# 状態遷移図に基づく対話型アニメーション作成ツールの提案

岡 本 秀 輔[†] 鎌 田 賢[†] 中 尾 隆 司[††]

アニメーションは，Web ページの例で代表されるように，情報伝達手段の重要な選択肢の 1 つである．特に，キーやマウスなどのユーザ入力によって，表示内容が変化するような対話型のアニメーションは，ユーザの興味を引きつける．しかし，ひとたびこのような対話型アニメーションを作ろうとすると，芸術的な素養のほかに，プログラミング技術やグラフィックスの知識が必要となり，素人には敷居が高すぎる．そこで，我々は小学生の利用も視野にいれた，対話型アニメーション作成ツールの設計と実装を行っている．ユーザは，GUI エディタで状態遷移図と表示画像を指定することにより，オブジェクト指向モデルに沿った形で，アニメーションに登場するキャラクタの動きを決めていく．そして，インタプリタにより動作を確認し，トランスレータにより Java や JavaScript といった対象コードへの変換を行う．変換の際には，Java の内部クラスや JavaScript の関数ポインタを用いることで，状態遷移図と対象コードの対応関係を保たせている．副産物として，このツールは楽しみながら情報処理を学ぶための教材となる可能性がある．状態遷移図を作成して，その動きをイメージすることは，情報処理教育の導入に適している．また，アニメーションに対応する中間言語表現や，対象形式への変換例を見ることは，プログラミングを含めた次のステップの教育に役立つ．

# Proposal of an Interactive Animation Authoring Tool Based on State-transition Diagrams

Shusuke Okamoto,[†] Masaru Kamada[†] and Takashi Nakao[††]

Animations provide an efficient means for presenting information as practiced on modern Web pages. Interactive animations are especially attractive, because these can show variable scenarios according to key and mouse inputs by viewers. Author of an interactive animation is required of two different abilities. One is the artistic imagination as any artistic work requires. The other is highly advanced knowledge and skill in programming and graphics. The latter requirement is a strong barrier for potential authors who have good imagination and no programming skills. In this paper, design and implementation of a tool for authoring interactive animations are reported. To make it intuitively comprehensible for non-programmers, the modern paradigm of object-oriented modeling and the classical state-transition diagram are employed. An animation is constructed as a collection of characters. Behavior of each character is specified by a state-transition diagram and pictures to express its appearance. The tool consists of GUI editors for diagrams and pictures, a built-in interpreter to test the animation, and a translator into target codes including Java and JavaScript. The translation mechanism keeps the coherency between diagrams and target codes, by utilizing inner classes in Java and function pointers in JavaScript. As a side effect, the present tool works also as an educational material for enlightening children on computer science. They may learn how to define dynamical objects while having fun with animations.

## 1. はじめに

アニメーション（以下単にアニメとする）とは，少しずつ内容の変化する一連の絵を連続的に表示することで，絵に描かれた内容を，それが動作しているように見せる技法である．もともとは映画やテレビのために利用されて来た技法であるが，昨今，Web ページや，情報端末などのインタフェースにおけるアニメの利用が増え，その重要度も増している．これらのコンピュータによって表示されるアニメは，ユーザからのキー入力やマウス入力によって，表示内容が変化する場合があり，そのような対話型のアニメは，単に表示される絵を楽しむだけでなく，ユーザがアニメの進行に関与するという楽しみによって，よりユーザの興味を引きつけるものとなっている．また，アニメを何か

† 茨城大学工学部情報工学科
Department of Computer and Information Sciences, Ibaraki University
†† 日立エンジニアリング株式会社技術開発部
Hitachi Engineering Co., Ltd.

