Vol.2012-ARC-202 No.21
Vol.2012-HPC-137 No.21
2012/12/14

形式的手法を用いたプロセッサ設計不具合の自動診断と修正

谷田 英生[1,a)]　Amir Masoud Gharehbaghi[2,3,b)]　藤田 昌宏[2,3,c)]

概要：本稿ではプロセッサ設計不具合を自動的に診断して修正する手法を提案する．手法は，ある命令セットをもつプロセッサに関して，逐次的に命令を実行して正常動作するモデルと，スーパースカラ・アウトオーブオーダ実行などの高性能化を実現する機構を含むが正常動作しないモデルが与えられた際に，後者のモデルの不具合を診断・修正する．本手法は，従来 RTL 設計やゲートレベルのネットリストのデバッグに用いられてきた，設計に対してマルチプレクサを挿入してその入力信号値を設計が正しく動作するように決定する手法に基づくものとなっている．決定された信号値は不具合を設計する修正の候補を表現し，各修正候補は候補を用いた修正を経た正常動作しないモデルが正常動作するモデルと等価な動作をすることが形式的に検証されると最終的な解として出力される．提案手法を，スーパースカラ・アウトオブオーダ実行や，タイミングエラー回復機構を備えた複雑なプロセッサを例題として評価を行った．評価のためプロセッサに挿入したバグは提案手法により診断・修正可能であることが確認された．

1. はじめに

プロセッサ設計が正しい動作を確認する検証には困難が伴い，設計フローの中でも工数を要している．また検証によって発見された不具合の診断・修正 (以下，DEDC: Design Error Diagnosis and Correction) はさらに自動化が困難で長い時間を要する．

不具合の診断は，不具合を含む設計を解析して，適切な変更を行うと不具合を修正できる可能性がある箇所を検出する操作である．不具合の修正は，診断により提示された箇所から，仕様に合致する動作を実現できるものを選択して仕様に合致しない動作を修正する操作である．不具合の修正後には検証を再度行なって，その不具合が無くなっており新たな不具合が生じていないことを検証する．

昨今では，プロセッサにはタイミングエラーからの回復などの複雑な機構を備えるものも多く，DEDC がますます困難になる傾向がある．そこで本稿では，形式的検証技術に基づくプロセッサの DEDC 手法を取り扱う．本稿で提案する手法では，形式的検証を行うツールとして UCLID [1], [2] を使用する．但し，提案手法は UCLID 以外の記号的実行 (Symbolic Execution) とプロパティの検査を行う任意のフレームワークを用いて実装することも可能である．提案する DEDC のフローを図 1 に示す．入力としては正常動作することが確認されている (Golden)UCLID モデルと，不具合を含んで正常動作しない (Erroneous)UCLID モデルが与えられる．これらのモデルは，必要に応じて v2ucl[3] などのツールを用いて半自動で生成することも可能である．

不具合の診断の際には，既存の DEDC 手法 ([4] など) と同様，不具合を含むモデルの複数の変数に対応する箇所にマルチプレクサを挿入する．そして，正常動作するモデル・不具合を含むモデル・マルチプレクサを挿入したモデルを組み合わせたモデルに対して，修正候補の抽出を目的としたプロパティを作成・検査して反例を基に自動的に修正候補の抽出を行う．また，それと同時に反例を基としてモデル間の挙動の相違を生むような入力シーケンスも自動生成する [5]．これは RTL 設計やゲートレベルのネットリストの DEDC を対象とした既存研究に対する提案手法の新規点の一つであるということができる．既存研究はシミュレーションを行った結果を基に修正候補を抽出するため，不具合を再現するような入力シーケンスとその入力シーケンスを与えた際に期待される出力を DEDC の際に必要とする．一方，本稿で提案する手法では，各モデル間の挙動の相違を生むような入力シーケンスを形式的手法を用いて自動生成する．形式的手法を用いた自動生成は，入力シーケンスとして可能なものを全探索するのと等価な操作になる．

不具合の修正の際には，診断の際に得られた修正候補から正しい動作が得られる可能性が高く設計に対する修正の

[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
[a)] tanida@cad.t.u-tokyo.ac.jp
[b)] amir@cad.t.u-tokyo.ac.jp
[c)] fujita@ee.t.u-tokyo.ac.jp

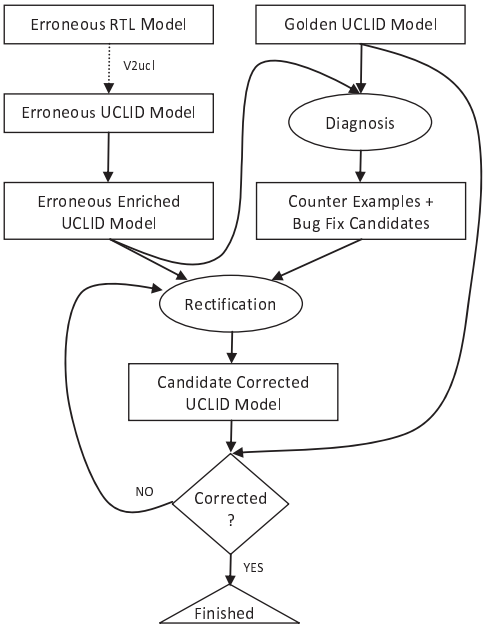

図 1　提案する DEDC のフロー

規模が小さい候補を選択して適用する．修正候補の選択の際には，文献 [6], [7] において提案されているようなバグモデルに基づく選択を行う．

最後に，不具合修正後のモデルでは不具合が修正されており，新たな不具合が生じていないことを形式的手法を用いて検証する．これは，正常動作することが確認されているモデルと修正後のモデルの等価性検証によって行う．既存手法は限られた入力シーケンスを用いたシミュレーションによって検証を行うため，充分な検証に必要なカバレッジを達成するためには，多くのテストパターンを用意する必要がある [8] が，等価性検証を用いる本手法ではそのような必要はない．

