オブジェクト指向方法論（OMT）に基づく動的モデルからのJavaコード生成

アリ ジョハル† 田中二郎†

システムの動的モデルから実行可能コードを生成することは，非常に重要な研究課題である．多くのCASEツールは，オブジェクトモデルすなわちシステムの静的構造から，ただヘッダファイルを生成するだけである．オブジェクト指向方法論（Object Modeling Technique：OMT）において，動的な振舞いは状態遷移図からなる動的モデルによって表される．本論文では，動的モデルから実行可能なオブジェクト指向コードに変換を行うための新しい手法について述べる．我々の手法では，状態遷移図における状態をオブジェクトとして扱う．そのオブジェクトは，状態のすべての遷移と動作をカプセル化したものである．遷移は状態オブジェクトの操作となる．ORタイプの状態階層は継承によって，ANDタイプの状態階層はオブジェクトコンポジションの概念を用いて実装する．我々は提案する手法を実際に実装し，動的モデルの仕様から実行可能なJavaコードを生成するシステムdCodeを開発した．dCodeは，まず動的モデルを表現した状態図から表を作成し，その表からJavaコードを生成する．dCodeをRhapsodyと比較した場合，dCodeの方が実行速度においては約40%も効果的であり，生成されるコードの量は5倍もコンパクトである．

Generating Java Code from the Dynamic Model Based on Object Modeling Technique

JAUHAR ALI† and JIRO TANAKA†

Implementing the dynamic behavior of a system is a challenge for object oriented software developers. Most of the CASE tools can only generate header files from the static structure of a system, usually refered as object model. In Object Modeling Technique (OMT), the dynamic behavior is represented by dynamic model, which consists of a set of state transition diagrams. This paper describes a new method to convert the dynamic model into executable object oriented code. Object oriented approach has been used to make the implementation code simple and easier to extend in case the dynamic model changes. In our approach, each state of the state diagram is given the status of an object, which encapsulates all the transitions and actions of the state. Transitions become operations of the state object. OR-type state hierarchy is implemented by *inheritance* and AND-type state hierarchy is implemented by using the concept of *object composition*. A system, dCode, has been developed that implements the proposed method and generates executable Java code from the specifications of the dynamic model. First the system transforms the state diagram representing the dynamic model into a table and then it generates actual code from the table. A comparison between Rhapsody and dCode has been performed, and it has been found that the code generated by dCode is approximately 40% more efficient and five times more compact than that of Rhapsody.

1. はじめに

オブジェクト指向に基づくソフトウェア開発方法論であるOMTにおいて，システムはオブジェクトモデル，動的モデル，機能モデルの3つの観点からとらえられる．オブジェクトモデルはシステムにおけるクラスとそれらの関係を表し，動的モデルは制御フローとオブジェクト間の相互作用を，そして機能モデルはシステムのデータの変化を記述する[32]．これらのOMTモデルは分析フェーズで導入されて，設計フェーズの中で次第に洗練されていく．実装フェーズにおいて，OMTモデルは実行可能コードに変換される．この設計・実装フェーズにおいて，最も重要なモデルは動的モデルである．この理由は動的モデルなしではオブジェクトモデル中のクラスの振舞いが分からなくなってしまうからである．

オブジェクト指向方法論は，分析・設計フェーズの

† 筑波大学電子・情報工学系
Institute of Information Sciences and Electronics, University of Tsukuba

ステップを詳細に述べているが，それらの多くはシステムの分析・設計モデルをどのように実装コードに変換できるのかを示していない．多くのプログラマにとって，分析・設計フェーズで作成したモデルを実装コードに変換することは困難である．動的モデルの場合においては変換はさらに困難である．そのため，オブジェクト指向モデルのうち，特に動的モデルにおいては，モデルを実装コードに変換するための方法を確立することが必要であり，さらにその方法を用いて自動的にコードを生成してくれるCASEツールが求められている．

現在のCASEツール[6),21),25),27)]はオブジェクトモデルから宣言的なヘッダファイルを生成するだけにとどまっている．オブジェクトモデルは静的構造を持つので，それからヘッダファイルを生成することは比較的容易である．状態遷移図からなる動的モデルは，その動的な特性のためコードを生成することは難しい．いくつかのCASEツール[3),18),22),26)]は状態遷移図から実行可能コードを生成する．しかしながら，Rhapsody[14),18)]を除くそれらのほとんどは，状態図の状態階層や並列性をサポートしていない．Rhapsodyはオブジェクトモデルと動的モデルからC++コードを生成するツールである．

本研究の目的は，動的モデルをオブジェクト指向で書かれた実行可能コードに自動的に変換する方法を提案することである．生成された実装コードは構造化されており，可読性が高くなければならない．状態図のすべてのコンポーネントが実装コードの中に明確に表されるべきである．そうしておけば，新しい状態や遷移が状態図に加わった場合でも，コードの変更が容易である．

本論文では，我々のアプローチを説明するために，2章において，カセットプレイヤシステムの例を与える．3章では状態図から実装コードに変換する我々の手法について説明する．例をあげて状態図の状態階層や並列性の扱いを述べる．4章ではそれ以外に考えられる方法について述べ，議論する．5章では実装コードを自動的に生成するシステムdCodeについて説明する．dCodeが含むモジュールと状態図からコードを生成するための細かいルールについて説明する．6章では生成された実装コードがどのように実行されるかということを示す．7章ではdCodeをRhapsodyと比較する．自動的に生成された実装コードがRhapsodyのコードよりコンパクト，また，効率的であることを述べる．8章および9章では各々関連研究と結論について述べる．

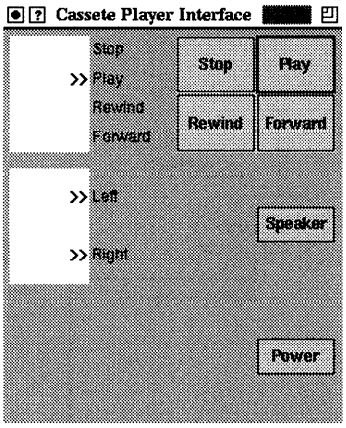


図1 カセットプレイヤのリモコン
Fig. 1 Remote control device for cassette player.

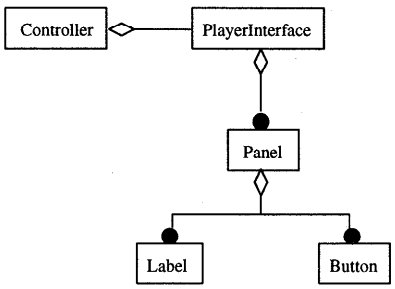


図2 カセットプレイヤシステムのオブジェクトモデル
Fig. 2 Object model of the remote control device.

