ソフトウェア工学 117－6
（1997．11．21）

問題フレーム「業務支援システム」について

新田稔 nitta@rd.enicom.co.jp
新日鉄情報通信システム(株) 技術部

「業務支援システム」は、出来事の通知を受けて状態の変化をトラッキングし、また問い合わせに対して現状態を答えることによって、業務担当者に対して世界の現状把握を支援するソフトウェアである。このソフトウェアは、業務世界で自律的に発生する現象や業務担当者が能動的に行なう行為と、それらが引き起こす状態遷移による業務世界のモデルから生成される。特定の業務フローを前提とせず、また業務世界全体を俯瞰するモデルを作成しないという点で、従来の技法と大きく異なる。モデルの作成は、出来事と状態遷移の洗い出し、状態遷移間の関連付け、業務フローの設計、と段階的に進められる。

Requirement Specification of "Business Support System"

NITTA Minoru
Systems Technology Department,
NIPPON STEEL Information and Communication Systems Inc.

"Business Support System" is a software which supports users to grasp the current state of the world by tracking the state transitions and replying the queries from the user. This software can be generated from the business domain model which describes events and state transitions in the business world.

1. はじめに

筆者らは、大規模アプリケーションの開発(仕様変更の頻度が高い)やリバースエンジニアリング(「処理のための処理」が多く本質をつかみ難い)の経験から、要求モデルからソフトウェアを自動生成することを強く考えるようになり、その技法を模索中である。しかし、どのような問題にも適用でき、またどのような形態のソフトウェアも生成できるという技法はない。そのためにまず、問題フレーム[1]として「業務支援システム」(以下「本技法」と呼ぶ)を設定した。本論文では、その対象問題領域、モデルの考え方、モデル作成プロセス、ソフトウェア生成の概要などを紹介する。

2. 対象領域の捉えかた

2.1. 本技法の立場

一般に、業務アプリケーションの分析・設計を行なう際のモデル化の対象領域として、与えられた業務フローを前提して開発しようとするソフトウェアを取る立場(たとえば[2])と、人間系を含めた業務システム全体をとる立場(たとえば[3])がある。ビジネス環境の変化が激しい昨今では、業務フローは変わり得るものであり、また抜本的な業務フローの見直しこそが分析の目的である場合も少なくないことから、本技法は後者の立場を取っている。

ただし、前者が目的のソフトウェアに注目すればよいのに対し、後者では、業務世界の構造とそのシミュレータとしてのソフトウェアの構造を考えなければならない。そのため、業務世界全体を俯瞰する非常に大きなモデルを作成しなければならず、また、業務世界を記述していたはずがいつのまにかソフトウェア内部のことを書いていたという事態に陥る恐れもあった。

本技法は、このモデル作成上の困難を回避するため、業務世界の出来事と状態遷移をベースにモデルを作成し、そのモデルから目的ソフトウェアを生成するというアプローチを採っている。

出来事と状態遷移は、たとえば、「顧客から発注があった」という出来事とそのときの状態遷移が、業務フローが「日次でまとめて注文処理をする」から「その場で注文処理をする」に変わっても不変であるように、特定の業務フローには依存しない。

業務フローは、業務の目的を「世界をある状態にすること」と設定しておき、どのように状態遷移をつなげればその状態に達するかを考えることで、設計することができる。筆者らは、「ソフトウェア技術者は、与えられた業務フローをそのまま受け入れてソフトウェアを開発するのではなく、業務フロー自体の設計にも積極的に参加すべきである」と考えている[4]が、本技法はこれを推進するものである。

2.2. 世界の構造

本技法では、世界は複数の『要素』から構成されると考える。要素は、内部状態は持たないが、他の要素と関連することにより様々な『状態』を取り、その状態は『出来事』によって変化する。業務とは、世界を望ましい状態に近づけ、あるいは回避すべき状態から遠ざけ、またその状態を保つことを目的とした人間の活動である。