以降の本稿の構成は以下のようになる．第 2 節において関連研究を取り上げる．第 3 節において提案手法の背景となる技術の説明を行う．そして，第 4 節と第 5 節においてそれらの技術を用いた不具合の診断・修正・検証手法の提案をする．第 6 節において提案手法の評価を行ない，第 7 節にまとめと今後の課題を記す．

2. 関連研究

開発現場における設計のデバッグは困難が伴う．そのため，自動的なデバッグを目的とした手法がいくつか提案され，ある程度の自動的なデバッグを実現している．しかし，多くのデバッグ手法はトランジスタレベル [9] もしくはゲートレベル [10], [11] で表現された回路を対象として，組み合わせ回路・順序回路の不具合の診断のみを行う．また，その際には回路の全てがスキャンチェインによって可観測であることを前提としている．既存研究 [8], [12] においては自動的な不具合修正も提案されているが，設計者による手動での修正も依然として必要となっている．

RTL 記述を対象として不具合の診断・修正を行う既存研究も存在する．既存研究 [13] においてはテストパターンを用いた不具合箇所の特定を行なっており，不具合の修正を行う研究も存在する [14]．多くの手法は不具合の修正の際には [6] などで提案されたバグモデルに基づく修正を行う．また，形式的手法に基づく不具合診断手法も存在する [4]．これらの手法は，正常動作することが確認されている (Golden) モデルを基に，実装の不具合を診断する．形式的手法に基づく手法では，充分なカバレッジを実現するテストパターンを用意する必要はないが，大規模な設計を扱おうとするとモデルに含まれる状態数の爆発が起きてしまい，検証対象とすることができない．形式的手法に基づく不具合診断手法においては，不具合を含む設計に対してマルチプレクサを挿入して，修正候補の抽出を目的とした特殊なプロパティの検査を行うことが一般的である．

本稿で提案する手法は関連研究に比較して，以下のような点での新規性が存在する．まず，トランジスタ・ゲートレベルもしくは RTL の設計記述を対象とした既存手法に比較して，提案手法はより抽象的に記述されたプロセッサのマイクロアーキテクチャを検証対象とする．また，本稿で提案する手法は RTL 記述を対象とした形式的手法と同様，設計にマルチプレクサを挿入して不具合の診断・修正を目的とした特殊なプロパティを検査するものであるが，診断・修正を容易にする新たなプロパティの導入を行う．さらに，修正の際には既存研究で提案されたバグモデルを用いるが，抽象的記述を用いることによって効率的に処理を行う．そして，修正に用いる論理関数は最初から再合成するのではなく，不具合を再現する入力が与えられたときの挙動に着目して修正を行う．また，提案手法は修正に対する検証の際に形式的手法を用いるため，テストパターンを用意する必要がない．

3. 背景となる技術

3.1 UCLID による設計の検証

UCLID[2] は CLUF(Counter arithmetic with Lambda expressions and Uninterpreted Functions) 論理によって記述されたシステムを検証する [1]．UCLID は独自の入力言語，記号シミュレータと決定手続により構成される．

UCLID は, 入力として uninterpreted functions and symbols や，arithmetic of counters，bit-vector arithmetic，restricted lambda expressions を用いて記述されたシステムを扱うことが可能である．UCLID の検証エンジンは，bounded model checking, correspondence checking, inductive invariant checking, counter arithmetic を用いるが量化子を用いない一階論理式による property checking，そして一部の全称量化子を用いた論理式による property checking を実装している．本研究では UCLID の correspondence checking と property checking の機能を使用する．

Vol.2012-ARC-202 No.21
Vol.2012-HPC-137 No.21
2012/12/14

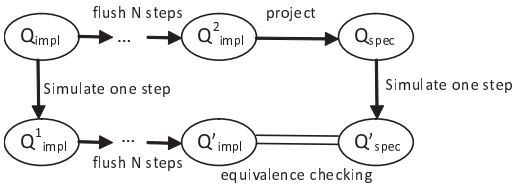

図 2　Correspondence Checking

3.2 Correspondence Checking

Correspondence checking はプロセッサの検証を目的として文献 [15] において初めて提案された手法である．Corresondence checking においては，プロセッサの実装モデルと命令セットモデル間で対応がとれることを検証する．

