有界モデル検査ツールを用いたC言語プログラム部分合成

木村 悠介[1,a)]　石山 薫太郎[2,b)]　藤田 昌宏[3,c)]

概要：従来から入出力の論理的使用からプログラムを合成する研究が進められてきている。これに対して最近、合成対象プログラムに対し、シンタックス上の制限を加えることで探索範囲を絞り、限られた数の入出力組のみで自動合成する手法が注目されている。比較的少数の入出力組からプログラムを合成できるので、仕様の詳細は不明だが処理内容の概要が分かっている場合などに適用できる。本研究ではこの従来手法を拡張し、テンプレートを利用した合成手法を提案する。当手法では、プログラム中の何カ所かの不明な部分に対して矛盾のない解を自動的に生成することができる。C 言語用の有界モデル検査ツールを利用した実装を利用して、数 100 行規模の暗号化プログラムや制御ソフトウェアを合成した結果について報告する。

Partial C-Program Synthesis on Bounded Model Checker

Kimura Yusuke[1,a)]　Ishiyama Kuntaro[2,b)]　Fujita Masahiro[3,c)]

Abstract: Research has been done on what is called "Program Synthesis," where a program is synthesized from a given logical specification between inputs and outputs. Recently, a syntax restricted synthesis method that uses relatively few input/output vectors has been suggested, and it has drawn a great deal of positive attention. In this method, the search space of the synthesized program is restricted by setting the limitations on the scope of the program syntax. This can be applied in case where the designers cannot fully specify the program but understand the general flow of the target program. By extending this method, we introduce a new program synthesis method that uses a program template. Our proposed method can automatically generate a correct solution for a given partially vacant program. The method has been implemented on a bounded model checker for C, and the experimental results of the encryption program and the control program, written in hundreds of lines of C code, are presented.

1. はじめに

コンピュータプログラムはプログラマによって書かれることが一般的であるが、自在にプログラムを記述できるプログラマを十分に確保するのは容易ではない。プログラマがいない小さなプロジェクトでコンピュータの処理能力を活用したい場合、プログラマが不足している場合、思い通りのプログラムを書けるプログラマがいない場合などで、プログラムを自動合成する技術があれば問題が解決できるかもしれない。プログラムを自動合成するというアイディアは 1970 年代にすでに紹介されており、いくつかの手法がこれまでに提案されている [1]。

プログラムを合成するためには、合成したいプログラムの仕様を与える必要が有る。仕様の与え方の 1 つは、プログラムの出力が満たすべき論理を記述する手法である。例えば、与えられた複数の入力の最小値を答えるプログラムを考える。n 入力 $(in_1, in_2, \ldots, in_n)$、1 出力 out のプログラムとすると $\forall i : out \leq in_i \wedge \exists j : out = in_j$ を満たせば良い。このような記述を与えることでプログラムを自動生成する手法が [2] などで説明されている。この手法は数学的な論理式は記述できるがプログラミング言語を記述できない場合に有用だが、論理式の記述方法を学習しなくてはならないので学習コストがかかる。数学的に正しい論理式を記述するのはプログラミング言語の習得と同じくらいの労力が

1 東京大学大学院工学系研究科電気系工学専攻
Dept. of Electrical Engineering and Information Systems, The University of Tokyo
2 東京大学工学部電子情報工学科
Dept. of Electrical Engineering and Information Systems, The University of Tokyo
3 東京大学大規模集積システム設計教育研究センター
VLSI Design and Education Center, The University of Tokyo
a) kimura@cad.t.u-tokyo.ac.jp
b) kuntaro@cad.t.u-tokyo.ac.jp
c) fujita@ee.t.u-tokyo.ac.jp

必要であるとも言え、それに適した人材確保も容易ではないと考えられる。

論理式で仕様を与える代わりに、いくつかの入出力組を与えることでプログラムを合成する手法 [3] が提案されている。上述と同様の最小値プログラムを例にすると、$(in_0, in_1, in_2, out) = (4, 10, 5, 4), (-4, 29, 5, -4)$ などといった入出力例を与えることで自動的にプログラムが合成される。これはプログラミング経験のない人でも仕様を与えることが出来る点で優れている。

