社団法人　情報処理学会　研究報告
IPSJ SIG Technical Report

2003－ＳＬＤＭ－110　(6)
2003／5／15

Ｃ言語でのハードウェア記述に対する効率的な等価性検証手法の提案

松本　剛史†　齋藤　寛††　藤田　昌宏†

† 東京大学大学院工学系研究科電子工学専攻　〒113-8656 東京都文京区本郷 7-3-1
†† 東京大学先端科学技術研究センター　〒153-8904 東京都目黒区駒場 4-6-1
E-mail: †{matsumoto,fujita}@cad.t.u-tokyo.ac.jp, ††hiroshi@hal.rcast.u-tokyo.ac.jp

あらまし　本研究では、Ｃ言語による 2 つのハードウェア記述の等価性を効率よく検証するための手法を提案する。本提案手法では、与えられた 2 つの記述より記述上の差異を特定し、これらの差異に関連のある部分のみを記述から抽出することによって、記号シミュレーションによる等価性検証の効率化を図っている。この研究で提案する手法により、より大規模な設計記述に対して、等価性検証を行うことが可能になる。
キーワード　等価性検証, Ｃ言語ベース設計, プログラムスライシング, 記号シミュレーション

An Equivalence Checking Method for C-Based Hardware Descriptions

Takeshi MATSUMOTO†, Hiroshi SAITO††, and Masahiro FUJITA†

† Department of Electronic Engineering, University of Tokyo　7-3-1 Hongo, Bunkyo-ku, Tokyo, 113-8656 Japan
†† Research Center for Advanced Science and Technology, University of Tokyo　4-6-1 Komaba, Meguro-ku, Tokyo, 153-8904 Japan
E-mail: †{matsumoto,fujita}@cad.t.u-tokyo.ac.jp, ††hiroshi@hal.rcast.u-tokyo.ac.jp

Abstract　In this paper, we propose a verification method to check the equivalence of two given C-based hardware descriptions efficiently. The main idea of the method is to extract away the descriptions as much as possible. For this purpose, textual differences between the descriptions are identified at first. Based on these differences, we extract the codes relevant to these differences by using a program slicer. These extracted codes are then verified by using symbolic simulation. Our proposed method was tested on several examples. Since the size of the descriptions was significantly reduced, our proposed method can verify large descriptions.
Key words　Equivalence Checking, C-based Design, Program Slicing, Symbolic Simulation

1. はじめに

VLSI の設計において、検証は設計の正しさを確かめるために必要不可欠な作業である。半導体の微細化技術が近年、飛躍的に進展したことに伴い、1 チップに搭載可能なトランジスタ数も著しく増大している。そのため、VLSI の設計も大規模化しており、検証作業はより困難になってきている。

VLSI の設計において、設計に誤りが発見された場合、設計過程のより上位のレベルに戻って修正作業をする必要が生じることがある。これは、設計の後期では、回路の詳細部分に至るまで設計を行わなければならず、設計記述の量が膨大になっているため、より抽象度の高いレベルで修正を行う方が容易な場合があるからである。このとき、設計過程の上位レベルで取り除くことが可能な設計の誤りが下位レベルで発見されると、時間とコストの両面で大きな損失となってしまう。このような修正作業によって生じる損失をできるだけ未然に防ぐため、設計過程の上位レベルにおける検証は極めて重要である。

近年、Ｃ言語をベースとした言語によるシステムの設計がさかんになってきている。これは、Ｃ言語ベースの言語を用いることによって、ソフトウェアとハードウェアを同じ言語で表現することができ、ソフトウェア- ハードウェア協調設計において大きな利点となるからである。さらに、Ｃ言語という広く一般的に使われているプログラム言語をベースにすることによって、新しい言語を習得する負担をより軽くすることができる。このようなＣ言語ベース言語には、SpecC [2]、SystemC [1] などがある。

ここでは、RTL (Resister Transfer Level) に至るまでのＣ言語ベースの上位設計と検証の流れとして図 1 を想定している。図 1 の前半部分では、ANSI-C のような標準的なＣ言語設計から、ハードウェア指向のＣ言語記述を導く。この部分におい

