Ｈ２０年度成果のまとめ

JASA 設計ワーキンググループ

～ 目次 ～

1. 背景 **3**

2. 応用編 **3**

2.1 条件の記述 4

2.2 遷移した状態に応じた処理の記述 5

3. 適用例 **8**

3.1 通信系 8

3.1.1 TCP/IP に適用した場合 8

3.1.2 UDP-IP 通信手順に適用した場合 11

3.1.2.1 システム仕様 11

3.1.2.2 伝送仕様 11

3.1.2.3 クライアント仕様 15

3.1.2.4 サーバー仕様 16

3.1.2.5 遷移表の作成 17

3.1.2.6 演習問題の実装例 18

3.1.2.7 実装結果から判明した課題 22

3.2 機器系 25

3.2.1 DVD プレーヤ 25

3.2.2 リモコン信号解析処理 28

4. アンケート結果分析とＷＧ活動への参考情報 **31**

5. まとめ 35

1. 背景

状態遷移表を用いた設計手法は設計段階でモデルの検証が容易となるので、設計の上流段階で品質を確保することができ、ツールによるソースコード自動生成機能を使えば、メンテナンス性も上がり、再利用が盛んになって開発効率向上に有効である。

昨年度はこの設計手法を組込みシステム設計の分野に更に広めるために、先ず基本編としてその基礎的な表記法や簡単な設計への適用例を示した。

今年度は実際に使うための設計手法とするべく、現実の様々なソフトウェアに適用していくために基本編では表記しきれなかった考え方や表記方法について、応用編としてまとめる作業に取り組んだ。

また、今年度も ET2008 でアンケートを実施し、現状の把握と開発・設計部門における分野別の傾向や、今後のＷＧ活動の参考にするための分析を行った。

2. 応用編

昨年度の基本編では図 2.1 の様に、動作欄に処理と遷移先のみを記述したが、これだけでは現実の複雑なシステムを記述するには十分ではない。

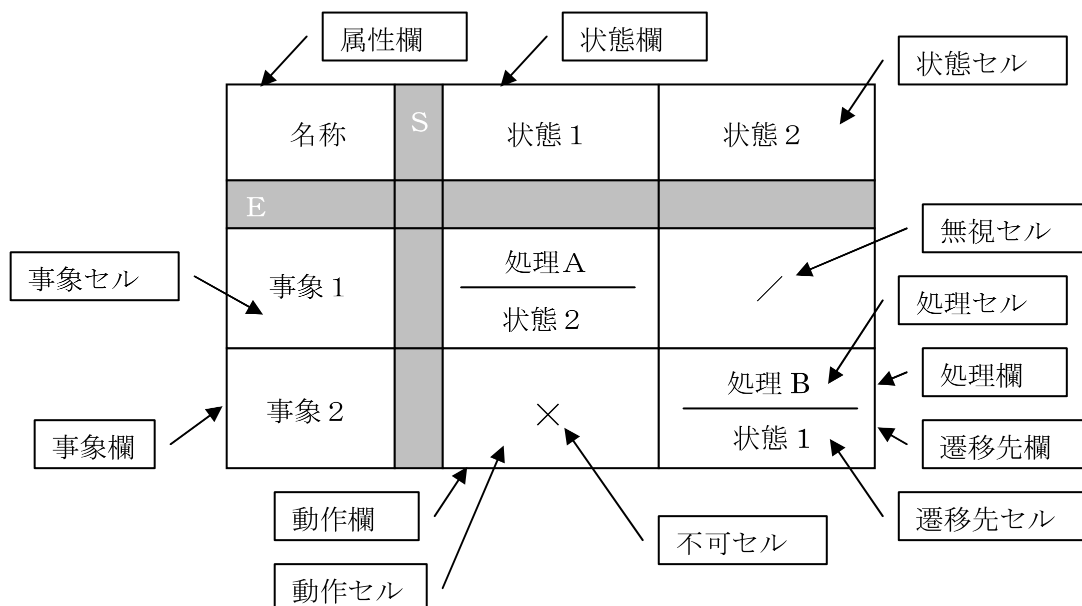

図 2.1 基本編の状態遷移表の表記と各部の名称

例えば、(1)条件によって遷移先が変わる物や、(2)遷移した状態に応じて処理を記述したい場合などである。それぞれ条件の記述、状態に応じた処理の記述として以降に説明する。

2.1 条件の記述

条件によって遷移先が変わる場合の表記。

例えば、イベントの回数をカウントして所定の回数になったら遷移するというものや、録画禁止番組のように外的要因で録画状態に遷移しないと言う場合の表記である。

処理と遷移先のみの表記だとカウント回数毎に多数の状態を記述しなければならなかったり、そもそも所定の回数が可変であるような場合は表を一意に作成出来ない。

そこで、条件欄を設け、表中に記載する形式を以下の様に表記する。

■条件の記述ルール

・動作セルを分割し、条件欄を設ける。

・分割は縦でも横でも構わない。

・処理と遷移先は必ず一体で表記する。

・遷移先が変わらないものは条件としない。

・処理が無い場合や遷移しない場合は痕跡を残した形で省略できる。

・ネストは一つ程度に抑え、多段にはしない。

・条件の数が増えるのは構わない。

図 2.1.1　条件の記述ルールに従った表記例

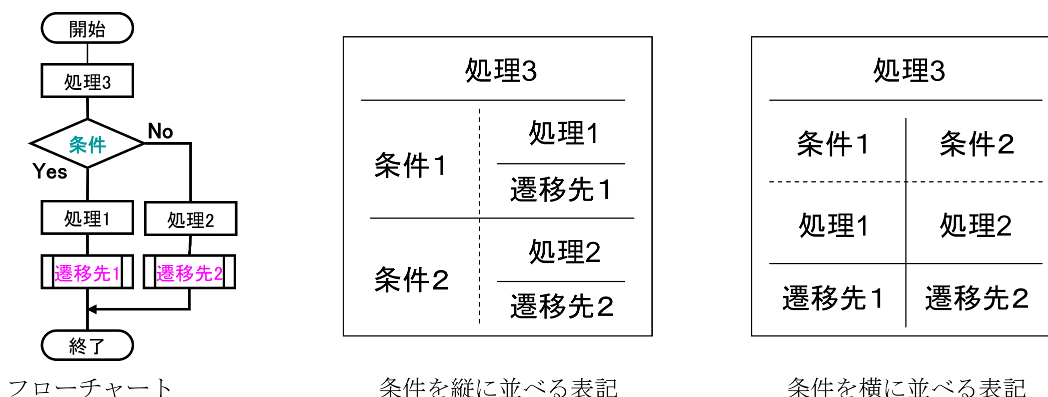

図 2.1.2　条件分岐の前に処理が入る場合の表記例

なお、上記ルールに、「遷移先が変わらないものは条件としない」があるが、遷移先は変わらないが、条件によって処理が変わる場合については、状態遷移という観点からここでは言及しないが、遷移先を同一とすることで、記述することは可能である。また、処理内容がその状態遷移表の他の遷移に影響が有る場合などは処理中に隠蔽するよりも、あえて表中に明記すべきである。

2.2 遷移した状態に応じた処理の記述

遷移した状態に応じた処理とは何か、そのことを説明する前に有限オートマトンのミーリマシンとムーアマシンについて言及しなければならない。

ミーリマシンとは現在の状態と入力によって出力が決定され、対してムーアマシンは入力に関係なく現在の状態のみによって出力が決定される。

これを論理回路と状態遷移図で表すと以下の様になる。

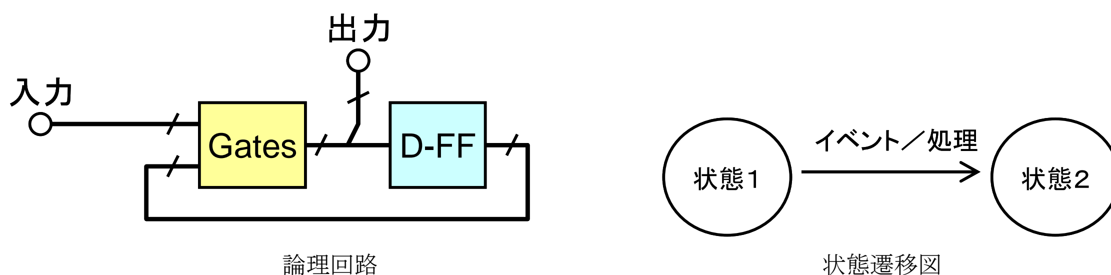

図 2.2.1　ミーリマシン

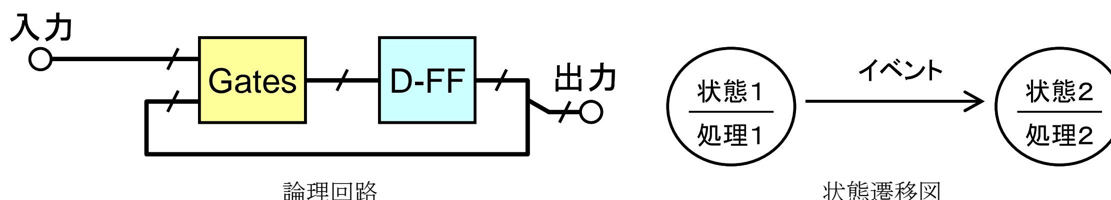

図 2.2.2　ムーアマシン

図 2.2.1 と図 2.2.2 を見て分かる通り、ミーリマシンでは、ある状態からイベントが来ると、処理を行ってから遷移するのに対し、ムーアマシンでは、遷移してから状態に応じて処理を行うという違いがある。このことから通信プロトコル系ではミーリマシンの考え方が、機器設計ではムーアマシンの考え方が経験上ちょうどマッチする。

このように根本的に設計思想の異なる状態遷移の考え方であっても、それが素直に表現できることが手法を普及させるためには必要な事であると考える。

状態遷移図の表記としては、図 2.2.1 のようにミーリマシンでは遷移線の上に処理を書き、ムーアマシンでは状態の中に処理を書くというように書き分けているので、状態遷移表でも同様の書き分けが必要であり、基本編で表記した状態遷移表ではイベントによって処理と遷移が行われ

ると表記するので、これはミーリマシンと言う事になる。

そこで、応用編ではムーアマシンの表記を定義する。図 2.2.3

Entry, Exit, Mode がそれぞれ、当該状態になった時に行う処理、当該状態から遷移する前に行う処理、当該状態にある間常に行う処理である。

なお、全ての状態で処理が無い項目は省略できる。(図 2.2.3 の Entry)

