社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

内部等価点の推定によるルールベース高位検証の高精度化

吉田　浩章[†,††]　　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 (VDEC)
〒 113–8656 東京都文京区本郷 7–3–1
†† 科学技術振興機構 戦略的創造研究推進事業 CREST
E-mail: †hiroaki@cad.t.u-tokyo.ac.jp, ††fujita@ee.t.u-tokyo.ac.jp

あらまし　高位設計記述間のルールベース等価性検証では，内部変数の等価点を前提として静的な依存関係や制御フローに基づいて定義された等価性規則をボトムアップに適用することで等価性を示す．従来手法では変数の名前によって内部等価点を求めるため，変数名の変更などにより対応が取れない場合には等価性を証明できない．本論文では，ランダムシミュレーションにより内部等価点を推定することによって，より正確な検証を実現する手法を提案する．また，例題を用いた計算機実験では本提案手法が現実的な最適化を適用した場合の等価性を高速に判定することが可能であることを示した．

キーワード　システムレベル設計, 形式的検証, 内部等価点, ランダムシミュレーション

Improving the Accuracy of Rule-based Equivalence Checking of High-level Descriptions by Identifying Potential Internal Equivalences

Hiroaki YOSHIDA[†,††] and Masahiro FUJITA[†,††]

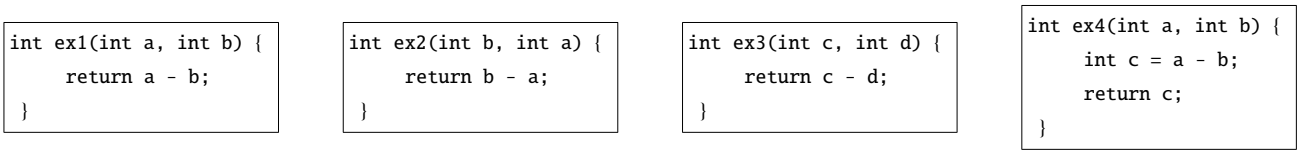
† VLSI Design and Education Center(VDEC), University of Tokyo
7–3–1 Hongo, Bunkyo-ku, Tokyo, 113–8656, Japan
†† CREST, Japan Science and Technology Agency
E-mail: †hiroaki@cad.t.u-tokyo.ac.jp, ††fujita@ee.t.u-tokyo.ac.jp

Abstract　Rule-based equivalence checking of high-level design descriptions proves the equivalence of two high-level design descriptions by applying the equivalence rules in a bottomup manner. Since the previous work derives the equivalence of the internal variables based on their names, the method often fails to prove the equivalence when variable names are changed. This paper proposes a method for improving the accuracy of the rule-based equivalence checking by identifying potential internal equivalences using random simulation. Experimental results using an example design shows that the proposed method can prove the equivalence of the designs before and after a practical design optimization.

Keywords　System-level design, formal verification, internal equivalences, random simulation

1. はじめに

近年の半導体技術の飛躍的な進歩によってシステム LSI の大規模化が進むにつれて，設計にかかる期間・コストが増大している．設計効率化の有効なアプローチとして，従来のレジスタ転送レベル設計よりも抽象度の高いシステムレベルにおける設計が注目されている．システムレベル設計ではシステム全体をシステムレベル設計記述言語によって記述するため，ソフトウェアとハードウェアを協調して設計できる，設計再利用が容易である，抽象度の高い記述により記述量を削減できる，などの利点により設計生産性の向上が期待できる．システムレベル設計記述言語としては，C 言語をベースにした SpecC 言語 [1] や SystemC 言語 [2] が広く使用されている．システムレベル設計ではその設計を支援する CAD 技術が重要であり，そのためシステムレベル自動合成・最適化技術の研究開発が活発に行われている．



(a) 元の設計　(b) 変数名が変更された場合　(c) 変数の対応が発見できない場合　(d) 中間変数など対応する変数がない場合

図 1　名前に基づく初期等価性によって等価性を証明できない例

このような利点の一方で，システムレベル設計における設計誤りを残したまま設計を進めてしまうと，後の段階において設計誤りが発見された場合には，再びシステムレベルからの設計手戻りが生じてしまい，結果的に設計コストの増大となってしまう．そのため，できるだけ早い段階での設計誤りの発見が最重要課題である．このような検証には，テストパタンを用いたシミュレーションでなく，網羅的に検証が可能な形式的検証が好ましい．システムレベル設計記述に対する形式的検証技術はこれまでいくつかの研究結果 [3]〜[5] が報告されているが，より効率的な手法の研究開発が求められている．