Correspondence checking では図 2 に示すように，実装モデルの初期状態 (Q_{impl}) から任意の入力を対象とした記号シミュレーションを 1 ステップ実行する ($Q_{impl} \rightarrow Q^1_{impl}$)．その後，プロセッサのパイプラインを N ステップかけてフラッシュして，全ての命令の実行が完了しプログラムから見た際のプロセッサの状態 (プログラムカウンタやメモリ・レジスタに保持されている値など) に命令の実行結果が反映された状態にする ($Q^1_{impl} \rightarrow Q'_{impl}$)．その後に，記号シミュレーションを開始した初期状態 (Q_{impl}) から再度記号シミュレーションを行う．今度は，実装モデルと命令セットモデルの状態間の対応付けが可能な状態へ到達するために，初期状態から直接 N ステップかけてパイプラインをフラッシュする ($Q_{impl} \rightarrow Q^2_{impl}$)．そして，実装モデルと命令セットモデルの状態間の対応付け (projection) を行う ($Q^2_{impl} \rightarrow Q_{spec}$)．さらに，命令セットモデル上で 1 ステップシミュレーションを行ない ($Q_{spec} \rightarrow Q'_{spec}$)，プログラムから見た際のプロセッサの状態が，実装モデルのもの (Q'_{impl}) と命令セットモデルのもの (Q'_{spec}) で等価であることを検証する．この等価性は以下の各式の論理積で表現される．

$$Q'_{impl}.\mathrm{PC} = Q'_{spec}.\mathrm{PC}\ ,\ Q'_{impl}.\mathrm{RF} = Q'_{spec}.\mathrm{RF}$$

$$Q'_{impl}.\mathrm{Mem} = Q'_{spec}.\mathrm{Mem}$$

4. 不具合診断手法の提案

本章では，形式的手法に基づく不具合診断手法の提案を行う．提案する手法は [5] において対話的な診断手法として提案されたものに基づくが，本研究では完全に自動化された診断手法を提案する．提案する手法は以下に示すような手順で構成される．

(1) 不具合を含むモデルの中で，不具合の原因になり得ると考えられる変数群を抽出する
(2) 不具合を含むモデルを元に，不具合の原因になり得ると考えられた各変数に対してマルチプレクサを挿入したモデルを作成する
(3) 正常動作するモデル・不具合を含むモデル・マルチプレクサを挿入したモデルの記号シミュレーションを行なって，最後にプロパティの検査を行うことにより不具合修正を可能とする変数の候補を抽出する
(4) 必要な変更規模の最小化を目的として，抽出された変数の候補群の最小化や変数の順位付けを行う

以降に，各手順の詳細を説明する．

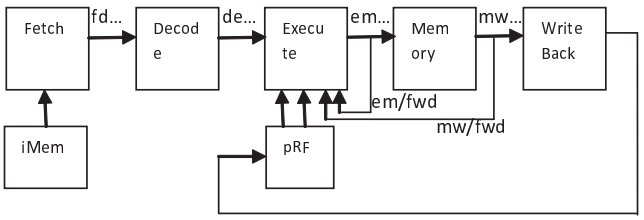

図 3　説明に用いるプロセッサの構成

4.1 説明に用いるプロセッサのモデル

手順の説明には，以下に記すような単純なプロセッサのモデルを用いる．モデルは，UCLID と共に配布されているプロセッサの設計例 DLX を簡略化したものとなっている．プロセッサは 5 ステージのパイプラインを持ち，レジスタ間での演算を行う命令を 1 種類，条件分岐を行う命令を 1 種類，それぞれ備える．プロセッサの構成を図 3 に示す．また，プロセッサを UCLID の入力言語を用いて記述したものを図 4 に示す．記述からは変数の定義や initial 文が削除されていることに注意されたい．記述内で分岐命令に対応する部分は赤い文字で示してある．以下では，この例を用いて提案する不具合診断・修正手法を説明する．

4.2 不具合の原因となり得る変数群の抽出

まず，不具合の原因となり得る変数群の抽出を行う．Correspondence checking による検証を行う際と同様に，提案手法ではデータパスは既に検証が行われており正しく動作するという前提で，uninterpreted function として抽象化する．よって，不具合の原因となり得る変数群には制御ロジックの変数が含まれ，それらは Boolean 型を持つ．抽出の際には，独立した変数として参照されていないような式についても中間変数を定義して抽出対象とする．図 4 においては，`fwd1` が導入された中間変数の例となる．

不具合の原因となり得る変数群には，設計に存在する全ての変数を含めることが可能であり，そのサイズは非常に大きなものとなってしまうことが考えられる．サイズを縮小する一つの手法としては設計者が不具合が存在すると考える箇所を手動で入力することが考えられる．

設計者が不具合が存在すると考える箇所を手動で入力する以外の手法としては，すでに検証が終了している既存の設計との変更点に着目する手法が考えられる．設計が逐次的に変更されていくという前提のもとでは，既存の設計から変更された箇所に不具合が含まれる可能性が高く，最初