しらの解説などに使用している場合には，ユーザの理解度に沿った進行や解説内容の変更が可能となる．

従来からのアニメは，連続的に表示される一連の絵の並びである．アニメに登場するキャラクタの動作は，1つのタイムラインに沿った固定のシナリオによって決められている．したがって，アニメを表示するためのコンピュータ制御はほとんど不要である．

対話型アニメの1つとして，Macromedia 社の Flash [1] や Adobe 社の LiveMotion [2] によって作られたアニメがある．これらでは，ユーザがアニメに対して要求を出すことが許されている．ユーザの選択によって，他のタイムラインへの切替えが生じ，別のシナリオを表示する結果となる．Flash を用いた対話型アニメの作成では，画像や動画のファイルを指定して，フレームと呼ばれるアニメのコマ割りを決定し，複数のタイムラインを構成する．ただし，この方法だけでは，複雑な動作の指定が繁雑となる．そこで，Flash5 からは，ActionScript という，JavaScript を起源とするプログラミング言語を用いて，アニメ作成ができるように拡張され[3]，さらに，ActionScript 2.0 はオブジェクト指向言語となった．つまり，Flash による対話型アニメ作成は，プログラミングともいえる．

他の種類の対話型アニメには，Broderbund 社の絵本の CD [4] に代表されるようなアニメがある．この種のアニメでは，もととなる絵本の1ページの中で，話の展開に沿った各キャラクタの動作を表示する．それとともに，ユーザのマウス入力によって，いわゆる，“隠れキャラ” を出現させ，一連の動作を一時的に表示する．

コーシングラフィックシステムズ社のキューティマスコット++ [5] は，このタイプのアニメ作成を可能とする開発環境である．キューティマスコット++では，オブジェクト指向パラダイムに沿って，対話型のデスクトップ・マスコットを作成する．各マスコットの定義は，1つのメインルーチンと複数のサブルーチンからなる．メインルーチンには，デフォルト状態で循環的に変化するマスコットの動作と表示すべき絵を指定する．サブルーチンにも同様の指定をするが，これらは，他のマスコットとの衝突やユーザ入力といったイベントによって開始される．

イベントをきっかけとして，あるサブルーチンが1回だけ呼び出され，一連の動作を処理した後に，必ずメインルーチンへと制御が戻る．この方式は，単純で理解しやすいが，1つの大きな欠点ともなっている．それは，サブルーチン呼び出しの履歴が保存されないために，状態が変化していくマスコットを簡単に作成できない点にある．たとえば，いも虫から繭を作り蝶へと変化するようなマスコットの定義が簡単にはできない．もう1つの欠点は，マスコットの定義が抽象化されていないために，同一の振舞いをするマスコットを10個作成するには，10個の同一定義を作らねばならないことである．

これまでの議論から，対話型アニメの作成には，オートマトン理論に基づいて，各キャラクタの動作を状態遷移図で指定する方法も有効ではないかと考えられる．なかば当然と思われるこの結論にもかかわらず，そのような原理に基づく対話型アニメ作成ツールは見当たらない．そこで我々は，状態遷移図によってアニメに登場するキャラクタの動作を定義することで，対話型アニメを作成するツールを提案する．各キャラクタの状態は一定時間ごとに遷移し，他のキャラクタとの衝突や，ユーザからの刺激といったイベントによって，遷移する次の状態が決まる．

状態遷移図は，丸（○）と矢印（→）という単純な図形で全体を表現する．そのため，GUI エディタによるビジュアル・プログラミングが可能となる．単純な記述ルールの図形をプログラミングに用いることで，小学生でさえも対話型アニメの作成方法を理解できるはずである．また，状態遷移図を作成して，その動きをイメージすることは，情報処理教育の導入に適している．そのため，ワープロソフトや表計算ソフトの使い方といった情報リテラシ教育に偏りがちな情報教育の導入に，オートマトン理論などの理論面の内容を含めやすくするという効果も期待できる．

## 2. プログラミング・モデル

我々の対話型アニメの指定は，キャラクタの動作クラスの定義と，どのキャラクタを同時に開始させるかを規定したグループ定義，そして全体構成のパラメータからなる．

動作クラスの定義は，状態遷移図を用いてムーア型機械（または順序回路）と同等の指定をする．入力をマウス/キー操作や他のキャラクタとの衝突といったイベント，出力をキャラクタの絵の表示にそれぞれ対応させると，状態遷移図における状態，遷移，入力，出力の関係は次のようになる．一定時間ごとに，入力とは無関係に内部状態に対応する出力がなされる．同時に入力の内容や有無によって，次の状態へと内部状態が変化する．

