

15-A-11

モデルベース開発への移行に向けた C 言語ソースコードに対する状態遷移抽出技術の適用¹

1. はじめに

近年、ソフトウェアの大規模化・複雑化が急速に進むとともに、仕様に対する追加・変更が頻繁に行われるようになってきている。そのため、これまで以上にソフトウェアの開発効率と品質の向上が求められている。このうち、ソフトウェアの品質を向上させるためには、テストに頼らず、要求定義・設計において不適合を排除しておくことが必要である。これを実現する開発手法として、モデルベース開発が活用されつつある[1]。モデルベース開発とは、プログラミング言語より抽象的な記述形式を持つモデルを活用し、要求定義・設計での不適合の削減に有効な開発手法である。

従来型の開発からモデルベース開発に移行する際には、モデルベース開発に必要なモデルを構築する必要がある。これには、二つの方法が考えられる。一つは、既存開発の設計ドキュメントに記載された設計情報に基づいてモデルを構築する方法である。もう一つは、ソースコードに基づいて設計情報を抽出しモデルを構築する方法である。一方、不適合の解消や派生開発の対応などによるソースコードの修正が起因となって、設計ドキュメントに記載された設計情報がソースコードから抽出した設計情報と整合しない場合がありえる。このような場合、設計ドキュメントではなく、ソフトウェアの実際の挙動を表したソースコードに基づいてモデルを構築する方が適切である。ただし、その構築の際には、ソースコードにおいて仕様に基づく記述と実装依存の記述との区別が容易ではないため工数がかかること、人手によりモデルを構築する場合にはモデルの抽象度のばらつきが発生すること、といった問題がある。

本編では、この問題を解決するために、組込みソフトウェアの C 言語ソースコードを対象とし、組込みソフトウェアに不可欠な状態遷移モデルを抽出する技術を提案する。さらに、この技術を実際のソースコードに適用した事例を紹介し、本技術の効果と課題を述べる。

まず、「2. モデルベース開発の期待効果と課題」において、モデルベース開発を適用した場合の期待効果と、従来型の開発からモデルベース開発へ移行する際の課題について述べる。その後、「3. C 言語ソースコードからの状態遷移抽出技術」でその課題を解決する C 言語ソースコードからの状態遷移抽出技術を説明し、「4. 適用事例」で技術の有効性を示す適用事

¹ 事例提供: 株式会社東芝 インダストリアル ICT ソリューション社 IoT テクノロジーセンター
川勝 則孝 氏

例を述べる。最後に「5. まとめと今後」でまとめと今後について述べる。

2. モデルベース開発の期待効果と課題

まず、2.1 でモデルベース開発を導入した際に期待される効果を説明する。次の 2.2 では、その効果を享受するために従来型の開発からモデルベース開発へ移行する際に考えられる課題について説明する。

2.1. モデルベース開発の期待効果

モデルベース開発では、設計情報として、プログラミング言語より抽象的な記述形式を持つ記述形式であるモデルを用いる。モデルには、離散的な制御の変化を表す代表的な例として状態遷移モデルがあり、連続的な計算の流れを表す代表的な例としてブロック線図がある。

- ・ 状態遷移モデル：離散的で継続的な外部からの刺激に対する反応動作を表すのに適したモデル（図 15-A-11-1 (a)）
- ・ ブロック線図モデル：連続的で継続的な外部からのデータ入力に対する計算を表すのに適したモデル（図 15-A-11-1 (b)）

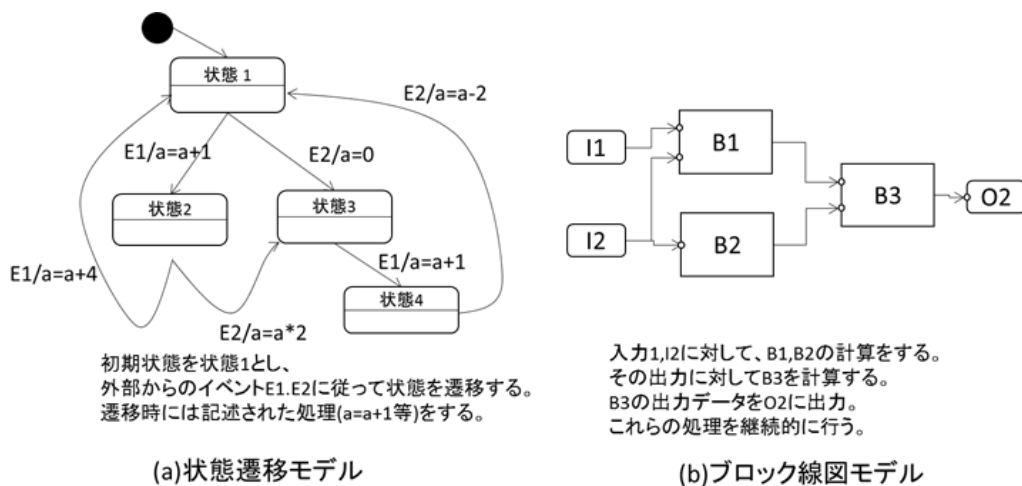


図 15-A-11-1 モデルベース開発でよく用いられるモデル

モデルベース開発では、モデルを用いたシミュレーションによって設計の正しさを検証する。例えば、シミュレーション結果に基づく動作ログの分析などで、設計の妥当性を検証することが挙げられる。このような検証により設計に起因する不適合を解消する。その結果、設計に起因する不適合がテストで発生するリスクが減少し、ソフトウェアの品質が向上すると考えられる。

2.2. モデルベース開発への移行時の課題

2.1 で説明した期待効果を享受するために、従来型の開発からモデルベース開発へ移行す

際には、モデルベース開発に必要なモデルを作成する必要がある。これには、設計ドキュメントに基づいてモデルを構築する方法と、ソースコードに基づいてモデルを構築する方法が考えられる。

