

状態遷移モデル抽出における 記号実行ツール KLEE および TRACER の比較評価

清水 貴裕^{1,a)} 山本 椋太¹ 吉田 則裕¹ 高田 広章¹

概要: 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。レガシー化したソースコードの振舞いを理解するため、ソースコードと対応した状態遷移表の抽出が提案されている。著者らはこれまでも記号実行ツール TRACER の出力を用いて、C 言語で記述されたソースコードから状態遷移表を抽出する手法について研究を行ってきた。しかし、従来手法ではソースコード中の入出力関数の呼び出しや define 命令によって定義された定数を抽出できない等の課題が存在した。こうした課題の軽減のため、本研究では記号実行ツール KLEE による出力を用いて状態遷移表の抽出を試みた。そして、状態遷移モデルの抽出における、記号実行ツール KLEE および TRACER の比較評価を行った。比較評価の結果、KLEE を利用したほうが入出力関数の呼び出しや define 等の命令を抽出できる可能性が高いことが分かった。一方で、KLEE を利用した場合には、TRACER を利用した場合と比べて、ソースコード中の各実行経路の分岐条件、およびそれに伴う処理の情報を抽出するための実装コストが高いことが分かった。

Comparative Evaluation of Symbolic Execution Tools KLEE and TRACER for Extracting a State Transition Model

TAKAHIRO SHIMIZU^{1,a)} RYOTA YAMAMOTO¹ NORIHIRO YOSHIDA¹ HIROAKI TAKADA¹

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。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 [11]。これらの背景から、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 [10]。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 [13]。

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について研究を行っている [11]。その一環として、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の状態遷移表の抽出が提案されている。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。また、状態遷移設計研究 WG の提案している状態遷移表は、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レベルの状態遷移を表しており、より粒度が細かいと言える。

状態遷移設計研究 WG では、コンパイル単位から手作業で状態遷移表を抽出する手順を提案してきたが、限られた

¹ 名古屋大学
Nagoya University, Nagoya, Aichi 464-8603, Japan
^{a)} takahiro915@ertl.jp

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状態遷移表を抽出するこ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の抽出を目的とした手法 [14] 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 [3][7][12] が主流であり、C 言語で記述されたコンパイル単位から状態遷移表を抽出するために利用することは難しい。

こうした課題を軽減するため、著者らは記号実行によって得られるコンパイル単位中の分岐条件、およびそれに伴う処理についての情報から、状態遷移表を抽出する手法について検討している。これまでも著者らは記号実行ツール TRACER[2] を用いて、C 言語から状態遷移表を抽出する手法について検討してきた [8][9]。従来手法では、TRACER による出力から実行経路ごとの分岐条件とそれに伴う処理の情報を抽出し、抽出した分岐条件と処理の情報、およびユーザが選択した状態変数から、状態遷移表を抽出した。しかし、従来手法ではソースコード中の入出力関数の呼び出しや define 命令によって定義された定数を抽出できない等の課題が存在した。

そこで本研究では、記号実行ツール KLEE による出力からソースコード中の分岐条件とそれに伴う処理を抽出し、状態遷移表を抽出することを試みた。そして、状態遷移モデルの抽出における、記号実行ツール KLEE および TRACER の比較評価を行った。比較評価を行った結果、KLEE を利用したほうが printf 関数等の入出力関数の呼び出しや define 等の命令を抽出できる可能性が高いことが分かった。一方で、TRACER を利用したほうが各実行経路に対応する分岐条件、およびそれに伴う処理の情報を抽出するための実装コストが低いことが分かった。

2. 記号実行ツールによる状態遷移表の抽出

本章では、まず 2.1 節において本研究で扱う状態遷移表 [11][17] について説明し、次に 2.2 節で記号実行ツール TRACER[2] を用いた状態遷移表の抽出手法について説明する。最後に、2.3 節で本研究で用いるもう 1 つの記号実行ツールである KLEE[1] について説明する。

2.1 状態遷移表

本研究では状態遷移設計研究 WG が提案する状態遷移表を参考にする。状態遷移設計研究 WG が提案する状態遷移表は、ソースコード中のある変数（状態変数）が取りうる値の集合を状態としている [11][17]。また、ある状態からの遷移およびそれに伴う処理が実行されるために満たされなければならない分岐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。本研究における状態遷移表の特徴は、関数内で状態遷

	state=1	state=2	1>state	state>2
t=1&10>s	s:=s+1 out:=s	s:=s+1 out:=0 (t)state:=3	s:=s+1 s:=s+1 (t)state:=1	s:=s+1 s:=s+1 (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 1 状態遷移表の例