具体的な出力としては，画像ファイルとアクションを指定する．アクションには，指定位置やランダムな位置への移動，現在位置から相対的な位置への移動，他

のキャラクタを追従する形の移動，グループ単位の新しい他のキャラクタの生成，アニメの終了などがある．

一方，入力は状態の遷移条件として指定する．具体的な入力としては，マウス入力，キー入力，キャラクタと壁との衝突，キャラクタどうしの衝突，同一状態を繰り返した回数，乱数に対応する閾値などである．複数の入力に対応するイベントが生じた場合には，あらかじめ決められた優先順位により，ただ 1 つのイベントが選択される．入力がない場合のデフォルトの遷移も指定しておく．また，ある状態から他の動作クラスとその初期状態を指定することで，階層的な状態遷移の指定も可能となっている．

グループ定義では，状態遷移として定義した動作クラスの名前と，そのクラスにおける初期状態の名前の組を，1 つのキャラクタの生成の指定として扱い，これを 1 つ以上指定する．`main` という名のグループで指定したキャラクタが，アニメの開始時に生成される．

全体構成のパラメータとしては，表示画面サイズと背景画像および状態遷移を起こす時間間隔がある．

## 3. GUI エディタ

動作クラスは**図 1** に示すエディタによって指定する．この図は,「左右にうごく」という名前の動作クラスの定義を示している．状態にあたる楕円には状態名と表示する絵を示し，遷移にあたる矢印には遷移条件となるイベント名をラベルとして示している．イベントが生じない場合のデフォルトの遷移にはラベルはない．各状態の属性はポップアップ・ウィンドウによって指定する．図のポップアップ・ウィンドウでは，状態名が「左へ」，表示する絵，アクションとして x 方向の相対的な移動を指定している．

グループ定義は，**図 2** に示すリストを作成する画面で行う．すでに定義した動作クラスの名前をプルダウン・メニューから選択し，次に，その右隣のプルダウン・メニューから，指定した動作クラスに含まれる状態を選択する．そして,「なかま」ボタンにより，これを同時に生成するキャラクタのグループのリストに加える．

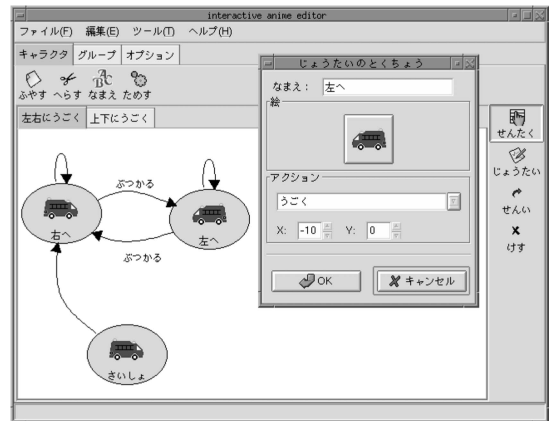


**図 1** 状態遷移図の編集画面
Fig. 1 Edit window for state-transition diagrams.

**図 2** グループ定義画面
Fig. 2 Group definition window.

## 4. 中間言語表現

### 4.1 中間言語の役割

GUI エディタで指定した対話型アニメは，インタプリタにより編集しながら動作確認を行うことができる．そして，完成したアニメは，最終的に，Java や JavaScript といった対象コードへと変換する．編集データを対象コードの形で保存しておくと，後から対象コードの種類を増やした場合に，対応できないなど得策ではない．そのため，図形で指定されたアニメを，テキストベースの中間言語表現で保存する．

また，中間言語表現は，単に，編集データを保存する目的のほかに，冗長性を減らす役割も担う．

GUI エディタで指定した状態遷移図は，状態に対応する出力として，キャラクタの絵を指定する．一方で，実際のアニメでは，絵だけが異なり，同じ動作をするキャラクタを定義したい場合が多い．たとえば，図 1 で指定した絵は消防車であるが，救急車も同じ動作で表示したい場合である．そのようなアニメの指定では，絵だけが異なる動作クラスを複数用意することになる．しかし，それをそのまま対象コードに変換したのでは，結果のコードが冗長となり，コードサイズが増加してしまう．そのため，中間言語表現では，表示する絵に対応する画像ファイルの情報を，状態遷移の情報とは分離した構成をとる．これにより，複数の

絵の組を，1 つの状態遷移に適応させる形のプログラムとなり，対象コードの冗長性を削減する．

中間言語表現としては，プログラムを手動で直接編集する可能性を残すために，iasl（interactive animation specification language）と呼ぶ，独自の記述言語を用いている．

