モデルベース開発に向けたソースコードからの状態遷移モデル抽出技術

Technology to Extract State Machine Models from Source Codes for Model-Based Software Development

川勝　則孝　　川田　秀司
■KAWAKATSU Noritaka　■KAWATA Hideji

近年，ソフトウェア開発では，大規模・複雑化への対応や頻繁な要求変更への迅速な対応が求められている。これらに効果的な手法として，プログラミング言語より抽象的な記述を用いたモデルベース開発が挙げられる。しかし，開発資産としてはこれまで多く蓄積されてきた設計書やソースコードではなく，新たにモデルを構築する必要があるため，従来型開発からモデルベース開発への移行が困難であるという課題があった。

この課題を解決するために東芝は，モデルベース開発における設計で活用されているモデルの一つである状態遷移モデルをソースコードから抽出する技術を開発した。この技術により，開発資産となるモデルを効率よく構築することができ，従来型開発からモデルベース開発への移行を加速できる。

In recent software development, it has become necessary to rapidly respond to not only the increasing scale and complexity of software but also to frequent specification changes. Model-based software development utilizing descriptions at a higher level of abstraction compared with programming languages is now attracting attention as an effective countermeasure. However, it is difficult to shift from conventional development based on source codes and design documents accumulated as development assets to model-based development based on models requiring many worker hours to build from these conventional development assets.

As a solution to this issue, Toshiba has developed a technology to extract state machine models, one of the most frequently used types of models for model-based development, from source codes. This technology can efficiently build models as development assets and is expected to contribute to an acceleration of the shift to model-based development.

1　まえがき

近年，ソフトウェア開発は，大規模・複雑化への対応や頻繁な要求変更への迅速な対応が求められている。これらに対応する効果的な手法として，モデルベース開発が挙げられる。モデルベース開発とは，開発工程の上流から下流に至るまで一貫してモデルを活用する手法である。モデルベース開発では，設計工程で問題を排除することが可能であり，これにより下流工程（実装・テスト工程）での不具合が削減され，開発期間の大幅な短縮が期待できる。

モデルベース開発におけるモデルとは，プログラミング言語よりも抽象的な記述を用いて，動作や計算の流れなどの各種側面に注目して設計や開発対象の環境を表現するものである。動作を表現する例として状態遷移モデルがあり，計算の流れを表す例としてブロック線図を用いたモデルがある。モデルベース開発では設計工程で問題を解決するために，設計のモデルを用いて検証を実施する。例えば，ソフトウェアの動作をモデルに基づいてシミュレーションすることで，その動作を検証し，設計した時点で動作の正しさを確かめることができる。

従来型の開発からモデルベース開発へ移行するためには，従来型開発の成果物であるソースコードから動作を表す仕様（動作仕様）を表すモデルを構築する必要がある（**図1**）。しかし，ソースコードからの人手によるモデルの構築には多大な工数を必要とするため，課題となっている。

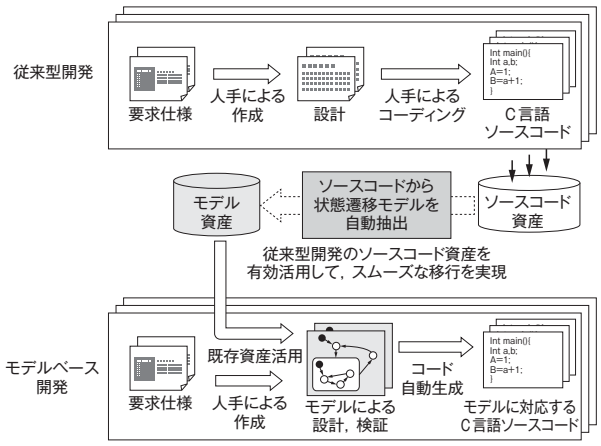

図1．従来型開発からモデルベース開発への移行 ― 従来型開発で蓄積しているソースコードから効率的にモデル資産を構築できると，モデルベース開発へスムーズに移行できる。

Shift from conventional to model-based software development

東芝は，これを解決するために，ソフトウェアの動作仕様を表す状態遷移モデルをソースコードから抽出する技術を開発した。ここでは，開発した技術の概要と，組込みソフトウェア