2. カセットプレイヤシステム

例として，リモコンで操作するカセットプレイヤを取りあげる（図1）．リモコンには右上に名前の付いたボタンStop，Play，Rewind，Forwardがあり，またその下にボタンSpeaker，Powerがある．左側の2つの矩形はカセットの現在の状態を示している．カセットがOffのときは，表示部分（2つの矩形）には何も表示されない．

このシステムのためのソフトウェアを開発することを考える．図2はこのアプリケーションのためのオブジェクトモデルを示している．PlayerInterfaceクラスはPanelクラスのインスタンスを保持しており，Controllerクラスと関連を持つ．Controllerクラスはシステム全体をコントロールする[1),2),29)~31)]．Panelのインスタンスには，ButtonやLabelのインスタンスが含まれる．いったんボタンが押されると，PlayerInterfaceはこれを知らせるために特殊なメッセージをControllerに送る．Controllerからの応答は，システムの状態に依存する．

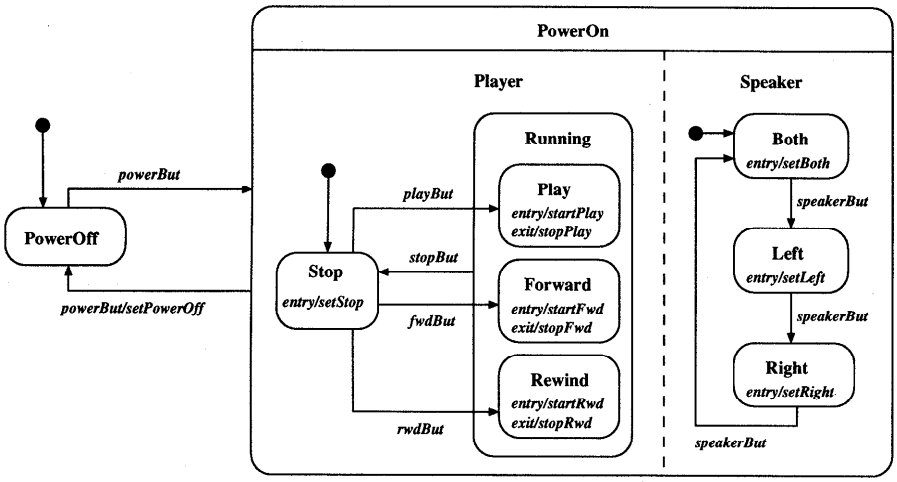


図 3 Controller の状態図
Fig. 3 State diagram of the Controller.

図 3 の状態図は，Controller の振舞いを示している．ノードは状態を，有向アークは遷移を表す．遷移はいつもイベントをともない，場合によって遷移と関連する動作もともなう．イベント名は遷移線上に記述される．遷移上の動作に対する記法は，それを起こすイベント名の後に斜線 "/" と動作の名前を続けて記す．たとえば，図 3 において powerBut，playBut などのイベントがあり，setPowerOff という動作がある．遷移上の動作の他に状態中に入場動作と退場動作がある．入場動作は，状態ボックス中に "entry /" と記してその後に記述される．なんらかの遷移によってある状態に入った際に，まず入場動作が実行される．退場動作は，状態ボックス中に "exit/" と記してその後に記述され，なんらかの遷移によって状態から出る際に実行される．たとえば，Play 状態に対して startPlay という入場動作と stopPlay という退場動作がある．

ここには取りうる状態が 2 つ（PowerOff, PowerOn）ある．これらの状態は powerBut イベントが発生したとき，どちらか一方がアクティブになる．PowerOn 状態は 2 つの並列状態（あるいは AND 状態）[11),12),15)] の組合せ（すなわち，Player と Speaker）である．これらの両方は PowerOn 状態がアクティブになったとき同時にアクティブとなる．各々の並列状態は，数多くの OR 下位状態 [11),12),15)] を持つ．たとえば，Player は Stop，Running という 2 つの下位状態を持ち，また Running は Play，Forward，Rewind という 3 つの下位状態を持つ．OR 状態の中で，アクティブとなりうるものは 1 つだけである．黒丸から遷移がある状態はその状態がデフォルトであるということを示している．

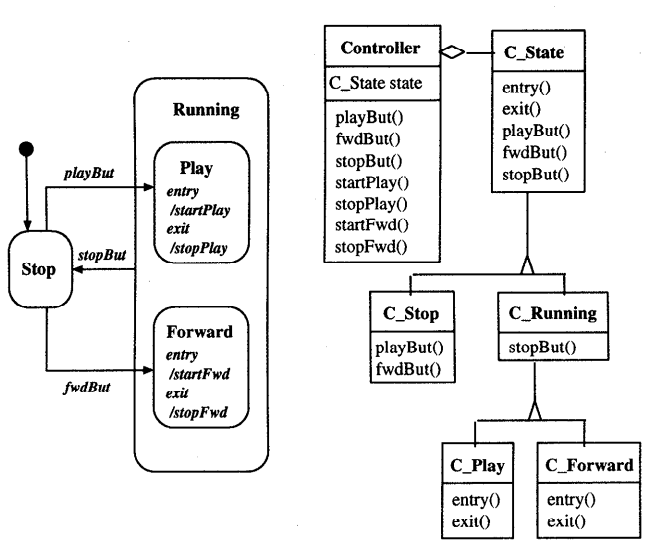
3. 状態図の実装

状態図（図 3）に示すように Contoller は，他のオブジェクトから要求があるとき，現在の状態によって異なった応答をする．たとえば，playBut という要求に対しては，Stop 状態か他の状態であるかによって異なった応答をする．

状態図を実装するために我々は状態クラスのアプローチを用いた．すなわち，各々の状態に対して，状態の振舞いを実装するクラス（状態クラス）を作る．これらのクラスを状態クラスと呼ぶ．我々のアプローチにおけるキーアイデアは，Controller の状態を表すために，C_state という抽象化クラスを導入したことである．C_state はすべての状態クラスに共通なインタフェースを宣言している．その目的は，状態クラスが状態図におけるすべてのイベントを受理できるように作るためである．我々は "Controller" という単語を表すために "C_" という接頭辞を用いる．

3.1 OR タイプの状態階層

状態図において，上位状態は下位状態を持ち，下位状態に共通な遷移を持つ場合がしばしば現れる．たとえば，図 3 において上位状態 Running は下位状態 Play，Forward，および，Rewind を持ち，stopBut イベントという共通な遷移を保持する．我々は状態図においての状態階層はオブジェクト指向プログラムにおけるクラス階層と類似性を持っていることを発見した．前者において，下位状態がその上位状態の振舞いを継承するように，後者においても，サブクラスがそのスーパクラスの振舞いを継承する．我々のアプローチでは，複数の状態を持つクラスの各状態はクラスと




```
class Controller{
  static C_State state;
  void playBut(){state.playBut();}
  ......
  static void startPlay(){...}
  ......}
class C_State{
  void entry(){}
  void exit(){}
  void playBut(){}
  ......}
class C_Stop extends C_State{
  void playBut(){exit();
    Controller.state=new C_Play();
    Controller.state.entry();}
  void fwdBut(){...}}
class C_Running extends C_State{
  void stopBut(){exit();
    Controller.state =new C_Stop();
    Controller.state.entry();}}
class C_Play extends C_Running{}
  void entry(){Controller.startPlay();}
  void exit(){Controller.stopPlay();}}
class C_Forward extends C_Running{
  void entry(){Controller.startFwd();}
  void exit(){Controller.stopFwd();}}
```

(a) 並列状態のある状態図　(b) クラス構造　(c) Javaコード

図 4 状態階層のある状態図の実装
Fig. 4 Implementing a state diagram having state hierarchy.

