

モデルベース設計への移行におけるソースコードからの 状態チャート自動生成手法

川田秀司^{†1} 川勝則孝^{†1}

状態チャートを用いたモデルベース設計手法は、設計を抽象的な挙動から段階的に詳細化・検証し、最終的にコード生成することで、開発工数の削減・品質の向上を可能とする開発手法である。しかし、従来開発からの移行では、設計書やソースコードなどの継承が難しいという問題があり、この解決法が望まれている。本稿では、この解決法として、従来開発での既存資産であるソースコードから、その挙動を表現する状態チャートを自動的に生成する手法を提案する。この手法では、ソースコードに含まれる挙動情報を階層のない状態チャートとして抽出し、さらにこの状態チャートを適切に階層化・並列化することで理解性を向上させる。

1. はじめに

近年、状態チャート[1]やブロック線図[3]を使ってソフトウェアを設計・検証し、最終的にコードを生成するモデルベース設計手法が注目されている。特に組み込みシステムに代表されるリアクティブシステムに対する上流設計では、状態チャートを用いたモデルベース設計が有効である。

状態チャートを用いたモデルベース設計手法では、状態チャートの階層・並列表現を活用し、設計を抽象的な挙動から段階的に詳細化する。この際、各設計段階で検証を行い、挙動の正しさを保証しながら設計を進めるため、開発工数の削減と製品品質の向上が可能となる。

しかし、従来の設計手法から設計資産を移行するには、既存の設計書やソースコードを元に新たに状態チャートを作成し、テスト・検証し直さなければならず、非常に多くの工数が必要となる。

本稿では、状態チャートを用いたモデルベース設計手法への設計資産の移行方法として、従来開発手法の設計資産であるソースコードからその挙動を表現する状態チャートを自動的に生成する手法を提案する。

この手法では、ソースコードから挙動情報を抽出し、等価な意味を持つ状態チャートをフラットな（階層を持たない）形式で抽出する。さらに、抽出した状態チャートを等価変換により階層化・並列化を行うことで、理解性を向上させ、派生開発の基盤となる設計情報として利用できるようにする（図 1-1）。

2 章ではソースコードからフラットな状態チャートを抽出する手法、3 章ではフラットな状態チャートを適切に階層化・並列化する手法をそれぞれ述べる。さらに、4 章では、具体的なソースコードを題材とした適用評価により、状態チャートの抽出と、その自動階層化・並列化結果を示す。

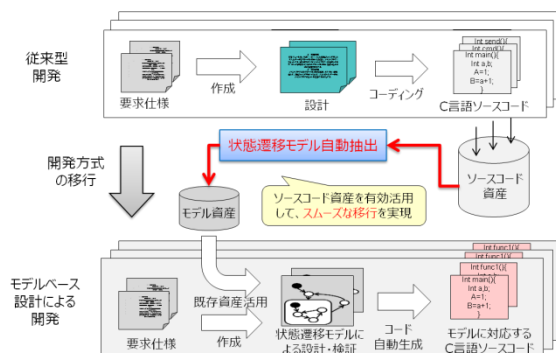


図 1-1 開発方式のモデルベース設計への移行

2. フラットな状態チャートの抽出

2.1 概要

状態チャートは、UML[1]で採用されている振舞いを表記するための図であり、David Harel により、状態遷移図をベースに作られた。遷移上にイベント、条件、アクションを表記することができ、その記述力は、通常のプログラム言語と同等であるため[a]、ソースコードが与えられれば、これと同等な挙動を表現する状態チャートを作ることが可能である。

ここで注目すべきことは、状態チャートでは「プログラムの状態」を表現するものとして、チャートを構成する「状態」と、チャート上で宣言されている変数の値の2種類の方法があることである。

図 2-1 と図 2-2 の状態チャートは、どちらも同じ簡易エアコンの挙動を示している。図 2-1 では、運転モードが冷房なのか暖房なのかを「状態」で表現しており、図 2-2 では「運転モード」という変数の値として保持している。

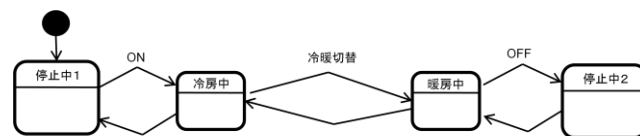


図 2-1 運転モードを状態として表した例

^{†1} (株)東芝 ソフトウェア技術センター
Software Engineering Center, Toshiba Corp.

a アクション記述言語に関しては本稿では言及せず、C言語をベースとした言語にて表現することとする

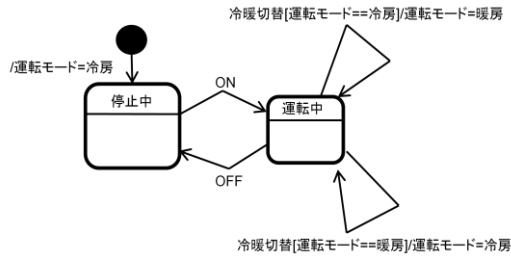


図 2-2 運転モードを変数として表した例

図 2-1 と図 2-2 の対応関係は、「停止中 1」には「停止中」かつ「運転モード=冷房」，「冷房中」には「運転中」かつ「運転モード=冷房」が，「暖房中」には「運転中」かつ「運転モード=暖房」が，「停止中 2」には「停止中」かつ「運転モード=暖房」が，それぞれ対応する。

