# 記号実行を利用した細粒度状態遷移表を抽出するリバースエンジニアリング手法

**Symbolic execution-based approach to extracting a fine-grained state transition table from source code**

清水 貴裕 * 山本 椋太 † 吉田 則裕 ‡ 高田 広章 §

**あらまし** 組込みシステム開発の現場では，レガシー化の影響によりソースコードの保守や再利用が困難となっている．組込みシステム向けのソフトウェアの多くは，状態遷移モデルに表すことにより設計を明らかにすることができる．本稿では，記号実行による解析によってソースコードの実行経路の情報を得られることに着目し，そこから細粒度状態遷移表を抽出する手法を提案する．また，提案手法により細粒度状態遷移表を抽出するツールを開発し，ツールのケーススタディを行う．

## 1 はじめに

組込みシステムのソフトウェア開発では，対象とするハードウェアの仕様変更に伴い，ソースコードを変更することが多い．同一システムとして出荷される製品であっても，工場毎にハードウェアにばらつきが生じるため，開発した工場ごとにソースコードを少しづつ変更することがある．また，既存製品と類似したシステムを開発する場合，既存製品のソースコードを部分的に変更することで，再利用することが多い [1]．これら背景から，組込みシステム開発ではソースコードの変更と再利用が繰り返される傾向にある．

組込みシステム開発では，性能の低い計算機で高い応答性を求められるため，C 言語でソースコードを記述することが多い [2]．C 言語は，オブジェクト指向言語と比べて抽象的な記述を行う能力が低い．そのため，ハードウェアの仕様変更にあわせてソースコードを変更する際に，条件分岐文を追加することでソースコードの複雑度を上昇させがちである [3]．特に，納期が迫ったプロジェクトでは，安易に可読性の低い条件分岐文を追加し，保守性や再利用性を低下させがちである．

組込みシステム技術協会の状態遷移設計研究 WG（以降，状態遷移設計研究 WG）では，複雑な条件分岐を含むソースコードを対象としたリバースエンジニアリングついて研究を行っており，その一環として細粒度状態遷移表を提案してきている [1]．細粒度状態遷移表は，コンパイル単位に含まれるファイル集合（以降，コンパイル単位）内における状態遷移の理解を支援するために提案されており，コンパイル単位内で宣言された変数の 1 つを状態を表す変数（状態変数）として，状態とイベントに対応する処理や状態遷移を表現した表である．細粒度状態遷移表が存在するならば，複雑な条件分岐を含むコンパイル単位であっても，保守や再利用時に細粒度状態遷移表と照らし合わせながら，理解を進めることができると考えられる．一般的な状態遷移モデルと比べると，システムレベルではなくコンパイル単位レベルの状態遷移を表している点が，細粒度と言える．

状態遷移設計研究 WG では，細粒度状態遷移表およびコンパイル単位から手作業で細粒度状態遷移表を抽出する手順を提案してきたが，限られた人的資源の中で，複雑な条件分岐を含むコンパイル単位から手作業で細粒度状態遷移表を抽出するこ

*Takahiro Shimizu, 名古屋大学
†Ryota Yamamoto, 名古屋大学
‡Norihiro Yoshida, 名古屋大学
§Hiroaki Takada, 名古屋大学

とは現実的ではない．状態遷移設計のリバースエンジニアリングを目的とした既存研究が存在するが，システムレベルの状態遷移モデルを抽出することを目的とした手法 [4] [5] [6] や，オブジェクト指向言語で記述されたソースコードを対象とした手法 [7] [8] [9] が主流であり，C 言語で記述されたコンパイル単位から細粒度状態遷移表を抽出するために利用することは難しい．

本研究では，複雑な条件分岐を静的に解析する手法である記号実行を利用し，C 言語で記述されたコンパイル単位から細粒度状態遷移表を抽出することを試みる．記号実行技術とは，ある変数に対して具体的な値ではなく，シンボルを割り当ててプログラムを擬似的に実行する技術のことである [10] [11]．提案手法は，ユーザ（実務者）がコンパイル単位内の変数から状態変数を指定すると，以下の手順で自動的に細粒度状態遷移表を抽出する．

1. ソースコードから，記号実行ツール TRACER [12] によって，記号実行グラフ（ソースコード中の条件とそれに対応する処理の情報をまとめたグラフ）[12] を生成する．
2. 生成された記号実行グラフから，条件と処理を抽出し，表形式にまとめる．
3. 条件と処理をまとめた表とユーザが選択した状態変数を基に，細粒度状態遷移表を抽出する．