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。

本研究における状態遷移表の例を図 1 に示す。本研究における状態遷移表では、表の列見出しが状態、表の行見出しがイベントを表している。本研究における状態とは、ユーザがソースコード中の変数の中から選択した状態変数がとりうる値の集合のことである。またイベントとは、ソースコードの各実行経路が実行される際の条件文の組み合わせから状態変数に関する条件文を取り除いたものである。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。

2.2 記号実行ツール TRACER を用いた状態遷移表の抽出

これまでも著者らは記号実行ツール TRACER[2] を用いて C 言語で記述されたソースコードから状態遷移表を抽出する手法について研究してきた [8][9]。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、分岐条件とそれに伴う処理をまとめたグラフ（以降、TRACER グラフ）を生成する。TRACER 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、ひし形で表されたものがソースコード中の条件分岐である。条件分岐のあとのエッジのラベルには、そのエッジへと進む際の分岐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。著者らの研究の手法は以下の手順 1 から手順 3 によって構成される。ここで、条件処理表とは、TRACER グラフ中のすべての実行経路について分岐条件とそれに伴う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出力である。

手順 1: TRACER によって、解析対象のソースコードから TRACER グラフを生成する。

手順 2: 生成された TRACER グラフから、ソースコード中の分岐条件とそれに伴う処理の情報を抽出し、条件処理表にまとめる。

手順 3: 条件処理表とユーザが選択した状態変数から、状態遷移表を抽出する。

手順 1 について説明する。解析対象の C 言語で記述さ

```

1  #include <stdio.h>    17      state = 3;
2  int state, out;       18      printf("state changed\n");
3                          19      break;
4  void task(){          20      default:
5      int s,t;           21      s++;
6      scanf("%d", &s);   22      state = 1;
7      scanf("%d", &t);   23      printf("state changed\n");
8      if(t == 1){       24      }
9      s++;              25      }
10     if(s < 10){        26      }
11     switch(state){     27      }
12     case 1:            28
13         out = s;       29     int main(){
14         break;         30         task();
15     case 2:            31         return 0;
16         out = 0;       32     }

```

図 2 ソースコード A

表 1 図 3 の TRACER グラフから抽出した条件処理表

条件部	処理部
t=1&10>s&state=1	s:=s+1 out:=s
t=1&10>s&state=2	s:=s+1 out:=0 state:=3
t=1&10>s&1>state	s:=s+1 s:=s+1 state:=1
t=1&10>s&state>2	s:=s+1 s:=s+1 state:=1
t=1&s>=10	s:=s+1
1>t	NONE
t>1	NONE

れたソースコードに対して TRACER を実行すると、ソースコード中の分岐条件と処理をまとめた TRACER グラフが生成される。TRACER によって図 2 のソースコードから生成した TRACER グラフのうち、ソースコード中の分岐条件と処理に関する部分を抜粋したものが図 3 である。

手順 2 について説明する。手順 1 において生成した TRACER グラフ中のすべての実行経路をたどり、条件処理表にまとめる。条件処理表の各行は各実行経路を表し、各実行経路ごとにその実行経路をたどる際の分岐条件とそれに伴う処理を参照できる。図 3 の TRACER グラフから抽出した条件処理表が表 1 である。

手順 3 について説明する。条件処理表の条件部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。

条件処理表の各行を 1 行目から順に読み込み、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。表 1 の条件処理表から、変数 state を状態変数として抽出した状態遷移表が図 1 である。

著者らの研究では、以上の手順によって一部のソースコードから状態遷移表を抽出した。しかし従来手法では、ソースコードから TRACER グラフを生成する前に、define 命令によって定義された定数を実際の値に書き換える必要があることや、printf 関数等の入出力関数の呼び出しを抽出できないといった課題があった。また、ソースコード中のループやポインタ変数等を含むソースコードに対する解析方法について、十分に検討できていないことも課題である。

2.3 記号実行ツール KLEE

KLEE[1] は LLVM ビットコードを読み取り、記号実行を行うツールである。KLEE は解析対象として C 言語を想定している。KLEE による記号実行を行うために必要な記述を挿入した C 言語で記述されたソースコードから、Clang^{*1}によって LLVM ビットコードのファイルを生成したうえで、KLEE の実行を行う。KLEE による記号実行によって、解析対象であるソースコードにおいて実行される実行経路の数や、その実行経路をたどる場合のシンボルを割り当てた変数の具体値等を知ることができる。