### 4.2 中間言語の例

iasl のプログラムは，キャラクタの画像の指定（`obj` 定義），状態遷移図にあたる定義（`trans` 定義），画像と状態遷移図と初期状態の 3 つ組を複数指定したグループ定義（`group` 定義），全体のパラメータからなる．

**図 3** は，図 1 および図 2 で指定した対話型アニメを iasl プログラムとして表現したものである．図中のタイプライタ体で示された `obj` や `tarns` は予約語である．

初めの `obj` 定義では，2 つの画像ファイルを指定し，これに `ftruck` という名前を付けている．ここで指定した画像ファイルには，C 言語の enum 型に類似して，0 からの番号が振られる．その番号は，状態遷移の出力指定に使う．

次の `trans` 定義では，図 1 に対応して，`init`，`right`，`left` の 3 状態を指定している．各状態の出力は，画像の番号と `rand` や `move` などのキャラクタの移動指定で定義する．ここで `rand` は，乱数によるキャラクタの位置決めを示し，`move` は相対的な移動を示す．状態 `init` は無条件に状態 `right` へと遷移することを示す．一方，状態 `right` と状態 `left` は，衝突（`bump`）の遷移条件により他方の状態に遷移し，衝突がない場合には同じ状態を繰り返すことを示す．

最後の `group` 定義では，`obj` 定義と `trans` 定義の組合せ，および `trans` 内の初期状態を指定することで，同時に生成するキャラクタを指定している．GUI エディタと同様に，`main` という名の `group` 定義は，アニメを開始したときに生成されるキャラクタとなる．この例では，2 台の消防車が生成される．

このプログラムでは，全体のパラメータの指定は省略されているので，アニメの画面サイズや状態遷移の時間間隔はデフォルト値が使われる．

## 5. 対象コードへの変換手法

中間言語プログラムはトランスレータによって，対象コードへと変換する．現在対応している対象コードには，JavaScript，Java があり，それぞれ特徴のある変換手法を用いている．

firetruck.ia

```
obj ftruck{ fr.xpm, fl.xpm };

trans LR {
   init(0, rand) right;
   right(0, move(10,0)) {
      bump :left;
      default:right;
   }
   left(1, move(-10,0)) {
      bump :right;
      default:left;
   }
}

group main {
   fork(ftruck, LR, init);
   fork(ftruck, LR, init);
}
```

**図 3** 中間言語によるプログラムの例
Fig. 3 Sample program in intermediate language.

firetruck.c

```
switch (state) {
case INIT: ....
        state = RIGHT; break;
case RIGHT: ....
        if (isbumped()) state = LEFT;
        else state = RIGHT;
        break;
....
}
```

**図 4** C 言語の case 文による状態遷移図のプログラム
Fig. 4 A program for state-transition diagrams in C with “case” statement.

### 5.1 状態遷移図のプログラム表現

状態遷移図のプログラム表現は，Aho らの Compilers[6] に代表されるような，**図 4** の形の case 文による状態の選択と次の状態の決定といった方法がよく用いられる．この方法は，たいていの手続き型言語で実装できるという利点を有するが，状態数に応じて巨大な case 文となってしまうという問題もある．

そこで変換の際には，JavaScript の関数ポインタや Java の内部クラスを用いることで，個々の状態の記述を互いに独立させて，状態遷移図と対象コードの対応関係がつくように工夫している．

### 5.2 ムーア型機械と対話型アニメーションの関係

動作クラスの基本モデルであるムーア型機械は，入力 $X$，出力 $Y$，状態 $Q$ および $Q'$，状態遷移関数 $\delta$，出力関数 $\omega$ とした場合，$Q' = \delta(Q, X)$ かつ $Y = \omega(Q)$

と表される．通常，入力 $X$ は，状態 $Q$ や出力 $Y$ とは独立に与えられるので，プログラムにおける状態遷移関数 $\delta$ と出力関数 $\omega$ の処理は，同時か，または，出力関数 $\omega$ の処理を先とすればよい．

