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

2005－ＳＬＤＭ－120（10）
2005／5／20

依存グラフを用いた局所的な記号シミュレーションによるC言語記述に対する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
†† 会津大学コンピュータ理工学部コンピュータハードウェア学科　〒965-8580 福島県会津若松市一箕町鶴賀
††† 東京大学大規模集積システム設計教育研究センター　〒113-0032 東京都文京区弥生 2-11-16
E-mail: †matsumoto@cad.t.u-tokyo.ac.jp, ††hiroshis@u-aizu.ac.jp, †††fujita@ee.t.u-tokyo.ac.jp

あらまし　本稿では、記号シミュレーションに基づくC言語記述の論理等価性検証のより効率的な手法を提案する。検証の対象は、C言語によって記述された上位設計のハードウェア部分である。一般的に、大規模な設計記述全体に対する記号シミュレーションは、大きな計算量を要するため、実現することが難しい。そこで、本研究では、設計記述間の差異が比較的小さい場合に、その差異の等価性を局所的に検証することを繰り返して、全体の等価性を導く手法を提案する。差異の等価性が示せない場合、提案手法では、依存グラフ上で検証範囲を拡大し検証を繰り返す。例題を用いた評価によって、従来手法に比べて小さな検証範囲で検証が可能であることを示した。

キーワード　Cベース設計、等価性検証、記号シミュレーション、依存グラフ

Equivalence Checking for C Description by Local Symbolic Simulation Using Dependence Graphs

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

† Dept. of Electronics Engineering, University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo, 113-8656 Japan
†† Dept. of Computer Hardware, The University of Aizu
Tsuruga, Ikki-machi, Aizu-Wakamatsu, Fukushima, 965-8580 Japan
††† VLSI Design and Education Center, University of Tokyo
2-11-16 Yayoi, Bunkyo-ku, Tokyo, 113-0032 Japan
E-mail: †matsumoto@cad.t.u-tokyo.ac.jp, ††hiroshis@u-aizu.ac.jp, †††fujita@ee.t.u-tokyo.ac.jp

Abstract　In this paper, we propose an efficient equivalence checking method for C descriptions based on symbolic simulation. The target of our method is high level hardware designs written in C language. Generally, symbolic simulation for whole large designs takes very long time so that it cannot be applied in practice. Our proposed method locally verifies each difference between the given descriptions and proves the equivalence of the whole descriptions from the results of local equivalence checking. In the method, if a difference cannot be proved to be equivalent, the verification areas are extended using dependence graphs and verified again. By experimental results, we have shown that our method could verify designs by analyzing smaller areas than the previous method.

Key words　C-based Design, Equivalence Checking, Symbolic Simulation, Dependence Graph

1. はじめに

近年のVLSIの大規模化・複雑化やシステムLSI設計などにおけるソフトウェアとの協調設計の必要性から、従来、設計が始められたいたRTL (Register Transfer Level) よりも抽象度の高いシステムレベルや動作レベルでの設計が広く行われるようになってきている。これは、システムレベルのような上位設計では、設計者が扱う記述量が少ないこと、設計資産の再利用が容易であること、ソフトウェアとの協調設計が可能であることなどによる設計生産性の向上が期待されるためである。その

ためには、システムレベルでの設計を支援する CAD 技術が必要不可欠であり、動作合成や HW/SW 協調シミュレーションによる検証の研究・開発がさかんに行われている。

システムレベルで生じた動作に関する設計誤りを残したまま設計を進めてしまうと、後に、システムレベルに戻って設計をやり直す必要が生じ、大きな損失となる可能性がある。そのため、システムレベルでの検証では、動作に関する設計誤りを可能な限り発見し、修正することが求められる。このような検証では、テストパタンに依存せず網羅的に検証が可能な形式的検証が有効であると考えられる。そこで、本稿では、C 言語で記述された上位設計を対象とした形式的な等価性検証手法を扱う。

提案する検証手法は、記号シミュレーションと呼ばれる形式的な手法に基づいて行われる。しかし、大規模な設計記述全体に対して、記号シミュレーションを行うことは、計算時間が非常に長くなるため、現実的には難しい。そこで、本稿では、図 1 に示すような、多くの回数にわたって設計記述を人手による (もしくは、対話的な) 詳細化・変換・最適化を行う上位設計フローを想定し、比較的差異の小さな 2 つの設計記述間の等価性を効率的に検証する手法を提案する。この設計フローでは、C 言語 (もしくは、SpecC 言語 [7] や SystemC 言語 [8] のような C ベース設計言語) を用いて、ハードウェア部分については動作合成ツールに入力するまで、ソフトウェア部分についてはコンパイラに入力するまでの設計過程を、主に人手によって行う。そのため、等価性を検証することによって設計誤りの挿入を防ぐことが重要となる。