Vol.2012-ARC-202 No.21
Vol.2012-HPC-137 No.21
2012/12/14

```
DEFINE
  fwd1 := case (* Forward Value 1 *)
    emValid & emType = RR &
      dest(emInstr) = src1(deInstr) : emValue;
    mwValid & mwDest = src1(deInstr) : mwData;
    default: deArg1;
  esac;
  fwd2 := case (* Forward Value 2 *)
    emValid & emType = RR &
      dest(emInstr) = src2(deInstr) : emValue;
    mwValid & mwDest = src2(deInstr) : mwData;
    default: deArg2;
  esac;
ASSIGN
  next[pPC] := case (* Program Counter *)
    emBranch : emTarget;     (* Branch *)
    default : succ(pPC);     (* Increment *)
  esac;
  (* Fetch Stage *)
  (* Read from instruction memory *)
  next[fdInstr] := imem(pPC);
  next[fdType] := itype(next[fdInstr]);
  next[fdPC] := pPC;
  next[fdValid] := ~emBranch;
  next[pRF] := Lambda(a) . case (* Register File *)
      (* Write back value *)
      mwValid & (a = mwDest) : mwData;
      default : pRF(a);
    esac;
  (* Decode stage *)
  next[deInstr] := fdInstr; (* Get from pipeline *)
  next[deType] := fdType;
  next[deArg1] := next[pRF](src1(fdInstr));
  next[deArg2] := next[pRF](src2(fdInstr));
  next[dePC] := fdPC;
  next[deValid] := ~emBranch & fdValid;
  (* Execute Stage *)
  next[emInstr] := deInstr;
  next[emType] := deType;
  next[emValue] := alu(op(deInstr), fwd1, fwd2);
  next[emBranch]:= ~emBranch&deValid & (deType=BR)
                   & take(op(deInstr),fwd1,fwd2;
  next[emTarget] := targ(dePC, imm(deInstr));
  next[emValid] := ~emBranch & deValid;
  (* Memory Stage *)
  next[mwData] := emValue;
  next[mwDest] := dest(emInstr);
  next[mwValid] := emValid & emType = RR;
```

図 4　説明に用いるプロセッサの UCLID による記述

に検証の対象とする必要があると考えられる．そのため，既存の設計から変更された箇所に出現する変数を不具合の原因となり得る変数群に含めることを提案する．さらに，含まれる変数に対して不具合の原因となり得る可能性の高さに対応するスコア付けを行うことも考えられる．具体的には，変更箇所に各変数が出現する回数をスコアとする．結果として，スコアは変数が設計が変更された箇所にどれくらいの影響度が有るかを表すようなものとなる．図 4 に示した例に含まれる変数 `emValid`,`emBranch` はそれぞれ 3,5 回，設計変更箇所に出現するため，これらの変数のスコアはそれぞれ 3 と 5 となる．本手法では，スコアの数値が高いほど，不具合の原因となりうる可能性が高いとみなす．そのため，図 4 に示した例では，`emBranch` のほうが `emValid` に比較して不具合の原因である可能性が高いとみなされる．

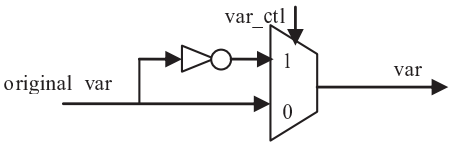

図 5　モデルへのマルチプレクサの挿入

4.3　マルチプレクサを挿入したモデルの作成

提案手法による不具合診断はマルチプレクサを挿入したモデルを用いて行われる．この不具合診断手法は RTL 記述やゲート・トランジスタレベルのネットリストを対象とした手法に近いものとなっているが，高い抽象度の記述を対象としている．不具合の原因となり得る変数群に含まれる各変数に対応する各配線に対して，図 5 に示すようにマルチプレクサを挿入する．

`var_ctl` は correspondence checking の際に任意の値を入力することが許される変数である．マルチプレクサの入力チャネル 0 には元の設計で変数が持つ値 (`original var`) を入力し，入力チャネル 1 にはその否定値を入力する．`var_ctl` に 1 が入力された場合，最終的な出力 `var` には元の設計の否定値が出力されることとなる．そのときに設計が期待された動作をするのであれば，マルチプレクサが挿入された箇所の設計を変更することが不具合修正の候補の一つであるということができる．本手法が扱っているプロセッサの制御ロジックに含まれる変数は全て Boolean 型であることを前提としているため，ある変数の値が期待された動作を妨げる場合，その値を反転すれば良い．

4.4　不具合修正を可能とする変数候補の抽出

本手法では，設計の不具合が単一変数の値を反転することによって修正可能という前提をおく．そのため，不具合修正を可能とする変数候補の抽出の際には，一度に一箇所ずつ不具合の原因となり得る変数を選択して対応する配線にマルチプレクサを挿入して，マルチプレクサが挿入したモデル群を作成する．

正常動作することが確認されている仕様に基づくモデル (ISA) と，それとは別の実装に基づくモデル (Impl) が与えられた際には，以下のプロパティを検証することによって両モデル間の等価性を検証することができる．

$$P_{impl} \cong \text{Impl.PC} = \text{ISA.PC} \;\&\; \text{Impl.RF} = \text{ISA.RF} \;\&\; \text{Impl.Mem} = \text{ISA.Mem} \quad (1)$$

同様にして，以下のプロパティを検証することによって仕様に基づくモデル (ISA) とマルチプレクサを挿入したモデル (Enr) の等価性を検証することができる．

$$P_{enr} \cong \text{Enr.PC} = \text{ISA.PC} \;\&\; \text{Enr.RF} = \text{ISA.RF} \;\&\; \text{Enr.Mem} = \text{ISA.Mem} \quad (2)$$

