Vol.2018-SLDM-184 No.2
2018/5/16

テンプレートと状態遷移表現を利用したシステム最適化手法

合田　瑛洋

東京大学大学院工学系研究科

電気系工学専攻

goda@cad.t.u-tokyo.ac.jp

藤田　昌宏

東京大学

大規模集積システム設計教育研究センター

fujita@ee.t.u-tokyo.ac.jp

あらまし **— 順序回路の形式的検証は入力と出力のシーケンス比較をする必要があるため組み合わせ回路の場合よりもはるかに難しく、FSMD(Finite State Machine with Data-path)は本質的には順序回路を表していると言えるためその形式的検証も同じように難しい。そこで、false negative な性質を持つものの効率的な検証を行うことができるアルゴリズムである「Karfa の手法」が提案されたが、これによってシステムの部分合成に用いられる手法である「テンプレートベースの合成」を FSMD に適用することができるようになった。本紙では Karfa の手法とテンプレートベースの合成を用いて FSMD の合成を行うアルゴリズムを提案し、実験を通してその手法で実際に合成が可能であることを示す。**

キーワード—FSMD, 最適化, Karfa の手法, テンプレートベースの合成

I. はじめに

組み合わせ回路間の等価性検証を考える際は、miter と呼ばれる図 1 のような回路をまず生成する。図のように miter では、2 つの回路の入力は共有され、出力は対応するもの同士がそれぞれ XOR ゲートに接続されてさらにその出力が OR ゲートに接続されるが、仮に 2 つの回路の出力が異なる場合、対応する XOR ゲートの出力が 1 になり、その結果 OR ゲートの出力も 1 となる。ゆえに、もし OR ゲートの出力が 1 になることがあれば 2 つの回路は等価でないということができ、また逆にもし OR ゲートの出力が常に 0、つまり回路が unsat であるということを示すことができれば、回路は等価であるということもできる。miter では、内部信号の対応を取ってより効率的な検証を行うこともできる。すなわち、回路の内部に等価な点を発見できれば、その点の片方をもう片方の点で置き替えることができるので、もし miter 内部に多くの等価な内部信号があれば回路全体を非常に小さくすることができ、検証を非常に効率的に行うことができる。

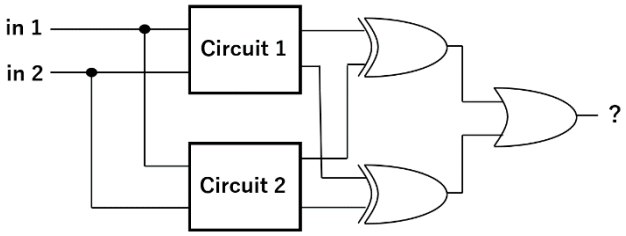
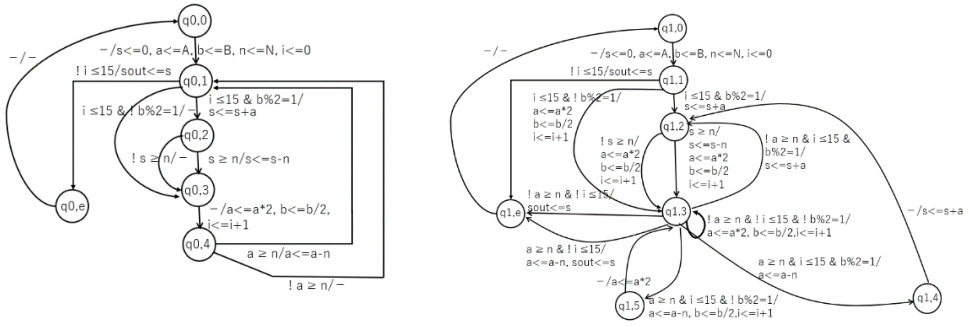