出来事には、世界で自律的に発生する『現象』と、業務担当者が能動的に引き起こす『行為』とがある。業務担当者は、世界の現状態を常に把握して、適切なタイミングで適切な行為を起こすことによって、その使命を果たすことができる。

業務担当者は、現時点の世界を見ただけではその状態を知ることはできないが、出来事の発生を検知し、それに基づいて実際に起きている変化を推測できるので、原理的には、その状態変化を記録していくことによって世界の現状態を知ることができる(出来事から状態遷移の推測が確率的にではなく決定的に行なえるものと仮定している)。

また、世界の変化を現象として検知する仕組みをどのように構築するかということも大きな課題ではあるが、特定業務業種の専門知識を要することであり、本技法の守備範囲外とする。

2.3. ソフトウェアの役割

業務担当者は、出来事から推測される状態変化の記録によって原理的には世界の現状態を知ることができるが、通常、業務世界は大きく複雑であり、手作業でこれを行なうのは非常に難しい。そこで、業務担当者を支援するソフトウェアが必要となる。

そのソフトウェアには、出来事からの状態遷移の推測方法がプログラムされる。業務担当者は、出来事の発生をソフトウェアに通知して状態遷移のトラッキングを行なわせ、また必要なときにソフトウェアに問い合わせることにより、いつでも世界の現状態を知ることができる(図1)。

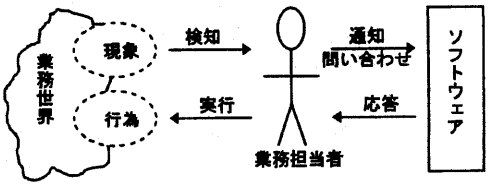

図1：ソフトウェアの役割

言い換えれば、ソフトウェアの役割は

(1) 業務担当者からの出来事の通知を受けて状態の変化を記録する

(2) 業務担当者からの問い合わせに対して世界の現状態を答える

である。また、すべての業務担当者が、世界の現状態を知ったからと言って次になすべき行為がわかるというわけではないので

(3) 業務担当者に対して行為のナビゲーションを行なう

もソフトウェアの役割になる。

出来事と状態遷移をトラッキングできれば、ソフトウェアの構造としては必要十分であるので、ソフトウェアの存在しない業務世界の記述から、それを導くことができる。本技法では、データ処理を目的ではなく手段であると考えるので、開発しようとするソフトウェアについての記述は行なわない。

2.4. 情報システムの構造

生成されるソフトウェアを含む情報システムの構造を図2に示す。業務担当者とソフトウェアとの間にリクエスト・ブローカー(RB)が存在する。RBは、複数の業務担当者からの出来事の通知や状態の問い合わせを受けてソフトウェアに伝える。ソフトウェアは、RBから通知を受けて応答をRB

に返す。RBは、システムからの応答を問い合わせ元の業務担当者に伝える。

また、担当者とRBとの間にはユーザ・インタフェース(UI)が存在する。UIの役割は、問い合わせや応答を担当者にとって入力・解読しやすい形(たとえば、メニューから項目を選択、売上高をグラフ表示、商品名一覧をアルファベット順にソートなど)に変換することである。

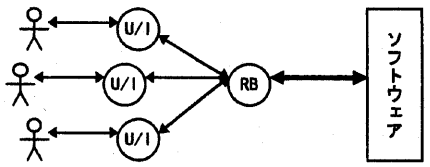

図2：業務支援システムの構造

本技法が目的とするのは、RBと情報をやり取りするソフトウェアであり、UIのデザインやRBの集配方式などは守備範囲外になる。

3. 出来事-状態遷移の記述

本技法における業務モデルの作成は、出来事(現象と行為)と状態遷移の洗い出しから始まる。これは、業務担当者に対する「どのような現象を検知するか」、「どのような行為を起こすことができるか」、「そのとき世界にはどんな変化が起きるか」という質問によって進めていくことができる。もちろんこれも容易な作業ではないが、業務世界全体を見渡したモデルを作成するよりは難易度はかなり低い。

