HDLコードに対するSMTソルバを用いた自動検証システムの提案

伊沢 亮一[1]　金谷 延幸[1]　藤原 吉唯[1]　竹久 達也[1,2]　丑丸 逸人[1,3]　有末 大[1]　牧田 大佑[1]
三村 聡志[1]　末田 卓巳[1]　井上 大介[1]

概要：HDL（Hardware Description Language）コードをシミュレーションで検証する時，実行されていないステートメントは未検証となってしまうため，コードのカバレッジ（網羅率）が検証の品質を計測する重要な指標の一つとなる．一般的なランダム検証ではモジュール（HDL コード）への入力をランダムに決めるため，十分なカバレッジが得られないことがある．そこで我々はブランチカバレッジを 100%にするための自動検証システムを提案する．提案システムの特徴はレジスタ値を直接設定する仕組みを検証対象のモジュールに入れ，モジュールの検証されていないステートに強制的に遷移させる点にある．これにより遷移先のステートメントを実行することでカバレッジを向上させる．モジュールへの入力とレジスタに設定する値はモジュールに含まれる if 文などの条件をもとに SMT（Satisfiability Modulo Theories）ソルバを用いて求める．実験では一般に公開されている IP コア 3 つを用意し，提案システムによりブランチカバレッジがいずれも 100% になることを確認した．加えて，ランダム検証の結果も比較対象として載せる．

キーワード：ハードウェア記述言語 (HDL), ブランチカバレッジ, Satisfiability Modulo Theories, FPGA

A Testing System using an SMT Solver for HDL Code

Ryoichi Isawa[1]　Nobuyuki Kanaya[1]　Yoshitada Fujiwara[1]　Tatsuta Takehisa[1,2]
Hayato Ushimaru[1,3]　Dai Arisue[1]　Daisuke Makita[1]　Satoshi Mimura[1]　Takumi Sueda[1]
Daisuke Inoue[1]

Abstract: When a module in HLD code is tested using a simulation, HDL-code coverage is one of the most important metrics for testing because part of the code is not checked if it is not executed. This makes it necessary to obtain as high coverage as possible. Because a test-input generator in traditional random testing generates input values to a module at random, it tends to fail to obtain a sufficient coverage. In this paper, we propose a testing system for modules written in HDL code, aiming at a branch coverage of 100% during a simulation. For this aim, our system previously inserts input pins to a target module for directly setting its register values. Our system then automatically changes the current state of a module to an unchecked state by setting the corresponding registers via the inserted pins. At this time, our system uses an SMT solver to obtain such register values for the state transition by satisfying branch-conditions contained in a module. With experiments using three IP (Intellectual Property) cores that are available as OSS (Open Source Software), we confirmed that our system was able to obtain a branch coverage of 100% for every IP core. In addition, we checked traditional random testing against the three IP cores, as a benchmark.

Keywords: Hardware Description Language, Branch coverage, Satisfiability Modulo Theories, FPGA

[1] 国立研究開発法人情報通信研究機構
National Institute of Information and Communications Technology (NICT)
[2] 株式会社ニッシン
Nissin Inc.
[3] 株式会社サイバーディフェンス研究所
Cyber Defense Institute, Inc.

1. はじめに

FPGA（Field-Programmable Gate Array）は医療系画像処理やロボット制御など様々な用途で使用されており [1]，不具合や脆弱性，ハードウェアトロジャン [2], [3], [4] の存在を検証する技術が希求されている．FPGA にプログラミングする回路は Verilog や VHDL（Very High Speed Integrated Circuit HDL）などの HDL（Hardware Description Language）で記述できる．HDL コードの検証方法として仕様外の動作を検知するためのアサーションを埋め込み，HDL コードをシミュレーションで動作させる方法があげられる．検証時，アサーションを埋め込んだ箇所が実行されなければ検知ができないため，HDL コードのカバレッジ (網羅率) は検証の品質を計測する重要な指標の一つとなる．

本稿では Verilog コードで記述されたモジュールを検証対象とし，ブランチカバレッジを 100% にするためのシミュレーションによる検証を行うことを目的とする．また，本稿ではブランチカバレッジを「if 文や case 文などの条件分岐の条件のうち，シミュレーションのある時点において True および False となったことがある条件の割合」と定義する．検証対象のモジュールに含まれる全ての条件が True および False になれば，その条件によって選択される両者のステートメントが実行されたことになり，ブランチカバレッジは 100%となる．

ランダム検証はモジュールに入力する値をランダムに決定する検証方法だが，ランダムに値を決めるためカバレッジが十分に上がらないことがある．そこでカバレッジを向上させることを目的とした RFUZZ[5] や STAR[6] などの従来システムが提案されている．RFUZZ は AFL（American Fuzzy Lop）[7] を利用することで，ある入力の変化に対するカバレッジの増減をフィードバックし，これを元に次の入力を決めていく．例えば，ある条件を True にした入力の一部を反転することで次の入力を決める．STAR は SMT（Satisfiability Modulo Theories）ソルバを利用することで，モジュールに含まれる条件を True （もしくは False）にするような入力を求めて，モジュールに与える．いずれのシステムも 100%のカバレッジが得られないことがある [5], [6].