図 1. miter の例

一方、2 つの順序回路同士を比較しようとすると、入力と出力のシーケンスを確認しなければならないため、組み合わせ回路間と比べて非常に難しい問題となる。一般的に順序回路の完全な等価性検証には、内部動作がループになるような長いシーケンスの入力の組に対する出力の組が必要だが、それを実際に処理するのは不可能であるため、「bounded」に代表される様々な制約付き等価性検証の手法が提案されている。Bounded の手法では、まず順序回路を一定のサイクル数展開し、展開された回路同士で組み合わせ回路と同様に検証を行うが、展開するサイクル数は現状では 1000 回程度が限界であり、それ以上のサイクルが回った後の挙動については検証できないという欠点がある。



(a)FSMD1　　(b)FSMD2

図 2. FSMD の例

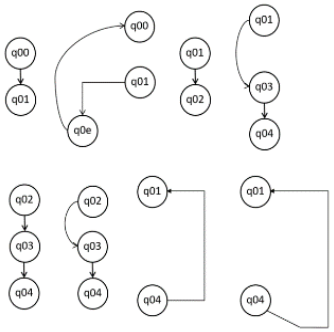


図 3. 分割された FSMD1

また、順序回路を検証する別の方法として、大きな回路を小さな部分に分割してそれらを比較するというものがある。しかしながらこの手法は、実際には等価であるものを不等価として判定してしまう「false negative」な性質を持つことがある。

FSMD(Finite State Machine with Data-path)[1]は、抽象度の異なる設計やシステムを記述できるという点で有用であり、活発に研究されてきている。例えば、状態遷移がクロックと同期して行われるとすればそれは RTL 記述そのものを表現していると言える一方、計算の順序のみを示していると見ればソフトウェアを表現しているということもできる。FSMD は本質的には順序回路を現していると見ることができるため完全な等価性検証は順序回路と同様に難しいが、その中で FSMD を分割して検証するアルゴリズムである「Karfa の手法」[2]が提案された。この方法では、FSMD はまず小さなパスに分けられ、それらのパス同士を比較して対応するものを見つけることで等価性検証を行うが、これによって大きな FSMD であっても小さなパスの問題に持ち込めるので効率的に検証を行うことができる。だが一方、この手法は FSMD 全体では等価であってもパス間の対応は取れない可能性があるという点で false negative な性質を持つ。

システムの部分合成の手法として「テンプレートベースの合成法」[3]が提案されており、設計に無限ループを含まない多くの種類のシステムの合成に用いられているが、先述のように Karfa の手法は FSMD をパスに分けてループを展開するので、これによって false negative な性質は残るもののテンプレートベースの合成法を FSMD にも適用できるようになった。本稿では、Karfa の手法とテンプレートベースの合成法を用いて FSMD の部分合成を行うアルゴリズムを提案し、実験によってその手法で実際に合成が出来ることを示す。

II. 関連研究

A. *Karfa の手法*

Karfa の手法の基本的なアルゴリズムをコード 1 に示した。本章では、具体例を用いてその詳細を説明する。

1. FSMD のパスへの分割

先述のように Karfa の手法では FSMD は分割されるが、それにあたって「カットポイント」という概念を導入する。ある状態が以下に示す 2 条件のうち一方を満たしているとき、その状態をカットポイントとする。

- 各システムの最初(システム開始時)の状態。
- 条件分岐を含む状態。すなわち、自身から矢印が 2 つ以上出ている状態。

図 2 の FSMD では、FSMD1 は([q0,0], [q0,1], [q0,2], [q0,4])の 4 つ、FSMD2 は([q1,0], [q1,1], [q1,2], [q1,3]).の 4 つのカットポイントを持つことになる。

次に、FSMD をカットポイントに従って分割していく。FSMD1 の例では、分割によって図 3 に示すような 8 つのパスが得られる。図に示されているように、FSMD はカットポイントで始まりカットポイントで終わるパスに分けられることになる。