して，また，下位状態は上位状態に対応するクラスのサブクラスとして実装する．上位状態の振舞いはスーパクラスのメソッド，下位状態の振舞いはサブクラスのメソッドとして実装する．このように継承のメカニズムを使うことによって，状態階層を実装することが可能になる．

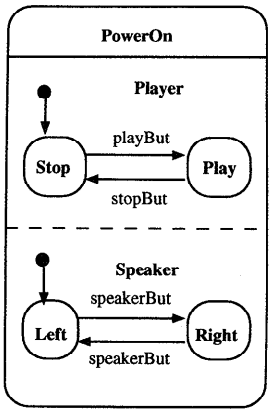
例として，図4に「図3を簡略化した状態図」,「結果として得られるオブジェクトモデル」と「Java コード」を示す．図4のオブジェクトモデルにおいて，Controller は C_state タイプの状態オブジェクトを持っている．状態オブジェクトは C_state のサブクラスのインスタンスを参照しており，システムの状態を表している．Controller は状態に関係する要求を state オブジェクトに処理させる．

図4の実装コードが示すように，システムの状態を変化させたときは，state の値も変化する．たとえば，システムの状態が Stop から Play になったときに，state の値も C_Stop クラスのインスタンスから C_Play クラスのインスタンスに置き換わる．状態オブジェクトはシステムの状態の変化に従ってそのクラスを変化させる．

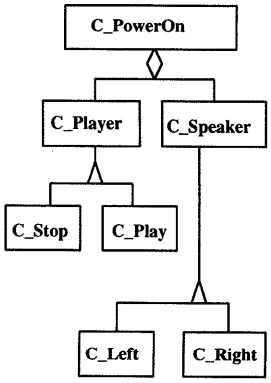
Running の下位状態は Play と Forward である．Running がアクティブになるとき，Play と Forward のどちらか一方が同様にアクティブとなる．Play と Forward の両方ともに，それがアクティブのときは stopBut というイベントに応答する．なぜならば，上位状態の Running からこのイベントに遷移があるからである．C_Play と C_Forward は C_Running クラスのサブクラスとなる．C_Running クラスにはそのサブクラスによって継承される stopBut() という操作の実装が含まれる．下位状態からの遷移がないため，サブクラスはただ entry と exit 操作の実装を含むだけである．

3.2 AND タイプの状態階層

状態図は並列状態（AND 状態）を含む場合もある．たとえば，図3において上位状態 PowerOn の下位状態 Player と Speaker が並列状態である．上位状態がアクティブになると，同時に下位状態もアクティブとなる．並列状態を用いると，複雑な状態図もコンパクトに記述できる[13]．我々のアプローチにおいて，アクティブな状態はオブジェクトのインスタンスによって表現される．並列状態を採用しているため，それらの上位状態がアクティブになるとき，並列下位状態の数と同じだけのオブジェクトを作り出すことを保証するメカニズムが必要となる．すなわち，AND 状態に対応するオブジェクトを所有する複合クラスとして AND 状態の上位状態は表現される．AND 状態の上位状態がアクティブになるとき，それに対応する複合クラスのインスタンスが作られる．複合クラスのインスタンス化はこのように AND 状態に相当するクラスのインスタンス化を保証する．複合クラスはたくさんの状態が同時にアクティブになっているかのように振る舞う．



(a) 並列状態のある状態図



(b) クラス構造

```
class C_PowerOn{
  static C_Player player= new C_Stop();
  static C_Speaker speaker = new C_Left();
  playBut(){player.playBut();}
  stopBut(){player.stopBut();}
  speakerBut(){speaker.speakerBut();}}
class C_Player {
  playBut(){}
  stopBut(){}}
class C_Stop extends C_Player {
  playBut(){C_PowerOn.player=new C_Play();}}
class C_Play extends C_Player {
  stopBut(){C_PowerOn.player=new C_Stop();}}
class C_Speaker {speakerBut(){}}
class C_Left extends C_Speaker {
  speakerBut(){C_PowerOn.speaker=new C_Right();}}
class C_Right extends C_Speaker {
  speakerBut(){C_PowerOn.speaker=new C_Left();}}
```

(c) Javaコード

図 5 並列状態のある状態図の実装
Fig. 5 State diagram with concurrent states and its implementation.

同様に，複合オブジェクトが削除されるとき，複合オブジェクトが所有するすべてのオブジェクトも削除される．AND 状態は抽象化状態であり，下位状態を含む．実装コードでは，AND 状態は抽象化クラスになり，下位状態のクラスに共通なインタフェースを提供する．

例として，**図 5** は「カセットプレイヤにおける Controller の状態図の一部」，「結果として得られるオブジェクトモデル」と「その実装コード」を示したものである．C_PowerOn は複合クラスであり，C_Player と C_Speaker クラスのタイプのオブジェクトを保持する．C_Player と C_Speaker は 2 つの並列状態を表している抽象化クラスであり，それらのサブクラスのためのインタフェースクラスとして使われる．実際にインスタンスの生成が可能なクラス C_Stop，C_Play と C_Left，C_Right は各々C_Player と C_Speaker のサブクラスとなる．図 5 の実装コードが示すように，複合クラスは AND 状態の中にある要求（イベント）をそれに対応する状態オブジェクトに任せる．たとえば，playBut は player 状態オブジェクトに，speakerBut は speaker 状態オブジェクトに任される．AND 状態の上位状態（PowerOn）から起こる遷移があれば，複合クラス（C_PowerOn）は実装コードを提供し，下位状態のオブジェクトには要求（イベント）を任せない．

4. 議　　論

我々の方法においては，状態と関連するすべての振舞いは 1 つのオブジェクトとして扱われる．状態のすべての振舞いを表すコードは単一の C_State サブクラスに含まれるため，新しいサブクラスや操作を定義することによって新しい状態や遷移を簡単に追加できる．我々の方法とは異なった他の方法としては内部状態を表す属性を用い，Controller クラスの操作で内部状態の値に応じた振舞いを行うことが考えられる．この方法の場合，状態遷移は変数に代入を行うことによって実装される．**図 6** に Controller の内部状態を属性の値として表す場合の speakerBut メソッドの実装コードを示す．これは効率的な方法であるかもしれないが，設計フェーズにおいて様々な状態や遷移として表れるシステムの実際の振舞いが，コードの中に隠されてしまう．コードを見ただけで，システムの振舞いを理解することは難しい．状態をオブジェクトとして表現するのは，遷移をもっと明確にし，コードを理解しやすくする．コードの中に状態図の特色を持たせ，またコードを動的モデルに戻すことにも役立つ．