本研究の貢献は，以下の通りである．

- 記号実行を利用して，C 言語で記述されたコンパイル単位から細粒度状態遷移表を静的解析のみで抽出する手法を考案した．
- 提案手法を実装したツールを，複数の小規模ソースファイルに適用し，細粒度状態遷移表を抽出できることを示した．

## 2 関連技術

### 2.1 記号実行

記号実行は，ある変数に対して具体的な値ではなく，シンボルを割り当ててプログラムを擬似的に実行する [10] [11]．ある実行経路についてプログラムの実行が終了したら，その実行経路中に現れた条件をまとめ，その実行経路が実行される際のパス条件を導出する．導出されたパス条件を解くことで，その実行経路が実行されうるのか，また，実行されうる場合は変数がどのような値のときに実行されるのかを知ることができる．

本研究で用いる記号実行ツール TRACER [12] について説明する．TRACER は，記号実行によって得られたすべての実行経路について，条件と処理をまとめたグラフ（以後，記号実行グラフと呼ぶ）を生成する．記号実行グラフは DOT ファイルで出力される．ここで DOT ファイルとは Dot 言語で書かれたファイルであり，Dot 言語とはプレーンテキストによってグラフを表現するための言語の一種である．記号実行グラフの例を図 1 の関数 task を用いて紹介する．図 1 のソースコードから生成された記号実行グラフのうち，関数 task 部分を図示したものが図 2 である．

記号実行グラフの各部について説明する．記号実行グラフはノードとエッジによって実行経路を表す．ノードのうち，関数の始まりと終わりのノードには色がついている．またノードのうち，ソースコード中の分岐を表すものはひし形であり，それ以外のノードは長方形である．ただし，関数の始まりのノードのみ長方形であっても分岐となることがある．ノードのラベルはソースコード中の位置を表し，ラベル中の func_task は関数 task 中のノードであることを表している．また，ノードのラベルには重複がない．分岐のあとのエッジのラベルには，そのエッジに進む際の条件が書かれており，処理が存在するエッジのラベルには，その処理が書かれている．ここで一般的な制御フローグラフでは図 1 の 3 行目の if(t==1) は $t==1$ と $t!=1$ の 2 つの経路に分岐するが，TRACER ではノード funk_task_p0#1 の後の分岐のように $t==1$, $t>1$, $t<1$ の 3 つの経路により表現される．また，点線で表されラベ

```
1   int state, out;
2   void task(int s,int t){
3     if(t==1){
4       s++;
5       if(s<10){
6         switch(state){
7         case 1:
8           out=s; break;
9         case 2:
10          out=0; state=1; break;
11        default:
12          s++; state=2;
13    }}}}
14    int main(){
15      int a,b;
16      scanf("%d", &a);
17      scanf("%d", &b);
18      task(a,b);
19      return 0;
20    }
```

**図 1　プログラム A**

ルに s と書かれたエッジはその後の実行経路が別のある実行経路と同一であることを表す.

### 2.2　細粒度状態遷移表

状態遷移設計研究 WG が提案する細粒度状態遷移表は，ソースコード中のある変数が取りうる値の集合を状態とし，ある状態からの遷移及びそれに伴う処理が実行されるために満足されなければならない条件をイベントとしている．これにより，ある状態とイベントの組み合わせのときにどのような処理や遷移が起こるかを表から参照することができる．細粒度状態遷移表の特徴は，関数内で状態遷移が起きると仮定し，関数内の条件分岐に基づいて状態遷移を表現していることである．

細粒度状態遷移表の読み方を図 3 を例に説明する．図 3 において表の上側が状態，表の左側がイベントを表している．それぞれの状態，イベントと対応するセルには処理が書かれており，処理のうち状態変数の値を変化させるものを遷移としている．また，遷移はそのことを表すために式の前に $(t)$ と書かれている．例として，状態が $state=1$ のときにイベント $t=1\,\&\,10>s$ が発生すると，$s := s+1$ と $out := s$ という処理が行われ，状態が $state=2$ のときにイベント $t=1\,\&\,10>s$ が発生すると，$s := s+1$ と $out := 0$ という処理と $state := 3$ という遷移が行われる．ここで，処理，遷移が書かれているセルの式の順番はソースコード中における時系列を表している．また，対応する処理，遷移が存在しないセルには NONE と書き表すこととしている．