(1,2) 式に含まれる変数で，PC はプログラムカウンタ，RF はレジスタファイル，Mem はメモリに対応する．

本研究では，実装に基づくモデル (Impl) が仕様に基づくモデル (ISA) と等価ではないために正しく動作せず，マルチプレクサを挿入したモデル (Enr) が仕様に基づくモデルと等価になって正しく動作する状況を求めることを目的とする．その際，プロパティを検査して違反される反例を求めることによって状況を求める．具体的には，以下のようなプロパティ P を検査する．

$$P \cong \sim (\sim P_{impl} \& P_{enr}) \quad (3)$$

ISA,Impl,Enr の各モデルを同一初期状態から記号シミュレーションしたとすると，$\sim P_{impl}\&P_{enr}$ が成立し得るということは実装に基づくモデル (Impl) にはプロパティに違反する不具合が存在して，マルチプレクサを挿入したモデル (Enr) には不具合が存在しないような状況が存在することを意味する．つまり，その際には挿入されたマルチプレクサによって不具合の修正が可能である．プロパティ P は $\sim P_{impl}\&P_{enr}$ の否定を取ったものとなっているので，P が違反される反例が存在することは不具合の修正が可能なことを意味する．反例は，どのサイクルでマルチプレクサの出力を反転させると不具合修正が実現できるかを示す．一方，P が常に充足される場合は，そのマルチプレクサを用いた不具合の修正方法がないことを意味する．上記のようにして，ある特定の反例に対して，実装モデルにマルチプレクサを挿入して信号を反転してプロパティに違反しないように制御することが可能なモデルを得ることができる．P が違反された際にマルチプレクサの選択信号 `var_ctl` に 1 が入力されている場合，マルチプレクサが挿入されている配線に対応する変数の値が正しくなく反転する必要があることを意味する．

P に対する反例を求める際には，最初は何ら制約を導入しない．この段階で UCLID によって反例を求めることができなかった際には，修正を可能とする変数の候補が得られず，修正できない．反例が得られた際には，第 5 節において提案する不具合修正手法の適用を行う．図 1 にも示すとおり，手法を用いて不具合修正ができなかった際には別の反例を用いた修正手法の適用を行う．別の反例を得るために，UCLID へ入力するプロパティには既に得られた反例を除外するような制約を追加する．異なる反例を用いた修正手法の適用は，不具合が修正できるか修正が不可能であると判断されるまで繰り返される．この詳細は第 5 節に記す．

4.5 変数候補群の最小化と変数の順位付け

変数候補の抽出を行った後には，修正を実現できない変数を除去して変数候補群の最小化を行う．修正を実現できない変数とは，変数に対応する配線にマルチプレクサを導入したモデルを用いた際には，(3) 式に示したプロパティ P を常に充足するような変数を指す．それは P が充足される際には，実装に基づくモデル (Impl) で不具合を発生させ，マルチプレクサを挿入したモデル (Enr) で不具合を発生させない命令列が存在しないことを意味するためである．その後に，第 4.2 節において提案した評価関数を用いて変数の順位付けを行う．用いる評価関数は，その変数を変更すると設計の他の部分にどの程度の影響度があるかを示すもので，静的解析の結果に応じて設計の他の部分に影響がある変数により高い優先順位を与える．

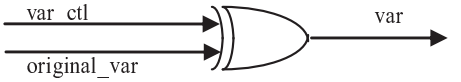

図 **6** XOR ゲートを用いた不具合修正回路

5. 不具合修正とその検証手法の提案

これまでにも記した通り，提案する不具合診断・修正手法はプロセッサの制御ロジックを対象としている．それ以外のデータパス上に存在する演算ユニットなどに対しては他の技術を用いた検証が行われていることを前提とする．そのため，Boolean 型の制御変数に代入される値を修正することとなる．代入される値は，他の変数を入力として表現可能な Boolean 関数の形式をとり，実際の回路では組み合わせ回路として実装される．

制御ロジックの不具合を修正するということは，変数に代入される値を決定する Boolean 関数を修正することとなる．修正の際には，図 5 に示すような，修正が行われる必要があるかを示す変数 `var_ctl` と修正前の関数の出力 `original_var` を用いて修正後の出力 `var` を決定するが，`var_ctl` が 1 の際にのみ `var` が `original_var` を反転したものとなるため，図 5 に示したマルチプレクサを用いた回路の代わりに図 6 に示すような XOR ゲートを用いた回路を使用することができる．

5.1 不具合修正手法

制御ロジックの不具合を修正することは，修正前の出力 `orignal_var` から修正後の出力 `var` を変更するかどうかを決定する `var_ctl` 信号の値を，設計内に含まれる他の変数の前のサイクルの値を用いて生成することに他ならない．一般的には，`var_ctl` 信号は全ての変数の値を入力として決定することができるが，修正の際に全ての変数を入力として用いるか検討することには以下のような問題が存在する．まず，全ての変数を検討対象として `var_ctl` を表現する式を構築すると，無駄に大きな式ができてしまう可能性がある．式に含まれる変数の一部は，`var_ctl` の決定に必要ない場合も考えられる．

そこで，以下のような制御ロジックの修正方法を提案す

る．提案手法では文献 [6], [7] において導入されたプロセッサに生じるバグのモデルを参考にする．これらの文献によると，プロセッサの設計不具合は提案手法で修正の対象としている制御ロジックに発生する．また，それらの不具合は多くの場合単純で，制御ロジックにおいて単一の条件をカバーし忘れたり単一の信号を考慮し忘れるようなものである．そのため，提案手法では以下のような戦略で不具合の修正を行う．ある特定の Boolean 関数 $F(v_1, ..., v_i)$ として表現可能な制御ロジックの不具合を修正する際には，まず新たな入力として用いる変数を追加せずに修正を行うことを試みる．変数を追加しない修正が不可能な場合，修正が実現されるまで変数を一度に一個ずつ追加して修正を試みる．変数を追加することは，修正前の回路には存在しない入力信号を追加することに対応する．