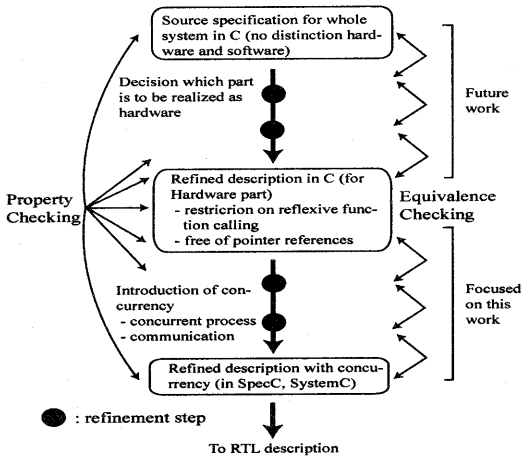

図 1　C言語ベース設計における検証

て、ハードウェアとソフトウェアの区分を決定する。この際、ハードウェアで実装される部分の記述については、ハードウェア記述用に標準的なC言語の機能の一部を制限したものを用いて記述する。制限する機能の例としては、ポインタの使用、関数の再帰呼び出し、などハードウェアでは実装されない機能が挙げられる。図 1 の流れの後半は、主にハードウェアで実現可能な並列性や他モジュールとの通信の導入を行う。ここでは、並列性が表現できる SpecC や SystemC のような言語を使うことになる。

本研究では、C言語を用いたハードウェア記述に対する等価性検証の手法を提案する。検証の対象としているハードウェア設計記述は、前述のように、C言語の機能の一部を制限して記述されているものである。図 1 の後半部分の等価性検証においては、並列性を扱えることが必要である。本研究では、並列性は扱わないが、将来、本研究で提案する手法に並列プロセスの検証を加えることにより、図 1 の後半部分の検証、つまり SpecC や SystemC の記述に対する検証、が可能になる。

この図 1 のような設計手法では、設計記述に少しずつ変更を加えていくため、 1 回の変更前後における設計記述は多くの場合、差異がそれほど大きくない。そこで、本研究では記述上の差異とプログラムスライシング手法 [10] を用いて、検証の対象となる記述量を小さくした効率的な等価性検証の手法を提案する。さらに、検証自体は記号シミュレーション [8] という形式的な手法によって行うこととした。一般に、形式的検証では、検証漏れがないという利点があるが、計算が複雑になり、設計規模が大きくなると計算量が膨大になってしまい、検証が不可能になってしまうことがある。そのため、できるだけ検証範囲を小さくすることは、より大規模な設計を扱うためには不可欠である。

2. 関連する研究

本研究がC言語による動作レベルから RTL レベルに至る設計過程における等価性検証を目指しているのに対し、文献 [6] では、C/C++ 言語と Verilog 言語の等価性検証を RTL レベルで実現している。文献 [3] では、本研究と同様に、記号シミュレーションを用いたC言語記述に対する等価性検証の手法を提案している。文献 [4] では、C記述と Verilog 記述の間の behavioral cocsistency を SAT を用いて検証する手法が紹介されている。この SAT を利用した検証は、本研究で用いる記号シミュレーションの代わりに利用することが可能である。

これらの研究と比較し、本研究では、記述上の差異に関係のある部分のみをプログラムスライシングを用いて抽出し、検証を行うことになっている。このため、本研究が提案する手法では、検証時間を大幅に短縮することができると考えられる。このことは、設計規模の増大によって検証時間が飛躍的に長くなってしまう形式的検証にとって、非常に重要な利点である。また、文献 [4] で提案されている手法と組み合わせて検証を行うことも可能である。

形式的検証のうちモデルチェッキングに VHDL 記述を対象としたプログラムスライシングを用いる研究もなされている [7]。プログラムスライシングでは、 1 つの記述から、ある変数にとって機能的に等価な部分のみを抽出することができるため、モデルチェッキングで検証する状態空間を大幅に縮小することができる。本研究では、このプログラムスライシング手法を等価性検証において利用している。

3. 基礎概念

3.1 記号シミュレーション

本研究では、等価性検証は記号シミュレーションという形式的な手法によって行われる。記号シミュレーションでは、変数をビットベクターのような具体的な数値としてではなく、あくまでも記号として扱う。そのため、0 、1 といった具体的な数値による検証よりも効率的な検証が可能である [8] 。