2. 対応するパスの探索

次に、分割したパスの中で互いに対応が取れているものを探す。ただし、ここでいう対応が取れているとは、パス同士が等価であることを意味している。

まず FSMD1 に関して、その最初のパスである[q0,0]→[q0,1]と等価なパスを FSMD2 から探すと、対応するパスとして[q1,0]→[q1,1]が直ちに見つかる。

```
Path1 = FSMD1 のパスの集合
Path2 = FSMD2 のパスの集合
P = Path1 の最初のパス
while (FSMD1 の全てのパスに対応するパスがそれぞれ FSMD
       から見つかっていない):
    Q = Path2 内で P と対応するパス
    if (Q != NULL):
        P = Path1 の次のパス
    else :
        Path2’ = Path2 のパスそれぞれを延長したもの
        Q = Path2’内で P と対応するパス
        if (Q != NULL):
            P = Path1 の次のパス
        else :
            P’ = P
            while (P’の全てのパスに対応が取れるまで):
               if (P’のパスはこれ以上延長できない):
                    return NOT EQUIVALENT
                P’ = P’のパスそれぞれを延長したもの
                P’’ = [‘’]
                for (P’ の要素一つずつに対して):
                    Q = Path2,、Path2’内で P’の要素
                        と対応するパス
                    if (Q == NULL) :
                        P’’ . append(P’)
                P’ = P’’
            P = Path1 の次のパス
FSMD1 と FSMD2 の役割を入れ替えてもう一度同じ処理を行う
return EQUIVALENT
```

コード 1. Karfa の手法

次に、FSMD1 の次のパスである[q0,1]→[q0,e]→[q0,0]、[q0,1]→[q0,2]、[q0,1]→[q0,3]→[q0,4]についてそれぞれ等価なパスを探すと、[q1,1]→[q1,e]→[q1,0]、[q1,1]→[q1,2]、[q1,1]→[q1,3]が見つかる。

再び同様に、今対応を取ったパスの終点である[q0,2]、[q0,4]から始まるパスに関して対応を探す。まず[q0,2]から始まるパスである[q0,2]→[q0,3]→[q0,4] (左側)と[q0,2]→[q0,3]→[q0,4] (右側)に関して探すと、[q1,2]→[q1,3] (左側)と[q1,2]→[q1,3] (右側)が対応するパスとして見つかる。

3. パスの延長

次に、[q0,4]から始まるパスとしてまず[q0,4]→[q0,1] (内側)を取って対応するパスを探しても、FSMD2 の中からは見つからない。このときはまず、FSMD2 の複数のパスを合わせて 1 つと見ることでパスを延長し、対応するものがないか探す。例えば、FSMD2 の 2 つのパスである[q1,1]→[q1,2]と[q1,2]→[q1,3]は、分割前の FSMD2 内では繋がっていたものである。そこで、この 2 つのパスを合体させ、新たに[q1,1]→[q1,2]→[q1,3]を 1 つのパスと見て問題とするパスと対応していないか確認する。このような動作を FSMD2 に関する「パスの延長」と呼ぶが、これを含めて対応するパスがないかを探す。

しかしながら、今回の例では FSMD2 に関してどのようにパスを延長しても[q0,4]→[q0,1] (内側)に対応するパスは見つからない。このときは、今問題にしている[q0,4]→[q0,1] (内側)のパスを同様に延長して考える。今回の例では、延長されたパスとして新たに[q0,4]→[q0,1]→[q0,e]→[q0,0]、[q0,4]→[q0,1]→[q0,2]、[q0,4]→[q0,1]→[q0,3]→[q0,4]が得られるが、これらに関して FSMD2 の延長したパスも含めて対応するものがないか探すと、それぞれ[q1,3]→[q1,e]→[q1,0]、[q1,3]→[q1,4]→[q1,2]、[q1,3]→[q1,5]→[q1,3]が見つかり、FSMD1 で延長された全てのパスに対して対応が取れたので[q0,4]→[q0,1] (内側)に関しては対応が取れたものとして先に進む。

次に[q0,4]→[q0,1] (外側)のパスについて考えると、同様にパスの延長によって対応するパスがあることが確認できる。これによって FSMD1 の全てのパスに関して対応が取れることが分かった。

4. 逆向きの対応

次に、FSMD1 と FSMD2 の立場を入れ替えて同じように検証を行う。冗長になるため詳細は省略するが、今回の例では FSMD2 の全てのパスに対して対応が取れるため、FSMD1 と FSMD2 は等価であるということが出来る。

