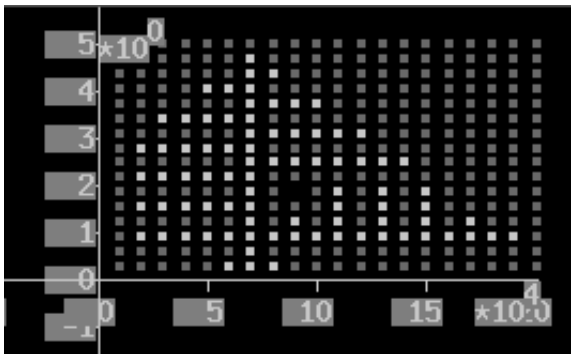
（ただし，SS のセル幅が 11 に戻るので SS を状態表示には使用しない．そのかわりとしてグラフを使うのである．cellular37.set を参照のこと）．



図表 5-2-4　色付き xyplot グラフ表示

下から上に向かって樹木が生長していくようすを動的に表示することができる．このモデル（細胞数 30×30=900）を実行すると， 30 回繰り返すのに 8 秒（一回あたり 0.27 秒）で

あった.



図表 5-2-5　障害物がある場合

障害物があり成長が阻害される場合を実行したい場合は，SS の初期値を定める事前準備 preprocess に，ssw(Wp,6,9,2)などを付加すればよい．ここでは，障害物としてセル(6,9)の状態をルールからはずれた 2 とした．実行結果は図表 5-2-5 のようになった．横に伸びることが阻害されても，いづれ縦に伸びていくようすが分かる（cellular37a.set を参照）.

2）細胞数の増加

【xyplot グラフ】今のところ，述語 show3 を使ったグラフの表示限界が 30×30=900 であるため，細胞数が増加すると，もはやグラフで表示するすることはできない．lambda の定義をコメントアウトし，グラフを表示しないで，細胞数を増加させることになる.

【スプレッドシート】グラフを表示しなければ，SS の書き込み限界が 200×600=120,000 セルなので, 原理的には 12 万個の細胞からなるオートマトンを実行することが可能である. 100×100 までは確認した（cellular35.set 参照）. 状態遷移 1 回が 1 秒である. しかしながら，初期値を SS に代入するとき，constantList の限界が 140×140 なので，これを超えるサイズなら部分的に 200 個の 0 のリストを作り，SS に何回か代入することになる．しかし，変更すべき細胞のリスト L が大きくなると処理限界に達する可能性はある.

【ファイル保存】大きなサイズの SS をパソコン画面に表示させることはできないので，動的に変化するシステム全体の状況をリアルタイムに把握することはできない．ファイルにデータを逐次保存しておき，シミュレーション終了後に市販ソフトなどを利用して，後からグラフ表示することになろう.

細胞数 30×30=900 の SS 全体を述語 appendf でファイルに保存しながら， 30 回繰り返すのに 5 秒であった．しかし，細胞数 100×100=1 万個の場合は，SS 全体を述語 xread で読み込み，述語 appendf でファイルに書き込むとき，エラーが発生する．サイズ 100×100 の行列を変数に代入できないので書き込めないようである．書き込みが中途半端に終わっている．これを回避するには，SS 全体でなく，変化すべき細胞の位置と値 [i,j,v]のリスト cc のみをファイルに保存しておき，シミュレーション終了後に，全体を再現する方法が考えられる.

また，C 言語の分かる技術者なら extProlog そのものを拡張することができる．ソースコード extProlog.zip の中のヘッダーファイル prologdefines.h に大域変数が定義されているので, 変数 MAX__LISTTEXT を 21000 に変更すると,

・サイズ 100×100 の行列をテキストウインドウに表示できる.
・サイズ 100×100 の行列をファイルに保存することができる.

ただし, ソースコードを変更すると不具合が発生することは避けられない. 特に述語 show3 は完全に利用できなくなるので注意せよ.