設計ドキュメントに基づいてモデルを構築する場合、実際に動作するソースコードと設計ドキュメントとの間で内容に乖離がある場合が想定され、正確なモデルが得られないリスクがある。乖離が発生する理由として、例えば、不適合に対するソースコードの修正時や派生開発のためのソースコードの改造時に設計ドキュメントへの反映が行われなかったことなどが考えられる。この場合、正確なモデルが得られないリスクを考慮すると、ソースコードに基づいてモデルを構築する方が適切であると考えられる。

ただし、ソースコードからモデルを構築する際には、次のような課題が考えられる。

(1) ソースコードからモデルを構築する工数の増大

モデルを構築するためには、対象ソースコードに対して、仕様に基づく記述と実装依存の記述とを区別する必要がある。これは容易ではなく労力がかかる可能性がある。

(2) 構築するモデルの抽象度のばらつき

ソースコードからモデルを抽出する作業を人手で行う場合には、作業者に依存するリスクがあり、モデルの抽象度のばらつきが発生する可能性がある。

3. C 言語ソースコードからの状態遷移抽出技術

「2. モデルベース開発の期待効果と課題」で説明した従来型開発からモデルベース開発への移行時の課題を解決するために、我々は C 言語ソースコードから状態遷移モデルを自動抽出する技術を開発した（図 15-A-11-2）。状態遷移モデルは、組込みソフトウェアでは必須である外部からの刺激に対する離散的な挙動を表すことができる。

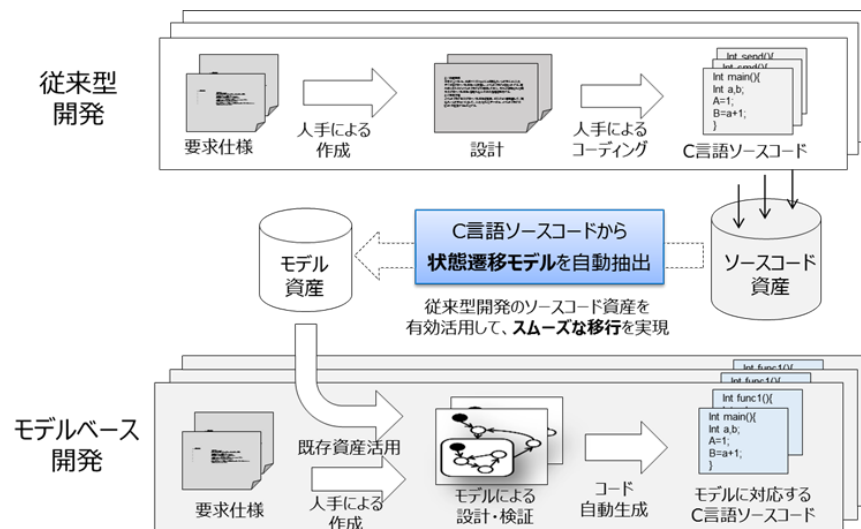


図 15-A-11-2 従来型開発からモデルベース開発への移行

ソースコードから状態遷移モデルを抽出するためには、まず、ソースコードから制御仕様を表す単一階層の状態遷移モデルを抽出し（図 15-A-11-3 (a)）、単一階層の状態遷移モデルに対して、モデルベース開発に適切な抽象度を確保するためにモデルの階層構造化を行う（図 15-A-11-3 (b)）。3.1 および 3.2 では図 15-A-11-3 (a),(b)の二つのステップの詳細を述べる。なお、以下の説明では、階層構造を含む状態遷移モデルの記述方式として、ソフトウェア開発で利用されるモデリング言語である UML2.0 (Unified Modeling Language) [2]の状態機械図を用いる。

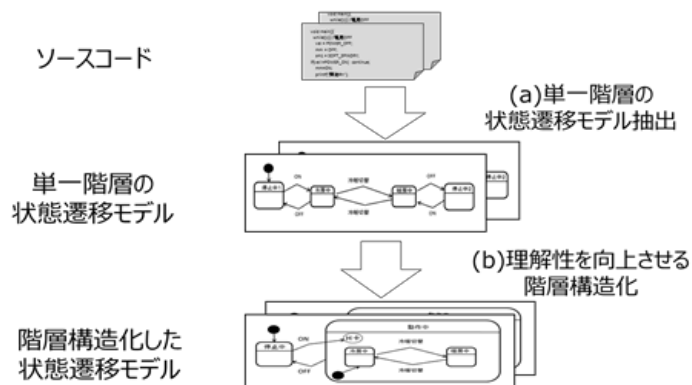


図 15-A-11-3 ソースコードからの状態遷移モデル抽出の概要

3.1. ソースコードからの単一階層の状態遷移モデルの抽出

本節では、ソースコードから単一階層の状態遷移モデルを抽出する方法について説明する。まず、3.1.1においてソースコードから状態遷移モデルの構築に必要な状態と遷移を識別する方法について説明する。次に、単一階層の状態遷移モデルを抽出する方法を 3.1.2 および 3.1.3 で説明する。3.1.2 では状態と遷移を抽出するために必要な変数の変化を表す計算式を抽出する方法を説明する。3.1.3 では、それに基づいて状態遷移モデルを抽出する手法を説明する。

3.1.1. ソースコードからの状態遷移モデルの捉え方

状態遷移モデルを抽出する対象は、並行動作を含まないもので、図 15-A-11-4 に示すような C 言語ソースコードである。対象となるソースコードが表す制御仕様として図 15-A-11-5 に示すような状態遷移モデルを抽出する。

