Vol.2011-SLDM-150 No.12
2011/5/19

反例と設計分割に基づく高位設計に対する効率的な設計修正支援手法

原 田 裕 基[†1]　松 本 剛 史[†2]　藤 田 昌 宏[†2,†3]

高位設計記述において、シミュレーションや形式的手法によって機能仕様に反する実行例 (反例) が発見された場合、その反例や機能仕様を参照しながら、設計記述をデバッグする必要がある。本稿では、このように反例に基づくデバッグ作業を支援する手法を提案する。具体的には、与えられた反例および正しい実行例から、全てのテストパタンを正しく実行するための設計記述修正の候補を形式的に求める。これにより、設計者は、修正すべき箇所と修正方法の候補を得ることができ、より効率的にデバッグ作業を行えることが期待できる。提案手法では、反例入力パタンによって正しい実行結果を得るためには、どの変数値を実行値とは異なる値に置換すれば良いか、を SMT ソルバーを用いて解いている。加えて、効率的に修正候補を求めるために、設計を分割し、部分的にこれを適用する手法を提案する。実験により、提案手法によって、設計中の設計誤りを正す修正を求めることがでることを示す。

Efficient High-Level Design Correction Support Based on Counterexamples and Design Division

Hiroki Harada,[†1] Takeshi Matsumoto[†2]
and Masahiro Fujita[†2,†3]

When one or more counterexamples are found by either simulation or formal methods in high-level design verification, we need to modify the given design descriptions by seeing the counterexamples and functional specification. In this paper, we propose a method to find design correction candidates to make the design under debugging behave correctly for all input patterns of counterexamples. With those correction candidates, efficiency of debugging will be improved. The proposed method solves which variable values should be replaced by other values from the originally assigned during the execution in order to make the output values correct for all given counterexamples. Also, to improve the efficiency of the method, we propose to divide a design into smaller portions and apply the method locally. Through the experiments, we show that the proposed method can derive the design correction that makes an erroneous design correct.

1. はじめに

C ベース設計記述言語によって、抽象度の高い記述からハードウェア設計を始める高位設計においては、設計の機能的な正しさの検証を高位において効率的に行うことができるという利点がある。同時に、RTL(Register Transfer Level) 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・修正することが高位設計では重要になっている。この高位設計における検証の重要性の高まりに伴って、高位設計に対するデバッグ支援の必要性も高まっている。本研究では、高位設計において反例が見つかった場合に、その反例に対する設計修正を支援する手法を提案する。

提案手法では、反例と同じ入力パタンを設計に与えた場合、設計中のどこを変更すれば正しい出力値が得られるか、を解くことによって、修正の候補を求める。より具体的には、与えられた入力パタンを実行した際に、設計記述中の各変数に代入される値と他の値の入れ換えを可能にした上で、どの変数値をどのような値と入れ換えれば、反例の出力が正しい出力になるかを求める。このとき、変数値の入れ換えが必要な箇所が修正が必要な箇所の候補となる。この定式化はワードレベル変数と演算子を用いた一階述語論理式として行い、SMT ソルバーを用いて解いている。

図 1 は、手法の基本的な考えを表した例である。図の (a) が仕様、(b) がそれを実装した高位設計記述である。ただし、(b) は設計誤りを含んでいる。この設計記述において、$\{A \leftarrow 10, B \leftarrow 5, in1 \leftarrow 3, in2 \leftarrow 2\}$ は仕様に対して反例となる。この反例の実行におけるデータフローを表したのが図の (c) である。反例の実行においては、変数 $t2$ の値は 15 であるが、これを -15 で置換することにより、出力値を正しくすることができる。提案手法では、この例のように、反例実行中で計算された値のうち、出力値を正しくできるような変

†1 東京大学大学院工学系研究科電気系工学専攻
Department of Electrical Engineering and Information Systems, The University of Tokyo
†2 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
†3 科学技術振興機構 戦略的創造研究推進事業 CREST
CREST, Japan Science and Technology Agency

