Vol.2014-SLDM-165 No.18
Vol.2014-EMB-32 No.18
2014/3/15

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

松本 剛史[1,a)]　城 怜史[2,b)]　藤田 昌宏[1,c)]

概要：近年、Look-up Table(LUT) のようなプログラム可能回路を利用した論理回路の自動デバッグ手法が提案されている。この手法では、回路中に LUT を挿入し、SAT ソルバーを用いて、回路が仕様を満たすような LUT の論理定義を求めることによって論理修正を行う。この考え方は、LUT の代わりに Uninterpreted Function(UIF) を利用することによって、ワード変数とそれらに対する演算が記述される高位設計に適用することができる。この場合、SAT ソルバーの代わりに、ワード変数や算術演算を含む論理式の SAT 問題を効率良く解くことができる SMT ソルバーが使われる。しかし、一般的に、UIF を利用した手法においては、得られた誤り修正の解を限られた種類の利用可能な演算 (器) で実現することは容易ではない。そこで、UIF ではなく、プログラム可能なデータパスを利用することにより、利用可能な演算によって実装可能な誤り修正を求める手法を提案する。提案するプログラム可能データパスでは、与えられた演算を最大 N 回まで適用する任意のデータパスが表現可能であり (N は指定された数)、その中に仕様を満たす構成が含まれるかを SMT ソルバーで解く。いくつかの例題に対して、仕様を満たすプログラム可能データパスの構成を求める実験を行い、その性能を評価する。

キーワード：高位設計, デバッグ支援, SMT ソルバー

Debugging High Level Designs using Programmable Datapaths and SMT Solvers

Takeshi Matsumoto[1,a)]　Satoshi Jo[2,b)]　Masahiro Fujita[1,c)]

Abstract: In this paper, we propose a method to correct high level designs. Recentrly, methods to rectify logic gate circuits utilizing programmable logics based on look-up tables (LUTs) inserted in the circuits for debugging purposes. The idea can be extended to high level (or word level) designs by using uninterpreted functions (UIFs) instead of LUTs. In such methods, some portions of a design under debugging are replaced with UIFs, and their definitions which can correct the design are explored by formulating them as satisfiability problems and solving them by SMT solvers, which are SAT solvers which can efficiently handle word variables and arithmetic operations. However, in general it is not at all easy to implement them with limited types of available arithmetic circuits. Our proposed method utilizes programmable datapaths, instead of UIFs, which realize any combination of applications of specified types of operations at most N times (N is a user-specified integer). Then, a correction of a given design can be obtained as a configuration of programmable datapaths, which can be solved similarly to the conventional methods using UIFs. We have confirmed that high level designs can be corrected through experiments with several example cases.

Keywords: High level design, Debugging support, SMT solver

[1] 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
[2] 東京大学工学系研究科電気系工学専攻
Dept. of Electrical Engineering and Information Systems, The University of Tokyo
[a)] matsumoto@cad.t.u-tokyo.ac.jp
[b)] jo@cad.t.u-tokyo.ac.jp
[c)] fujita@ee.t.u-tokyo.ac.jp

Vol.2014-SLDM-165 No.18
Vol.2014-EMB-32 No.18
2014/3/15

1. はじめに

ハードウェア設計を高位から開始する場合、RTL 以降の設計記述に設計誤りを残してしまうことによって生じる手戻りを避けるためにも、高位設計における検証によって可能な限り多くの設計誤りを発見・修正することが重要である。このため、高位において多くのバグ修正作業が行われることになるため、高位設計に対するデバッグ支援技術の必要性も高い。本研究では、高位設計において、仕様を満たすような設計修正方法を自動的に求める手法を提案する。