一方で，我々の扱う対話型アニメでは，入力として他のキャラクタとの衝突のイベントを含むので，入力 $X$ は，他のキャラクタの状態に依存している．また，衝突判定に用いるキャラクタの位置情報は，ある状態で表示しているキャラクタの画像の大きさに依存するので，出力処理の中でキャラクタの位置情報を決定したい．したがって，ムーア型機械のモデルに沿って，そのままプログラムを記述すると，次のような処理がアニメの画面更新ごとに必要となる．

(1)　すべてのキャラクタに対する出力の処理
(2)　衝突判定
(3)　すべてのキャラクタに対する状態遷移の処理

この方法では，出力と状態遷移の処理で，別々に，すべてのキャラクタを扱わなければならず，キャラクタの数が増えたときの処理のオーバヘッドが問題となる．そこで，画面更新に対する処理の位相をずらし，次の方法をとった．

(1)　衝突判定
(2)　すべてのキャラクタに対して
　(a)　状態遷移の処理
　(b)　出力の処理

この処理で表現している内容は，$Q' = \delta(Q, X)$ かつ $Y = \omega(Q')$ であり，厳密にはムーア型機械と異なる．しかし，状態遷移図として示される内容とアニメの表示とに矛盾がなく，処理のオーバヘッドも小さい．

### 5.3　JavaScript プログラムへの変換

**図 5** は，図 3 のプログラムを JavaScript に変換した結果の一部である．関数 `trans_LR_right` は，状態遷移 `LR` 中の状態 `right` 用の処理である．引数の `e` は，個々のキャラクタを表す構造体であり，引数の `out` は，状態に対応する出力処理を行うか，状態遷移を処理するかを決めるフラグである．

`out` が非 0 で，出力用の処理を行う場合には，`setImage()` 関数により，構造体 `e` の `iasl_img` メンバ配列から表示する画像を取り出し，設定する．そして，その画像の表示位置を `move_rel` 関数によって決める．`out` が 0 で，遷移の処理をする場合には，`if` 文により，遷移条件を確定し，関数ポインタの `iasl_func` メンバを書き換えることで，`e` が示すキャラクタの状態を遷移させる．

関数 `iaslTimeoutFunc()` は，どのアニメにも共通の関数であり，全体の動作を制御する．この関数は，

firetruck.js

```
function trans_LR_right(e,out){
  if (out) {
     setImage(e, e.iasl_img[0]);
     move_rel(e, 10, 0);
     return;
  }
  if (e.iasl_bumped) {
     e.iasl_func = trans_LR_left;
     return;
  }
  e.iasl_func = trans_LR_right;
  return;
}

function iaslTimeoutFunc(){
  ...
  checkBumped();
  for (i=0; i<iasl_inst.length; i++){
     e = iasl_inst[i];
     e.iasl_func(e, 0); // transition
     ...
     e.iasl_func(e, 1); // output
     ...
  }
}
```

**図 5**　JavaScript による状態遷移図のプログラム
Fig. 5　A program for state-transition diagrams in JavaScript.

一定時間ごとに呼び出され，そこでは，`iasl_inst` 配列に入ったキャラクタの構造体を 1 つ 1 つ取り出し，`iasl_func` に設定された関数を 2 回ずつ呼び出す．2 回の呼び出しのうち，1 回目で状態遷移の処理を行うので，`iasl_func` が変化した場合には，1 回目と 2 回目とで異なった関数を呼び出すこととなる．

なお，前述のように，このプログラムはムーア型機械のプログラム表現と厳密には異なる．しかし，その違いは `iasl_func` の指す関数の呼び出し方だけであり，呼び出される関数の表現方法は，厳密にムーア型機械を表現する場合でも，これと同じ方法でよい．

### 5.4　Java プログラムへの変換

Java の場合には，関数ポインタがないため，JavaScript と同じ変換手法を用いることができない．そこで，Java 特有のインタフェース型と内部クラスのオブジェクトへの参照を用いて，関数ポインタに代わる変換を行っている．

