ソフトウェア基礎論 28－1
（1989. 3. 10）

マルチウィンド・エディタシステムのオブジェクト指向仕様記述法

Object-Oriented Specification Description Method for Multiwindows-Editor Systems

堀川 英明　　田山 典男
Hideaki Horikawa　　Norio Tayama
（岩手大学）
(Iwate University)

あらまし　本論文では、エディタや対話型ユーザインタフェースに対して、その操作的仕様から実行可能なプログラムを生成することを目指すという立場から、オブジェクト指向ＲＳＴモデルとそれに基づく仕様記述言語ＯＲＳＴ／ＳＰＥＣを提案している。このモデルは、エディタの論理的構成を表現するオブジェクト指向モデル、オブジェクトの内部仕様を表現する状態遷移モデルと刺激応答モデル、オブジェクト間の関係仕様を表現する制約関係モデルからなる複合モデルである。エディタ仕様記述への適用例を示す。

Abstruct　In this paper, we propose a new object-oriented model and an operational specification description language for editor systems and interactive user-interface-processing systems. We call it "Object-oriented RST model". The model consists of object-oriented model, constraint-relationship model, stimulus-response model and state-transition model. We indicate the operational specification descriptions of a multiwindows editor by means of the language based on the object-oriented RST model.

1．まえがき

近年、新しい計算機環境や利用方式が開発されるのにともなって"使いやすい"ユーザインタフェースが要求されるようになり、多くのアプリケーションシステムにおいて、マルチウィンドウ機能などを備えた対話型の高度なユーザインタフェース環境も実現されるようになった。しかし、このようなシステムでは、システムの状態や処理の対象とされるデータがユーザからの指令によって任意に変更されるという特徴があり、システムの複雑さに起因する開発負担の増大が問題となる。これに対して、システムのユーザインタフェース部の仕様記述法を与え、それを実行可能なプログラムに変換し実行するというプロトタイピングシステムが提案されている[1]。しかし、ユーザインタフェース処理だけを扱うようなプロトタイピングシステムでは、エディタのように、ユーザインタフェース処理と編集対象に対する内部的処理とが密接にかかわっているようなシステムに対しては、必ずしも有効ではない。

このような背景から、エディタのような対話型のユーザインタフェースをもつアプリケーションシステムについて、その仕様を直観的にわかりやすく記述できる仕様記述法とその仕様記述から実行可能なプログラムを生成できるシステムが実現できるならば非常に有効であると考えられる。

一方、オブジェクト指向モデル[2]は、人間が実世界の事物をとらえるときの考えかたに近く自然であるということから、プログラミング言語として注目されており、最近、このオブジェクト指向モデルを仕様記述言語にも取り入れようとする試み的考察が報告され

ている[3]。オブジェクト指向はソフトウェアの部品化・再利用の観点からみても優れた性質をもっている。

そこで、本稿では、対話型のユーザインタフェースをもつアプリケーションシステムとしてエディタに焦点を当て、オブジェクト指向の操作的仕様から実行可能なプログラム生成するシステムの開発を目指している。そのために本稿では、まずエディタの仕様記述を行うために要求される事項を明らかにし、エディタの仕様記述に適したモデルとして、オブジェクト指向モデル、状態遷移モデル[1][4]、刺激応答モデル[3]を融合し、これに制約関係[5][6][7]の概念を加えた新しい複合モデルを設定する。さらに、このモデルに基づいてプログラム仕様を記述するための仕様記述言語ＯＲＳＴ／ＳＰＥＣを設定し、実行可能なプログラムへの変換方法の概略を示す。この複合モデルを『オブジェクト指向ＲＳＴ（Relationship, Stimulus-response and state-Transition）モデル』と呼ぶ。

2．エディタ仕様記述法の開発方針とアプローチ

エディタは対話型システムの典型的な例であり、またユーザからの指令によってシステムの状態や編集対象をランダムに操作できる機能をもつ必要があることから、その開発においては、次のようなことが問題となる。

（１）　対象データの処理がユーザインタフェースの処理と絡んで、いずれの処理も複雑になりがちである。

（２）　対象データの処理とユーザインタフェースの処理との完全な分離が難しい。

（３）　システム状態や対象データをランダムに操作する処理では、数値計算を行うようなバッチ型の処理と比べて、プログラム全体の軸となるような基本的なアルゴリズムが必ずしも明確に定まらないため、プログラムの開発者による個人差が出易くなる。