文献 [1] では、SAT 問題 (充足可能性判定問題) を利用した論理ゲート回路に対する誤り位置特定手法が提案されている。この手法では、与えられた全ての反例に対して回路が正しく動作するためには、どのゲートにおける論理関数を変更する必要があるかを求めている。具体的には、回路中のゲート 1 つずつに対して、ゲート出力にマルチプレクサ (MUX) を挿入し、元の回路を実行して得られる値と任意の値を選択できるようにし、どのゲートにおいて元の回路とは異なる値を選択すれば回路が仕様を満たすことができるか、を SAT 問題として定式化している。その結果、設計中の修正箇所候補を求めることができる。この考え方は、RTL 設計にも適用可能である [2]。RTL では、ビットベクタとして表現されたワード変数やそれらに対する算術演算を効率的に扱う必要があるため、SAT ソルバーに代わりに SMT ソルバーが用いられる。

前述の手法では、誤り箇所の候補は得られるが、修正方法は得ることができない。そこで、文献 [3] では、回路中のゲート出力に MUX を挿入するのではなく、ゲートを Look-up Table(LUT) で置換することによって、そのゲートの論理関数を任意に変更できるように定式化し、LUT の論理定義を求めることによって修正方法の候補を得る手法が提案されている。仕様を満たすような LUT の論理定義を求める手法の詳細については、次節で述べる。この手法も、LUT の代わりに Uninterpreted Function(UIF) を用いることによって、RTL 設計や高位設計に適用可能である。この場合、与えられた反例に対して設計が仕様通りの動作をするように、設計中の UIF の定義を SMT ソルバーによって求めることになる [5]。

高位設計中のワード変数に対する論理演算や算術演算は、RTL 設計に変換する際には、与えられたライブラリ中にある利用可能な演算器のいずれかにマッピングされる必要がある。しかし、文献 [5] の手法では、誤り修正方法として UIF の定義のみを得る。UIF の定義は、多くの SMT ソルバーでは、算術式のような形ではなく、入出力関係の列挙として得られる。また、その定義が部分的なものであることも多い (つまり、全ての入力空間に対して定義されるわけではない)。このような (部分的な) 入出力関係の列挙として表現された誤り修正方法から、利用可能な演算器による実装を求めることは容易ではない。図 1 は、この UIF を利用したデバッグ例を示している。ここでは、元の設計にある $in1 + (in2 << 1) + (in3 << 2)$ が誤りを含んでおり、正しくは $in1 + (in2 << 2) + (in3 << 4)$ であるとする。この例で示すような算術演算は、フィルタ設計などで頻繁に行われているものである。文献 [5] のような手法では、元の算術式を UIF $F(in1, in2, in3)$ で置換し、与えられた反例 (入力パタン) に対して仕様通りの結果が得られるような UIF の定義を SMT ソルバーによって求める。この例では、入力パタンとして $(in1, in2, in3) = (0, 0, 1), (0, 1, 1,), (1, 1, 2)$ を与えることとする。その結果として得られる UIF の定義は、例えば、$F(0, 0, 1) = 8 \wedge F(0, 1, 1) = 12 \wedge F(1, 1, 2) = 21$ のようなものであり、利用可能な演算器を考慮した実装・算術式表現とはかけ離れたものである。