## 3　提案手法

本研究で提案する細粒度状態遷移表の抽出手法を説明する．本研究の手法は，以下の 3 つの手順によって構成される．

**手順 1:** ソースコードから，TRACER によって記号実行グラフを生成する．
**手順 2:** 生成された記号実行グラフから，細粒度状態遷移表を抽出するための中間状態として条件処理表を抽出する．
**手順 3:** 条件処理表とユーザが選択した状態変数から，細粒度状態遷移表を抽出する．

次節以降で**手順 2,3** について説明する．

### 3.1　条件処理表の抽出

TRACER によって生成された記号実行グラフから，条件処理表を抽出する方法について説明する．条件処理表とは，記号実行グラフ中のすべての実行経路について条件と処理を表にまとめたものであり，状態遷移表を抽出するための中間状態として抽出する．

条件処理表の抽出は以下の手順によって行う．

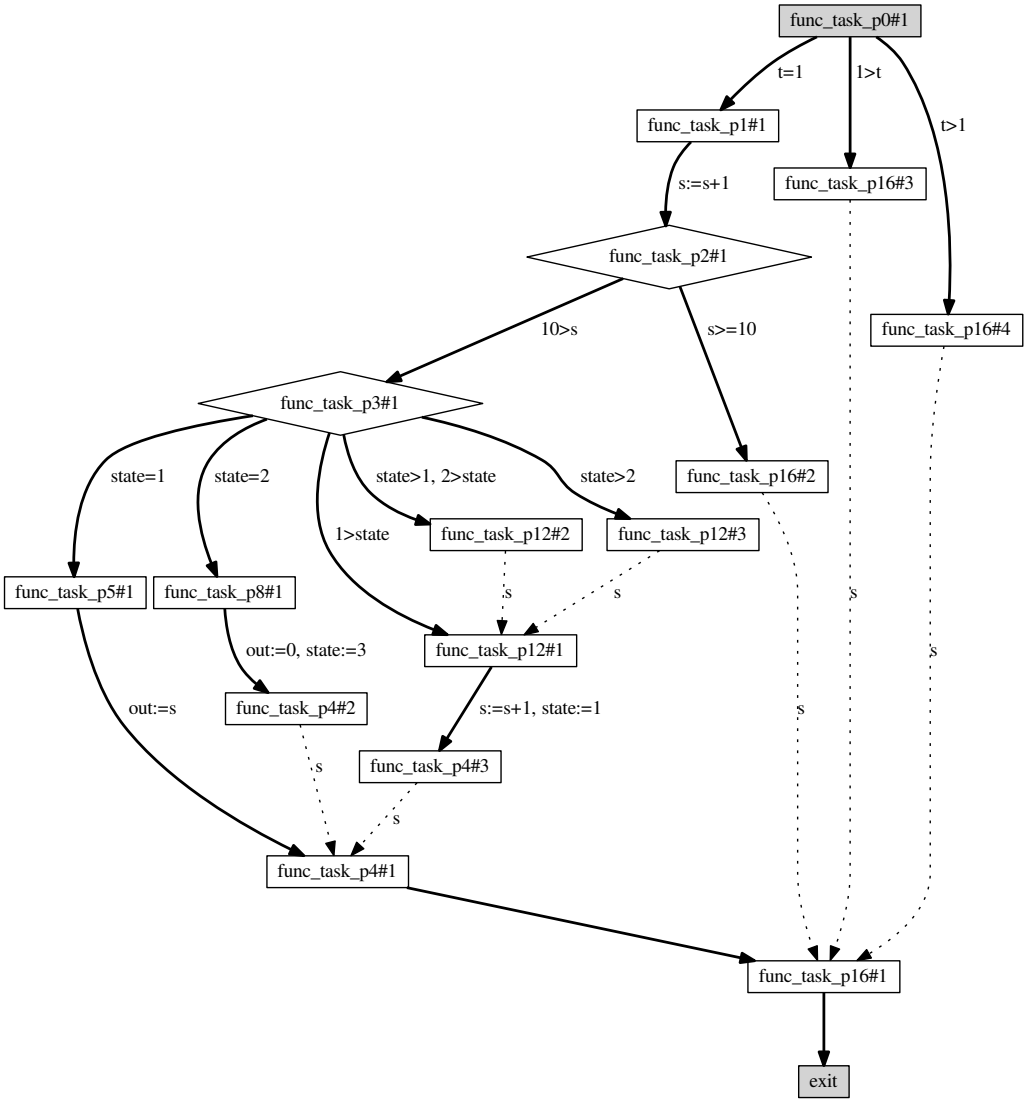


**図 2　関数 task 部分の記号実行グラフ**

**手順ア**　記号実行グラフの始点を見つけ，**手順イ**に移る．
**手順イ**　記号実行グラフをたどる，分岐にたどり着いた場合には**手順ウ**に移り，実行経路の終点にたどり着いたら**手順エ**に移る．
**手順ウ**　まだ進んでいない分岐に進み，**手順イ**に移る．
**手順エ**　たどり終えた実行経路の条件と処理を抽出し条件処理表に書き込む．未探索の経路が残っている分岐のうち最後の分岐に戻り，**手順ウ**に移る．未探索の経路が残っていない場合は記号実行グラフの探索を終了する．

ここで，始点とはそのノードへと続くエッジが存在しないノードとし，終点とはそのノードから続くエッジが存在しないノードのこととする．また，実行経路の条件が変数の型を考慮すると infeasible な経路が存在するため，このときは**手順エ**において条件と処理を条件処理表に加えないこととする．

図 1 のソースコードから生成した記号実行グラフ (図 2) を例に条件処理表の抽出

| 状態 / イベント | state=1 | state=2 | 1>state | state>2 |
|---|---|---|---|---|
| t=1&10>s | s:=s+1<br>out:=s | s:=s+1<br>out:=0<br>(t)state:=3 | s:=s+1<br>s:=s+1<br>(t)state:=1 | s:=s+1<br>s:=s+1<br>(t)state:=1 |
| t=1&s>=10 | s:=s+1 | s:=s+1 | s:=s+1 | s:=s+1 |
| 1>t | NONE | NONE | NONE | NONE |