KLEE は記号実行を行う際、条件分岐によって発生する各実行経路を、木構造のデータによって保持している。著者らは、このデータが 2.2 節における TRACER グラフに相当すると考え、これまでもこのデータの出力を試みてきた。しかし、KLEE はテストケース生成を目的としたツールであり、解析対象のソースコードの振舞いを把握するためのインタフェースは用意されていなかった。

上記の木構造のデータ中の分岐条件の依存関係、および分岐条件のソースコード中での行番号についての情報を出力するオプション “-output-exectree” が開発された。このオプションは、KennyMacheka^{*2}によって KLEE の GitHub リポジトリからフォークされたブランチ^{*3}において開発された。このオプションは、2019 年 11 月 12 日現在では KLEE のメインブランチ^{*4}に実装されていない。本研究ではこのオプションを利用し、KLEE による出力から状態遷移表を抽出する。

3. 比較評価

3.1 KLEE を用いた状態遷移表の抽出手順

KLEE を用いた状態遷移表の抽出手順について説明す

*1 <https://clang.llvm.org/>

*2 GitHub 上のアカウント名

*3 <https://github.com/KennyMacheka/klee>

*4 <https://klee.github.io/>

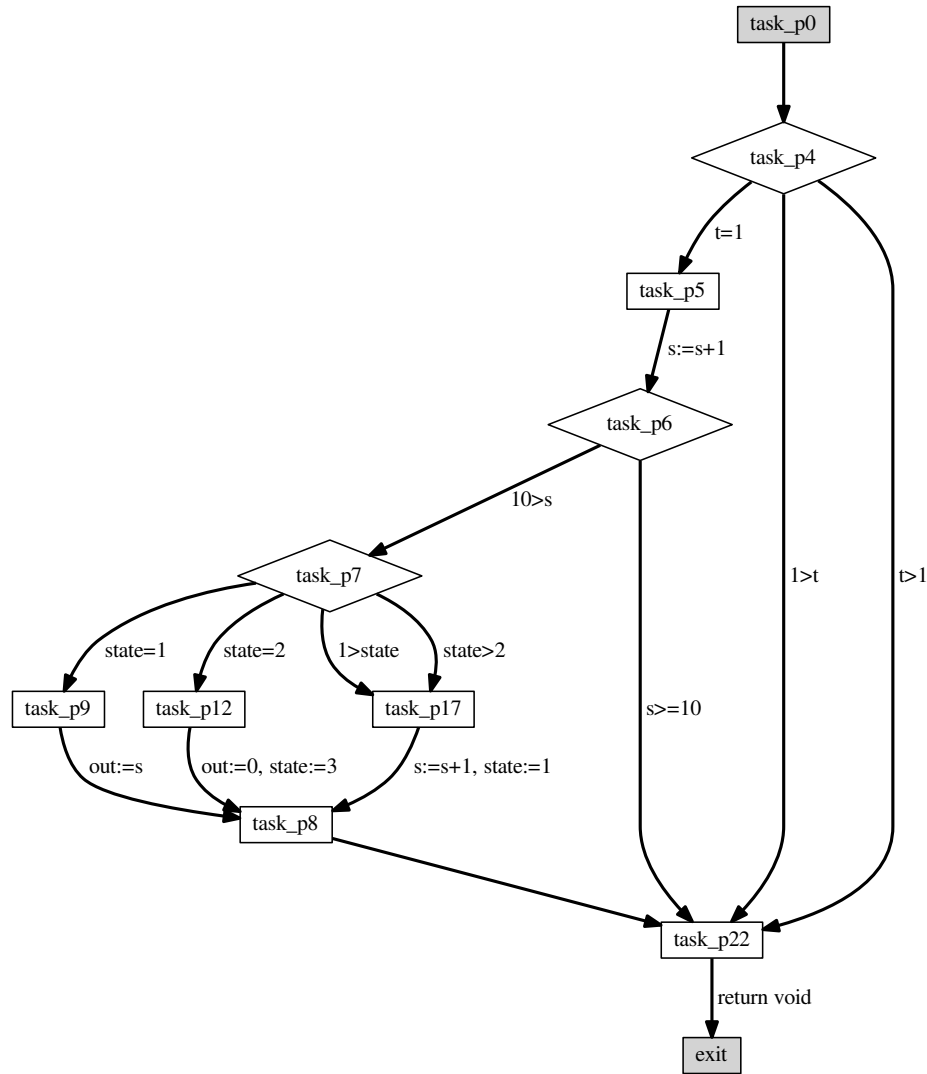


図 3 図 2 から生成した TRACER グラフ (抜粋)