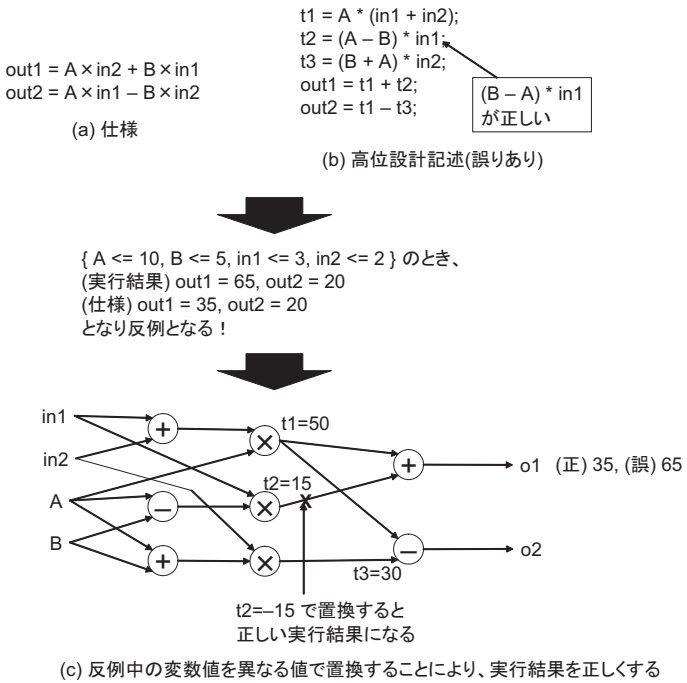

図 1 修正候補を求める例

数値の置換があるかどうか、をワードレベル論理式の充足可能性判定問題として解く。このとき、変数値の置換を行う数は、1 つでも複数でも良い。また、任意の値による置換によって、正しい出力値とできる場合、設計者はその変数値となるように設計を修正する必要がある。文献[3] では、修正方法の候補を同時に求める手法も提案している。

上述の基本手法では、設計記述が大きい場合や与える反例が多い場合、充足可能性について解くべき式が大きくなり、充足可能性を判定するための時間が長くなるという問題点がある。そこで、本稿では、反例を実行することによって得られるデータフローを分割して、部分ごとに修正個所・方法の候補を探す手法を提案する。このとき、分割されたデータフローにおける出力値を変更しても、設計全体の出力値を正しくすることができない場合には、その部分の中で反例の出力値を正しい値に変えるような修正箇所・方法が存在しない。これを利用して、そのような部分への手法の適用を省略し、効率化を図っている。実験によって、提案手法により、実際の設計例題において、反例を正すために修正すべき箇所・方法を求めることができることを示す。

提案手法では、反例とともに、反例と同じ入力パタンに対する正しい出力値が必要である。そのため、提案手法を適用するためには、仕様やゴールデンモデルの存在が前提となる。ハードウェア高位設計においては、以下のような場合での適用が考えられる。

- ソフトウェアプログラムとして記述された動作をハードウェア化した際に、その高位設計記述において設計誤り (反例) が見つかった場合
- 高位設計における設計詳細化・最適化の過程において設計誤り (反例) が見つかった場合
- アサーション検証・プロパティ検証によって、高位設計記述が満たすべき動作の性質を満たしていない反例が見つかた場合

以上の場合では、得られた反例が正しい実行例であるためには、出力がとるべき値が分かるため、提案手法を適用することが可能である。一方、正確にどの出力値が誤っているか、その正しい値は何か、が分からない場合には、提案手法を適用することができない。

本稿の構成は以下の通りである。第 2 節で、関連研究としてゲートレベル回路で設計誤り箇所の特定をする手法を紹介する。第 3 節では、反例中の変数値を置換して出力値を正しくする場合に、任意の値によって置換する場合と想定される修正方法で計算された値によって置換する場合の定式化について述べる。第 4 節で、反例トレースの分割によって効率化された手法を提案する。第 5 節で実験結果を示し、第 6 節で結論を述べる。