| t>1 | NONE | NONE | NONE | NONE |

**図 3**　細粒度状態遷移表の例

**表 1**　図 1 のソースコードの条件処理表

| 条件部 | 処理部 |
|---|---|
| $t=1\ \&\ 10>s\ \&\ state=1$ | $s := s+1$<br>$out := s$ |
| $t=1\ \&\ 10>s\ \&\ state=2$ | $s := s+1$<br>$out := 0$<br>$state := 3$ |
| $t=1\ \&\ 10>s\ \&\ 1>state$ | $s := s+1$<br>$s := s+1$<br>$state := 1$ |
| $t=1\ \&\ 10>s\ \&\ state>2$ | $s := s+1$<br>$s := s+1$<br>$state := 1$ |
| $t=1\ \&\ s>=10$ | s:=s+1 |
| $1>t$ | NONE |
| $t>1$ | NONE |

方法を説明する．今回は説明の簡略化のため図 2 のノード func_task_p0#1 を記号実行グラフの始点であるとしノード exit を終点とするが，本例を対象とした TRACER の出力では main 関数の始めのノードが始点，終わりのノードが終点となる．始点 func_task_p0#1 から記号実行グラフをたどる．func_task_p0#1 は分岐ノードであるため，ここでは $t=1$ のエッジをたどるとする．その後の分岐はそれぞれ，ノード func_task_p2#1 では $10>s$ のエッジを，ノード func_task_p3#1 では $state=1$ のエッジをたどるとする．記号実行グラフの終点 exit にたどり着いたら，たどり終えた実行経路についてエッジのラベルから条件と処理を抽出し，条件処理表に書き加える．この実行経路の場合，条件は $t=1\&10>s\&state=1$ であり，処理は $s := s+1, out := s$ である．条件と処理を抽出し終えたら，最後の分岐であるノード func_task_p3#1 まで戻る．以上をすべての実行経路をたどり終えるまで繰り返す．また，処理が存在しない場合は，条件処理表の処理部に NONE と書き加えることとする．記号実行グラフのすべての実行経路をたどり終え，完成した条件処理表が表 1 である．ここで，記号実行グラフには変数の型を考慮すると実際には実行されない条件となる実行経路が存在する．今回の例ではノード func_task_p3#1 からノード func_task_p12#2 へのエッジのラベルは $state>1, 2>state$ となっているが，state は int 型であるため，この実行経路は実際には実行されない．よってこの実行経路