	S	状態1	状態2	状態3
E		0	1	2
Entry		／	／	／
Exit		／	処理B	処理B
Mode		／	／	処理A
イベント1	0	処理2 状態2	／	処理2 状態2
イベント2	1	／	処理1 状態1	処理1 状態1
イベント3	2	／	処理3 状態3	処理2 状態2

図 2.2.3　状態で行う処理を追加した状態遷移表

このように動作欄に処理を書いたのは、通常の処理と同様に表記出来る方が良いだろうと言う配慮からである。

ただ、通常の動作欄と紛れやすいことと、複数の状態で共通的に行う処理を記述出来ないため、標準形とするには、さらに検討が必要である。

また、ムーアマシンの例として、ＯＳを使わない機器設計の場合、図 2.2.4 のように、無限ループを一定周期で回し、周期的なポーリングを行う入力と、毎回ポートに信号を出す出力があり、出力用のデータは状態によって決定付けられるようにアーキテクチャ設計を行う。

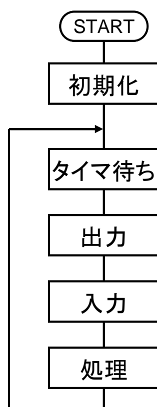

図 2.2.4　一定周期でポーリングする場合の基本構造

一方、状態遷移図や状態遷移系の処理をフローチャートで示すと図 2.2.5 のようになり、これは図 2.2.4 と同じ処理構造を持つことが分かる。

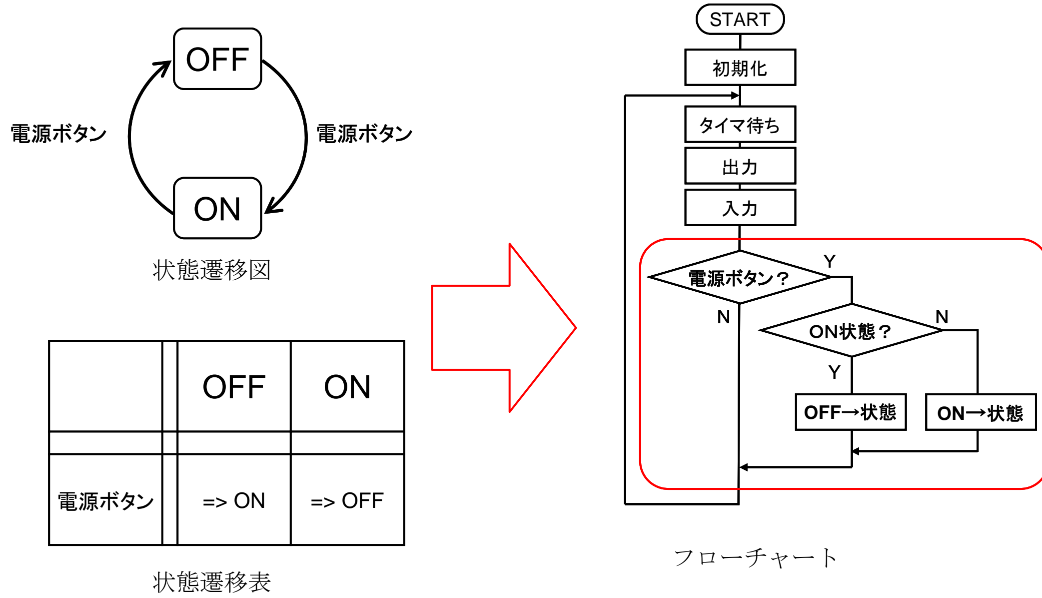

図 2.2.5　状態遷移とフローチャート

3. 適用例

以下にそれぞれの設計思想での適用例を示す。

3.1 通信系

以下 TCP/IP と UDP-IP の場合を示す。

3.1.1 TCP/IP に適用した場合

公開されている状態遷移図を状態遷移表に変換した例を示す。

【状態遷移図】

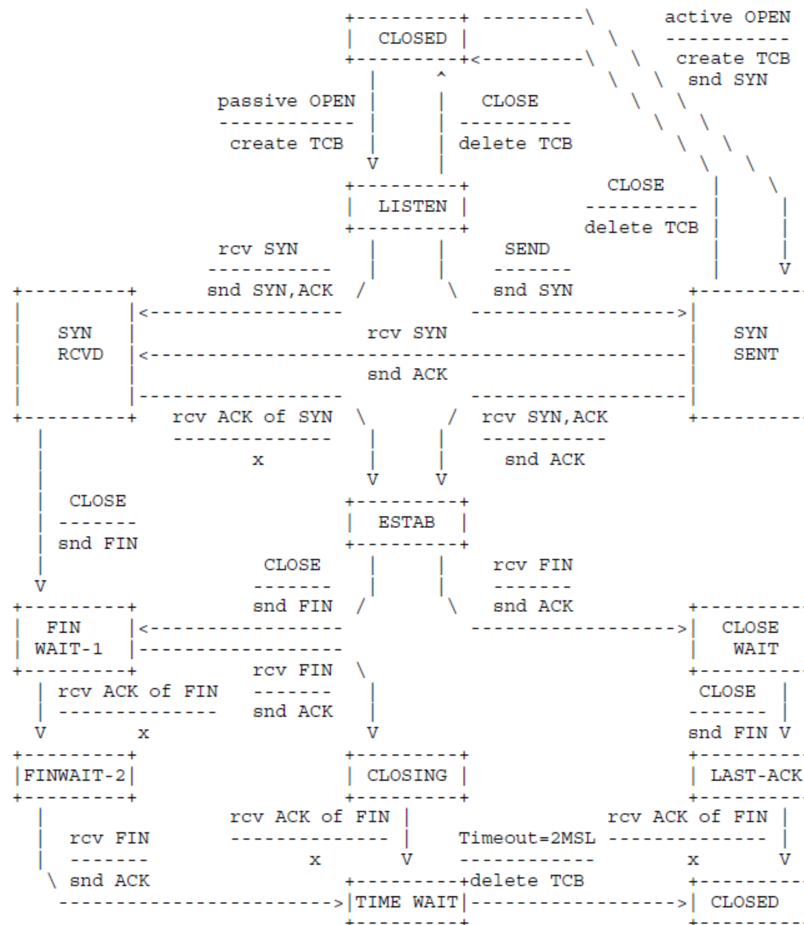

図 3.1.1.1　TCP の状態遷移図

（RFC793：TRANSMISSION CONTROL PROTOCOL より引用）

ただし、この状態遷移図は、RFC793 において、

NOTE BENE: this diagram is only a summary and must not be taken as the total specification.

と示されており、概要を示したものであり、TCP の仕様の全てではない。

【状態遷移表】

以下に、状態遷移図から変換した状態遷移表を示す。

E ＼ S		CLOSED 0	LISTEN 1	SYN_SENT 2	SYN_RCVD 3	ESTABLISHED 4	CLOSE_WAIT 5	LAST_ACK 6	FIN_WAIT_1 7	FIN_WAIT_2 8	CLOSING 9	TIME_WAIT 10
appl:passive_open	0	LISTEN										
appl:active_open	1	send:SYN SYN_SENT										
appl:send_data	2		send:SYN SYN_SENT									
appl:close	3		delete TCB CLOSED	delete TCB CLOSED	send:FIN FINWAIT1	send:FIN FIN_WAIT_1	send:FIN LAST_ACK					
timeout	4											CLOSED
recv:SYN	5		send:SYN,ACK SYN_RCVD	send:SYN,ACK SYN_RCVD								
recv:RST	6											
recv:ACK	7				ESTABLISHED			CLOSED	FIN_WAIT_2		delete TCB TIME_WAIT	
recv:FIN	8					send:ACK CLOSE_WAIT			send:ACK CLOSING	send:ACK TIME_WAIT		
recv:SYN,ACK	9			send:ACK ESTABLISHED								

図 3.1.1.2　TCP の状態遷移表

状態遷移図では正常ケースだけが記入されるため、状態遷移表に変換してみると例外ケースが記述されていないことが分かる。

パケットの受信イベント(５～９)に着目し、例外ケースの観点を記入する。
この例外ケースについては、先に示した状態遷移図には記述されてはいないため、仕様書の記述から記載する必要がある。
以下に、例外ケースの観点を加えた状態遷移表を示す。
（例外ケースは背景が黄になっている部分である）

E ＼ S		CLOSED 0	LISTEN 1	SYN_SENT 2	SYN_RCVD 3	ESTABLISHED 4	CLOSE_WAIT 5	LAST_ACK 6	FIN_WAIT_1 7	FIN_WAIT_2 8	CLOSING 9	TIME_WAIT 10
appl:passive_open	0	LISTEN										
appl:active_open	1	send:SYN SYN_SENT										
appl:send_data	2		send:SYN SYN_SENT									
appl:close	3		delete TCB CLOSED	delete TCB CLOSED	send:FIN FINWAIT1	send:FIN FIN_WAIT_1	send:FIN LAST_ACK					
timeout	4											／ CLOSED
recv:SYN	5	send:RST CLOSED	send:SYN,ACK SYN_RCVD	send:SYN,ACK SYN_RCVD	send:RST CLOSED	send:RST CLOSED	send:RST CLOSED	send:RST CLOSED	send:RST CLOSED	send:RST CLOSED	send:RST CLOSED	send:RST CLOSED
recv:RST	6	／	／	／		delete TCB CLOSED	delete TCB CLOSED	delete TCB CLOSED	delete TCB CLOSED	delete TCB CLOSED	delete TCB CLOSED	delete TCB CLOSED
recv:ACK	7	send:RST, ACK CLOSED	send:RST ／		ESTABLISHED			CLOSED	FIN_WAIT_2		delete TCB TIME_WAIT	
recv:FIN	8	／	／	／	CLOSE_WAIT	send:ACK CLOSE_WAIT	／	／	send:ACK CLOSING	send:ACK TIME_WAIT	／	／
recv:SYN,ACK	9	send:RST, ACK CLOSED	send:RST ／	send:ACK ESTABLISHED	send:RST CLOSED	send:RST CLOSED	send:RST CLOSED	send:RST CLOSED	send:RST CLOSED	send:RST CLOSED	send:RST CLOSED	send:RST CLOSED

図 3.1.1.3　例外ケースを加えた状態遷移表

次に、パケットの同一の受信イベントで、条件によって遷移先が変化するケースを記入する。
具体的には､SYN_SENTの状態でrecv:RSTのイベントがあった場合に､CLOSEDの状態でappl：passive_open または appl ： active_open のどちらかのイベントが発生したかによって処理と遷移先が異なる。
この条件付ケースについては、先に示した状態遷移図には記述されてはいないため、仕様書の記述から記載する必要がある。
以下に、条件付ケースの観点を加えた状態遷移表を示す。
（条件付ケースは背景が黄になっている部分である）