ソフトウェアの制御仕様を表す状態遷移モデルは、外部環境からの刺激をイベントとし、有限個の状態（図 15-A-11-5 の角無し矩形）と、指定されたイベントを受信した際に状態の変化と実行される処理を表す遷移（図 15-A-11-5 の状態間を結ぶ有向線分）で表現されるものである。遷移が生じるきっかけを表す受信イベントと遷移する条件を表すガード条件を記述し、遷移が生じるときに実行する動作（アクションと呼ばれる）を記述する。図 15-A-11-5 の各遷移に対して、「受信イベント[ガード条件]/アクション」という形式で記述されているものである。

```

1: int st;          ← 状態変数
2: int a,b;
3: ....(略)
4: void tsk1(void){ ← エントリー関数
5: .... (略)
6: dly_tsk(100);/*時間待ち*/
7: if( a>10 ){      ↑ 状態分割ポイントP
8:   st=1;
9:   b=b+1;
10: } else{
11:   st=2;
12: }
13: dly_tsk(100);/*時間待ち*/
14: .... (略)      ↑ 状態分割ポイントQ
15: }

```

図 15-A-11-4 対象ソースコードの例

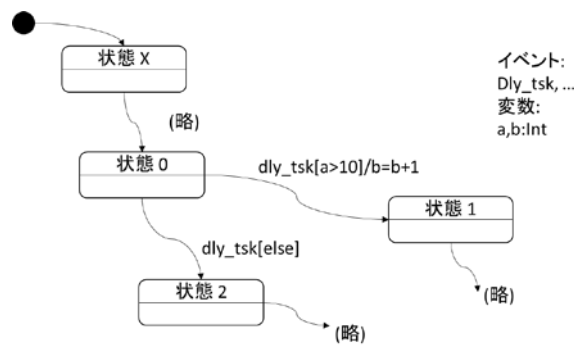


図 15-A-11-5 ソースコードの例に対応する状態遷移モデル

ソースコードから状態と遷移を識別する基本的な考え方を、図 15-A-11-4 のソースコードとそれに対応する図 15-A-11-5 の状態遷移モデルを参照しながら説明する。遷移は、外部環境からのイベントの検知後、ガード条件が満たされた際に生じる。その後、指定されたアクションが実施され、次の状態が確定することで遷移が完了する。図 15-A-11-5 の状態遷移モデルにおける状態 0 を開始点とする二つの遷移は、図 15-A-11-4 のソースコードにおいて、6 行目の「dly_tsk0」呼出しから 12 行目までの if 分岐を含むソースコードに対応する。「dly_tsk0」関数呼出しは、イベントの検知待ちを表す API (Application Programming Interface) 呼出しである。これは、OS (Operating System) が提供する指定時間待ちやタスク通信の受信待ち等を表すものであり、ここでは状態分割ポイントと呼ぶ。一方、それぞれの遷移の実行条件は、ソースコード上の if 分岐 (7 行目) の then 節 (8 行目～9 行目)、else 節 (11 行目) をそれぞれ実行する条件「a>10」と「!(a>10)」である。

それぞれの遷移の完了時点で、次の状態が確定する。これは、遷移完了後の次の状態におけるイベントの検知を待つ API 呼出し (状態分割ポイント) の直前の時点となる。図 15-A-11-5 の状態 1 と状態 2 が確定する遷移直後の状態分割ポイントは、図 15-A-11-4 の 13 行目の「dly_tsk0」呼び出しである。

状態は、ソフトウェアの設計時に想定していると考えられる動作状況を識別する変数 (状態変数と呼ぶ) の値を用いて区別されるものである。また、状態変数以外の変数に関する処理は、遷移のアクションとして表す。ここでは、設計者が抽出すべき状態の観点に関連する状態変数を選択することを前提としている。

図 15-A-11-4 の例を用いて、状態遷移モデルを抽出するために事前に設計者が特定すべき情報を示す。ソースコードの記述のうち抽出範囲を示す関数を特定する（図 15-A-11-4 の 4 行目の「`tsk10`」）。次に、対象の待ちの API を特定することで状態分割ポイントを特定する（図 15-A-11-4 の 6、13 行目の「`dly_tsk 0`」）。さらに、ソースコード内で定義されている変数の中から状態変数を特定する（図 15-A-11-4 の 1 行目の「`st`」）。

3.1.2. 記号実行を利用した変数値の変化を表す計算式の抽出

ソースコードから状態の変化と遷移記述（受信イベントとガード条件とアクション）を抽出するために、二つの隣接する状態分割ポイント間のソースコード部分を実行した結果を表す変数値の変化を表す計算式を抽出する。これを実現するために、記号実行[3]を活用する。

記号実行とは、ソースコードの任意の部分の実行結果を網羅的に表現するため、実行開始時における変数値を記号で表し、実行終了時における変数値をその記号を使った式として構築する技術である。記号実行の結果は、実行パスを選択する条件（制約条件）とそれを選択した場合の計算式の対応で表される。実行パスの制約条件は、各分岐の選択を表す条件の論理積で表すことができる。記号実行は、ソースコードから関数の外部仕様を表す決定表を抽出する技術[4]など様々な技術で活用されている。

図 15-A-11-4 のソースコードについて状態分割ポイント P から Q の間に記号実行を適用する例を図 15-A-11-6 に示す。この例では P において「`st=0`」を前提としている。