表 2　対象とするソースコードの概要

| | LOC | 関数の数 | 条件分岐文の数 | 変数の数 | 実行経路数 |
|---|---|---|---|---|---|
| 最大 | 139 | 3 | 21 | 8 | 26 |
| 最小 | 27 | 2 | 3 | 2 | 4 |
| 平均 | 72.33 | 2.667 | 9.222 | 4.000 | 14.56 |

の条件，処理は**手順エ**において条件処理表に加えないこととしている．

### 3.2　細粒度状態遷移表の抽出

細粒状態遷移表の抽出は以下の手順によって行う．

**手順オ**　条件処理表とユーザが選択した状態変数から，イベントと状態を決定する．

**手順カ**　条件処理表の処理部の内容を細粒状態遷移表の対応するセルに書き込む．

**手順オ**について説明する．条件処理表の条件部の中から状態変数が出現する条件式を重複のないように取り出し状態とする．状態の数だけ列を作成し，取り出した状態を列見出しとして書き込む．また，状態変数に関する条件式を取り出した後の条件式のうち，重複するものを消去し，残ったものをイベントとする．イベントの数だけ行を作成し，イベントを行見出しとして書き込む．

**手順カ**について説明する．条件処理表を上から順に見ていき，条件部に状態変数に関する条件式が含まれる場合は，条件部の条件式に対応するイベントと状態が交わるセルに処理部の内容を書き込む．条件部に状態変数に関する条件式が含まれていない場合は，その条件部と同一のイベントの行のすべてのセルに，処理部の内容を書き込む．ここで，処理部の内容を書き込む際，状態変数への代入を表す式は遷移であることがわかるように式のはじめに $(t)$ と書き加える．条件処理表のすべての行を参照した後に細粒度状態遷移表中に処理，遷移が書き込まれていないセルが存在する場合がある．この場合，そのセルには****と書き込む．

図 1 のソースコードでは，例として $state$ を状態変数とする．$state$ を状態変数とした理由は，条件分岐文で使用されており，かつその条件分岐内部で値の更新が行われている変数の 1 つであるからである．状態は $state=1$, $stete=2$, $1>state$, $state>2$ の 4 つとなる．また，イベントは $t=1\,\&\,10>s$, $t=1\,\&\,s>=10$, $1>t$, $t>1$ の 4 つとなる．これらの状態とイベントについて，状態列とイベント行を作成する．表 1 を参照しながら，各イベント，状態に対応する処理，遷移を書き込むと，図 3 のような細粒度状態遷移表が完成する．

## 4　ケーススタディ

提案手法の**手順 2,3** を自動で行うツールを開発した．本章では提案手法を正しくツールとして実装できることを確かめるためのケーススタディを行う．

開発したツールへの入力は，TRACER によって生成された記号実行グラフを表す DOT ファイルと状態変数名であり，出力は細粒度状態遷移表を表す TSV ファイルである．

対象ソースコードは，状態遷移設計研究 WG が活動の一環として作成した 9 つのソースコードである（表 2）．ソースコードはいずれも状態遷移およびイベントの発生の両者が表現されているものである．ここで，表 2 の条件分岐文の数とはソースコード中の if 文と switch 文の数であり，実行経路数とは 3.1 節において記号実行グラフをたどる際に見つかった実行経路の数である．今回の対象ソースコードに含まれる条件分岐文はすべて if 文と switch 文により構成される．また，対象ソースコード中の変数はすべて整数型である．

### 4.1 手順

ケーススタディは以下の手順で行う．

**手順 I:** ソースコードの整形，および記号実行グラフの抽出．
**手順 II-I:** 手作業による細粒度状態遷移表の抽出．
**手順 II-II:** ツールによる細粒度状態遷移表の抽出．
**手順 III:** 手作業及びツールによって抽出した細粒度状態遷移表の比較．

はじめに，**手順 I** で TRACER が解析できるように対象のソースコードの整形を行う．そして，整形を行ったソースコードに対して TRACER を実行して記号実行グラフを生成する．

つぎに，**手順 II-I** で記号実行グラフから手作業で細粒度状態遷移表を抽出する．DOT ファイルを PDF ファイルに変換し，記号実行グラフを図式化する．図式化した記号実行グラフから，3 節の手法によって手作業で細粒度状態遷移表を作成する．

つづいて，**手順 II-II** で開発したツールによって記号実行グラフから細粒度状態遷移表を抽出する．本研究で開発したツールの入力として TRACER によって生成された記号実行グラフを与え，細粒度状態遷移表を抽出する．また，状態変数はソースコード中の変数のうち，条件分岐文で使用されており，かつその条件分岐内部で値の更新が行われているものを各ソースコードにつき 1 つずつ指定する．出力された TSV ファイルを表計算ソフトで開き，内容を確認する．