なお、以下の説明中のモデルの表記法は、まだ試作中のものであり、作成プロセスを説明するためのものと考えていただきたい。

3.1. 出来事の記述

業務担当者が検知する現象と、行なうことができる行為を洗い出して

event ***出来事名*** { }
action ***出来事名*** { }

の形式で記述する。{}内には状態遷移を記述するが、この時点では空でよい。

出来事名は、現象で見ることができる要素や、行為の直接の対象となる要素を示すパラメタを含む。パラメタの形式は

[***要素の種別名*** : ***要素変数名***]

である要素の種別は、この段階では、自然な分類によるものと考えてよい。また要素変数は、個々のパラメタを識別するのに用いられる。たとえば

action [利用者:a]に[本:b]を貸し出す { }
event [利用者:a]から[本:b]が返却された { }

ここで、[利用者:a]と[本:b]がパラメタであり、この行為や現象が、「利用者」であるひとつの要素aと「本」であるひとつの要素bを対象にしていることを示している。

このように、現象の名称に過去形の動詞を、行為の名称に現在形の動詞を用いると、日本語として自然になる。最終的に記述内容の正当性を承認するのは、技術者側ではなく業務担当者側であるので、日本語として読める記述にすることの意義は大きい。

行為の洗い出しの際、世界の状態に関する情報を記録することは行為には含めない。出来事-状態遷移の記述によって、どのような情報を記録すべきかを分析するのであるから、そこに、ある情報を記述するという行為を入れてはならない。たとえば

[利用者:a]への[本:b]の貸し出しを記録する

という行為はありえない。ここが、データ処理を目的ではなく手段であると考えるモデリングの特徴である。

3.2. 状態遷移の記述

出来事の洗い出しに続いて、それらの出来事のによって引き起こされる状態遷移を記述する。本技法では「要素は、他の要素と関連することにより様々な状態を取る」と考えるので、ひとつの要素についての「状態Aから状態Bに移る」という遷移ではなく、世界全体についての「その出来事の後では、ある要素群はある状態にある/ない＝世界ではある状態が成立している/いない」という遷移を扱う。

状態には

パラメタ1 役割1 パラメタ2 役割2.. < 関連名 >

という形式の名称を付ける(役割はパラメタを順不同にするため)。パラメタ中の要素の種別名は、出来事名中のパラメタとの変数名の一致から推測できるので、省略してよい。たとえば

```
[a]が[b]を<借りている>
```

このように、役割に助詞を、関連名に状態を示す動詞を用いると、状態名は日本語として読める。

状態遷移の記述として、出来事の後に必ず成立している状態(事後成立状態)は

```
set 状態名;
```

と書く。成立していない状態(事後不成立状態)は状態名の前に not を付ける。たとえば

```
action [利用者:a]に[本:b]を貸し出す
  { set [a]が[b]を<借りている>; }
event [利用者:a]から[本:b]が返却された
  { set not [a]が[b]を<借りている>; }
```

3.3. 制御条件と制約条件

同じ出来事であっても出来事が起きる前の状態によって事後状態が異なる場合、その事前状態を制御条件と呼び、状態遷移を

```
if 制御条件
  { 条件が成立していたときの事後状態 }
else
  { 成立していなかったときの事後状態 }
```

の形式で記述する。たとえば

```
action [トグルスイッチ:s]を押す {
  if [s]は<ONである>
    { set not [s]は<ONである>; }
  else
    { set [s]は<ONである>; }
}
```

また、ある状態のときにだけその行為を起こすことができる、あるいはその現象が起き得るという場合、その状態を制約条件と呼び

```
constraint 制約条件 ;
```

の形式で記述する。たとえば

```
action [利用者:a]に[本:b]を貸し出す {
  constraint not [b]は<貸し出し禁止である>;
  set [a]が[b]を<借りている>;
}
```