以上のような理由から、エディタを実現するプログラムは、理解容易性、柔軟性、保守性、再利用性などの面で問題が生じ易いと考えられる。

一方では、エディタのような対話的な性格の強いシステムほどユーザの好みによってシステム仕様の変更が要求される可能性が高いため、その開発が試行錯誤的に行えることが望ましい。しかし、理解容易性や柔軟性が悪いプログラムでは、試行錯誤的な開発は困難である。

以上のような種々の問題点に対して、その解決を図るべく本研究では次のようにアプローチした。

（１）　エディタにおけるユーザインタフェース処理と内部データに対する処理との両方の仕様を統一的に記述するためのモデルとして、オブジェクト指向ＲＳＴモデル』を設定する。

（２）　オブジェクト指向ＲＳＴモデルに沿ってプログラムの仕様を記述できるような仕様記述言語を設定する。

（３）　仕様記述言語によって記述された仕様を実行可能なプログラムに自動的に変換する変換システムを作り、自動生成されたプログラムを実際に実行してみることによりその振る舞いを確かめることができるようにする。

なお本研究においては、仕様記述の対象としては、簡単なマルチウィンドウスクリーンエディタを対象としてモデルの設定を行っている。また、仕様記述からプログラムへの変換では、変換対象言語を「オブジェクト指向プログラミング言語Ｃ＋＋」[10]に設定している。

3．エディタのモデル化
－オブジェクト指向ＲＳＴモデル－

3．1　オブジェクト指向ＲＳＴモデルによるエディタのモデル化

図１にマルチウィンドウエディタの概念的構成を示す。このような、マルチウィンドウエディタをオブジェクト指向ＲＳＴモデルでは、図２に示すようにモデル化する。図２は、画面上に３枚のウィンドウが開かれている状態を表している。まず、各ウィンドウはそれぞれ「ウィンドウ」、「テキストバッファ」、「編集制御部」の３つのオブジェクトから構成される。そして、これら全体を管理するための「ウィンドウ管理部」のオブジェクトがある。また、ここでは「キーボード」と「ディスプレイ」もそれぞれオブジェクトとして考える。

さて、オブジェクト指向ＲＳＴモデルでは、これらのオブジェクトの内部仕様を、その振る舞いの特性によって「状態遷移モデル」、「刺激応答モデル」のいずれかによってモデル化する。またオブジェクト間の関係は制約関係によって表現する。これらについては、以下の節で順に述べる。

3．2　オブジェクトの内部仕様を表現する状態遷移モデルと刺激応答モデル

一般にオブジェクト指向モデルでは、オブジェクトの内部の記述については、ほとんどの処理をメッセージ通信の形で記述するということ以外に、特にこれといった規範はない。したがって、システムの構造はうまく表すことができても、オブジェクトの内部仕様は必ずしも理解しやすいものとはならない。そこで、オブジェクト指向ＲＳＴモデルでは、オブジェクトを状態機械としてとらえて、オブジェクトの内部仕様を「状態遷移モデル」と「刺激応答モデル」のいずれかのモデルにしたがって記述することによりオブジェクトの振る舞いの意味をより明確に表現する手法を提供しようとする。