に試行した結果について述べる。

2　ソースコードからの状態遷移モデルの抽出

この章では，ソースコードから状態遷移モデルを抽出する技術について述べる。この技術は，二つのステップから構成される（**図2**）。まず，動作仕様を表す単一階層の状態遷移モデルをソースコードから抽出する。次に，単一階層の状態遷移モデルに対して，理解性を向上させるために階層構造化を行う。以下に，それぞれのステップについて詳細を述べる。

2.1　単一階層の状態遷移モデルの抽出

この節では，動作仕様を表す単一階層の状態遷移モデルをソースコードから抽出する技術について述べる。

まず，状態遷移を抽出するために注目するソースコードの性質について例を用いて述べる。状態遷移モデルを捉える対象は，並列動作を含まないもので，**図3**に示すようなC言語ソースコードが対象となる。ソースコードに対して，状態遷移抽出のポイントとなる補足情報を追加する。ここでは，ソースコードの記述のどの範囲で動作仕様を抽出するかを表すために対象関数を指定する（図3の4行目のtsk1()）。また，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（状態を識別する値を持つ変数）を指定する（図3の1行目のst）。更に，ソフトウェアが実行されるシステムの外部環境の変化（イベント）

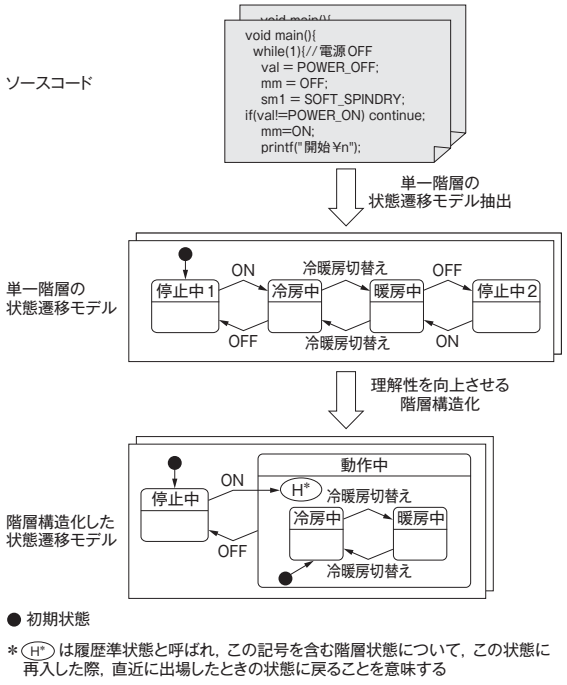

図2．ソースコードから状態遷移モデルを抽出する技術の構成 — 単一階層の状態遷移モデルをソースコードから抽出し，更に理解度を向上させるために階層構造化する。

Configuration of state machine model extraction technology

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

図3．対象ソースコードの例 — モデル抽出のために，対象関数はtsk1，状態変数はst，状態分割ポイントは時間待ちを表すdly_tsk()を指定する。

Example of input source code for state machine extraction

を検知する箇所（状態分割ポイント）を特定する（図3のdly_tsk()の記載のある6，13行目）。これは，イベントの検知の明示的な記述であり，指定時間待ちを表すものである。

次に，単一階層の状態遷移モデルを抽出する技術の詳細を述べる。ソフトウェアの動作仕様から状態と遷移を捉えるためには，状態変数などがどのように変化するかを抽出する必要がある。通常のソフトウェアの実行では，具体的に設定される変数の値に従って実行される一つの実行パスについてだけ，変数の変化を識別する。これに対して，仮想的に実行の解析を行う記号実行は，あらゆる実行パスについて具体的な変数の値を定めずに変数の変化を表す計算式を網羅的に抽出できる。

以下に，記号実行を利用した，遷移に相当する部分コードの実行による変数値の変化を表す計算式の抽出と，抽出された結果を使って状態遷移モデルを構築する方法について述べる。

2.1.1　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遷移モデルとして動作仕様を抽出するためには，遷移に相当する部分コードの実行による変数値の変化を抽出する必要がある。この部分コードは，二つの状態分割ポイント間のコードに相当する。これを実現するために，記号実行[1]を活用する。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[2]でも活用されている。