制御条件や制約条件が複数の状態の論理和、論理積である場合には、状態を or または and で繋ぐ。状態の不成立が条件なら、状態の前に not を置く。

3.4. 不定複数の要素に対する出来事

出来事が不定個複数の要素を対象にする場合は、*で終わる要素の変数名を用いる。たとえば

```
action [利用者:a]に[本:B*]を貸し出す {
  constraint not [本:b]は<貸し出し禁止である>;
  set [a]が[b]を<借りている>;
}
```

は、この行為が、「利用者」の要素 a と、「本」の要素のセット B*を対象に行なわれることを示す。要素のセットに属する各要素の状態遷移は、各要素を示すパラメタを

```
foreach 各要素を示すパラメタ <= 元のパラメタ
  { 事後状態 }
```

の形式で設定して記述する。各要素を示すパラメタや元のパラメタ中の要素の種別名は省略してよい。たとえば

```
action [利用者:a]に[本:B*]を貸し出す {
  foreach [b] <= [B*]
    { set [利用者:a]が[本:b]を<借りている>; }
}
```

出来事が不定個複数の異なる種別の要素の組み合わせを対象にする場合は、パラメタを+で繋ぐ。たとえば

```
event [顧客:a]から
          [部品:B*]+[数量:C*]の注文があった {
  foreach [b]+[c] <= [B*]+[C*]
    { set [a]が[b]を[c]だけ<注文している>; }
}
```

3.5. パラメタに現れない要素

出来事が起きる前の状態によって事後状態に関与する要素が異なる場合、その要素は、出来事のパラメタには現れないが、出来事のパラメタになる要素となんらかの関連(導出関連)を持つ。

そのような要素が関与する状態遷移の記述は、導出関連を

```
use 状態名 ;
```

の形式で設定しておく。このとき、目的の要素は一個に特定されるとは限らないので、不定個複数の要素を示すパラメタを用いる。たとえば

```
event [本:b]が返却された {
  use [利用者:A*]が[b]を<借りている>;
  foreach [a] <= [A*]
    { set not [a]が[b]を<借りている>; }
}
```

4. 状態間の関連の記述

出来事-状態遷移の記述が概ね出来上がった時点で、事後状態の一覧を作成し、状態間の関連や要素の種別間の関連の記述を始める。従来、状態遷移をベースにしたモデルでは各状態遷移間の関係の表明が希薄になりがちであったが、本技法では、要素群についての状態遷移を記述するので、各要素群に共通に含まれる要素によって状態遷移間の関係をつけることができる。

4.1. 同名異意の訂正

出来事-状態遷移の記述において、同じ状態名が複数箇所に現れる場合、それらが本当に同じ状態を指すものであるのか検討する。もし、異なる状態に対して同じ名称が付けられていたときには、異なる名称に変更する。

また、要素の種別名についても、異なる種別に同じ名前が付けられていないかを検討し、もしあれば異なる名称に変更する。

4.2. 種別の関係

出来事-状態遷移の記述中のパラメタの種別名だけが異なる状態名については、1)同じ種別に異なる名称が付けられていないか、あるいは、2)ある種別(下位種別)の要素は必ず他の種別(上位種別)の要素であるという種別間の包含関係がないか、を検討する。

同じ種別に異なる名称が付けられている場合、異なる名称を用いることに特別な理由がないときには同じ名称に変更する。

種別の等価関係：異なる名称を用いることになんらかの理由があるときには

パラメタ1 == パラメタ2 ;

の形式で種別の等価関係を記述する。たとえば

[本:a] == [書籍:a];

種別の包含関係：種別間に包含関係がある場合には、下位種別の要素を識別する状態を、出来事-状態遷移の記述から見つけるか、あるいは新たな状態を設定して

下位種別のパラメタ == 状態名 ;

の形式で記述する。たとえば、下位種別「管理職」の要素を上位種別「従業員」の要素から識別する状態の名称が

[従業員:a]が[部署:d]の<部長である>

であれば

[管理職:a] == [従業員:a]が[部署]の<部長である>;