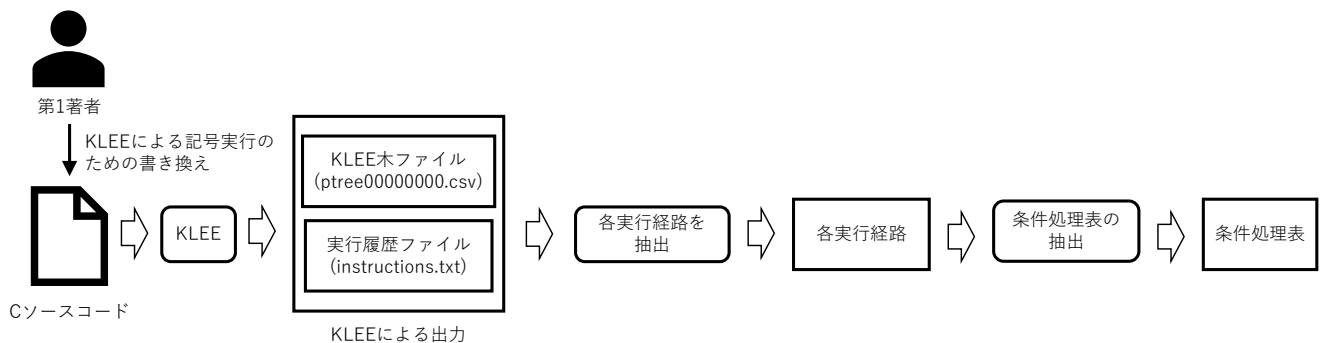


図 4 KLEE による出力を用いた条件処理表の抽出手順

る。KLEE を用いて解析対象のソースコードから 2.2 節の条件処理表を抽出する手順の概要は図 4 のとおりである。ここで各実行経路は、その実行経路が実行される際にたどるソースコード中の行番号によって表される。条件処理表が抽出できたら、第 1 著者が指定した状態変数について 2.2 節の手順 3 と同様に状態遷移表を抽出する。以下で図 4 について詳細を説明する。なお本手順において、ソースコー

ドの書き換え部分以外はツールとして実装する。

3.1.1 ソースコードの書き換え

はじめに以下の変更 A から変更 D について、第 1 著者が手作業でソースコードに変更を加える。図 2 のソースコードに対してこれらの変更を行ったソースコードが図 5 である。

変更 A: klee/klee.h をインポートする (図 5 の 2 行目)。

```

1  #include <stdio.h>
2  #include "klee/klee.h"
3  #define ON 1
4  int state, out;
5
6  void task(){
7      int t,s;
8      klee_make_symbolic(&state, sizeof(state), "state");
9      klee_make_symbolic(&t, sizeof(t), "t");
10     klee_make_symbolic(&s, sizeof(s), "s");
11     //scanf("%t", &t);
12     //scanf("%s", &s);
13     if(t == ON){
14         s++;
15         if(s < 10){
16             if(state==1){
17                 out = s;
18             }
19             else if(state==2){
20                 out = 0;
21                 state = 3;
22                 printf("state changed\n");
23             }
24             else{
25                 s++;
26                 state = 1;
27                 printf("state changed\n");
28             }
29         }
30     }
31 }
32
33 int main(){
34     task();
35     return 0;
36 }

```

図 5 ソースコード A'

変更 B: klee_make_symbolic 関数の呼び出しを挿入する (図 5 の 8~10 行目). 引数には記号実行の際に条件分岐を考慮したい変数をとる.

変更 C: scanf 関数の呼び出しをコメントアウトする (図 5 の 11,12 行目).

変更 D: switch 文を if 文に書き換える (図 5 の 16~28 行目).

これらのうち, 変更 A, 変更 B は KLEE による記号実行を行うための変更であり, 変更 C を行うのは KLEE の実行が現実的な時間で終了しなくなるためである. また変更 D を行うのは, これ以降の手順で switch 文による条件分岐の解析方法を確立できていないためである. なお, 図 5 のソースコードでは, define 命令による定数の定義に対応しているか確認するため, 変数 t に関する条件文において, define 命令で定義した定数を用いている.