記号シミュレーションでは、等価であると判明した変数、式を要素とする EqvClass という集合を用意し、そこに代入関係から得られる等価な関係をまとめていく。ここで、記号シミュレーションに基づく等価性検証におけるいくつかの規則を次に示す。

- 記号シミュレーションはプログラムの先頭 (main 関数) から開始する
- 変数に対する代入があった場合、その左辺と右辺をともに同じ EqvClass の要素とする
- 変数や式は、等価な変数、式によって置き換えることができる
- 関数名と引数が等価である 2 つの関数は、uninterpreted function として等価であるとする
- 既に存在する全ての EqvClass のどれとも等価でない変数が現れた場合、その変数を要素とする新たな EqvClass を作る
- 条件分岐が生じた場合には、その時点で実行される可能性のある分岐についてのみシミュレーションする。条件判定ができない場合は、全ての分岐をシミュレーションする
- 記号シミュレーションの終了時に、検証の対象となる 2

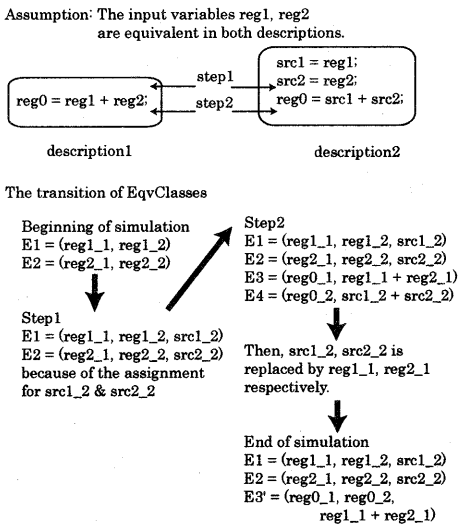

図 2　記号シミュレーションの例

変数が同じ EqvClass に属していれば、検証結果は等価である

図 2 は、記号シミュレーションを用いた等価性検証の例である。ここでは、両記述中の変数 $reg0$ の等価性を検証する。また、変数 $reg1$ と変数 $reg2$ は入力として与えられる値であり、両記述で等価であると仮定している (E1 と E2 に相当)。さらに、一般的に記述 1 中の変数 v を v_1、同様に記述 2 中のものを v_2 と表すことにしている。

まず、図中の step1 までに、記述 2 での変数 $src1$ と $src2$ に対する代入がシミュレーションされる。この代入により、$src1\_2$ と $src2\_2$ はそれぞれ E1 と E2 に付け加えられる。同様に、代入関係に基づいて step2 に至るまでに、E3 と E4 の 2 つの EqvCalss が新たに作られる。ここで、$src1\_2$ と $reg1\_1$、$src2\_2$ と $reg2\_1$ はそれぞれ E1、E2 より等価である。そこで、E3 と E4 の 2 つの EqvClass は 1 つにまとめることができる。この結果、両記述中の変数 $reg0$ の等価性が証明された。

記号シミュレーションでは、基本的に等価な変数 (式) の置き換えのみから等価性を導くため、a + a と 2 * a のような例では等価性を示すことができない。そこで、これらの表現の等価性を検証するために SVC (Stanford Validity Checker) [9] を補完的に利用する。

3.2　プログラムスライシング

プログラムスライシングは、プログラム中からある変数に関係のある部分のみを抜き出す手法である [10] 。この手法を用いることによって、検証したい変数に関係のある部分のみを記述中から抜き出すことができるため、検証する記述量を減らすことができる。プログラムスライシングには、バックワードスライシングとフォワードスライシングの 2 通りがある。

バックワードスライシングは、プログラム中のある変数に影響を与えている部分を全て抜き出す手法である。バックワードスライシングの例を図 3 に示す。プログラムスライシングでは、スライシングの始点としようとする部分を slicing criterion と