現在よく用いられているシステムレベル設計手法 [6], [7] では，設計記述に対して対話的な設計の変換や詳細化を支援するツールを提供している．この手法では，設計記述を主に人手で局所的かつ段階的に変更していくことで最終的な設計記述を作成する．よって，この各段階において変換前後の等価性を確認しながら設計を進めていくことによって設計誤りを防ぐことが可能となる．

このような用途を目的とした高位記述の検証手法として，ルールベース等価性検証が提案されている [8], [9]．この手法では静的な依存関係やプログラムフローに基づいて定義された等価性規則をボトムアップに適用することで 2 設計間の等価性を示す．等価性規則を部分グラフのパターンとして見なすことで，ルールベース等価性検証は 2 抽象構文木間のマッチング問題とみなすことができるため，ルールベース等価性検証手法は効率的に検証を行うことが可能である．またこの手法では，設計記述における依存関係や制御フローを抽象構文木に追加した拡張システム依存グラフ ExSDG [10] を用いることで，効率的な実装を実現している．

ルールベース等価性検証ではボトムアップに等価性を証明していくため，変数といった設計の基本構成要素の等価性をあらかじめ求めておく必要がある．どのようにしてこの初期等価点を発見するかが課題となる．[8], [9] では，ボトムアップ証明の際の初期等価点をどう求めるかを明らかにしていない．また [8], [9] における実装では名前に基づいて初期等価点を求めているが，以下のような場合には検証は失敗してしまう．

- 変数名が変更された場合
- 変数の対応が発見できない場合
- 中間変数など対応する変数がない場合

このような例を図 1 に示す．図 1(a)-(d) はどれも機能的に等価であるが，名前に基づいた初期等価点ではこれらの等価性を証明することはできない．上記のような設計変更は頻繁に行われるものであり，これらの変更に対して検証ができないことは実用的に問題がある．

本論文では，内部等価点をランダムシミュレーションによって推定することで上記のような場合においても検証を可能にする手法を提案する．本論文は次のような構成となっている．まず第 2 節と第 3 節において，提案手法で用いる要素技術である拡張システム依存グラフ ExSDG とルールベース検証手法について説明する．次に，提案手法であるランダムシミュレーションによる内部等価点推定手法および推定された内部等価点に基づく等価性検証手法について説明する．最後に計算機実験結果を示す．

2. 拡張システム依存グラフ ExSDG

システム依存グラフ (System Dependence Graph; SDG) [11] は，Horwitz らによって提案されたグラフ構造で，主にプログラムスライシング手法において使用されている．SDG では各節が設計記述内の文や表現に対応しており，それらの間の依存関係を辺として表現する．主な依存関係には以下に示すデータ依存，制御依存，宣言依存がある．

データ依存　文 s1 において代入された変数が，他の文で再代入されることなく文 s2 において使用される場合，s1 から s2 にデータ依存が存在する．

制御依存　文 s2 の実行が文 s1 によって制御される場合，s1 から s2 に制御依存が存在する．

宣言依存　文 s1 において宣言された変数が，文 s2 において使用される場合，s1 から s2 に宣言依存が存在する．

拡張システム依存グラフ (ExSDG) は，抽象構文木に SDG の依存辺を追加したものである [10]．システム設計記述を ExSDG によって表現することで，元の記述を構文的な表現と設計記述内の依存関係を同時に効率的に表現することができる．拡張システム依存グラフの例を図

