ソフトウエア工学 93－1
（1993．7．8）

# プロトタイピングのためのオペレーショナルモデルOPM

宮本 衛市† 何 克清‡

†北海道大学工学部情報工学科 ‡武漢大学ソフトウェア工学国家重点研究実験室

ソフトウェア開発にとって、その上流工程である要求仕様の策定および基本設計などはきわめて重要であるとともに、最大の難所となっており、これに対してプロトタイピングが有力な手法の１つと考えられている。本稿では、オブジェクト指向に基づくプロトタイピングのオペレーショナルモデルを提案する。このモデルは状態遷移を基本的な枠組とし、一方では階層的にソフトウェアの状態を詳細化してゆく過程を横糸とし、他方ではソフトウェア機能をより基本的な機能部品に展開してゆく過程を縦糸として、両方向の等価性を保ちながらソフトウェア開発の上流工程を織りなしてゆこうとするものである。このモデルの特徴は、ソフトウェア機能の記述が上記の横および縦のいずれの方向でも上位で閉じていることであり、したがって任意の詳細化でソフトウェア機能のシミュレートができ、さらに詳細化を続けてゆけば概念設計あるいは詳細設計へと連続的に進めてゆくことができることである。

# Operational Model OPM for Prototyping

Eiichi Miyamoto† Keqing He‡

†Division of Information Engineering, Faculty of Engineering, Hokkaido University
Kita 13 Nishi 8, Kita-Ku, Sapporo 060, Japan
‡State Key Laboratory of Computer Software Engineering, Wuhan University
Wuhan 430072, People's Republic of China

For software development, it is very important and is considered to be the most difficult problem to meditate the software specification and its fundamental design. Prototyping technique has proved to be quite powerful for the preliminary phases of software development. In this report, we propose an operational model for prototyping based upon object-oriented state transition. In this model, on the one hand, states of software are refined into more detailed states hierarchically, and on the other hand functions of software are refined into more fundamental function elements. Both of the refinements preserve the functionality of the software by equivalence relations. At any stage of refinement, the prototype constitutes a valid description of the software at that level of abstraction. Therefore, software can be simulated at arbitrary refined stages, and be developed towards conceptual or detailed design continually.

# 1 はじめに

高品質で、かつ頑健なソフトウェアを構築するために、ソフトウェアプロトタイピングはきわめて有効な手法と考えられている。特に、ソフトウェアの要求仕様を実証するために、計算機を利用してソフトウェア機能をシミュレートすることは、プロトタイピングの主たる目的である。そこで、プロトタイピング向けのオペレーショナルな仕様を記述するためには、オペレーショナルモデルが必要である。特に、プロトタイピングは人と計算機とのインタフェースが重要であり、従って、可視化が容易で、かつ記述力のあるオペレーショナルモデルが要求され, さまざまなモデルが提案されてきた [1],[2],[3],[4],[5] 。これらの中で、Harel の Statecharts モデルは、基本的には有限状態機械モデルを拡張したもので、状態遷移図を用い、構造のあるシステムの状態を把握するために、状態の詳細化の枠組を与え、大規模システムの挙動解析に強力な手段を提供している。しかし、ソフトウェアシステムの仕様を記述するためには抽象化した記述能力が不十分であり、他のモデルによる記述の支援を受けなければならない [6] 。

Coleman らは Statecharts を応用したオブジェクトの設計法を提案した [7] 。そこでは、システムを構成するオブジェクトを状態遷移に基づいて設計しようとしており、各オブジェクトをオブジェクト向けに Statecharts を拡張した Objectcharts で記述する。酒井はオブジェクト指向によるシステムの概念設計のため、オブジェクトの振舞いに基づくコントラクトの概念を定義した [8] 。これにより、オブジェクト固有の振舞いの表現を基礎とし、オブジェクト間での、一貫性制約や振舞い協調の記述に基づいて設計を進めていく方法を提案している。これらに対し、、われわれはオブジェクトの各状態をオブジェクト化し [9] 、プロトタイピングのオペレーショナルモデルとして十分な柔軟性と記述力のあるモデル OPM(Object-oriented Prototyping Model) を提案する。これは機能オブジェクトの状態の段階的詳細化を横糸とし、各機能オブジェクトをより基本的な機能オブジェクトに分解していく過程を縦糸とし、この両者が同値関係を保ちつつ 2 次元的にソフトウェアシステムを展開していくために基本となるモデルである。