**図 6** に，これまでの例を Java に変換した場合のプログラムの一部を示す．クラス `iaslChar` は，すべてのキャラクタに共通する項目を定義している．このクラスは，状態遷移と出力を扱うために，`st` という `ssm` 型の変数を持っている．`ssm` 型は，`transit()` と

firetruck.java

```
class iaslChar extends ImageButton {
  ssm st; //sequential state machine
  ...
  public void doit() {
    st.transit();
    ...
    st.output();
  }
}

class trans_LR extends iaslChar {
  imgArray img;
  ...
  class stat_right implements ssm {
     public void output() {
       setImage(img, 0);
       move_rel(10, 0);
     }
     public void transit() {
       if (bumped != 0) {
          trans_LR.this.st
                = new stat_left();
          return;
       }
       trans_LR.this.st
                = new stat_right();
       return;
     }
  }
  ...
}

public class iasl extends Applet ... {
  ...
  public void run() {
    ...
    checkBumped();
    for (int i=0; i<curr.size(); i++) {
      iaslChar b;
      b = (iaslChar)curr.elementAt(i);
      b.doit();
      ...
    }
  }
  ...
}
```

**図 6** Java による状態遷移図のプログラム
Fig. 6 A program for state-transition diagrams in Java.

output() の 2 つのメソッドを持ったインタフェース型であり，クラス iaslChar を継承する各キャラクタのクラス定義の中で，内部クラスとしてこのクラスを実装する．

iaslChar を継承するクラス trans_LR は，中間言語表現の trans 定義 LR から生成したキャラクタのクラスである．状態を定義する内部クラスとして，stat_right がある．この内部クラスの output() メソッドでは，表示するキャラクタの絵とその場所を決定する．transit() メソッドでは，スーパクラスで宣言された変数 st を変更することで，状態の遷移を表現している．

これらのクラス定義により，各キャラクタの状態遷移と出力は，iaslChar クラス内の doit() メソッドを呼び出すだけとなり，結果として transit() と output() の 2 つのメソッドが呼ばれる．また，この方法を用いると，中間言語表現における状態遷移図の定義となる部分を，ほぼそのまま iaslChar クラスのサブクラスとして表現すればよく，他の実装の部分をスーパクラスにまとめられる．そのため，状態遷移図と対象コードの対応関係が分かりやすくなる．さらに，この変換結果も，JavaScript と同様に，厳密なムーア型機械のプログラム表現として用いることができる．

## 6. スクリーンショット

**図 7** および**図 8** に，作成した対話型アニメのスクリーンショットを示す．図 7 のパンダは，これまで説明してきた左右に動く動作に，正面を向いて静止する状態を加えて定義している．さらに，マウス操作に反応して，他のパンダを作り出したり，手を振りながら上下に動くカエルに変身したりする．上下に動くカエルは，マウス操作により元の左右に動くパンダに戻る．このパンダとカエルの関係は，階層的な状態遷移図の指定によって表現している．

図 8 の例は，対話型アニメというよりビデオゲームとなっている．UFO，宇宙人，敵と味方のミサイル，砲台，得点，終了ボタンなどをそれぞれ独立に定義し，衝突によってそれぞれの動きを変化させる．得点を増やすときには，透明なキャラクタを得点の下位桁の場所に生成し，衝突のイベントによってカウントアップを行う．

## 7. 関連研究

Logo 言語[7] は，子どもを対象とした教育用プログラミング言語である．タートルグラフィックスと呼ばれるグラフィックス機能が特徴であり，亀の形をしたカーソルに対して，方向と移動量のコマンドを与えることで絵を描く．Logo は単一図形を描くための手続き型言語であるが，MIT Media Lab. の StarLogo[8] は，この亀の動き自体をアニメとしてとらえ，Logo の考えを拡張して，数千もの亀を同時に制御できるようにしている．我々の対話型アニメの作成とは異なり，手続きの指定が基本となっている．

**図 7** パンダとカエル
Fig. 7 Pandas and frogs.

**図 8** インベーダゲームもどき
Fig. 8 Yet another invader game.

Karel++[9] と Alice[10] は，ともに，初心者プログラマ向けのプログラミング言語である．Karel++は 2 次元世界でロボットを制御し，Alice は 3 次元世界でオブジェクトを制御する．どちらもタートルグラフィックスに類似して，方向づけと移動のコマンドによってオブジェクトを動作させる．コマンド列は手続きやタスクといった単位でまとめることができる．Alice では，“プログラムの状態” を意識するが，我々のように，それを状態遷移図でとらえようとはしていない．