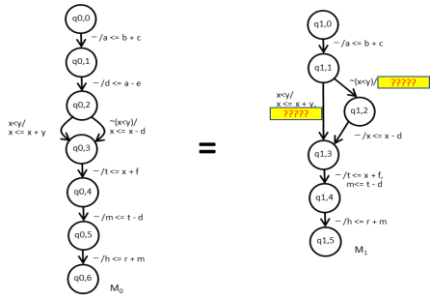


図 4. 2 つのシステム

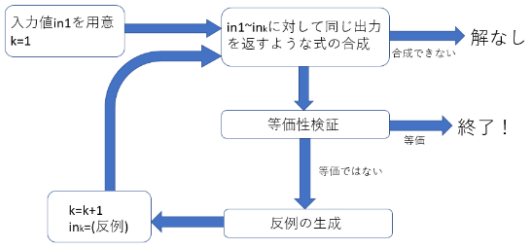


図 5. テンプレートベースの合成のフローチャート

B. テンプレートベースの合成法

テンプレートベースの合成法は、図 4 のような 2 つのシステムがあると仮定したとき、ブランクに代入文などを挿入して 2 つのシステムが等価になるような式を自動合成するアルゴリズムである。なお、ここでは簡単のため、合成されうる式の形を「a=b+c」のように 2 つの変数の演算結果を 1 つの変数に代入するような形の代入文に限定されるとする。

図 6. 提案手法のフローチャート

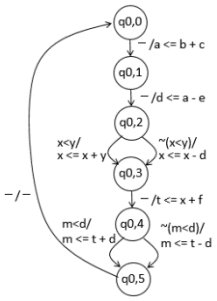


図 7. FSMD の例

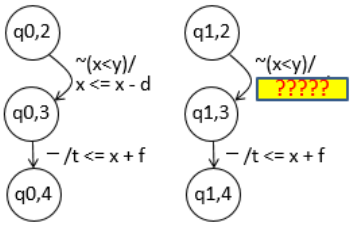


図 8. 対応の取れないパス

図 5 にアルゴリズムの大まかな流れを示した。本章ではこれを図 4 の具体例を用いて解説するが、それに際して図 4 左側のシステムを「specification」、右側のシステムを「template」と呼ぶ。

1. 1 つの入力の組の計算

まず最初に、1 つ適当な変数の入力値の組を用意し、specification に代入して出力値を得る。例えば今、入力を (a,b,c,d,e,f,h,m,r,t,x,y)=(1,2,3,4,5,6,7,8,10,11,12,13) (以降 in1) とした場合、specification による計算を経て出力値として (a,b,c,d,e,f,h,m,r,t,x,y) = (5,2,3,0,5,6,41,31,10,31,25,13) (以降 out1)を得る。

2. 式の決定

もしこれら 2 つのシステムが等価であるならば、その必要条件として template に in1 を代入したときの出力値は out1 と等しくなるはずである。そこで in1 と out1 の値をそれぞれシステムの最初と最後から代入してブランクの直前と直後でどのような値になるかを調べると、ブランクの直前で (5,2,3,4,5,6,7,8,10,11,25,13)、直後で (5,2,3,0,5,6,-,-,10,-,25,13)となることが分かる。ここで、ブランクの直後の値のうち確定できないものはハイフンとした。

これら 2 つの変数の組を比較すると、ブランクの前後で d の値に変化があることが分かる。そこで、ブランクの式によって代入される変数は d で確定とし、次にブランク後の d の値である 0 を代入できるような式を考えると、代入式は e-a もしくは a-e であることもただちにわかる。そこで今回は、解である可能性のある式(以降解候補)として d=e-a の式を取って合成を進めることとする。

3. 等価性検証

次に、今合成した解候補をブランクに代入した状態で等価性検証を行う。ここでもし等価であれば正しい解が得られて合成終了ということになるが、等価でなかった場合は、2 つのシステム間で出力値が異なるような入力値の組が 1 つ以上あるということなので、そのうちの 1 つ(以降 counter example)を合成する。今回の例はまだ等価でないので、counter example として例えば (a,b,c,d,e,f,h,m,r,t,x,y) = (0,1,1,0,1,0,0,0,0,0,0,1)が合成される。