2. 関 連 研 究

ある論理ゲート回路または RTL 回路で発見された反例に対して、その実行における信号値のいくつかを変更することによって、正しい出力値を得ることができるかどうか、を調べ、デバッグに利用する手法は文献[1] で提案されている。この手法では、反例の実行により計算された値を異なる値と置換する可能性がある各ネットにマルチプレクサを挿入し、与えられた回路によって計算された値と任意の値のどちらかを選択できるようにする (図 2(b))。そして、マルチプレクサを挿入した回路において、反例の入力パタンを与えて正しい出力パターンを得るような各信号の値を充足可能性問題として解く。解が存在した場合、その変数割り当てにおいて、マルチプレクサの制御信号値が”1” である箇所が修正箇所の候補である。図 2 の例で、反例が $\{x_1 \leftarrow 1, x_2 \leftarrow 0, x_3 \leftarrow 1, x_4 \leftarrow 0\}$ であると仮定する。このとき、反例の出力値は $z = 1$ であるため、正しい出力値は $z = 0$ である。図 2(b) の回路に対して、同じ入力パターンで正しい出力値を得るための変数値の割り当てを充足可能性判定問

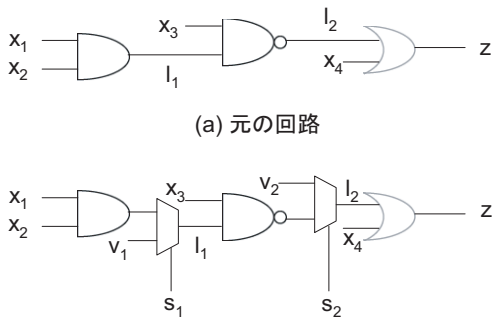

図 2 論理ゲート回路における修正箇所特定の例

題として解くと、例えば、$s_1 \leftarrow 1, v_1 \leftarrow 1$ が得られる。これは、ネット l_2 の値が回路で計算された値 ("0") ではなく、"1" に置換することによって、正しい出力値を得ることができることを意味している。

上述の Smith らによる手法がビットレベルで論理式を作り、充足可能性を解いているのに対し、文献[2),3)] では、ワードレベルで論理式を作り、同様の問題を解いている。文献[2)] では、Smith らの手法をワードレベルのデータパス回路に適用し、SMT ソルバーを用いて充足可能性を解いている。この手法では、任意の値によって元の信号値を置換するため、正しい出力値を得るために修正すべき箇所の候補数が多くなってしまい、設計者がその中から適当なものを選ぶ必要がある。そこで、文献[3)] では、修正方法に一定の仮定 (バグモデル) を置くことによって、候補の数を減らすことに成功している。ただし、バグモデルを含む定式化では、従来手法よりも式が大きくなるため、大規模な設計記述を扱うことや多くの変数について修正箇所となり得るかどうかを調べることができない。そのため、本稿では、反例の実行結果を分割することにより、より効率的に修正箇所と修正方法の候補を求める手法を提案する。

3. 提 案 手 法

3.1 全体の流れ

提案手法の全体の流れを図 3 に示す。まず、与えれた高位設計記述と反例の入力パタンから、反例の実行を表すデータフローグラフ (Data Flow Graph: DFG) を抽出する。これ