なお、状態の名称中、該当の要素以外の要素を指すパラメタには変数名を書かない。また右辺には、複数の状態名を and や or で繋いだ状態の論理積、論理和を書くこともできる。

種別の等価関係、包含関係の正当性は、上位種別の名称を下位種別の名称に置き換えても出来事-状態遷移の記述に矛盾が生じないことと、上位種別→下位種別の有向グラフにループがないことによって確認する。

4.3. 状態間の関連

出来事-状態遷移の記述において、1)同じ状態に異なる名称が付けられていないか、2)ある状態(主状態)が成立しているときには常に他の状態(従状態)も成立している/いないという状態間の包含関係がないか、また、3)ふたつの状態が同時に成立することはないという状態間の排他関係がないか、を検討する。

同じ状態に異なる名称が付けられている場合、異なる名称を用いることに特別な理由がないときには同じ名称に変更する。

状態の等価関係：異なる名称を用いることになんらかの理由があるときには

equivalent-for *パラメタ1+パラメタ2...*
{ *状態名1* ; *状態名2* ; ... }

の形式でそれら状態名を記述する。状態名1、2...は、すべて、パラメタ1、2...を含まなければならない。たとえば

```
equivalent-for [利用者:a]+[本:b] {
  [利用者:a]が[本:b]を<借りている>;
  [本:b]が[利用者:a]に<貸し出されている>;
}
```

状態の包含関係：状態間の包含関係は

従状態の名称 <== ***主状態の名称*** ;

の形式で記述する。右辺には、複数の状態名を and や or で繋いだ状態の論理積、論理和を書くこともできる。従状態の名称に含まれるパラメタは必ず右辺にも含まれなければならないが、右辺にだけ含まれるパラメタはあってもよく、またそのパラメタが右辺の 1 箇所にだけ現れるのなら変数名は書かない。たとえば

```
[製品:g]は<即日出荷できる> <==
    [倉庫:s]に[製品:g]の<在庫がある> and
        [倉庫:s]は<市内に立地している>;

[製品:g]は<注文引き受け可能である> <==
    [倉庫]に[製品:g]の<在庫がある> or
            [製品:g]は<自社製である>;

[本:b]は<貸し出し中である> <==
            [利用者]が[本:b]を<借りている>;

not [本:b]は<図書館にある> <==
    [利用者]が[本:b]を<借りている>;
```

従状態は、右辺に and がないなら、事後不成立状態であってもよいが、事後成立状態であってはならない(not が付いているときはその逆)。右辺に and があるなら、事後不成立状態であってもいけない。また、主状態→従状態の有向グラフにループがあってはならない。これに違反する場合は、なんらかの誤りがある。

状態の排他関係：要素(複数でもよい)の取り得る状態の間に排他関係がある場合には、それを

```
exclusive-for パラメタ 1+パラメタ 2...
  { 状態名 1 ;  状態名 2 ; ... }
```

の形式で記述する。状態名 1、2...は、すべて、パラメタ 1、2...を含まなければならない。たとえば

```
exclusive-for [スイッチ:a]
{ [a]は<ON である>; [a]は<OFF である>; }
```

なお、等価関係、包含関係、排他関係の中には、出来事-状態遷移の記述に現れる状態以外に、新たな状態を設定して記述してもよい。

4.4. ここまでの記述の完全さの検査

業務担当者による次の検査をパスしたとき、状態関連の記述は完成したと言える。

- ✓ 出来事がすべて列挙されている
- ✓ 出来事の状態遷移の記述は正しい
- ✓ 種別や状態の関係がすべて記述されている

また補助的に、ツールなどを作成して、次の検査を行なう。

- ✓ 出来事-状態遷移の記述において制約条件や制御条件にだけ現れる状態、および、状態間の関連の記述の際に新たに設定された状態は、事後状態として使われている状態から状態間の関係を用いて導くことができる
  →事後状態から導くことができない条件は、その条件が含んでいるパラメタのみに基づく演算または業務担当者の知識により成立の真偽が判断できるものであることを確認する。
- ✓ 出来事-状態遷移の記述において、事後成立状態と事後不成立状態はペアになっている(状態間の関係を用いて導いてもよい)
  →事後成立状態だけの場合は、その状態の要素が増えることが業務目的としてふさわしいことを確認する。