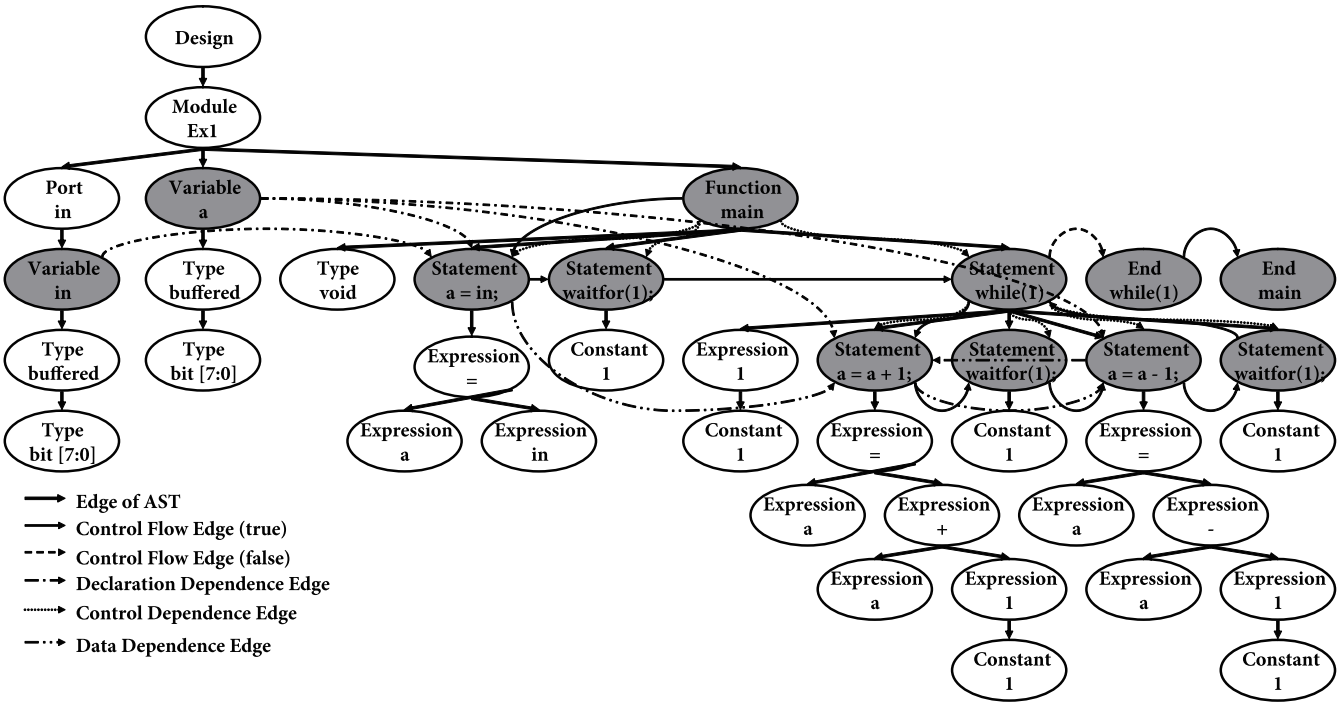


図 2　拡張システム依存グラフ ExSDG の例 [10]

2 に示す．ルールベース等価性検証において，各等価性規則は ExSDG のパターンとして表現される．

3. ルールベース等価性検証手法

ルールベース等価性検証手法では静的な依存関係や制御フローに基づいて定義された等価性規則をボトムアップに適用することで 2 設計間の等価性を示す．ここでは 2 つの ExSDG において各変数および定数の等価性はすでに求めてあると仮定する．各設計は ExSDG によって表現されているとすると，各等価性規則を ExSDG 上のパターンとして見なすことができる．よってルールベース等価性検証は 2 抽象構文木間のマッチング問題とみなすことができるため，ルールベース等価性検証手法は効率的に検証を行うことが可能である．各等価性規則は有用であると思われる規則を列挙したもので，規則の集合は十分ではあるが必要十分ではない，つまりどのような設計に対しても検証可能であるというわけではないということに注意されたい．

以下に主な等価性規則を示す．以下の規則において $\equiv$ は設計間の等価性関係を，$\triangleq$ は定義関係を表す．特に明示がない限り文・式は ExSDG 中の節を表すものとする．

ルール 1 (式) *2* つの式に対応する抽象構文木が等価であれば，*2* 式も等価である．抽象構文木における式の等価性は，交換法則・結合法則・分配法則といった四則演算の性質 *(*例*:* $c*(a+b) \equiv (b+a)*c$*)* や，等号・不等号の性質 *(*例*:* $x < y \equiv y > x$*)* といった性質を含めて判定される．

ルール 2 (代入文) ある代入文 N_{A2} において右辺に現れるある変数を N_V とし，N_E をある式とする．このとき，以下のすべての条件が成立するならば $N_V \equiv N_E$ である．

（*1*） N_V が代入文 N_{A1} において定義されているとき，N_{A1} の右辺と N_E が等価である．

（*2*） N_{A1} から N_{A2} の間に，N_{A1} の右辺に現れる変数の再定義が行われない．

ルール 3 (条件文) 条件文は *C* 言語などにおける *if* 文であり，条件式，*Then* 節と *Else* 節を持つ．*2* つの条件文 S_1,S_2 を $S_1 \triangleq$ ***if*** C_1 ***then*** N_{T1} ***else*** N_{E1} と $S_2 \triangleq$ ***if*** C_2 ***then*** N_{T2} ***else*** N_{E2} とする．