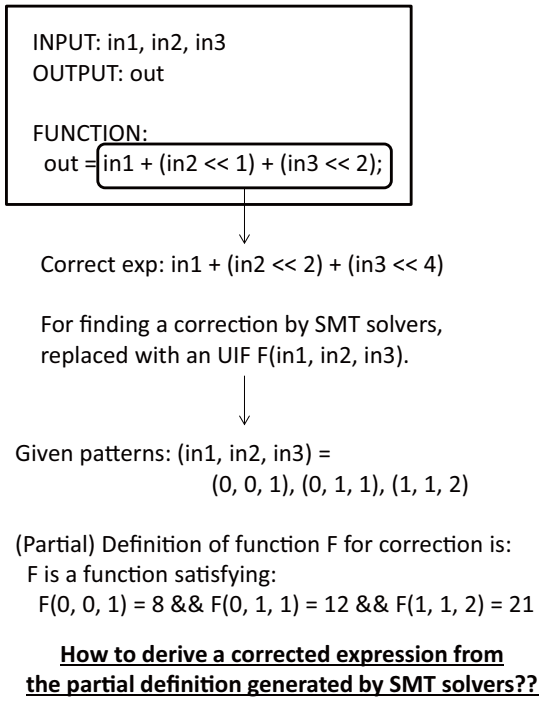

図 1 UIF を用いたデバッグ支援手法

本研究では、UIF の代わりに、修正箇所をプログラム可能なデータパスとして表現することにより、利用可能な演算器で実装可能な誤り修正方法を求める手法を提案する。図 2 は、本研究で用いるプログラム可能なデータパスの例である。詳細は第 3 節で述べる。提案手法では、修正方法がこのプログラム可能なデータパスの構成として得られるため、修正方法を利用可能な演算器によって実現することができる。

本稿の構成は、以下の通りである。第 2 節で、関連研究として、SAT ソルバーを用いた誤り位置特定手法と修正方法導出手法について述べる。第 3 節で、提案するプログラム可能データパスを用いた誤り修正方法導出手法を述べる。第 4 節で、いくつかの例題に対する実験結果を示し、第 5 節で結論を述べる。

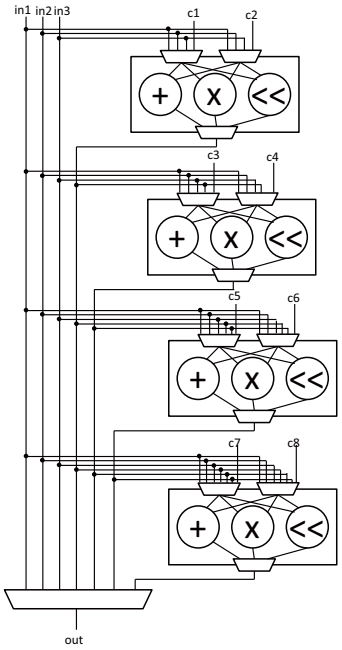

図 2 プログラム可能データパスの例

2. 関連研究

2.1 マルチプレクサを利用した誤り位置特定

第 1 節で述べたように、ゲート回路や RTL 回路に MUX を挿入し、仕様を満たすためには、どの論理ゲート (RTL であれば演算器) の出力値を変更する必要があるかを SAT 問題へ定式化することによって求める手法が提案されている [1], [2]。図 3 は、ゲート回路に対して、この手法における MUX 挿入例を示したものである。図 3(a) が誤りを含む元の回路であり、入力パタン $\{x_1 \leftarrow 1, x_2 \leftarrow 0, x_3 \leftarrow 1, x_4 \leftarrow 0\}$ が反例であるとする。この入力パタンに対する元の回路の出力は $z = 1$ となるため、正しい出力値は $z = 0$ である。このとき、誤り位置特定手法は、図 3(b) に示すような MUX が挿入された回路を生成する。回路中に挿入された 2 つの MUX によって、変数 s_1, s_2 の値に応じて、変数 l_1, l_2 は、元の回路中で計算された値と自由変数 v_1, v_2 をそれぞれ選択できるようになる。この MUX が挿入された回路、与えられた反例パタン (入力パタン)、および、それに対する正しい出力値から以下の式を生成する。

$$F_{with\_mux} \wedge (x_1 \wedge \neg x_2 \wedge x_3 \wedge \neg x_4) \wedge \neg z \qquad (1)$$

F_{with_mux} は MUX 付き回路の論理、$x_1 \wedge \neg x_2 \wedge x_3 \wedge \neg x_4$ は入力パタン、$\neg z$ は正しい出力値を表している。この式が充足可能である場合、自由変数 v_1, v_2 を導入することにより、出力値を正すことができる。具体的にどの変数を自由変数で置換すればよいかは、SAT ソルバーが出す変数割当てから分かる。ワード変数で表現された回路に対して適用する場合には、Boolean SAT 問題ではなく、SMT 問題 (ワード変数や算術演算を含む論理式の充足可能性判定問題) として同様に定式化することができる。