4. 再合成・再検証

Counter example が生成された場合は最初に戻り、最初に決めた入力値と counter example の両方に対して出力値が等しくなるような式を合成し、再び等価性検証をして、…というようにシステムが等価になるまで繰り返すことで、最終的に合成を終了する。なお、このアルゴリズムは内部にループを含むが、それを回る回数はさほど多くならないことが経験則的に知られている[2]。

III. 提案手法

図 6 に提案手法の大まかな流れを示した。本章ではそのアルゴリズムの詳細を、図 7 の FSMD を例にとって説明する。ここで、図 7 左側の FSMD を FSMD1、右側の FSMD を FSMD2 とする。

1. FSMD の分割

まず最初に、2 つの FSMD を Karfa の手法に則ってパスに分割する。これによって、FSMD1 は[q0,0] →[q0,2]、[q0,2] →[q0,4](左側)、[q0,2] →[q0,4](右側)、[q0,4] →[q0,0](左側)、[q0,4] →[q0,0](右側)の 5 つのパスに、FSMD2 は[q1,0] →[q1,2]、[q1,2] →[q1,4](左側)、[q1,2] →[q1,4](右側)、[q1,4] →[q1,0](左側)、[q1,4] →[q1,0](右側)の 5 つのパスに分けられる。

2. 対応するパスの探索

次に、パスに分けられた FSMD に Karfa の手法を適用し、対応するパスを探す。ただしこのとき、ブランクを含むパスは除外して考える。

まず、FSMD1 のパスについて考える。今回の例では、[q0,0]→[q0,2]と[q1,0]→[q1,2]、[q0,2] →[q0,4](left one)と[q1,2] →[q1,4](左 側) 、 [q0,4] →[q0,0](左 側) と [q1,4] →[q1,0](左側)、[q0,4]→[q0,0](右側)と[q1,4]→[q1,0](右側) がそれぞれ対応し、[q0,2]→[q0,4](右側)に対応するパスは見つからないことが分かる。

次に、FSMD 同士の役割を入れ替え、FSMD2 のパスについて考えると、全く同じように対応が取れ、除外されている[q1,2]→[q1,4](右側)を除く全てのパスに関して対応を取ることができる。

3. 対応の取れないパス間の合成

ここで、対応の取れていないパスをまとめる。今回の例で対応が取れていないものは図 8 のようになる。

ここで挙げた以外のパスについては既に対応が取れていることを考えると、Karfa の手法のアルゴリズムからこれらのパスがお互いに対応すれば FSMD 全体が等価であるということができる。そこで、これらが互いに対応するような式をテンプレートベースの合成法を用いて合成する。これによって FSMD の部分合成が成功するというアルゴリズムである。

IV. 実験

1. 実験 1

実験 1 では、FSMD 間、プログラム間、RTL 記述間においてそれぞれ提案手法で実際に部分合成が可能であることを確かめるため、7 つの例題について合成を行った。

表 1 に実験結果を示した。表 1 で、example 1 と example 2 は FSMD 間、example 3、example 4、example 5 がプログラム間、example 6 と example 7 が RTL 記述間の合成である。また、lines はプログラムや RTL 記述をそれぞれ C 言語、Verilog で表現した時の行数、blanks はブランクの数、states は FSMD の状態数、paths は Karfa の手法によってパスに分けたときのパスの数、equivalence check は等価性検証を行った回数、time は実行時間を示しており、1 は specification、2 はブランク付システムを表している。

結果から分かることとして、全ての例題に関して合成が成功したことに加え、その実行時間は状態数よりもパスの数に大きく影響を受けていることが分かった。

2. 実験 2

実験 2 では、FSMD にどの程度の制約を与えれば式が合成できるのかを明らかにするため、同じ FSMD 間でブランクの数を増やしながら実験を行い、それに伴う実行時間の変化を計測した。