そこで、まずオブジェクトの状態を次のように定義

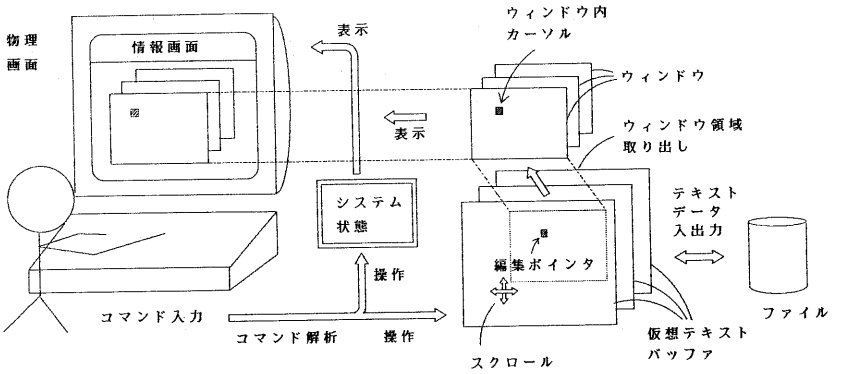

図1　マルチウィンドウエディタの概念的構成

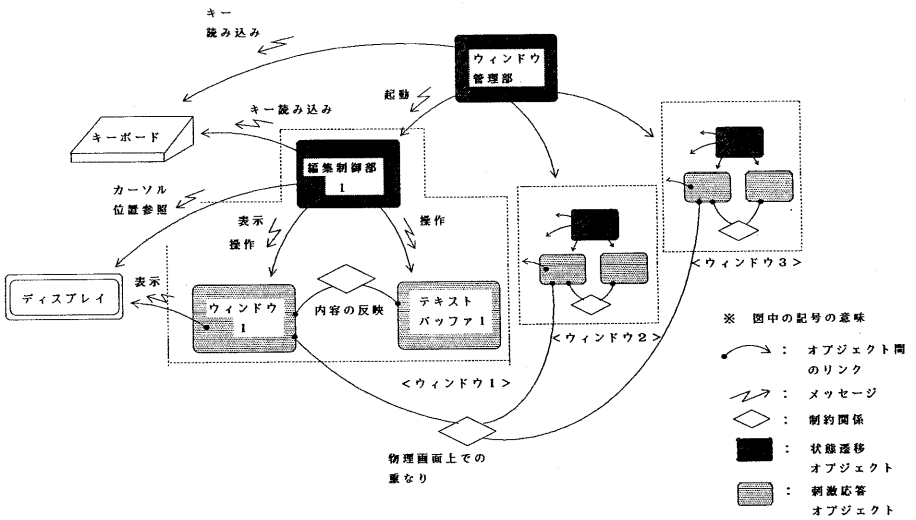

図2　オブジェクト指向ＲＳＴモデルによるマルチウィンドウエディタのモデル化

する。

［オブジェクトの状態の定義］

オブジェクト Obj の状態とは、Obj がもつインスタンス変数　v1 , v2 , ... , vN がもつ値と、インスタンスメソッドの実行の進行状況を表す C とのベクトルである。

ここで、まずインスタンスメソッドの実行の進行状況によって決まる状態を第一義にとると、オブジェクトの振る舞いを状態遷移モデルによってとらえることができる。そして、このようなオブジェクトを「状態遷移オブジェクト」と呼ぶ。

本来、状態遷移モデルは、外部との入出力によってシステムの状態が遷移していく振る舞いをうまくモデル化できるという特長をもっている。また、アプリケーションシステムのユーザインタフェース部の仕様を状態遷移図によって記述し、その記述から実行可能なプログラムを導出するといったプロトタイピングシステムへ応用している例もある[1]。

そこで、オブジェクト指向ＲＳＴモデルでは、エディタシステムを構成するオブジェクトのうち、コマンド入力やメッセージ出力などのユーザとの対話部分を司るオブジェクトについては、状態遷移オブジェクトとしてモデル化する。状態遷移モデルによって対話処理をとらえることにより、例えば「ユーザからコマンドを受け取った後にエディタの状態がどのように変化し何をするのか」といった、システムの状態遷移がより明確に表現できる。図２では、ウィンドウ管理部オブ

ジェクトや編集制御部オブジェクトを状態遷移オブジェクトとしている。

次に、インスタンス変数の値によって決まる状態を第一義にとると、オブジェクトの振る舞いを刺激応答モデルによってとらえることができる。このようなオブジェクトを「刺激応答オブジェクト」と呼ぶ。刺激応答オブジェクトの仕様は、"外部からの刺激"、"前状態のテスト"、"状態操作"、"外部への応答"の組みによって記述する。刺激応答モデルでは、「オブジェクトが外部からの刺激に対して、どのような状態のときにどのように振る舞うか」ということを状態と操作の組み合わせにより明確に記述するために、オブジェクトの内部仕様がより把握しやすくなる。図2では、ウィンドウオブジェクトやテキストバッファオブジェクトを刺激応答オブジェクトとしている。

3．3　オブジェクト間の関係仕様を表現する制約関係モデル

従来のオブジェクト指向モデルでは、システムのすべての構成要素をオブジェクトとしてモジュール化してしまうために、図3に示すような複数のオブジェクトの間の依存関係を自然な形で表現することが困難であるという問題点が指摘されている[5]。従来のオブジェクト指向モデルにおいては、このような関係を表現するのに、依存関係の処理をいずれか一方のオブジェクトに行わせるか、新しく依存関係管理オブジェクトを導入するかのいずれかの方法を用いる。しかし、前者では、依存関係管理処理をもつオブジェクトの仕様が特殊化されてしまい、部品としての汎用性を損なうことになる。一方、後者では、依存関係という問題領域には実体として現れない概念をもオブジェクトとして定義することになり、システムのモデルを自然な形で表現する上で好ましくない。

そこで、オブジェクト指向ＲＳＴモデルでは、「制約関係」の概念を導入してオブジェクト間の依存関係をあくまでも関係としてとらえ、オブジェクトとは別の枠組みで扱うことにより、このような問題を解決する。