以下、第 2 章では、われわれが考えているプロトタイピングモデルに対する基本的な立場を明らかにする。第 3 章では、オブジェクト指向と状態遷移に基づくプロトタイピングのオペレーショナルモデルを定義する。第 4 章では、OPM の適用例を紹介し、第 5 章では本稿で提案したモデルの問題点を議論し、今後の研究課題を挙げる。

# 2 オブジェクト指向に基づく階層的プロトタイピングプロセス

プロトタイピングでは、ソフトウェアの要求仕様を機能に基づいてシミュレートすることを目的としている。この機能を明解に分析するために、機能オブジェクトをオートマトンに見立てて状態遷移を作成し、各状態を機能的に扱うために、状態オブジェクトで表現しようとするのがわれわれの意図である。しかし、ソフトウェアが大規模になるとその機能も複雑になり、状態遷移も複雑化し、その結果ソフトウェアの機能の見通しが悪くなるおそれが出てくる。これに対し、2 つのアプローチでの詳細化が考えられる。1 つは状態の詳細化であり、他の 1 つが機能の詳細化である。

ソフトウェア機能の状態とは、あるレベルの観点から分類したソフトウェアの内部状態の同値類である。したがって、ソフトウェアに対する見方が詳細化していけば、状態数も当然増えていくはずである。このとき、それ以前に分類されていた状態を御破算にして新らたな分類を行なうのではなく、すでに分類されている個々の状態を、必要に応じさらに詳細化しようとするのが、われわれの提案するモデルにおける 1 つの特長である。

一方、ソフトウェア機能の詳細化は、その機能を実現するための要素機能に展開し、それらをどのように合成すれば本来の機能を発揮できるのかを指示することでも可能である。これは、いわば

機能部品への展開であり、個々の機能も再び状態遷移を伴って定義されるものとする。このとき、機能部品は親の機能の、いわば内部部品ともいうべきもので、親の機能の中に隠蔽されていることを原則とする。

図1はプロトタイピングの過程を模式的に表したものである。同図で横方向への詳細化は、逐次プロセスの詳細化に対応し、状態オブジェクトをさらに細分した状態オブジェクトに再帰的に詳細化を進めていく過程である。したがって、この過程は同値類で分類したソフトウェア機能をさらに細分することを意味する。

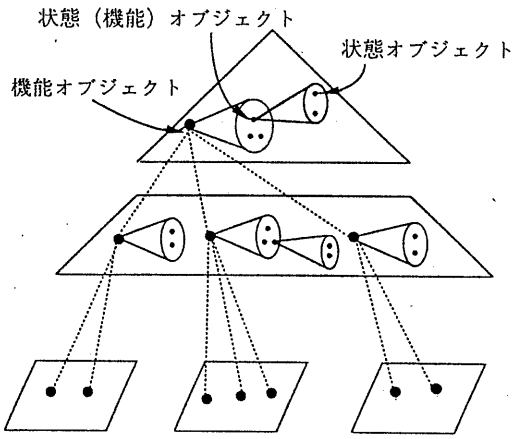


図 1: ソフトウェア機能の詳細化

一方、縦方向への詳細化は並列プロセスによる詳細化に対応し、より基本的な機能オブジェクトで元の機能オブジェクトの実現を図るプロセスである。これは、いわば親の機能オブジェクトの同値類によるソフトウェア機能の分類を、より基本的な機能要素の同値類による分類で構成される直交空間へ写像することを意味する。上位の機能オブジェクトは、基本的にはそれのみで仕様記述を完結させることができる。下位の機能オブジェクトは、より基本的な機能要素をもち、上位で記述された仕様の再定義に相当する。しかも、上位と下位の仕様の間には、等価な同値類の対応を前提としている。

Harel らは、直並列にシステムの状態を詳細化してゆく過程を仕様軸とし、その仕様を実現してゆく上での条件等を逐次考慮しながらシステムを直並列状態で詳細化してゆく過程を実現軸として、関数的に捉えられないリアクティブ・システムを設計してゆくべきことを主張している [10]。しかし、直並列プロセスと直並列プロセスの直交化によるシステムの詳細化の意味付けは難しく、そこでは具体的な方法論には触れていない。本稿は、直列プロセスと並列プロセスの直交化によりシステムの挙動を詳細化してゆき、仕様策定と概念設計を、いわば連続的に進めてゆくことのできるモデルを提案しようとするものである。