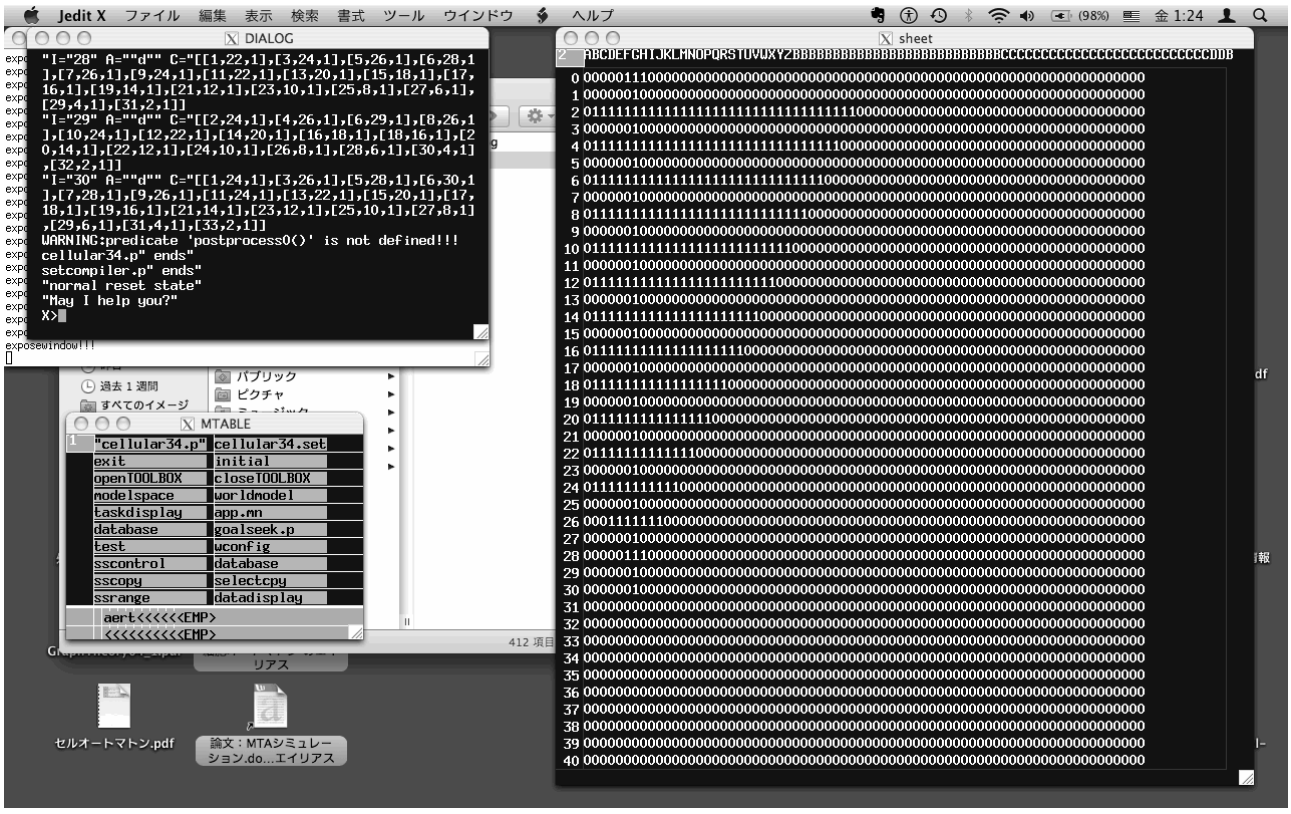
本稿では，細胞オートマトンを CAST 言語を用いて構成し，モデル理論アプローチによるシミュレーション開発実行環境 Simcast で実際に実行した.

一応，グラフ表示しなければ多数の細胞数でもシミュレーションは可能である．細胞の数が増えると処理速度は遅くなるが，スプレッドシートを利用すると高速化することができる．もしもグラフ表示したいならば表示限界があるので注意が必要である.

【宿題】人工生命（Artificial Life）研究の発端となった"Life Game"について調べ，そのゲームを作成してみよ.