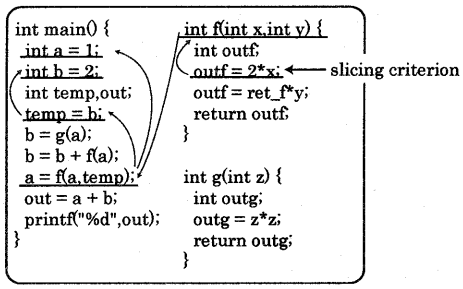

図 3　バックワードスライシングの例

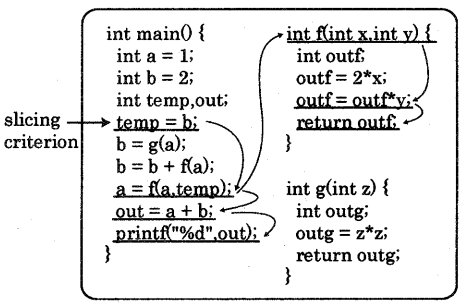

図 4　フォワードスライシングの例

して指定する。この例では、$outf = 2 * x;$ が slicing criterion である。この変数 x は、関数 f の引数 x の影響を受けるため、図 3 で示すように関数の宣言部分が抜き出される。さらに、この関数宣言は、呼出側である $a = f(a, temp);$ の影響を受けているため、この部分も抜き出される。このようにして、図 3 で下線を引いて示した部分が抽出される。

一方、フォワードスライシングは、ある変数に影響を受けている部分を全て抜き出す手法である。図 4 は、フォワードスライシングの例である。ここでは slicing criterion として、$temp = b;$ をスライシングの始点とした。このとき、$a = f(a, temp);$ は変数 $temp$ の影響を受けるため、スライシングにより抜き出される。同様にして、図 4 において下線を引いて示した全ての部分が、影響を受ける部分として抽出される。

4.　検証の流れ

4.1　全体の流れ

本研究で提案する 2 つのC言語を用いたハードウェア設計記述に対する等価性検証の流れは、図 5 に示す通りである。検証には既存の手法である記号シミュレーションを用い、検証範囲を減少させるために、記述上の差異とプログラムスライシングを利用している。以降の各節では、図 5 の流れを詳細に説明する。

4.2　前　処　理

検証に先立ち、マクロ展開やコメント文の削除のような前処理が行われる。また、この段階で、$printf()$ や $getchar()$ のような入出力に関する関数を取り除く。これは、これらの入出力関数はC言語レベルでは使われるが、ハードウェアとして実装されないからである。

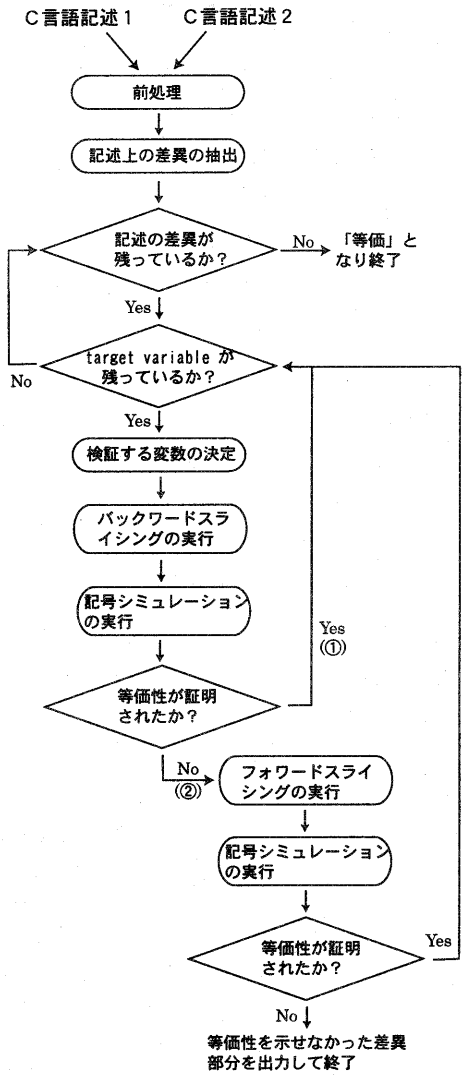

図 5　等価性検証の流れ

さらに、両記述について DG (Dependency Graph) を構築する。DG はコントロール、データのフローを表すものである [4] 。本研究では、検証をプログラムが実行される順番に沿って行うために DG の情報が必要である。通常、プログラムスライシングツールは DG のようなグラフを基にして解析を行っているので、ここではそれを使うこととする。

4.3　記述上の差異の抽出

次に、両記述間のテキスト上の違いを抽出する。ここでは、UNIX の標準コマンドである diff コマンドを用いる。 diff コマンドでは、与えられた 2 つの記述から純粋にテキストとして異なる部分を取り出すことができる。そのため、diff を実行するために、 2 つの記述形式を統一しておく必要がある。記述間の形式が統一されていないと、検証に無関係な「違い」が多く抽出され、検証手順が複雑になってしまう。統一すべき記述形式には、スペースの挿入などが挙げられる。さらに、得られた