	S	CLOSED	LISTEN	SYN_SENT	SYN_RCVD	ESTABLISHED	CLOSE_WAIT	LAST_ACK	FIN_WAIT_1	FIN_WAIT_2	CLOSING
E		0	1	2	3	4	5	6	7	8	9
appl:passive_open	0	LISTEN									
appl:active_open	1	send:SYN SYN_SENT									
appl:send_data	2		send:SYN SYN_SENT								
appl:close	3		delete TCB CLOSED	delete TCB CLOSED	send:FIN FINWAIT1	send:FIN FIN_WAIT_1	send:FIN LAST_ACK				
timeout	4										
recv:SYN	5	send:RST CLOSED	send:SYN,ACK SYN_RCVD	send:SYN,ACK SYN_RCVD	send:RST CLOSED	send:RST CLOSED	send:RST CLOSED	send:RST CLOSED	send:RST CLOSED	send:RST CLOSED	send:RST CLOSED
recv:RST	6	/	/	/	active_open: delete TCB CLOSED passive_open: / LISTEN	delete TCB CLOSED	delete TCB CLOSED	delete TCB CLOSED	delete TCB CLOSED	delete TCB CLOSED	delete TCB CLOSED
recv:ACK	7	send:RST, ACK CLOSED	send:RST /		ESTABLISHED			CLOSED	FIN_WAIT_2		delete TCB TIME_WAIT
recv:FIN	8	/	/	/	CLOSE_WAIT	send:ACK CLOSE_WAIT	/	/	send:ACK CLOSING	send:ACK TIME_WAIT	/
recv:SYN,ACK	9	send:RST, ACK CLOSED	send:RST	send:ACK ESTABLISHED	send:RST CLOSED	send:RST CLOSED	send:RST CLOSED	send:RST CLOSED	send:RST CLOSED	send:RST CLOSED	send:RST CLOSED

図 3.1.1.4　条件付きケースを加えた状態遷移表

このように、まずは状態遷移図で正常ケースと代表的な例外ケースを記載して、全体の動作を設計する。その後、状態遷移図を基に状態遷移表を作成し、例外ケースをもれなく記載して、詳細設計を行うことで、それぞれの良い点を活かした品質の良い設計が可能となる。
このとき、状態遷移表で追加した部分を状態遷移図にフィードバックする必要はない。それは、図が複雑になってしまい、全体の動作を把握し易いと言う状態遷移図の良い点を損なってしまうからである。

3.1.2 UDP-IP 通信手順に適用した場合

以下に実際の仕様から状態遷移図および状態遷移表を作成する過程を示す。

3.1.2.1 システム仕様

UDP-IP 通信手順を用いて、クライアントからサーバーにファイルを転送することを想定する。

サーバーを立ち上げておき、任意のクライアントからサーバーにファイルを転送することを考える。UDP-IP 手順では、1 回に送れるデータ量は、パケットサイズからパケットの情報部を差し引いた量のみとなるので、パケットのデータ部サイズ以下のファイルを除いて、通常のファイルの転送では複数パケットを使って転送することになる。

3.1.2.2 伝送仕様

UDP-IP では、転送パケットが確実に転送先に届くことを保障しないし、コネクションの管理も UDP-IP 層では行なわない。

クライアントから UDP-IP 転送を行なう場合、必ず返送をもらい、返送が定義されたタイマー値以内で到着しない場合は、定義された回数分リトライを行う。なお、サイズ等は 32Bit 整数値とする（文字型データは除く）。

(1) パケットの基本構成

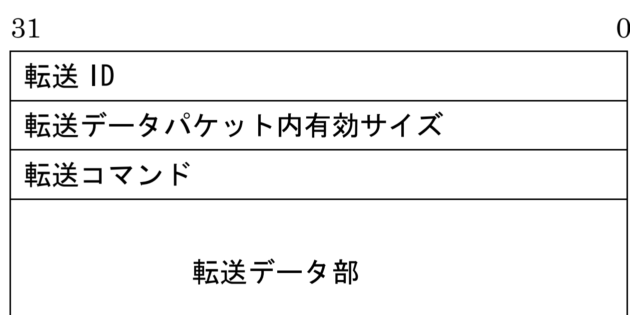

図 3.1.2.2.1　パケットの基本構成

転送 ID　：　サーバーがクライアントのセッションを管理するための ID。

クライアントが connection 要求を行なったときにサーバーから発行する数値。

次回のクライアントの要求時、この転送 ID を付けて送信する。

転送データパケット内有効サイズ：転送データ部のバイトサイズ。

転送コマンド：転送データの種類を表すための数値。

パケットデータの最大サイズは、1280 バイトとする。

(2) connection 要求

転送 ID = -1
転送データパケット内有効サイズ = X
転送コマンド = 1
クライアントのアドレス

図 3.1.2.2.2　connection 要求

転送データパケット内有効サイズは、クライアントのアドレスデータのバイトサイズとなる。

(3) サーバー応答

転送 ID = X （次回要求用の転送 ID）
転送データパケット内有効サイズ = 4
転送コマンド = 1000 (OK), 2000 (NG)
クライアントの要求転送コマンド

図 3.1.2.2.3　サーバー応答

(4) ファイル情報転送

転送 ID = X
転送データパケット内有効サイズ = Y
転送コマンド = 10
転送するファイルの総バイトサイズ
ファイル名（文字型）

図 3.1.2.2.4　ファイル情報転送

転送データパケット内有効サイズは、4 バイト＋ファイル名のサイズとする。

(5) ファイル本体転送

転送 ID = X
転送データパケット内有効サイズ = Y
転送コマンド = 11
転送ファイルの分割番号
転送ファイル本体バイナリのバイトサイズ
ファイル本体バイナリ（データ部）

図 3.1.2.2.5　ファイル本体転送

転送データパケット内有効サイズは、8 バイト＋転送ファイル本体バイナリのバイトサイズとする。転送ファイル本体バイナリのバイトサイズが「0」の場合、ファイル転送完了通知と見なす。転送ファイルの分割番号は、1 パケットサイズ以上のファイルを分割して転送する場合、それぞれのパケットに通番（1～）を付け、管理を行なうための番号である。

(6) disconnect 要求

転送 ID = X
転送データパケット内有効サイズ = 0
転送コマンド = 99

図 3.1.2.2.6　disconnect 要求

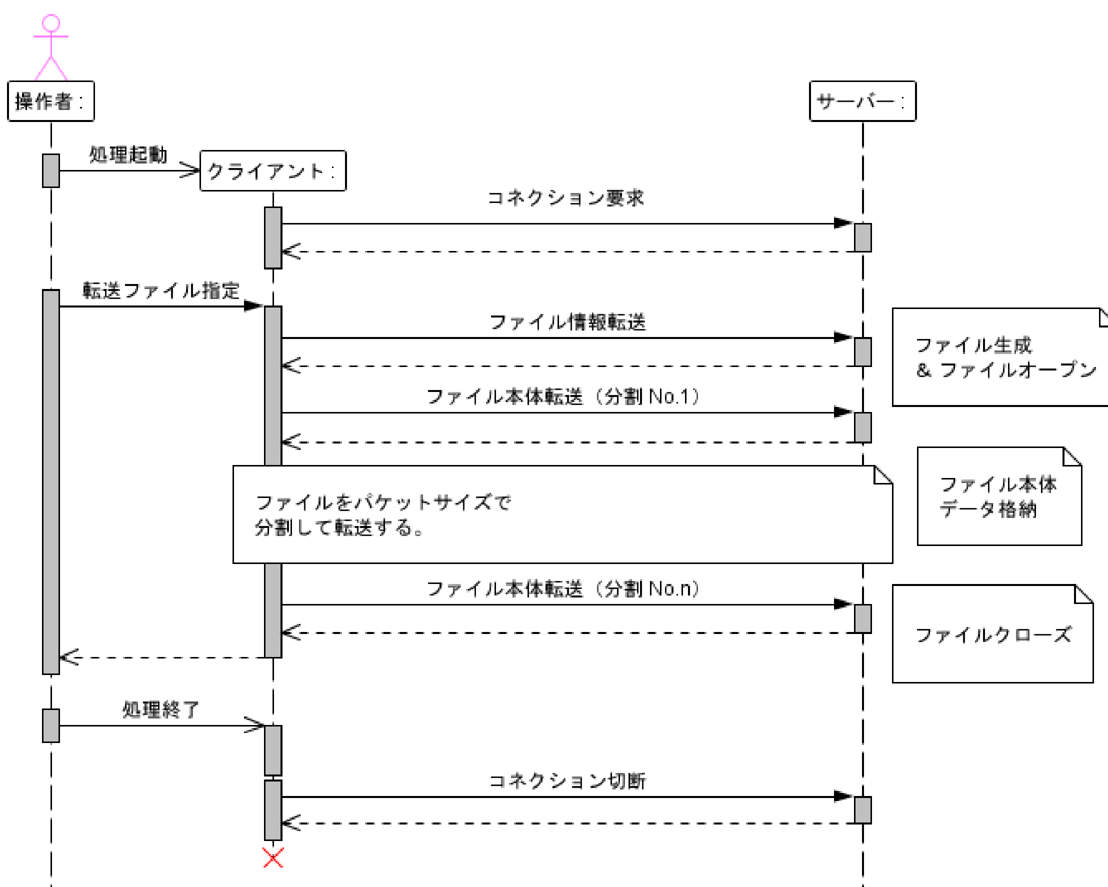

図 3.1.2.2.7　シーケンス図

本転送ツールでは、操作者がクライアントを起動するとサーバーにコネクション要求を掛け、サーバーとのセッションを確立する。次に、転送するファイル名を与えるとはじめにファイル名やファイルの転送サイズ等の情報をサーバーに送信し、その後、ファイル本体部の転送を開始する。転送対象となったファイルについて、UDP パケットデータ部サイズ以上のサイズがある場合、分割して転送する。

操作者が、クライアントに対し終了を入力するとサーバーとのコネクションを切断し、クライアントのプロセスは終了する。

ファイルを分割転送する場合は、ファイル情報の転送後、ファイル本体転送を複数回繰り返すことにより実施する。このときにサーバー応答が無い場合は、同一パケットを再度転送する（リトライを掛ける）。