本稿ではブランチカバレッジを 100%にするための自動検証システムを提案する．提案システムは検証対象となるモジュールの Verilog コードを読み込み，そのモジュールのレジスタを直接設定する仕組みを入れ込むことで，シミュレーション時に条件を強制的に True もしくは False にする．これによりモジュールのステートを未検証のステートに強制的に遷移させ，遷移先のステートメントを実行することでカバレッジを向上させる．条件を True/False にするためのモジュールへの入力およびレジスタの値は SMT ソルバで求める．評価実験では OpenCores[*1] や GitHub[*2]で公開されている OSS（Open Source Software）の IP コアを 3 つ用意し，ブランチカバレッジを 100%にできることを確認した．また，評価実験ではランダム検証の結果を比較対象として示す．

本稿における我々の貢献は次の 2 点である: 1) モジュールのステートを強制的に遷移させる自動検証システムを提案した．これにより未検証の箇所をなくすことが可能となる．2) 実際に使用されている OSS の IP コアを用いて提案システムの評価を行った．なお，本稿の提案システムはブランチカバレッジを 100%にすることのみを目的にしているが，アサーションなどによる仕様外の動作を検知する仕組みは今後取り組む予定である．また，本稿ではモジュールのレジスタを直接書き換えることによる副作用の有無に関する検証を行なっておらず今後取り組む予定である．

2. 基礎知識

2.1 ランダム検証

図 1 に Verilog のサンプルーコードを示す．このコードのモジュール `example` は入力ポート `clock` と `reset`，`in_a` を有し，出力ポート `out_c` を有する．また，レジスタ `reg_state` と `reg_tmpout` を有する．このモジュールの検証方法として，入力ポートに任意の値を入力し，Vivado[8] や Verilator[9] などのシミュレータや FPGA（Field Programmable Gate Array）により動作させることで，レジスタの値や出力ポートの値が仕様外であればエラーとして検出する方法があげられる．図 1 のコードでは，19 行目に不具合があり，19 行目が実行されればシミュレータの機能などによりその不具合を検出できるものと仮定する．

ランダム検証とはモジュールの入力ポートに対し，ランダムな値を与える検証方法である．モジュール `example` の `in_a` は 16 ビット幅の値を受けつけるため，$0 <= \texttt{in\_a} <= 2^{16}$ の範囲でランダムに値を選択する（ただし，モジュールの同期用の `clock` とレジスタのリセット用の `reset` は除く）．手動でモジュールに適した固有の入力パターンを作成する方法と比べ，ランダム検証はモジュールに依存せず入力を機械的に決定できるため自動化がしやすく，またモジュールの設計者が意図しない入力により発生する不具合を発見できる可能性がある．

2.2 カバレッジ

Verilog コードが仕様通りの動作をするかを検証する上で，検証の網羅性を示す指標としてカバレッジ（網羅率）があり，カバレッジの種類を表 1 に示す [10]．ここで表中のステートメントとは Verilog コードの実効行（コメントを

[*1] Open Cores: https://opencores.org/
[*2] https://github.com/

```
 1: module example(clock,reset,in_a,out_c);
 2:   input clock, reset;
 3:   input [15:0] in_a;
 4:   output out_c;
 5:   reg [1:0] reg_state;
 6:   reg reg_tmpout;
 7:
 8:   always@(posedge clock) begin
 9:     if (reset == 1) begin
10:       reg_state <= 0;       /* Branch 1 */
11:       reg_tmpout <= 0;      /* Branch 1 */
12:     end
13:     else if (in_a == 16'h1234)
14:      reg_state <= 1;        /* Branch 2 */
15:     else if (reg_state == 1 && in_a == 16'h5678)
16:       reg_state <= 2;       /* Branch 3 */
17:     else if (reg_state == 2)
18:       reg_tmpout <= 1;      /* Branch 4 */
19:     ... /* <- Suppose some bugs are found here.*/
20:     end
21:     assign out_c = reg_tmpout;
22: endmodule
```

図 1 Verilog サンプルコード

除く行）を意味し，ブランチとは条件分岐においてある条件が満たされたときに実行されるステートメントである．図 1 のサンプルコードであれば Branch 1〜Branch 4 がブランチである．カバレッジが高いほど，実行されたステートメントやステートメントの組み合わせは多く，未検証の箇所が少ないことを意味する．

本稿ではブランチカバレッジを採用し，ブランチカバレッジを「トグルした条件の数 / 条件の総数」と定義する．ここでトグルした条件とはある時点においてこれまでに True および False になったことがある条件を意味する．モジュール `example` の (`reset == 1`) であれば，`reset` に 0 および 1 が入力されたことがあれば，(`reset == 1`) はトグルした条件となる．また，モジュール `example` に含まれる条件の総数は 4 であり，ある時点までに (`reset == 1`) と (`in_a == 16'h1234`) のみトグルしていた場合は，その時点でのブランチカバレッジは 0.5 (50%) となる．