最後に，**手順 III** でツールによって抽出した細粒度状態遷移表と手作業で抽出した細粒度状態遷移表を見比べ，イベント，状態，処理，遷移について内容が同一であるか確認する．

### 4.2 結果

ケーススタディの結果について説明する．まず，開発したツールによって細粒度状態遷移表を自動抽出した結果を説明する．すべての対象ソースコードについて，開発したツールによって記号実行グラフから細粒度状態遷移表を抽出することができた．このとき，対象ソースコードに#define によって定義された定数は実際の値に書き換え，printf 関数，time 関数はコメントアウトしたうえで記号実行木を生成した．抽出された細粒度状態遷移表に形式上の間違いはなかった．また，自動抽出したものと手作業で作成したものとで，イベントと状態を見比べた結果，イベントや状態の順番が違うものはあったが状態遷移用の内容はすべての対象ソースコードについて一致していた．

例として，対象ソースコードのうちの 1 つから抽出した記号実行グラフ (図 4) とツールによって抽出した細粒度状態遷移表 (図 5) を紹介する．図 5 の細粒度状態遷移表には形式的な間違いはなく，手作業で抽出した細粒度状態遷移表と一致していた．

### 4.3 考察

ケーススタディの結果から，開発したツールによってすべての対象ソースコードの記号実行グラフから正しく細粒度状態遷移表を抽出できることがわかった．開発したツールによる細粒度状態遷移表の抽出はいずれも数秒で行われ，手作業の場合と比べ細粒度状態遷移表を抽出するコストが減少している．また，抽出した細粒度状態遷移表からソースコード中の条件と処理についての情報を知ることができ，より複雑なソースコードにも対応できればソースコードの振舞いを知るための手助けとなりうる．

本研究では記号実行による生成物から状態遷移表を抽出している．制御フローグラフからすべての実行される経路を列挙し，かつ各経路が実行される条件を導出することは困難である．一方記号実行による生成物では，これらがすでに求められており実行経路の情報を得るのが容易である．また複数の条件分岐が登場し，実行経路をたどる際の条件が複雑になるような場合にも，その実行経路が実行されうるのか，実行されるならばどのような条件のときに実行されるのかを導くことができる．