記号実行とは，ソースコードの任意の部分の実行結果を抽象的に取り出すため，実行開始時における変数値を記号で表し，実行終了時の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。実行パスを選択する条件（制約条件）は，各分岐の選択を示す条件で表すことができる。複数の分岐が含まれる場合，制約条件は全ての分岐の制約条件を同時に満たすものとなる。

図3のソースコードに対して状態分割ポイントPからQの間

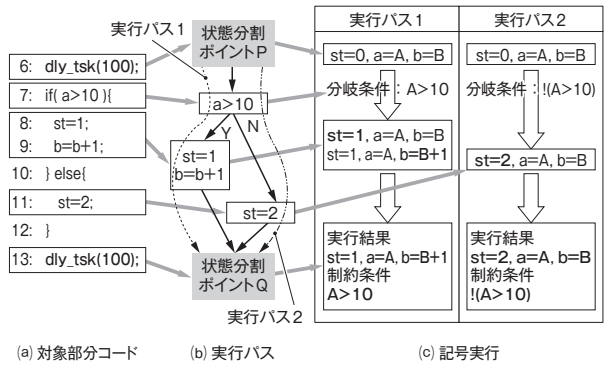

図4. 記号実行の例 — 対象部分コードに対応する実行パスを求め，各実行パスに対して記号実行を実施する。

Example of application of symbolic execution to source code

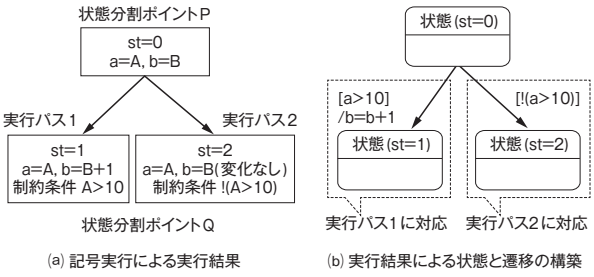

図5. 記号実行の結果による遷移と状態の構築 — 記号実行の実行結果から状態と遷移を構築する。

Construction of transitions and states resulting from symbolic execution

に記号実行を適用する例を，**図4**に示す。この例では，Pにおいてst=0を前提としている。PからQまでの範囲（図4(a)）の実行パスは図4(b)のようになる。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4(a)の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4(a)の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。

2.1.2 抽出された変数値の変化に基づく状態遷移モデルの構築　2.1.1項では，各状態分割ポイント間の部分コードに対して，状態遷移モデルの構築に必要な変数（状態変数など）の変化を表す計算式を求める方法を示した。これにより抽出される計算式を用いて，既存の状態から遷移と状態を構築する。

遷移と状態を構築する方法を，図3の例を用いて述べる。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4(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5**(a)に示す。これにより，図5(b)のような遷移と状態が構築される。

変数の計算式のうち状態変数に対する式は，状態を構築するために用いる。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となる（図5(b)の左下）。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となる（図5(b)の右下）。

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5(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5(b)の右の遷移のアクションはない。更に，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。

既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。遷移と状態を構築する方法を新たな状態が構築されなくなるまで繰り返して，状態遷移モデル全体を構築する。

2.2 状態遷移モデルの階層構造化

この節では，単一階層の状態遷移モデルの階層構造化を実現する技術について述べる。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UML2.0（Unified Modeling Language）[3]の状態機械図を用いる。

単一階層の状態遷移モデルは，状態数が多くなると，開発者が動作仕様全体を把握することが難しくなる。一般に，状態遷移モデルのグラフ表現では，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[4]。この問題を改善するために，状態遷移モデルの中に潜在的に含まれる階層構造を自動的に検出して状態遷移モデルを構築する。

以下に，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と，理解性の観点から候補モデルの絞込みについて述べる。

2.2.1 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　階層構造化された状態遷移モデルの構築では，上位と下位の二つの状態遷移モデルに分割することを基本とする。二つのモデルは，**図6**に示すように，通常階層，履歴階層，あるいは並列階層のいずれかを持つモデルを表す[3]。この基本単位としての構築を状態遷移モデルに繰り返し適用することで，入れ子構造として複数の階層構造化を実現する。

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全て洗い出す。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，上位のみ，上位・下位の両方，下位のみのモデルに遷移と状態を追