本稿の提案システムは検証対象の Verilog コードのブランチカバレッジを 100%にするために動作する．コンディショナルカバレッジやパスカバレッジへの対応は今後の課題とする．なお，ブランチカバレッジはステートメントカバレッジを包含しており，ブランチカバレッジが 100%であれば，ステートメントカバレッジも 100%となる．

2.3 ステートの遷移および問題定義

ステートの遷移: 図 1 のモジュール `example` はモジュールのステートを保持するためのレジスタ `reg_state` を有し，always 文によりステート遷移が記述されている．その always 文には 4 つの条件分岐が含まれ，各分岐はあるステートに対応し，入力に応じて次にどのステートに遷移するかを表している．例えば，15 行目の else if 文は「`reg_state` が 1 のステートにおいて，入力 `in_a` が 5678（16 進数）であれば，`reg_state` が 2 のステートに遷移する」ことを表している．このように HDL コードで記述されたモジュールは「レジスタによってステートを実現し，あるステートにおける入力によって次のステートを決定するステートマシンをもつ順序回路」[11], [12] として記述される．

表 1 カバレッジの種類 [10]

カバレッジの種類	内容
ステートメント カバレッジ	各ステートメントの実行の有無
ブランチ カバレッジ	条件分岐の各ブランチの実行の有無
コンディショナル カバレッジ	条件分岐の条件が複数ある場合，それぞれの条件の True/False
パスカバレッジ	複数の条件分岐がある場合，選択されたブランチの組み合わせ

モジュール `example` のブランチカバレッジを 100%にするためには，always 文に記述された 4 つの条件を True にする必要がある．ここで，図 1 のコードにおいて，16 ビット幅の入力 `in_a` を含む条件 `if (in_a == 16'h1234)` に対し，ランダムに `in_a` の値を入力すると，この条件が True になる確率は「$\prod_{i=1}^{16} 1/2$」($= 1/2^{16}$) であり，必要となる試行回数の期待値は 2^{16} である．

モジュール `example` はステートを含むため，100%のブランチカバレッジを得るためには，さらに多くの試行回数が要求される．例えば，図 1 のコードにおいて，`out_c` の値が 1 になるにはレジスタ `reg_state` の値が 2 にならなければならない．そのためには (`in_a == 16'h1234`) を True にすることで `reg_state` を 1 にしたあと，(`in_a == 16'h5678`) を True にしなければならない．このように，ステートが含まれる条件を満たすためには，そのステートに遷移させた上で，適切な入力を与えなければならない．

問題定義: 一般的に Nbit 入力の M 個のステートを含むモジュールでは，あるステートに遷移する適切な入力を得る確率 X は「$\prod_{i=1}^{n}(1/n)$」($= 1/2^n$) となり，初期ステートから必要なステートに遷移する確率は「$\prod_{j=1}^{M} X$」($= 1/2^{mn}$) となる．このとき，ブランチカバレッジを 100%にするために必要な試行回数の期待値は 2^{mn} である．これをランダム検証により実用的な時間で達成することは難しい．

入力のビット幅が広く，かつ複数のステートを有するモジュールに対しても，100%のブランチカバレッジを得ることを提案システムの目的にしている．

2.4 SMT ソルバ

SMT（Satisfiability Modulo Theories）とは与えられた条件を True にする変数の値が存在するか（充足可能），存在しないか（充足不能）を判定する問題のことである [13]. 例えば，ある変数 x と y を含む条件 $(x+y>1)$ を True にする x と y の値が存在するかを判定する．この問題は充足可能であり，$x=1$ と $y=1$ が解の一つである．図 1 のコードであれば，(`in_a == 16'h1234`) を満たすような `in_a` の値の存在や (`reg_state == 1 && in_a == 16'h5678`) を満たすような `reg_state` と `in_a` の値の存在を判定する問題である．

SMT ソルバとは与えられた条件が充足可能か充足不能かを自動で判定するツールであり，Z3[13] や Boolector[14], STP[15] などがある．充足可能である場合はその解も出力する．SMT ソルバに条件 (`reg_state == 1 && in_a == 16'h5678`) を与えると，条件を True にする `reg_state` と `in_a` の値として，それぞれ 1 と 5678（16 進）を得ることができる．本稿の提案システムは検証対象のモジュールに入力ポートとレジスタの値を決める際に SMT ソルバを利用する．

3. SMT ソルバを用いた自動検証システム

3.1 基本アイデア

提案システムはランダム検証などの従来手法では 100% のブランチカバレッジの達成が難しいモジュールに対して，より短いシミュレーションサイクルで 100%のブランチカバレッジを達成することを目的としている．2.3 節の問題定義で述べたように，検証対象のモジュールへの入力のビット幅が広く，そのモジュールが複数のステートを有する場合，ランダム検証では 100%のブランチカバレッジを得ることは難しい．

提案システムでは SMT ソルバを用いることで，ある条件を True もしくは False にするための入力とステートを得てから，レジスタに値を直接設定することで未検証のステートに強制的に遷移させる．レジスタ値の設定のために，提案システムはモジュールの Verilog コードを読み込み，新たな入力ポートをモジュールに追加することで実現する．