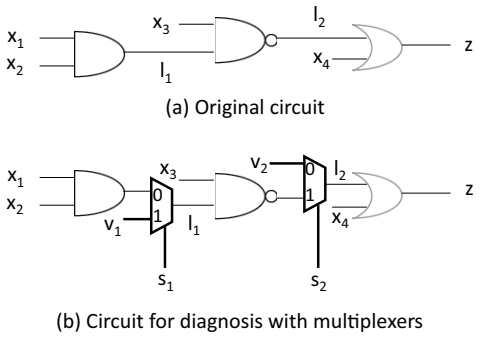

図 3 誤り位置特定手法におけるマルチプレクサ挿入

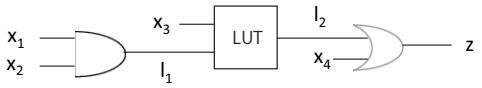

図 4 誤り修正を求めるための look-up table による置換

2.2 Look-up table を利用した誤り修正方法の導出

回路中に MUX を挿入する代わりに、文献 [3] で提案されている手法では、一部のゲートを LUT で置換することにより、誤り修正方法を求めることを実現している。図 4 に示すように、一部のゲートを LUT で置換し、仕様を満たすような LUT の論理定義を求めることによって、正しい回路においてそのゲートで実現するべき論理関数が分かるため、誤り修正を行うことができる。この問題は、以下の QBF 式の真偽を判定する問題として定式化できる。

$$\forall x_1, x_2, x_3, x_4 \exists v_{lut} (F_{with\_lut} = Spec), \qquad (2)$$

v_{lut} は LUT の真理値表を表す変数、F_{with_lut} は LUT 付き回路の論理、$Spec$ は回路の仕様をそれぞれ表す。この式が真と評価される場合、回路を仕様と等価にする LUT の定義 (v_{lut}) が存在し、それによって回路を修正可能である。

この QBF 式の真偽判定を SAT 問題を繰り返し解くことによって行う手法が提案されている [3], [4], [5]。そこでは、まず、全ての入力パタンではなく、いくつかの入力パタンについて、$F_{with\_lut} = Spec$ を満たす LUT の定義を求める。これは、次の式の SAT 問題を解くことで得られる。

$$(F_{with\_lut}(p_1, v_{lut}) = Spec(p_1)) \wedge ... \\ \wedge (F_{with\_lut}(p_n, v_{lut}) = Spec(p_n)), \qquad (3)$$

$p_1, ..., p_n$ は入力パタンを表す。この式が充足可能である場合、得られた v_{lut} が LUT 定義の候補である。次に、この候補が全ての入力パタンに対して成り立つかどうかを以下のどちらかの方法で確認する。成り立たない場合には、反例として新たな入力パタンが得られ、それを式 (3) に追加し、QBF 式を満たす LUT の定義が求まるか、式 (3) が充足不可能となるまで繰り返す。

- **等価性検証** LUT の論理が v_{lut} である場合の F_{with_lut} と仕様 $Spec$ が等価であるかを判定し、非等価の場合には、その反例を p_{n+1} として追加する

- **唯一性検証** 既存のパタン $p_1, ..., p_n$ に対しては F_{with_lut} が等価になるが、ある他のパタン p_{n+1} に対しては非等価になるような v_{lut} とは異なる v'_{lut} が存在するかどうかを SAT 問題として解くことができる。そのような v'_{lut} が存在する場合には、p_{n+1} が追加するパタンである

本研究においても、プログラム可能データパスの構成を求める問題は QBF 式の真偽判定として定式化できるため、本節で述べた手法によって、SAT ソルバーを用いて解を求めることとする。