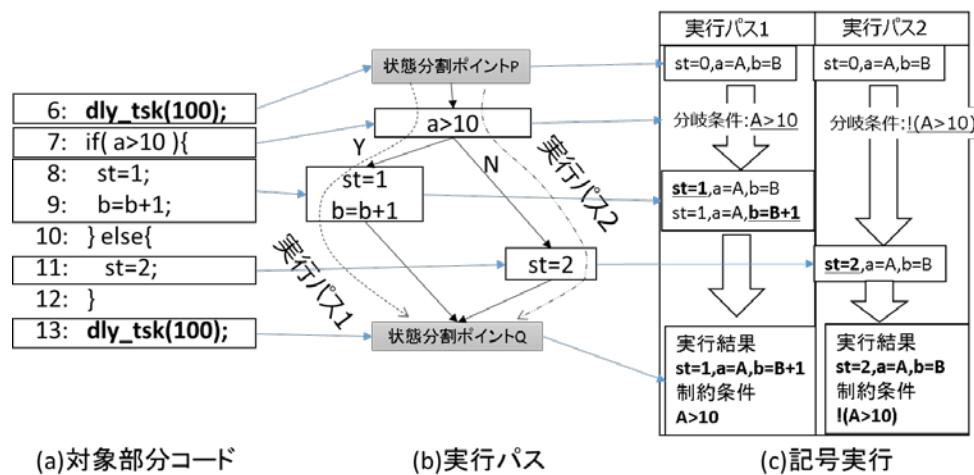


図 15-A-11-6 記号実行の例

P から Q までの範囲（図 15-A-11-6 (a)）の実行パスは図 15-A-11-6 (b)のようになる。すなわち、条件式「`a>10`」が成り立つ場合（実行パス 1）、成り立たない場合（実行パス 2）の二つである。実行パス 1 では、「`st=1, b=b+1`」（図 15-A-11-6 (a) 8、9 行目）が実行され、分岐条件は「`A>10`」、実行結果は、「`st=1, a=A, b=B+1`」となる。このパスの制約条件は、「`A>10`」となる。

ここで、記号 A、B は、状態分割ポイント P における変数 a、b の値を表す。一方、実行パス 2 では、「st=2」（図 15-A-11-6 (a) 11 行目）が実行され、分岐条件は「!(A>10)」、実行結果は、「st=2、a=A、b=B」となる。このパスの制約条件は、「!(A>10)」となる。このようにして、記号実行を用いると図 15-A-11-6 (a) のソースコードを実行したときの変数の変化式が図 15-A-11-6 (c) のような形で、実行パスの制約条件と変数の変化式リストの組として得られる。

3.1.3. 抽出された変数の計算式に基づく状態遷移モデルの構築

3.1.2 で説明した変数の変化を表す計算式を求める方法に基づき、状態遷移モデルを構築する方法を説明する。まず、初期状態を一つ作成し、3.1.2 の方法で抽出される計算式を用いて、初期状態を含む既に作成されている状態を開始点とする遷移と遷移する先の状態を構築する。これを繰り返すことで、状態遷移モデル全体を構築する。

既に作成されている状態を開始点とする遷移と遷移する先の状態の構築について、図 15-A-11-6 で示した記号実行の結果を用いた例で説明する。状態分割ポイント P から Q までの範囲に記号実行を適用した結果（図 15-A-11-6 (c)）から状態分割ポイント P、Q における変数値を表す式を抜粋したものを図 15-A-11-7 (a) に示す。これらの変数の計算式を用いると、図 15-A-11-7 (b) のような状態と遷移が構築される。実行パス 1 の実行結果は、状態変数 st に対する式「st=1」で表される状態とその状態に向かう遷移（図 15-A-11-7 (b) 左下）となる。実行パス 2 の実行結果は、「st=2」で表される状態とその状態に向かう遷移（図 15-A-11-7 (b) 右下）を表す。状態変数以外の変数に対する式は、遷移に記述されるアクションとする。

実行パス 1 の結果にある変数 b に対する式「b=B+1」は、図 15-A-11-7 (b) の左の遷移のアクション「b=b+1」に対応する。一方、実行パス 2 では値の変化がないので、図 15-A-11-7 (b) の右の遷移のアクションはない。さらに、実行パスの制約条件からガード条件を構築する。実行パス 1 の制約条件から左の遷移のガード条件は「a>10」、実行パス 2 の制約条件から右の遷移のガード条件は「!(a>10)」となる。

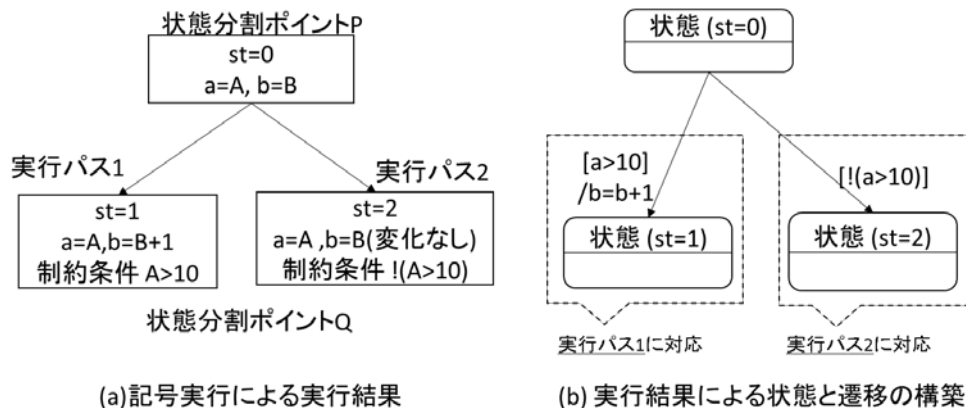


図 15-A-11-7 記号実行の結果による遷移と状態の構築

遷移は、状態分割ポイントにおけるイベントの入力待ちが解除されガード条件が満たされたときに生じる。その後、アクションが実行され次の状態が確定することで遷移が完了する。遷移上に形成される受信イベントの記述は、単純に状態分割ポイントにある API 呼出し記述のみで識別するのではなく、イベントの種別がわかるように受信イベントを示す値を識別して抽出する。

すでに構築されている状態の中で、状態変数の値が一致するものがあれば同じ状態とみなす。既存の状態から上記で示した遷移と状態の構築を新たな状態が構築されなくなるまで繰り返して、あり得る状態とそれらを結ぶ遷移をすべて抽出する（図 15-A-11-8 (a)）。同じ状態とみなされる状態を 1 つにまとめることで状態遷移モデル全体を構築する（図 15-A-11-8 (b)）。