3.1.2 KLEE の実行

つぎに, 変更を行ったソースコードに対して KLEE を実行する. 本研究では KLEE 実行の際, オプションとして “-output-exectree”, “-debug-printf-instructions=src:file”, および “-search=dfs” の 3 つを指定する. “-output-exectree”

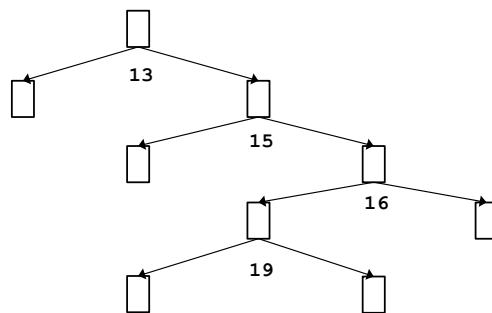


図 6 KLEE 木

オプションを指定することで, 解析対象のソースコードに対して記号実行を行った際に通過した条件分岐の依存関係を表したグラフ (以降, KLEE 木) を表す csv ファイル ptree00000000.csv (以降, KLEE 木ファイル) が得られる. 図 5 のソースコードから得られた KLEE 木ファイルが表す KLEE 木が図 6 である. 条件分岐の下に数字はソースコード中の行番号を表している. 図 6 において条件分岐の左側に進むエッジはソースコードの該当部分が偽である場合を, 右側に進むエッジは真である場合を表している. また, “-debug-printf-instructions=src:file” オプションを指定することで, 記号実行において実行された命令列を表す instructions.txt (以降, 実行履歴ファイル) が得られる. なお, 記号実行の探索方法について, “-search=dfs” オプションによって探索方法を深さ優先探索に指定する.

3.1.3 実行経路の抽出

KLEE 木ファイル, および実行履歴ファイルが出力されたら, これらのファイルから各実行経路を抽出する. 各実行経路は, 以下の手順によって抽出する. ここで, 本研究ではソースコードの終端については第 1 著者が既知であるものとする.

手順ア 実行履歴ファイルの行番号のうち, KLEE 木ファイルに示されている条件分岐, およびソースコードの終端に印をつけ, 手順イに進む.

手順イ 実行履歴ファイル中の行番号を順に見ていき, 条件分岐にたどり着いたら今回の手順イの実行においてたどった行番号をリストに保存し, 手順ウに進む. ソースコードの終端にたどり着いたら, 今回の手順イの実行においてたどった行番号をリストに保存し, 手順エに進む.

手順ウ この条件分岐は偽に進むとリストに記憶し, 手順イに進み, 実行履歴ファイルの続きを見ていく.

手順エ 現在リストに保存されているすべての行番号, および条件分岐の真偽の情報を実行経路の 1 つとして保存し, 手順オに進む.

手順オ 最後に保存したリストの情報を削除し, その直前の条件分岐で偽に進んでいる場合には, 真に進むと記憶し直し, 手順イに進み, 実行履歴ファイルの続きを

```
int main(){
    task();
}
void task(){
    int t,s;
    klee_make_symbolic(&state, sizeof(state), "state");
    klee_make_symbolic(&t, sizeof(t), "t");
    klee_make_symbolic(&s, sizeof(s), "s");
    #TRUE# if(t == ON){
        s++;
    }
    #TRUE# if(s < 10){
    #FALSE# if(state==1){
    #TRUE# else if(state==2){
        out = 0;
        state = 3;
        printf("state changed\n");
    }
}
return 0;
```

図 7 ソースコード中の記述に変換した実行経路

見ていく。直前の条件分岐が真である場合には、その条件分岐についての情報もリストから削除し、手順 α をもう一度行う。リストが空になった場合には、実行情報の抽出を終了する。

3.1.4 条件処理表の抽出

ソースコード中の各実行経路を抽出できたら、各実行経路の分岐条件およびそれに伴う処理を抽出するため、行番号をソースコード中の記述に変換する。図 5 のソースコード中の実行経路の 1 つについて、ソースコードの記述に変換したものが図 7 である。#TRUE#, #FALSE# はその行に含まれる分岐条件の真偽を表している。

このように各実行経路がたどるソースコードの記述を得られたら、各実行経路について分岐条件とそれに伴う処理を抽出し、条件処理表にまとめる。条件部については以下の手順で抽出する。

手順 i ソースコード中の記述に変換した実行経路の各行を順に読み込み、#TRUE# で始まる行にたどり着いたら、手順 ii に進む。#FALSE# で始まる行にたどり着いたら、手順 iii に進む。実行経路の終端にたどり着いたら、条件部の抽出を終了する。