KIDSIM[11] は，テキストでプログラムを書くことなく，GUI のみでアニメ表示のシミュレーション・プログラムを構築するシステムである．これは，“図形書き換えルール” と “デモンストレーション” の 2 つのアイデアから成り立っている．前者は，画面の一部を更新するために用いるもので，ルールは画像とそれらの特性から構成される．表示画面の一部が特性で示されたある状況にマッチした場合に，ルールを適用して画面を更新する．後者は，マウス操作などの，ユーザによる操作の過程を記録することで，それを画面更新に用いるものである．書き換えルール，if-then ルールまたはプロダクション・システムといった AI の分野の技術で成り立っている．同様の技術を用いたものに，Viscuit[12] や AgentSheets[13] がある．

Copycat[14] は，3 次元迷路の中で，敵と撃ち合いながら，目的物を探すゲームを構築するためのプログラミング環境である．Copycat もデモンストレーションによって，迷路を進むエージェントを訓練していく．その特徴はユーザとの対話的な訓練にある．訓練モードでゲームを行うと，エージェントが未知の条件に遭遇した場合に，ゲームが一時中断し，ユーザによる訓練が求められる．ユーザは，デモンストレーションによりエージェントを教育する．Copycat はエージェントの内部制御モデルに，ムーア型機械を用いているが，それを示す文献 14) に詳細は示されていない．

## 8. 図形書き換えルールとの比較

上述の関連研究を大別すると，タートルグラフィックスに基づくアニメ記述と，図形書き換えルールに基づくアニメ記述とに分かれる．GUI によるプログラミングという面で考えると，図形書き換えルールによる方法が，我々の状態遷移図による方法と近い関係にある．そこで，図形書き換えルールと我々の方法との比較について述べる．

KIDSIM，Viscuit，AgentSheets などで用いられている図形書き換えルールは，書き換えの条件を図形だけで示せる場合に，直感的かつ簡潔なものとなる．たとえば，車と信号のアニメを考えると，車の前に青信号がある場合には車を進め，赤信号がある場合には車を止めるといったルールが，車と青信号，車と赤信号という図形だけで指定できる．Viscuit のサンプルではこの例を効果的に説明している．我々のアニメ記述では，離れたキャラクタの状態を直接調べる方法がないために，信号の変化によって何らかのイベントを発

生させ，それを車に伝えなければならず，やや複雑となる．

一方で，図形が同じでも状況によって異なる動作をさせたいときには，我々のアニメ記述の方が簡潔となる．たとえば，数回点滅してから色が変化する歩行者信号やトランポリンで上下運動する人を記述したいときである．どちらの場合も，同じ図形を異なる状況で使いたい．しかし，それをすると，図形書き換えルールでは，ある瞬間の図形を見ただけで次の図形を決めることができなくなり，問題が生じる．信号では点滅を繰り返すか，色を変化させるかは，点滅の回数によって決まる．そして，トランポリンでは，人が上に移動するか，下に移動するかは，それ以前の状況に依存している．そのため，図形書き換えルールでは，それらの状況によって別々の図形を用いるか，または，図形以外の特性を付加的に指定しなければならず，類似のルールが並ぶことにより記述の簡潔さが失われてしまう．これに対して状態遷移図では，そのような “状況” を簡潔に記述することができる．

他の比較すべき点としては，我々の状態遷移図によるアニメ記述では，図形書き換えルールとまったく対照的に，キャラクタの動作の定義から図形を分離できる点があげられる．これにより，中間言語の役割で説明した例にあるように，絵だけが異なり同じ動作をするキャラクタの指定が容易になる．さらに，一連の動作に名前が付けられるために，ライブラリ化して再利用することも可能となる．左右に動く，上下に動く，壁に沿って動くといった基本動作が当初からライブラリ化されていれば，絵だけを指定するだけでオリジナルアニメが作成でき，ツールを使い始める利用者の興味をうまく引き付ける．

## 9. おわりに

小学生の利用も視野にいれた，対話型アニメーション作成ツールを提案し，その設計と実装について示した．また，実装においてムーア型機械に相当するプログラムの構成方式を示した．一般によく用いられている case 文による状態遷移の表現ではなく，Java の内部クラスや JavaScript の関数ポインタを用いることで，状態遷移図と対象コードの対応関係を分かりやすくしている．