# 3 オペレーショナルモデル OPM

本章では、プロトタイピング時においてオブジェクトが置かれる環境を述べ、それに基づくプロトタイピングのオペレーショナルモデル OPM を提案する。

## 3.1 オブジェクトの環境

OPM で基本になるのは機能オブジェクトの定義である。オブジェクトは他のオブジェクトから発せられたメッセージを受取ると、そのメッセージに対応した行動を起こすものとする。メッセージの伝達は、いわばオブジェクトの外部で起こっていることであり、これをイベントと呼ぶ。これに対し、メッセージに対応した行動はオブジェクトの内部で行なわれることであり、オブジェクト機能の実現に対応する。

いま、図1に示すオブジェクト群を、ある時点での動的な振舞いを示すと図2のようになる。状態オブジェクトは機能オブジェクトにつき1つだけ存在している。イベント $e$ はフィールド $P$ に存在する機能オブジェクトに伝達され、各オブジェクトは自身の状態オブジェクトに転送する。もし、フィールド $P$ にいくつかの機能オブジェクトが存

在するときには、それらオブジェクトが発生するイベントもフィールド内の機能オブジェクトに伝達される。各機能オブジェクトは受け取ったイベントを状態オブジェクトに転送し、末端により近い状態オブジェクトがそのイベントを優先的に受理する。もし、そのイベントを受理する状態オブジェクトが存在しなければ、そのイベントは機能オブジェクトで一時ホールドされるものとする。

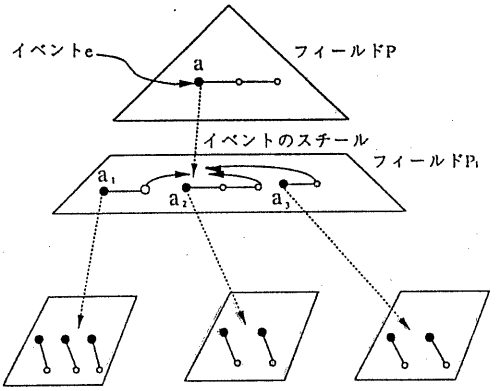


図 2: オブジェクトの動的振舞い

一方、機能オブジェクト $a$ を機能要素で詳細化した機能要素オブジェクト群 $a_i(i=1,2,3)$ はフィールド $P_1$ を構成している。$P_1$ では各 $a_i$ が $a$ とは独立に活動し、従って $P_1$ の存在を $a$ は関知する必要はないが、$a_i$ は親の機能オブジェクトである $a$ が受取ったイベントを盗み取るものとする。$a_i$ が相互に発生するイベントはフィールド $P_1$ 内に伝達されるが、これらはいわば内部的なイベントである。一方、親の $a$ が発生するイベントに相当するイベントも、$a_i$ のいずれかから発生されるはずであるが、各フィールドは閉じているので、フィールド外に伝達されることはない。

## 3.2 機能および状態オブジェクトの定義

前節のようなオブジェクト環境を前提として、機能オブジェクトを次のように定義する。

機能オブジェクト ::=

object *object_name* :

stateset {*state_name*, }$^+$;

{state *state_name* :

{*receiving_event_name*/

*next_state_name*,

*sending_event_name*

({*object_name.state_name*, }$^+$);

}$^+$

}$^+$

end (1)

ただし、イタリック体はメタ変数を、{}$^+$は 1 回以上、{}$^*$は 0 回以上の繰返しを意味するメタ記号を表し、式を簡単にするため最後に現われる区切り記号は省略できるものとしている。上の定義では、まず機能オブジェクトに名前を与え、次いで stateset 行で状態オブジェクト名を列挙し、その後各状態オブジェクトごとに、受け付けるイベントとそれに対する遷移先の状態オブジェクト名、および発生するイベント名とその送り先のオブジェクト名を定義する。

機能オブジェクトのある状態オブジェクトをさらに詳細化するときには、その状態オブジェクトを機能オブジェクトと見なし、次のように定義する。

状態オブジェクト ::=

object *object_name.state_name* :

stateset {*state_name*, }$^+$;

in {*state_name*(*event_name*), }$^*$;

out {*state_name*(*event_name*), }$^*$;

{state *state_name* :

{*receiving_event_name*/

*next_state_name*,

*sending_event_name*

({*object_name.state_name*, }$^+$);

}$^+$

}$^+$

end (2)