3. 提案手法

3.1 全体の流れ

図 5 に提案手法の全体の流れを示す。誤り修正方法の導出は、与えられた高位設計、入力パタン、仕様に対して行われる。仕様については、本研究では、等価性検証が可能であることを前提とせず、シミュレーション可能であれば良い。これは、高位設計においては、MATLAB や C プログラムといったシミュレーションモデルが仕様となることが多いためである。

提案手法では、まず、誤りを含む高位設計をデータフローグラフ (DFG: Data Flow Graph) または制御データフローグラフ (CDFG: Control Data Flow Graph) で表現する。次に、設計のデータフローの一部分をプログラム可能データパスで置換する。これは、既存手法において、論理ゲート回路の一部を LUT で置換したり、高位設計の一部を UIF で置換することに相当する。設計にループがある場合には、一定回数の展開を行う必要がある。ループ展開を行った場合、得られる誤り修正方法は、展開回数以下のループ実行を正しくすることができる方法となり、それを超える繰り返し回数のループ実行については正しい誤り修正とならない可能性がある。プログラム可能データパスと置換する箇所については、第 2.1 節で紹介したような誤り位置特定手法で得られる結果を利用することとし、本論文における議論の対象としない。

最後に、誤り修正方法として、プログラム可能データパスを含む設計が仕様と同じ動作をするためのデータパスの構成を求める。この修正方法の求め方は、第 2.2 節で述べた、SAT 問題 (提案手法では、ワード変数と算術演算が含まれるため SMT 問題) を繰り返し解くことによって QBF 問題を解く手法に基づく。このとき、仕様と設計の等価性検証が可能であることを前提としていないため、修正方法が正しいかどうかは等価性検証ではなく、第 2.2 節で述べた解の唯一性の確認によって行う。既存手法では、誤り修正方法は、図 1 で示したような入出力対応の集合ではなく、データパスの構成として得られるため、利用可能な演算器を用いてどのように誤り修正を実装すれば良いかを知ることができる。

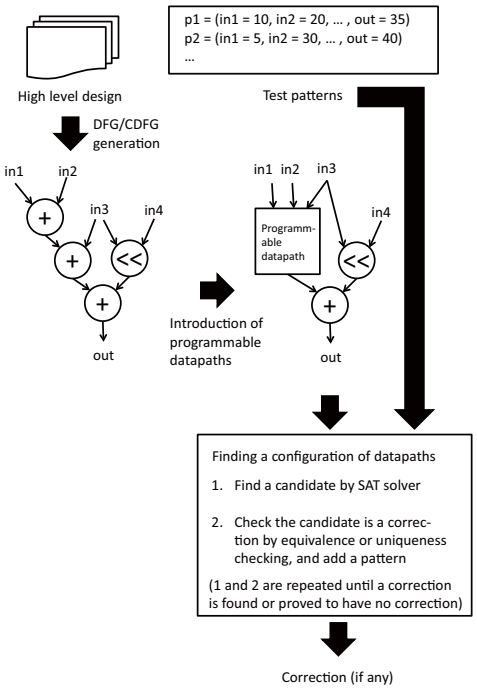

図 5 全体の流れ

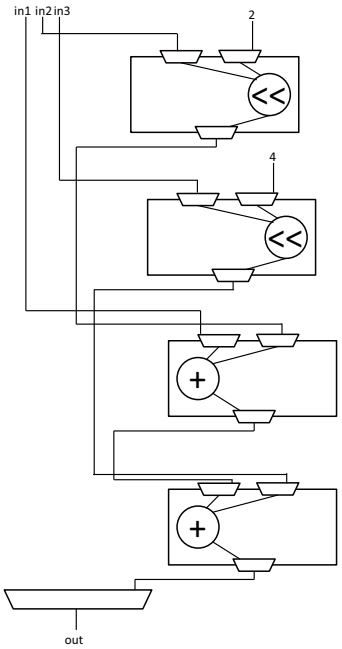

図 6 図 1 の例に対する修正方法の例

3.2 プログラム可能データパス

DFG の一部を UIF で置換し、設計を正しくするための UIF の定義を求めることによって誤り修正方法を求める場合、任意の関数が修正方法の候補となる。それらの中には、利用可能な演算器の組合せでは実装が不可能、あるいは、非常に困難な関数が含まれている。しかし、設計誤りのある高位設計は、設計者の意図通りに正しく設計されていた場合でも、利用可能な演算器の組合せで実現可能 (合成可能) であるはずである。そのため、利用可能な演算器で実現が困難な修正方法は、初めから候補から外してしまって