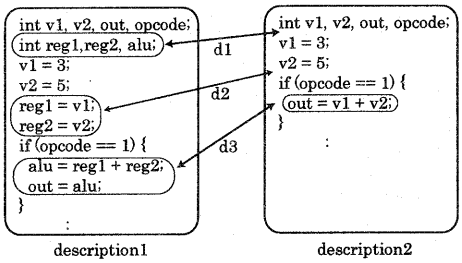

図 6　記述間の差異の抽出の例

記述の違いを DG に従って、実行される順番に並べる。等価性検証は、得られた差異の 1 つ 1 つに対して、並べられた順番に沿って行われる。

図 6 は、記述上の差異を抽出し、実行順に番号を付けた例である。この例では、 diff の実行によって 3 点の差異が抽出され、それらに d1, d2, d3 と実行される順番に番号を付けている。

4.4　記号シミュレーション、プログラムスライシングを用いた検証

検証は、既に得られている記述の差異のそれぞれに対して行われる。すべての差異に対する検証結果が等価であれば、記述全体としても等価であると証明されたことになる。そうでない場合には、検証結果が等価でなかった差異部分などの情報を出し、検証結果は非等価となる。

4.4.1　検証する変数の決定

1 つの記述の差異中には、複数の代入式が含まれていることもあり得る。そのような場合には、等価性検証の対象となる変数、 target variable を決定しなければならない。 target variable は、両記述で宣言がされており、その記述の差異中で代入をなされている変数である。両記述で宣言されている変数 v に対する代入が片方の記述でしか行われない場合、代入のない方の記述に $v = v;$ のような式を挿入することによって、検証ができるようにする。

図 6 の例では、 d1 と d2 には target variable は存在しない。これは、 d1 では代入が行われていないため、d2 では代入を受けている変数 $reg1$ 、 $reg2$ が片方の記述でしか定義されていないためである。一方、 d3 中の変数 out は、両記述で宣言されており、 d3 内で代入を受けているので検証の対象となる。

4.4.2　バックワードスライシングの実行

検証の対象となる target variable のペアが決定した後に、バックワードスライシングを実行する。これによって、記号シミュレーションで扱わなければならない記述量を削減することができる。検証すべき target variable のペアが、それぞれスライシングの起点となる slicing criterion になる。

4.4.3　記号シミュレーションの実行

バックワードスライシングによって、検証する変数に影響を与えている部分が抜き出された後に、それらの抽出部分に対して、記号シミュレーションを用いて等価性を検証する。この検

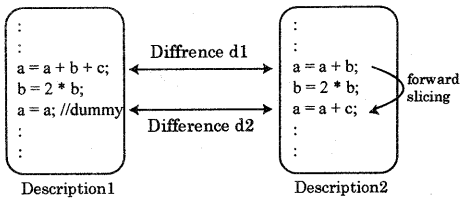

図 7　フォワードスライシングを用いた検証の例

証の結果に応じて、図 5 に示したように、場合分けが生じる。

- 検証した変数の等価性が示せた場合 (図中の ① に相当)

等価性を示すことができた場合には、次の target variable の検証を行う。もし、記述の差異中に未検証の target variable が残されていない場合には、次の記述の差異へ移る。

- 検証した変数の等価性が示せなかった場合 (図中の ② に相当)

等価性を示すことに失敗した場合には、その変数がプログラムの下流に与える影響も考慮して全体の等価性を判断しなければならない。この場合、検証に失敗した変数からフォワードスライシングを実行して、その抽出部分に対して、記号シミュレーションを行う。

このようなフォワードスライシングを用いた検証によって、等価性が示せる場合の例が図 7 である。この例では、 d1 と d2 の 2 つの記述の差異がある。そこで、まず差異 d1 についてバックワードスライシングを用いて、検証をすると結果は等価ではない。そこで、さらに検証を行うために、 d1 で代入を受けている変数 a からフォワードスライシングを行う。この部分に対して記号シミュレーションを行うと、記述 1 の $a = a$;、記述 2 の $a = a + c$; において、両記述の全ての変数の組が等価になる。なお、この例では、差異 d2 については、d1 の検証時に既に検証されたので、新たに検証されることはない。