この様に，ステートチャートでは何を状態として表現するかによりその記述が異なるため，ソースコードからステートチャートを抽出する際は，何を状態とするが重要となる．その候補は，状態が有限個であることを考慮すると，以下のものが考えられる．

1. 値域が離散有限である変数に対し，その変数の値
2. ある変数に対し，その変数の値が属する有限個の区間（変数の区間はプログラム中の分岐条件から求められるもの）．

本稿では 1 の「値域が離散有限である変数」（必要なら使用者がさらに絞り込みを行なったもの）が与えられたとして，ステートチャートを生成する方法を示す．2 については議論を省略するが，本稿で示す方法から容易に導くことが可能である．

2.2 定義

本節では次節で説明するアルゴリズムのために必要な定義を行う．

定義 2.2.1. 構文ポイント，ポイント

プログラムの各命令のプログラム上の位置を構文ポイントという．文脈上明らかな場合は単にポイントという．

定義 2.2.2. 実行パス

プログラムの実行を表すもので，実行順序順に並んだ構文ポイントの列を実行パスという．ある実行パスがポイント A で始まりポイント B で終わっている時，このパスを A から B への実行パスという [b]．

定義 2.2.3. 状態分割ポイント

状態を分割する位置を状態分割ポイントという．通常，イベント待ちに対応する処理を指定する(例えば OS の待ちを表す API 群)．プログラムの開始位置は状態分割ポイントとする．

b 2つのポイント間の全実行パスの集合は無限集合となる場合がある（例えば，無限に繰り返される可能性のあるループを含む場合）．

定義 2.2.4. 状態変数

状態を張る変数として与えられる変数を状態変数という．

状態遷移モデルの状態空間 T は，状態分割ポイントの集合を G ，状態変数を $x_1, x_2, \dots, x_n$ とし，各変数の値域の集合を $X_1, X_2, \dots, X_n$ とすると， $T = \{(g, (a_1, a_2, \dots, a_n)) | g \in G, a_i \in X_i\}$ で表される．生成される状態遷移モデルの状態の集合は T の部分集合となる．

定義 2.2.5. 状態変数書換えポイント

状態変数の値を変更するポイントを状態変数書換えポイントという．

定義 2.2.6. 連結

A から B への実行パスが存在するとき，A は B と連結であるという．

定義 2.2.7. 独立

状態分割ポイント A, B が連結であり，変数 x のスコプが A, B を含むとする．A から B へのすべての実行パスで x の B での値が， x の A での値に依存しないとき， x は A, B 間で独立であるという．

定義 2.2.8. 後続状態分割ポイント，後続連結，後続連結パス

状態分割ポイント A, B が連結であるとする．A から B へのある実行パス α で，B が最初に現れる状態分割ポイントであるとき，B は A の後続状態分割ポイントと呼ぶ，また，A は B と後続連結であるという．さらに， α を A から B の後続連結パスと呼ぶ．

定義 2.2.9. 被支配状態変数書換えポイント，被支配パス

A は B と後続連結であるとする．状態書換えポイント p がある A から B の後続連結パス α に含まれるとき， p は A の被支配状態変数書換えポイントという．このとき α を p の A に対する被支配パスという．

定義 2.2.10. 書換えパス，書換えパス集合 (1)

A は B と後続連結であるとする．A から B の全後続連結パスの集合において，A での状態変数の値に関わらず B での状態変数の値が等しくなる後続連結パスのいくつかを 1 つにまとめることで作られる集合を，A から B への書換えパス集合という．また，この集合の元を A から B への書換えパスという．

定義 2.2.11. 書換えパス，書換えパス集合 (2)

A の全後続連結パスの集合で，A での状態変数の値が $(a_1, a_2, \dots, a_n)$ であるとき，後続状態分割ポイント B での状態変数の値が等しくなる実行パスのいくつかを 1 つにまとめることで作られる集合を， $(a_1, a_2, \dots, a_n)$ に対する A から B への書換えパス集合という．また，この集合の元を $(a_1, a_2, \dots, a_n)$ に対する A から B への書換えパスという [c]．

c 全後続連結パスの集合が無限集合の場合でも，書換えパス集合は有限となる場合がある．書換えパス集合が無限になる場合でも，状態変数の値 $(a_1, a_2, \dots, a_n)$ に対する書換えパス集合は有限になる場合がある（しかし無限に

定義 2.2.12. 拡大写像, 補完変数

状態変数を $x_1, x_2, \dots, x_n$, G を状態分割ポイントの集合とする。 $A \in G$ である A に対し、

$$f(A) \in \{x | x \text{ は } A \text{ をスコープに含む,} \\ \text{状態変数以外の変数の部分集合}\}$$

とし、 $B \in G$, B が後続分割ポイントとなる $A \in G$, A から B への書換えパス p としたとき、 A , B , p の定め方によらず、 A での状態変数の値と $f(A)$ の変数の値から、 B の状態変数の値と $f(B)$ の変数の値を定めることができるならば f は拡大写像であるという。また、 f が拡大写像であるとき、 $f(A)$ を f における A の補完変数と呼ぶ。