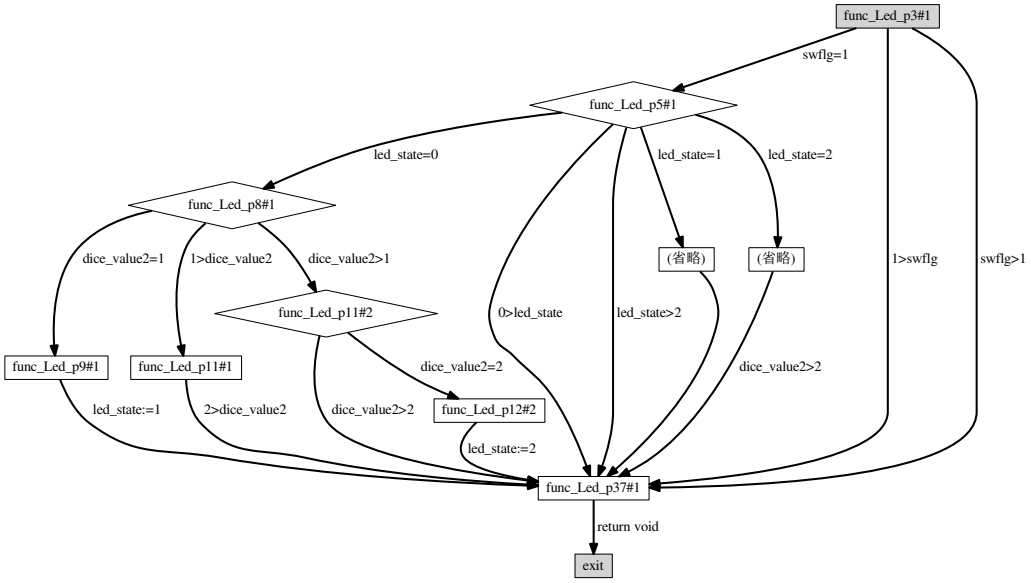


図 4　対象ソースコードのうちの 1 つから生成した記号実行グラフ

| | led_state=0 | led_state=1 | led_state=2 | 0>led_state | led_state>2 |
|---|---|---|---|---|---|
| swflg=1&dice_value2=1 | (t)led_state:=1 | (t)led_state:=2 | (t)led_state:=0 | **** | **** |
| swflg=1&1>dice_value2&2>dice_value2 | NONE | NONE | NONE | **** | **** |
| swflg=1&dice_value2>1&dice_value2=2 | (t)led_state:=2 | (t)led_state:=0 | (t)led_state:=1 | **** | **** |
| swflg=1&dice_value2>1&dice_value2>2 | NONE | NONE | NONE | **** | **** |
| swflg=1 | **** | **** | **** | NONE | NONE |
| 1>swflg | NONE | NONE | NONE | NONE | NONE |
| swflg>1 | NONE | NONE | NONE | NONE | NONE |

図 5　図 4 から自動で抽出した細粒度状態遷移表

今回のケーススタディでは，提案手法のツール化が可能であることを確かめたが，その有用性については確かめていない．今後被験者に人手とツールの両方で状態遷移表を抽出してもらい，状態遷移表を抽出するまでの時間や抽出した状態遷移表の正確性を比較するような実験を行う必要がある．またその他にも，記号実行を用いることの有用性があるか確かめるため，今後記号実行を用いない手法との比較も行う必要がある．

## 5　関連研究

状態遷移図の自動抽出を目的とした手法がいくつか提案されているが，そのほとんどは実行時情報を基に生成する手法である [13] [14]．本研究で提案している手法は記号実行を利用しており，実機環境がなくとも適用することができる．組込みシステムでは，実行時情報を取得するためのテスト環境を構築することにコストがかかる場合があり，実行時情報を必要とする手法を適用することが現実的ではない場合がある．Walkinshaw らは，抽象実行を用いてソースコードから状態遷移図を抽出す

る手法を提案している [4]．彼らの手法が抽出する状態遷移図は，メソッド呼出し文や例外処理の発生による遷移のみを扱う粒度の大きいものである．Walkinshaw らの手法では関数を状態遷移点としており，システム全体の状態遷移図を抽出している [5] が，本研究では細粒度状態遷移表を定義しており，関数呼び出しよりも粒度を小さく表現するために関数中の条件分岐を扱っている．

さらに，Walkinshaw らは，実行トレースから状態遷移表を自動的に生成するツールを提案している [6]．動的に状態遷移表を抽出しているため，実行されなかったパスは考慮されない．また，機械学習を用いて状態遷移の条件を導出しているため，実際のトレースに対応していない値が閾値となる場合がある．

また，オブジェクト指向言語によって記述されたソースコードに対してリバースエンジニアリングを行い，状態遷移モデルを抽出する手法がある [7] [8] [9]．文献 [7] では，以下の 3 つを用いて状態遷移モデルのリバースエンジニアリングを行う．

- 抽象的な状態値のドメイン
- 具体的な状態値から抽象的な状態値への写像
- 与えられたプログラム中のすべてのプリミティブな命令の抽象的な意味

本稿の提案手法では，これらの制約が定義されていなくとも状態遷移表を抽出することができる．文献 [8] では，Java PathFinder による記号実行を利用して状態遷移モデルの抽出を試みている．この研究では，状態変数の選択を記号実行の結果から自動的に実施している．そのため，ユーザが注視したい変数を指定することができず，効果的な状態遷移表を抽出できると断定できない．また，文献 [9] では，C++ で記述されたコードに対して記号実行による状態遷移モデルの抽出を試みている．この研究では，イベントを関数呼び出しとしており，関数呼び出しをトリガをとして状態遷移が実行される．

我々の研究グループでは，状態変数の定義を満たす変数を提示する対話型 UI が実装している [15]．この対話型 UI を本研究のツールに利用すれば，本研究においてユーザが状態変数を選択する際のコストを小さくすることができると考えられる．

## 6 まとめ

本研究では，状態遷移設計を含むソースコードから記号実行を用いて細粒度状態遷移表を抽出する手法を提案した．また，TRACER の生成物から細粒度状態遷移表を抽出するツールを開発し，ケーススタディを行うことで開発したツールにより細粒度状態遷移表を抽出できることを確認した．開発したツールはソースコードの振舞いを知るための手助けとなりうると考えられる．記号実行による生成物から状態遷移表を抽出することで，制御フローグラフから状態遷移表を抽出する場合と比べ，実行経路の導出が容易になると考えられる．