5. 実験結果

本章では、提案した検証手法によるいくつかの実験例を示す。提案手法の実装はまだ完全ではないため、一部は人手で行った。実験は、ある設計記述に対して、何らかの変更を加えた記述を用意し、それらの記述間での等価性を検証した。なお、実験では、プログラムスライシングを行うツールとして、GrammaTech 社の CodeSurfer [5] を用いた。

5.1 実験 1：ハードウェア資源のマッピング

ハードウェアを実装する過程で、演算器やレジスタといった資源をマッピングすることが必要である。ここでは、変更前の記述として、 RISC プロセッサのシミュレーションのためのC言語記述を用いた。その記述に ALU のマッピングを行った記述を変更後の記述とした。変更前後で記述の差異は 8 箇所であった。そのうちのいくつかを図 8 に示す。この図からも分かるように、マッピングに伴って、新たに変数が宣言されたり、新たな代入が行われたりしている。

検証は、全部で 6 組の target variable に対して行われた。この 6 回の検証は、全て等価であると正しく証明された。検

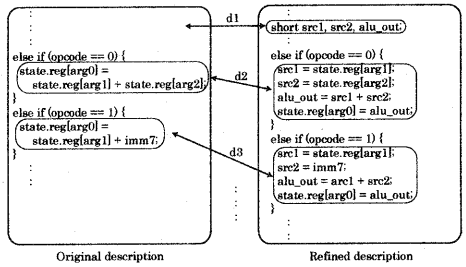

図 8　ALU のマッピング前後での記述の差異

表 1　実験 1 の結果

	Total lines	Extracted lines	Reduction Ratio
Original	83	57	31%
Refined	104	74	29%
8 differences, 6 target variables			

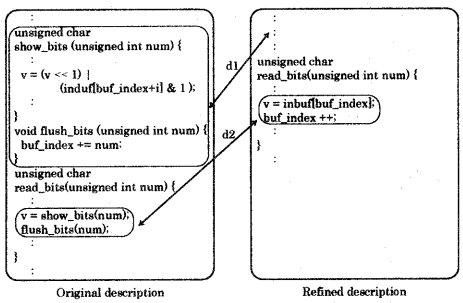

図 9　インライン展開の前後における差異

証範囲については、表 1 のようになった。この表から、検証範囲の全記述量に対する減少率は、それぞれ、変更前の記述では 31% 、変更後の記述では 29% になっていることが分かる。

5.2 実験 2：関数の展開による最適化

ハードウェアの実装時には、さまざまな最適化が行われる。そのうちの 1 つとして、関数のインライン展開による変更を加え、変更前後の記述の等価性を検証した。ここでは、変更前の設計記述として、ハフマンデコーダのC記述モデルを取り上げた。この設計記述に対して、図 9 に示すようなインライン展開を行った。変更前の記述にある 2 つの関数 $show\_bits(num)$ と $flash\_bits(num)$ が、変更後の記述では、インライン展開されている。この結果、図中の d1 から分かるように、変更後の記述では関数呼出や関数内での変数の宣言がなくなっている。また、変更前の記述の d1 において記述されていた処理が、変更後の記述の d2 において記述されている。

検証の結果、2 つの記述間での等価性が示された。この実験では、記述の差異は 2 箇所あり、検証すべき target variable は 2 組であった。記述間の差異が、プログラムの初期に存在したため、検証範囲はもとの記述に対して大幅に減少した。その結果が、表 2 である。この表から、検証範囲の全体の記述量に対する減少率は、変更前の記述で 58% 、変更後の記述で 73%になったことが確認できる。

表 2　実験 2 の結果

	Total lines	Extracted lines	Reduction Ratio
Original	49	21	58%
Refined	41	11	73%
2 differences, 2 target variables			

表 3　実験 3 の結果

	Total lines	Extracted lines	Reduction Ratio
Original	632	131	79%
Refined	630	129	80%
6 differences, 6 target variables			