異なった状態の振舞いが C_State のサブクラスに割り当てられているため，我々のアプローチは多くのクラスを作っているように見えるかもしれない．これは単一のクラスに振舞いを実装するよりコンパクトではない．しかしながら，このような割当ては条件文を除くことができる．大きな条件文はコードを理解しにくくし，部分修正や拡張が難しくなるため，望ましくない．状態における各々の状態遷移や動作を 1 つのクラスにカプセル化することは，状態図のよりオブジェクト指向的な実装である．さらに，コードの自動生成のために，状態図の要素（状態と遷移）とプログラム要素（クラスやメソッド）を 1 対 1 に対応させることは自然である．このような理由から，Rhapsody[14),18)] などの有名なコード生成ツールは状態をクラスとして表現する．

```
class Controller {
  int mainState = 1; // 1=PowerOff, 2=PowerOn
  int playerState = 1; // 1=Stop, 2=Running
  int runningState; // 1=Play, 2=Forward, 3=Rewind
  int speakerState = 1; // 1=Both, 2=Left, 3=Right

  public void speakerBut(){
    switch (mainState){
      case 1:{
        ... ;break;}
      case 2:{
        switch (speakerState){
          case 1:{
            speakerState = 2; setLeft();break;}
          case 2:{
            speakerState = 3; setRight();break;}
          case 3:{
            speakerState = 1; setBoth();break;}
        }
    }
  }
  ...
}
```

図 6 状態を属性の値で表した状態図の実装
Fig. 6 Implementing a state diagram using data values to represent states.

我々のアプローチでは，C_State のサブクラスは，遷移が起こったときにインスタンス化される．結果として得られたコードは効率的でないように見える．この問題の解決策は，それよりも先に C_State の全サブクラスのオブジェクトを作り，それから，遷移を実行している間，新しいものをインスタンス化する代わりに状態オブジェクトに適切なインスタンスを代入する．この方法は Controller クラスの中に全状態オブジェクトのリストを持つことによって簡単に実装することができる．

5. 自動コード生成

前章で述べた我々のアプローチを用いて，動的モデルから自動的に実装コードを生成するシステム dCode を開発した．まず，dCode は状態図を表に置き換え，次に表から実際のコードを生成する．dCode への入力は Design Schema List Language (DSL)[9),10)] で書かれた動的モデルの仕様である．dCode からの出力は実行可能な Java[8)] コードである．DSL は仕様言語であり，理解しやすいテキスト形式で OMT のモデルを簡単に表現する．中島ら[23),24)] は OMT のモデルをインタラクティブに描くことができ，それから自動的に DSL に変換するツールを開発した．カセットプレイヤの状態図（図 3）を DSL で表現したものを**図 7** で示す．DSL において，各々のステートメントはセミコロンによって区切られている．初めの 2 つの行は状態図を構成するノードとアークを宣言している．ノードは状態，アークは遷移を意味する．各々のノードは，OSTDN という文字列で始まるステートメントで表現され，同じように，各々のアークは，OSTDA で始まるステートメントで表現される．

```
OSTD(g1)[nodes{n1,n2,n3,n4,n5,n6,n7,n8,n9,n10,n11,n12,n13,n14,
  n15},arcs{a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11,a11}];

OSTDN(n1)[loc(35:35),size(3:3),ostdnAttr(name:START)];
OSTDN(n2)[loc(30:50),size(20:10),ostdnAttr(name:PowerOff)];
OSTDN(n3)[loc(63:18),size(90:70),ostdnAttr(name:PowerOn,
  concurrent{n4,n11})];
OSTDN(n4)[loc(63:25),size(60:70),ostdnAttr(name:Player,
  substates{n5,n6,n7})];
OSTDN(n5)[loc(44:70),size(3:3),ostdnAttr(name:START)];
OSTDN(n6)[loc(65:58),size(12:14),ostdnAttr(name:Stop,
  entry/setStop)];
OSTDN(n7)[loc(90:35),size(25:55),ostdnAttr(name:Running,
  substates{n8,n9,n10})];
OSTDN(n8)[loc(95:43),size(20:13),ostdnAttr(name:Play,
  entry/startPlay,exit/stopPlay)];
  .....
OSTDA(a1)[from(n1,side:BOTTOM,off:1),to(n2,side:TOP,off:5)];
OSTDA(a2)[from(n2,side:TOP,off:10),to(n3,side:LEFT,off:25),
  ostdaAttr(name:powerBut)];
OSTDA(a3)[from(n3,side:LEFT,off:45),to(n2,side:BOTTOM,off:10),
  ostdaAttr(name:powerBut/setPowerOff)];
OSTDA(a4)[from(n5,side:BOTTOM,off:1),to(n6,side:TOP,off:3)];
OSTDA(a5)[from(n6,side:TOP,off:6),to(n8,side:LEFT,off:6),
  ostdaAttr(name:playBut)];
  .....
```

図 7 DSL 形式で書かれた Controller の状態図
Fig. 7 State diagram of the Controller in DSL format.

dCode は Java 言語で開発されており，基本的に Transformer と Code Generator という 2 つのモジュールからなる．以下ではこれら 2 つのモジュールの機能を簡単に説明する．

5.1 Transformer

このモジュールは DSL 形式で与えられた状態図の仕様を読み込み，様々な状態，それらの下位状態や遷移を識別する．それから，すべての情報を記録した表を作成する．状態図において，デフォルト状態は，黒丸からのアークによって指定された状態である．この黒丸は，単に近くのノードがデフォルト状態であることを示しているだけである．しかしグラフィカル指向の DSL はデフォルト状態と他のノードを同じように扱っている（START という名前を持つ）．Transformer では，このようなノードをすべて無視し，その代わりにその先のノードをデフォルト状態として登録する．**図 8** は Controller の状態図（図 3）から Transformer モジュールによって作られた表を示している．

5.2 Code Generator

このモジュールは Transformer によって作られた表から情報を取り出し，Java コードを生成する．**図 9** において実線部分は生成されたコードのクラス構造を示している．**図 10** はカセットプレイヤの動的モデル

State ID	State Name *=default	Super state	Substates *=concurrent	Entry	Exit	Transitions		
						Event	Action	Next State
n2	PowerOff*					powerBut		n3
n3	PowerOn		n4,n11 (*)			powerBut	setPowerOff	n2
n4	Player	n3	n6,n7					
n6	Stop*	n4		setStop		playBut		n8
						fwdBut		n9
						rwdBut		n10
n7	Running	n4	n8,n9,n10			stopBut		n6
n8	Play	n7		startPlay	stopPlay			
n9	Forward	n7		startFwd	stopFwd			
n10	Rewind	n7		startRwd	stopRwd			
n11	Speaker	n3	n13,n14,n15					
n13	Both*	n11		setBoth		speakerBut		n14
n14	Left	n11		setLeft		speakerBut		n15
n15	Right	n11		setRight		speakerBut		n13

図 8 Transformer モジュールによって生成された表
Fig. 8 Table created by the Transformer.