プログラム記述には無限のパターンがあるので、仕様を与えただけでプログラムを合成することはできない。そのため、予め合成されるプログラムに制限を設ける。多くの場合、制限は以下のものに設定される。

- プログラム中に現れる演算子の種類
- プログラムのコントロールフロー
- プログラムの文法

ただ、制限の与え方は手法によって異なる。Syntax Guided Synthesis (Sygus)[4] 手法では、プログラミング言語の定義に非常によく似た形でプログラムの大枠を与えている。例えばソースコード 1 は 2 つの入力と 10,20,30 という 3 つの定数の四則混合演算を定義した Sygus 記述である。$in0 + in1 - 30$ や $(in0 - 20) * in1/10$ などの計算を表現することが出来る。この他にも、プログラムの大枠をほとんど与えてしまい、一部の定数を空欄として値を探索する手法もある。

ソースコード **1** Sygus

```
1 ((Start Int (in0 in1 10 20 30)
2   (+ Start Start)
3   (- Start Start)
4   (* Start Start)
5   (div Start Start)
6 ))
```

本論文では、C 言語向け有界モデル検査ツールを用いたプログラムの自動合成手法を提案する。有界モデル検査ツールは本来コード中の様々なバグ（配列の範囲外アクセスや NULL ポインタ、0 除算、アサーションエラー）を発見するためのツールである。これらのツールはバグを発見する過程でソフトウェアを CNF 式に変換しており、SAT ソルバーなどで解を導出している（あるいは SMT 式を生成して SMT ソルバーで解いている）。この機能を上手く使う工夫をしてプログラムを合成する。なお、仕様は入出力パターンとして与える。プログラマ（プログラムを合成したい人）は、入出力の論理的仕様を明確に表記する必要はないが、与えられた入力に対する正しい出力は答えられるものとする。また探索範囲を制限するためにプログラマはテンプレートを用意するものとする。このテンプレートは C 言語で表記され、一部分が空欄になったものであるとする。簡単に C 言語の文章を生成するために「ALU-MUX ブロック」を導入する。

本論文では、第 2 節で提案するプログラム自動合成手法について説明する。特に 2.1 節ではどのようにして C 言語で記述されたテンプレートを作成するか、2.2 節では入出力組からプログラムを合成する方法を説明する。第 3 節では提案手法を用いた実験を行い、提案手法が適切に動作することを示すとともに、どの程度の規模のプログラム合成が可能かの議論を行う。最後に結論を述べる。

2. 提案手法

本手法では以下が準備されているものとする。

- プログラムのテンプレート
- 入力に対する正しい出力を知っている人（プログラムや論理式を使って入出力間の関係を書き下せる必要はない）。あるいは Matlab などによる正しく動作するシミュレータやエミュレータ
- Assert 文を扱うことが出来る有界モデル検査ツール

大まかな流れは以下の通りである。

(1) いくつかの初期入出力組を元にプログラムを 2 つ合成

(2) 2 つの合成結果が異なる動作をする 1 つの入力パターンをツールが自動で生成する

(3) 人間またはシミュレータ/エミュレータがその入力に対する正しい出力を指示する

(4) 以前与えた入出力組に加えて、再合成する

このループを繰り返すことによって、唯一のプログラムが得られるか、条件を満たすプログラムが存在しないかを決定することが出来る。

2.1 テンプレート

有界モデル検査ツールは C 言語中の assert 文に反する変数の値の割り当てを示すことが出来る。この機能を活用すれば未割り当ての変数を決定することが出来るが、それだけではプログラムを合成することは出来ない。そこで、ここでは図 2.1 に示す ALU-MUX ボックスを導入する。これは MUX（マルチプレクサ）と ALU（演算回路）が組み合わされた構造になっており、図中の ctrl 信号を調整することで値を選ぶことが出来る。図 1 では $in_0 + in_1$,$in_1 >> in_2$,$in2 \times in_0$ などを表すことが出来る。ポイントは、プログラム文を表現するということを、ctrl 信号を選択するという問題に帰着させている点である。このことによって、無限に存在するプログラムの書き方を制限するとともに、あくまで最適なパラメータを探索することしか出来ない既存ツールの制限のもとにプログラム文を合成することが出来る。