ルール *3.1* $C_1 \equiv C_2 \wedge N_{T1} \equiv N_{T2} \wedge N_{E1} \equiv N_{E2}$ ならば $S_1 \equiv S_2$

ルール *3.1* $C_1 \equiv \neg C_2 \wedge N_{T1} \equiv N_{E2} \wedge N_{E1} \equiv N_{T2}$ ならば $S_1 \equiv S_2$

ルール 4 (逐次合成) *C* 言語などの手続き型言語では，複数の文からなる部分 *(*複文*)* は指定された順序に従い逐次的に実行される．ある複文はそれらの間にデータ依存がない限り，実行順序を変えても機能的に等価である．*2* つの複文 P_1,P_2 を $P_1 \triangleq N_{1,1};...;N_{1,n}$ と $P_2 \triangleq N_{2,1};...;N_{2,n}$ とする．このとき以下のすべての条件が成立するとき $P_1 \equiv P_2$ である．

（*1*） すべての i に対して $N_{1,i} \equiv N_{2,\pi(i)}$ となるような置換写像 π が存在する．

（*2*） すべての P_1 におけるデータ依存 $N_{1,i} \longrightarrow N_{1,j}$ に対して，P_2 において $N_{2,\pi(j)}$ が $N_{2,\pi(i)}$ より後に実行される．

```
L1:  x0 -= x1;
L2:  x8 = x0 + x1;
L3:  x2 = x1 - (W2 + W6) * x2;
```
(a) 元の設計

```
L2:  x8 = x0 + x1;
L1:  x0 -= x1;
L3:  x2 = x1 - (W2 + W6) * x2;
```
(b) (a) と非等価

```
L1:  x0 -= x1;
L3:  x2 = x1 - (W2 + W6) * x2;
L2:  x8 = x0 + x1;
```
(c) (a) と等価

図 3　逐次合成規則による等価性判定例

（*3*）　すべての P_2 におけるデータ依存 $N_{2,\pi(i)} \longrightarrow N_{2,\pi(j)}$ に対して，P_1 において $N_{1,j}$ が $N_{1,i}$ より後に実行される．例えば図 *3* の例では，*x0* に関して *L1* から *L2* に対してデータ依存が存在する．そのため *L1* と *L2* を入れ替えた *(b)* は *(a)* と非等価となる．一方で *L2* と *L3* の間にはデータ依存が存在しないため，*L2* と *L3* を入れ替えた *(c)* は *(a)* と等価である．

ルール 5 (ループ) *for* 文や *while* 文などのループ文に関しては，ループ展開されているものと仮定する．よって，ループの等価性は上記の逐次合成ルールによって判定される．

ルール 6 (並列合成) *SpecC* 言語などのシステムレベル設計記述言語では，複数の文の並列実行を指定する *par* 文が存在する．

ルール *6.1* (並列-並列)　2 つの複文を $C_{P1} \triangleq \textbf{\textit{par}}\{N_{1,1};...;N_{1,k}\}$ と $C_{P2} \triangleq \textbf{\textit{par}}\{N_{2,1};...;N_{2,k}\}$ とする．このとき，$\forall i, 1 \leqq i \leqq k, N_{1,i} \equiv N_{2,\pi(i)}$ となる置換写像 π が存在するならば，$C_{P1} \equiv C_{P2}$ である．

ルール *6.2* (逐次-並列) 2 つの複文を $C_S \triangleq N_{1,1};...;N_{1,k}$ と $C_P \triangleq \textbf{\textit{par}}\{N_{2,1};...;N_{2,k}\}$ とする．このとき，$\forall i, 1 \leqq i \leqq k, N_{1,i} \equiv N_{2,\pi(i)}$ となる置換写像 π が存在し，かつどの i,j $(i \neq j)$ に対してもデータ依存 $N_i \longrightarrow N_j$ が存在しなければ，$C_S \equiv C_P$ である．