も構わないと考えられる。本論文で提案するプログラム可能データパスでは、構成を変えることにより、あらかじめ与えられた利用可能な演算器による演算を最大 N 回まで含む任意のデータフローを実現可能である。このプログラム可能データパスの例を図 2 に示した。

一般に、プログラム可能データパスは以下の 3 つのパラメータに対して定義できる。

- データパスへの入力変数の数 N_{in}
- 最大演算回数 N_{op}
- 適用可能な演算種類の集合 OP

図 2 に示す例では、$N_{in} = 3$, $N_{op} = 4$, $OP = \{Add, Mult, Shift\}$ である。プログラム可能データパスでは、OP に含まれる演算を任意に最大 N_{op} 回実施可能であり、これは図ではそれぞれ四角で囲まれた部分に相当する。それぞれの演算 (四角で囲まれた部分) には、入力側に 2 つと出力側に 1 つの MUX がある。入力側から n 個目の演算における入力側 MUX では、N_{in} 個の入力変数、1 個目から $(n-1)$ 個目までの演算出力、1 つの定数値から、演算の入力となる 1 つを選択する。また、演算の出力側 MUX では、演算種類の選択を行う。最終的に、N_{in} 個の入力変数と N_{op} 個の演算出力から 1 つの値を選択し、プログラム可能データパス全体の出力とする。プログラム可能データパスによって置換された部分が複数の出力を持つ場合、それぞれの出力について、プログラム可能データパスによる置換を行う。

3.3 修正方法の導出

本節では、一部がプログラム可能データパスで置換された DFG に対して、DFG 全体が仕様通りに正しくなるようなデータパスの構成 (データパスにおける MUX の制御変数値と定数値) を求める手法を述べる。まず、第 2.2 節で述べた既存手法と同様に、いくつかの入力パタン $p_1, ..., p_n$ に対して、仕様を満たすデータパスの構成を以下の式の充足可能性判定問題として解く。

$$(F_{pd}(p_1, v_{mux}) = Spec(p_1)) \wedge ... \\ \wedge (F_{pd}(p_n, v_{mux}) = Spec(p_n)) \quad (4)$$

$F_{pd}(p_i, v_{mux})$ は、プログラム可能データパスの構成が v_{mux} であるときに、入力パタン p_i に対する DFG 全体の出力値を表す。また、$Spec(p_i)$ は、入力パタン p_i に対する出力値の仕様を表す。v_{mux} は、データパス中の全ての MUX 制御変数と定数のベクトルとして表される。なお、$Spec(p_i)$ の値を求めるための仕様は、高位設計や回路として記述されている必要はなく、与えられたパタンに対する正しい出力値を生成できるものであればよいため、高位合成不可能なシミュレーションモデルであっても構わない。式 (4) の充足可能性は、SMT ソルバーによって解くことができる。充足可能である場合、v_{mux} の値を得ることができる。充足不可能である場合には、データパスがどのような構成をとっても、設計を修正することができないことが証明される。