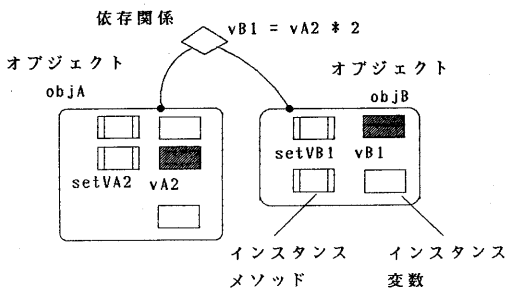

図3　オブジェクト間の依存関係

なお、オブジェクト指向ＲＳＴモデルに基づく仕様記述では、制約関係を課すために、変数を利用する。そこで、制約関係の定義方法には、制約関係を課す変数のタイプや制約関係の与えかたによっていくつかの種類を設定する。

（１）　基本データ型変数の制約関係

基本データ型の値を格納する変数に課された制約関係は、その変数に格納される"値"に対して意味をもつ。制約関係の内容は、図4に示すように「制約条件式」と「制約処理ステートメント」よって与える。制約条件式は、制約処理ステートメントを起動する条件を表すものであり、変数間に成り立つべき関係を論理式によって表明する「関係表明」による記述方法とある変数の値更新をチェックするためにその変数を指定する「値更新変数表明」による記述方法とを設ける。これらの制約関係は、変数の値が変化したときに評価され、起動される。

（２）　クラス型変数の制約関係

クラスから生成されたオブジェクトを指す変数に課された制約関係は、その変数によって指示されたオブジェクトが受け取る"メッセージ"に対して意味をもつ。制約関係の内容は、図5に示すように「レシーバ変数」、「レシーブメッセージ」、制約処理ステート

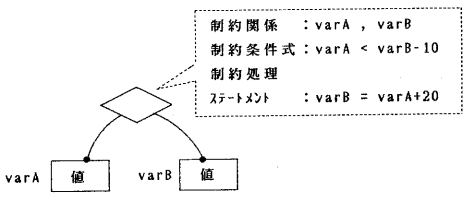

図4　基本データ型変数間の制約関係

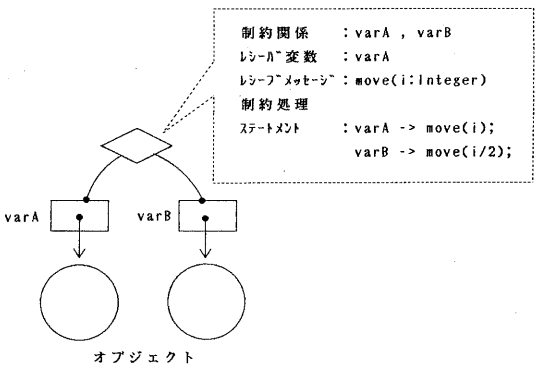

図5　クラス型変数間の制約関係

メントによって記述される。この制約関係は、レシーバ変数で指定した変数が指示しているところのオブジェクトが、レシーブメッセージで指定したメッセージを受け取ったときに、制約処理ステートメントが実行される、というものである。

（３） 静的制約関係と動的制約関係

制約関係の課し方には制約を課す変数の名前を直接用いてその変数に対する制約を静的に決めてしまう「静的制約関係」と制約関係への登録および解消を実行文中で陽に指定して行う「動的制約関係」とがある。

ところで、これらの制約関係が、複数の変数の間にいくつも定義されて、複雑に絡み合う場合には、その保持が難しくなるが、このような場合、オブジェクト指向ＲＳＴモデルでは、制約関係に対して、記述順に逐次的に評価することを約束している。

さて、図２に示したマルチウィンドウのモデル化においては、ウィンドウオブジェクトとテキストバッファオブジェクトおよびウィンドウオブジェクト間の関係を制約関係によって表現している。ウィンドウとテキストバッファの間には、テキストバッファの内容が常にウィンドウ内の表示に反映しているという関係があり、テキストバッファの内容が書き換えられたときには、それをウィンドウに送ってディスプレイに表示させるという制約関係を課している。また３つのウィンドウがディスプレイ上に重なりをもって表示されていることは、一つのウィンドウが変化したときに他のウィンドウも表示し直すという制約関係によって表現している。