5.1.1 入力変数を追加しない不具合の修正

変数を追加しないで修正を行う際には，修正を行う必要があり `var_ctl` が 1 となる各サイクルにおいて，修正対象の関数に既に引数として含まれる変数の値を用いた積項を構成して修正に用いる．`orignal_var` が変数 $v_1, ..., v_i$ の関数として表現される場合，`var_ctl` が 1 となる各サイクル j に対して，積項 $C_j(v_1, ..., v_i)$ を構成する．そのような積項が 1 となる各サイクルで 1 となる信号として `var_ctl` は表現することが可能である．

以下では図 4 に示した単純な例を用いて説明を行う．ここでは，変数 `emValid` の値が 4 サイクル目に 1 となることが期待されているとする．`emValid` のとる値は依存する変数が元の設計と変化しないという前提をおくと 1 サイクル前の `emBranch` と `deValid` を入力とする関数となるため，提案手法では入力となる変数の反例が得られた 1 サイクル前つまり 3 サイクル目の両変数の値を調べる．その値が仮にそれぞれ 1 と 0 であったとした際には，積項として `emBranch & ~deValid` を得る．結局，修正後の `emValid` の値は修正前の `emValid` を表現する関数と得られた積項の排他的論理和となり，以下の式で表されることとなる．

```
(~emBranch & deValid) & (emBranch & ~deValid)
```

なお，この例について得られた 4 サイクル目の `emValid` のみを変更する修正は，`emBranch`,`deValid` がそれぞれ 1,0 となるようなサイクルが 3 サイクル目以外に存在しない場合のみ妥当であることに注意する必要がある．他のサイクルでも 3 サイクル目と両変数の値が同一となり，その際に修正を行なってはいけないような状況では，この修正は妥当ではない．他のサイクルでも両変数の値が同一となると必ず `var_ctl` が 1 となって修正が行われるためである．

上記では，第 4 節の不具合診断手法により得られた反例の 1 つを用いた不具合修正の手順を説明した．このようにして得られた修正が第 5.2 節に示す検証手法により正しくないと判断された際には，第 4 節の手法を用いて更に反例を作成して不具合の修正を試みる．作成した反例を追加して，これまでに不具合修正に用いた反例と追加した反例の全てにおいて修正対象となる変数が修正されるようにした式を作成し，その 妥当性を検証する．検証によって修正が正しいと証明されるか，これ以上反例が得られないまで，その操作を繰り返す．なお，新たな反例が得られなくなった状態でも正しい修正が得られないことは，入力変数を追加せずに正しく不具合を修正することが不可能であることを表す．なお，追加する反例の作成の際には不具合の修正を行う前の実装に基づくモデルを使用する．

5.1.2 入力変数を追加する不具合の修正

入力変数を追加しない不具合の修正ができなかった際には，変数を追加して不具合の修正を行う．変数の追加の際には，第 4.2 節で導入した不具合の原因となりうる可能性を表すスコアに基づいて，一度に 1 変数ずつ制御ロジックの入力に含めた場合に不具合の修正が可能であるかを調べる．1 変数のみを追加した際に不具合を修正できない場合には，スコアが高い変数の組から順に 2 変数を追加して不具合が修正できないかを調べる．この手続きは不具合が修正されるか，追加する変数の組み合わせとして全ての可能な組み合わせを用いても正しく不具合を修正できないという結論が得られるまで行う．このようにして，変数を追加しないと修正ができない不具合を修正することが可能である．修正に対する検証を行なって，修正が正しくないと判断されたときの処理は変数を追加しない修正を行うときと同様のものとなる．

変数を追加することによる不具合の修正例を図 4 に出現する `emValid` 例を用いて説明する．`emValid` を表現する関数は，不具合の修正前には `emBranch` と `deValid` のみを含むが，不具合の修正時に `mwValid` も含めることを許容して `emBranch`,`deValid`,`mwValid` によって表現することとする．その際には，修正が必要な条件を表す積項はその 3 変数を含むものとなる．以降は，変数を追加しない場合と同様その 3 変数を用いた修正を行う．

5.2 不具合修正の検証手法

第 5.1 節の手順で得られた不具合修正を経たモデルは，修正を求める際に使用された反例に対応する入力においてのみ正しく動作することが保証されている．多くの反例を使用した修正を行った際にも，やはり，修正に用いた反例に対応する入力を与えられた際に正しく動作することのみが保証される．そのため，他の入力を与えられた際にも不具合修正後の設計が正しく動作することを保証する，不具合修正に対する検証を行う必要がある．

不具合修正に対する検証を行う手段としては，第 3.2 節に紹介した形式的手法に基づく correspondance checking を行う．これはシミュレーションを検証の際に用いる多くの既存手法に対する重要な差異の一つである．シミュレー

Vol.2012-ARC-202 No.21
Vol.2012-HPC-137 No.21
2012/12/14

ションを用いる既存手法は，充分なカバレッジを実現するテストパターンを多数用意してシミュレーションを行って不具合修正の妥当性を検証する必要がある．しかし，提案手法では抽象度の高い記述を用いて実装を修正した後のモデルが仕様に基づくモデルと等価な動作をすることを形式的に検証して，修正の妥当性を保証する．