4. 提案手法

4.1 ランダムシミュレーションによる内部等価点推定

前節で説明したように，ルールベース等価性検証手法では 2 設計間の変数の等価性はすでに求まっているものとして等価性規則をボトムアップに適用することで設計全体の等価性を証明する．[8], [9] ではこの初期内部等価点をどのようにして求めるかについては言及しておらず，実装においては変数の名前に基づいて等価性を推定している．しかし図 1 で説明したように，名前に基づいた変数の内部等価点推定では典型的な設計変更に対しても検証に失敗してしまう．

本稿ではランダムシミュレーションを用いた内部等価点推定手法を提案する．ランダムに生成したある入力列を検証対象の 2 設計に与えてシミュレーションを実行し，同じ値の列を持つ変数を内部等価点として推定する．このようなランダムシミュレーションを用いた内部等価点推定手法は，レジスタ転送レベルやゲートレベルの検証では一般的に用いられている手法である [12]．ここではシステムレベル設計記述などの高位記述に対して，ランダムシミュレーションを用いた内部等価点推定を行う問題を考える．

入力: 2 設計の ExSDG
出力: 変数の同値類集合

1: ExSDG を SSA 表現に変換
2: 外部入力ポートを発見
3: 各外部入力のパタンをランダムに生成
4: 各代入文の直後にモニタを挿入
5: シミュレータをコンパイル・実行
6: モニタ結果より各変数のシグネチャを生成
7: シグネチャに基づいて変数の同値類集合を求める

図 4　ランダムシミュレーションを用いた内部等価点推定の基本手続き

問題 4.1　*2* つの *ExSDG* 内の変数に対応する節の集合に対する同値類集合を求める．

ここで，同値類とはある節 r と等価である節の集合 $[r] \triangleq \{n \in N_1 \cap N_2 \mid r \equiv n\}$ である (N_1 と N_2 は各 ExSDG 内の節の集合)．

図 4 にランダムシミュレーションを用いた内部等価点推定の基本手続きを示す．以下，手続きの各段階について詳細に説明する．まず与えられた ExSDG を静的単一代入 (SSA) 表現に変換する．これにより関数内における各変数に対する代入は高々一回であることが保証される．そのため，より高い精度で変数の等価性を判定することが可能となる．次に必要があれば与えられた設計の外部入力ポートを発見し，各外部入力に対して決められた長さの入力列をランダムに生成する．本稿では，ある入力に対して計算結果の出力が生成される一連の実行を 1 サイクルと呼ぶ．また，与えられた設計はサイクル単位での実行が可能であると仮定する．これは Untimed モデルであれば N 回のハンドシェイク，Timed モデルであれば N 単位時間などといった方法で 1 サイクルを指定することによっても実現できる．

次に，各代入文の直後に代入された変数の値を表示するモニタ文を挿入する．SSA 表現に変換することによって各変数の代入は関数内で高々一回となっているが，あ

る関数は 1 サイクルで複数回実行される可能性があり，このため同変数に対して複数回の代入が行われる．これはあるモジュールが複数インスタンス化された場合やある関数を 1 サイクル内で複数回呼び出した場合に起こる．これらを区別するために，モジュールや関数の複製化を行う elaboration と呼ばれる処理を行う場合もあるが，検証の効率化のためにも複製を行わずに処理を行うことが望ましい．このため，本手法では各変数代入のモニタの際に変数名や変数値とともに実行時コンテクスト情報も出力する．このコンテクストは現在の関数がどのモジュールインスタンスのどの関数呼び出しから呼び出されたかを表している．このようにコンテクストによって区別される変数をコンテクスト付き変数と呼ぶ．このように実行時にコンテクスト情報を出力することによって静的にモジュールや関数の複製を行う必要がないため効率的である．この手法を用いることによって，各コンテクスト付き変数値は各サイクルにおいて高々一回しか出力されないことが保証される．出力されなかった場合にはそのコンテクスト付き変数は無効値を持つとする．

その後に，与えられた設計とパタン発生モジュールを接続してシミュレータをコンパイルし実行する．出力された各変数のモニタ結果から各変数のシグネチャを生成する．シグネチャとはコンテクスト付き変数の値の列であり，これはサイクル数の同じ長さを持つ．計算機実装ではシグネチャは変数値をある区切り文字で連結した文字列として表現される．同じシグネチャを持つ変数は等価である変数であり，異なるシグネチャを持つ変数は非等価である．ここで，等価であるとはシミュレーションを実行したサイクル内においてのみであることに注意されたい．また，ある変数が複数のコンテクストを持つ場合には，そのシグネチャ集合が等価であれば 2 変数は等価である．最後に同じシグネチャを持った変数を同値類とすることで，変数の同値類集合を求める．