3.1.2.3 クライアント仕様

サーバーに対しファイルを転送する手順を以下に示す。

(1) サーバーにファイル転送を知らせ、サーバーから転送 ID を取得する。
(2) 転送するファイル名とファイルサイズを転送 ID と共に送信する。
(3) ファイルを 1280 バイト単位で分割し、転送 ID と転送サイズを添えて転送を行なう。
(4) 複数ファイルを転送する場合、(2)から(3)を繰り返す。
(5) 転送作業が終了した場合、転送終了をサーバーに通知する。

遷移図で動作を表現すると以下となる。

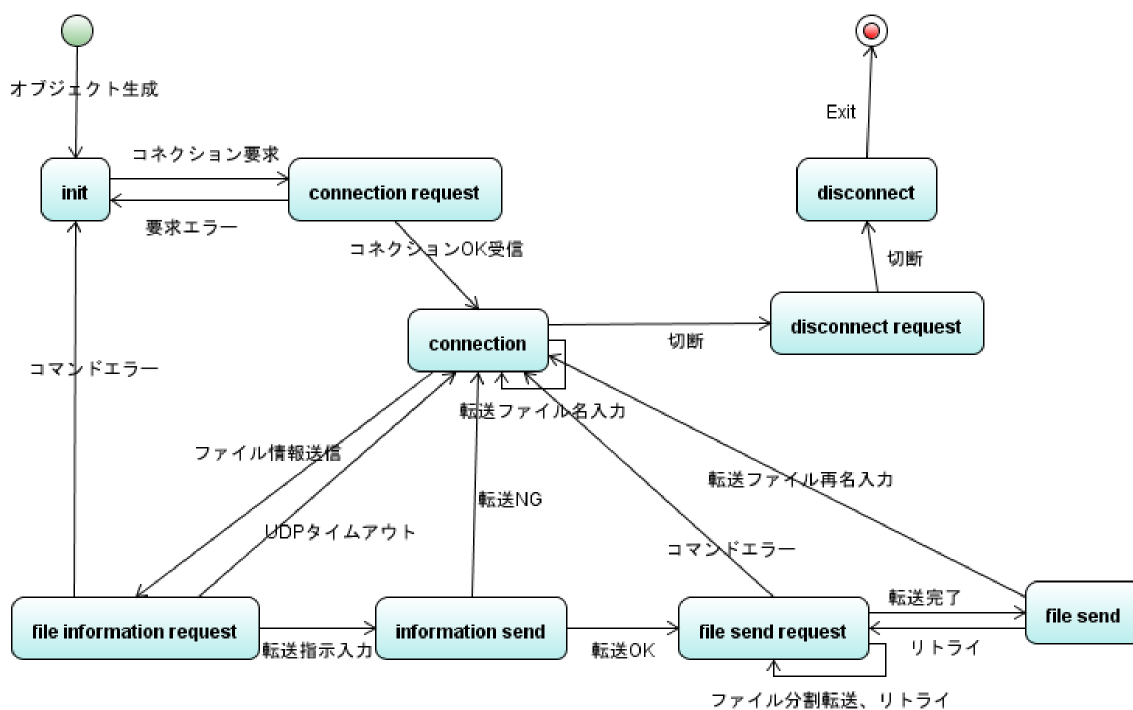

図 3.1.2.3.1　クライアントの状態遷移図

3.1.2.4 サーバー仕様

サーバーは、クライアントからの処理を行う受動的な動きとなる。

クライアントの要求に対して、処理を行ない、結果のパケットをクライアントに返す。

遷移図で動作を表すと以下となる。

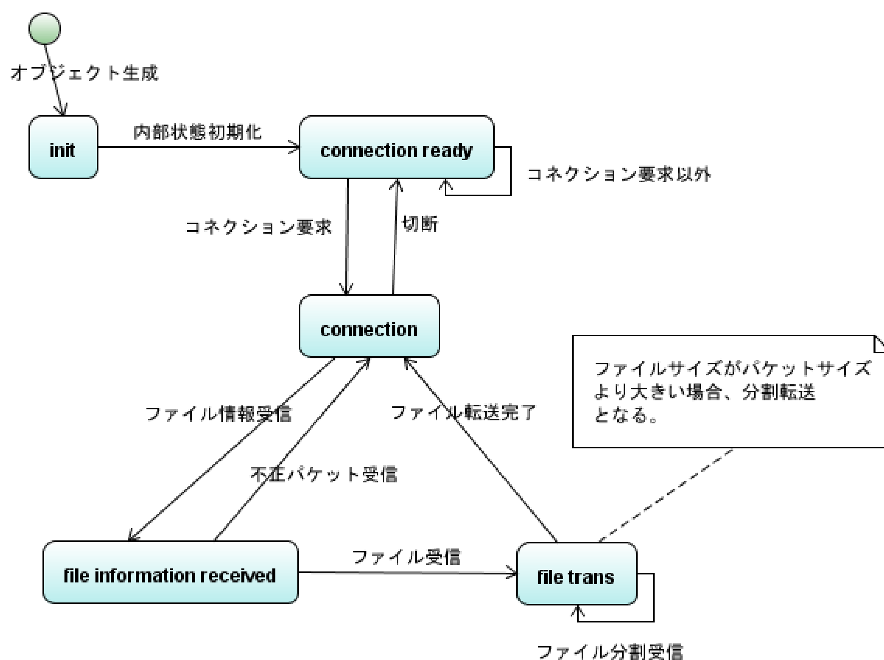

図 3.1.2.4.1　サーバーの状態遷移図

「connection」,「file information received」,「file trans」 の各状態は、クライアント毎に管理され、同時に複数の状態が存在することを許す仕組みを実装する必要があるが、複雑になり、本質的な議論から外れるので、本事例では、クライアントの同時コネクションは許さない仕様とした。

なお、同時に複数の状態をクライアント毎に管理するケースは、一般的に「Web サーバー」と呼ばれるソフトウェアでは必須となっているので、多重化した状態遷移については将来的な課題として残しておく。

3.1.2.5 遷移表の作成

遷移図を元に遷移表を作成する。

状態遷移表は、以下に示す要領で作成する。

	S	state1	state2
E		1	2
入力1	1	処理A 正常 | エラー 処理B | 処理C state2 | state1	
入力2	2		処理A3 正常 | エラー 処理B3 | 処理C3 state2 | state1

図 3.1.2.5.1　状態遷移表の例

状態遷移表の最上段は、遷移状態名を横並びに書く。表の左端は、処理に入力されるデータの種類を書く。

状態に対応して、入力のイベントが発生した場合、対応する処理を書き、遷移する遷移先の状態名を書く。

状態の一枡の基本は、以下に示すモジュール構成を取る。

共通処理の後に、遷移先に分かれる判定を置くパターンにしたのは、共通処理の中で、異なる遷移先に遷移する必要性がある場合があるためである。例えば、本演習の例となった遷移図上で、小さい菱形で表現している部分がそれに当たる。

3.1.2.6 演習問題の実装例

状態遷移の考え方を用いてロジックを実装して遷移図、遷移表との対応を考えてみる。

実装例では、以下に示すような構成を作成して、状態遷移の管理を行なう。なお、実装例の言語は Java である。また、実行環境は、Java 6 が動作する PC 環境を使う。

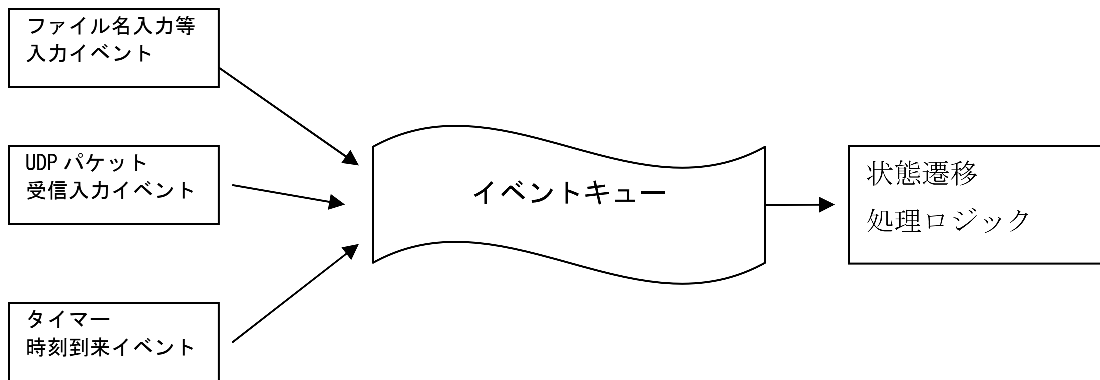

図 3.1.2.6.1　実装例の構成

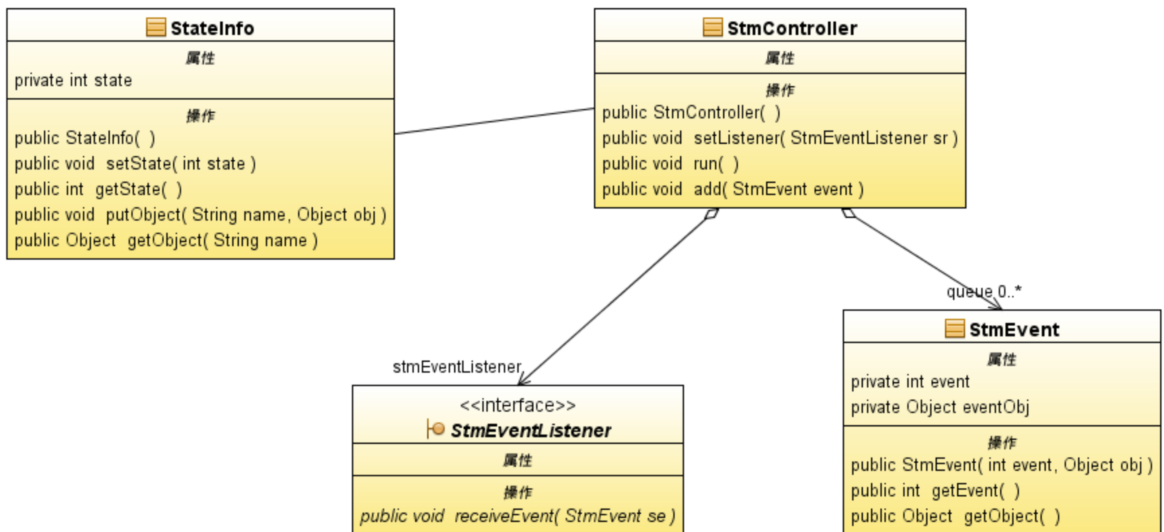

図 3.1.2.6.2 アクティビティ図

すべての状態遷移ロジックに入力されるイベントは、イベントキュー（FIFO）に入力され、順番に遷移表処理ロジックに注入される。実装例では、それぞれの入力イベントの処理と遷移表処理ロジックは別々の独立したスレッド上で動作させ、見かけ上、独立して動作するようにする。