提案する検証手法の特徴は、検証する 2 つの記述間の差異の等価性を記号シミュレーションによって局所的に検証していき、それぞれの差異の等価性から全体の等価性を導く点にある。これによって、記述全体を記号シミュレーションすることなく、検証を行うことが可能である。しかし、この提案手法では、基本的には、検証している差異が等価であることが示されるまで、検証範囲を拡大して検証を続けるため、ある差異が等価でない場合には、検証範囲が非常に大きくなってしまう可能性がある。そこで、特定の変換や最適化を対象として、等価でない場合であっても、小さな検証範囲で検証が可能な手法の検討も行う。

本稿の構成は以下の通りである。第 2 節で本研究に関連する研究、第 3 節で本研究で用いる要素技術についてそれぞれ述べる。第 4 節で提案する検証手法の詳細を述べ、第 5 節で予備的な実験結果を示す。最後に、第 6 節で結論と今後の課題を述べる。

2. 関連研究

等価性検証においては、2 つの設計記述間で対応を取ることにより内部等価点の候補を求め、それらの等価性を調べていくことによって効率化を図ることが大規模設計への適用のために重要となる。文献 [2] では、C 言語のサブセットによって記述された RTL 記述と、そこから得られる論理合成用の Verilog 記述との間の等価性検証手法が提案されている。また、文献 [1] では、C ライクな記述言語による動作記述と動作合成後の RTL との等価性検証を行っている。これらの研究では、記述間の類似性が必要とされたり、合成環境からの情報を利用したりして、設計間の対応を取っている。一方、本研究では、先に示した図 1 のような設計フローを想定し、記述間のテキストの差異が特定可能な 2 つの記述を対象とするため、その差異を基にして対応を取ることができる。加えて、記述変更によって生じる差異が小さいことが見込まれる特定の最適化や変換に対しても、対応を取ることが可能である。

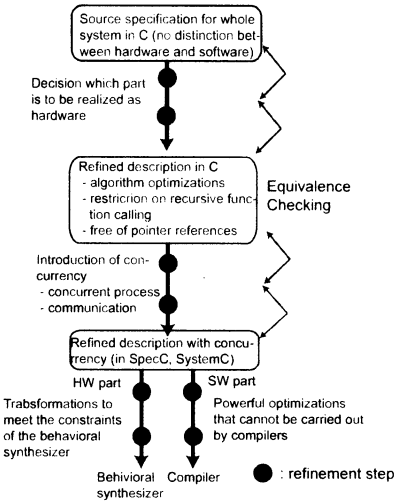


図 1　C 言語を用いたシステムレベル設計フロー

本研究と同様に、記号シミュレーションに基づいた等価性検証手法としては、文献 [3] で提案されている手法がある。この文献では、記述間の差異を利用して、等価性検証中に行われる不必要な等価性判定ルーチンの実行を減少させることによって効率化された手法が提案されている。しかし、関数単位で検証が行われるため、検証する関数全体を記号シミュレーションする必要があり、その中に多くの実行パスが存在する場合には、記号シミュレーションされる記述量が指数的に増加してしまうという問題がある。これに対して、本稿では、記述間の差異の等価性を局所的に検証することを重ねることによって、全体の等価性を示す手法を提案する。この手法では、それぞれの差異が等価である場合には、非常に小さな検証範囲で短時間で検証を行うことが可能であり、等価でない差異が含まれている場合であっても、記号シミュレーションされる記述量は、全実行パスよりは少なくなる。

3. 提案手法で用いる要素技術

3.1 記号シミュレーションによる等価性検証

記号シミュレーションとは、記述中の変数の値や演算の意味を解釈せずに行われるシミュレーション手法である。1 つの記号によって、その変数の全ての値を表すことができるため、テストパタンによらない網羅的な解析が可能である。記号シミュレーションに基づく等価性検証は、文献 [4] で RTL 設計記述用に提案されている。ここでは、記号シミュレーションによって、代入関係や置換によって、等価であると分かった変数・式