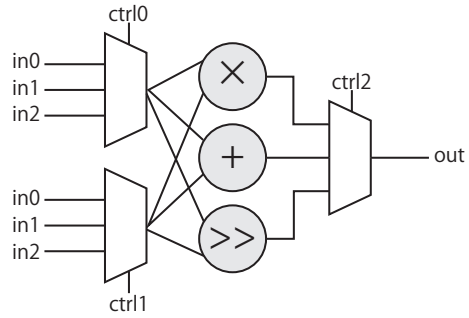

図 1　ALU-MUX ボックス

ソースコード 2　ALU+MUX

```
1  int alumux(int inputs[], ctrl0, ctrl1,
       ctrl2){
2    int in0,in1;
3    // MUX
4    in0=inputs[ctrl0]; in1=inputs[ctrl1];
5    // ALU
6    switch(ctrl2){
7    case 0:
8      return in0*in1;
9    case 1:
10     return in0+in1;
11   default:
12     return in0>>in1;}
```

実際にはこの ALU-MUX ブロックは C 言語によって表現されなければならない。コード 2 に C 言語で実装されたものを示した。$ctrl_0, ctrl_1, ctrl_2$ に値を設定すれば、プログラム文を生成できる。

これだけではただ 1 つの演算子を持つプログラム文しか生成できないので、実際にはこのブロックを複数つなげて $in_0 + in_1 \times in_2$ や $(in_0 >> in_1) + in_2$ などの演算子を複数持つプログラム文を生成する。このとき、ブロックをいくつどのように接続するかはプログラマ次第である。また未知の定数を導入することもできる。C 言語の大枠の中にこのような ALU-MUX ブロックを埋め込むことでテンプレートが完成する。

2.2　入出力から C 記述を生成する手法

生成される C 記述 $Ans(input)$ はテンプレート $Tmpl()$ とそのパラメータ P を用いて以下のように表現できるものとする。前述のテンプレートが簡単のために $Tmpl(P)$ と表現されており、P は ctrl などの探索した割り当ての組であるもとのする。

$$Ans(in) = Tmpl(in, P) \tag{1}$$

ここで、C 記述は合成したいプログラム $Spec$ と同じ挙動を示す必要があるので、求めたいパラメータ P は以下のように書き表せる。

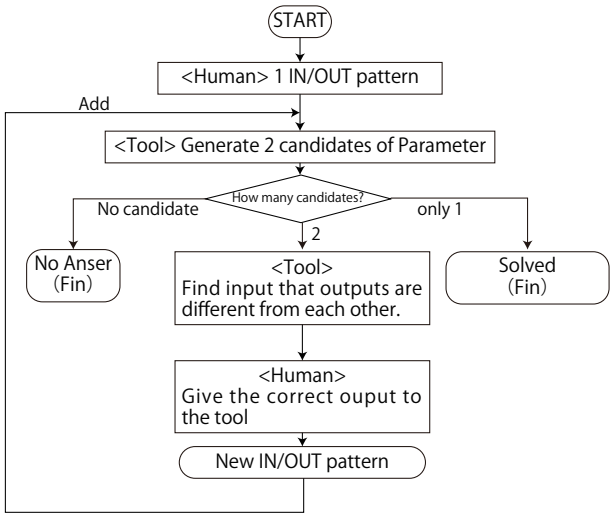

図 2　Synthesis Flow

$$\exists P \forall in : Ans(in) = Tmpl(in, P) = Spec(in) \tag{2}$$

P について定式化することが出来たが、ここで 1 つ問題がある。式 (2) 内で $Spec$ は関数として示されているが、第 2 節冒頭で述べた前提ではプログラマは正しい出力を答えられても入出力の論理関係はわからない。そのため、$Spec$ は関数として書き下せるものではなく、式 (2) を形式的に解くことが出来ない。