本研究は今後多言語対応の実現を考えている．また，状態遷移設計を含むコンパイル単位の自動識別や，状態遷移設計を含むコード片の自動抽出も今後の課題である．ループ文を含むコンパイル単位は，全体を 1 つの細粒度状態遷移表に変換できないことが多いと考えられるため，細粒度状態遷移表に変換可能な部分を自動的に特定し，ユーザに提示する手法を考案したいと考えている．

本研究では，一部のソースコードから状態遷移表を抽出できることを確認できた．今後割込み処理やポインタ関数等，組込みシステム向けのプログラムに多くあらわれる記述が存在するソースコードを中心に TRACER が記号実行木を生成できるかを確かめる必要がある．さらには，TRACER 以外の記号実行ツールについてもさらなる調査を進め，より適切な記号実行ツールの選定を行う必要がある．また開発したツールにおいても，どのように状態遷移表を抽出すべきか議論が必要な場合が存在する．ソースコード中に for 文等のループが存在する場合や，複数の変数を状態変数に指定したい場合がこれに該当する．

**謝辞** 青木 奈央氏をはじめとする組込みシステム技術協会状態遷移設計研究 WG の関係者には，本研究について多くの助言をいただきました．ここに謝意を記します．本研究は，JSPS 科研費 JP16K16034 の助成を受けたものです．

## 参考文献

[ 1 ] 竹田彰彦．状態遷移表によるレガシーコードの蘇生術．日経テクノロジーオンライン, 2016. http://techon.nikkeibp.co.jp/article/COLUMN/20150519/418967/.
[ 2 ] 高田広章．組込みシステム開発技術の現状と展望．情報処理学会論文誌, Vol. 42, No. 4, pp. 930–938, 2001.
[ 3 ] 鵜飼敬幸. TOPPERS/SSP への組込みコンポーネントシステム適用における設計情報の可視化と抽象化．第 9 回クリティカルソフトウェアワークショップ (WOCS2), 2011. https://www.ipa.go.jp/files/000005299.pdf.
[ 4 ] Neil Walkinshaw, Kirill Bogdanov, Shaukat Ali, and Mike Holcombe. Automated discovery of state transitions and their functions in source code. *Software Testing, Verification & Reliability*, Vol. 18, No. 2, pp. 99–121, 2008.
[ 5 ] Sarfraz Khurshid, Corina S Păsăreanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *Proc. TACAS*, pp. 553–568. Springer, 2003.
[ 6 ] Neil Walkinshaw, Ramsay Taylor, and John Derrick. Inferring extended finite state machine models from software executions. *Empirical Software Engineering*, Vol. 21, No. 3, pp. 811–853, 2016.
[ 7 ] Paolo Tonella and Alessandra Potrich. *Reverse engineering of object oriented code*. Springer, 2005.
[ 8 ] Tamal Sen and Rajib Mall. Extracting finite state representation of java programs. *Software & Systems Modeling*, Vol. 15, No. 2, pp. 497–511, 2016.
[ 9 ] David Kung, Nimish Suchak, Jerry Gao, Pei Hsia, Yasufumi Toyoshima, and Chris Chen. On object state testing. In *Proc. COMPSAC 94*, pp. 222–227, 1994.
[10] James C King. Symbolic execution and program testing. *Communications of the ACM*, Vol. 19, No. 7, pp. 385–394, 1976.
[11] 上原忠弘. シンボリック実行を活用した網羅的テストケース生成. *FUJITSU*, Vol. 66, No. 5, pp. 34–40, 2015.
[12] Joxan Jaffar, Vijayaraghavan Murali, Jorge A Navas, and Andrew E Santosa. TRACER: A symbolic execution tool for verification. In *Proc. of CAV 2012*, pp. 758–766. Springer, 2012.
[13] Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. Inferring state-based behavior models. In *Proc. WODA*, pp. 25–32, 2006.
[14] Tao Xie and David Notkin. Automatic extraction of object-oriented observer abstractions from unit-test executions. In *Proc. ICFEM*, pp. 290–305, 2004.
[15] 山本椋太, 吉田則裕, 竹田彰彦, 舘伸幸, 高田広章. 組込みソフトウェアを対象とした状態遷移表抽出手法. 電子情報通信学会技術研究報告, Vol. 116, No. 127, pp. 13–18, 2016.