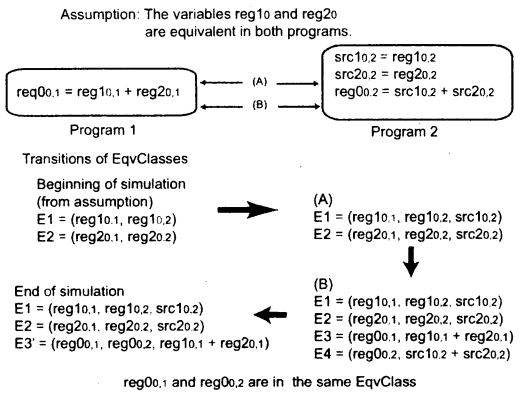


図 2　記号シミュレーションによる等価性検証

を Equivalence Class (EqvClass) と呼ばれる集合に集めることを繰り返していく。そのため、両記述で対応する入力変数を等価であると仮定して、記号シミュレーションを進め、等価性を示したい変数が、最終的に同じ EqvClass に属していれば、その変数の等価性が証明されたことになる。

本研究では、この文献 [4] の手法をもとに、C 言語記述用の記号シミュレータを実装し、等価性検証を行った。C 言語記述での等価性検証の例を図 2 に示す。この例では、両記述の変数 $reg1$ と $reg2$ が等価であると仮定して、変数 $reg0$ の等価性を検証している。

記号シミュレーションでは演算の意味を解釈しないため、図 2 の記述中の加算が乗算や除算であっても、全く同じ計算量で等価であることを示せる。しかし、演算の意味を解釈しないために、$a*(b+c)$ と $a*b+a*c$ のような等価性を示すことができない。そこで、本研究では、Cooperating Validity Checker (CVC) という、線形の範囲の算術演算を含んだ式の論理妥当性を判定するツールを必要に応じて使用し、等価性判定性能を向上させている。

3.2　システム依存グラフ

本研究では、ある差異に対する局所的な検証範囲を決定したり、その検証範囲を拡大したりする際に、依存関係に従った操作が行われる。そこで、提案手法では、それらの操作を依存グラフ上で行うことにした。依存グラフには、いくつかの種類があるが、本研究では、文献 [6] で提案されているシステム依存グラフ (SDG: System Dependence Graph) を用いている。SDG では、関数間に及ぶ依存関係を正確に解析することができるという利点がある。本研究では、商用のプログラム解析ツールである CodeSurfer [9] で作られた SDG を用いている。図 3 に C 言語記述とその SDG の例を示す。SDG には、データ依存エッジと制御依存エッジがある。前者は、変数を定義しているノードから、それを使用しているノードに対して引かれ、後者は条件分岐や関数の開始点に相当するノードから、その制御を受けているノードに対して引かれる。