4.2 推定内部等価点を用いた等価性検証

4.1 節で推定された内部等価点によってすべての変数の等価性が求まれば，3 節のルールベース等価性検証手法を適用することで変数名の変更などに対しても等価性を証明することが可能である．しかし現実的にはシミュレーションに基づく手法であるため，必ずしも変数の正しい等価性がすべて求まるわけではない．ここではそのような場合においても等価性を判定することが可能な手法を提案する．

変数の等価性が求まらない場合の一つとして，生成されたランダム入力列ではある条件が満たされないために実行されない部分がある場合が考えられる．この問題は与えられた条件を満たすような入力を生成する手法 [13] を用いることで対処可能である．よって以下では設計内のすべての変数の等価性の推定が可能であると仮定する．

入力: 2 設計の ExSDG G_1, G_2
推定内部等価点の同値類集合
出力: ExSDG 内の節の同値類集合

```
repeat
  for all G1 内の節 N1 do
    N1 の種類に対応した等価性規則を適用
    if 等価性を発見 then
      N1 と G2 内の節 N2 を同値類に追加
    else if 推定等価性に矛盾が発見 then
      矛盾した等価性を同値類から削除
      その等価性から導出された等価性を同値類から削除
    end if
  end for
until 最後の繰り返しで同値類集合に変更があった
```

図 5　推定内部等価点を用いたルールベース等価性検証の基本手続き

よって変数の正しい等価性がすべて求まらなかった場合には，いくつかの変数は非等価であるにもかかわらず等価と判定されているはずである．シミュレーション結果が異なる場合には非等価であるため，ある変数が等価であるにもかかわらず非等価と判定されることはないことに注意されたい．本稿で提案する等価性検証手法によって，多くの場合この誤った等価性は解決され，正しい等価性を判定することが可能である．図 5 に推定内部等価点を用いたルールベース等価性検証の基本手続きを示す．入力は検証対象の 2 設計に対応する ExSDG G_1 と G_2 と 4.1 節で推定した内部等価点の同値類集合である．検証手続きの各繰り返しにおいて，G_1 内の各節 N_1 に対して節の種類に応じた規則を適用し，もし N_1 と G_2 内の節 N_2 が等価であれば，これら 2 つの節を同値類とする．また，規則適用時に推定等価性に矛盾が発見された場合には対応する推定等価性およびそれから導出された等価性を同値類から削除する．これらを新たな等価性が発見されなくなるまで繰り返し，最終的に ExSDG のすべての節の等価性が判定されれば 2 設計は等価と判定される．

等価性検証手続き中に推定内部等価点に矛盾が発見される例を図 6 に示す．この例では，ランダム入力列に a=123 が含まれていない場合には (a) と (b) どちらの場合においても b と c が常に等価となるため，同値類は {ex5/b,ex5/c,ex6/b, ex6/c} となる．この同値類を用いて単純にルールベース等価性検証を行うと 2 設計は等価と誤って判定される．しかし，ex5 の S3 と ex6 の S3’ が等価となるのは，この同値類が互いに素な部分集合 {ex5/b, ex6/c} と {ex5/c,ex6/b} になる場合のみである．これは ex5 の S2 と ex6 の S2 が等価となる同値類の互いに素な部分集合 {ex5/b,ex6/b} と {ex5/c,ex6/c} に矛盾する．

```
int ex5(int a, int b) {
S1:    int c, d;
S2:    c = (a == 123) ?  a :  b;
S3:    d = b - c;
S4:    return d;
}
```

(a) 元の設計

```
int ex6(int a, int b) {
S1:    int c, d;
S2:    c = (a == 123) ?  a :  b;
S3':   d = c - b;
S4:    return d;
}
```

(b) 変更後の設計 (d = b - c から d = c - b に変更)

図 6　等価性検証手続き中に推定内部等価点の矛盾が発見される例

5. 提案手法の実装および評価結果

本稿で提案した内部等価点の推定によるルールベース等価性検証手法を高位設計記述の検証フレームワークである FLEC 上に C++言語で実装した．FLEC は SpecC によって記述された設計を入力として，抽象構文木を生成した後，依存解析を行って ExSDG を生成する．シミュレータの生成には SpecC リファレンスコンパイラ [14] を用いた．