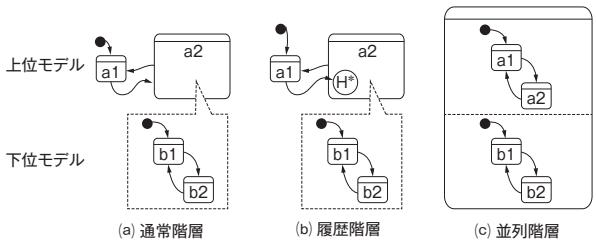

図6. 状態遷移モデルの3種類の基本的な階層構造 — 通常の階層構造，履歴を記憶する階層構造，及び並列動作を表す構造を示す。

Basic hierarchy consisting of three state machine models

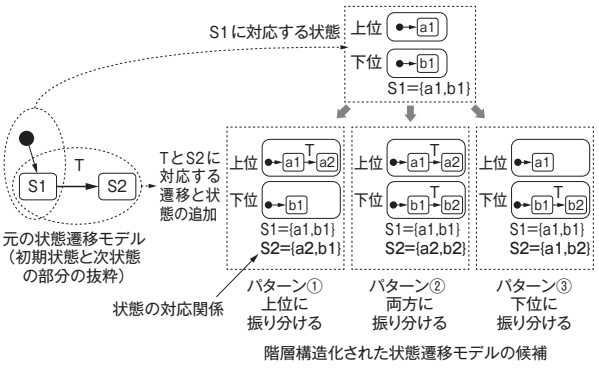

図7. 元の状態遷移モデルの遷移と状態を上位・下位モデルに振分け — 元のモデルの各状態と遷移を，3種類の振分けパターンにより，上位・下位モデルに振り分けていく。

Partitioning of transitions of original model into upper and lower resulting models for hierarchical restructuring

加する3種類である。**図7**は，元のモデルの遷移Tを振り分けている例を示している。遷移Tに対して3種類のパターンで遷移を追加し，それぞれのパターンの遷移に応じて状態が追加されている。

これを元の状態遷移モデルの初期状態からはじめ，全ての遷移に対して網羅的に実施する。上位モデルと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。このとき，上位モデルと下位モデルの遷移先の状態の組が，元の状態遷移モデルの状態と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，階層化の候補から外す。一方，全ての遷移と状態が振り分けられた状態遷移モデルは，階層化の候補となる。

2.2.2 理解性の観点による候補の絞込み このような方法で得られる状態遷移モデルには，理解性の観点から有益でないものが含まれる場合がある。そのような候補のうち，理解性の観点で基準を満たさないものを候補から外す。この基準には次のようなものがある。

(1) 一方が空の状態遷移モデルになるようなむだな階層を形成しない

(2) 同じガード条件となる遷移を一方の状態遷移モデルに集める

(3) 並列階層より解釈が容易な通常・履歴階層を優先する

3 試行結果

C言語ソースコードを対象として，2章で述べた技術を組み込んだプロトタイプツールを開発した。このツールを用いて家電製品を想定したソフトウェアのC言語ソースコードから状態遷移モデルを抽出することを試行した。対象は組込みソフトウェアで，ユーザーによる動作モード設定とその設定に従って動作するものである。