5.3　実験 3：MAXSAT を題材にした実験

この実験では、比較的大きな設計記述に対して提案手法を適用するため、MAXSAT (MAXimum SATisfiability) を計算するためのC言語記述に変更を加え、その変更前後での等価性を検証した。MAXSAT は、x1, x2, ... , xn の n 個のブール代数を用いて表現された m 個の項 (C1, C2, ... , Cm) と実数 w1, w2, ... , wm との線形結合 (C1 * w1 + C2 * w2 + ... + Cm * wm) の最大値を与えるような組合せ (x1, x2, ... , xn) を求める問題である [11] 。

この実験では、これまでのようにハードウェアへの実装を目的とした変更というよりはむしろ、規模の大きな記述に対する検証の効率化を実験するために、等価性が保たれるように 6 箇所の部分を意図的に変更した。検証結果は、表 3 のようになった。全記述に対して検証範囲は、変更前の記述、変更後の記述の両方でおよそ 20% ほどになり、大幅に縮小されたことが確認された。

6.　結論と今後の課題

本研究では、C言語によるハードウェア設計記述に対する効率的な等価性検証の手法を提案した。検証の手法そのものは、記号シミュレーションという既存の手法であるが、記述間の差異とプログラムスライシング手法を利用して、検証範囲を小さくすることを行った。このため、提案した検証手法を用いることによって、従来よりもより大きな設計記述を扱うことが可能となる。また、このような事実は、いくつかの例題に提案手法を適用することによって確認された。

今後の課題としては、第一に、この手法に沿った実装を完成させることが挙げられる。さらに、ハードウェアへと実装を進める際に現れる並列プロセスを含めた等価性検証の手法について研究を深める必要がある。このとき、標準的なC言語では、並列プロセスは表現できないため、 SpecC のような言語の等価性検証を考えていくことになる。そのためには、記号シミュレーション、プログラムスライシングなどの要素技術が並列性を扱えるように拡張していかなければならない。本研究で提案した検証手法は、前掲の図 1 の後半部分の等価性検証を扱うことを目指すものであった。今後は、図の前半部分、機能レベルを表現した標準的なC言語記述からハードウェア向けの制限されたC言語に至るまでの過程における等価性検証についても研究をしていくつもりである。

文　献

[1] SystemC: http://www.systemc.org/

[2] D. Gajski, J. Zhu, R. Doemer, A. Gerstlauer, and S. Zhao, "SpecC: Specification Language and Methodology", *Kluwer Academic Publisher*, Mar. 2000.

[3] H. Saito, T. Ogawa, T. Sakunkonchak, M. fujita, T. Nanya "An Equivalence Checking Methodology for Hardware Oriented C-based Specifications", *Proc. IEEE International High Level Design Varidation and Test Workshop*, Cannes, France, Oct. 2002.

[4] E. M. Clarke and D. Kroening, "Hardware Verification Using ANSI-C Programs as a Reference", *Proc. Asia South Pacific Design Automation Conference*, pp.308–311, 2003.

[5] CodeSurfer: http://www.grammatech.com/products/codesurfer/

[6] L. Semeria, A. Seawright, R. Mehra, D. Ng, A. Ekanayake, B. Pangrle, "C-Based RTL Methodology for Designing and Verifying a Multi-Threaded Processer", *Proc. Design Automation Conference*, pp.123–128, 2002.

[7] E. M. Clarke, M. Fujita, S. P. Rajan, T. Reps, S. Shankar, and T. Teitelbaum, "Program Slicing for VHDL", *Proc. Conference on Correct Hardware Design and Verification Methods*, pp.298–312, 1999.

[8] G. Ritter, "Formal Sequential Equivalence Checking of Digital Systems by Symbolic Simulation", PhD thesis, Darmastadt University of Technology and Universite Joseph Fourier, 2000.

[9] C. W. Barrett, D. L. Dill, J. R. Levitt, "Validity Checking for Combinations of Theories with Equality", *Proc. Internatioal Conference on Formal Methods in Computer-Aided Design*, pp.187–201, Nov. 1996

[10] M. Weiser, "Program slices: Formal, psychological, and practical investigations of an automatic program abstraction", PhD thesis, University of Michigan, 1979.

[11] M. G. C. Resende, L. S. Pitsoulis, P. M. Pardalos, "Fortran Subroutines for Computing Approximate Solutions of Weighted MAXSAT Problems using GRASP", *tech. rep.*, AT&T Research, Murray Hill, NJ, 1998.