付録 5.2.1　スプレッドシートによる高速化

```
//cellular34.set ( 2 dim. cellular automaton w/ SS only)
preprocess() <-> //SS ウインドウを開き，初期値を書き込む
        MM.g:=30, NN.g:=30,
        MN.g:=product(genIndex(1,MM.g-1),genIndex(1,NN.g-1)),
        Rule.g:=[
        [[[1,1,1],[0,0,0],[0,0,0]],1], [[[0,1,0],[0,0,0],[0,0,0]],1], //縦に伸びる
        [[[1,0,0],[1,0,0],[1,0,0]],1], [[[0,0,0],[1,0,0],[0,0,0]],1], //右に伸びる
        [[[0,0,1],[0,0,1],[0,0,1]],1], [[[0,0,0],[0,0,1],[0,0,0]],1]], //左に伸びる
        getWp("sheet",Wp),
        (Wp>0)->(Wp.g:=Wp)
        otherwise (
                openlocalsheet("",1,0,NN.g,MM.g,x1,y1,1,1,1,Wp1), Wp.g:=Wp1,
                writess(Wp.g,"r", 1,1, constantlist(0,[MM.g, NN.g]) ),
                replacess(Wp.g,[[1,5,1],[1,6,1],[1,7,1]])
        );
inputsequence()="d"; times()=30;
initialstate()=c <-> c:=[ ];
delta(c,a)=cc <->
        cc:=defList(p1(z,[i,j],[]),member([i,j],MN.g)), //変化チェック
        replacess(Wp.g,cc);
p1(z,[i,j],[]) <->
        N:=projectss(Wp.g,i-1,j-1,i+1,j+1),
        member([N,v],Rule.g ), z=[i,j,v];
```



5.3 じゃんけん集団の進化ゲーム

5.3.1 進化的安定性

本稿では，進化ゲーム（Evolutionary Game）のシミュレーションモデルを CAST 言語を用いて構成し実行する．具体的には，競合関係が三すくみの関係にある 3 種類の生物の人口分布の変化を動的に計算する[ﾇ]．三すくみの関係とは，生物 1 は生物 2 より強く，生物 2 は生物 3 より強く，生物 3 は生物 1 より強いことを言う．（じゃんけんでの，グー，チョキ，パーの関係に相当する．）ここでは，人口分布のバランスが崩れた場合に，その後の人口分布がどのように変化してゆくかをシミュレートする．例えば一時的に生物 1（いつでもグーだけ出す人）の人口が増加したと仮定しよう．しかしながら，餌となる生物 3（チョキ）が少なくなっていることに注意しなければならない．餌に出会えない生物 1 の子孫は減少していくかもしれない．また，生物 3（パー）は生物 1（グー）より強いので，確実に餌にありつけて子孫を残すことができるだろう．時間が経過すると，最終的にはバランスのとれた人口分布に戻るだろう．このバランスのとれた状態を（多型集団の）**進化的安定状態**（Evolutionarily Stable State: ESS）と呼ぶ.

5.3.2 進化ゲーム

限られた領域内で餌をめぐって競合している 3 種の生物 1,2,3 が混在する生態系（多型集団）を考える．全個体数は無限とし，現在の人口分布を c = (c1, c2, c3)とする．ただし，c1+c2+c3=1 である．各個体は 1 対 1 ランダム対戦を行ない，適応度 w1, w2,w3 が定まることとする．次世代の人口分布 cc = (cc1, cc2, cc3)は，各生物の適応度が高い方がより多く子孫を残せるように決まるものとする（図表 5-3-1）

現世代の人口分布 c = (c1,c2,c3)
↓
ランダム対戦
↓
適応度 w = (w1,w2,w3)
↓
次世代の人口分布 cc = (cc1,cc2,cc3)

図表 5-3-1　人口分布の変化

具体的には次のように定式化する.

w1 = 5*c2 -4*c3 +20, w2 = -7*c1 +8*c3 +20, w3 = 2*c1 -c2 +20,　--- (1)
w = c1*w1+c2*w2+c3*w3,　--- (2)
cc1 = c1*w1/w, cc2 = c2*w2/w, cc3 = c3*w3/w .　--- (3)

式(1)における値 20 は生得的な適応度である．それらにランダム対戦後の餌の獲得量を加算している．係数の正負から競合関係が三すくみの関係にあることも分かる．式(2)は適応度の平均値 w を求めるものであり，式(3)は適応度が高いほど次世代に多くの子孫を残せることを表している.