「StmEventListener」インターフェースの「receiveEvent」メソッドは、イベントキューにイベントデータが入力された場合に呼ばれるメソッドである。クライアント、サーバーのそれぞれの処理は、このインターフェースをそれぞれの処理に応じて実装する。

メソッドのパラメータの「StmEvent」オブジェクトのパラメータはイベントの登録情報を保持しているオブジェクトである。イベントに対しては、イベントの種類に応じて番号を割付けてイベントを区別する。

状態遷移表のロジックは、まず、状態で処理分岐させ、さらにその下でイベントによる処理分岐があり、さらにイベント登録されたオブジェクトの内容で処理分岐させる構造になっている。

状態の設定は、それぞれの分岐の一番下位の部分で行うことになる。

実装した最終的な状態遷移処理ロジック（部分）を以下に示す。

```
 67     public void receiveEvent(StmEvent se) {
 68         try {
 69             int eventNo = se.getEvent();
 70             T.Trace.outln("-- State=" + stmEventState + ", event=" + eventNo);
 71             switch (stmEventState) {
 72                 case INIT:
 73                     if (eventNo == 0) {
 74                         appInitAction(se);
 75                     }
 76                     break;
 77                 case CONNECTION_READY:
 78                     if (eventNo == 0) {
 79                     } else if (eventNo == 1) {
 80                         UDPPacket pk = (UDPPacket) se.getObject();
 81                         InetAddress ia = pk.getFrom();
 82                         T.Trace.outln("receive packet:" + pk.toString());
 83                         int command = pk.getCommand();
 84                         if (command == 1) {
 85                             int sessionNo = getSessionNo();
 86                             StateInfo si = new StateInfo();
 87                             si.setState(sessionNo);
 88                             session.put(sessionNo, si);
 89                             sendMsg(ia, sessionNo, true, command);
 90                             stmEventState = StmState.CONNECTION;
 91                         } else {
 92                             int id = pk.getId();
 93                             sendMsg(ia, id, false, command);
 94                         }
 95                     }
 96                     break;
 97                 case CONNECTION:
 98                     if (eventNo == 0) {
 99                     } else if (eventNo == 1) {
100                         UDPPacket pk = (UDPPacket) se.getObject();
101                         InetAddress ia = pk.getFrom();
102                         int command = pk.getCommand();
103                         int id = pk.getId();
104                         if (command == 10) {
105                             StateInfo si = session.get(id);
106                             T.Trace.outln(pk.getBodyString() + ", " + pk.getIntBody(0));
107                             si.putObject("size", pk.getIntBody(0));
108                             si.putObject("filename", pk.getBodyString());
109                             oldPkNo = 0;
110                             sendMsg(ia, id, true, command);
111                             stmEventState = StmState.FILE_INFORMATION_RECEIVED;
112                         } else if (command == 99) {
113                             if (session.get(id) != null) {
114                                 session.remove(id);
115                             }
116                             sendMsg(ia, id, true, command);
117                             stmEventState = StmState.CONNECTION_READY;
118                         } else {
119                             sendMsg(ia, id, false, command);
120                             stmEventState = StmState.CONNECTION_READY;
121                         }
122                     }
123                     break;
124                 case FILE_INFORMATION_RECEIVED:
125                     if (eventNo == 0) {
126                     } else if (eventNo == 1) {
127                         UDPPacket pk = (UDPPacket) se.getObject();
128                         InetAddress ia = pk.getFrom();
129                         int command = pk.getCommand();
130                         int id = pk.getId();
131                         StateInfo si = session.get(id);
132
133                         if (command == 11) {
134                             BufferedOutputStream bs =
```

図 3.1.2.6.3　実装例のリスト

実装したロジックの構造を示すと以下になる。

```
switch(状態){
   イベント番号の取り出し
   case 状態 1 :  ←----- 1段目の分岐
                  if(イベント番号1){  ←----- 2段目の分岐
                       イベントオブジェクトの取り出し
                       If(条件 1){   ←----- 3段目の分岐、オブジェクトの内容で条件を作成
                            処理1
                            遷移先指定
                       }else if(条件 2){
                            ・・・・・・
                            遷移先指定
                       }
                  }else if(イベント番号2){
                       ・・・・・・・・・・・
                  }else{
                       処理 デフォルト
                       遷移先指定
                  }
                  break
   case 状態 2 :
                  ・・・・・・・・・・・・・
                  ・・・・・・・・・・・・・
                  ・・・・・・・・・・・・・
}
```

ロジック的には「状態」の分岐と「イベント」の分岐のレベルを入れ替えることも可能であるが、新しい状態を追加するケースを考えると上記の順番が妥当なように思える。

もし、「イベント」で case 分岐しているとすると、新しい状態を追加するときには、それぞれの case ブロックに状態に関する分岐を追加することになり、まとまりが良くない。

本実装事例では、遷移先の指定は、ロジックのネストの一番下位で設定される。例で示した遷移図では遷移先指定にまで至るロジックの表現は出来ない。無理やり表現しようとすると中間状態を導入して、遷移図を書くようにしなければならない。

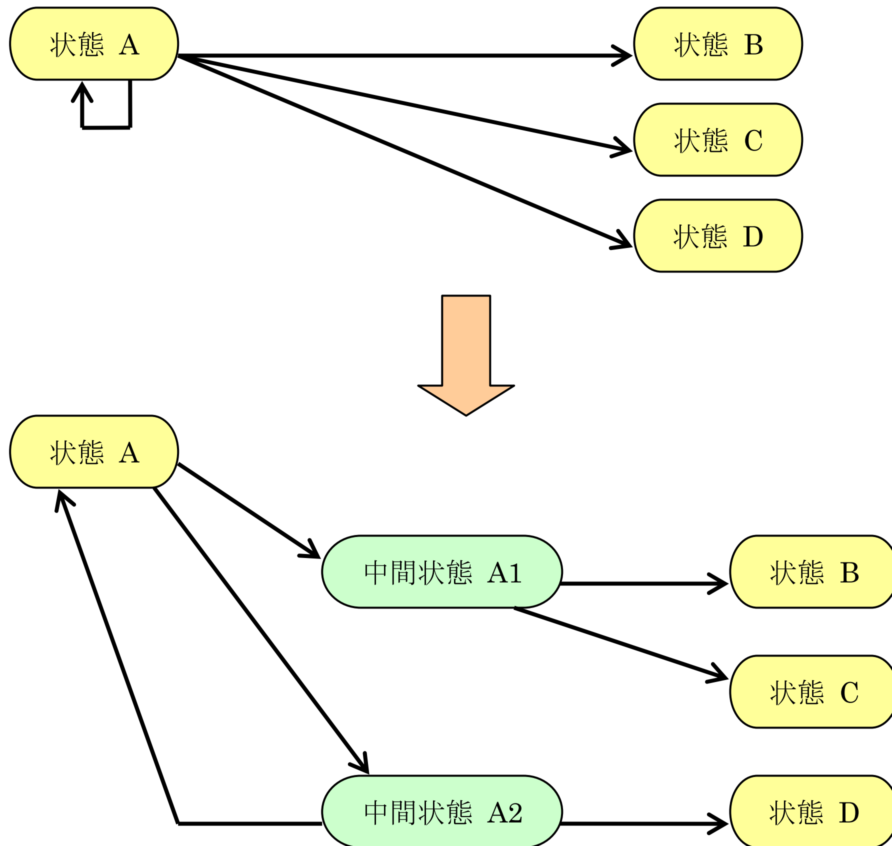

図 3.1.2.6.4　ロジックのネストの例

このように中間状態を導入することで、状態遷移図をよりロジックに近い表現に直すことが可能になる。また、中間状態からの遷移に対しては、仮想的な遷移手続きを取り、遷移ロジックを駆動する仕組みも必要になる。

より正確に、ロジックに対応するように遷移図を書こうとすると、状態数が増大し、図が複雑になり、図から全体を把握することが次第に困難になる。ここまでして、状態遷移図に忠実に表現する必要性があるのか？と言う疑問が出てくる。

一方、状態遷移表を使うと、中間状態を作らなくても表現が可能になり、不要な中間状態を定義しなくてもロジックと表の表現の一致を図ることが出来る。

3.1.2.7 実装結果から判明した課題

この例題の実装結果から、以下に示す課題もあることが分かった。

1. 処理自体の終了表現
   実装例のクライアント処理では、最後にクライアント処理自体が終了することになる。現在の状態遷移表ではこの処理自体の終了する場合の表現方法が無い。

社団法人 組込みシステム技術協会 Japan Embedded Systems Technology Association

2. 状態以外のデータ保持方法

状態遷移図でも状態遷移表でも、状態間に渡って状態以外のデータも保持しておく仕組みを用意する必要性がある。現在の表現方法では、このデータ変数を管理する仕組みがない。ロジックを組む際に使うデータの有効期間や有効範囲をなるべく限定することが望ましい。必要性がないのにグローバルで保持する変数にすべて入れるのは良くない。また、状態内で特定の状態に居るときのみ保持する必要のある変数については、他の状態からアクセスできないようにしておくことが望ましい。変数の有効範囲を限定させることはソフトウェアの品質を向上させる有効な手法である。Web アプリケーションのフレームワークの中にはこの様な仕組みがある。状態遷移図の記法でも、このような仕組みを導入したい。

3. 遷移表の内容記述の粒度

遷移表では、表内部に分岐ロジックを記述する必要が出てくるが、判定条件等を記述する際にどのくらいの粒度で記述するのが適切かと言うことが問題になる。

4. エラーの例外処理

Java、C++、PHP、JavaScript、perl 等の言語では、try{ ・・・・}catch ブロック内でエラーが発生した場合、Exception を投げて catch で指定されたブロックを実行するように書くことが出来る。この機能を使うとエラー発生後のエラー処理が簡単になるというメリットがあるが、現時点の遷移図も遷移表もこの機構を表現できない。

5. 共通ロジックの扱い

本事例のソースコードを見て分かるように、イベントで分岐したあとのロジックで同じソースコードが繰り返し見られる（80,81,83 行目のコード）。メンテナンス上は、繰り返し同じコードが出て来ることは望ましくない。しかし、これらのコードを switch ブロックの外に出すと共通に出来るが、反対に共通化ロジックの中にイベントの判定を入れる必要が出て来ることや、ワークでのみ使う変数の有効範囲が広がり、メンテナンス上、望ましくない面も出て来る。遷移表からロジックを作成する場合、ソースコードの作成効率を重視するか、共通処理を集約し、ソースコードのステップ数をなるべく少なくすることを重視するか、いろいろ課題がある。