定義 2.2.13. 基底変数

A を状態分割ポイントとする。 A の状態変数と A の補完変数を合わせたものを A の基底変数という。

定義 2.2.14. 独立関連

状態変数書換えポイント p が状態分割ポイント A の被支配状態変数書換えポイントとする。 p において、参照式中に現れる変数 x が、ある A に対する被支配パスで、状態分割ポイント A での状態変数の値と定数で表せないとき、 x は p で A と独立関連という。

例題 2.2.1. 独立関連

状態変数を a_1 , a_2 とする。図 2-3 において、(1)では x は p で A と独立関連となっているが、(2)では x は p で独立関連でない。なぜなら(2)では x は a_1+1 で置き換えることができるからである。

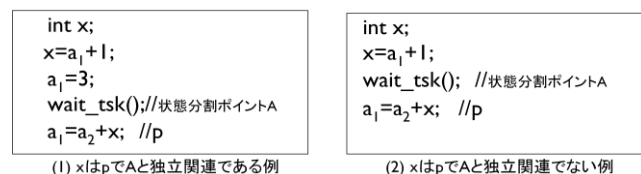


図 2-3 独立関連の例

定義 2.2.15. 支配変数

x は p で A と独立関連であるとする。すべての A に対する被支配パスで、スコープ内に A が入っている変数 $y_1, y_2, \dots, y_n$ と定数の A での値により p での x の値が表せるとき、 $y_1, y_2, \dots, y_n$ を x の支配変数という [d]。

2.3 アルゴリズム

本節では、ソースコードと状態変数が与えられたとき、この状態変数の値を状態とするステートチャートを自動的に抽出するアルゴリズムを示す。アルゴリズムの構成は図 2-4 の様になる。

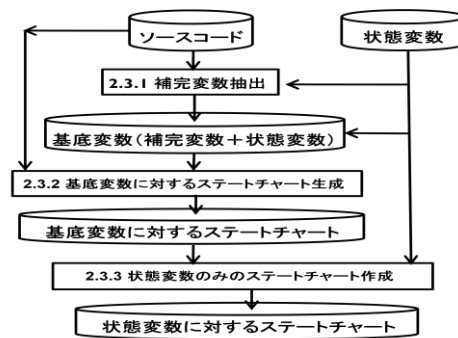


図 2-4 アルゴリズムの構成

2.3.1 補完変数抽出

ここでは補完変数の抽出アルゴリズムについて示す。状態変数は何らかの形で与えられたと仮定する。補完変数の抽出アルゴリズムは以下の Alg. 2-1 のようになる。

状態変数 e を $x_1, x_2, \dots, x_n$ とし、状態分割ポイントの集合を G とおく。また各状態分割ポイント $A \in G$ に対応する初期補完変数リストを $S_A = \{\}$ 、 A の被支配状態変数書換えポイントを P_A とする。 $L = G$ として、以下の様に各状態分割ポイントに対する補完変数リストを求める。

- 1 L からマークのついていない状態分割ポイント A を取り出す。全てにマークが付いていれば終了。
- 2 A にマークをつける。
- 3 foreach $p \in P_A$
 - 3.1 foreach $y \in \{x | x \text{ は } p \text{ に現れる変数で、} A \text{ を含んだスコープを持つ}\}$
 - 3.1.1 y が p で A と独立関連でない [f] なら 3.1 へ
 - 3.1.2 $y_1, y_2, \dots, y_n$ が y の支配変数 [g] としたとき、 $S_A := S_A \cup \{y_1, y_2, \dots, y_n\}$ 、 A のマークを取る。
 - 3.1.3 foreach $B \in \{y \text{ のスコープ内で } A \text{ と連結な状態分割ポイント}\}$
 - 3.1.3.1 y は B , A 間で独立でなく、 B の状態変数リストに y が含まれないならば、 $y_1, y_2, \dots, y_n$ が y の支配変数としたとき、 $S_B := S_B \cup \{y_1, y_2, \dots, y_n\}$ 。 B にマークがあればマークを取る。
- 4 1 に戻る。

Alg. 2-1 補完変数の抽出

この結果、状態変数に対し、各状態分割ポイントの補完変数が求まる。よって、各状態分割ポイントの基底変数が定まる。

2.3.2 基底変数に対するステートチャート生成

ここでは、基底変数に対するステートチャートを生成するアルゴリズムを Alg. 2-2 に示す。

各状態分割ポイントに対し、基底変数リストが作成されていることを仮定する。また、生成されるステートチャートの各状態 s は、 $s = (D, L)$ (D = 状態分割ポイント集合、 L = 基底変数リストに対応するその値) と表されるとする。

- e 初期状態変数はユーザが指定しても良いし、プログラム内の変数のうち離散有限である値域を持つ変数から値域の数等の基準で自動的に選択してもよい。
- f 独立関連の完全な判定は難しい。よって明確で無い時は独立関連であると見なして処理する。
- g 支配変数が見つけれない場合は失敗

なる場合も残る)。
d 支配変数が存在しない場合もある。

プログラム開始ポイントを SP, SP での基底変数リスト $S_{SP} = \{v_1, v_2, \dots, v_n\}$, 基底変数の初期値を $a_1, a_2, \dots, a_n$ とする. 状態集合を $ns = \{(SP, a_1, a_2, \dots, a_n)\}$, 遷移集合 $ts = \{\}$ とし, 以下の手順で状態遷移モデルを作成.