提案システムはレジスタ値を直接設定することで強制的にステートを遷移させるため，ランダム検証などでステートを順に遷移させながら検証する方法とは異なる．しかしながら， 専門家がモジュールに応じたテストベンチを記述する場合であっても，モジュールをあるステートに手動で遷移させて上で，そのステートにおける各ブランチを検証する場面がある．このようなステートの遷移を提案システムでは自動で行う．

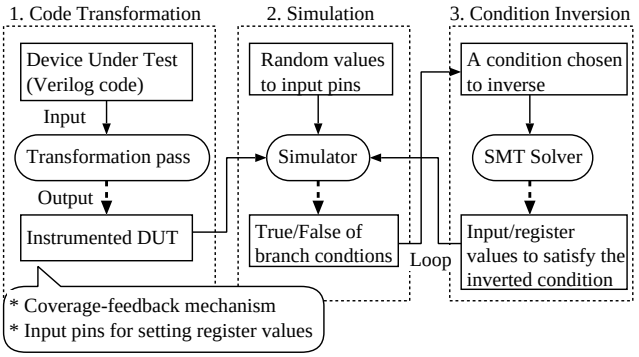

図 2 提案システムの構成と処理の流れ

3.2 提案システムの構成

図 2 に提案システムの構成と処理の流れを示している．提案システムは 3 つの機構で構成され, 一つ目の機構はコード変換処理部（Code Transformation）である．検証対象のモジュール（DUT: Device Under Test）の Verilog コードを読み込み，シミュレーションの各サイクルにおいてカバレッジを計測する仕組み（Coverage feedback mechanism）を埋め込む．提案システムでは，各条件が True であったか False であったかを計測することで，どのブランチが選択されたかを得る．また，モジュール内のレジスタ値を設定するための入力ポートを追加する

二つ目の機構はシミュレーション部（Simulation）である．最初の 1 サイクルはモジュールの入力ポート全てにランダムな値を入れて，どのブランチが選ばれたかを取得する．次のサイクルからは後述する SMT ソルバから得た値を入力に使用する．

三つ目の機構は条件反転部（Conditional Inversion）である．これまで選択されていないブランチが選択されるように，SMT ソルバでモジュールへの入力やレジスタの値を取得する．例えば，ある条件がこれまでのサイクルにおいて False だった場合はその条件を True にするための入力やレジスタの値を取得する．取得した値を次のサイクルに使用して新規ブランチを選択することでブランチカバレッジを向上させる．

3.3 各機構の機能

コード変換処理部: 検証対象の Verilog コードを読み込み，各条件のトグルの有無を検知するための出力ポートをモジュールに追加する．ある条件が True になると 1，False になると 0 を出力するポートである．条件と対応する出力ポートは組にして保存しておく．これは条件に対応するブランチが選択されたか否かを取得し，選択されていないブランチを SMT ソルバにて処理させるためである．

条件にレジスタが含まれる場合，条件をトグルさせるために，モジュールのレジスタの値を書き換える．このため，レジスタ値を設定するときに 1 を与える入力ポート

signal_set_regs および設定するレジスタ値を与える入力ポート reg_value をモジュールに追加する．レジスタ値設定用信号が 1 のとき，対象のレジスタに値が設定される．reg_value はモジュールに含まれるレジスタの数だけ用意する．

シミュレーション部: 各入力ポート $\mathbf{I} = (a_0, a_1, \cdots, a_n)$ に対してランダムな値を入力する．ここで n はモジュールの入力ポートの数とし，各ポートに対して入力された値を $\mathbf{R} = (r_0, r_1, \cdots, r_n)$ とする．signal_set_regs は 0 とする．このあと t サイクル $(t = 0, 1, \cdots)$ のシミュレーションを実施し，各条件の真偽値 $\mathbf{C} = (c_0, c_1, \cdots, c_k)$ の True/False を取得する．ここで $c_i \in \{True, False\}$ $(i = 0, 1, \cdots)$ であり，k はモジュール内に存在する条件の数である．$\mathbf{C}$ をもとに，t サイクル目までのシミュレーションにおいて各条件がトグルしたかどうかを $\mathbf{S} = (s_0, s_1, \cdots, s_k)$ に記録する．具体的には $\mathbf{S}$ の各要素の初期値は 0 とし，各サイクルのシミュレーション後に，s_i が 0 のとき c_i が False であれば s_i に 1 を代入し，s_i が 0 のとき c_i が True のとき s_i に 2 を代入する．また，s_i が 2 のとき c_i が False，もしくは s_i が 1 のとき c_i が True であれば s_i に 3 を代入する．それ以外は s_i の値を変化させない．よって，s_i が 3 であれば c_i はトグル済みであることを示し，s_i が 1 であればいずれのサイクルおいても False だったことを示す．また，s_i が 2 であればいずれのサイクルにおいても True だったことを示す．各サイクルにおいて，ブランチカバレッジを「値が 3 の $\mathbf{S}$ の要素数 / k」をとして求める．ブランチカバレッジが 1.0（100%）になるとシミュレーションを終了する．