[ﾇ] J.メイナード・スミス『進化とゲーム理論：闘争の論理』産業図書 (1985)

さて，このような計算と描画を行なうために，モデル理論アプローチではオートマトンを作成する．オートマトン（正確には状態機械）の状態は人口分布とする．c = (c1,c2,c3). 一回の状態遷移で 1 世代が経過するものとする．ここでは 1000 世代の変化を見たいので，オートマトンは 1000 回の状態遷移を行なうことになる．

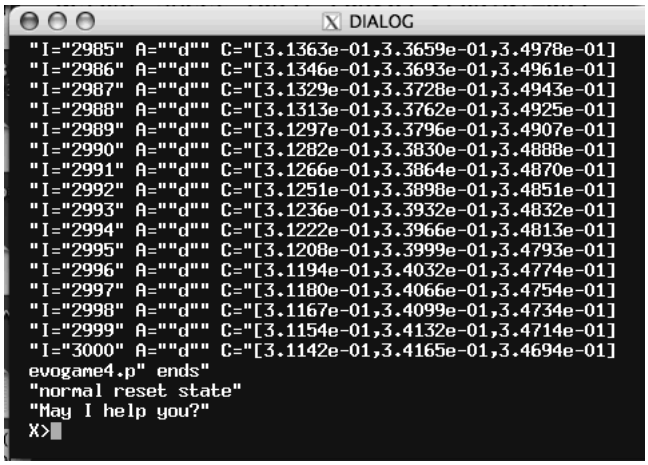
5.3.3 言語 CAST による表現

上記の計算を行なうオートマトンの CAST 言語によるモデルは次のように構成できる．

```
//evogame1.set
inputsequence()="d"; times()=3000;
initialstate()=c0 <-> c0:=[0.5,0.3,0.2];
delta([c1,c2,c3],a)=cc <->
   c:=[c1,c2,c3],
   d1:=[0,5,-4], d2:=[-7,0,8], d3:=[2,-1,0],
   [w1,w2,w3]:=sum([d1*c,d2*c,d3*c])+20,
   w:=c1*w1+c2*w2+c3*w3, r:=1,
   [cc1,cc2]:=[c1+(r*c1*(w1-w)/w),c2+(r*c2*(w2-w)/w)],
   cc:=[cc1,cc2,1-cc1-cc2];
```

図表 5-3-2　進化ゲームのモデル

図表 5-3-2 のモデルでは，まず初期状態が c0:=[0.5,0.3,0.2]から始まる．状態遷移関数 delta は，前述の式(1), (2), (3)を計算している．また，強制的に時間を進めるためにダミーの"d" を 3000 回入力している（inputsequence()="d"; times()=3000;）．図表 5-3-3 は Simcast でコンパイルし実行した結果である．3000 回の状態遷移を行ない，最終的に c = (0.31142, 0.34165, 0.34695)が得られている．