- 1 ns からマークのついていない状態を 1 つとり出しこれを s とする. s にマークをつける. 全てにマークがついているなら終了[h].
 - 2 $s = (D, L)$ に対し D の後続状態分割ポイントを $D_1, D_2, \dots, D_p$ とする.
 - 2.1 foreach $D_i \in \{D_1, D_2, \dots, D_p\}$
 - 2.1.1 s に対する D から D_i への変換パスの集合を求める.
 - 2.1.2 有限集合として求まらないときは失敗
 - 2.1.3 有限集合として求まったとき, これを $\{P_{i1}, P_{i2}, \dots, P_{ik(i)}\}$ とする. 各パス P_{ij} を通るための条件を && で書き並べたものを C_{ij} , そのパスで実行される処理を; で書き並べたものを A_{ij} とおく. D にて D_i の基底変数リストに含まれる各変数値に L が定める値を代入してパス P_{ij} を仮想実行[i]時, 各 D_i で D_i の基底変数リストの変数の値は定数となる (基底変数リストの決め方より明らか). よって, これを L_{ij} とする[j].

$$ts := ts \cup \{(D, L) \rightarrow [C_{i1}] / A_{i1} \rightarrow (D_i, L_{i1}), (D, L) \rightarrow [C_{i2}] / A_{i2} \rightarrow (D_i, L_{i2}), \dots, (D, L) \rightarrow [C_{ik(i)}] / A_{ik(i)} \rightarrow (D_i, L_{ik(i)})\}$$
 - 2.2 $ns := ns \cup \{(D_i, L_{i1}), (D_i, L_{i2}), \dots, (D_i, L_{ik(i)})\}$
- 但し ns でマークがついているものはマークをつけたままにする. 1 に戻る.

Alg. 2-2 基底変数に対する状態チャートを作成

Alg2-2 で得られた状態チャートの遷移には, 明示的なイベント記述がないが, 次の様な方法出にイベントを導入することができる.

各遷移 $t (\in ts) = (D, L) \rightarrow [C] / A \rightarrow (D', L')$ について,

- 状態分割ポイント D のブロック解除の事象
- 上記に加えて, D において値の変化のある変数の条件式を, C から抜き出す.

2.3.3 状態変数のみの状態チャート作成

ここでは, 前節で求めた状態チャートから, 基底変数のうち, 補完変数を取り除いた状態変数に対する状態チャートを作成するアルゴリズムを説明する. ns_1, ts_1 を補完変数を含んだ状態チャートとする. 状態変数のみの状態チャート ns_2, ts_2 は Alg. 2-3 のアルゴリズムにて作成する.

```

1  $ns_2 := \{\}, ts_2 := \{\}$ 
2 foreach  $s \in ns_1$ 
  2.1  $s'$  は s の補完変数の値を取り除いたものとしたとき,  $ns_2 := ns_2 \cup \{s'\}$ 
3 foreach  $u \in ts_1$ 
  3.1  $u = S_1 \rightarrow E[C] / A \rightarrow S_2$  とする[k].  $s_1$  の補完変数を  $y_{11}, y_{12}, \dots, y_{1n}$ , その値を,  $a_{11}, a_{12}, \dots, a_{1n}$ ,
  
```

- h このアルゴリズムは停止する条件は状態分割ポイントの各状態変数の値域が離散有限となることである.
- i 仮想実行パス P 上の構文ポイントを上から順に, 表されている計算・代入式の意味に従って, 変数の値マップを更新する.
- j 記述「 $S \rightarrow [C] / A \rightarrow S'$ 」は, 状態 S から S' の遷移で, ガード条件 C, アクション A のものを表す.
- k 記述「 $s_1 \rightarrow E[C] / A \rightarrow s_2$ 」は, 状態 s_1 から s_2 への遷移で, イベント E, ガード条件 C, アクション A のものを表す.

s_2 の補完変数を $y_{21}, y_{22}, \dots, y_{2m}$, その値を, $a_{21}, a_{22}, \dots, a_{2m}$, s_1, s_2 よりそれぞれの補完変数の値を取り除いたものを s'_1, s'_2 とすると,

$$ts_2 := ts_2 \cup \{s'_1 \rightarrow E[C \& y_{11} = a_{11} \& y_{12} = a_{12} \& \dots \& y_{1n} = a_{1n}] / A; y_{21} = a_{21}; y_{22} = a_{22}; \dots; y_{2m} = a_{2m} \rightarrow s'_2\}$$

Alg. 2-3 状態変数のみの状態チャート図作成

3. ステートチャートの階層化・並列化

3.1 概要

状態チャートは, 挙動を分かりやすく, もしくは少ない記述量で表現する有用な記法を数多く取り入れており, 特に, 階層や並列表現を用いると, 複雑な挙動の表現を理解性の高いチャートとして表現することが可能である (図 3-1, 図 3-2, 図 3-3).