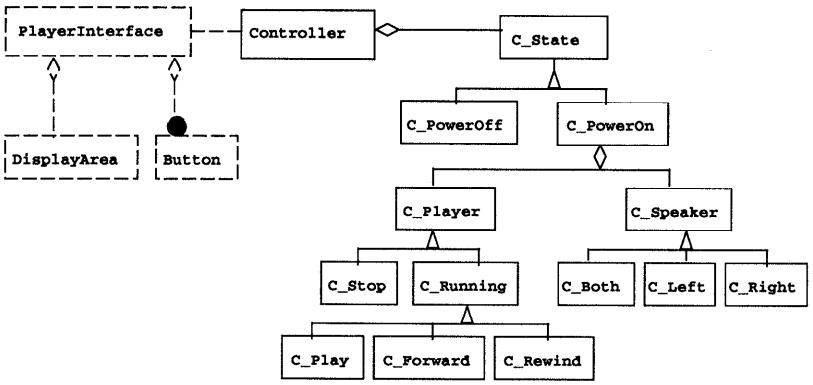


図 9 カセットプレイヤシステムのために生成されたコードのクラス構造
Fig. 9 Class structure of the generated code for the cassette player.

(図 3) から生成された実際のコードの一部を示している．コード生成に際しては，前章で述べた我々のアプローチに従う．コード生成のための詳細なルールは以下のようになる．

(1) C_State タイプの状態オブジェクト state をともなう Controller クラスが定義される．他のオブジェクトからの要求，すなわち状態図におけるイベントは，Controller クラスのメソッドになる．これらのメソッドの本体コードは，state オブジェクト内の同じメソッドへの呼び出しを含む．これが意味するのは，state オブジェクトに要求を任せてしまうということである．状態図におけるすべての動作は Controller クラスのメソッドになる．ユーザは，これらのメソッドに本体コードを付け加える．main メソッドは Controller クラスの中に宣言される．

(2) すべての状態クラスに共通のインタフェースを提供するために，抽象化クラス C_State が定義される．C_State は，状態図におけるすべてのイベントに対する操作の空宣言を含む．各々の状態クラスは，それ自身のイベント（操作）の実装コードを持つ．entry と exit 操作のための空宣言も C_State クラスの中に行う必要がある．

(3) クラスは各々の状態に対して宣言される．クラスの名前は，状態の名前に由来する．もし，状態が入

```
class Controller {
  public static C_State state;
  public void powerBut(){state.powerBut();}
  public void playBut(){state.playBut();}
  .....
  public static void setPowerOff(){...}
  public static void startPlay(){...}
  .....
}
class C_State {/* Empty declarations for entry(), exit()
    and all methods of the subclasses of C_State*/}

class C_PowerOn extends C_State{
  static C_Player player; static C_Speaker speaker;
  void entry() {player=new C_Stop(); player.entry();
    speaker=new C_Both();speaker.entry();}
  void exit(){player.exit();speaker.exit();}
  void powerBut() {exit();Controller.setPowerOff();
    Controller.state=new C_PowerOff();Controller.state.entry();}
  void playBut() {player.playBut();}
  .....
}
class C_Player {/* Empty declarations for entry(), exit()
    and all methods of the subclasses of C_Player*/}

class C_Stop extends C_Player {
  void entry(){Controller.setStop();}
  void playBut() {exit();C_PowerOn.player = new C_Play();
    C_PowerOn.player.entry();}
  void fwdBut() {exit();C_PowerOn.player = new C_Forward();
    C_PowerOn.player.entry();}
  void rwdBut() {exit();C_PowerOn.player = new C_Rewind();
    C_PowerOn.player.entry();}
}
class C_Running extends C_Player {
  void stopBut() {exit();C_PowerOn.player = new C_Stop();
    C_PowerOn.player.entry();}
}
class C_Play extends C_Running {
  void entry(){Controller.startPlay();}
  void exit(){Controller.stopPlay();}
}
.....
```

図 10 カセットプレイヤのために生成されたコードの一部
Fig. 10 Part of the generated code for the cassette player.

場動作や退場動作を持てば，entry と exit という名前を持つメソッドが各々，クラスの中に定義される．これらのメソッドの本体は，それに対応する入場動作や退場動作への呼び出しを含む．状態が他の状態の下位状態であれば，OR タイプや AND タイプの状態階層ができる．

OR タイプ状態階層： 下位状態に対応するクラスが上位状態に対応するクラスのサブクラスになる．上位状態からの遷移（図 3 では Running から stopBut）のためのメソッドはスーパクラスに定義され，サブクラスに継承される．下位状態からの遷移のためのメソッドがサブクラスで定義される．

AND タイプ状態階層： AND 状態の上位状態（図 3 では PowerOn）に対応するクラスは下位状態の数と同じだけのオブジェクトを含む複合クラスとなる．複合クラスには，各々の AND 状態のために状態オブジェクトが定義される．状態オブジェクトの名前とタイプは AND 状態の名前に由来する．複合クラスの entry メソッドは，AND 状態の中のデフォルト状態を表すオブジェクトを作り，これらのオブジェクトによって状態オブジェクトを初期化する．たとえば，player は C_Stop クラスのインスタンスとして初期化される．複合クラスの exit メソッドは，各々の状態オブジェクトの exit メソッドを呼び出す．AND 状態の各々のイベントのために複合クラスにメソッドが定義される．このメソッドはイベントに対応する AND 状態クラスのメソッドを呼び出す．

AND 状態に対応するクラスは抽象化クラスとなり，それ自身のサブクラスのインタフェースとして振る舞う．このクラスは他のどんなクラスからもサブクラス化されない．entry と exit 操作に加えて，その下位状態のイベントに対応する操作の空宣言を含む．

(4) どのような状態のイベントも対応するクラス内のメソッドとなる．メソッドのための本体コードもまた自動的に生成される．これには以下の 4 つが含まれる．(1) 現在の状態の exit メソッドを呼び出すもの，(2) 遷移の動作を呼び出すもの，(3) 現在の状態オブジェクトを削除し新しい状態クラスのオブジェクトを生成するコード，(4) 新しい状態の entry メソッドを呼び出すもの．

6. 生成コードの実行

前に述べたように，図 9 はカセットプレイヤ用の内部アプリケーションのクラス構造を示している．実線で描かれた枠はプレイヤの動的モデルから dCode によって生成されたクラス群を表し，点線で描かれたものはオリジナルのオブジェクトモデルからのドメインクラスを表す．Controller クラスのクライアントは，Controller の状態の振舞いを実装するクラスに関して知る必要はない．クライアントは Controller オブジェクトに要求を送るだけである．Controller オブジェクトは要求をその状態オブジェクトに送る．

Controller クラスは，main メソッドの中でインスタンス化された PlayerInterface クラスのオブジェクトを持つ．PlayerInterface オブジェクトをインスタンス化している間，Controller オブジェクトはそれ自身を引数として送る．それは，ユーザがあるボタンを押したときに，PlayerInterface オブジェクトが後ほど，Controller オブジェクトに要求を送り付けることができるようにするためである．PlayerInterface オブジェクトからの要求は Controller のイベントとなる．

コードをコンパイルした後，Controller クラスが

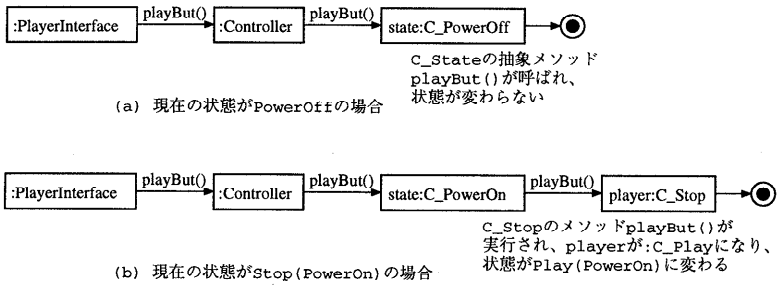


図 11 Play ボタンが押されたときの操作手順
Fig. 11 Sequence of operations when the Play button is pressed.