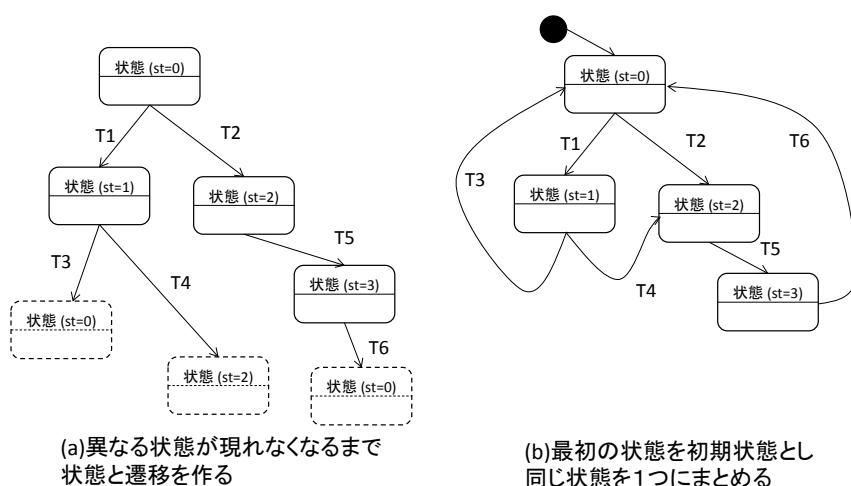


図 15-A-11-8 状態遷移モデルの構築

3.2. 状態遷移モデルの階層構造化

3.2 では、3.1 で説明した方法で得られる単一階層の状態遷移モデルを階層構造化する方法について説明する。

単一階層の状態遷移モデルは、状態数が多くなると、設計者がモデル全体を把握することが難しくなる。状態遷移モデルなどのグラフ表現では、7～9 のアイテムを超えると理解が困難になると言われている[5]。そこで、モデルの中に潜在的に含まれる階層構造の候補を網羅的に検出し、それらのうち理解性を向上させるために有効な候補を選択して階層構造化する。以下では、まず、階層構造化された状態遷移モデルの網羅的な候補の構築について説明する。次に、理解性向上の観点から候補のモデルを絞り込むことについて説明する。

3.2.1. 階層構造化された状態遷移モデルの網羅的な候補の構築

状態遷移モデルの階層構造化は、単一階層の状態遷移モデルを図 15-A-11-9 に示するような三種類のいずれかの階層構造に変換し、さらにこの変換を繰り返し適用することで実現する。

図 15-A-11-9 (a)は、状態に入場した際の詳細な挙動を表す入れ子の状態遷移モデルを記述

する通常の階層構造の状態遷移モデルを表し、ここでは通常階層と呼ぶ。図 15-A-11-9 (b) は、通常階層と同様な入れ子の状態遷移モデルであるが、状態に再入した際に前回出場した際の状態の履歴を再現する階層構造の状態遷移モデルを表し、ここでは履歴階層と呼ぶ。これらに加えて、図 15-A-11-9 (c)は、UML 状態機械図の表記では、状態を破線で二つに分割した領域に状態遷移モデルをそれぞれ記述する形式で表されるもので、それぞれ独立に変化する状態遷移モデルを表し、ここでは並行階層と呼ぶ。並行階層は、独立して変化する二つのグループの状態変数によって識別される状態群を表現するのに適した階層構造である。これらを合わせて、状態遷移モデルの基本的な階層構造とする(図 15-A-11-9)。

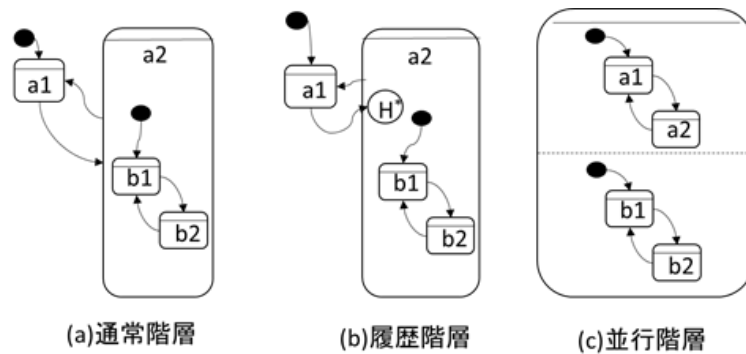


図 15-A-11-9 階層構造化で構築される三種類の基本的な階層構造

階層構造化を効率的に実現するために、一つのアルゴリズムで三種類の階層構造に対応できるようにする。具体的には、単一階層の状態遷移モデルを便宜的に二つの状態遷移モデルに分離して扱う。図 15-A-11-10 に示すように上位モデルと下位モデルと呼ぶ二つのモデルに分離することで、三種類の階層構造の違いを同様に扱えるようにする。