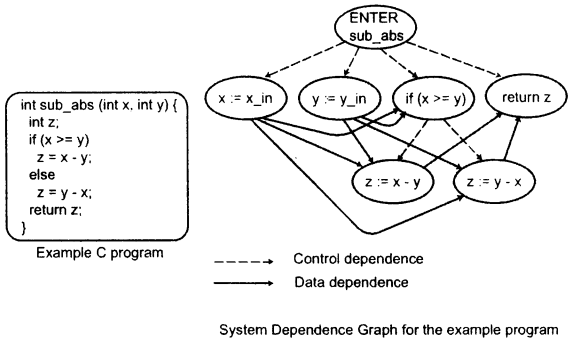


図 3　C 言語記述とそのシステム依存グラフ

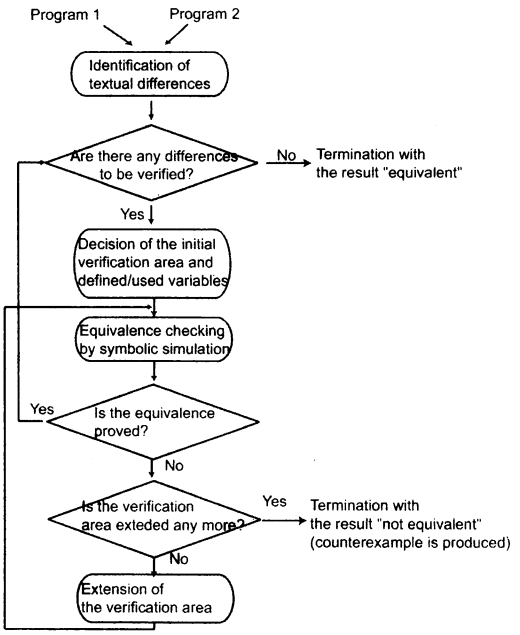


図 4　検証の流れ

4.　提案する検証手法

検証手法の全体の流れを図 4 に示す。入力として、2 つの C 言語記述と入力変数、出力変数が与えられる。それらに従って、記号シミュレーションによって等価性を検証し、「等価」もしくは「非等価」の結果を出力する。「非等価」の場合には、EqvClass の集合から成る反例も併せて出力する。提案手法では、単純に記述全体を記号シミュレーションするのではなく、続く節で述べるように、記述間の差異を特定し、その差異を局所的に検証することで効率化を図っている。全ての差異に対して、等価であることが示された場合のみ検証結果は「等価」となる。一方、1 つでも等価でない差異があった場合には、検証結果は「非等価」となる。

4.1　検証する記述に対する制限

本研究では、検証する C 言語記述が以下の制限を満たしてい

る必要がある。

- ポインタの使用がない (もしくは、全てのポインタはどの変数を指すか解析されている)
- 動的なメモリ確保がない
- ループは有限回数だけ展開されている
- 再帰関数がない (もしくは、有限回数だけ展開されている)

これらの制限は、記号シミュレーションを可能にするために必要である。これらの制限を満たさない文が検証範囲に含まれた場合、そこで検証を終了する。ただし、検証範囲に含まれない限りにおいては、ポインタやループ構造が含まれていても問題はない。

4.2 記述間の差異の特定

記号シミュレーションを開始する前に、両記述間の差異を特定する。これは、UNIX の標準コマンド diff によって行うことができる。先にも述べた通り、本研究では、検証の対象としている 2 つの記述間の差異は比較的小さく、diff コマンドによって十分に差異が特定可能であることを仮定している。ここで、特定された差異の 1 つ 1 つについて、局所的に記号シミュレーションを行い、等価性を検証していく。

4.3 初期検証

ここでは、ある差異に対する 1 回目の検証について述べる。この検証の検証範囲は、それぞれの記述から得られた差異に相当する SDG ノードである。ここで、この局所的な検証範囲に対して等価性検証を行うために、次のように、局所的な入出力変数を決定する。

- **局所的な検証範囲に対する入力変数** 差異中の代入文、条件評価において使用されている変数。SDG 上で、検証範囲外から範囲内に入ってくるデータ依存エッジに対応
- **局所的な検証範囲に対する出力変数** 差異中の代入で代入を受けて定義されている変数。SDG 上で、検証範囲内から範囲外に出ていくデータ依存エッジに対応

ここで、この局所的な検証範囲に対する出力変数について、変数名に基づいて対応を取り、対応が取れた変数について、その 1 組の変数が等価であるかどうかを検証する。このとき、次のどちらかの条件を満たす場合には、その局所的な検証範囲に対する入力変数が等価であると仮定して検証してよい。

- それまでの記号シミュレーションによって等価であることが証明されている
- どの記述の差異にも含まれず、どの差異からも依存を受けていない

初期検証の場合には、全ての文が差異に含まれているため、2 つ目の条件が満たされることはないが、2 回目以降の検証では、この条件を満たす文が検証範囲に加えられることは十分にあり得る。

ここで、全ての局所的な検証範囲に対する出力変数の等価性が示されれば、この差異に対する検証は終了する。しかし、1 つでも等価性が示せない出力変数がある場合には、検証範囲を拡大して検証を続ける。

4.4 検証範囲の拡大

初期検証によって、等価であることが示せなかった変数がある場合、検証範囲を拡大して再び検証を行う。これは、以下のような 2 つの場合に、初期検証の結果のみによって、全体として「非等価」という結果を出すことは false-negative な結果 (実際は等価であるのに、検証結果が非等価という結果を出すこと) になるためである。

- 検証範囲を拡大することによって、局所的な検証範囲の出力変数が等価であることが分かる
- 局所的な検証範囲の出力変数は等価ではないが、記述全体の出力変数がその影響を受けない