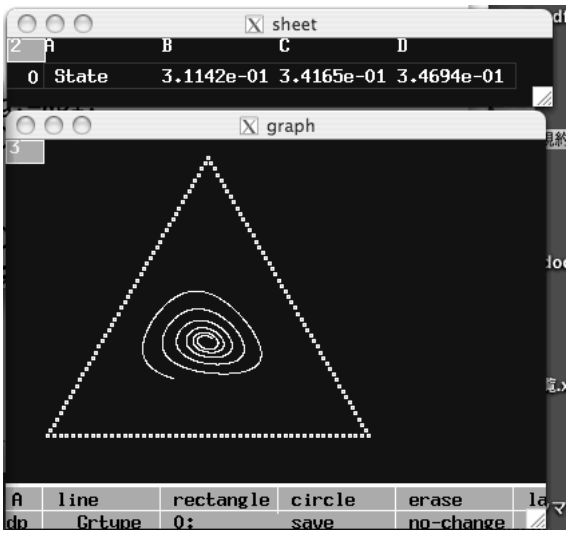


図表 5-3-3　実行結果

図表 5-3-2 のモデルでは，式(3)に無かった変数 r がある．cc1 の定義式を展開すれば分かるように, r=1 のときは式(3)と同じであり, 離散時間モデルを表わしている．ここで, r=0.01 などに変更した場合は，連続時間モデル（微分方程式を離散化したもの）になる．その結果，数値は少しずつ変化するようになるだろう．

5.3.4　グラフ表示

以上が計算の本質的なところであるが，グラフを表示するために，若干の述語を追加したものを付録 5-3-1 に掲載した．付録 5-3-1 のモデル evogame5.set を Simcast でコンパイルし実行すると，図表 5-3-4 のように 3 角形の中に渦巻き状の図形が描かれていく．



図表 5-3-4　グラフ表示の結果

図表 5-3-4 の 3 角形の
　下左端は生物 1 が多い状態　c = (1,0,0),
　下右端は生物 2 が多い状態　c = (0,1,0),
　上端は生物 3 が多い状態　　c = (0,0,1)
に対応している．描画の結果を見ると，生物 1 が多い初期状態から出発し次第に生物 3 が増加して行く．しかし，ある程度生物 3 が増加すると，こんどは生物 2 が増加していく様子が分かる．時間が経過するにつれ，外側から内側に渦巻き状に点が描かれて行き，最終的には 3 角形の中心, つまり人口分布が(1/3,1/3,1/3)に近づいていくことが分かる. これは, c=(1/3,1/3,1/3)が**進化的安定状態**であることを意味する．

CAST 言語による付録 5-3-1 のモデル evogame4.set では，まず，事前準備 preprocess として起動直後にスプレッドシートとグラフウインドウを開いている（述語 makesheet と wopen）．また plaingraph(Wp2.g)はグラフウインドウの座標軸を消去する述語である．さらに，関数 defList を使って外枠の 3 角形のグラフを描いている．ただし，変数 u はダミーであり常に空である．

スプレッドシートは現在の状態を数値で表示するためのものである．時間の経過とともに状態 c = (c1,c2,c3) が変化していく様子が表示される．しかし，3000 回の状態遷移が終了したあと，再び evogame5.set を実行すると「続きの 3000 回」を実行することができるように，初期状態をスプレッドシートから読み込むように定義している（initialstate()=c0 <-> [c0]:=**projectss**(Wp.g,1,2,1,4)）.

グラフウインドウは，状態 c の遷移を図形として描くためのものである．述語 showPoint(c)を作成し，3 角形の中に点を描くことができる．変数 c=(c1,c2,c3)は 3 次元空間の点であるが，c1+c2+c3=1 であるので，実際には空間座標(1,0,0), (0,1,0), (0,0,1)を端点とする 3 角形（空間上の平面）上の点である．したがって，showPoint(c)は，点 c=(c1,c2,c3)をグラフウインドウ（2 次元）上に描くための座標変換（ (c1,c2,c3)→(x,y) ）を行っている．

本節では，進化ゲームのモデルを CAST 言語を用いて構成し，モデル理論アプローチによるシミュレーション開発実行環境 Simcast で実際に実行し，状態遷移の経過を描画した．その結果，それなりの計算はできるし一応のグラフ表示が可能であることが分かった．しかしながら，外枠の 3 角形を描くのに複数の点で表示したが，直線を描く適切な述語が必要である．