6. 提案手法の評価

6.1 評価に用いたプロセッサ

提案手法はスーパスカラ・アウトオブオーダ実行機構を備えるプロセッサを例題として評価を行った．スーパスカラ構成を用いたこのプロセッサでは同時に複数の命令がフェッチ・デコードされる．デコード後に命令はリオーダバッファに送られ，演算対象のオペランドのデータがアクセス可能な命令からアウトオブオーダ実行される．演算結果はリオーダバッファに保存され，その後レジスタファイル・メモリにインオーダで書き戻される．プロセッサは RAZOR-II[16] タイミングエラー回復機構も備える．

6.2 提案手法の実装

提案する自動的な不具合の診断・修正手法を，6,500 行程度の C++で記述されたプログラムとして実装を行った．このプログラムは診断・修正対象の設計および正常動作する設計の UCLID 記述によるモデルを入力とする．そして，前者のモデルに対してマルチプレクサを挿入して UCLID による (3) 式に示したプロパティ（とさらなる反例を得るために修正したプロパティ）の反例を収集する．そして，その反例群を元に修正を行ったモデルを UCLID 記述として出力する．反例の収集などには UCLID v.3[2] を使用し，UCLID は SAT ソルバとして Minisat 2.2[17] を使用する．

評価の際には 2.5GHz で動作する Intel Core2Duo CPU と 4GB のメモリを搭載した PC を使用した．

6.3 不具合の挿入とその診断・修正

提案手法の評価を目的として，第 6.1 節で説明を行ったプロセッサに対して以降に示すような不具合を挿入した．

まず最初に，プロセッサのエラー回復機構に関係のない不具合を挿入した．挿入の際には記述に用いられている変数をランダムに一つ選択して，変数の値を決定する case 文のうちエラー回復動作に関係ないものを削除した．また，第 4.2 節では，設計の変更箇所に着目して不具合の原因となり得る変数群を抽出する手法を提案したが，この実験では全ての変数を不具合の原因となり得る変数群に含めた．

記号シミュレーションを行なってプロセッサが正しく動作することを検証するためには 11 サイクルのシミュレーションを行う必要があった．そのため，リセット状態に対応する具体値を持つ最初のサイクルを除いた 10 サイクルにおいて，各変数に対する不具合の修正の可能性があることになる．

表 1 プロセッサのエラー回復機構以外の箇所に挿入した不具合を用いた評価結果

Variable	# UCLID runs	# candidate cycles	# potential fix cycles	Error Corrected?	diagnosis time (s)	correction/certification time (s)	overall time (s)
Buggy	1	2	1	Y	17.3	5.8	23.1
Worst	8	12	11	N	132.1	65.7	197.8
Median	4	8	6	N	71.1	34.6	105.7
Best	1	1	0	N	6.3	0.0	6.3

設計内の各変数にマルチプレクサを挿入して不具合修正を試みた結果を表 1 に示す．最初の行 (Buggy) は不具合の挿入を行う際に選択した変数を対象として不具合修正を行った際の結果を表す．他の行 (Worst/Median/Best) は，不具合が挿入されていない変数を対象として不具合修正を試みて不具合が修正できないとの結論が得られたときの結果を表す．なおこの 3 行は，表の 2 列目に記した UCLID を実行する必要があった回数がそれぞれ最悪値・中央値・最良値であった際の結果を表す．

表の 3 列目に示したのは UCLID を繰り返し実行して行った不具合修正を可能とする変数候補の抽出 (第 4.4 節) の際に，マルチプレクサの制御信号を 1 としてマルチプレクサが挿入された変数の値を反転すると期待された出力が得られるとされたサイクル数の累計である．4 列目には，入力変数を追加しない不具合修正 (第 5.1.1 節) を行なって期待した出力を得られたサイクル数の累計を示した．ただし，不具合修正を行なって期待した出力を得られた場合にも最初の行 (Buggy) に対応する修正以外は第 5.2 節に示した検証の結果，不具合を正しく修正できないとの結果が得られていることに注意する必要がある．挿入した，一部の case 文を削除するという不具合の性質上，入力変数を追加しない不具合修正手法で不具合を修正できている．

表の 6 列目と 7 列目には，修正を行なうための反例を得るプロパティ ((3) 式) を繰り返し検査する操作の所要時間と修正後のモデルに対して正しく動作することが保証されているモデルとの correspondence checking を行なって不具合修正を検証した際の所要時間を示す．最後の 8 列目には，提案手法全工程の所要時間を示す．なお，提案手法の全工程所要時間のうち，独自のプログラムで実装を行ったマルチプレクサの挿入や反例の解析の部分は充分小さく，UCLID とその内部で動作する SAT ソルバの実行時間が大部分を占める．

上記の評価結果においても確認された通り，本手法では実際に設計が正しく動作するような修正のみが最終的な結果として出力される．また，ある変数に着目して不具合の

修正を試みた際には，4 回程度の UCLID の実行で修正が可能であるかどうかの判定ができている．

次に，第 4.2 節において提案を行った，設計の変更箇所に着目して不具合の診断・修正を行う手法の評価を行った．この際には，プロセッサのタイミングエラー回復機構に不具合を挿入した．そして，タイミングエラー回復機構を備えないプロセッサとの設計の変更点だけでなく設計全体を対象として不具合の診断・修正を行う場合と，変更点に対応するタイミングエラー回復機構にのみ着目して不具合の診断・修正を行う場合の比較を行った．多くの設計は逐次的に変更が行われて，最近に変更が行われた部分に不具合が存在する可能性が高いことを考えると，設計間の変更点に着目して診断を行う手法は有用であると考えられる．結果，設計全体を対象として不具合の診断・修正を行なった際には 12 変数が診断の対象となり不具合が挿入された変数を発見して修正するまでに 10 変数を用いた不具合診断・修正を試みる必要があった．一方，タイミングエラー回復機構を備えないプロセッサとの差分に相当するタイミングエラー回復機構のみを対象とした際には，7 変数を用いた不具合・修正を試みることによって，不具合が挿入された変数を発見して修正することができた．