本手法では、以下に示すように繰り返し、パラメータ P を更新することで正しい結果を得る手法を採る。まず、ある特定の入力 in_0 について設計データと同じ挙動を示すパラメータ P_0^0 を合成する。なお、下の式の右辺には入力として具体的な値 in_0 が与えられているので、プログラマまたはシミュレータが正しい出力を提供できる。

$$\exists P_0^0 : Tmpl(in_0, P_0^0) = Spec(in_0) \tag{3}$$

式 3 を満たすようなパラメータ P_0 は複数ある可能性があるが、そのうち P_0^0 とは異なる出力となるような入力パターンが存在するものを P_0^1 として、以下のように定義できる。

$$\begin{aligned} \exists P_0^1,\ in_1 :\ & Tmpl(in_0, P_0^0) = Spec(in_0) \\ \wedge\ & Tmpl(in_0, P_0^1) = Spec(in_0) \\ \wedge\ & Tmpl(in_1, P_0^0) \neq Tmpl(in_1, P_0^1) \end{aligned} \tag{4}$$

この時の in_1 はいくつかの P_0 の候補を区別し正しい P を導き出すために必要なものであるから、新しく次のような数式を得ることが出来る。

$$\begin{aligned} \exists P_1^0\ :\ & Tmpl(in_0, P_1^0) = Spec(in_0) \\ \wedge\ & Tmpl(in_1, P_1^0) = Spec(in_1) \end{aligned} \tag{5}$$

繰り返し、新しい in_2 を得ることが出来る。

$$\exists P_1^1,\ in_2:$$
$$Tmpl(in_0, P_1^0) = Tmpl(in_0, P_1^1) = Spec(in_0)$$
$$\wedge Tmpl(in_1, P_1^0) = Tmpl(in_1, P_1^1) = Spec(in_1)$$
$$\wedge Tmpl(in_2, P_1^0) \neq Tmpl(in_2, P_1^1) \quad (6)$$

さて、n 回目の時に P_n^0 を得るための式は以下の通りである。ここでもしこの式を満たすような P_n^0 が存在しない場合、与えられたテンプレート下でシミュレータと同じ挙動を示すものは生成出来ないことになる。

$$\exists P_n^0 : Tmpl(in_0, P_n^0) = Spec(in_0)$$
$$\wedge\ Tmpl(in_1, P_n^0) = Spec(in_1)$$
$$\vdots$$
$$\wedge\ Tmpl(in_n, P_n^0) = Spec(in_n) \quad (7)$$

最後に in_{n+1} を探すための式は以下のようになる。この数式を満たすような in_{n+1} がない場合には、P_n^0 が求めるパラメータであり、求める C 言語記述は $Tmpl(in, P_n^0)$ である。

$$\exists P_n^1,\quad in_{n+1}:$$
$$Tmpl(in_0, P_n^0) = Tmpl(in_0, P_n^1) = Spec(in_0)$$
$$\wedge\ Tmpl(in_1, P_n^0) = Tmpl(in_1, P_n^1) = Spec(in_1)$$
$$\vdots$$
$$\wedge\ Tmpl(in_n, P_n^0) = Tmpl(in_n, P_n^1) = Spec(in_n)$$
$$\wedge\ Tmpl(in_{n+1}, P_n^0) \neq Tmpl(in_{n+1}, P_n^1) \quad (8)$$

2 つのパラメータを生成し、それらを区別する入力パターンを追加し続けることを繰り返すことで、いつかパラメータの候補が 1 つだけになるか、1 つも候補が得られなくなる。よってシミュレータとテンプレートのみで C を生成することが出来る。この流れをまとめたものを図 2 に示した。

この手法の利点は、必ずしもすべての入出力組に言及せずにプログラムを合成することができる点である。次節で示すが、たくさんの組み合わせが考えられる入力に対して高々 10 個程度の入出力組を与えるだけでパラメータが 1 つに定まることが多い。

逆にテンプレートを用いた手法は必ずしも正しいプログラムを生成しないことがある。2 入力の関数 $f(in_0, in_1)$ を考える。この関数は $f(in_0, in_1) = in0 + in1$ であるものとする。この時テンプレートが制御信号 $ctrl$ としてコード 3 で与えられるとする。

ソースコード **3** Wrong Template

```
1 int f(int in0, int in1, int ctrl){
2   if(ctrl==0) return in0*in1;
3   else return in0-in1;
4 }
```