結果を表 2 と図 10 に示した。ただし図 10 では、縦

表 1. 実験 1 結果

	lines 1	lines 2	blanks	states 1	states 2	paths 1	paths 2	equivalence check	time [s]
example 1	–	–	2	7	6	3	3	31	21.7
example 2	–	–	1	6	7	8	12	323	38.2
example 3	28	30	1	6	6	3	3	23	2.62
example 4	41	44	1	6	3	4	5	74	7.98
example 5	44	45	1	7	6	7	7	180	11.2
example 6	28	30	1	6	6	3	3	23	2.62
example 7	33	32	2	3	3	3	3	27	9.36

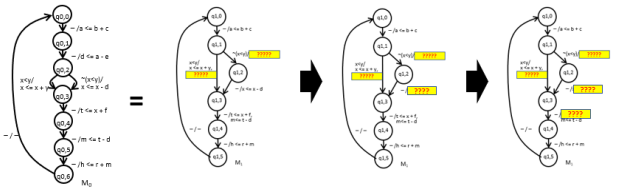


図 9. 実験 2 で用いた FSMD

表 2. 実験 2 結果

blanks	2	3	4	5	6	7
equivalence check	31	32	40	40	40	38
time[s]	21.7	24.8	52.9	70.5	103.6	190

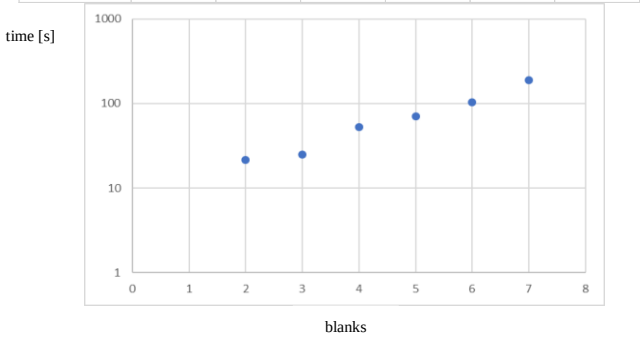


図 10. 実験 2 結果

軸に実行時間を対数目盛で、横軸にブランクの数を取っている。グラフはおおよそ直線であるため実行時間はブランクに対して指数的に変化しているということが分かるが、これは解候補を生成する際の探索範囲がブランクの数に対して指数的になるためそれに伴って実行時間も指数的に大きくなるからだと考えられる。

3. 実験 3

提案手法の欠点の 1 つとして、対応の取れないパスを含む FSMD 間に提案手法を適用すると、Karfa の手法を適用する部分で、小さなパスから延長された大きなパスまでの全てに対して対応が取れないことを確認しなければならないため、実行時間が大きくなってしまう可能性があることが挙げられる。そこで実験 3 では、対応の取れないパスを含む FSMD 間で実行時間を測定し、実行時間がどの程度変化するのか調べることとした。

表 3 と図 11 に結果を示した。ここで図 11 の縦軸は実行時間を、横軸は対応の取れないパスの数をそれぞれ線形に取っている。グラフはおよそ直線であるため実行時間は線形に変化していると言えるが、それについては以下のような考察が考えられる。

表 3. 実験 3 結果

not corresponding paths	0	1	2	3	4	5	6	7
equivalence check	323	371	536	714	815	979	1021	1126
time[s]	38.2	46.1	63.6	82.3	92.9	112	117	131