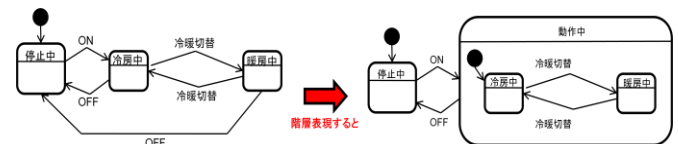


図 3-1 通常階層表現

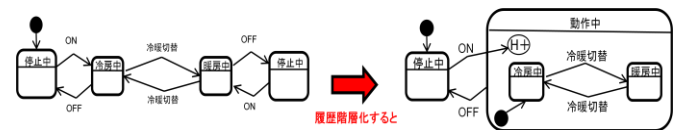


図 3-2 履歴階層表現

階層表現では, 浅い履歴状態と深い履歴状態を利用できる. 履歴状態は様々な使い方ができるが, 本稿では,

- 履歴状態を使用しない階層 (通常階層と呼ぶ).
- 深い履歴状態のみ使用し, かつ, その階層への遷移は履歴階層のみである階層 (履歴階層と呼ぶ).

の 2 つのパターンに限る. また, 並列表現では, 一方のチャートの遷移でもう一方のチャートの状態が実行中であるかどうかを参照することを許す.

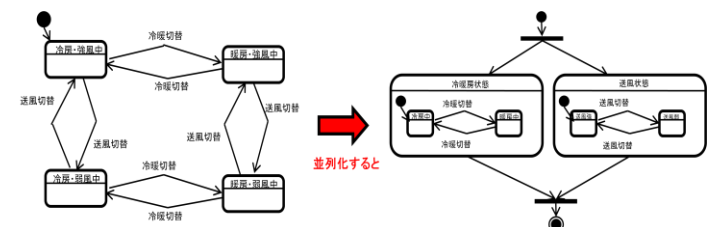


図 3-3 並列表現

2 章で説明した状態チャートの抽出法で生成される状態チャートは, 上記で説明した階層や並列表現を使用しないフラットなものであるため, 現実的な規模のプロ

グラムからは、巨大な状態チャートが抽出されることになるが、階層・並列表現を使って書換えを行うことにより、理解性の高い状態チャートに書換えることが可能である。

階層・並列記述への書換え結果は1つとは限らない。遷移記述のガード条件式の記述を利用することで、様々な形の並列化、階層化を行える場合も多い(図 3-4)。

本技術の目標は抽出された状態チャートの理解性を高めることであり、書換え候補から、理解性の高い形態の候補を絞り込むことが重要である。

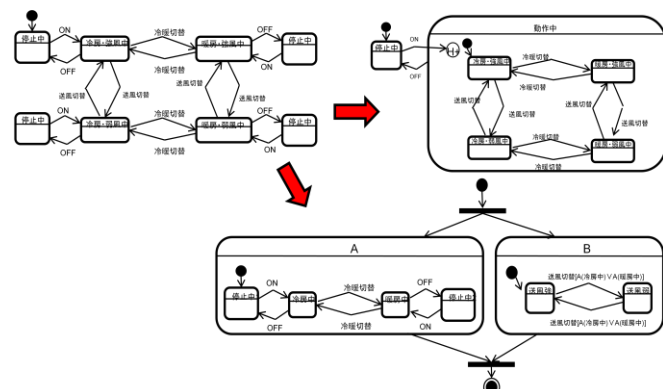


図 3-4 複数の書換えが存在する例

ここで、2種類の階層表現(通常および履歴)は、以下に示すような条件(階層化条件と呼ぶ)を満たす並列表現である。

● 通常階層表現(図 3-5)

- ✓ 下位に対応する状態チャートの初期状態からの遷移(以後この遷移を初期遷移と呼ぶ)は、全て、上位の階層化状態の時のみ起きる。
- ✓ 上位の階層化状態からの遷移と、同じイベント、同じ条件で、下位状態チャートの全ての状態から初期状態への遷移がある(以後、この遷移を階層脱出遷移と呼ぶ)。

● 履歴階層表現(図 3-6)