手順 ii 条件部にすでにその他の条件式が書かれている場合には、条件部の最後にアンパサンドを書き足す。実行経路中の条件式の文字列を、条件部に書き足す。手順 i に進み、実行経路を続きからたどる。

手順 iii 条件部にすでにその他の条件式が書かれている場合には、条件部の最後にアンパサンドを書き足す。エクスクラメーションマークのあとに、実行経路中の条件式に括弧を付与し、条件部に書き足す。手順 i に進み、実行経路を続きからたどる。

処理部については以下の手順で抽出する。

手順 α ソースコード中の記述に変換した実行経路の各行を順に読み込み、変数への代入、および外部関数の呼び出しのいずれかであれば手順 β に進む。実行経路の終端にたどり着いたら、処理部の抽出を終了する。

手順 β 処理部に、行頭のタブ文字を削除した実行経路の文字列を書き足す。この際、すでにその他の処理が書かれている場合には、改行してから書き足す。手順 α に進み、実行経路を続きからたどる。

こうして、各実行経路について、分岐条件とそれに伴う処理が抽出され、条件処理表にまとめられる。

3.2 KLEE を用いた状態遷移表の抽出結果

3.1 節の手順によって図 5 のソースコードから抽出した条件処理表が表 2 である。条件部には変数 t, s, state についての分岐条件が抽出された。また処理部には、変数 s のインクリメント、変数 state, out への値の代入、および、klee_make_symbolic 関数、printf 関数の呼び出しが抽出された。この条件処理表において、4 行目が図 7 の実行経路を表している。

この条件処理表から、状態変数に変数 state を選択して抽出した状態遷移表が図 8 である。状態として、!(state==1)&!(state==2), !(state==1)&state==2, state==1 の 3 つが抽出された。またイベントとして、!(t==ON), t==ON&!(s<10), t==ON&s<10 の 3 つが抽出された。

3.3 KLEE と TRACER の比較

状態遷移モデルの抽出における、KLEE および TRACER の違いを 4 つの観点から考察する。各観点からの考察を 3.3.1 節から 3.3.4 節において述べる。

3.3.1 入出力関数は抽出可能か？

KLEE を用いて状態遷移表を抽出した場合にはソースコード中で printf 関数の呼び出しを抽出することができた。一方で、TRACER を用いた際には printf 関数の呼び出しを抽出することができなかった。このことから、KLEE を用いた場合の方が、解析対象のソースコード中の入出力関数の呼び出しを抽出できる可能性が高いと考えられる。しかし、KLEE の場合も klee_make_symbolic 関数で指定した変数を printf 関数の引数に取ると KLEE 実行時にエラーメッセージ^{*5}が表示される。またソースコード中に scanf 関数の呼び出しが含まれる場合は、KLEE の実行が現実的な時間で終了しない。

3.3.2 define 命令による定数定義は解析可能か？

KLEE を用いて状態遷移表を抽出した場合にはソースコード中の define 命令による定数定義をそのまま解析することができた。一方で、TRACER を用いた際には define 命令による定数の定義については、該当する定数を実際の値に書き換えを行ってから TRACER の実行を行っている。この点から、KLEE を用いた場合の方が define 等の命令に対応できる可能性が高いと考えられる。

*5 external call with symbolic argument: printf

表 2 3.1 節の手順によって図 5 のソースコードから抽出した条件処理表

条件部	処理部
!(t==ON)	klee_make_symbolic(&state, sizeof(state), "state"); klee_make_symbolic(&t, sizeof(t), "t"); klee_make_symbolic(&s, sizeof(s), "s");
t==ON&!(s<10)	klee_make_symbolic(&state, sizeof(state), "state"); klee_make_symbolic(&t, sizeof(t), "t"); klee_make_symbolic(&s, sizeof(s), "s"); s++;
t==ON&s<10&!(state==1)&!(state==2)	klee_make_symbolic(&state, sizeof(state), "state"); klee_make_symbolic(&t, sizeof(t), "t"); klee_make_symbolic(&s, sizeof(s), "s"); s++; s++; state = 1; printf("state changed\n");
t==ON&s<10&!(state==1)&state==2	klee_make_symbolic(&state, sizeof(state), "state"); klee_make_symbolic(&t, sizeof(t), "t"); klee_make_symbolic(&s, sizeof(s), "s"); s++; out = 0; state = 3; printf("state changed\n");
t==ON&s<10&state==1	klee_make_symbolic(&state, sizeof(state), "state"); klee_make_symbolic(&t, sizeof(t), "t"); klee_make_symbolic(&s, sizeof(s), "s"); s++; out = s;