表 1 Largest Power

#	Description	Time(s)	Vars	Clauses
1	Get P	0.037	7,084	16,391
2	Get input	0.095	15,819	33,996
3	Get P	0.084	13,632	32,347
4	Get input	0.169	23,753	53,910
5	Get P	0.197	20,180	48,303
6	Get input	0.226	29,185	66,485
7	Get P	0.180	26,728	64,259
8	Get input	0.520	35,799	64,259
	Total	1.508	既存研究 [6]: 8s	

初期入出力組 $(in0, in1, out) = (0, 0, 0)$ が与えられた場合、$ctrl_0 = 0, ctrl_1 = 1$ の 2 つのパラメータが得られる。この 2 つを区別する入力には $(in0, in1) = (1, 0)$ があり、この時の出力 $out = 1$ である。2 つの入出力組 $(0, 0, 0), (1, 0, 1)$ を満たすパラメータは唯一 $ctrl = 1$ だけであり、よって新たに得られる C は $out = in0 - in1$ となる。これは明らかに誤ったテンプレートである。例えば入力 $(1, 1)$ に対しては正しい出力は 2、新たに得られた C の出力は 0 となる。このように、テンプレートが元の設計に対して十分でない場合には誤った C が抽出される可能性がある。

3. 実験

3.1 実験環境

本実験では有界モデル検査ツールとして CBMC (Version 5.4)[5] を使った。実験に使用したコンピュータは Linux Kernel 4.6.4 で動作しており、Core i7-3770(3.4GHz),16GB メモリを搭載している。実験に使用したプログラムはすべてシングルスレッドで動作している。

3.2 Largest Power

当実験は CBMC を用いてプログラムが合成できることを示すと同時に、SMT ソルバを使った既存合成手法 [6] との比較を行うことを目的としている。

Largest Power プログラムは、入力を超えない最大の $2^n - 1$ (ただし $n \in \mathbb{Z}$) を求めるプログラムである。入出力例を以下に示す。$2^n - 1$ は下位ビットがすべて 1 で埋まる点に注目したい。

- 7(b00000111): 3(b00000011, n=2)
- 16(b00010000): 15(b00001111, n=4)
- 13(b00001101): 7(b00000111, n=3)

このプログラムは、コード 4 を用いると正しく動作することが示されている。本実験ではこのアルゴリズムの最後の 1 行が未知のものであると仮定して、四角で囲った部分を ALU-MUX ブロックに置き換えてテンプレートを作成した。具体的には、ALU-MUX ブロックが 3 個順番につながった形になっており、演算は |,>> のいずれかが選択できるようになっている。各ブロックには入力として、x・定

ソースコード 4　LargestPower のアルゴリズム

```
1 uint32_t largest_power(uint32_t x){
2   x = (x >> 1) | x;
3   x = (x >> 2) | x;
4   x = (x >> 4) | x;
5   x = (x >> 8) | x;
6   x = ((x >> 16) | x) >> 1;
7   return x;
8 }
```

数・自分より前にあるブロックの出力が接続されている。実験結果を表 1 に示す。左端の番号ごとに CBMC を動作させており、奇数回目で入出力組に対する 1 つの正しい P を探索、複数回目で 2 つの P を区別する入力パターンを探索している。8 回目で UNSAT となり、パラメータ P を 1 つに絞ることが出来、正しいプログラムが合成できた。合成には合計で 4 つの入出力組を用いた。

第 2 節の最後でも述べたが、得られたプログラムはあくまで与えられた 4 つの入出力組に対する正しさしか保証していない。テンプレートの質が悪い場合には期待するプログラムが得られない可能性が有る。本実験では正しいプログラムが合成できるようなテンプレートを利用しているので正しいプログラムが合成できているように、テンプレートの正しさが保証できるのならば必ず正しいプログラムが得られる。

既存研究 [6] は SMT ソルバ Z3 の API を用いて同様の問題を解いている。本研究は SAT ソルバを用いている点、ソルバの API を使用せずに C 言語をテンプレートとして与えている点で異なる。既存研究では SMT では 5 倍近い時間がかかってパラメータが導出されている。これは SMT ソルバを使用したことによる速度低下が原因と考えられる。