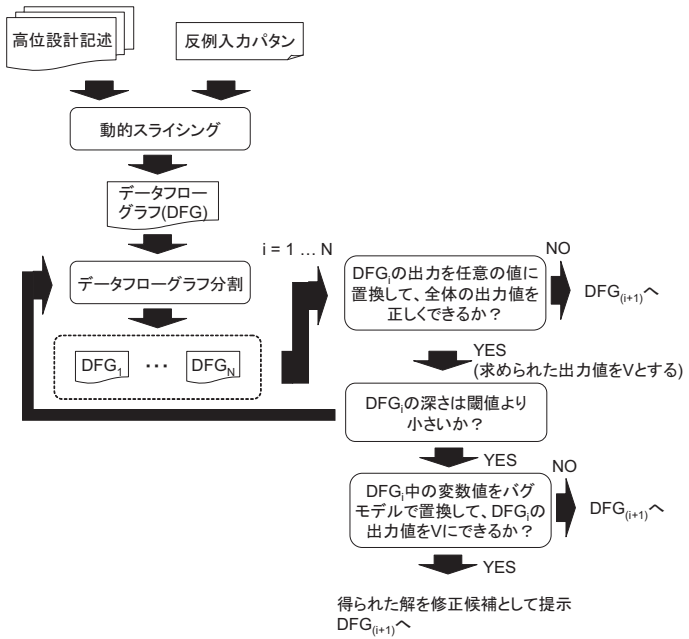

図 3 全体の流れ

は、設計記述に動的プログラムスライシング手法[4)] を適用することによって行っている。複数の反例がある場合には、DFG は反例の数だけ作られる。次に、得られた DFG を分割する。これは、DFG の深さを半分にするように分割を行う。このとき、1 つ 1 つの分割された DFG について、その出力値を変更することによって、全体の出力値を正しい値にすることができるかどうかを調べる。もし、ある分割された DFG の出力値をどのように変更しても全体の出力値を正しくできない場合には、その DFG 内の修正によって出力値を正しくすることができないため、それ以上の解析を行わない。

出力値を V に変更することによって、全体の出力値を正しい値にすることができる場合、その分割された DFG 内をどのように修正すれば、その DFG の出力値が V になるかを求める。これは、文献[3)] で提案しているバグモデルを用いた修正箇所・方法を求める手法によって行う。

3.2 反例実行を表すデータフローグラフ抽出

与えられた反例入力パタンで設計記述を実行した際のトレースをデータフローグラフとして表す。これにより、第 2 節で述べたようなマルチプレクサを挿入した設計を用いた誤り候補の特定手法を適用することができるようになる。本研究では、データフローグラフの抽出には、動的プログラムスライシング手法[4] を利用している。動的プログラムスライシングとは、実際に実行された部分のうち、指定した変数に依存を与えている部分のみを抽出する手法である。これを用いることによって、出力値に影響を与えていない部分を範囲から除くことができる。

3.3 変数値置換の定式化

本節では、提案手法において、反例の出力値を正しくするために変更すべき変数値を求めるための定式化を述べる。定式化は、任意の値によって変更する場合と、修正方法を仮定して変更する場合の 2 通りがある。

3.3.1 任意の値によって変更する場合

図 4(a) は、ある反例を実行を表したデータフローグラフである。入力変数が a, b, c、出力変数が d である。この実行において、$(a + b)$ の値を任意の値 v に変更することができるように、DFG 中にマルチプレクサを挿入したものが図 4(b) である。

このマルチプレクサが挿入された DFG のデータフローを満たすような制約式は以下のようになる。

$$(t = s?v : a + b) \wedge (d = t + c)$$

これに、反例入力パタンの値と、それに対する正しい出力パタンの値を制約として加え、充足可能性を判定する。充足可能である場合、その変数割当てにおいて、制御信号 (例では s) が”1” になったマルチプレクサが挿入された箇所 (例では t) の値をマルチプレクサの入力値 (例では v) で置換すれば、正しい出力を得ることができる。

このとき、多くのマルチプレクサの制御信号が同時に”1” になるような変数割当ては、多くの箇所を同時に修正する必要があることを意味するため、設計誤りが少数であることが仮定できる場合には、”1” となる制御信号の数に制限を加えることで、修正すべき箇所の範囲を絞ることができる。また、出力変数値そのものを変更する場合など、SMT ソルバーから得られた解が修正すべき箇所ではないと判断できる場合には、その解を除く制約を追加した上で、再度、解を求めることにより、出力値を正しくするために修正すべき箇所を探索することができる。