これらのトレードオフは、単純に状態遷移表だけを使っていては、克服できないと思われる。DI(Dependecy Injection)と組み合わせるとか、遷移表と別の技術と組み合わせて克服する必要がある。

6. 状態遷移表のロジック部の増大

状態遷移表のロジック部は、持っている状態の数と入力データの種類の数が多くなると

大きくなる。状態遷移表は 2 次元の表であるので、ロジックのネストは少なくとも２つ以上のネストとなる。状態遷移表をそのままロジックにするとその部分のモジュールのソースコードステップ数が増大し、そのままではソースコードの可視性が低下してしまう。状態遷移表は整理されて見易いが、ソースコードに落とした場合、非常に煩雑になってしまい、ソースコードが見にくくなる。

3.2 機器系

以下機器系の例として DVD プレーヤとリモコン信号受信ドライバの適用例を示す。

3.2.1 DVD プレーヤ

ＤＶＤプレーヤのボタンと動作モードをそれぞれイベントと状態として基本編で記述すると図 3.2.1.1 のようになる。

DVDプレーヤ	S	停止	再生	一時停止	早送り	早戻し
E		0	1	2	3	4
再生	0	再生処理 / 再生	／	再生処理 / 再生	再生処理 / 再生	再生処理 / 再生
停止	1	／	停止処理 / 停止	停止処理 / 停止	停止処理 / 停止	停止処理 / 停止
一時停止	2	／	一時停止処理 / 一時停止	再生処理 / 再生	一時停止処理 / 一時停止	一時停止処理 / 一時停止
早送り	3	／	早送り処理 / 早送り	／	／	早送り処理 / 早送り
早戻し	4	／	早戻し処理 / 早戻し	／	早戻し処理 / 早戻し	／

図 3.2.1.1　ＤＶＤプレーヤの状態遷移表

しかし、機器設計者から見ると、例えば再生ボタンを押して再生モードになる場合、再生処理をしてから再生モードに遷移するのではなく、再生モードに遷移したので再生処理を行うのであるから、この表記法では実情を表していないと感じるのである。

さらに、各動作モードを細かく見ていくと、例えば再生モードでは、始めにファイルをオープンする処理（再生開始処理）があり、再生中はファイルからデータを読み出す処理（再生中処理）を行い、停止する場合はファイルをクローズする処理（再生終了処理）を行う訳であるが、これらの処理を追記しようとすると図 3.2.1.2 のようになる。

DVDプレーヤ	S	停止	再生	一時停止	早送り	早戻し
E		0	1	2	3	4
再生	0	停止終了処理 再生開始処理 / 再生	／	一時停止終了処理 再生開始処理 / 再生	早送り終了処理 再生開始処理 / 再生	早戻し終了処理 再生開始処理 / 再生
停止	1	／	再生終了処理 停止開始処理 / 停止	一時停止終了処理 停止開始処理 / 停止	早送り終了処理 停止開始処理 / 停止	早戻し終了処理 停止開始処理 / 停止
一時停止	2	／	再生終了処理 一時停止処理 / 一時停止	一時停止終了処理 再生開始処理 / 再生	早送り終了処理 一時停止開始処理 / 一時停止	早戻し終了処理 一時停止開始処理 / 一時停止
早送り	3	／	再生終了処理 早送り開始処理 / 早送り	／	／	早戻し終了処理 早送り開始処理 / 早送り
早戻し	4	／	再生終了処理 早戻し開始処理 / 早戻し	／	早送り終了処理 早戻し開始処理 / 早戻し	／

図 3.2.1.2　ＤＶＤプレーヤの状態遷移表に処理を追加

しかし、この表記では再生中処理のような、再生モード中に行う処理を記述出来ない上、それぞれの開始処理は遷移先に応じて記述し、終了処理は状態に応じて記述することになり、非常に煩雑でわかりにくい表になっている。

そこで今回応用編で追加した、状態で行う処理の表記を加えると、図 3.2.1.3 の様に再生中処理も記述出来る上に、開始処理や終了処理もすべて状態に応じて一カ所に記述すれば良いので、非常にすっきりと見易くなり、機器設計者が考えている実際の処理をそのままの形で表記することが出来る。この図では動作欄には遷移先のみとなるが、基本編で説明したように処理と遷移先を常に一対として扱うために何もしない処理欄も記述している。

DVDプレーヤ	S	停止	再生	一時停止	早送り	早戻し
E		0	1	2	3	4
Entry		停止開始処理	再生開始処理	一時停止開始処理	早送り開始処理	早戻し開始処理
Exit		停止終了処理	再生終了処理	一時停止終了処理	早送り終了処理	早送り終了処理
Mode		停止中処理	再生中処理	一時停止中処理	早送り中処理	早戻し中処理
再生	0	／ 再生	／	／ 再生	／ 再生	／ 再生
停止	1	／	／ 停止	／ 停止	／ 停止	／ 停止
一時停止	2	／	／ 一時停止	／ 再生	／ 一時停止	／ 一時停止
早送り	3	／	／ 早送り	／	／	／ 早送り
早戻し	4	／	／ 早戻し	／	／ 早戻し	／

図 3.2.1.3　ＤＶＤプレーヤの状態遷移表に状態で行う処理の表記を追加

さらに、早送りや早戻しが３段階ある場合、それぞれの段階をすべて状態とすると、下図のようになる。

DVDプレーヤ	S	停止	再生	一時停止	早送り1	早送り2	早送り3	早戻し1	早戻し2	早戻し3
E		0	1	2	3	4	5	6	7	8
Entry		停止開始処理	再生開始処理	一時停止開始処理	早送り1開始処理	早送り2処理	早送り3開始処理	早戻し1開始処理	早戻し2開始処理	早戻し3開始処理
Exit		停止終了処理	再生終了処理	一時停止終了処理	早送り1終了処理	早送り2終了処理	早送り3終了処理	早戻し1終了処理	早戻し2終了処理	早戻し3終了処理
Mode		停止中処理	再生中処理	一時停止中処理	早送り1中処理	早送り2中処理	早送り3中処理	早戻し1中処理	早戻し2中処理	早戻し3中処理
再生	0	／ 再生	／	／ 再生	／ 再生	／ 再生	／ 再生	／ 再生	／ 再生	／ 再生
停止	1	／	／ 停止	／ 停止	／ 停止	／ 停止	／ 停止	／ 停止	／ 停止	／ 停止
一時停止	2	／	／ 一時停止	再生	／ 一時停止	／ 一時停止	／ 一時停止	／ 一時停止	／ 一時停止	／ 一時停止
早送り	3	／	／ 早送り1	／	／ 早送り2	／ 早送り3	／	／ 早送り1	／ 早送り1	／ 早送り1
早戻し	4	／	／ 早戻し1	／	／ 早戻し1	／ 早戻し1	／ 早戻し1	／ 早戻し2	／ 早戻し3	／

図 3.2.1.4　ＤＶＤプレーヤの状態遷移表に早送り早戻し各３段階を追加

この場合も、機器設計者から見ると、例えば早送り１、早送り２、早送り３は段階が異なるだけで、まとめて早送りモードと考えるので、応用編の条件の記述を追加して下図の様に表を小さくして記述出来る。

つまり、いずれの段階でも早送りには違いがないのでこれを同じ状態として扱い、段階は条件とするのである。

ただ、この例では、条件によって遷移先が変わる訳ではないので、状態遷移表内に記述する必要は無い。しかい、遷移する場合とイベントが共通で有るため、遷移せずに条件によって処理が異なるとして、あえて記述している。

DVDプレーヤ	S	停止	再生	一時停止	早送り	早戻し
E		0	1	2	3	4
Entry		停止開始処理	再生開始処理	一時停止開始処理	段階1 早送り開始処理	段階1 早戻し開始処理
Exit		停止終了処理	再生終了処理	一時停止終了処理	早送り終了処理	早戻し終了処理
Mode		停止中処理	再生中処理	一時停止中処理	早送り中処理(段階)	早戻し中処理(段階)
再生	0	／ 再生	／	／ 再生	／ 再生	／ 再生
停止	1	／	／ 停止	／ 停止	／ 停止	／ 停止
一時停止	2	／	／ 一時停止	 再生	／ 一時停止	／ 一時停止
早送り	3	／	／ 早送り	／	3段階目 | 否 ／ | 段階加算 － | －	／ 早送り
早戻し	4	／	／ 早戻し	／	／ 早戻し	3段階目 | 否 ／ | 段階加算 － | －

図 3.2.1.5　ＤＶＤプレーヤの状態遷移表で、段階を条件として表記

このように、部分的な相違を条件とすることによって表を小さくまとめ、アルゴリズムを理解し易くする効果もある。

3.2.2 リモコン信号解析処理

赤外線リモコンの代表的な信号フォーマットは以下の様な物である。

これを受信解析する処理のアルゴリズムを状態遷移表で記述することを考える。

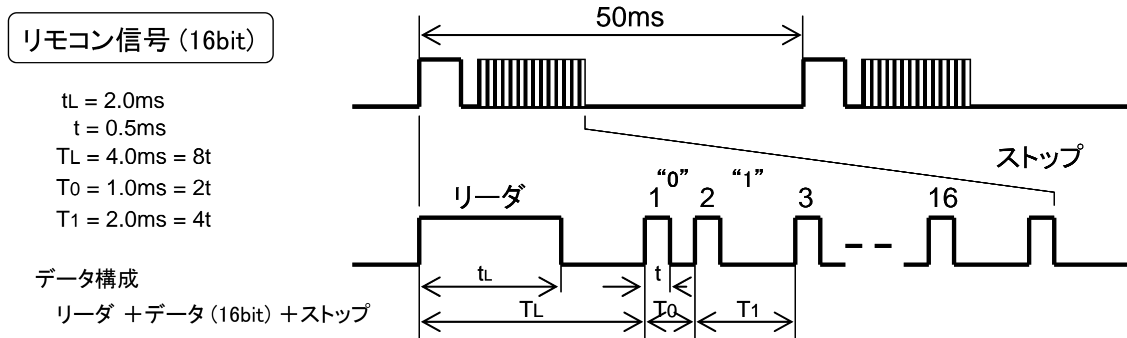

図 3.2.2.1　代表的な赤外線リモコン信号フォーマット

このようなリモコン信号の解析処理では、一見するとイベントと状態を抽出しづらく感じる。そこで、このような場合、まず概要となる状態遷移図を作成し、それを表にしていけばよい。