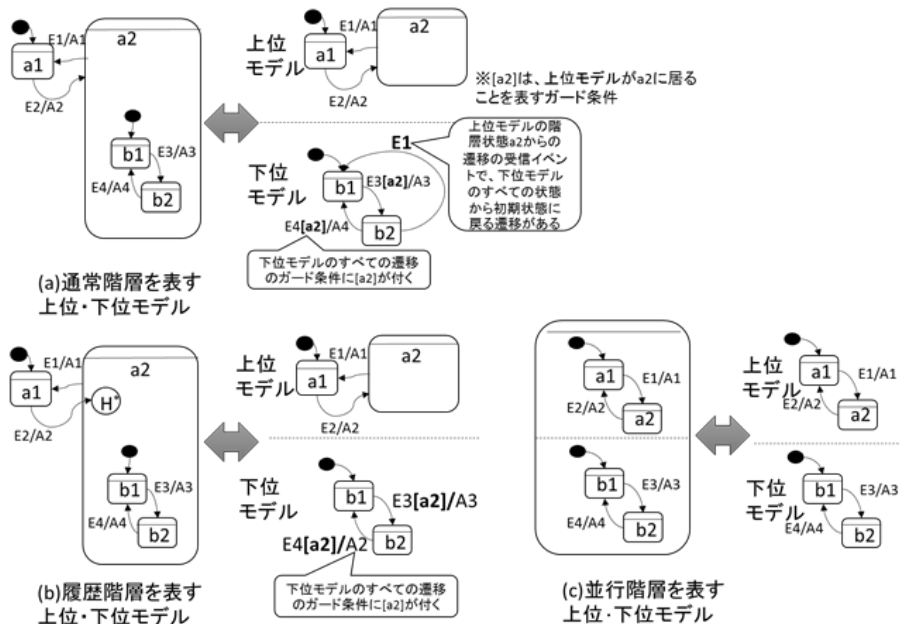


図 15-A-11-10 通常階層、履歴階層、並行階層に対応する上位・下位モデル

階層構造化の可能性を網羅的に検出するために、元のモデルの各遷移と状態に対して、上位・下位のモデルへ遷移と状態を振り分けるパターンをすべて洗い出す。ここで考慮するパターンは、元のモデルの遷移と行先の状態に対して、「上位モデルのみ、上位モデル・下位モデルの両方、下位モデルのみ」の遷移と状態を対応させる。ただし、上位モデル・下位モデルの両方を振り分ける場合は、対応しない挙動が起こらないように下位モデルの遷移のガード条件として、上位モデルの状態の位置に関する条件が付加される。

図 15-A-11-11 は、元のモデルの最初の遷移 T を振り分ける例を示している。この例では、元のモデルの状態 $S1$ が、上位モデルの状態 $a1$ 、下位モデルの状態 $b1$ の組み合わせに対応していることを前提に、元のモデルの遷移 T と状態 $S2$ の対応を上位のモデルと下位のモデルに振り分けるパターンを示している。

- (1) 上位モデルに遷移と状態を振り分けるパターンでは、上位モデルの状態 $a1$ から遷移 T を生成し、遷移の到達先として状態 $a2$ を生成する。元のモデルの状態 $S2$ には、上位モデル状態 $a2$ と下位モデルの状態 $b1$ の組み合わせを対応させる。
- (2) 上位モデル、下位モデルの両方に振り分けるパターンでは、上位モデルの状態 $a1$ から遷移 T を生成し、遷移の到達先として状態 $a2$ を生成し、下位モデルの状態 $b1$ から遷移 T にガード条件「 $a1$ 」(これは上位モデルの状態が $a1$ にあることを意味する) を付加したもの生成し、到達先として状態 $b2$ を生成する。元のモデルの状態 $S2$ には、上位モデル状態 $a2$ と下位モデルの状態 $b2$ の組み合わせを対応させる。
- (3) 下位モデルに遷移と状態を振り分けるパターンでは、下位モデルの遷移 T を生成し、遷移の到達先として状態 $b2$ を生成する。元のモデルの状態 $S2$ には、上位モデル状態 $a1$ と下位モデルの状態 $b2$ の組み合わせを対応させる。

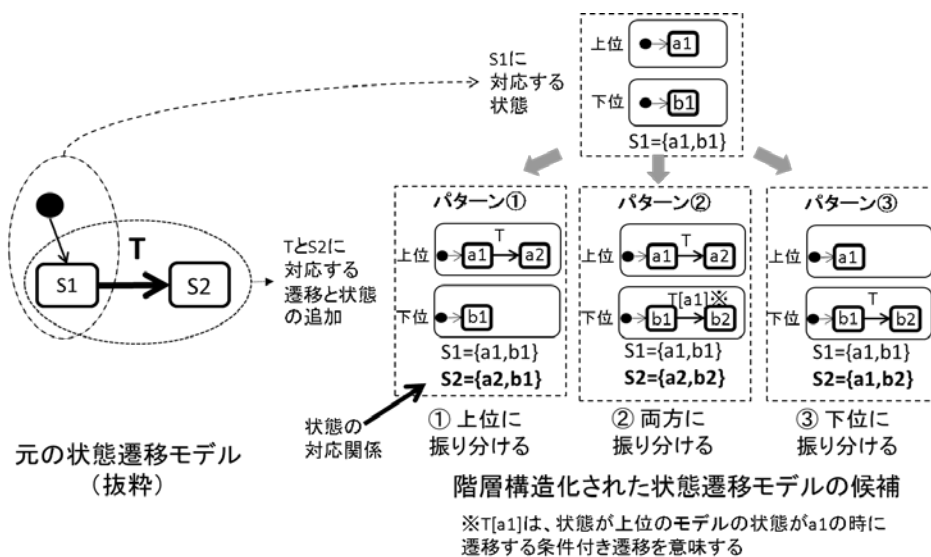


図 15-A-11-11 元の状態遷移モデルの遷移と状態を上位・下位モデルに振り分け

これを元の状態遷移モデルの初期状態からはじめ、すべての遷移に対して網羅的に実施する。上位モデル・下位モデルに遷移を追加する際、遷移先は既存の状態か新規の状態となる。ここで、元のモデルの状態に対して、上位モデル・下位のモデルの遷移先の状態の組が、1対1に対応しなければならない。この対応に矛盾が生じる時点で、階層化の候補からはずす。一方、すべての遷移と状態が振り分けられたモデルは、階層化の候補となる。

3.2.2. 階層構造化候補の絞り込み

3.2.1 で説明した方法で得られる階層構造化した状態遷移モデルの候補は、遷移の振り分けの自由度が高いため、複数個得られることがある。階層構造化の候補をできる限り少なくするために、階層構造化のアルゴリズムに対して、候補の絞り込みを実施する。絞り込む方法は、階層構造化のアルゴリズムの適用時に、定められた基準を満たさないものを候補からはずしていくものである。この基準は、理解のしやすさの観点を考慮して次のようなものとしている（図 15-A-11-12）。

- ・ (a)一方が空のモデルになる様な無駄な階層を形成しない（実質的に階層が作られないと考えられる）
- ・ (b)同じガード条件となる遷移を一方のモデルに集める（同じ意味の記述は一方のモデルに固めておくという意味がとりやすいと考えられる）