コマンドラインから実行されるとき，main メソッドが実行され，PlayerInterface オブジェクトが作られる．PlayerInterface オブジェクトは，図 1 のようなインタフェースを表示する．ユーザがあるボタンを押すとメッセージが Controller オブジェクトに送られる．Controller はその状態オブジェクト state に送ってきたメッセージを送る．メッセージ（イベント）に対して現在の状態から遷移があれば，それに対応するメソッドが実行され，状態が変化する．遷移がなければ，C_State クラスにある空のメソッドが実行される．**図 11** は Play ボタンが押されたときの操作手順を示している．長方形がオブジェクトを，線がメッセージを意味している．さらに，コロンが付いたクラス名がそのクラスのインスタンスを意味している．

7. Rhapsody との比較

我々は動的モデルから自動的に実行可能コードを生成するシステムとして dCode を開発した．我々が研究を開始した時点では分からなかったが，その後の関連研究の調査により状態階層や並列状態を扱えるツールとして Rhapsody[18] があることが判明した．本章では，dCode を Rhapsody と比較することによって，dCode の特徴を述べる．

Rhapsody[14),18)] は O-Mate[13)] の流れを汲み，システムのオブジェクト図，状態遷移図，メッセージシークエンスチャートを作り，それから自動的にシステムの C++コードを生成するツールである．状態図をコードに変換するまでの過程については文献 13)，14) には十分に触れられていないが，Rhapsody が生成するコードを見ることで理解可能である．

7.1 Rhapsody と dCode のメカニズムの比較

Rhapsody と dCode の両方は状態をオブジェクトとして扱う．Rhapsody ではイベントがクラスとなるが，dCode ではイベントをメソッドとして実装する．Rhapsody は状態階層をポインタを使って実装するが，dCode では状態階層が継承によって実装される．イベントが起こったときに，どの遷移が実行されるかをサーチするために Rhapsody は switch 文（条件文）を使っているが，dCode では実行される遷移がポリモフィズムの原則で決定される．**表 1** は Rhapsody と dCode のメカニズムの比較をまとめたものである．

表 1 Rhapsody と dCode が利用したメカニズムの比較
Table 1 Comparing the mechanisms used by Rhapsody and dCode.

状態図の要素	Rhapsody での実装	dCode での実装
状態	クラス	クラス
イベント	クラス	メソッド
状態階層	ポインタの利用	継承
遷移のサーチ	条件文	ポリモフィズム

表 2 Rhapsody と dCode が生成したコードの比較 (1)
Table 2 Comparing the compactness of the code generated by Rhapsody and dCode.

	Rhapsody	dCode
ソースコード：行数	1031	231
ソースコード：バイト数	19891	5410
クラス数	26	14

7.2 Rhapsody と dCode が生成したコードの比較

我々はカセットプレイヤの例を使い，Rhapsody によって生成されたコードと dCode が生成したコードとを比較した．比較を平等にするために Rhapsody によって生成された C++コードを Java コードに書き直す．比較から以下のようなことが分かる．

(1) dCode が生成したコードの方がよりコンパクトである．Rhapsody から生成された C++コードはあまりにも冗長である．Java に書き直した後では，コードは短くなってはいるが，我々のものよりもまだ 5 倍も長い（**表 2**）．

その理由は以下のとおりである．Rhapsody はすべての状態やイベントをクラスとして扱う．まず抽象

表3 Rhapsodyとと dCodeが生成したコードの比較 (2)
Table 3 Comparing the efficiency of the code generated by Rhapsody and dCode.

	Rhapsody (x) (ミリ秒)	dCode (y) (ミリ秒)	改良 $(x-y)/x*100$
遷移がないイベントにかかった総合時間 (a)	127.8000	54.3000	
遷移がないイベント1つにかかった平均時間 ($a/556$)	0.2299	0.0977	57.50%
遷移があるイベントにかかった総合時間 (b)	144.7500	114.6500	
遷移があるイベント1つにかかった平均時間 ($b/444$)	0.3260	0.2582	20.80%
すべてのイベントにかかった総合時間 ($a+b$)	272.5500	168.9500	
1つのイベントにかかった平均時間 ($(a+b)/1000$)	0.2726	0.1690	38.00%

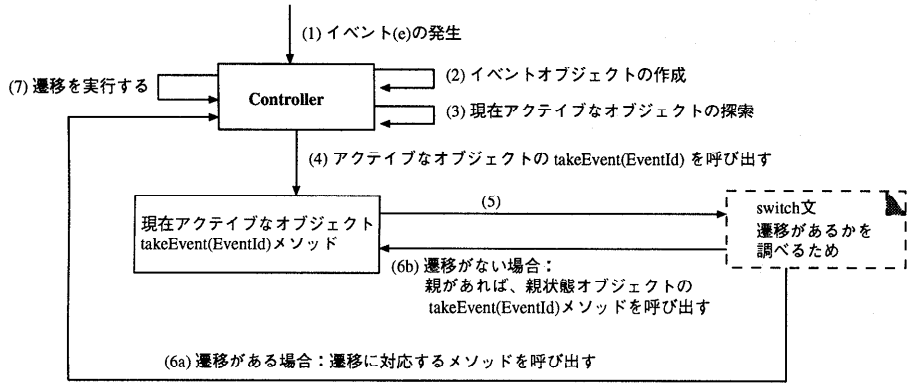
化クラスのStateからAndState, ComponentState, OrState, LeafStateという4クラスが派生する. 4クラスは各々AND状態の上位状態, AND状態, OR状態の上位状態, 末端状態という4タイプの振舞いを実装し, 大きなクラスである. 状態図においての状態はクラスとなり, 状態のタイプに依存する4クラスの1つにサブクラス化される. このように各状態は4タイプのいずれかのクラスに分類される. 各状態クラスにはイベント上での遷移を判別するためのswitch文(条件文)を含んでいるtakeEvent(EventId)メソッドがある. Controllerは状態図のすべての状態に対するオブジェクトを持っている. 各状態オブジェクトは上位状態オブジェクトへのポインタと状態がアクティブであるかどうかを教えてくれるboolean in()メソッドを持っている. 状態図においてのイベントも同じようにクラスとなる. 各イベントのクラスは抽象化クラスOMEventから派生する. 各々の遷移に対するメソッドはControllerクラスの中に定義される. この理由でRhapsodyが生成したコードは我々のコードより数多くのクラスができてしまい, ソースコードがあまりにも冗長である.