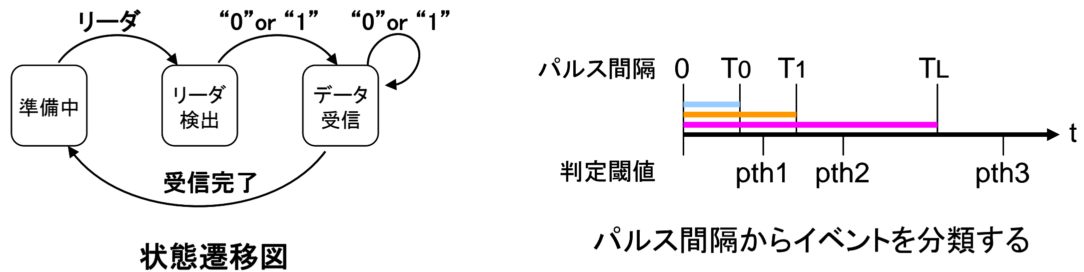

図 3.2.2.2　状態遷移図とイベント分類

この状態遷移図をもとに状態遷移表を作成すると下図の様になる。

リモコン受信	S	準備中	リーダ検出	データ受信中
E		1	2	3
パルス幅がリーダ	1	／ リーダ検出		
パルス幅がデータ1	2		1ビット目データ1受信処理 データ受信中	データ1受信処理 受信完了 ｜ 未完了 結果格納 ｜ ／ 準備中 ｜ －
パルス幅がデータ0	3		1ビット目データ0受信処理 データ受信中	データ0受信処理 受信完了 ｜ 未完了 結果格納 ｜ ／ 準備中 ｜ －

図 3.2.2.3　状態遷移図から作成した状態遷移表

さらに、この状態遷移表に例外ケースなどを追加していくと下図の様になる。

リモコン受信	S	準備中	リーダ検出	データ受信中
E		1	2	3
パルス幅がリーダより長い	1	／	／ 準備中	／ 準備中
パルス幅がリーダ	2	／ リーダ検出	／ リーダ検出	／ リーダ検出
パルス幅がデータ1	3	／	1ビット目データ1受信処理 データ受信中	データ1受信処理 受信完了 ｜ 未完了 結果格納 ｜ ／ 準備中 ｜ －
パルス幅がデータ0	4	／	1ビット目データ0受信処理 データ受信中	データ0受信処理 受信完了 ｜ 未完了 結果格納 ｜ ／ 準備中 ｜ －

図 3.2.2.4　詳細化した状態遷移表

条件に記入した受信完了はデータをカウントして１６ビットに達した時とするのである。

もし、このような条件の表記が出来ない場合、１ビット受信、２ビット受信、．．．１６ビット受信と、全てを状態にしなければならない。

この例のように、ビット数を条件とした方が表も小さくすることが出来、処理のアルゴリズムを理解し易くなる。

さらにこの状態遷移表をよく見ると、リーダ検出とデータ受信中は１ビット目かそれ以降かの違いだけで、処理内容自体は同じであることが分かる。従ってこの２つの状態を１つにまとめて下図の様に簡略化出来る。

リモコン受信	S	準備中	データ受信中
E		1	2
パルス幅がリーダより長い	1	／	／ 準備中
パルス幅がリーダ	2	受信初期化 データ受信中	受信初期化 データ受信中
パルス幅がデータ1	3	／	データ1受信処理 受信完了 ｜ 未完了 結果格納 ｜ ／ 準備中 ｜ －
パルス幅がデータ0	4	／	データ0受信処理 受信完了 ｜ 未完了 結果格納 ｜ ／ 準備中 ｜ －

図 3.2.2.5　簡略化した状態遷移表

最初に考えた解析アルゴリズムを状態遷移図にしたとき、思いつくまま解析の手順を順番に図にしていった訳であるが、もっと単純なアルゴリズムにすることが出来た。

もちろん、始めからこのようなアルゴリズムを思い付けば、この単純な状態遷移表を直接作る事が出来たかも知れない。しかし、どのような場合でもすぐに最適なアルゴリズムを思い付くことが出来るとは限らない。

上記のように思い付くまま状態遷移表を作っても、表の上で簡略化をすることによりアーキテクチャの最適化を行う事が出来る。このことは状態遷移表を使えば、特別なスキルを必要とせず最適化されたアルゴリズムを作成することができる事を意味する。

4. アンケート結果分析とWG活動への参考情報

今年度も昨年度に引き続いて ET2008 でアンケートを実施し、開発・設計部門における分野別の傾向や今後のWG活動の参考にするための分析を行った。今回のアンケートは以下の点を考慮して集計し、「2008 年度 設計手法標準化アンケート 集計結果」として纏め、JASA のホームページ（http://www.jasa.or.jp/top/activity/activityguid/technicalad.html#01）に掲載しているので参考にしていただきたい。

・アンケートの目的は、ソフトウェア設計フェーズで使っている設計手法を把握するためなので、「設計・開発部門」のデータだけを対象に分析することとした。

・本WGは、状態遷移表を使った設計手法の普及を目的として活動しているので、状態遷移表に関係する内容に着眼した。例えば、今後使ってみたい表記法として、UML ツールが一番となっているが、これに関する分析は行っていない。

・クラス図／コンポーネント図／ユースケース図は、１セットの UML ツールとして、重複分を除いて集計した。

・内製化ツールとその他ツールは内容が把握できないため、分析の対象外とした。

・母数が少ないデータは１件の違いで、比率が大きく変化してしまうため、有意なデータとはしなかった。

本書では、この集計結果に掲載しなかった部分について、WG活動の目的に沿うような分析を追加で行ったので、結果を以下に記載する。

（１）各業界で使われている代表的な設計手法について（図 4.1）

まず、業界毎に使われている設計手法に違いがあるのではないかと考え、各業界で使われている代表的な手法について分析してみた。その結果、以下のような傾向が分かった。

①フローチャートと状態遷移図はどの業界でも、ほぼ５０％以上使われている。

②フローチャート以外の別なツールが、より多く使われている業界としては、

家電、通信端末　：状態遷移表
家電、自動車関連：状態遷移図
家電　　　　　　：　シーケンス図

があげられる。

これは、通信端末がプロトコル（通信手順）を中心に設計するために、早くからコマンド（イベント）やステータス（状態）という概念を持って設計されてきたためではないかと考えられる。また、家電や自動車関連はフローチャートでは表現しにくい「状態」という概念を取り入れて設計する必要が出てきたためと考えられる。

③UML はコンピュータ・情報機器、家電、自動車関連の業界で比較的良く使われているが、通信端末の業界ではあまり使われていなかった。

これは、コンピュータや情報機器で構成されているシステム開発の場で UML というモデリング言語が誕生、発展し、普及段階になってきた時に、家電や自動車関連のソフトウェアが大規模化してきて、モデリングが必要となり、UML が採用されたためではないだろうか。一方、通信端末はシステムを構成する一要素であり、モデリングの必要性があまりなく、既に採用されていた設計手法（状態遷移図、状態遷移表）が引き続き、使われているためではないかと考えられる。

この分析から、状態遷移表を使った設計手法を普及させる候補の業界としては、状態遷移図は良く使われているが状態遷移表はあまり使われていない、自動車関連、ＦＡ機器、医療機器、があげられるのではないだろうか。

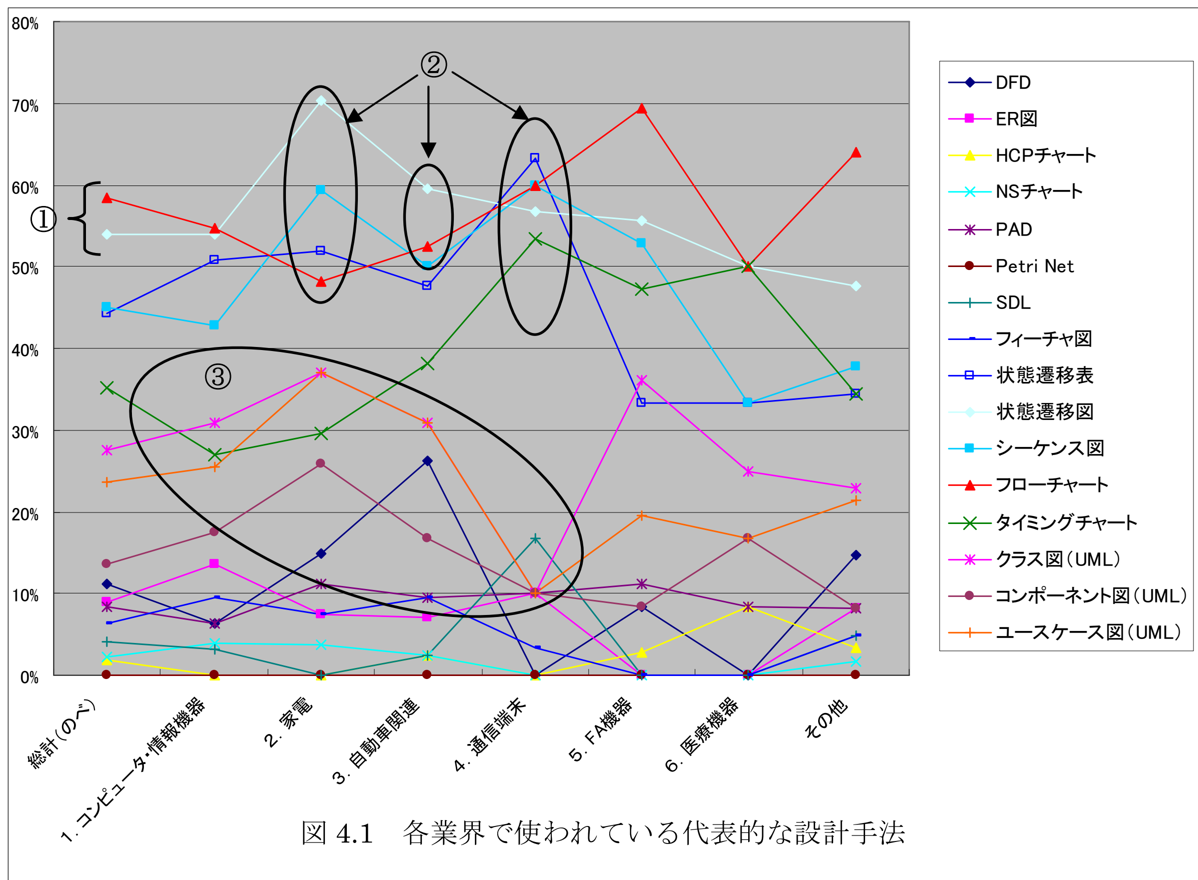

図 4.1　各業界で使われている代表的な設計手法

（２）各業界で使われている代表的なツール（図 4.2）

次に代表的な設計手法をどのようなツールを使って効率化、標準化を図っているかを知るために、各業界で使われている代表的なツールを分析してみた。その結果、以下のような傾向が分かった。