３．４ オブジェクト指向ＲＳＴモデルの特長のまとめ

オブジェクトＲＳＴモデルによるマルチウィンドウエディタのモデル化では、それぞれのオブジェクトを状態遷移オブジェクトまたは刺激応答オブジェクトとして大きく分類することにより、対話型処理を中心とするエディタ全体の振る舞いがウィンドウ管理部と編集制御部のオブジェクトによって実現され、一方でテキストの操作や表示などの具体的な処理がウィンドウとテキストバッファのオブジェクトによって実現されることを明解に表現することができる。また、制約関係を用いた関係表現によって、オブジェクト間のリンクや各オブジェクトの内部仕様が単純かつ独立性の高いものになり、部品としての汎用性も向上する。例えば、テキストバッファオブジェクトは、その仕様が自分と関係するウィンドウオブジェクトに依存しなくなっている。

このように、オブジェクト指向ＲＳＴモデルでは、従来のオブジェクト指向モデルにおける平板な表現に対して、オブジェクトの働きかたに注目したオブジェクトの分類と、制約関係によるオブジェクト間の関係表現を用いたより自然なモデル化を可能にしている。

４．オブジェクト指向ＲＳＴモデルに基づく仕様記述言語ＯＲＳＴ／ＳＰＥＣ

４．１ ＯＲＳＴ／ＳＰＥＣの特徴

仕様記述言語ＯＲＳＴ／ＳＰＥＣは、第３章で述べたオブジェクト指向ＲＳＴモデルに基づいてプログラム仕様を記述するために設定した仕様記述言語である。ＯＲＳＴ／ＳＰＥＣでは、状態遷移オブジェクトと刺激応答オブジェクトの仕様および制約関係を記述するために［状態遷移クラス定義部］、［刺激応答クラス定義部］、［制約関係定義部］の３つの部分から構成される。また、配列や列のデータ構造に対して結合や分離などをはじめとする演算を多数用意し、エディタに特有な文字列操作を簡潔に記述できるようにしている。

４．２ 基本的な言語要素

（１） 型、変数

型には大きく別けて整数型や文字型などの基本データ型、配列型、列型などの構造型、状態遷移クラス型などのクラス型の３種類がある。

変数は、基本型の場合には、データの値を格納し、クラス型の場合には、オブジェクトを指し示すものである。したがって基本型変数の代入は値の代入となり、クラス型変数の代入は指示するオブジェクトへの結合となる。

（２） データ、オブジェクトの操作

基本型に対しては、四則演算などの基本的な演算を用意する。また、配列型、列型に対する演算の例をいくつか示す。

［例］ 配列型、列型の演算

```
a , b : sequence [] of Character;
a = "Taro";
b = "Your name";
b = b | {' '} | "is " | a | {'.'};
                    //  = "Your name is Taro."
length(c);          //  = 18
```

クラス型のインスタンスオブジェクトに対して操作を行うためには、オブジェクトに対してメッセージを送る。メッセージ送信は「オブジェクト名 -> セレクタ名」によって行う。また、インスタンスオブジェクトの生成は「new」によって行う。

このほか、特殊な演算として「存在演算」、「検索演算」、「反復演算」などを用意する。例えば、存在演算は以下のような演算である。

［存在演算］

```
exist ( i : [[ min .. max ]]
                        such that 論理式 )
```

min <= i <= max（i は整数型）に対して、論理式が真となるような i が存在するならば true を返し、なければ false を返す。

4．3　クラス定義部

状態遷移クラス定義部は、以下のような構成をもつ。

［状態遷移クラス定義部］

```
<< state transition class >>        クラス名
< super class >              スーパークラス名
< type >                          型定義部
< instance variable >          変数宣言部
< instance creation method >
            インスタンス生成メソッド定義部
< state transition method >
                   状態遷移メソッド定義部
<< end class >>
```

ここで、状態遷移メソッド定義部は状態遷移メソッド定義の並びからなり、各状態遷移メソッドの定義は以下のようなものである。

［状態遷移メソッド定義］

```
 state set: 状態集合名（引数宣言の並び）: 返値型
 start state: 開始状態名
 execute: 実行ステートメント
{result: 実行ステートメントの実行結果値
   newstate: 遷移先状態名
{instate: 状態名
 execute: 実行ステートメント
{result: 実行ステートメントの実行結果値
 newstate: 遷移先状態名                  } }
{end state: 終了状態名
 execute: 実行ステートメント}
```

※　{ } は、その組みが二つ以上あってよいことを示す。

一方、刺激応答クラス定義部は、状態遷移メソッド定義部のところを刺激応答メソッド定義部としたもので、そのメソッド定義は以下のようなものである。

［刺激応答メソッド定義］

```
 stimulus: 刺激名（引数宣言の並び）: 返値型
{prestate: 前状態テスト式
 update: 状態操作ステートメント
 response: 応答式                }
```