(2) 我々のコードの方がRhapsodyのコードよりさらに効率的である. dCodeとRhapsodyによって生成されたコードの効率を比較するために, 1000の要求の同じシークエンスをControllerクラスに送るという実験を行った. このような1000の要求に対して, 444は遷移し, 556のイベントは遷移されないで無視された. それぞれのイベントに対し, イベント処理が行った時間を計算した. すべてのアクションメソッドを空にして, 遷移(状態変化)を実行している時間のみを計測した. より正確な結果を得るため, 実験を20回繰り返してその平均をとった. 実験はSun SPARC Station 10上で行った. **表3**における実験結果によると遷移なしのイベントの処理において, 我々のコードは57.50%, Rhapsodyより効率的である. 遷移するイベントにおいて, 20.80%効率的である. 総合すると38.29%, Rhapsodyより効率的である.

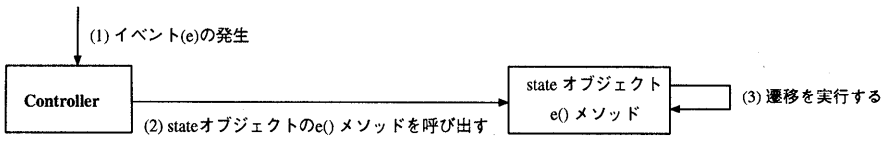
その理由は以下とおりである. **図12**に示すように, Rhapsodyが生成するコードの実行イメージはdCodeのものと異なる. Rhapsodyのコードでは, Controllerに他オブジェクトから要求が送られてくると, Controllerは要求をイベントと解釈しイベントオブジェクトを作る(イベントの発生). さらにControllerはアクティブな状態オブジェクトのtakeEvent(EventId)メソッドを呼び出し, イベントオブジェクトを渡す. takeEvent(EventId)メソッドに含まれているswitch文(条件文)から遷移があることが分かれば, Controllerの中での遷移に対するメソッドが呼ばれ実行される. 遷移がなければイベントは上位状態に対応するオブジェクトに送られる. Rhapsodyの実装ではこれらの実行にかなりの時間を費やさなければならない.

一方, dCodeでは状態はクラスになるが, イベントはメソッドになる. 状態階層は継承によって実装されるため, 下位状態オブジェクトが上位状態オブジェクトへのポインタを持たない. 各状態クラスにはそれに対応する状態からの遷移のメソッドが定義される. イベントが発生すると, イベント上に遷移がある場合, 遷移に対する現在の状態オブジェクトの中のメソッドが実行される. イベント上に遷移がない場合は, 状態オブジェクトの中にメソッドがないため, C_Stateの抽象メソッドが呼ばれるだけで他になにもしない. これは時間があまりかからない操作である. このようにdCodeではイベント上に遷移がないときの処理時間は大幅に短縮される. また, 遷移があるときでもある程度の時間を費やすがRhapsodyのような条件文を含まないため実行時間が短くなる.

(3) Rhapsodyのコードは理解しにくい. Rhapsodyは, 状態の振舞いは状態クラスを作って実装するが, 遷移探索コードを状態クラスのtakeEvent(EventId)メソッド内の条件文においてあてはめている. 遷移はControllerクラス内のメソッドとして実装され, 現在の状態オブジェクトがイベント上に遷移を見つけたときに実行される. これはコードを理解しにくくしている.



(a) Rhapsodyが生成したコードの実行イメージ



(b) dCodeが生成したコードの実行イメージ

図 12 Rhapsody と dCode が生成したコードの実行イメージ
Fig. 12 A conceptual view of the execution sequence in code generated by Rhapsody and dCode.

dCode が生成するコードはイベントを操作呼び出しに置き換えている．ポリモフィズムに基づいて適切なメソッドが選ばれる．遷移コードは対応する状態クラス内のメソッドに置き換えられている．それで，すべての状態や遷移は条件文を使うことなしに明示的になる．こうすることでより理解しやすいコードとなる．

8. 関連研究

最も関連深い研究は，Harel と Gery[13),14)] が開発した Rhapsody[18)] である．これはオブジェクトモデルと動的モデルから C++コードを生成するものである．前の章で示したように，dCode が生成するコードの方がよりコンパクトかつ能率的であり簡潔なものである．StateMaker[22)]，ROOM[26)]，および，StP[3)] も状態図からコードを生成できるが，これらは状態図における状態階層や並列な状態をサポートしていない．原田ら[9),10)] はオブジェクト指向システムの動的振舞いを実装するために状態図の代わりにイベントトレース図に注目している．彼らはイベントトレース図から C++コードを生成するためにイベントトレース図の新表記法を提案している．

我々の状態図から実装コードに変換するメカニズムは State パターン[7)] といくつかの共通点を持つが，State パターンは状態階層や並列な状態に関して考えていない．今回示したカセットプレヤの例には表さなかったが，ガード付き遷移やイベントなしの遷移の扱いが dCode では可能である．Joung ら[20)] はアイコンを状態図に付加させ，それからシステムをアニメーションさせるコードを生成する方法を研究している．

これらに加えて，他にも商用的に使用できる CASE ツールがある．これは様々な OMT の図をグラフィカルに描けるエディタであり，これからいくつかの実装コードを生成することができる．代表的なものは Rational Rose[27)] で，これは様々な UML (Unified Modeling Language)[28)] や OMT の図を作ることができるインタラクティブなグラフィカルエディタを提供している．Rational Rose は基本的に，モデリングやドキュメンテーションのツールであるためオブジェクトモデルからただヘッダファイルを生成するだけであり，状態図からコードを生成することはできない．Object Oriented Designer[21)]，Object Domain[25)]，および，MacA & D[6)] もオブジェクトモデルからただヘッダファイルを生成するだけである．生成されたコードはただ宣言部を含んでいるだけで，直接実行することはできない．

SoftReuse システム[16),33)] は，ER モデルをベースとした非手続き型の仕様記述言語 PSDL[17)] で記述されたドメインモデルからプログラム仕様を抽出し，C

プログラムを生成する．プログラム仕様を抽出する前に属性に対する制約を記述し，入出力データを指定する．これは，OMTでいえば，オブジェクトモデルと機能モデルのみからコードを生成することに相当する．本システムにおいては，動的モデルを考慮しないので，動的振舞いや制御を持つシステムへの応用は困難である．

9. 結　論

状態図によって表すOMTの動的モデルを実装コードに変換する方法を提案した．状態はクラス，遷移は条件文がない操作として表される．これは状態図のコンポーネントを明確にし，生成された実装コードは理解しやすいものとなる．この手法は状態図に含まれる状態階層や並列状態を扱う．

提案された手法は我々のコード生成システムdCodeに実装されている．これは，動的モデルの仕様を自動的に実行可能なJavaコードに変換できる．Rhapsodyと比較するとdCodeによって生成されたコードは処理速度においては約40%も効果的で，コードの量も5倍もコンパクトなものとなる．

我々のアプローチの適用範囲は，あるイベントに対してシステムの現在の状態に依存して異なる応答するreactiveシステムである．dCodeでは，状態図を1つしか扱えないという限界がある．これを解決するためには，dCodeの改良を行い，複数の状態図を同時に入力させてコードを生成することができるようにする必要がある．我々のアプローチは，オブジェクトの内部状態に依存するクラスの振舞いにもうまく実装できる．付け加えて，たいていのオブジェクト指向方法論[4),5),19)]がオブジェクトの動的な振舞いを表現するのに状態図を使うため，本論文の手法はOMT以外の方法論にも適用することができる．