	!(state==1)&!(state==2)	!(state==1)&state==2	state==1
!(t == ON)	klee_make_symbolic(&state, sizeof(state), "state"); klee_make_symbolic(&t, sizeof(t), "t"); klee_make_symbolic(&s, sizeof(s), "s");	klee_make_symbolic(&state, sizeof(state), "state"); klee_make_symbolic(&t, sizeof(t), "t"); klee_make_symbolic(&s, sizeof(s), "s");	klee_make_symbolic(&state, sizeof(state), "state"); klee_make_symbolic(&t, sizeof(t), "t"); klee_make_symbolic(&s, sizeof(s), "s");
t == ON&!(s < 10)	klee_make_symbolic(&state, sizeof(state), "state"); klee_make_symbolic(&t, sizeof(t), "t"); klee_make_symbolic(&s, sizeof(s), "s"); s++;	klee_make_symbolic(&state, sizeof(state), "state"); klee_make_symbolic(&t, sizeof(t), "t"); klee_make_symbolic(&s, sizeof(s), "s"); s++;	klee_make_symbolic(&state, sizeof(state), "state"); klee_make_symbolic(&t, sizeof(t), "t"); klee_make_symbolic(&s, sizeof(s), "s"); s++;
t == ON&s < 10	klee_make_symbolic(&state, sizeof(state), "state"); klee_make_symbolic(&t, sizeof(t), "t"); klee_make_symbolic(&s, sizeof(s), "s"); s++; s++; (t)state = 1; printf("state changed\n");	klee_make_symbolic(&state, sizeof(state), "state"); klee_make_symbolic(&t, sizeof(t), "t"); klee_make_symbolic(&s, sizeof(s), "s"); s++; out = 0; (t)state = 3; printf("state changed\n");	klee_make_symbolic(&state, sizeof(state), "state"); klee_make_symbolic(&t, sizeof(t), "t"); klee_make_symbolic(&s, sizeof(s), "s"); s++; out = s;

図 8 KLEE によって得られた情報から抽出した状態遷移表

3.3.3 条件分岐はどのように表現されるか？

TRACER によるソースコードの解析では、if(t==1) のような条件分岐は、 $t = 1$, $1 > t$, $t > 1$ の 3 つの経路へと進むよう解析され、また TRACER グラフにおいてもそのように表現される (図 3)。そのため、TRACER を用いた場合には、その後の分岐条件、および処理が同一である冗長な実行経路が抽出される。また、解析対象とするソースコードの規模が大きくなればこのような冗長な実行経路はさらに多くなることが予想される。一方で、KLEE を用いた場合には条件分岐はすべて真偽の 2 つの経路に進むため、こうした冗長な実行経路は抽出されづらいと考えられる。

3.3.4 実行経路は抽出しやすいか？

KLEE を用いた場合には、KLEE によって出力される KLEE 木ファイルによって得られる条件分岐の情報を用い

て、実行履歴ファイルから実行経路を抽出する必要がある。一方、TRACER を用いた場合、TRACER の出力としてはじめからグラフが出力され、ソースコードの振舞いを把握しやすい。そのため、TRACER を用いた方が実行経路の抽出にかかる実装コストが低いと考えられる。

4. 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 [4], [16]。本研究で提案している手法は記号実行を利用しており、実行環境がなくとも適用することができる。組込みシステム開発では、実行環境を構築することが容易でないことがあり、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可能性がある。

Said らは、組込みシステム向けのレガシーソフトウェアから状態機械を抽出する手法を検討している [5]。組込みシステムに適用するべく、実務者がどのような基準で状態変数を選ぶかを実験的に明らかにした。彼らは状態機械の抽出手法を提案したが、抽出された状態機械は規模が大きく煩雑なものとなったため、遷移し得ない遷移条件の削減や遷移条件の単純化を行っている [6]。

Walkinshaw らは、拡張有限状態機械 (EFSM, Extended Finite State Machine) の推論を試みた。従来の EFSM では、モデルが演繹的ではなく、どのような流れで計算されるかを予測できなかったため、その拡張として、実行中の変数の計算の流れをモデル化した [15]。この抽出は、実行トレースを利用した動的実行にもとづくものであり、手法として遺伝的プログラミングを用いている。

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 の実装を行っている [17]。この対話型 UI を利用すれば、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低減できると考えられる。