このような場合に、false-negative な検証結果を出すことを防ぐために、提案手法では、次の 2 つの依存関係に基づく検証範囲拡大手法を用いる。

- **後方への拡大** 検証範囲のある入力変数に依存を及ぼしている全ての文を検証範囲に加える。SDG 上では、入力変数に対応するエッジの元のノードを検証範囲に加えることに相当
- **前方への拡大** 検証範囲のある出力変数が依存を与えている全ての文を検証範囲に加える。SDG 上では、出力変数に対応するエッジの先のノードを検証範囲に加えることに相当

一般的に、局所的な検証範囲には、複数の入力変数・出力変数が存在するため、これらの拡大手法をどの変数に対して適用するのかの選択が重要になる。なお、検証範囲が拡大された場合には、その新たな検証範囲に対する入出力変数を定義し、それに従って等価性検証を行う。なお、検証範囲の拡大は、記述全体の入出力変数かプログラムの始点・終点に達した場合には行うことはできない。

4.4.1 後方への拡大を用いた検証例

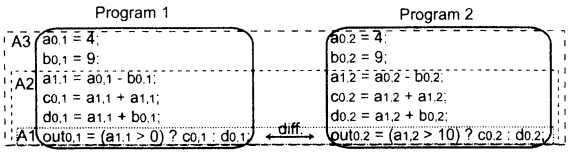
図 5 は、差異の等価性が示せなかった場合に、後方へ検証範囲を拡大することによって、その差異の等価性が示せるようになる例である。

まず、差異のみを検証範囲として等価性検証を行うと、図の A1 が検証範囲となり、変数 a_1, c_0, d_0 が入力変数、変数 out_0 が出力変数となる。しかし、3 つの入力変数について何の情報もないため、出力変数の等価性は示すことができない。そこで、後方への検証範囲拡大を 1 回だけ行い、図の A2 を検証範囲として再び検証を行う。しかし、この 2 回目の検証でも変数 out_0 の等価性を示すことができないため、再び後方への検証範囲拡大を行う。図の A3 を検証範囲とした 3 回目の検証では、新たに $a_0 = 4, b_0 = 9$ という情報が加えられる。結果的に、この新たに加わった代入文によって、変数 a_1 の値が-5 であることが分かり、変数 out_0 が等価であることを導くことができる。

4.4.2 前方への拡大を用いた検証例

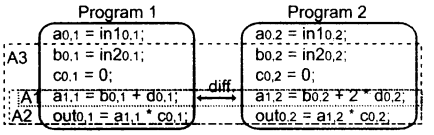
図 6 は、差異が等価でないことが証明された場合に、前方へ検証範囲を拡大することによって、記述全体の出力変数は等価であることが示せる例である。

ここでも同様に、初期検証は差異部分のみを検証範囲として行われるため、その検証範囲は図の A1 となる。A1 での等価性検証は、変数 a_1 を出力変数、変数 b_0, d_0 を入力変数として行われるが、等価であることを示すことはできない。そこで、この



Input variables:
Output variables: out
A1 (difference only)
  Equivalence checking of out0
  Equivalence of out0 is not proved (a_1, c_0, and d_0 are unknown)
A2 (1st backward extension)
  Equivalence checking of a_1, c_0, d_0, out0
  Equivalence of a_1, c_0, d_0, and out0 is not proved (a_0 and b_0 are unknown)
A3 (2nd backward extension)
  Equivalence checking of a_0, b_0, a_1, c_0, d_0, out0
  Equivalence of a_0, b_0, a_1, c_0, d_0, and out0 is proved
Result
  out0 is equivalent in both programs

図 5　後方への拡大を用いた等価性検証例



Input variables: in1, in2
Output variable: out
A1 (difference only)
  Equivalence checking of a_1
  Equivalence of a_1 is not proved (b_0 and d_0 are unknown)
A2 (1st forward extension)
  Equivalence checking of a_1 and out0
  Equivalence of a_1 and out0 is not proved (b_0, d_0, a_1, and c_0 are unknown)
A3 (1st backward extension)
  Equivalence checking of b_0, c_0, a_1, and out0
  Equivalence of b_0, c_0 and out0 is proved
Result
  out0 is equivalent in both programs

図 6　前方への拡大を用いた等価性検証例

例では、前方へ検証範囲を拡大を行っている。その結果、図の A2 が新たな検証範囲となる。しかし、2 回目の検証でも A2 の出力変数である変数 out_0 の等価性を示すことができない。これ以上は、前方へ拡大できないため、後方へ拡大し、検証範囲 A3 を得る。ここで、$c_0 = 0$ であるため、変数 out_0 は変数 a_1 の値によらず 0 になることが分かる。