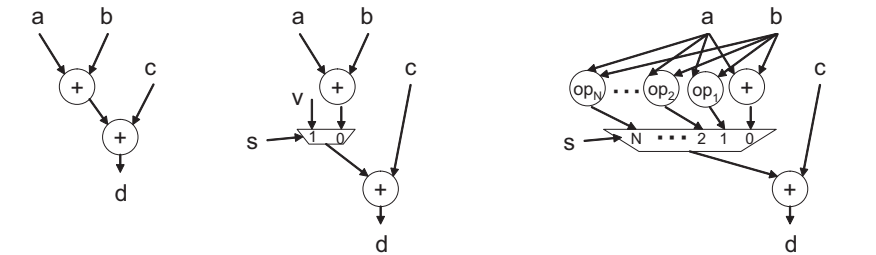

図 4 反例実行で計算された値を置換するための変換例

3.3.2 修正方法を仮定して変更する場合

前節の手法では、修正をすべき箇所の候補は分かるが、変更後に代入されるべき値しか分からないため、反例を正しい実行にするために、どのように修正をすれば良いかを知ることができない。そこで、マルチプレクサによって、任意の値を選択する代わりに、異なる演算の結果や異なる変数値を選択することによって、修正方法も合わせて求める手法も利用する。

図 4(c) は、異なる演算の結果から 1 つを選択して、元の変数値を置換できるようにマルチプレクサを挿入した例である。このマルチプレクサを含むデータフローを表す制約式は以下のようになる。

$$((s = 0) \Rightarrow (t = a + b)) \wedge ((s = 1) \Rightarrow (t = op_1(a, b))) \wedge \cdots \wedge$$
$$((s = N) \Rightarrow (t = op_N(a, b))) \wedge (s \geq 0) \wedge (s \leq N)$$

前節の任意の値によって変更する場合と同様に、反例入力パタンと正しい出力パタンの制約を加えて充足可能性を判定することにより、どの部分をどのように修正すれば、正しい結果を得ることができるかを調べることができる。バグモデルとしては、演算種類の誤り以外にも、演算に用いる変数の誤り、演算結果を代入する変数の誤り、配列インデックスの誤り、などが考えられる。

3.4 実行結果データフローの分割

バグモデルを利用して修正箇所・方法を求める場合、生成された式の充足可能性判定に要する時間が、修正可能箇所の数 (マルチプレクサを挿入した箇所の数) に対して指数的に増

加する。そのため、大規模な設計記述において、バグモデルを用いて修正方法まで求めようとする場合には、一度に扱う DFG の大きさを小さくする必要がある。

提案手法では、生成した DFG を分割することによって、充足可能性判定の実行時間を短縮している。DFG の分割は、各ノードについて出力からの深さをトポロジカルソートによって決定した後、DFG 全体の深さの半分のエッジで切り離すことによって行う。ここで、切断された各エッジの入力側については、そのエッジを出力として、そこにデータフローを及ぼしている全てのノードから成る DFG を作る。この処理によって、切断した部分より出力側に 1 つ、切断されたエッジの入力側にエッジにつき 1 つの DFG が作られる。

それぞれの DFG は、次節で述べるように、修正候補を含む可能性があるかどうかを調べる。この結果、修正候補が DFG 内に含まれる可能性があるとされた場合で、かつ、DFG の深さが与えられた閾値より大きい場合には、さらに分割を行う。

3.5 DFG が修正候補を含む可能性の判定

ここでは、分割された DFG が、その中に反例に対する修正候補を含む可能性があるかどうかを判定する。その手順は次の通りである。以下では、分割された入力側の DFG を G_i、出力側の DFG を G_r、G_i と G_r を分割する際に切断されたエッジを e_i、反例の実行におけるそのエッジに相当する値を v_i、全体の出力変数の集合を OUT とする。