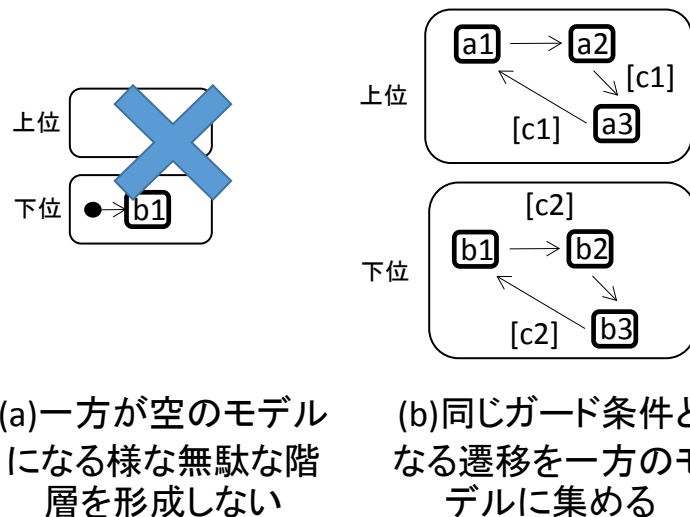


図 15-A-11-12 理解のしやすさの観点による候補の絞り込み

4. 適用事例

C 言語ソースコードを対象として、「3. C 言語ソースコードからの状態遷移抽出技術」で説明した技術を組み込んだツールを開発し、組み込みソフトウェア（C 言語記述のソースコード）に適用した。

本編で示す適用事例は二つあり、一つ目は、比較的小規模な事例で C 言語ソースコードか

ら状態遷移モデルを抽出する技術の有効性を示すものである。二つ目は、製品レベルの規模を持った事例で本技術の適用可能レベルを示すものである。

4.1. 小規模な組込みソフトウェアに対する適用事例

対象は、仮想の洗濯機の制御において、簡単なユーザインターフェースとそれによる設定に応じて実行動作を指示するソフトウェアである。なお、対象のソースコードは、本技術の効果を確かめるために実装の範囲を絞り込んでおり、ソースコードの規模は 300 行程度になっている。また、タスクは一つである。状態変数は、状態を記憶することを想定している変数を選択した（4 個）。

まず、3.1 で説明した方法を適用し、単一階層の状態遷移モデルを抽出した(図 15-A-11-13)。抽出された状態遷移モデルは状態数と遷移数が多く、一度に把握することは難しい。抽出された階層構造のない状態遷移モデルは、一度に理解することは困難である（状態数が 21、遷移数が 66）。

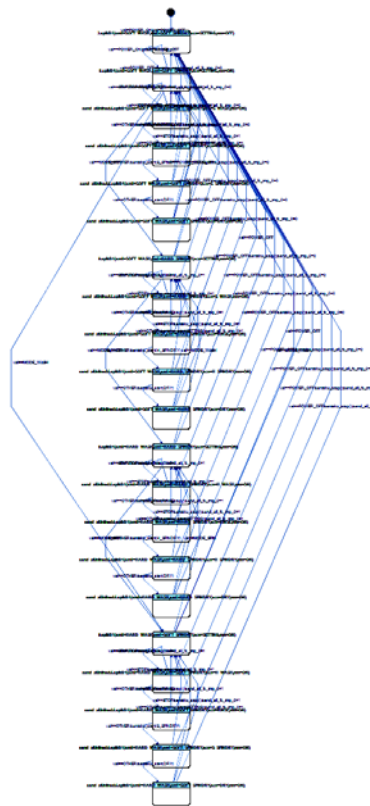


図 15-A-11-13 ソースコードから抽出された単一階層の状態遷移モデル

次に、モデルの抽象度を上げるために、この状態遷移モデルに対して、3.2 で説明した階層構造化を実施し、候補を絞り込む技術を適用した。適用結果として、図 15-A-11-14 に示すモデルが唯一の階層構造化候補として構築された。階層構造化が実施され、元の状態遷移モデルに比較して理解がしやすくなっていることが確認できる。このモデルでは、通常階層、

履歴階層、並行階層を組み合わせた階層構造が構築されている。

候補のモデルの数が 1 つになったのは、3.2.2 で説明した候補の絞込みが効果的に働き、理解しにくい候補が排除されたことを示している。また、抽出されたモデルは、適度に階層構造が適用され、一つの階層内の状態数は、2～6 程度になっており、理解しやすいと考えられる。

この試行の結果、ソースコードから制御仕様を表す状態遷移モデルを抽出でき、さらに階層構造化により理解しやすい状態遷移モデルが構築できた。このモデルは、モデルベース開発のモデルとして適切な抽象度を持っており、C 言語ソースコードから状態遷移モデルを抽出する技術の有効性を確認できた。