4．4　制約関係定義部

制約関係の定義は、「制約対象変数」の指定、制約条件表明、制約処理ステートメントからなる。例えば、動的制約関係定義部は以下のようなものである。

［制約関係定義部］

```
<< constraint >>                  制約関係名
< type >                          型定義部
< reference variable >      変数参照宣言部
< vairable >              関係情報変数宣言部
< entry method >          登録メソッド定義部
< constraint method >     制約メソッド定義部
```

動的制約関係への登録と解消はそれぞれ次のような式によって行う。

```
 entry 制約関係名（ 制約対象変数の並び ）
 cancel 制約関係名( 制約対象変数 , * , ... , * )
```

また、制約メソッドは次のように記述する。

［制約メソッド定義］

```
{where: 値関係表明式
 else: 制約処理ステートメント}
{ifupdate: 値更新変数
 then: 制約処理ステートメント}
{reciver: レシーバ変数
 recive message: レシーブメッセージ
 then: 制約処理ステートメント}
```

5．　マルチウィンドウエディタの仕様記述への適用

本節では、仕様記述言語ＯＲＳＴ／ＳＰＥＣを用いてマルチウィンドウエディタの仕様記述を行った例を示す。例題としては、3章でモデル化を行ったエディタを取り上げ、以下のような編集機能を設定して、その仕様記述を行ってみた。

［編集機能仕様］

（1）　テキスト編集モードでは、二次元の仮想テキストバッファに対して、一文字入力（挿入／上書）、一文字削除（デリート／バックスペース）上下左右へのカーソル移動、改行入力などが可能である。ウィンドウ内の表示は、ウィンドウの端でのカーソル移動に対して、上下左右にスクロールする。

（2）　各ウィンドウは、上一行がタイトル表示で、それ以外は編集用の領域となる。また、ウィンドウのフレームは表示しない。

（3）　編集モードからのウィンドウ操作は、位置の変更、大きさと位置の変更、クローズなどがある。

以下に、このうちの一組のウィンドウに当たる部分の仕様を記述した例として、図6から図8に編集制御部の仕様を定義する状態遷移クラス EditControl 、テキストバッファの仕様を定義する刺激応答クラス TextBuffer、テキストバッファとウィンドウの間の制約関係 EchoBack の仕様記述リストの一部を示す。

6．　仕様／プログラム変換処理系の概要

仕様／プログラム変換処理系は、「仕様解析部」、「プログラム生成部」、「部品管理部」の3つの部分から構成される。

入力された仕様に対して、まず仕様解析部で構文解析を行い各種の情報をテーブルに格納する。次に、入力された仕様に不足しているクラス定義などの情報を抽出し、部品ファイル管理部では、これらを部品ファイルから取り出してプログラム生成部に与える。ここ

```
<< state transition class >>
    EditControl
< instance variable >
    have  aTxtBuf : stimulus TextBuffer;    //  テキストバッファ
    have  aWindow : stimulus EditWindow;    //  ウィンドウ
    know  aDisplay : parts CharDisplay;     //  ディスプレイ
    know  aKeybord : parts Keybord;         //  キーボード
< instance creation method >
    init: EditControl(deL,dpC,szL,szC : Integer;
                      tytle : sequence [80] of Character;)
                      aDp : parts CharDisplay;
                      aKb : parts Keybord)
    update: //  オブジェクトの生成
            aTxtBuf = new TextBuffer(100);
            aWindow = new EditWindow(deL,deC,szL,szC,tytle);
            //  動的制約関係への登録
            entry EchoBack(aTxtBuf.edBff,aTxtBuf.bPtr,aWindow);
                ・・・・・・・・・・・・・・・・・
< state transition methods >
    //  状態集合  －  編集コマンドループ
    state set: editLoop()
               key : Character;
    //  開始状態  －  コマンド入力待ち
    start state: getCommd
    execute: key = aKeybord -> getChar();
    result: '¥[CSR_RIGHT]'
        newstate: cursorRight
    result: '¥[CSR_LEFT]'
        newstate: cursorLeft
      ・・・・・・・・・・・・・・・
    result: [[' '..'~']]
        newstate: putChar
      ・・・・・・・・・・・・・・・
    result: '¥[F1]'
        newstate: moveWindow
    result: '¥[F5]'
        newstate: editEnd
      ・・・・・・・・・・・・・・・
    //  状態  －  カーソル右
    instate: cursorRight
    execute: aTxtBuf -> toRightIdx();
    newstate: getCommd
      ・・・・・・・・・・・・・・・
    //  状態  －  一文字書き込み
    instate: putChar
    execute: aTxtBuf -> insChar(key);
    result: true
        newstate: getCommd
    result: false
        newstate: full
      ・・・・・・・・・・・・・・・
    //  状態  －  ウィンドウ移動
    instate: moveWindow
    execute: this -> loctIn();
    newstate: getCommd
    //  終了状態
    instate: editEnd
    execute: return
<< end class >>
```