条件反転部: $\mathbf{C}$ のうち，トグルしていない条件（$\mathbf{S}$ のうち値が 3 になっていない条件）を選択する．その条件に対する s_i が 1 (常に False) であれば，条件を True にするための入力およびレジスタの値を SMT ソルバを用いて求める．例えば，(a_0 == 1 && a_1 == 1) という条件を True にするためには a_0 および a_1 を 1 にすればよい．また，s_i が 2（常に True）であれば，条件を False にするための入力およびレジスタの値を SMT ソルバを用いて求める．

入力 $\mathbf{I}$ の要素から $\mathbf{C}$ の条件を True/False にするもののみ SMT ソルバで求めた値を代入し，その他の入力は前の値を用いた新たな入力値を生成する．また，SMT ソルバの結果によりレジスタの変更が必要と判断された場合は新たなレジスタ値を求める．この条件反転部の出力を用い，シミューレーション部は次のサイクルである $t + 1$ のシミュレーションを実施する．このとき，レジスタの値を変更する場合は signal_set_regs を 1 に入力し，レジスタ値を対応する reg_value に入力する．

3.4 実装

コード変換処理部: 入力された Verilog コードを Python のパッケージ pyverilog[16] を用いて抽象構文木を作成する．この抽象木のルートからノードを辿りながら条件分岐命令を記憶しておき，各分岐条件に対応する出力ポートをノードとしてモジュールの入出力宣言に挿入する．例えば，条件 (in_a == 16'h1234) であれば，モジュールに出力ポート output cover_0; を追加して，その値を assign cover_0 = (in_a == 16'h1234) ? 1 : 0; [*3]として出力する．これにより (in_a == 16'h1234) が True のとき cover_0 が 1 になり，False のとき 0 となる．条件 $\mathbf{C} = (c_0, c_1, \cdots, c_k)$ のそれぞれに対して出力ポート (cover_0, cover_1,⋯,cover_k) を挿入する．

```
always@(posedge clock) begin
  if (signal_set_regs == 1) begin
    /* レジスタへの値代入処理 */
    reg_state <= reg_value_for_reg_state;
  end
  else begin
    /* 本来の処理 */
  end
end
```

図 3 レジスタ値設定用信号の埋め込み

モジュールのレジスタに任意の値を設定するために，抽象構文木を走査する際に，条件として使用されているレジスタを記憶しておき，レジスタ設定用入力ポート input signal_set_regs と記憶した各レジスタの値を変更するための入力ポート（例: あるレジスタ reg [1:0] reg_state に対する input [1:0] reg_value_for_reg_state）をモジュールのポート宣言に追加する．signal_set_regs が 1 のときレジスタの値を変更するため, 図 3 のように各 always ブロックの先頭に対応するノードに if (signal_set_regs == 1) else 相当のノードを挿入する．最後に，抽象構文木を pyverilog の ASTCodeGenerator クラスに入力して，Verilog コードを生成する．

シミュレーション部: コード変換処理部で変換した Verilog コードを Verilator[9] を用いて C++言語に変換する．モジュールを top という名称で生成したとすると，top->clock=1 や top->in_a=0 のようにモジュールの入力ポートに値を入力でき，top->eval() を呼び出すことで入力した値によりシミュレーションが行われる．

シミュレーション実施時には，モジュールの入力 $\mathbf{I}$ の各要素には random 関数で生成した値を代入する．ただし，clock（クロック信号）は top->eval() を呼び出す ごとに 0 と 1 を交互に代入する．ここで clock が 0 のときのシミュレーションの 1 ステップと，その次の clock が 1 のときの 1 ステップをあわせて 1 サイクルとする．また，reset（入力の初期化信号） はシミュレーションの 1 サイクル目の

[*3] x = (a == b) ? 1 : 0; は 3 項演算子を意味する．

み 1 を代入してモジュールのレジストリの初期化を行い，その後は reset の値は 0 で固定する．1 サイクルのシミュレーションを実施する度に，条件 **C** に対応する出力ポート (cover_0, cover_1,⋯, cover_k) のそれぞれの値を確認して，各条件が常に False か，常に True か，トグルしたかを **S** に記録する．

条件反転部: SMT ソルバは Python のパッケージ z3[17] を用いた．条件のステート $\mathbf{S} = (s_0, s_1, \cdots, s_k)$ を先頭から順に確認し，値が 3 でない要素をひとつ取り出す．ここではこの要素を仮に s_j する．s_j の値が 1（常に False）のとき，c_j が True になるように SMT ソルバ z3.Solver に代入して解決する．例えば，c_j が `(a == 0)` かつ a のサイズが 1 ビットであれば，python のコード `a = z3.BitVect("a", 1)` で a を 1 ビットサイズのビットベクトルとして定義し，`solver = z3.Solver()` で SMT ソルバのインスタンスを生成した後，`solver.add(a == 0)` で c_j を False にするよう条件を追加する．この後，`solver.check()` で解が存在するかを確認してから，`solver.model()[a]` で a の値 0 が取得できる．また，s_j の値が 2 のとき，c_j が False になるための a の値を求めるには `solver.add(z3.Not(a == 0))` として条件を否定してから追加する．a の値を求めた後，a がモジュールの入力であれば，`top->a` に求めた値を代入して，モジュールの他の入力はそのままにする．その後，シミュレーションの処理に戻る．