2.1節で説明した技術を適用し，単一階層の状態遷移モデルを抽出した（**図8**）。抽出された状態遷移モデルは状態数と

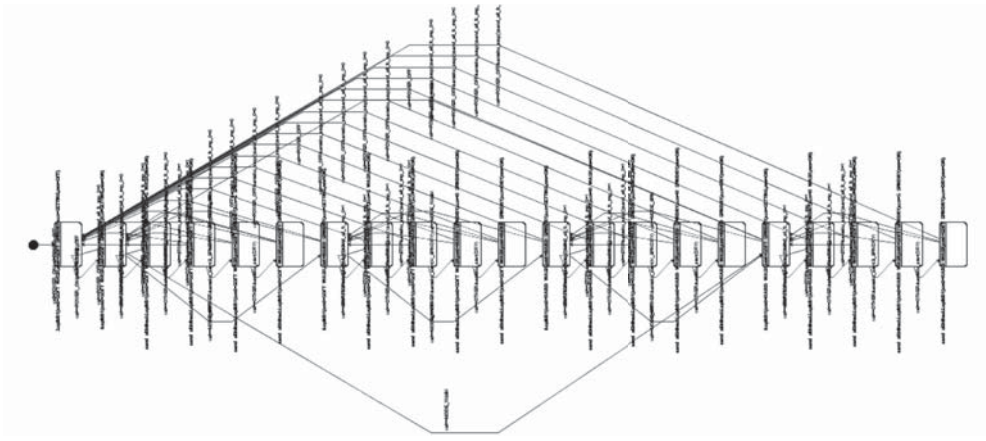

図8. ソースコードから抽出された単一階層の状態遷移モデル — 抽出された階層構造のない状態遷移モデルは，一度に理解することは困難である（図は状態数が21，遷移数が66の例）。

State machine model with single hierarchy extracted from source code

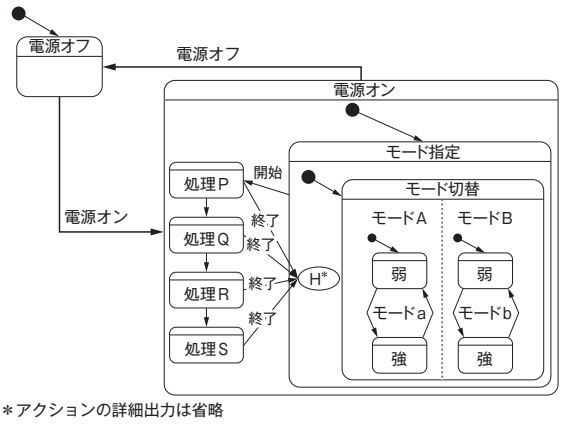

図9．階層構造化された理解しやすい状態遷移モデル ― 階層構造化が実施され，元の状態遷移モデルに比較して理解性が向上した。このモデルでは，通常階層，履歴階層，並列階層を組み合わせている。

Easy-to-understand hierarchically structured state machine model

遷移数が多く，一度に把握することは難しい。

次に，理解性を向上させるために，この状態遷移モデルに対して，2.2節で述べた技術により階層構造化を行った。階層構造化の結果として，**図9**に示す状態遷移モデルが一つだけ構築された。状態遷移モデルの数が一つになったのは，2.2.2項で述べた候補の絞込みが効果的に働き，理解性の低い候補が削除されたことを示している。また，得られた状態遷移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。

この試行の結果，ソースコードから動作仕様を表す状態遷移モデルを抽出でき，更に階層構造化により理解性の高い状態遷移モデルが構築できた。この状態遷移モデルは，モデル資産として活用できることが期待される内容であり，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した。

4　あとがき

従来型の開発からモデルベース開発へのスムーズな移行を実現するために，ソースコードから状態遷移モデルを抽出する技術について述べた。試行結果では，比較的小規模なソースコードを用いて，理解性の高い状態遷移モデルが抽出できることを確認した。この技術により，既に開発したソフトウェアのソースコードからモデル資産を効率的に構築できるようになる。

今後は，モデルベース開発が有効な対象分野の製品に対して，この技術の適用範囲を拡大し，モデルベース開発への移行を加速していく。

文　献

(1) King. J. C. Symbolic execution and program testing. Commu. ACM. **19**, 7, 1976. p.385－394.
(2) 酒井政裕 他. 記号実行によるプログラム改造支援技術. 東芝レビュー. **67**, 12, 2012, p.35－38.
(3) Object Management Group. "Unified Modeling Language (UML)". <http://www.uml.org/>, (accessed 2014-04-21).
(4) Miller, G. A. The Magical Number Seven, Plus or Minus Two: Some Limits on Out Capacity for Processing Information. Psychological Review. **63**, 1956, p.81－97.

川勝　則孝　KAWAKATSU Noritaka
ソフトウェア技術センター ソフトウェア設計技術開発担当参事。モデルベース設計を中心とするソフトウェア設計技術の開発に従事。日本ソフトウェア科学会会員。
Corporate Software Engineering Center

川田　秀司　KAWATA Hideji
ソフトウェア技術センター ソフトウェア設計技術開発担当主務。モデルベース設計を中心とするソフトウェア設計技術の開発に従事。情報処理学会会員。
Corporate Software Engineering Center