付録 5.3.1 グラフ表示を含むモデル

```
//evogame5.set
preprocess() <->
   getWp("sheet",Wp), (Wp>0)->(Wp.g:=Wp)
      otherwise ( makesheet(2,5,Wp1), Wp.g:=Wp1,
      writess(Wp.g,"r",1,1,[" State",0.5,0.3,0.2]) ),
   getWp("graph",Wp2), (Wp2>0)->(Wp2.g:=Wp2)
      otherwise ( wopen(Wp3,"xyplot"), plaingraph(Wp3), Wp2.g:=Wp3 ),
   u:=defList(p(y,x,[]),member(x,genIndex(0,50)));
p(y,x,[]) <->
   showPoint(2,[1-x/50,x/50,0]),   //3 角形を描く
   showPoint(2,[0,1-x/50,x/50]),
   showPoint(2,[x/50,0,1-x/50]);

inputsequence()="d"; times()=1000;
initialstate()=c0 <-> [c0]:=projectss(Wp.g,1,2,1,4);
delta([c1,c2,c3],a)=cc <->
   c:=[c1,c2,c3], d1:=[0,5,-4],d2:=[-7,0,8],d3:=[2,-1,0],
   [w1,w2,w3]:=sum([d1*c,d2*c,d3*c])+20,
   w:=c1*w1+c2*w2+c3*w3, r:=0.1,
   [cc1,cc2]:=[c1+(r*c1*(w1-w)/w),c2+(r*c2*(w2-w)/w)],
   cc:=[cc1,cc2,1-cc1-cc2];
lambda(c)=b <-> b:=0,
   showPoint(0,c),
   writess(Wp.g,"r",1,2,c);

showPoint(z,[c1,c2,c3]) <->
   aX:=30, aY:=220, length:=240,
   x:=aX +((1-c1+c2)*length/2),
   y:=aY -(c3*sqrt(3)*length/2),
   xwrite(Wp2.g,"r",[x,y,x+z,y+z]);
```

巻末付録

巻末付録 1 ダウンロードとインストール

開発・実行環境 Simcast は次のサイトで公開されている.

http://www2.toyo.ac.jp/~asahi/research/simulation/

ここにアクセスし「ダウンロード&インストール」ページに進むと,

【Simcast09 ダウンロード】
1）必要システム　　Unix OS (Linux Fedora, Mac OS X) で動作中
2）シミュレーション開発・実行環境
・extProlog.zip（バージョン 090821)---1,445,500byte
・simcast.zip　（バージョン 090921)---299.423Byte

上記のように表示されるので, 2 つのファイル extProlog.zip と simcast.zip をダウンロードすること.

【インストール手順】
1）simcast.zip を解凍するとディレクトリ simcast ができる．そのディレクトリを適当な場所におく．どこでもよい.
　もしもディレクトリ Mac0S_X ができても不要なので削除する.
2）extProlog.zip を解凍するとディレクトリ extProlog ができる.
　そのディレクトリの中で **make** し，実行可能ファイル xsheet を作る.
3）実行可能ファイル xsheet をディレクトリ simcast に複写する.
4）ディレクトリ simcast の中で xsheet を起動し，動作確認として jar.p が動けばインストール完了である.

【make の方法】
extProlog.zip を解凍してできるディレクトリ extProlog 内を見ると，Makefile が並んでいる.

Makefile
Makefile for PC
Makefile for Mac

Mac OS の場合は Mac アプリケーション開発環境 Xcode が必要である（最新の Mac OS には，あらかじめバンドルされている）．前提となる Xcode があれば，Makefile for Mac を使用して make できる．そのためには，一番上の Makefile を別のファイル名（例 Makefile.org）に名称変更し，Makefile for Mac を Makefile に名称変更すればよい．変更後, ターミナルウインドウを開き, make を実行すると実行ファイル xsheet が生成される.

Linux OS (Fedora 13)の場合は, 適当なテキストエディターで一番上の Makefile を開き, その内容の USE_POSTGRES マクロおよびリンクオプションの-lpq を外す。つまり,

```
CFLAGS = -g -o2 -DLinux -I/usr/X11R6/include -L/usr/X11R6/lib #for Linux
LFLAGS = -lX11 -ltermcap -lm #default
```

ように 2 ヶ所だけ変更する。変更後，ターミナルウインドウを開き，make を実行すると実行ファイル xsheet が生成される．

巻末付録 2 トラブルシューティング

インストール関連

Q. xsheet が make できません.
A. Makefile の内容を変更してみてください.
1) Linux フルインストールでない場合（たとえば, SQL をインストールしていないとき）,
　　付録 1 にしたがって Makefile の中の sql に関する指定を外してください.
2）MTA サイト[1]に Ubuntu での Makefile の指定がありました.