4. 評価実験

4.1 実験目的とデータセット

本評価実験では提案システムがブランチカバレッジが 100%となるのに必要となるサイクル数を計測する．比較対象としてランダム検証も実験に用いる．また，同一の対象に対するランダム検証の実験も行った．ブランチカバレッジは 3.3 節のシミュレーション部に記載した通りに求める．例えば，あるモジュール内に条件が 10 個あり，シミューションのある時点において 2 つの条件がトグルしていれば，2/10 でカバレッジは 0.2（20%）となる．

実験用データセットとして，より実用的な IP コアを対象とするため，OSS のモジュール i2c_master_byte_ctrl[*4]，cordic_demod[*5]，elelock[*6] の 3 つを用いる．それぞれのモジュールの入力ビット幅と条件数，Verilog コード行数は表 2 に示す．ここでコード行数は空行とコメント行，begin のみの行，end のみの行を除いてカウントした．また，括弧の中にコメント行など除かない全行数を記載している．

表 2 評価用データセット

モジュール名	入力ビット幅	条件数	コード行数 (*)
i2c_master_byte_ctrl	35	17	170 (344)
cordic_demod	68	16	131 (205)
elelock	24	27	165 (208)

*: コード行数の括弧の値はコメント行などを除かない全行数

4.2 実験結果

提案システムおよびランダム検証それぞれで i2c_master_byte_ctrl モジュールを検証したときのブランチカバレッジの変化を図 4 に示す．横軸はシミュレージョン開始時から経過したサイクル数 $i\,(= 0, 1, 2, \cdots)$ を示し，縦軸はそのサイクルにおけるブランチカバレッジを示している．図中のランダム検証を RAND，提案システムを RAND_SMT として表記している．ランダム検証は 27 サイクル目にブランチカバレッジが 83%，144 サイクル目に 94%となっているがその後ブランチカバレッジは上がらず，524 サイクル目で 100%となった．524 サイクル目まで 100%にならなかった理由だが，i2c_master_byte_ctrl は c_state というレジスタを有しており，5 つのステート（ST_IDLE, ST_START, ST_READ, ST_WRITE, ST_ACK, ST_STOP）に遷移するが，144 サイクル目以降，ST_STOP のステートに遷移しなかったためである．一方，提案システム（RAND_SMT）は 3 サイクル目にブランチカバレッジが 27%，5 サイクル目で 72%，27 サイクル目に 100% となった．提案システムのほうも選択されていない最後のブランチは ST_STOP のブランチだったが，条件反転部により必要なレジスタ値を求め，ステートを強制的に遷移させている．

cordic_demod モジュールのカバレッジの変化を図 5 に示す．ランダム検証（RAND）は 8 サイクル目にブランチカバレッジが 87%となった後，次にブランチカバレッジが上がったのは 500 サイクル目で 93%，532 サイクル目に 100%となった．500 サイクル目までブランチカバレッジが上がらなかった理由だが，cordic_demod は state というレジスタを有しており，5 つのステート（STATE_IDLE, STATE_SHIFT_LOAD, STATE_SHIFT, STATE_ADD, STATE_DONE）に遷移するが，ステートが STATE_ADD に留まり 30 サイクル動作しないと True にならない条件（`step_counter == 'd30`）が 500 サイクル目まで True にならなかったためである．8 サイクル目以降，ステートが STATE_ADD 以外に遷移していた．一方，提案システム（RAND_SMT）は 10 サイクル目でカバレッジ 100%になっている．最後の条件は `state == STATE_DONE` であり，強制的に未検証の STATE_DONE のステートに遷移させている．

elelock モジュールのカバレッジの変化を図 6 に示す．ランダム検証（RAND）は 1500 サイクル目までの結果を図には載せておりブランチカバレッジは 74%だった．その後，500000 サイクル目でブランチカバレッジは 88%だった．elelock は 10 ビット幅の入力 `decimal` があるが，decimal

*4 https://opencores.org/projects/i2c
*5 https://github.com/analogdevicesinc/hdl
*6 https://github.com/zyu-c/verilog_elelock

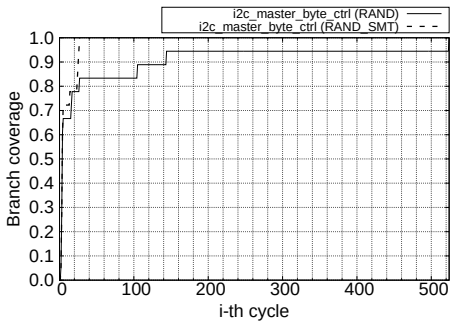

図 4 ブランチカバレッジ (i2c_master_byte_ctrl)