4.4.3　検証範囲拡大の戦略

先にも述べたように、1 回の検証範囲拡大の方法は、検証範囲に対する入出力変数の組み合わせの数だけ存在するため、どれが最も検証結果を早く得るために最適であるかを決定するのは難しい。そこで、ここでは、一般的に有効であると考えられる検証範囲拡大手法を挙げる。ただし、どの手法が有効であるかは例題に大きく依存する。

- 記述全体の入力変数に達するまで後方への拡大のみを繰り返す (差異の等価性を優先的に調べる)
- 記述全体の出力変数に達するまで前方への拡大のみを繰り返す (差異が影響が及ぶかどうかを優先的に調べる)
- 後方への拡大と前方への拡大を交互に行う
- 1 回の拡大で、後方と前方の拡大を同時に行う

また、検証範囲の拡大回数に上限を与え、その回数に達しても等価性が示せない場合には、その時点で「非等価」という結果を出して検証を終えるという戦略も考えられる。

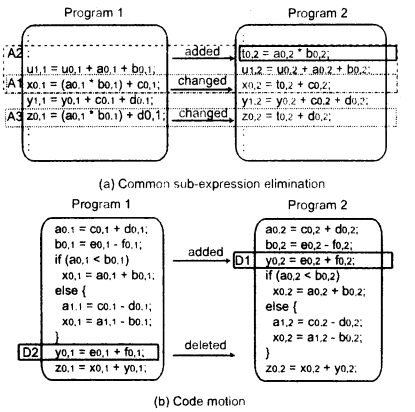


図 7　共通部分式の削除とコード移動の例

4.5　特定の変換・最適化を対象とした検証範囲拡大手法

前節で述べたように、等価でない差異がある場合には、検証範囲が繰り返し拡大され、記号シミュレーションすべき記述量が大きくなってしまう可能性がある。そこで、特定の記述変換・最適化を対象として、検証範囲拡大の手順をあらかじめ決めておき、それに従って検証を行うことが考えられる。このような特定の変換のみを対象とした手法を導入することによって、不必要に検証範囲の拡大を繰り返すことなく、その変換が等価であるかどうかを検証することができる。以下では、その例として、共通部分式の削除とコード移動の検証について説明する。

4.5.1　共通部分式の削除

共通部分式の削除は、図 7(a) に示すように、記述中で複数回使われている部分式を一度計算して一時変数に代入しておき、その部分式を使用するごとに計算することを省く最適化手法である。ここでは、記述間の差異に基づいて、共通部分式の削除の前後での等価性を検証することを考える。

共通部分式の削除では、2 種類の差異が生じる。一方は、共通部分式を計算して一時変数に代入する部分であり、他方は、その一時変数を使用する部分である。検証は、後者の代入文に対して行う。例えば、図の A1 の等価性を検証しようとしたとき、変数 t_0 が一時変数であることが分かっているため、この変数 t_0 に対する代入文が検証範囲に含まれるように、後方へ検証範囲を拡大すればよい。そうすることによって、差異 A1 の等価性を容易に検証することができる。ここで、変数 c_0 からの後方への拡大や前方への拡大を行わないことが、共通部分式の削除を対象とした検証範囲拡大手法の効果的な部分である。なお、差異 A3 の検証では、既に変数 t_0 が記号シミュレーション済みであり、その情報を持っているため、検証範囲の拡大を行わずに等価性を検証できる。

4.5.2　コード移動

ここでは、最も簡単なコード移動として、図 7(b) に示すような同一パス内でのコード移動の等価性を考える。記述の差異から、図の D1 と D2 で代入されている変数 y_0 が等価である

表1 実験結果

	exp1	exp2	exp3	exp4
Result	Eqv	Not eqv	Eqv	Not eqv
Total nodes	672	672	672	672
Nodes in diff	96	96	96	96
Verified nodes	96	11	96	43
Verification time (verified nodes)	1.5 sec	< 1 sec	1.5 sec	< 1 sec
Verification time (whole programs)	559 sec	12 sec	543 sec	112 sec

かどうかを検証すればよいことが分かる。D1 と D2 だけでは、変数 e_0, f_0 がそれぞれ等価かどうか分からないため、これらの変数から後方へ検証範囲を拡大することを考える。このとき、この例では、変数 e_0, f_0 は記述全体に対する入力変数であるため、等価であることと仮定して検証を行っている。これより、D1 と D2 における変数 y_0 も等価であることが分かり、コード移動は正しく行われていたと結論付けることができる。