7. まとめと今後の課題

本稿では，形式的手法を用いたプロセッサに含まれる設計不具合の自動診断・修正手法を提案して評価した．逐次的に命令を実行して正常動作するプロセッサのモデルと，スーパースカラ・アウトオフオーダ実行などの高性能化を実現するが正常動作しないプロセッサのモデルをが与えられ，両モデルが等価な動作をしない際に，提案手法は形式的手法に基づき設計のどの部分に不具合があるかを診断して修正手法を求める．求めた修正手法の正しさは形式的手法に基づく correspondence checking により検証する．

提案した手法は，スーパースカラ・アウトオブオーダ実行やタイミングエラー回復機構を備えたプロセッサを例題として評価され，例題に挿入された不具合を検出・修正できることが確認された．さらに，逐次的な設計の変更が行われた際に，設計の変更点にのみ着目して不具合の診断・修正を行う手法についても，提案・評価を行った．

今後の研究課題としては，設計に複数の不具合が含まれた際の自動診断・修正手法や，設計の RTL 記述を入力とした手法，そして不具合の発生箇所を探索する際により効率良く候補を絞り込む手法などが挙げられる．

参考文献

[1] R.E. Bryant, S.K. Lahiri, and S.A. Seshia. Modeling and Verifying Systems Using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions. In *Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 78–92. Springer Berlin Heidelberg, 2002.

[2] *UCLID ver. 3.0.* available on-line at http://uclid.eecs.berkley.edu/wiki/index.php/Downloads.

[3] *Abstracting Verilog Designs to the Term-Level for Verification with UCLID.* available on-line at http://uclid.eecs.berkeley.edu/v2ucl/.

[4] G. Fey, S. Staber, R. Bloem, and R. Drechsler. Automatic Fault Localization for Property Checking. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(6):1138 –1149, 2008.

[5] B. Alizadeh, A.M. Gharehbaghi, and M. Fujita. Pipelined Microprocessors Optimization and Debugging. In *Reconfigurable Computing: Architectures, Tools and Applications*, volume 5992 of *Lecture Notes in Computer Science*, pages 435–444. Springer Berlin Heidelberg, 2010.

[6] D. Van Campenhout, T. Mudge, and J.P. Hayes. Collection and Analysis of Microprocessor Design Errors. *Design Test of Computers, IEEE*, 17(4):51 –60, 2000.

[7] M.N. Velev. Collection of High-Level Microprocessor Bugs from Formal Verification of Pipelined and Superscalar Desgins. *Internal Test Confernce*, pages 138–147, 2003.

[8] Yu-Shen Y., S. Sinha, A. Veneris, and R.K. Brayton. Automating Logic Rectification by Approximate SPFDs. In *Asia and South Pacific Design Automation Confernce, 2007*, pages 402 –407, 2007.

[9] A. Kuehlmann, D.I. Cheng, A. Srinivasan, and D.P. LaPotin. Error Diagnosis for Transistor-Level Verification. In *Design Automation Confernce, 1994*, pages 218 – 224, 1994.

[10] J.C. Madre, O. Coudert, and J.P. Billon. Automating the Diagnosis and the Rectification of Design Errors with PRIAM. In *Computer-Aided Design, 1989. Digest of Technical Papers, 1989 IEEE International Conference on*, pages 30 –33, 1989.

[11] A. Veneris and I.N. Hajj. Design Error Diagnosis and Correction via Test Vector Simulation . *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 18(12):1803 –1816, 1999.

[12] Yibin Chen, S. Safarpour, J. Marques-Silva, and A. Veneris. Automated Design Debugging With Maximum Satisfiability. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 29(11):1804 –1817, 2010.

[13] V. Boppana, I. Ghosh, R. Mukherjee, J. Jain, and M. Fujita. Hierarchical Error Diagnosis Targeting RTL Circuits. In *VLSI Design, 2000. Thirteenth International Conference on*, pages 436 –441, 2000.

[14] Kai hui Chang, I. Wagner, V. Bertacco, and I.L. Markov. Automatic Error Diagnosis and Correction for RTL Designs. In *High Level Design Validation and Test Workshop, 2007. HLVDT 2007. IEEE International*, pages 65 –72, 2007.

[15] J. Burch and D. Dill. Automatic verification of pipelined microprocessor control. In *Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 68–80. Springer Berlin Heidelberg, 1994.

[16] S. Das, C. Tokunaga, S. Pant, W.-H. Ma, S. Kalaiselvan, K. Lai, D.M. Bull, and D.T. Blaauw. RazorII: In Situ Error Detection and Correction for PVT and SER Tolerance. *Solid-State Circuits, IEEE Journal of*, 44(1):32 –48, 2009.

[17] *Minisat ver. 2.2.* available on-line at http://minisat.se/MiniSat.html.