- ✓ 下位に対応する状態チャート図の全ての遷移は、上位の階層化状態の時のみ起きる。

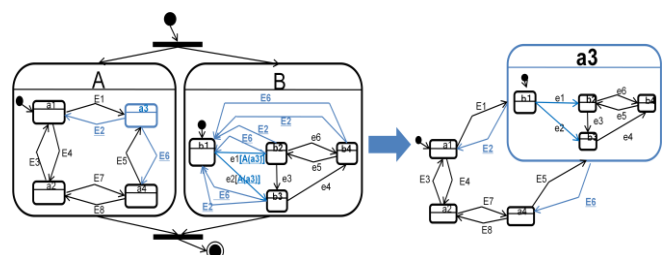


図 3-5 並列表現の通常階層表現への書換え

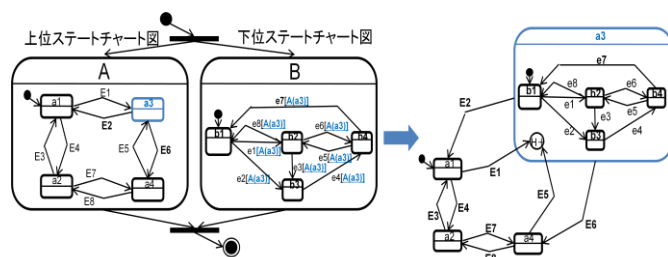


図 3-6 並列表現の履歴階層表現への書換え

よって、上記の各制約を満たす様な状態チャートへの書換えができる並列化アルゴリズムであれば、作成された並列化された状態チャートをチェックし、階層化が可能なものは階層化することで、並列化・階層化が可能となる。

以下では、まずフラットな状態チャートを自動的に並列化するアルゴリズムを示す。この結果は階層化条件を判定し、階層化とみなせるものは階層化と見なす。さらに、階層化・並列化に関して、**理解性基準**(理解しやすい・理解し難い)の判定基準)や、**理解性指標**(理解し易さの指標)の導入により、より理解しやすいものを選択する方法を示す。

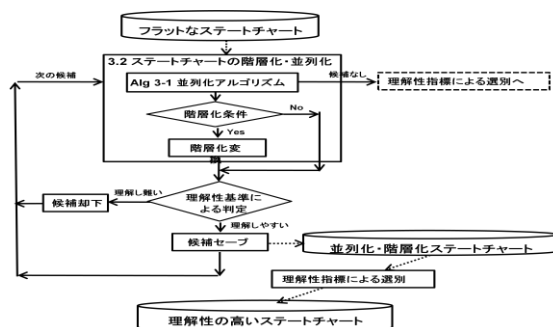


図 3-7 ステートチャートの階層化・並列化

3.2 ステートチャートの階層化・並列化

本節では、状態チャートを自動的に階層化・並列化するアルゴリズムを示す。アルゴリズムの基本は並列化で、生成された状態チャートを通常階層化制約、履歴階層化制約を満たすかをチェックすることで階層化とみなせるものは階層化する。

Alg. 3-1 は並列化候補を1つ見つけるアルゴリズムである。このアルゴリズムが成功した場合に、1つ分割パターンが見つかる。失敗した場合、選択候補が残っているものとも近い①, ②, ③の選択まで戻り、別の候補を網羅的に探索することで全ての分割パターンを見つけることができる。

元の状態チャートを S 、生成される状態チャートを A 、 B とする。

1 S の初期状態を s_1 以外の状態の集合を JS 、 A, B の初期状態を a_1, b_1 であるとする。

1.1 A, B の状態の集合(状態リスト)を J_A, J_B 、元の状態チャート S と作られる

ステートチャート A,B の状態の対応を示すデータを $JT=\{(s_1,(a_1,b_1))\}$, $JT'=\{(s_1,(a_1,b_1))\}$ とする. JT' は最終的に, 状態対応データリストの結果となる.

1.1.1 JT から 1 つ取り出す. これを, $(s_i, (a_i,b_i))$ とする. JT が空なら 1.1.5 へ.

1.1.2 S の遷移リストから, s_i からの遷移を 1 つ取り出す. 空なら 1.1.1 へ.

これを $s_j \leftarrow E[C]F \rightarrow s_j$ とする. 以下の検索パターン①, ②, ③のどれかを選び実行する.

① $s_j=(a_u,b_v)$ として A に 遷移 $a_s \leftarrow E[C \wedge B(b_v)]F \rightarrow a_s$ と状態 a_t を加える. JT, JT' に $(s_j,(a_u,b_i))$ を加える.

② $s_j=(a_u,b_v)$ として A に 遷移 $a_s \leftarrow E[C \wedge B(b_t)]F \rightarrow a_u$, 状態 a_u を, B に 遷移 $b_t \leftarrow E[C \wedge A(a_s)]F \rightarrow b_v$, 状態 b_v を加える. JT, JT' に $(s_j,(a_u,b_v))$ を加える.

③ $s_j=(a_u,b_v)$ として B に 遷移 $b_t \leftarrow E[C \wedge A(a_s)]F \rightarrow b_v$ と状態 b_v を加える. JT, JT' に $(s_j,(a_u,b_v))$ を加える.

1.1.3 それぞれ, ある a_i,b_i に対し $(s_j, (a_i,b_i)) \in JT'$ なら, $s_j=(a_i,b_i)$ として(つまり ①なら $a_u=a_i,b_v=b_i$, ②なら $a_u=a_i,b_v=b_i$, ③なら $a_s=a_i,b_t=b_i$ として), 整合性をチェックする. 整合性が取れていない [I] ならば失敗.