例題として逆離散コサイン変換 (IDCT) の最適化前後の SpecC 設計記述を用いた．最適化前の例題では各行・各列の処理を逐次的に実行しているが，最適化によりそれらを並列に実行するように記述が変更されており，また変数名も変化している．従来手法では変数の対応が取れないため検証不能となり，等価性の判定ができなかった．提案手法ではランダムシミュレーションにより内部等価点の推定を行ったため変数の対応を求めることが可能となり，2 設計が等価との判定を行った．本実験では，ランダムシミュレーションを数サイクル行うことで変数の対応を求めることが可能であった．実行時間は数秒程度であり，また実行時間の大部分はシミュレータのコンパイル時間であった．シミュレータのコンパイル時間は一般的に設計規模に比例すること，またルールベース等価性検証問題もグラフマッチング問題と見なせることから，本手法は大規模設計に対しても適用可能であると思われる．

6. ま と め

本稿では，効率的な高位設計記述間の等価性検証手法の一つであるルールベース等価性検証手法の高精度化手法として，ランダムシミュレーションを用いた内部等価点推定手法を提案した．また，例題を用いた計算機実験では本提案手法が現実的な最適化を適用した場合の等価性を高速に判定することが可能であることを示した．

文　献

[1] R. Dömer, A. Gerstlauer, and D. Gajski. SpecC Language Reference Manual, Version 2.0. SpecC Technology Open Consortium, 2002.

[2] T. Grötker, S. Liao, G. Martin, and S. Swan. System Design with SystemC. Kluwer Academic Publishers, 2002.

[3] S. Abdi and D. Gajski, “A formalism for functionality preserving system level transformations,” in *Proc. Asia and South Pacific Design Automation Conference*, 2005.

[4] M. Fujita, “Equivalence checking between behavioral and rtl descriptions with virtual controllers and datapaths,” *ACM Transactions on Design Automation of Electronic Systems*, vol. 10, no. 4, pp. 610–626, Oct 2005.

[5] T. Matsumoto, H. Saito, M. Fujita, “Equivalence Checking of C Programs by Locally Performing Symbolic Simulation on Dependence Graphs,” in *Proc. of International Symposium on Quality Electronic Design*, pp.370–375, Mar. 2006.

[6] R. Dömer, A. Gerstlauer, J. Peng, D. Shin, L. Cai, H. Yu, S. Abdi, and D. Gajski, “System-on-Chip Environment: A SpecC-based Framework for Heterogeneous MPSoC Design,” *EURASIP Journal on Embedded Systems*, 2008.

[7] A. Gerstlauer, J. Peng, D. Shin, D. Gajski, A. Nakamura, D. Araki, and Y. Nishihara, “Specify-Explore-Refine (SER): From Specification To Implementation,” in *Proc. of ACM/IEEE Design Automation Conference*, pp. 586–591, Jun. 2008.

[8] Subash Shankar, 藤田昌宏, “ボトムアップ解析に基づく SpecC 記述間の等価性検証,” 電子情報通信学会研究会研究報告 Vol.106, No.32, pp.1-6, 2006 年 5 月.

[9] S. Shankar and M. Fujita, ”Rule-Based Approaches for Equivalence Checking of SpecC Programs,” in *Proc. ACM/IEEE International Conference on Formal Methods and Models for Codesign*, pp. 39–48, Jun. 2008.

[10] T. Nishihara, D. Ando, T. Matsumoto, and M. Fujita, “ExSDG : Unified Dependence Graph Representation of Hardware Design from System Level down to RTL for Formal Analysis and Verification,” in *Proc. International Workshop of Logic and Synthesis*, pp. 83–90, May 2007.

[11] S. Horwitz, T. Reps, and D. Binkley, “Interprocedural slicing using dependence graphs,” *ACM Transactions on Programming Languagesa nd Systems*, vol. 12, no. 1, pp.26–60, Jan. 1990.

[12] F. Krohm, A. Kuehlmann, and Arjen Mets, “The use of random simulation in formal verification,” in *Proc. International Conference on Computer Design*, pp. 371–376, 1996.

[13] I. Bongartz, A.R. Conn, N. I. M. Gould and Ph. L. Toint, “CUTE: Constrained and Unconstrained Testing Environment,” *ACM Transactions on Mathematical Software*, vol. 21, no. 1, pp. 123–160, 1995.

[14] SpecC Reference Compiler. `http://www.cecs.uci.edu/ specc/reference/`.