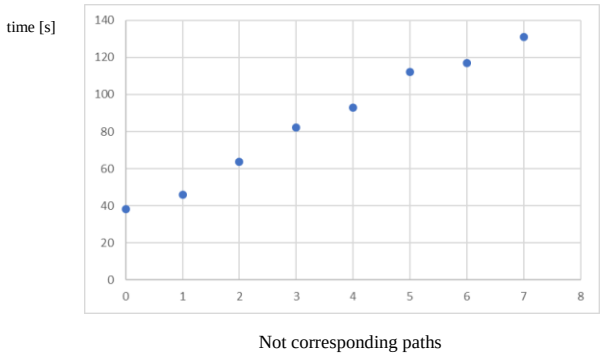


図 11. 実験 3 結果

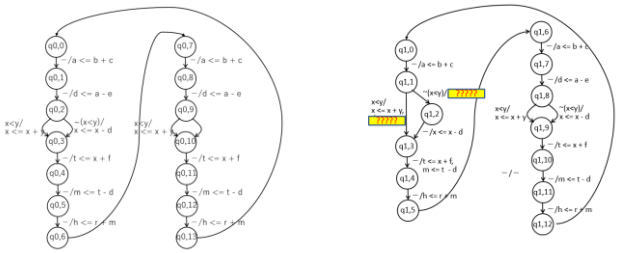


図 12. 繋がれた FSMD

表 4. 実験 4 結果

connected systems	5	10	15	20	25	30
equivalence check	570	1297	2027	2754	3490	4214
time [s]	90.2	175	258	310	410	474

まず、以下のように文字を定める。

- 対応が取れるパスの数 : N1
- 対応が取れないパスの数 : N2
- パスの対応を見つけるためにかかる時間: T1
- 対応が取れないことを確認するためにかかる時間: T2

このとき、提案手法の前半であるパスの対応を取る部分にかかる時間は N1*T1+N2*T2 である。

さて、この状態からさらに 1 つのパスで対応が取れなくなった時、パスの対応にかかる時間はおおよそ(N1-1)*T1+(N2+1)*T2 であり、その差分は T2-T1 である。この値は N1 や N2 の値に因らないので実行時間の増加量は対応が取れる/取れないパスの数に因らず一定であるから、線形な変化になっていると考えられる。

4. 実験 4

実験 4 では、機械的に大きな FSMD を作り、それらを用いて合成を実行した。具体的には、図 4 のシステムそれぞれの後ろに図 4 の左側のシステムをいくつか繋ぎ、それらの最後の状態を最初の状態と繋ぐことで FSMD として合成をした。システムを 1 つ繋いだ状態の FSMD を図 12 に示した。なおこのモデルは、大きなプログラムなどにおいてその一部に最適化を施して FSMD に変換した状況を模している。

結果を表 4 に示した。結果から、ある程度の大きさの FSMD についてこの手法で問題なく合成が完了することが分かった。

V. まとめと今後の展望

実験結果から、提案手法は以下のような特徴を持つことが分かった。

- プログラムと RTL 記述の両方に関して部分合成ができる。
- 実行時間は主にパスとブランクの数に影響され、特にブランクの数に対して指数的に変化する。
- 等価でない FSMD 同士で合成を実行したときは、対応が取れないパスの数に対して線形的に実行時間が変化する。

今後の展望の一つとして、提案手法を最新の HW 設計手法[4]に応用することが考えられる。この設計手法は、HW を SW で用いられているようにトップダウン法[5]に則って設計するものだが、その中で RTL 記述ではなく複数クロックからなる命令の集合をしようとして与えるので、提案手法を応用して 2 つの FSMD のうち片方はそのまま、もう片方はパスの集合として与えて合成を実行するような実装ができればこの設計手法への応用が可能だと考えられる。

参考文献

[1] Patrick R. Schaumont “Finite State Machine with Datapath” A Practical Introduction to Hardware/Software Codesign pp 113-156 06 October 2012

[2] C Karfa, C Mandal, D Sarkar, S R Pentakota “A Formal Verification Method of Scheduling in High-level Synthesis” Quality Electronic Design, 2006. ISQED ’06. 7th International Symposium on

[3] M.Fujita “Toward Unification of Synthesis and Verification in Topologically Constrained Logic Design“ Proceedings of the IEEE (Volume:103 Issue:11 Nov.2015)

[4] J Urdahl, S Udupi, T Ludwig, D Stoffel, W Kunz “Properties first? A new design methodology for hardware, and its perspectives in safety analysis” Computer-Aided Design (ICCAD), 2016 IEEE/ACM International Conference on K. Elissa, “Title of paper if known,” unpublished.

[5] K Beck (2003) “Test-Driven Development By Example” Addison-Wesley Professional