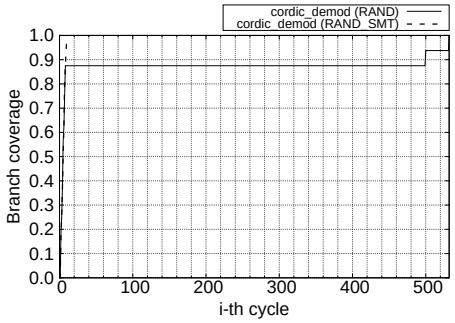

図 5 ブランチカバレッジ (cordic_demod)

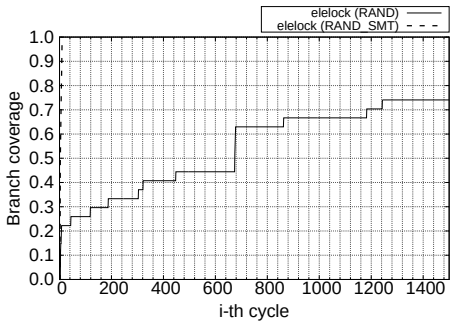

図 6 ブランチカバレッジ (elelock)

が取り得る値は 10 種類のみである．ランダムに decimal に値を入力すると 10 種類の値に当たる確率は $10/2^{10}$ となる．ランダム検証では `decimal` に対して遷移を生じさせない値が与えられるサイクルが多かった．また，elelock はレジストリ state があり，6 つのステートを有するため，ステートを遷移させることが難しかった．提案システムでは elelock に対しても 10 サイクル目でブランチカバレッジ 100%になっている．

4.3 考察

提案システムではレジストリを書き換えることで，ステートを遷移させることができ，ブランチカバレッジを効率的に上げることができることが実験から分かった．特に，遷移を生じさせる確率が非常に低い elelock のようなランダム検証ではブランチカバレッジを十分に上げることができない，つまり 2.3 の問題定義で述べたような遷移が複雑なステートがあるモジュールに対しては提案システムが有効であるといえる．しかしながら，レジストリの書き換えによってステートを遷移させることは仕様外の挙動を引き起こす懸念もある．そのため，ランダム検証と提案システムを併用するとよりより効率的でかつ現実的な検証が可能となる．初めはランダム検証によりモジュールを検証して，検証できていないブランチを記録しておく．次に，提案システムよりそのブランチを検証できるようなステートに遷移させてから検証する方法などが考えらえる．また，本稿ではブランチカバレッジを上げることのみに着目したが，検証したいステートに遷移・固定させ，様々な入力を試してモジュールの出力が仕様の通りか検証する方法も考えらえる．このように提案システムの SMT ソルバとレジストリの書き換えの仕組みはランダム検証など従来手法と協調することでより効果的な検証方式を実現することが期待できる．

5. 関連研究

5.1 カバレッジ向上を目的とした検証

STAR[6] は効率的にカバレッジを向上させることを目的とし，ランダム検証に SAT（boolean SATisfiability problem）ソルバを組み合わせて RTL コードの検証を行う．あるサイクルのシミュレーションにおける，各条件の一部を反転させて，次のサイクルでは別のブランチを選択する．例えば，`(a > b)` と `(d < e)` ともに True のとき，次のサイクルでは `(a > b)` と `(d >= e)` が True になるような入力を SAT ソルバで得る．ただし，STAR ではレジスタを書き換えないため，レジスタを含む条件を SAT ソルバで扱わない．そのため，入力が含まれる条件を True/False に変えるための入力を与えながら，レジスタを含む条件が True になるまでシミュレーションを繰り返さなければならない．STAR に対する実験 [6] では 4 つのモジュールに対してブランチカバレッジを求めており，100%のブランチカバレッジが得られなかったモジュールが存在している．これに対して本稿の提案システムではレジスタを直接書き換える仕組みを設けることでブランチカバレッジを 100%を達成することができる．

RFUZZ[5] は AFL（American Fuzzy Lop）[7] を用いて RTL コードに対してカバレッジ解析を行う．AFL のアルゴリズムに従い，ある入力によって新規ブランチが選択されると，その入力をもとに次の入力を生成することで，カバレッジを効率的に得ることを目的にしている．また，RFUZZ は RTL コードを FPGA 上で動作させることで，ソフトウェアシミュレーションと比べて，高速なカバレッジ解析を実現している．RFUZZ はファジングを採用しており，カバレッジ解析に関して提案システムとはアプローチが異なる．

Ghosh らの手法 [18] はモジュール（RTL コード）中のあるステートメントに指定し，そのステートメントがあるブ

ランチを選択（実行）するためのモジュールへの入力を自動生成することを目的とする．Ghosh らの手法では VHDL を ADD（Assignment Decision Diagram）に変換して，ある選択したステートメントに関連するステートメントや条件を抽出する．Ghosh らの手法ではステートメントを 1 つ指定し，そこから遡って，そのステートメントを実行している．提案システムではブランチカバレッジを 100%にすることを目的にしている点で異なる．

5.2 エラー検出

5.1 節ではカバレッジ解析に関する研究について述べたが，本節では回路に含まれるエラーを検出することを目的とした研究について述べる．これは提案システムにおいてもエラー検出をする仕組みを導入することが今後の課題としてあげられるためである．