ここで、詳細化される状態オブジェクトは、親の機能オブジェクトの名前を前につけて呼ぶことにする。 in および out を先行させた行が (1) 式と異なる箇所であり、in 行はかっこ内のイベントが発生してこの状態オブジェクトに遷移したときの初期状態オブジェクトを示し、イベントが省略されているときは、イベントには無関係にとる初期状態オブジェクトを示す。out 行はこの状態オブジェクトから遷移するときの内部状態オブジェクトを表わす。もし、out 行が省略されていれば、この状態オブジェクトは任意の内部状態オブジェクトから遷移することができるものとする。ただし、in 行および out 行で指定するイベントは親の状態オブジェクト群が受取るイベントである。(2) の本体で、この状態オブジェクトが詳細化された、いわば状態内部の状態遷移を記述する。

### 3.3 機能オブジェクトの設計詳細化

機能オブジェクトを、より基本的な機能オブジェクトに展開するとき、次のようにオブジェクトの分解を定義する。

オブジェクト分解 ::=
　decomposition *object_name* :
　　$\{more\_basic\_object\_name,\}^+$
　end　　(3)

新しく定義すべき機能オブジェクトは (1) を用いて記述すればよいが、3.1 で述べたように親のオブジェクトが受取るイベントを盗み取るので、親のオブジェクトが受け取るイベントを含めて定義しておかなければならない。

(3) で定義されたオブジェクト群は、静的な枠組が記述されたにすぎない。そこで、展開した後でも、親の機能オブジェクトの状態の意味が保持されるように、展開前後における状態の同値関係を次式により宣言する。

同値関係 ::=
　equivalence *object_name* :

$$\{S_i^o = \sum_k S_{j1_k}^{o_1} S_{j2_k}^{o_2} \ldots S_{jn_k}^{o_n};\}^+$$

　end　　(4)

ここで、$S_i^o$はオブジェクト $o$ が状態 $i$ にいることを表す述語と考え、上式は親の機能オブジェクトの状態 $i$ が機能要素オブジェクト $o_1 \sim o_n$の状態の組合せを論理積で表したもののうち、状態 $i$ と同等のものを論理和で集めて記述することを意味している。実際には、状態を基本命題とみなせば、組合せ回路の簡略化の問題に帰着させることができる。

(4) 式による宣言は、いわば機能を詳細化していくときのアサーションに相当し、詳細化される各層間での制約を記述している。この制約を基にして、ソフトウェアのプロトタイピングを進めていく過程での検証を行なうことができる。

## 4 OPM の適用例

OPM は要求仕様の詳細化と設計の詳細化の両者に適用することができ、さらにこの両者を統合することができるモデルでもある。そこで、本章では多機能時計を例に取上げ、OPM を用いてプロトタイピングを行なってみる。

### 4.1 多機能時計の仕様作成

ここでは、通常の時計機能のほか、アラーム機能およびストップウォッチ機能つきの多機能時計を考えてみる。この時計は電池を挿入すると働き始め、その時点では時刻表示を行なうが、左ボタンを押すと電子ブザーを 1 回鳴らしてアラーム表示に、さらに左ボタンを押すとストップウォッチ表示に切り換り、さらに左ボタンを押すと時刻表示に変わるものとすると、この時計の機能は OPM では次のように表わすことができる。なお、イベントの送り先は特定されているものとして、以降の記述では割愛する。

object 多機能時計:
　stateset 時刻表示, アラーム表示,

ストップウォッチ表示;
in 時刻表示 (電池挿入);
state 時刻表示:
左ボタン押し/アラーム表示,
電子ブザー 1 回鳴らし;
state アラーム表示:
左ボタン押し/ストップウォッチ表示,
電子ブザー 1 回鳴らし;
state ストップウォッチ表示:
左ボタン押し/時刻表示,
電子ブザー 1 回鳴らし;
end (5)

ここで、多機能時計オブジェクトの相手となるオブジェクトは利用者を想定している。

さらに、時刻表示の仕様の詳細化を進める。時刻表示には時計としての時刻表示モードと表示時刻の修正モードを考えると、多機能時計の時刻表示オブジェクトの機能は次のように表わされる。

object 多機能時計. 時刻表示:
stateset 時計モード, 修正モード;
in 時計モード (左ボタン押し);
out 時計モード (左ボタン押し);
state 時計モード:
右ボタン押し/修正モード,
点滅表示への切換え;
state 修正モード :
右ボタン押し/時計モード,
非点滅表示への切換え;
end (6)