4.5. 例題：本の貸し出しの出来事と状態遷移

出来事-状態遷移の記述

```
event [利用者:a]が[本:b]を持って貸し出し窓口に来た
  { set [a]が[b]の<貸し出しを希望している>; }

action [利用者:a]に[本:b]を貸し出す {
  constraint [a]が[b]の<貸し出しを希望している>;
  use [日付:d]は<返却期限である>
  set [a]が[b]を[d]までに<返す予定で借りている>;
}

event [本:b]が返却された
  { [b]は<図書館にある>; }

action [利用者:a]に[本:b]の返却を催促する {
  constraint [利用者:a]に[本:b]の<返却を催促すべき>;
  set [a]が[b]の<返却を催促されている>;
}
```

状態間関連の記述

```
[利用者:a]が[本:b]を<借りている>
  <= [a]が[b]を[日付]までに<返す予定で借りている>;
```

```
[利用者:a]に[本:b]の<返却を催促すべき>
  <== [a]が[b]を[日付:d]までに<返す予定で借りている>
      and [d]は<一週間以上前である>;

exclusive-for [利用:a]+[本:b] {
  [a]が[b]の<貸し出しを希望している>;
  [a]が[b]を<借りている>;
}

not [本:b]は<図書館にある>
  <== [利用者]が[b]を<借りている>;

[利用者:a]が[本:b]を<借りている>
  <== [a]が[b]の<返却を催促されている>;
```

4.6. ソフトウェアの生成(1)

ここまでの記述から

(1) 現象や行為の通知を受けて状態の変化を記録する

(2) 行為の制約条件の問い合わせに対して、その真偽や、条件を満たす要素の組み合わせを答える

(3) 現象や行為の制約条件の違反に対して、エラーメッセージを出す

(4) 事後状態から判断できない制御条件や制約条件については、操作者に問い合わせるか、あるいは、あらかじめ用意された判断モジュールを呼び出す

というソフトウェアを生成することができる。

このソフトウェアは、業務フローには依存していないので、業務フローがどのように設定されても、そのまま利用することができる。

5. 業務フローの記述

制約条件が満たされているときに該当の行為を行なうことによって業務は進むが、効率的に業務目的を達成するため、あるいは、いつ何をなすべきかを完全には理解していない業務担当者のためには、「ある現象を検知したときにはある行為を起こす」や「ある行為に続いて他の行為を起こす」というノウハウ(=業務フロー)が必要である。

5.1. セッションの記述

すべての現象や行為をひとつの業務フローにするのではなく、いくつかのセッションに分ける。

各セッションは

```
session セッション名
  when このセッションのきっかけとなる現象の名称
{ 業務内容の記述 }
```

の形式で記述する。

業務フローの作成手順には、業務目標の状態にするにはどのように状態遷移をつなげればよいかを考えるトップダウン方式と、好ましくない状態を解消していくボトムアップ方式があるが、ここでは後者の方式(=ある状態をどの程度の期間容認できるかに注目して業務フローをセッションに分ける)を説明する。

5.2. 事後状態のリアルタイム的解消

まず、即刻解消されなければならない状態を事後状態としてもたらす現象について、その現象をきっかけに始まるセッションを設定し、業務内容として、その状態を解消する行為を

```
do 行為の名前 ;
```

と記述する。

さらに、その行為の事後状態も即刻解消されなければならない場合には、その状態を解消する行為を続けて記述する。事後状態によって状態を解消する行為が異なるのなら

```
if 条件1
{ do 条件1のときに行なうべき行為の名称 ; }
else if 条件2
{ do 条件2のときに行なうべき行為の名称 ; }
else ...
```