Sheeran らの手法 [19] はモジュールの各出力ポートの値の組をステートとして表し，各ステートが保持する値およびステート間の遷移は仕様通りかを検証することを目的としている．例えば，モジュールの 1 ビットの出力ポート (a，b，c) の値と遷移を $(1,0,0) \rightarrow (0,0,1)$ のように表し，あるステート間での遷移が本来起こり得ない場合，または本来取り得ない値（例: a と b が 0 のとき，c は 0 にならない）のとき，エラーとして検出する．

Ciesielski らの手法 [20] は多項式の算術演算回路の入出力が仕様通りかどうかを検証することを目的としている．本手法では回路（ゲートレベル）の出力から回路の入力まで遡りながら演算式を自動生成することを特徴とし，事前に与えられる出力の仕様と演算式を比較してエラーがないかを検出する．

6. まとめ

本稿では効率的にブランチカバレッジを 100%にすることを目的にした自動検証システムを提案した．提案システムの特徴は検査対象のモジュールのステートを遷移させるためのレジスタ値を SMT ソルバで求め，レジスタを直接書き換えることで強制的に未検証のステートを遷移させる点にある．実験では OSS の IP コア 3 つに対して 100%のブランチカバレッチを 10〜30 サイクル程度と非常に短いサイクルで達成した．今後，他のカバレッジを指標とした検証の実装やエラー検証の仕組みを実装する予定である．

参考文献

[1] Melanie Berg, Kenneth LaBel, and Jonathan Pellish. New developments in fpga: Seus and fail-safe strategies from the nasa goddard perspective, 2016. https://ntrs.nasa.gov/api/citations/20160014713/downloads/20160014713.pdf.

[2] Oliver Söll, Thomas Korak, Michael Muehlberghuber, and Michael Hutter. Em-based detection of hardware trojans on fpgas. In *2014 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, pp. 84–87, 2014.

[3] Apostolos P. Fournaris, Lampros Pyrgas, and Paris Kitsos. An fpga hardware trojan detection approach based on multiple parameter analysis. In *2018 21st Euromicro Conference on Digital System Design (DSD)*, pp. 516–522, 2018.

[4] Sanchita Mal-Sarkar, Robert Karam, Seetharam Narasimhan, Anandaroop Ghosh, Aswin Krishna, and Swarup Bhunia. Design and validation for fpga trust under hardware trojan attacks. *IEEE Transactions on Multi-Scale Computing Systems*, Vol. 2, No. 3, pp. 186–198, 2016.

[5] Kevin Laeufer, Jack Koenig, Donggyu Kim, Jonathan Bachrach, and Koushik Sen. Rfuzz: Coverage-directed fuzz testing of rtl on fpgas. In *Proceedings of the International Conference on Computer-Aided Design*, ICCAD '18, 2018.

[6] Lingyi Liu and Shabha Vasudevan. Star: Generating input vectors for design validation by static analysis of rtl. In *2009 IEEE International High Level Design Validation and Test Workshop*, pp. 32–37, 2009.

[7] Michał Zalewski. american fuzzy lop. https://lcamtuf.coredump.cx/afl/.

[8] Inc. Xilinx. Vivado. https://www.xilinx.com/products/design-tools/vivado.html.

[9] W. Snyder. Veripool. https://www.veripool.org/verilator/.

[10] 兼平靖夫. Rtl コード・カバレッジ解析入門. In *Design Wave Magazine*, pp. 104–111. CQ 出版, 2002.

[11] Edward F. Moore. Gedanken-experiments on sequential machines. In *Automata Studies*, pp. 129–153. 1956.

[12] George H. Mealy. A method for synthesizing sequential circuits. *The Bell System Technical Journal*, Vol. 34, No. 5, pp. 1045–1079, 1955.

[13] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340, 2008.

[14] Robert Brummayer and Armin Biere. Boolector: An efficient smt solver for bit-vectors and arrays. In *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 174–177, 2009.

[15] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Proceedings of the 19th International Conference on Computer Aided Verification*, CAV'07, p. 519–531, 2007.

[16] Shinya Takamaeda-Yamazaki. Pyverilog: A python-based hardware design processing toolkit for verilog hdl. In *Applied Reconfigurable Computing*, Vol. 9040, pp. 451–460, Apr 2015.

[17] Leonardo de Moura. Z3 api in python. https://ericpony.github.io/z3py-tutorial/guide-examples.htm.

[18] Indradeep Ghosh and Masahiro Fujita. Automatic test pattern generation for functional rtl circuits using assignment decision diagrams. In *Proceedings of the 37th Annual Design Automation Conference*, DAC '00, p. 43–48, 2000.

[19] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a sat-solver. In *International conference on formal methods in computer-aided design*, pp. 127–144, 2000.

[20] Maciej Ciesielski, Cunxi Yu, Walter Brown, Duo Liu, and André Rossi. Verification of gate-level arithmetic circuits by function extraction. In *Proceedings of the 52nd Annual Design Automation Conference*, DAC '15, 2015.