図６　状態遷移クラス　EditControl の仕様記述例

で、部品には「仕様部品」と「プログラム部品」の二つの種類がある。仕様部品はＯＲＳＴ／ＳＰＥＣによってそれまでに記述された仕様を仕様部品ファイルに登録したもので、一方、プログラム部品は、筆者等がＣ＋＋を図式化して設計した図式言語ＤＩＡＬ／Ｃ＋＋によって作成されたプログラムをプログラム部品ファイルに登録したものである。

プログラム生成部では、仕様解析部で作成されたテーブルをもとに、Ｃ＋＋プログラムを構成し、また、部品ファイル管理部から与えられた部品を組み合わせて、最終的なプログラムを生成する。

ここでは、プログラム生成部における変換処理の一つとして、制約関係を保持する処理の実現方法を示す。

本処理系では、制約関係の保持を行うために制約関係の評価と制約処理を実行するための新たなオブジェ

```
<< stimulus response class >>
    TextBuffer
< type >
    EditChar : Character
    where [['¥[EOF]','¥[CR]',' ' .. '~']]; // 取り扱う文字タイプ
< instance variable >
    clmMax : Integer  where [[ 79 .. 119 ]]; // バッファ最大幅
    edBff : sequence [][] of EditChar;       // バッファ領域
    bPtr : struct{ lin : Integer;            // バッファポインタ
                   clm : Integer; };
    insMode : Enum { #INSERT , #OVER }; //  書き込みモードフラグ
< instance creation method >
        ・・・・・・・・・・・・・
< stimulus response method >
    //  刺激応答  －  編集ポインタを右へ
    stimulus: toRightPtr()
            //  カラム位置が右端でない
    prestate: bPtr.clm < lastIdx(edBff[bPtr.lin])
        update: bPtr.clm = bPtr.clm+1;
            //  カラム位置が右端でありかつ行位置が最下行でない
    prestate: (bPtr.clm == lastIdx(edBff[bPtr.lin]))
            && (bPtr.lin < lastIdx(edBff))
        update: bPtr.lin = bPtr.lin+1;
                bPtr.clm = 0;
        ・・・・・・・・・・・・・
    //  刺激応答  －  一文字挿入
    stimulus: insChar(c: EditChar): Boolean
            //  行の長さが上限に達していない
    prestate: lastIdx(edBff[bPtr.lin]) < clmMax
        update: edBff[bPtr.lin]
                = edBff[bPtr.lin][ .. bPtr.clm-1]
                    | {c}
                    | edBff[bPtr.lin][bPtr.clm .. ];
                this -> toRightPtr();
        response: true
            //  それ以外
    prestate: other
        response: false
        ・・・・・・・・・・・・・
<< end class >>
```

図７　刺激応答クラス　TextBuffer の仕様記述例

```
<< constraint >>
    EchoBack
< type >
    ptype : struct { lin : Integer;
                     clm : Integer; };
< reference variable >
    buff : sequence [][] of Character; // テキストバッファ領域
    bfPtr : ptype;                     // 編集ポインタ
    window : EditWindow;               // ウィンドウオブジェクト
< variable >
    wdToBfLin : Integer;    //  テキストバッファ領域上での
    wdToBfClm : Integer;    //  ウィンドウ位置
< entry method >
    entry: EchoBack(buff,bfPtr,window)
    update: wdToBfLin = 0;
            wdToBfClm = 0;
< constraint method >
    //  値要素  －  バッファ領域
    ifupdate: buff[i][j]
    then: window -> putToBuff(bfPtr.lin-wdToBfLin,bfPtr.clm
                              -wdToBfClm,buf[i][j]);
    //  値要素  －  編集ポインタ
    ifupdate: bfPtr
    then: window -> setCsor(bfPtr.lin-wdToBfLin,bfPtr.clm
                            -wdToBfClm);
    //  値関係  －  バッファポインタ＆ウィンドウ位置
    where: bfPtr.lin >= wdToBfLin
    else: wdToBfLin = wdToBfLin-2;
          window ->
              setBuffer(buff[wdToBfLin .. ][wdToBfClm .. ]);
            ・・・・・・・・・・・・・
<< end constraint >>
```