Q. tps.zip が解凍できません.
A. このサイト「CAST 言語でシミュレーション」では tps.zip を使用しません.

---

extProlog が起動して，青い画面がでているとき.

Q. Dialog ウィンドウで「ファイル名.set＋enter」しても，normal reset state に戻ります.
　そして，terminal window に，line 1: command not found となります.
　さらに，ファイル名.p の中身がありません.
A. Macintosh なら改行が CR のままになっているかも知れません．改行を LF に指定して保存してください.

言語 CAST 関連

Q. CAST 言語の解説はどこにありますか.
A. 教科書第４章またはMTAサイト[1]「ダウンロード」ページの最終行から入手できます.

Q. CAST 言語の教科書未記載のコマンド（述語）はどこで調べてるのですか.
A. MTA サイト[1]「背景資料」の(.p)時代の資料「マニュアル第２版」に extProlog の述語がありますので，それを試して成功した場合に紹介してます.

Q. グローバル変数にグローバル変数の値を直接に代入すること(B.g:=A.g)ができません.
A. ローカル変数を経由してください．X:=A.g, B.g:=X とします.

モデル作成

Q. 暴走したとき，どうすれば止められますか
A. terminal ウインドウをクリックして,
　　cntrl キーを押しながら c キーを押し下げてください.
　選択を要求されますので，１，２，３，４のどれかを入力してください.
　わたしは，常に４（quit/exit）をしています.

Q. モデル作成の解説はどこにありますか.
A. オートマトンは教科書[1]第３章，問題解決システム（Solver）は第５章にあります.

Q. なぜ以前はオートマトンだけ，inputsequece0()などとゼロをつけていたんですか.
A. 以前は述語 initialstate が問題解決システムと衝突し，同時に２種類が動かなかったから 0 をつけていました．その後工夫して Simcast09 以降はゼロをつけなくなりました.

Q. 簡略に関数定義すると失敗します.
 func([out]);
 out("temp.lib")=cardinality(temp.lib);　が失敗しました.
A. 裏技を多用しないでください．教科書通りに,
 func([out]);
 out("temp.lib")=N <-> N:=cardinality(temp.lib);
 と定義してください.

Q. 問題解決システム（Solver）やオートマトンを動かすと，
　preprocess() is not defined!
　と表示されます.
　いちおう，表示されても正しく動きますが，異常ありませんか.
A. これは異常ではありません.
　予想したものがないときに，「それはない」と言っているだけです.
　実は，教科書未記載ですが，Solver でも「述語」として事前処理と事後処理を定義できます.
　定義してなくても，構いません.

あとがき

本書では，筆者らが開発したシミュレーション開発実行環境 Simcast の概要紹介とシミュレーション開発方法論およびその適用例を紹介した．その結果次の３点が判明した．

１）Simcast の適用範囲が非常に広いこと．
２）すべてのモデルは同じ言語 CAST で記述できて，しかも多くのモデルが簡単なこと．
３）複雑なシステムでも，一貫した同じ原理に基づいて段階的に多層的に構築できること．

これらの利点は，言語 CAST の柔軟性および Simcast がオートマトンだけでなく問題解決システムも活用できることに起因するものである．

現時点で残された問題点は，第 3.6 節で見たようにオートマトンから Solver を呼び出せるが，しかし Solver の中から別の Solver を呼び出せないことである．回避策として，与えられた Solver をオートマトン化して結合することは可能である．これは，extProlog をオブジェクト指向言語に改変すれば根本的に解消できるかも知れないが，今後の課題である．

著者：旭　貴朗（東洋大学経営学部教授），
高原康彦（東京工業大学名誉教授），
齋藤敏雄（日本大学生産工学部教授），
柴　直樹（日本大学生産工学部教授）

索引