ここで、右ボタンがモード切換えに使われており、時刻表示から脱出するためには時計モードになっていなければならないことを示している。

アラーム表示やストップウォッチ表示の各状態の詳細化も行なって、多機能時計の OPM による仕様を図示すると、図 3 のようになる。

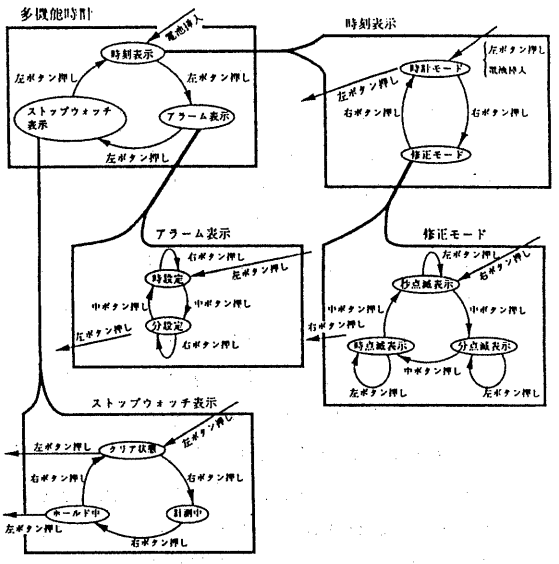


図 3: 多機能時計の OPM による仕様図

## 4.2 多機能時計の設計

前節の仕様に基づき、多機能時計の概念設計を行なってみる。(1) の機能仕様から、機能要素として時刻処理部、アラーム処理部、ストップウォッチ処理部、表示部および電子ブザーを考え、これら機能要素を宣言する。

decomposition 多機能時計:
時刻処理部, アラーム処理部,
ストップウォッチ処理部, 表示部, 電子ブザー
end (7)

各機能要素オブジェクトは、親の多機能時計へのイベントを盗み取って、自分のイベントとする。例えば、時計処理部を次のように設計することができよう。

object 時刻処理部:
stateset 時刻表示中, 時刻非表示;
in 時刻表示中;
state 時刻表示中:
左ボタン押し/時刻非表示,
アラーム処理部の起動;

```
state 時刻非表示:
  時刻処理部の起動/時刻表示中,
    表示時刻の送出;
end                                    (8)
```

ここで、時刻非表示が受け取るイベントは、ストップウォッチ処理部が発する機能要素間で授受するイベントである。

多機能時計の状態と各機能要素の状態との関連を記述すると、次のようになる。

```
equivalence 多機能時計:
  時刻表示 = 時刻処理部.時刻表示中
     *アラーム処理部.アラーム非表示
     *ストップウォッチ処理部.
      ストップウォッチ非表示;
  アラーム表示 =
      アラーム処理部.アラーム表示中
     *時刻処理部.時刻非表示
     *ストップウォッチ処理部.
      ストップウォッチ非表示;
  ストップウォッチ表示 =
      ストップウォッチ処理部.
      ストップウォッチ表示中
     *時刻処理部.時刻非表示
     *アラーム処理部.アラーム非表示;
end                                    (9)
```

上の宣言は、多機能時計の状態が3つの機能要素の連繋動作で実現されることを表わしている。

このような機能要素で詳細化した多機能時計を図示すると、図4のようになる。

## 5 おわりに

本稿では、ソフトウェア機能のプロトタイピングのためのオペレーショナルモデルを、オブジェクト指向に基づき、状態遷移の基本的な枠組の上で構築したことについて述べた。ここで、ソフトウェアの状態とは、ソフトウェアの動作状態を何らかの意味で同値分割したときの個々の要素である。OPMはこの要素をさらに細分化する状態詳細化の方向と、同値分割を多次元への同値分割へ展開する機能詳細化の方向を統合したモデルである。

本稿で提案したOPMでは、状態の識別子のみで各状態を表現しており、状態の属性を表現する手段を陽には定義していない。Z [11], Objectchartsなどでは属性を表す変数とその性質を述語で宣言する手段を用意しており、今後OPMの言語化において考慮すべき課題である。

また、機能要素オブジェクトに展開したときのイベントのやりとりによるオブジェクト間の協調動作は、図4に示すように一般にきわめて錯綜するため、なんらかの構造化を図ることが不可欠であり、今後の大きな課題である。

## 参考文献


[1] Zave, P.: An Operational Approach to Requirements Specification for Embedded Systems, *IEEE Trans. Software Eng.*, Vol.SE-8, No.3, pp.250-269 (1982).

[2] Balzer, R.: Operational Specification as the Basis for Rapid Prototyping, *ACM SIGSPFT Software Eng. Notes*, Vol.7, No.5, pp.3-16 (1982).