(1) e_i から各出力変数 $out \in OUT$ に至るパス上にあるノードから成る DFG $G_{r_i} \subseteq G_r$ を作る
(2) G_{r_i} における制約式を作る
(3) e_i 以外の G_{r_i} への入力値は、反例の実行で計算された値として、制約に加える。e_i に相当する値は、実際の反例で計算された値と異なることを制約に加える。また、出力は正しい値であることを制約に加える
(4) 制約式の充足可能性を判定する

充足可能である場合、G_i の出力である e_i の値をある異なる値 v へ変更することによって、全体の出力を正しくすることができることを意味する。つまり、G_i の出力が v になるような修正を G_i 内で行えば、全体の出力を正しくすることができる。一方、充足不可能な場合には、分割された DFG G_i をどのように修正しても全体の出力は正しくならないため、G_i 内に修正候補はないと言える。

一方、分割した DFG のうち、出力側の DFG G_r については、修正候補を含むかどうかの判定は行わずに、次節で述べる修正箇所・方法を求める手法を適用する。その際、入力は、分割された各エッジ e_i が反例入力パタンで実行されたときの値を用いる。

3.6 修正個所・方法候補の導出

分割され、深さが閾値以下になった DFG に対して、第 3.3.2 節で述べた修正方法を仮定した手法によって、反例入力パタンを与えた場合に出力される値を正しい値にするために、修正すべき箇所・方法の候補を求める。このとき、第 4 節の実験結果でも示すように、変数値を変更するためのマルチプレクサを挿入する箇所数に対して、充足可能性判定の時間が指数的に増加するため、分割された DFG に対して適用することにより、効率化が期待できる。

また、分割された DFG ごとに得られた修正候補は、全て設計者に示される。これは、与えられた反例入力パタンに対して正しい値を出力するための修正は一般的には複数通りあり得るためである。

3.7 複数の反例や正しい実行例が与えられた場合

反例が複数与えられた場合には、反例ごとに DFG を作成し、修正候補を求める。このとき、設計記述中で同じ変数値に相当する DFG のエッジにマルチプレクサを挿入する際には、全ての DFG において制御信号を共有する。これによって、全ての反例において、同じ部分を修正することができる。制御信号が共有されていない場合、反例ごとに異なる修正の候補が得られるため、ある反例に対して修正を行っても、他の反例に対する修正とはならない可能性がある。

反例の他に、正しい実行例がある場合には、その正しい出力値が得られるように制約を与えることで同様に扱うことができる。この場合、実行におけるある値を修正することは、その値を変更した場合でも、正しい出力値を得ることができることを意味する。反例の実行を表した DFG に挿入されたマルチプレクサの制御信号を共有することによって、「反例入力パタンに対して正しい値を出力し、かつ、正しい実行例は変わらずに正しい」を満たす修正を求めることができる。

4. 予備実験結果

実験として、逆離散コサイン変換 (IDCT) の高位設計記述を用いて、設計規模に対する実行時間の評価および反例実行を表す DFG を分割する手法の適用を行った。充足可能性を判定するソルバーとしては、Boolector[5] を用いた。実験は Xeon 2.6GHz の CPU、8GB のメモリを持つ計算機上で行った。

規模に対して、処理時間がどのように変化するかを評価するため、IDCT 設計に含まれるループを取り出し、その繰り返し回数を 1〜8 回に変化させて、手法を適用した。この実験では、DFG の分割は行わず、与えられた設計規模に対して修正候補を求める時間を充足可