3.3　飛行船ホバリングのための制御プログラム

この実験では、小形のラジコン飛行船 (Maneuver) に搭載されている制御プログラムの一部を用いた。この飛行船には超音波センサーが搭載されており、床からの高度を測定できるようになっている。ホバリング時には機体を有る一定上の高さに留まらせる必要があるものとし、高度に応じてモータの回転速度を調整するものとする。

予め搭載されているプログラム例をコード 5 に示す。このコード例は高度に応じて 30,50,80 の 3 種類のモータ速度が選択されるようになっている。規定よりも高度が低い場合には機体を状態させなくてはならないのでこのような場合分けが必要だが、モータ回転数が連続的な値で変化しないので制御が雑である。この実験の目的は枠で囲まれた”speed=50” の部分を ALU-MUX ブロックで置き換え (図 3)、より円滑な制御を実現することとする。プログラムを合成するには、与えられた高度に対する望ましいモータ出力をプログラマが答える必要がある。この解答は事前に計算して用意されているものとする。

ソースコード 5　Maneuver

```
1 int propeller_speed(int altitude){
2       int speed, lower=15, target=30;
3       if(altitude>target){
4             speed=30;}
5       else if(lower<altitude){
6             speed=50;
7       }else{ speed=80;}
8 return speed;}
```

得られた結果を表 2 に示す。合計で 3 つの入出力組を用いて、”alt*254+90” という答えを得ることが出来た。詳しく説明すると、1 回目に (20,50) という入出力に対する P を 2 つ求め、2 つを区別する入力 22 を発見している。2 回目では先ほど得られた (22,46) を入力組に加えて新しい P を 2 つ導出し、それらを区別する入力 21 を求めた。3 回目では (21,48) を加えて P を求めるも、2 つの P を得られなかったので、最後に 1 つだけ P を導出している。このようにして合計 19 分程度で新しいプログラムを求めることが出来た。

もとのプログラムとの比較のために図 4 を示す。紫の線が元のプログラム、緑の線が合成されたプログラム、青の線は考えられる悪い合成例である。紫と緑のデータを比べると、新たに合成したもののほうがモータ速度が段階的に変化していてより安定した制御を行っていることが分か

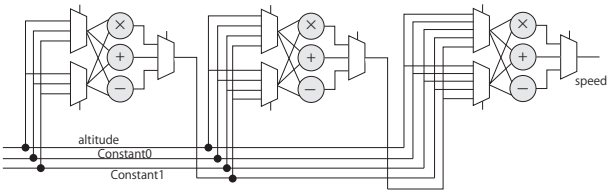

図 3　ALU-MUX blocks in Maneuver

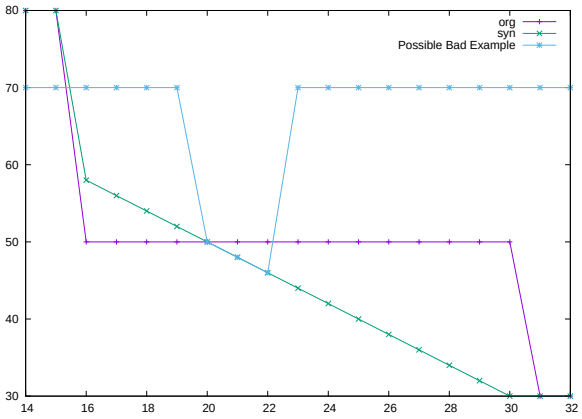

図 4　Solution Assessment

表 2　Maneuver

Iter	Time	SAT Statistic		Added	Solutions	
		# of Variables	# of Clauses	In/out constraint	1st	2nd
1	0.048s	6119	17322	(20,50)	50	(238-alt)*93
2	0.171s	8879	25266	(22,46)	254*(83+alt)	alt*131*(30-alt)
3	18m44s	11639	33210	(21,48)	UNSAT(only 1 or 0 solution)	
4	0.128s	4370	12168	-	alt*254+90	-
Total	18m47s			3 sets	alt*254+90	

表 3　AES

#	Description	Time	Vars	Clauses
1	Get P	32min	277,259	1,271,415
2	Get another P	130min	277,270	1,271,456
3	Get input	27sec	564,068	3,713,881
4	Get P	376min	554,421	2,870,737
5	Get another P	431min	554,432	2,870,778
	Total	970min		