次に、得られた v_{mux} が設計修正の本当の解であるかどうかを検証する。ここでは、第 2.2 節で述べた解の唯一性を調べることによって、v_{mux} が解であるかどうかを判定する。これは、既に与えられている入力パタン $p_1, ..., p_n$ の全てに対して仕様通りの正しい値を出力するが、新たな入力パタン p_{n+1} に対しては異なる値を出力する異なる v_{mux} とは異なる構成 v'_{mux} が存在するかどうかを解くことによって行われる。そのような入力パタン p_{n+1} が存在しない場合には、$p_1, ..., p_n$ に対して正しい値を出力する構成 v_{mux} は、その他の任意のパタンに対しても同じ出力値を持つことになり (そのため、区別することができない)、条件を満たす構成が唯一 v_{mux} のみであることになる。これは、以下の式の充足可能性判定問題に帰着される。

$$((F_{pd}(p_1, v'_{mux}) = Spec(p_1)) \wedge (p_1 \neq p_{n+1})) \wedge ... \\ \wedge((F_{pd}(p_n, v'_{mux}) = Spec(p_n)) \wedge (p_n \neq p_{n+1}) \\ \wedge(F_{pd}(p_{n+1}, v_{mux}) \neq F_{pd}(p_{n+1}, v'_{mux})) \\ \wedge(v_{mux} \neq v'_{mux}) \quad (5)$$

この式が充足不可能である場合には、v_{mux} に対応するプログラム可能データパスの構成を誤り修正方法の解とする。充足可能である場合には、得られた p_{n+1} を式 (4) に追加し、繰り返す。

図 6 は、図 1 で示した例に対して、提案するプログラム可能データパスを利用した誤り修正方法導出手法によって得られた修正方法の例である。この図では、見やすくするために、MUX で選択されない演算や変数は除かれている。この結果から、プログラム可能データパスによって置換された部分の正しい関数は、$in1 + (in2 << 2) + (in3 << 4)$ であり、この関数が与えられた演算種類 (加算、乗算、左シフト) によって実現可能であることが分かる。

4. 実験結果

本節では、第 3 節で述べた手法をいくつかの例題に適用し、その結果と実行時間を示す。例題としては、図 1 に示した例、与えられた数の中で"1" のビット数を求める計算 (図 7)、与えられた整数を超えない最大の 2 の冪乗を求める計算 (図 8) の 3 つを用いた。プログラム可能データパスで置換した部分は、それぞれの図中で示されている。SMT ソルバーとしては、Microsoft 社で開発された z3[6], [7](version 4.3.1) を用いた。本節で報告する全ての実験結果は、8GB のメモリと Intel Xeon 2.90GHz を有する計算機上で行った。

表 1 に実験結果を示す。表中では、図 1、図 7、図 8 に示す例題は、ぞれぞれ filter, bit_count, power と記されている。表中の UNSAT は、新たに追加すべき入力パタンが存在しない場合 (充足不可能になる場合) の SMT 問題を解く

表 1 実験結果

例題	プログラム可能データパス			結果	実行時間		繰返し
	N_{in}	N_{op}	OP		全体	UNSAT	回数
filter (8 bit)	3	4	$\{+, *, <<\}$	成功	30 秒	26 秒	4
filter (16 bit)	3	4	$\{+, *, <<\}$	成功	285 秒	264 秒	6
filter (32 bit)	3	4	$\{+, *, <<\}$	Time out	>3 時間	—	≥3
bit_count	1	3	$\{+, >>, \&\}$	成功	2,442 秒	1,993 秒	13
power (a)	1	3	$\{>>, \mid\}$	成功	8 秒	2 秒	5
power (b)	1	6	$\{>>, \mid\}$	Time out	>3 時間	—	≥11

```
uint32_t bit_count(uint32_t x) {
  x = (x & 0x55555555) + ((x >> 1) & 0x55555555);
  x = (x & 0x33333333) + ((x >> 2) & 0x33333333);
  x = (x & 0x0F0F0F0F) + ((x >> 4) & 0x0F0F0F0F);
  x = (x & 0x00FF00FF) + ((x >> 8) & 0x00FF00FF);
  x = (x & 0x0000FFFF) + ((x >> 16) & 0x0000FFFF);
  return x;
}
```