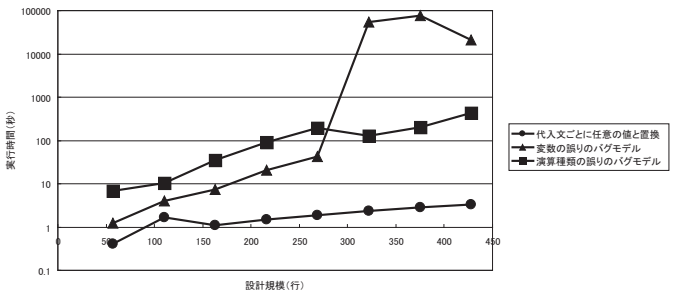

図 5 規模に対する実行時間の評価結果

表 1 得られた修正候補の総数

	MUX 総数	バグ 1	バグ 2	バグ 3	バグ 4	バグ 5
任意の値による置換	38	17	18	18	18	18
バグモデル (変数の誤り)	67	0	1	0	1	0
バグモデル (演算種類の誤り)	68	1	1	1	0	1

能性判定するソルバーの実行時間を評価するために、DFG の分割は行なっていない。図 5 に評価結果を示す。また、繰り返しが 1 回の場合に、得られた修正候補の総数を表 1 に示す。これらの結果より、任意の値で置換する手法に比べ、仮定される修正方法 (バグモデル) を適用できる箇所を探す手法の方が、修正候補数を絞れていることが分かる。一方で、バグモデルを用いる場合には、任意の値で置換をする場合に比べて長い処理時間を要している。

次に、先の実験で用いた最も大きな例題に対して、DFG の分割を適用した手法の適用を行った。反例を元に作られた DFG は 17 個に分割された。それぞれの DFG が修正候補を含むかの判定は、全て 0.1 秒以下で行うことができた。次に、各 DFG に対して、バグモデルを用いて修正候補を求める方法を適用したところ、いずれも 1 秒以内で充足可能性判定を行うことができた。元の設計記述全体に対してバグモデルを用いた手法を適用した場合、数時間を要しており、分割によって効率化できることが示された。

5. 結論と今後の課題

本稿では、高位設計の検証において反例が得られた際に、その反例入力パタンに対して正しい出力値を計算することができる修正の候補を求める手法を提案した。提案手法では、実際に反例を設計記述中で実行して得られた値と、任意の値または仮定する修正が行われた後の値を置換することによって、反例の出力値を仕様で定められた正しい値とすることができるかどうかを調べている。これは、反例の実行を表すデータフローグラフに修正された値を置換するためのマルチプレクサを挿入し、反例入力パタン・正しい出力パタンを制約として充足可能性を解くことによって実現している。設計記述の規模が大きい場合、この充足可能性判定を解く時間が指数的に長くなるため、データフローグラフを分割して効率的に修正候補を求める手法を提案した。

本稿で述べた手法では、反例の実行からデータフローだけを取り出して、その中で修正候補を探索するため、制御フローの誤りによって出力値が正しくないときには、設計者がデバッグを行うために有用な修正方法を得ることができない。これは、制御フローに関わる条件文や分岐が誤っている場合に、データフローを修正して反例の出力が正しくなるようにしても、設計誤りを取り除くことはできない場合が多いためである。今後の課題として、条件分岐や条件文の誤りによる反例に対しても、有効な修正候補を求める手法を研究する。

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, 2005.
2) S. Mirzaeian, F. Zheng, and K.-T. Cheng, “RTL Error Diagnosis Using a Word-Level SAT-Solver,” *Proc. of International Test Conference*, Vol.6, pp.1–8, Oct. 2008.
3) 原田, 西原, 松本, 藤田, “充足可能性判定に基づくシステムレベルデバッグ支援手法におけるバグモデルの導入による効率化,” 情報処理学会研究報告, Vol.2010-SLDM-145, No.10,, pp.1–6, 2010 年 5 月.
4) X. Zhang, H. He, N. Gupta, and R. Gupta, “Experimental Evaluation of Using Dynamic Slices for Fault Location,” *Proc. of Internarional Symposium on Automated Analysis-driven Debugging*, Vol. 6, pp. 33–42, Sep. 2005.
5) R. Brummayer and A. Biere, “Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays,” *Proc. of International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, LNCS, Vol. 5505, pp.174–177, 2009.