5. まとめ

本研究では記号実行ツール KLEE によって得られる出力からソースコード中の分岐条件、およびそれに伴う処理を抽出し、それらから状態遷移表を抽出した。また、状態遷移モデルの抽出における、記号実行ツール KLEE および TRACER の比較評価を行った。比較評価の結果、KLEE を用いた状態遷移表の抽出においては、入出力関数の呼び出しや `define` 等の命令を抽出できる可能性が高いことが分かった。一方で、KLEE を用いた場合、TRACER と比較すると実行経路の抽出のための実装コストが高いことが分かった。

今後の課題としては以下を検討している。

- ループやポインタ変数を含むソースコードについて、KLEE による解析を試み、解析可能を確認する。解析可能であれば、状態遷移表での表現方法を検討する。またこの課題について検討する中で、KLEE による出力と TRACER による出力を組み合わせる用いることも検討する。
- 3.1.1 節において KLEE による記号実行のため `klee.make_symbolic` 関数の呼び出しの挿入を行ったが、この挿入を自動で行いたいと考えている。このためには、条件分岐に使用される変数、そしてその変数とデータ依存や制御依存の関係にある変数を自動で取得する必要がある。
- 3.1.3 節において、ソースコードの終端は第 1 著者が既知であるものとした。しかし、ソースコードによっては実行経路ごとに終端の位置が異なり、終端を把握するのが困難な場合がある。そのため、構文解析等を用い、実行経路の終端を自動で取得することを検討する。

謝辞 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いた。ここに謝意を記す。本研究は、JSPS 科研費 JP18H04094 の助成を受けたものである。

参考文献

- [1] Cadar, C., Dunbar, D. and Engler, D.: KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs, *Proc. of OSDI 2008*, pp. 209–224 (2008).
- [2] Jaffar, J., Murali, V., Navas, J. A. and Santosa, A. E.: TRACER: A symbolic execution tool for verification, *Proc. of CAV 2012*, pp. 758–766 (2012).
- [3] Kung, D., Suchak, N., Gao, J., Hsia, P., Toyoshima, Y. and Chen, C.: On object state testing, *Proc. of COMP-SAC 1994*, pp. 222–227 (1994).
- [4] Lorenzoli, D., Mariani, L. and Pezzè, M.: Inferring state-based behavior models, *Proc. of WODA 2006*, pp. 25–32 (2006).
- [5] Said, W., Quante, J. and Koschke, R.: On state machine mining from embedded control software, *Proc. of ICSME 2018*, pp. 138–148 (2018).
- [6] Said, W., Quante, J. and Koschke, R.: Towards understandable guards of extracted state machines from embedded software, *Proc. of SANER 2019*, pp. 264–274 (2019).
- [7] Sen, T. and Mall, R.: Extracting finite state representation of Java programs, *Software & Systems Modeling*, Vol. 15, No. 2, pp. 497–511 (2016).
- [8] 清水貴裕, 山本椋太, 吉田則裕, 高田広章: 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, 第 25 回ソフトウェア工学の基礎ワークショップ (FOSE 2018), pp. 15–24 (2018).
- [9] Shimizu, T., Yoshida, N., Yamamoto, R. and Takada, H.: Symbolic Execution-Based Approach to Extracting a Micro State Transition Table, *Proc. of TAV-CPS/IoT 2019*, pp. 1–6 (2019).
- [10] 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).
- [11] 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, (オンライン), 入手先 (<http://techon.nikkeibp.co.jp/article/COLUMN/20150519/418967/>) (2016).
- [12] Tonella, P. and Potrich, A.: *Reverse engineering of object oriented code*, Springer (2005).
- [13] 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).
- [14] Walkinshaw, N., Bogdanov, K., Ali, S. and Holcombe, M.: Automated Discovery of State Transitions and Their Functions in Source Code, *Software Testing, Verification & Reliability*, Vol. 18, No. 2, pp. 99–121 (2008).
- [15] Walkinshaw, N. and Hall, M.: Inferring computational state machine models from program executions, *Proc. of ICSME 2016*, pp. 122–132 (2016).
- [16] Xie, T. and Notkin, D.: Automatic extraction of object-oriented observer abstractions from unit-test executions, *Proc. of ICFEM 2004*, pp. 290–305 (2004).
- [17] 山本椋太, 吉田則裕, 青木奈央, 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, 電子情報通信学会論文誌 D, Vol. J102-D, No. 3, pp. 151–162 (2019).