の形式で記述する。いくつかの状態を解消しなければならないのなら

```
foreach 解消すべき事後状態の名称
  { do 状態を解消する行為の名称 ; }
```

の形式で記述する。

また、事後状態が現象によって解消されるときには、その現象を

```
expect 現象の名称 ;
```

と記述する。

5.3. 事後状態のバッチ的解消

次に、ある限度期間(たとえば1日間、1週間、1

ヶ月間...)容認できる事後状態をもたらす現象や行為(行為はすでに他のセッションの業務内容に現れているもののみ)については、きっかけとなる現象として適当な記号(たとえば日次、週次、月次...)を用いてセッションを設定する。その先頭に、解消すべき事後状態を

```
use 状態の名称 ;
```

と記述し、その状態を解消する行為を記述する。さらにそれらの行為がもたらす事後状態の容認限度期間が現在記述しているセッションの期間と等しいか短いときには、その状態を解消する行為の記述を続ける。

解消しなくてよい事後状態をもたらす現象については、その現象がきっかけとなるセッションを設定する。業務内容の記述は空でよい。

すべての現象と行為が、いずれかのセッションに現れていれば、業務フローの記述は完成である。また、いずれのセッションにも現れない行為については、その必要性を再考する。

5.4. 例題：本の貸し出しの業務フロー

```
session 本の貸し出し
when [利用者:a]が[本:b]を持って貸し出し窓口に来た {
  do [利用者:a]に[本:b]を貸し出す;
}

session 本の返却の催促 when 週次 {
  use [利用者:A*]に[本:B*]の<返却を催促すべき>;
  foreach [a]+[b] <= [A*]+[B*] {
    do [a]に[b]の返却を催促する;
    expect [b]が返却された;
  }
}

session 本の返却受け取り
  when [本:b]が返却された { }
```

5.5. ソフトウェアの生成(2)

業務フローの記述から、行為のナビゲーションを行なうソフトウェアを生成することができる。具体的には、各セッションについて、きっかけとなる現象のパラメタを伴って起動されると、まずその状態遷移を記録するソフトウェア(前段階で生成されたもの)を呼び出す。

次に、業務内容に行為の指定がある場合には、その行為の実行を操作者に指示し、実行完了の確認入力を待って、状態遷移を記録するソフトウェア(前段階で生成されたもの)を呼び出す。

行為の選択や繰り返しは、その条件が事後状態から導くことができるものはソフトウェアが判断できる。そうでない条件については、操作者に問い合わせるか、あらかじめ用意された判断モジュールを呼び出す。

なお、上にも述べたように、本技法で生成されるソフトウェアは、RBと情報をやり取りするソフトウェアであるので、UIのデザインやRBの集配部分は別途に作成しなくてはならない。

6. おわりに

以上、業務世界のモデルからアプリケーション・ソフトウェアを生成するための問題フレーム+ソフトウェア・アーキテクチャのである「業務支援システム」について、対象問題領域、モデルの考え方、モデル作成プロセスなどを紹介した。

今後、少し規模の大きい例題を記述することにより、有効性・実用性を確認したいと考えている。皆様からコメントやご意見を頂ければ幸いである。

参考資料など

[1] Jackson, M.: Software Requirements & Specifications: a lexicon of practices, principles and prejudices, Addison-Wesley(1995) (玉井,酒匂訳:ソフトウェア博物誌 世界と機械の記述, トッパン(1997))

[2] Jacobson, I., et al.: Object-Oriented Software Engineering: A Use Case Driven Approach, Addison-Wesley(1992) (西岡他訳:オブジェクト指向ソフトウェア工学 OOSE, トッパン(1995))

[3] 岡部ら：ビジネス要素モデリング「MELON」, 電子情報通信学会 D-I Vol.J79-D-I No.10(1996)

[4] 新田稔：業務プロセスのフォーマルなモデリング "formaler"の概要, 情報処理学会 第114回ソフトウェア工学研究会(1997)