5. 評価結果

本稿で提案した検証手法全体は、現在実装中であるため、簡単な評価として、検証範囲が全体に対してどれほどの大きさになるのかを測定した。ここでは、2 つの C 言語記述の SDG の総ノード数、差異に含まれるノード数、最終的な検証範囲に含まれるノード数を測定した。また、参考として、検証範囲に含まれたノードに対する 1 回だけの記号シミュレーション時間と全記述に対する記号シミュレーション時間を Xeon 2.4GHz プロセッサ、2GB メモリを持つ PC を用いた実験によって求めた。例題として、逆離散コサイン変換を行う C プログラムに対する最適化前後の記述を用いた。検証範囲の拡大手法としては、1 回の拡大で前方と後方に 1 段階ずつ拡大し、入力と出力に達するまで行った。

評価結果は表 1 のようになった。この表から明らかなように、検証範囲に含まれるノード数は全体に比べて非常に小さくなっている。実験 2 と実験 4 において、検証範囲となったノード数が差異に含まれるノード数よりも少なくなっいるのは、全ての差異を検証する前に、反例が得られたためである。さらに、記号シミュレーション時間も短時間で可能であった。実際には、これに加えて、検証範囲の決定・拡大のための時間が必要であるが、検証時間の大部分は記号シミュレーションによって費されると考えられる。この評価実験では、等価でない場合にも、検証範囲が全体に比べて非常に小さくなった。これは、例題の性質に依存する面もあるが、差異に依存関係のある部分のみを検証範囲としているためである。また、等価な場合の実験結果からは、局所的に等価性が証明できる差異の多い変換に対して、提案手法が全体に対して十分に小さな検証範囲で等価性を検証することができたといえる。

6. 結論と今後の課題

6.1 結　論

本稿では、記号シミュレーションに基づく C 言語記述を対象とした効率的な等価性検証手法を提案した。提案手法では、記述間の差異に対して局所的な記号シミュレーションを行うことによって、記号シミュレーションを行う範囲を減少させることが可能である。局所的な検証範囲の検証では、それぞれの差異の等価性が示されるまで、依存関係に従って検証範囲を拡大することによって、誤った検証結果を得ることを防いでいる。そのため、一般的に、等価でない差異がある場合には、検証範囲が大きくなってしまう可能性がある。そこで、併せて、特定の変換・最適化を対象とした検証手法拡大手法の可能性について検討した。例題を用いた実験評価によって、検証範囲に含まれる依存ノード数が全体に比べて小さいこと、その記号シミュレーション時間も短いことを示した。

6.2 今後の課題

今後の課題としては、有効な検証範囲拡大手法を考案することが挙げられる。特に、第 4 節の最後で述べたような特定の変換や最適化を対象とした検証範囲拡大手法を数多く用意し、設計者が行った変換・最適化の種類が与えられた場合に、非常に高速に等価性を検証することが、実用上は重要であると考えられる。加えて、検証システムの実装を進め、より多くの例題への適用と評価を行っていく予定である。

文　献

[1] 竹中崇, 向山輝, 若林一敏, 中田勝, 前川晃, 山際馨, "動作合成前後の動作記述と RTL 記述の論理等価性検証," 第 17 回回路とシステム軽井沢ワークショップ論文集, pp.555-560, 2004.

[2] L. Semeria, A. Seawright, R. Mehra, D. Ng, A. Ekanayake, and B. Pangrle, "RTL C-Based Methodology for Designing and Verifying a Multi-Threaded Processor," *Proc. of 39th Design Automation Conference*, pp.123-128, 2002.

[3] 松本 剛史, 齋藤 寛, 藤田 昌宏, "C 言語を対象とした記述間の差異に基づく効率的な等価性検証手法," 電子情報通信学会技術研究報告, Vol.103, No.702, pp.61-66, 2004 年 3 月

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

[5] A. Stump, C. Barret, and D. Dill, "CVC: a Cooperating Validity Checker," *Proc. of 14th International Conference on Computer-Aided Verification*, pp.500-504, 2002.

[6] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural Slicing Using Dependence Graphs," *ACM Transactions on Programming Languages and Systems*, Vol.12, No.1, pp.26 60, 1990.

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

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

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