る。しかし、テンプレートによっては青の線に示したようなプログラムが得られる可能性が有る。これは与えた入出力を満たすだけで、他の入力に対しては常に 70 一定の出力を行っている。本実験の目的を考えると、良いプログラムではない。このように、テンプレートの質次第で合成結果は変化する。

3.4 AES

本手法がどれくらい大きなプログラムにまで適用可能か調べるために、AES 暗号化プログラムを例題に用いる。

AES256 は SubBytes, ShiftRows, MixColumns, AddRoundKey という 4 つの操作を順番に 14 回繰り返すことで平文を暗号化する。特に SubBytes は入力された値を変換表 (sbox) をもとに異なる値にする操作であり、本実験ではこの変換表の一部を空欄にして正しく合成できるかどうか確かめた。sbox には 256 個の 8bit 変数が格納されており、初めの 3 つを未知のものとした。プログラムは 200 行程度で記述されており、CBMC で扱い易いよう標準ライブラリなどを用いない純粋な C 言語で書かれている。

実験結果を表 3 に示す。本実験では以前の実験とは違い、P を 1 つ、P をもう 1 つ、それらを区別する入力を 1 つ、という順番に CBMC を用いている。第 2 章で紹介した方法を実行している点で変わりはない。およそ 16 時間で、正しい sbox の空欄３つ分を合成することができ、あわせて２つの入出力組を用いた。8bit 変数３つの組み合わせは合わせて $2^{8^3} = 2^{24}$ 通り存在するが、たった２つの入出力組で合成出来る点に着目したい。また 14 回繰り返し実行する 200 行の C プログラムの一部を合成できている点も注意したい。

4. 結論と今後の課題

本研究では有界モデル検査ツールを用いた C 言語プログラム自動合成手法を紹介した。この手法ではプログラムのテンプレートを用意し、入力に対する正しい出力をいくつか答えることでプログラムを合成することが出来る。実験では本手法が正しく動作することを示した上で、数百行の C プログラムでも動作することを AES 暗号化プログラムで示した。副次的な成果ではあるが、SMT ソルバを利用する既存手法と比べて SAT ソルバベースのツールを使った本手法が 5 倍程度高速に動作することもわかった。

今後の課題として、合成をインクリメンタルな SAT 手法を用いることなどが挙げられる。本手法では SAT ソルバに与えられる CNF 式は以前のものに条件を追加しただけであるが、実際には一から解き直しをしている。インクリメンタル手法を導入すれば高速化が期待できる。

参考文献

[1] Gulwani, S.: Dimensions in program synthesis, *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming - PPDP '10*, New York, New York, USA, ACM Press, pp. 13–24 (online), DOI: 10.1145/1836089.1836091 (2010).

[2] Srivastava, S., Gulwani, S., Foster, J. S., Srivastava, S., Gulwani, S. and Foster, J. S.: From program verification to program synthesis, *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '10*, Vol. 45, No. 1, New York, New York, USA, ACM Press, p. 313 (online), DOI: 10.1145/1706299.1706337 (2010).

[3] Susmit Jha, Sumit Gulwani, S. A. S. A. T.: Oracle-Guided Component-Based Program Synthesis.

[4] Alur, R., Bodik, R., Juniwal, G., Martin, M. M. K., Raghothaman, M., Seshia, S. A., Singh, R., Solar-Lezama, A., Torlak, E. and Udupa, A.: Syntax-Guided Synthesis, *2013 Formal Methods in Computer-Aided Design*, p. 10 (online), DOI: 10.1109/FMCAD.2013.6679385 (2013).

[5] Clarke, E., Kroening, D. and Lerda, F.: A Tool for Checking ANSI-C Programs, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)* (Jensen, K. and Podelski, A., eds.), Lecture Notes in Computer Science, Vol. 2988, Springer, pp. 168–176 (2004).

[6] MATSUMOTO, T., JO, S. and FUJITA, M.: プログラム可能データパスと SMT ソルバーを利用した高位設計デバッグ手法, *IEICE technical report. Computer systems*, Vol. 113, No. 497, pp. 103–108 (2014).