1.1.4 1.1.1 へ

1.1.5 A,B について, それぞれ, 同じ状態からの遷移で, 条件の状態制約のみが違うものは V を使って 1 つにまとめる [m].

1.1.6 A の各状態に対して, JT' から同時に起こる B のステートチャートの状態の集合を見つける. この際, この状態からの遷移で, 条件部の状態制約とこの集合が一致するものがあればこの遷移の状態制約を取り除く. A,B を逆にして同様の処理を行う.

1.1.7 最終的な結果のステートチャートは, A, B について, 通常階層化条件, 履歴階層化条件を満たすものは, それぞれの階層化をしたステートチャートとし, それ以外の場合は, A, B を並行動作するステートチャートとする.

Alg. 3-1 ステートチャートの自動階層化・並列化

3.3 理解性の高い階層化・並列化の選択

3.2 節のアルゴリズムでは, 元のステートチャートによっては非常に多くの候補が出力される. しかし, それらの多くの理解性は高くない. このため以下の二つの方法で候補を絞り込む.

- **理解性基準による判定**: ある判定基準により, 理解性を判定し, 許容できるもののみを残す.
- **理解性指標による選別**: ある理解性を示す指標により, 理解性の優劣を定量的に比較し, より良いもののみを残す.

例) 同じイベントを異なる階層化された, もしくは並列化されたステートチャートが受けるものは悪い.

例) 通常階層化>履歴階層化>並列化(依存なし)>並列化(依存あり)として, 最上位のグループのみ残す.

4. 適用評価

本章では, これまで述べたアルゴリズムによるソースコ

1 S では異なる状態 s_m, s_n が, どちらもある a_i, b_i に対し $s_m=(a_i,b_i)$, $s_n=(a_i,b_i)$ となる場合, もしくは, ある s_m が $(a_i,b_i) \neq (a_n,b_i)$ に対して, $s_n=(a_i,b_i)$, $s_m=(a_i,b_i)$ となる場合である.

m 例えば $a_s \leftarrow E[C \wedge B(b_i)]F \rightarrow a_t$, $a_s \leftarrow E[C \wedge B(b_i)]F \rightarrow a_t$ をまとめると $a_s \leftarrow E[C \wedge (B(b_i) \vee B(b_i))]F \rightarrow a_t$

ードからのステートチャート生成の例を示す. 題材は簡易洗濯機のボタンによるモード設定・動作制御の部分である. 図 4-1 に題材としたソースコードと 2 章のアルゴリズムを適用して生成されたフラットなステートチャートを示す. 対象コードは 81 行, 状態変数 4 つ(mm, sm1, sm2, scn) である. 状態分割ポイントは 2(slp_tsk(), slp_tsk_with_sentaku()) で, イベント種別はその返り値(val の値に対応)とする. また, この例では各ポイントの補完変数は抽出されなかった(状態変数=基底変数).

生成された状態数は 21, 遷移数は 66 であり, 比較的小規模な例であるが理解は難しい[n].

```
#include <stdio.h>
enum SWITCHSTATE { POWER_ON=100, POWER_OFF,START,
MODE_SPIN,MODE_WASH,STOP,OTHER};
enum SWITCHSTATE val = POWER_OFF;
enum POWERSTATE {OFF=200, ON};
enum POWERSTATE mm = OFF;
enum MODESTATE1 {SOFT_SPINDRY=400, HARD_SPINDRY};
enum MODESTATE1 sm1 = SOFT_SPINDRY;
enum MODESTATE2 {SOFT_WASH=500, HARD_WASH};
enum MODESTATE2 sm2 = SOFT_WASH;
enum SINGLESTATE {S_WASH=600,H_WASH,RINSE, S_SPINDRY,
H_SPINDRY,DRY,SETTING};
enum SINGLESTATE scn=SETTING;
int slp_tsk();
int slp_tsk_with_sentaku();
void sentaku_start(enum SINGLESTATE sc);
int send_all();
void sentaku_stop();
void main(){
while(1){//電源 OFF
val = POWER_OFF;
mm = OFF;
sm1 = SOFT_SPINDRY;
sm2 = SOFT_WASH;
scn = SETTING;
val=slp_tsk();//待ち
if(val!=POWER_ON) continue;
mm=ON;
printf("開始\n");
while(mm==ON){
val=slp_tsk();//待ち
switch(val){
case POWER_OFF://電源をきる
mm=OFF;
break;
case START://START ボタン
if(send_all()==0) mm=OFF;
scn=SETTING;
break;
case MODE_SPIN://モードボタン
if(sm1==SOFT_SPINDRY) sm1=HARD_SPINDRY;
else sm1=SOFT_SPINDRY;
break;
case MODE_WASH://モードボタン
if(sm2==SOFT_WASH) sm2=HARD_WASH;
else sm2=SOFT_WASH;
break;
}
}
}
}
int send_all(){//返り値 1 正常終了 0 OFF
if(sm2==SOFT_WASH) scn=S_WASH;
else scn=H_WASH;
sentaku_start(scn);//洗濯機稼働中
while(scn <= DRY){
val=slp_tsk_with_sentaku();
if(val==OTHER){//正常終了
if(scn==RINSE){//次は脱水
if(sm1==SOFT_SPINDRY) scn=S_SPINDRY;
else scn=H_SPINDRY;
} else if(scn==S_SPINDRY||scn ==H_SPINDRY){//次は乾燥
```

n 図表示において, 人が理解し易いアイテム数は 7 ± 2 であると言われている.

```

    scn=DRY;
  } else if(scn==S_WASH || scn==H_WASH){//次はすすぎ
    scn = RINSE;
  } else {//それ以外
    scn = scn + 1;//次の過程へ
  }
  sentaku_start(scn);//洗濯機稼働中
} else {//ボタン
  if(val==POWER_OFF){
    sentaku_stop();
    return 0;
  } else if(val==STOP){
    sentaku_stop();
    return 1;//この時は終了しない
  }
}
}
return 1;
}