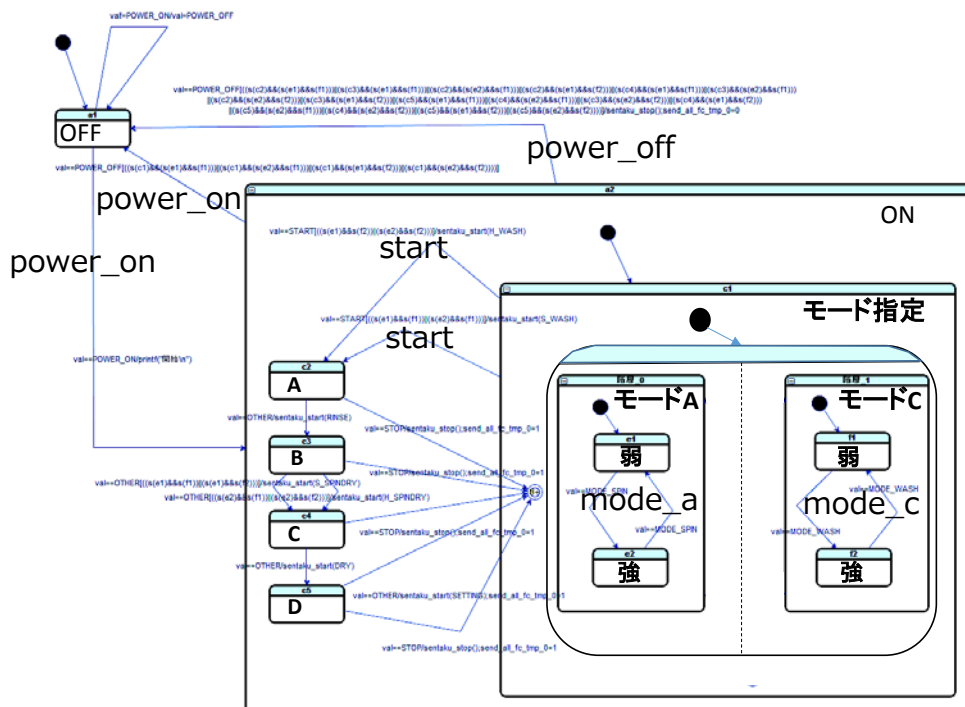


図 15-A-11-14 階層構造化された理解しやすい状態遷移モデル

4.2. ある制御機器の製品ソフトウェアに対する適用事例

対象は、ある制御機器の製品ソフトウェアである。このソースコードは、C 言語で記述されており、3 万行の規模である。ベースとなる OS は μ ITRON4.0[6]互換の OS である。タスクの構成は、全体の動作モードを司るタスクが 1 つあり、その他の 30 個のタスクは、その状態に従って動作するものである。

まず、3.1 で説明した単一階層の状態遷移モデルを抽出する方法を適用する。適用対象は、システムの起動時から動作し他のタスクを制御すると考えられるタスクとして、全体の動作モードを司るタスクとした。状態分割ポイントは、外部からの刺激の取り込みを待ち合わせ

る API である「rcv_mbx」（メールボックスの受信待ち）、及びタスクの時間待ちを表す API である「dly_tsk」（タスクの指定時間待ち）とした。状態変数は、グローバル変数でありかつ関数からの読み書きをされている可能性があるものから、変数名などを参考にして状態を保持しているとみなされるものを選択した（12 個）。

3.1 で説明した方法を適用すると単一階層の状態遷移モデルを抽出することができた。この状態遷移モデルは、状態が 102、遷移が 353 の規模であった。状態や遷移の数は多く、そのままでは理解が困難であると考えられる。

次に、この状態遷移モデルについて、3.2 で説明したモデルの階層構造化を適用したところ、階層化の候補が一つもないという結果となった。この結果の妥当性を示すために、対象の状態遷移モデルには階層構造化できる要素を含まないことを目視により確認した。

この試行の結果により、本技術が製品規模のソフトウェアに適用できるレベルに達していると考えられる。なお、この試行では、対象タスクによる状態変数の変化の情報のみを使って状態遷移モデルを抽出した。この対象タスクと並行動作する他のタスクと協働で同一の変数を更新するような変数を状態変数として取り扱いができるようになると、階層化の対象となりうる状態遷移モデルが抽出される可能性がある。このことを実現する技術の拡張は、今後の課題である。

5. まとめと今後

近年、ソフトウェアの品質向上のために活用されつつあるモデルベース開発は、プログラミング言語より抽象的なモデルを活用し、要求定義、設計での不適合の削減に有効な開発手法である。従来型の開発からモデルベース開発に移行する際には、ソースコードに基づいてモデルを構築することが有効である。ただし、ソースコードにおいて仕様に基づく記述と実装依存の記述との区別が容易ではないため工数がかかること、人手によりモデルを構築する場合にはモデルの抽象度のばらつきが発生すること、といった課題があった。

本編では、この課題を解決するための技術として、C 言語ソースコードからの状態遷移抽出技術を提案した。本技術は、ソースコードから状態遷移モデルを抽出し、さらに階層構造化を実施するものである。これを実施することで、モデルベース開発に用いるために適切な抽象度を持つ状態遷移モデルを得ることができる。本技術を C 言語で記述された小規模な組み込みソフトウェアに適用した結果、モデルベース開発で用いるために適切な抽象度の状態遷移モデルを抽出できることが確認でき、技術の有効性が確認できた。また、制御機器の製品ソフトウェアへ試行した結果、実適用レベルに達していることを確認した。

今後は、モデルの理解しやすさを向上するための適切な状態変数の選択方法、また、より大規模なソースコードからモデルを抽出可能にすることや並行動作に関する記述を含むソースコードからモデルを抽出可能にすることなどを行い、本技術の適用範囲の拡大を図る。

参考文献

- [1] 独立行政法人情報処理推進機構 技術本部 ソフトウェア・エンジニアリング・センター：
「組込みシステムの先端的モデルベース開発実態調査」調査報告書、2012.8.23 [オンライン].
Available、<https://www.ipa.go.jp/files/000004608.pdf>
- [2] Object Management Group : Unified Modeling Language(UML)、2014.4.21 [オンライン]. Available、<http://www.uml.org/>
- [3] C. J. King : Symbolic execution and program testing、Communication of the ACM.
1976.7.19
- [4] 酒井政裕、岩政幹人：記号実行によるプログラム改造支援技術、東芝レビュー第 67 巻、
12 月号、2012
- [5] G. A. Miller : The Magical Number Seven, Plus or Minus Two: Some Limits on Out
Capacity for Processing Information、Psychological Review 63、1956
- [6] 坂村健：μITRON4.0 標準ガイドブック、パーソナルメディア、2001 年
- [7] M. Weiser : Program Slicing、Proceedings of the Fifth International Conference on
Software Engineering、1981

掲載されている会社名・製品名などは、各社の登録商標または商標です。

独立行政法人情報処理推進機構 技術本部 ソフトウェア高信頼化センター (IPA/SEC)