図 7 ”1” のビット数を数える計算

```
uint32_t largest_pow(uint32_t x) {
  x = (x >> 1) | x;
  x = (x >> 2) | x;
  x = (x >> 4) | x;                    (b)
  x = (x >> 8) | x;
  x = ((x >> 16) | x) >> 1;
  return x;
}                                      (a)
```

図 8 入力値を超えない最大の 2 の冪乗を求める計算

ために要した時間である。誤り修正方法を求めることができた場合、データパスの構成を決定するために必要なパタン数は非常に少ないことが分かる。また、充足不可能となる最後の SMT 問題を解くための時間が全体の実行時間のほとんどを占めている。タイムアウトとなった 2 つのケースでも、正しい修正方法が数分以内に得られているが、その解が唯一であるかの確認に時間がかかったためにタイムアウトとなっている。これらの結果から、提案手法で誤り修正方法の解が見つかる場合、その解の唯一性の確認が実行時間上のボトルネックになっていると言え、手法改善等による高速化が必須である。

5. 結論と今後の課題

ワード変数を含む高位設計のデバッグ支援において、従来の Uninterpreted Function を利用した誤り修正方法の導出では、得られた修正方法を利用可能な演算器を用いて実装することが容易ではないという問題があった。本研究では、これを解決するために、与えられた演算の最大 N 回までの任意実行を表すことができる、プログラム可能データパスを利用した誤り修正方法の導出を提案した。提案手法では、誤り部分をこのデータパスで置換した上で、SMT ソルバーを繰り返し実行することによって、設計が仕様通りの動作をするようなデータパスの構成を求め、それによって誤り修正方法を得ることができる。いくつかの例題に対する実験を通して、提案手法によって実際に修正方法を得ることができることを示した。

今後の課題として、SMT ソルバーによって得られるデータパス構成候補の検証の高速化が挙げられる。前節でも議論した通り、提案手法では、解候補は短時間で求めることができるが、その解の唯一性の検証時間 (特に、解が唯一である場合の SMT 問題を解く時間) が非常に長くなっている。この問題の解決方法としては、UIF を利用した既存手法を併せて適用することが考えられる。文献 [5] では、修正を表す UIF の定義自体は比較的短時間で求められている。そこで、まず、UIF の定義として修正を求め (この時点で、その解の唯一性の検証は完了している)、それを満たすようなプログラム可能データパスの構成を求めることにより、提案手法において非常に長い実行時間を要するプログラム可能データパス構成候補の唯一性検証を除くことができると期待できる。

参考文献

[1] A. Smith, A. Veneris, M. F. Ali, and A. Viglas, “Fault Diagnosis and Logic Debugging Using Boolean Satisfiability,” *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, Vol.24, No.10, pp.1606–1621, Oct. 2005.

[2] S. Mirzaeian, F. Zheng, and K.-T. Cheng, “RTL Error Diagnosis Using a Word-Level SAT-Solver,” *Proc. of IEEE International Test Conference 2008*, pp.1–8, Oct. 2008.

[3] S. Jo, T. Matsumoto, and M. Fujita, “SAT-Based Automatic Rectification and Debugging of Combinational Circuits with LUT Insertions,” *Proc. of IEEE 21st Asia Test Symposium 2012*, pp.19–24, Nov. 2012.

[4] M. Janota, W. Klieber, J. Marques-Silva, and E.M. Clarke, “Solving QBF with Counterexample Guided Refinement,” *Proc. of 15th International Conference on Theory and Applications of Satisfiability Testing*, pp.114–128, 2012.

[5] M. Fujita, T. Matsumoto, and S. Jo, “Partial synthesis through sampling with and without specification,” *Proc. of International Conference on Computer Aided Design 2013*, pp.787–794, Nov. 2013.

[6] L. de Moura and N. Bjoerner, “Z3: An Efficient SMT Solver,” *Proc. of International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp.337–340, 2008.

[7] Z3 Home: http://z3.codeplex.com/