今後，実際に小学生に使ってもらい，楽しみながら対話型アニメを作ることができるかを評価する予定である．また，このツールを使ってオートマトンの授業用の教材を作成することも検討している．

今回示したツールは，GTK+で書かれたものであるが，Java のみによる i$\alpha$PPLI 用のプロトタイプシステム[15),16)] も別途実装を行っている．また，実現しているトランスレータの対象コードは，Java および JavaScript であるが，将来的には，ゲームボーイアドバンス，マイクロソフト Windows や X-Window 用のコードに対応する予定である．

## 参 考 文 献


1) Macromedia: Flash MX 2004. http://www.macromedia.com/software/flash/
2) Adobe: LiveMotion. http://www.adobe.com/products/livemotion/
3) 植木友浩：標準 Web デザイン講座 ActionScript for FLASH MX 2004, 翔泳社 (2004).
4) Mayer, M.: *JUST GRANDMA AND ME*, Broderbund Software, Inc. (1991).
5) コーシンストアー：キューティマスコット++. http://ashir.net/plaza/kohshinstore/SOFTWARE/CMTPL/CMPLUS.html
6) Aho, A.V., Sethi, R. and Ullman, J.D.: *Compilers, Principles, Techniques and Tools*, Addison-Wesley (1986).
7) Papert, S.: *Mindstorms: Children, Computers and Powerful Ideas*, 2nd Edition, Perseus Books (1993).
8) Media Laboratory and Teacher Education Program, MIT: StarLogo. http://education.mit.edu/starlogo/
9) Bergin, J., Stehlik, M., Roberts, J. and Pattis, R.: *Karel++ A Gentle Introduction to the Art of Object-Oriented Programming*, John Wiley & Sons (1997).
10) Dann, W., Cooper, S. and Pausch, R.: Making the connection: programming with animated small world, *ACM SIGCSE Bulletin, Proc. 5th annual SIGCSE/SIGCUE ITiCSE conference on Innovation and technology in computer science education*, 3, Vol.32, pp.41–44 (2000).
11) Smith, D.G.: KIDSIM: Programming Agents Without a Programming Language, *COMM. ACM*, Vol.37, No.7, pp.55–67 (1994).
12) 原田康徳：Viscuit. http://www.viscuit.com/
13) AgentSheets, Inc: AgentSheets. http://agentsheets.com/
14) Banyasad, O.: Copycat: A cooperative gaming environment for programming adversarial agents by children, *Abstruct paper for Special Event: Children's Programming Odyssey held at 2001 IEEE Symposia on Human-Centric Computing Languages and Environments* (*HCC'01*) (2001).
15) 鎌田 賢，岡本秀輔，中尾隆司：状態遷移図にもとづくインタラクティブ・アニメーションのオー

サリング・ツール，茨城大学大学院 SVBL/茨城大学工学部 CCRD 平成 15 年度第 2 回成果報告会，pp.126–129 (2004).

16) Morioka, T.: A tool for authoring interactive animation based on state transition diagram, Master's thesis, Graduate School of Science and Engineering, Ibaraki Univeristy (Feb. 2002).

(平成 16 年 3 月 30 日受付)
(平成 16 年 8 月 3 日採録)

**岡本　秀輔**（正会員）

昭和 40 年生．平成 6 年成蹊大学大学院博士後期課程修了．平成 6 年電気通信大学大学院助手．平成 9 年同大学院講師．平成 11 年茨城大学工学部，同大学大学院講師．並列処理プログラミング，プロセッサアーキテクチャの研究に従事．日本ソフトウェア科学会，IEEE Computer Society，ACM 各会員．

**鎌田　　賢**

昭和 37 年生．昭和 63 年筑波大学大学院博士課程修了．昭和 63 年筑波大学電子・情報工学系助手．平成元年同大学講師．平成 4 年茨城大学工学部，同大学大学院助教授．線形システム理論，信号処理，ウェブアプリの研究に従事．電子情報通信学会，IEEE，EURASIP 各会員．

**中尾　隆司**

昭和 31 年生．昭和 57 年広島大学理学部卒業．昭和 57 年日立エンジニアリング（株）入社．平成 3 年米国アラバマ大学大学院修士課程修了．平成 9 年日立エンジニアリング（株）主任技師．信号処理，画像処理，携帯端末応用の開発に従事．電子情報通信学会会員．