①総計では、ベスト３が MATLAB、ZIPC、Jude となっているが、業界によって使われている比率が異なり、MATLAB、ZIPC が多い業界では Jude が少なく、逆に Jude が多い業界では MATLAB、ZIPC が少ない傾向にある。特に家電業界と自動車関連では極端にその傾向が現れて

社団法人 組込みシステム技術協会 Japan Embedded Systems Technology Association

いる。

これは、自動車業界がモデルベース開発を中心に採用し、ツールベンダーがこの業界に力を入れて活動をしてきたのに対して、家電業界は UML を採用し、開発ツールしては基本機能を無償で使える Jude を使い出した結果ではないだろうか。

②次に、状態遷移表の設計手法を点線で表してみた。この点線で表した状態遷移表とその設計手法用ツールである ZIPC の関係を見てみると、ツール利用は手法採用の４分の１程度でしかなかった。

これは、ツールが販売される前からその設計手法を採用していたために、Ｅｘｃｅｌや内製等の別なツールを使っていて、新たなツールを採用するメリットが見出せないためではないかと考えられる。

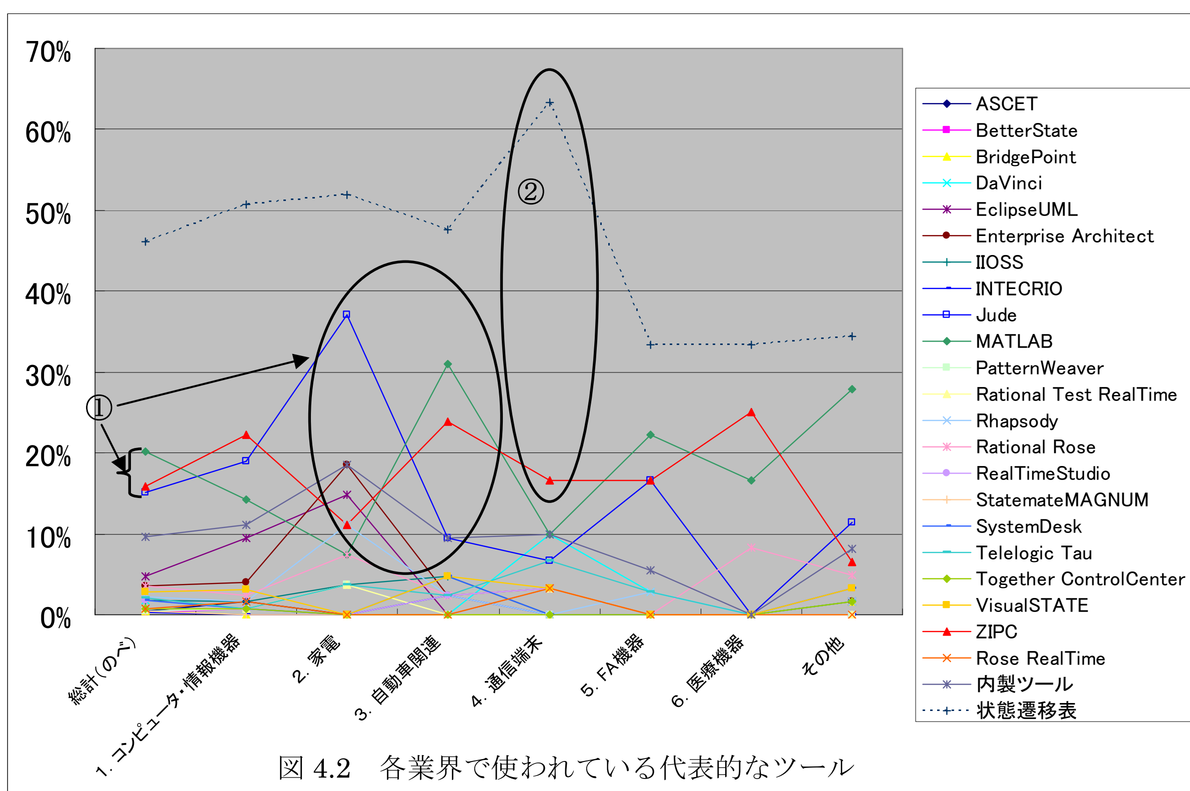

図 4.2　各業界で使われている代表的なツール

（３）効果があったツール（アンケート集計結果　図 6.7.2／表 6.7.3.2／表 6.7.4）

次に、どのようなツールで効果があったかを分析してみた。その結果、以下のような傾向が分かった。

①よく使われているツールベスト３の比率は、ZIPC：18%、Jude：16%、MATLAB：19%であったのに対して、効果があったツールは ZIPC：26%、Jude：15%、MATLAB：13%、と ZIPC が他のツールの２倍程度の効果を示していた。更に、Jude、MATLAB は１名ではあるが「効果が無かった」と指摘しているのに対して、ZIPC が「効果が無かった」と指摘している人は一人も

いなかった。

②効果の順番は、品質向上、作業の標準化、生産性の向上、となった。また、ZIPC が品質向上に効果があったと考えている人の 3 分の 2 が生産性向上にも効果があったと言っているのに対して、状態遷移表や MATLAB の表だけを採用している人達は品質向上には効果があったが生産性向上には効果が無かったと考えていることも分かった。これは、状態遷移表設計手法だけを採用しても、品質向上に役立つが、ツールが持っている便利な機能を使えないため、生産性向上への効果が出ていないことを伺わせている。

（4）今後の普及目標業界（図 4.3）

最後に、状態遷移図と状態遷移表を使っている人について、更に分析を深めてみた。この結果、以下のようなことが分かった。

①通信端末業界が唯一、状態遷移表が状態遷移図よりも多くなっており、両方使っている比率も一番多かった。これは、通信端末では状態遷移が明確になっており、且つ、昔から状態遷移表が多く使われていたからではないかと推測される。

②今後は、状態遷移図は使っているが、状態遷移表は使っていない、家電、自動車関連、などの業界に状態遷移表設計を普及させることが有効と考えられる。

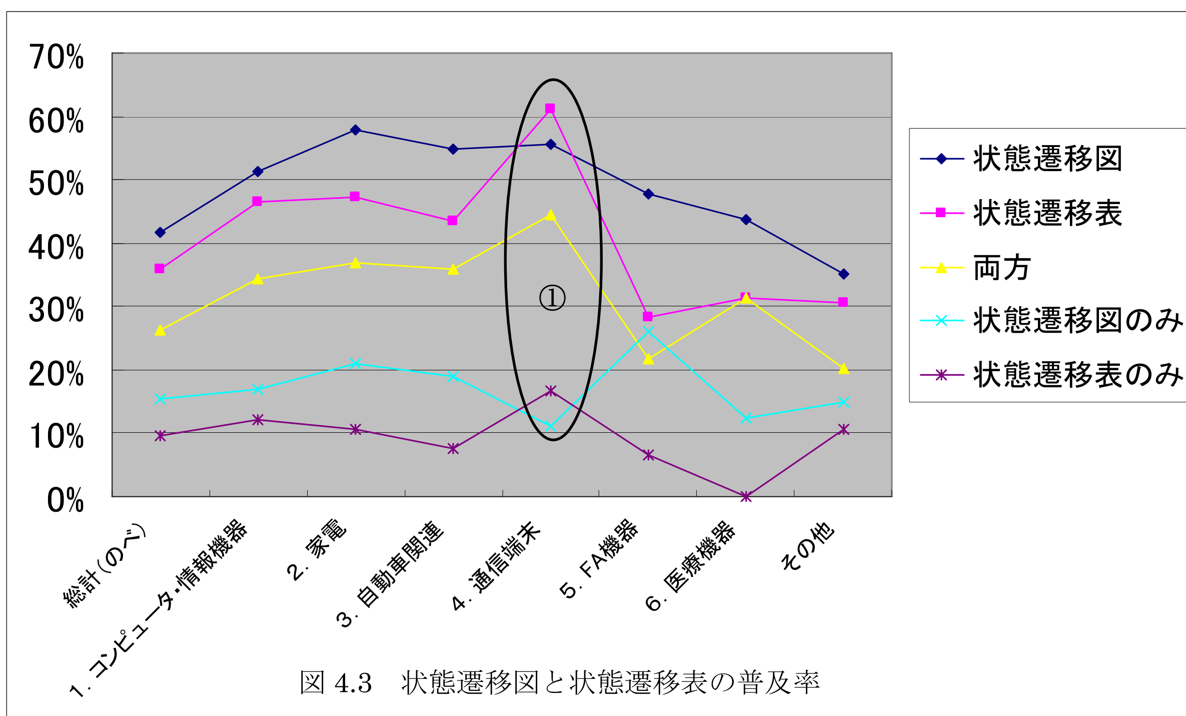

図 4.3　状態遷移図と状態遷移表の普及率

以上の事から、設計ＷＧで「状態遷移表設計が設計漏れを防ぎ、（上流設計での）品質向上に効果があり、更にツールの活用が生産性向上に効果があるはずだ。」と議論していた事が裏付けられたと考えられる。

今回の分析は以上で終わるが、今後は、ツールの機能範囲と効果があるツールの関係を調査、分析し、代表的なツールについて、以下のような機能範囲を知り、ユーザーが欲しがっているツールの機能範囲を予想すると面白いと考える。

(1) 単なるロジック、処理の流れを設計するためのツール

(2) モデルベースで設計を行い、コード生成もツールで可能なツール

(3) (2)に加えて、さらに組込み CPU の実行環境もシミュレート出来るツール。

5. まとめ

H20 年度では、状態遷移表を実際のシステムに適用するため、H19 年度の基本編に加えて、「条件の記述」と「遷移した状態に応じた処理の記述」を検討した。しかし、階層化について未検討な上、「実装結果から判明した課題」も出てきた。

そこで、H2１年度は、状態遷移表設計手法の標準化作業を加速するため、基本編と今まで議論して追加した応用編の両方の内容と、階層化などの案も含めて、予め文書を作成し、それについて、議論を進めていく形とする。その際、定例会だけでなくメーリングリストも活用して意見交換を行いながら議論を進め、手法書としてまとめていく作業を行っていきたい。