```

図 4-1 題材としたソースコード

図 4-3 はこのステートチャートに 3 章で説明した階層化・並列化アルゴリズムを再帰的に適用した例である。結果として最初は通常階層化、次に履歴階層化、最後に並列化が行われている。尚、遷移のイベントから、各状態の意味を推測し、状態の中に説明文字を挿入している。階層化・並列化により理解性が高まっていることがわかる。

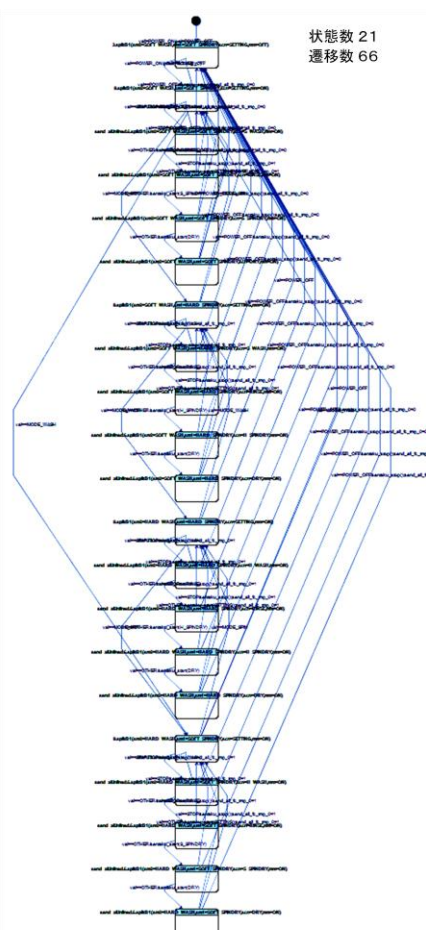


図 4-2 生成されたステートチャート

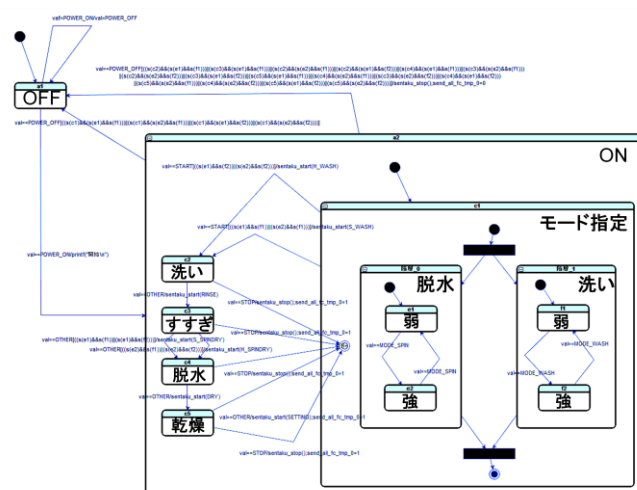


図 4-3 自動階層化・並列化アルゴリズムを繰返し適用

5. おわりに

本稿では、従来の手法による設計から、ステートチャートを用いたモデルベース設計手法による設計への移行を支援するため、ソースコードから等価な挙動をするステートチャートを抽出し、更にその理解性を向上させたステートチャートに変換する手法を示した。この手法では、モードや状態を管理する変数と、外部刺激を入力するコードに注目し、変数の値の組を状態、入力のタイミングと値をイベントとして、ステートチャートを生成する。更に、ステートチャートの階層化・並列化を行うアルゴリズムを繰り返し適用する。この結果、挙動を保持しながら理解性を向上した階層・並列構造を持つステートチャートが作成される。

この手法を洗濯機の題材に適用評価することにより、実際のソースコードからどのようなステートチャートが生成されるかを示した。

今後は、より理解性の高いステートチャート群を効率よく抽出できるように技術を最適化するため、

- 理解性（または、より広い概念である保守性）を定量化するためのステートチャートの評価指標
- 上記指標の高いステートチャートのみを生成する探索技術

について研究する。

参考文献

- [1] Scott W. Ambler, The Object Primer 2nd Edition, New York: Cambridge University Press, 2001.
- [2] David Harel, Statecharts: A Visual Formalism for Complex Systems, Department of Applied Mathematics, The Weizmann Institute of Science, 1986.
- [3] Bertola A. et al, A Matlab-Simulink flickermeter model for power quality studies, IEEE PES 11th International Conference on Harmonics and Quality of Power, Sep 2004