[3] Wasserman, A. I.: Extending State Transition Diagram for Specification of Human-Computer Interaction, *IEEE Trans. Software Eng.*, Vol.SE-11, No.8, pp.699-713 (1988).

[4] De Marco, T: Structured Analysis and System Specification, *Prentice-Hall*(1989).

[5] Harel, D.: Statecharts: A Visual Formalism for Complex Systems, *Science of Computer Programming*, Vol.8, pp.231-274 (1987).

[6] Harel, D., Lachover, H., Naamad, A., Pnueli, A., Politi, M., Sherman, R., Shtull-Trauring, A. and Trakhtenbrot, M.: STATEMATE: A

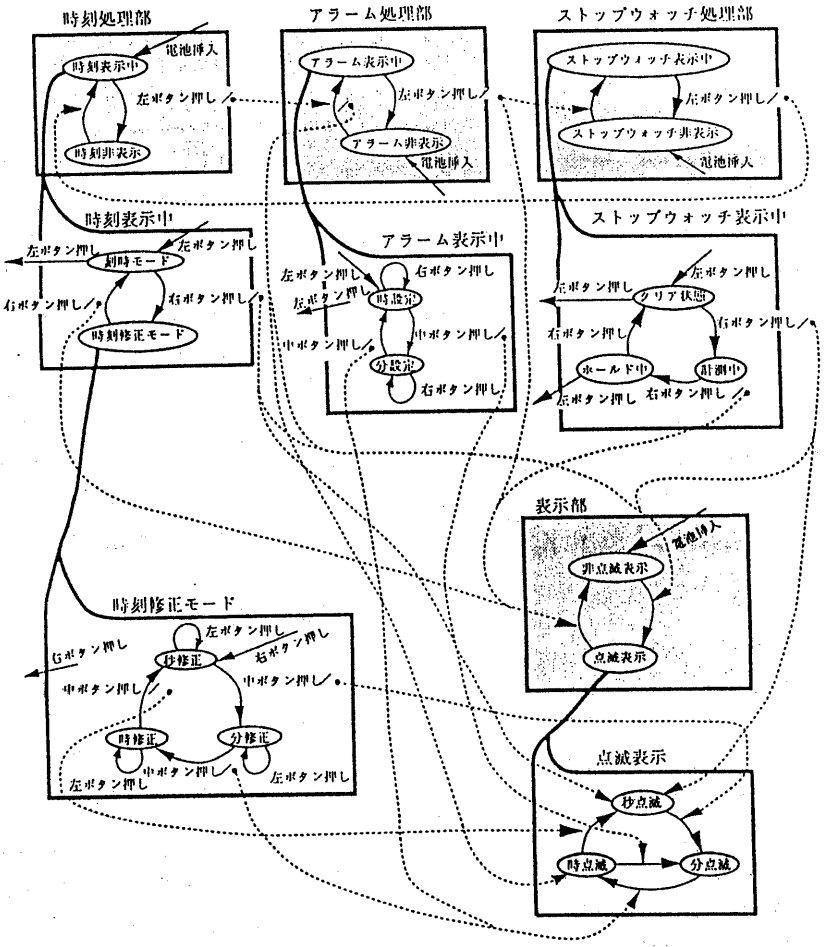


図 4: 機能要素による多機能時計の詳細化


Working Environment for the Development of Complex Reactive Systems, *IEEE Trans. Software Eng.*, Vol.SE-16, No.4,pp.403-414 (1990).

[7] Coleman, D., Hayes, F. and Bear, S.: Introducing Objectcharts or How to Use Statecharts in Object-Oriented Design, *IEEE Trans. Software Eng.*, Vol.SE-18, No.1, pp.9-18 (1992).

[8] 酒井博敬：オブジェクトの振舞いに関するコントラクトの設計について，情報処理学会論文誌, Vol.33, No.8, pp.1052-1063 (1992).

[9] 松永, 小山, 松井, 橋本, 山本：オブジェクト指向によるグラフィカルユーザインタフェース記述について，情報処理学会 オブジェクト指向ソフトウェア技術シンポジウム論文集, Vol.91, No.2, pp.77-86 (1991).

[10] Harel, D and Pnueli, A.: On the Development of Reactive Systems, *Logics and Models of Concurrent Systems*, pp.477-498 (1985).

[11] Spivey, R.: The Z Notation - A Reference Manual, *Prentice Hall* (1989).