図８　制約関係　EchoBack の仕様記述例

クトを導入するという方法を用いる。例えば、基本データ型の変数に課された制約関係に対しては、制約を評価するための複合オブジェクトを導入し、これらのオブジェクトに変数の値の変更操作を仲介させる。一方、変数がクラス型の場合には、オブジェクトへのメッセージを仲介することになる。例えば図9に示すように、制約関係が課されている変数 varA は、制約関係を保持するために生成される変数オブジェクト ObjVarA のインスタンス変数とする。そして、varA への操作は、すべて ObjVarA へのメッセージに変換する。このメッセージに対するメソッドには、動的制約関係の保持を依頼する処理と、静的制約関係を保持する処理とをもたせ、varA への操作に対して制約関係を保持できるようにする。

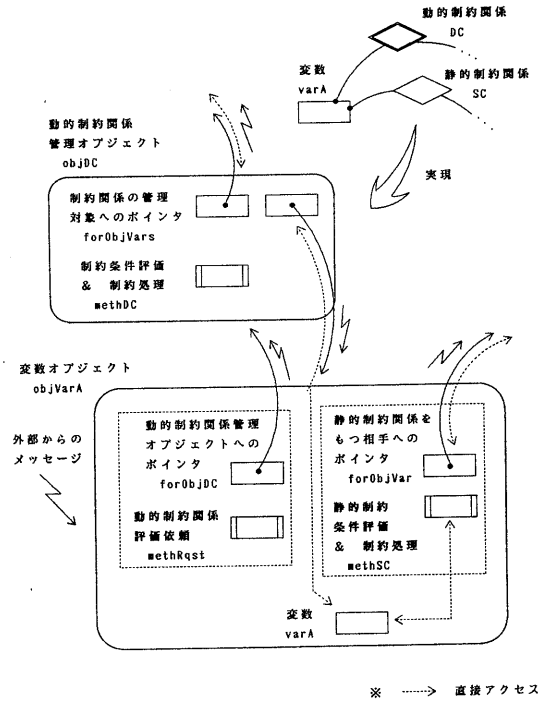

図9　制約関係を保持するためのオブジェクト構造

7．あとがき

本稿では、エディタのような対話型ユーザインタフェース処理部と内部のデータ処理部とを結び付けてモデル化することが必要となるシステムに対して、その操作的仕様を自然に表現する「オブジェクト指向ＲＳＴモデル」を提案しており、このモデルに基づく仕様記述言語ＯＲＳＴ／ＳＰＥＣを設定して、具体的にエディタの仕様記述例を提示した。最後に実行可能なオブジェクト指向プログラミング言語Ｃ＋＋への変換方法の概要を明らかにした。

本稿の仕様記述法をより実用的な複雑な対象にあてはめてみて、どれだけ適用できるのか、またどのような問題が生ずるのかについて今後更に考察を進める必要がある。

参考文献

(1) A.I.Wasserman : Extending State Transition Diagrams for the Specification of Human-Computer Interaction, IEEE Trans.on SE, Vol.SE-11, No.8, pp.699-713(1985).

(2) 大特集：オブジェクト指向プログラミング，情報処理，Vol.29，No.4(1988).

(3) 松本：オブジェクト指向型操作的仕様に関する一考察，情報処理学会論文誌，Vol.29，No.3，pp273-280(1988).

(4) Y.Wang : A Distributed Specification Model and Its Prototyping, IEEE Trans.on SE, Vol.SE-14, No.8, pp.1090-1097(1988)

(5) 中島：制約伝搬機構を内蔵するオブジェクト指向言語：COOL，情報処理学会論文誌，Vol.30，No.1，pp.101-108(1989).

(6) 鈴木（編）：オブジェクト指向，共立出版(1985).

(7) 橋本：データ中心のプログラム仕様記述法，井上書院(1988).

(8) 田中，谷口，奥井：スクリーンエディタの代数的仕様記述法とその実現，電子情報通信学会論文誌，Vol.J71-D，No.7，pp1207-1217(1988).

(9) 特集：エディタ，Vol.25，No.8(1984).

(10) 大特集：自動プログラミング，Vol.28，No.10(1987).

(11) B.Stroustrup(著)，斎藤(訳)：プログラミング言語 C++，トッパン(1988).

(12) 松本：ソフトウェア工学演習，朝倉書店(1984).

(13) 伊藤，本井田，内平：ソフトウェア開発のためのプロトタイピングツール，啓学出版(1987).

(14) 有澤：ソフトウェアプロトタイピング，近代科学社(1986).