謝辞 本研究の一部はEAGL事業推進事業の研究助成を受けて実施されたものである．本論文に関して有益な助言をいただいた丁錫泰氏と小川徹氏に感謝する．

参考文献

1) Ali, J. and Tanaka, J.: Automatic Code Generation from the OMT-based Dynamic Model, *Proc. 2nd World Conference on Integrated Design and Process Technology*, Austin, TX, Vol.1, pp.407–414, SDPS (1996).
2) Ali, J. and Tanaka, J.: Generating Executable Code from the Dynamic Model of OMT with Concurrency, *Proc. IASTED International Conference on Software Engineering (SE'97)*, San Francisco, CA, pp.291–297, IASTED (1997).
3) Aonix: *StP: Software Trough Pictures.* http://www.ide.com/index.html.
4) Booch, G.: *Object Oriented Design with Applications*, Benjamin/Cummings, Redwood, CA (1991).
5) Desfray, P.: *Object Engineering: The Fourth Dimension*, Addison-Wesley, Reading, MA (1994).
6) Excel Software: MacA&D. http://www.excelsoftware.com/index.html.
7) Gamma, E., Helm, R., Johnson, R. and Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA (1995).
8) Gosling, J., Joy, B. and Steele, G.: *The Java Language Specification*, Addison-Wesley, Reading, MA (1996).
9) 原田　実，藤澤照忠，寺平将高，山本幸二，濱田奨：SOMEにおける動的モデリングの詳細化と設計図の自動レイアウトCAMEO/DとC++プログラムからの設計図の逆生成OORE，OO'96シンポジウム，東京，pp.111–118，情報処理学会 (1996).
10) 原田　実，北本和弘，岩出隆志：制御構造とイベント送信を図示できる構造化オブジェクトモデリング環境SOME，OO'97シンポジウム，東京，pp.136–144，情報処理学会 (1997).
11) Harel, D.: Statecharts: A visual formalism for complex systems, *Science of Computer Programming*, No.8, pp.231–274 (1987).
12) Harel, D.: On Visual Formalisms, *Comm. ACM*, Vol.31, No.5, pp.514–530 (1988).
13) Harel, D. and Gery, E.: Executable Object Modeling with Statecharts, *Proc. 18th International Conference on Software Engineering*, pp.246–257, IEEE (1996).
14) Harel, D. and Gery, E.: Executable Object Modeling with Statecharts, *Computer*, Vol.30, No.7, pp.31–42 (1997).
15) Harel, D. and Naamad, A.: The STATEMATE Semantics of Statecharts, *ACM Trans. Software Engineering and Methodology*, Vol.5, No.4, pp.293–333 (1996).
16) 橋本正明，廣田豊彦，横田和久：ドメインモデルに基づくソフトウエア再利用に関する一実験，情報処理学会論文誌，Vol.36, No.5, pp.1040–1049 (1995).
17) Hashimoto, M. and Okamoto, K.: A Set and Mapping-based Dectection and Solution Method for Structure Clash between Program Input and Output Data, *Proc. Computer Soft-*

ware and Application Conference, pp.629–638, IEEE (1990).
18) i-Logix Inc.: *Rhapsody.* http://www.ilogix.com.
19) Jacobson, I.: *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley, Reading, MA (1992).
20) Joung, S., Ali, J. and Tanaka, J.: Automatic Animation from the Requirements Specifications based on Object Modeling Technique, *Proc. International Symposium on Future Software Technology* (*ISFST-97*), Xiamen, China, pp.133–139, Software Engineers Association, Japan (1997).
21) Kim, T.: Object Oriented Designer. http:// www.qucis.queensu.ca/Software-Engineering/blurb/OOD.html,ftp.x.org.
22) MicroGold Software: *StateMaker*, NJ, 08807. Internet 71543.1172@compuserve.com, http://www.worldwidemart.com/mattw/software/Windows3.X/demo/.
23) Nakashima, S., Ali, J. and Tanaka, J.: An Automatic Layout System for OMT-based Object Diagram, *Proc. 2nd World Conference on Integrated Design and Process Technology*, Austin, TX, Vol.2, pp.82–89, SDPS (1996).
24) 中島 哲，田中二郎：OMT 法に基づくオブジェクト図の自動レイアウトシステム，OO'96 シンポジウム，pp.103–110, 情報処理学会，東京 (1996).
25) Object Domain Systems: *Object Domain.* http://www.object-domain.com/.
26) ObjectTime Limited: *ROOM.* http://www.objectime.on.ca/.
27) Rational Software Corporation: *Rational Rose.* http://www.rational.com.
28) Rational Software Corporation: *Unified Modeling Language* (*UML*). http://www.rational.com.
29) Rumbaugh, J.: Controlling code, *Journal of Object-Oriented Programming*, Vol.6, No.2, pp.25–30 (1993).
30) Rumbaugh, J.: Objects in the Twilight Zone, *Journal of Object-Oriented Programming*, Vol.6, No.3, pp.18–23 (1993).
31) Rumbaugh, J.: OMT: The Dynamic Model, *Journal of Object-Oriented Programming*, Vol.7, No.9, pp.6–12 (1995).
32) Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorensen, W.: *Object-Oriented Modeling and Design*, Prentice Hall, Eaglewood Cliffs, NJ (1991).
33) Yokota, K., Hashimoto, M. and Sato, M.: An Experiment on Reusing Program Specifications Described with Conceptual Data Model and Dependency Constraint-based Language, *Proc. International Conference on Computing and Information*, pp.324–328 (1992).

(平成 10 年 2 月 9 日受付)
(平成 10 年 9 月 7 日採録)

Jauhar Ali（正会員）

1966 年生．1987 年 Peshawar 大学理学部卒業．1990 年同大学大学院計算機学科修士課程修了．1991 年より同大学 Islamia College 講師．1995 年筑波大学大学院工学研究科博士課程編入，1998 年博士（工学）．1998 年 9 月より筑波大学電子・情報工学系外国人研究者．オブジェクト指向方法論とその自動プログラムコード生成に興味を持っている．電子情報通信学会会員．

田中 二郎（正会員）

1951 年生．1975 年東京大学理学部卒業．1984 年 Utah 大学計算機科学科博士課程修了（Ph.D.）．新世代コンピュータ技術開発機構および富士通研究所を経て，1993 年より筑波大学電子・情報工学系助教授．プログラミング言語やヒューマンインタフェース，ソフトウェアの設計論に興味を持つ．新世代コンピュータ技術開発機構においては，並列論理型言語 KL1 の設計や実装の仕事に従事．最近はオブジェクト指向に基づく仕様からのアニメーションの作成や自動プログラムコード生成に興味を持っている．本学会グループウェア研究会連絡委員．ACM，IEEE Computer Society，日本ソフトウェア科学会，電子情報通信学会，人工知能学会，計測自動制御学会各会員．