線形計画法に基づく逐次化を利用したシステムレベル設計での動作並列化前後での等価性検証手法

松本　剛史[†]　Thanyapat　Sakunkonchak[††]　齋藤　寛[†††]　小松　聡[††]　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
††† 会津大学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
E-mail: matsumoto@cad.t.u-tokyo.ac.jp, thong@cad.t.u-tokyo.ac.jp, hiroshis@u-aziu.ac.jp,
komatsu@cad.t.u-tokyo.ac.jp, fujita@ee.t.u-tokyo.ac.jp

あらまし　システムレベル設計では、逐次実行として記述された仕様を対象アーキテクチャに割り当てる際に、並列プロセスが導入される。また、アーキテクチャの変更や最適化のために、スケジューリングが変更されることもある。本稿では、このような並列化前後や異なるスケジューリングの設計に対する等価性検証手法を提案する。提案手法では、与えられた 2 つの設計記述の並列部分を逐次化した後に、等価性検証を行う。このため、並列プロセス間の順序の違いによって生じる中間状態を考慮せずに検証を行うことができる。逐次化では、線形計画法ソルバーを決定手続きとして利用し、依存関係のある並列実行における実行順序の一意性を判定している。実験として、SpecC で記述された実例題に対する検証結果を示す。

An Equivalence Checking Method for System-Level Designs Under Different Schedulings Applying Sequentialization with ILP Solvers

Takeshi MATSUMOTO[†], Thanyapat SAKUNKONCHAK[††], Hiroshi SAITO[†††],
Satoshi KOMATSU[††], and Masahiro FUJITA[††]

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

Abstract　In system-level design, concurrency is introduced when a given sequential specification is mapped to processing elements. Also, schdulings can be refined to obtain more optimized designs. In this paper, we propose an equivalence checking method for such designs under different schedurings. In the method, a pair of given deisngs are sequentialized at first, and then, checked the equivalence of the two sequential descriptions. In the sequentializaion, ILP solvers are utilized as a decision procedure to check whether concurrent statements dependending on each other are always executed in an identical order. We show the experimental results on some real examples in SpecC.

1. はじめに

システム LSI などのソフトウェア/ハードウェア協調設計では、従来の RTL 設計などに比べて抽象度の高い設計記述が可能なシステムレベル設計が行われることが多くなってきている。システムレベル設計では、

- 高速なシミュレーション
- 柔軟なアーキテクチャ変更・SW/HW 分割

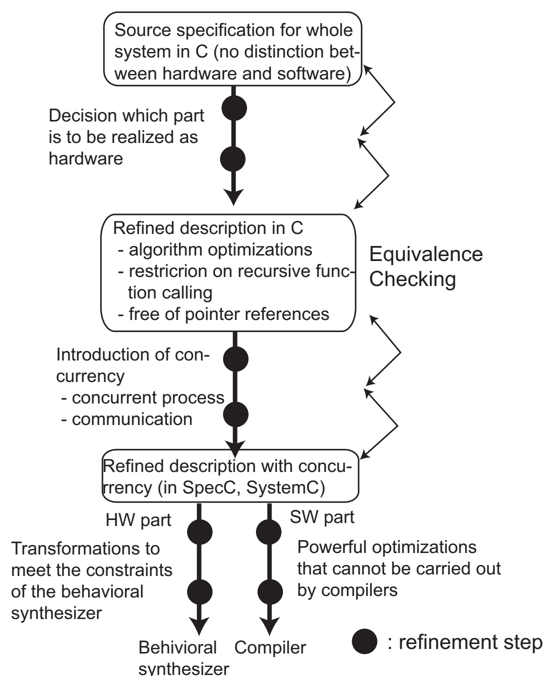


図 1　システムレベル設計

- 少ない記述量での設計記述

が可能であり、その結果、より最適なアーキテクチャをより短時間で探索することができると考えられる。

システムレベル設計では、C 言語や C 言語ベースの設計記述言語が用いられることが多く、与えられた仕様記述から、HW 部分に対する動作合成可能な記述と SW プログラムの導出が行われる。図 1 は、典型的なシステムレベル設計の流れを示したものである。システムレベル設計では、基本的には、システムが行うべき動作とその実行順序が決定されるが、図に示すように、その過程で並列動作の導入やスケジューリングの変更などが行われる。本稿では、このような並列化やスケジューリングの変更の前後の設計に対する等価性検証手法を提案する。

一般的に、並列プロセスを含む動作の等価性を検証する場合、各ステートメントの実行順序の非決定性を考慮した検証が必要である。この場合、多くの interleave 状態を含めた等価性検証を行う、もしくは、partial order reduction 法を適用しながら検証を行う必要がある。しかし、多くの設計においては、どのような実行順序で並列プロセス内のステートメントが実行されたとしても、同じ計算結果を得るように同期設計が行われる。逆に、実行順序の非決定性によって、計算結果が変わるような設計は、同期部分に設計誤りがある可能性が非常に高いといえる。そこで、提案する検証手法では、与えられた並列性を持つ設計を逐次化し、逐次化された 2 つの設計に対して等価性検証を行う。

並列動作を逐次化するためには、並列動作で許されている全ての実行順序に対して、計算結果が等価であることを保証する必要がある。そのために、本研究では、互いに依存関係を持つ並列動作部分の実行順序が一意であるための条件を整数線形式として表現し、その条件が満たされるかどうかを線形計画法ソルバー (ILP ソルバー) を用いて判定する。このとき、依存関係のある並列実行部分についてのみ、実行順序の一意性を調べる必要があるため、一般的な設計においては、非常に短時間に行うことができると予想される。

提案手法では、逐次化を行った後に等価性検証を行うため、等価性を検証する処理は並列実行のない動作記述を検証する場合と同じように行うことができる。このとき、2 つの逐次化された動作記述中の差異がある部分のみを記号シミュレーションによって局所的にに検証する。そのため、スケジューリング変更の際に、モジュール・関数の分割が行われたり、動作記述が変更されたりした場合であっても、等価性を検証することができる。また、SpecC [1] で記述されたいくつかの実例題に対して行われたスケジューリング変更の検証を行い、提案手法によって、高速に等価性を検証できることを示した。

本稿の構成は以下のようである。第 2 節で関連研究を紹介する。第 3 節で提案する検証手法を述べ、第 4 節で提案手法による実験結果を示す。最後に、第 5 節でまとめと今後の課題と示す。

2. 関 連 研 究

2.1　システムレレベル設計におけるスケジューリングの検証

システムレベル設計におけるスケジューリングの検証としては、文献 [3]〜[5] のようなデータパス付きの有限状態機械 (FSM: Finite State Machine)、例えば Finite State Machine with Datapaths (FSMD) や Extended Finite State Machine (EFSM)) として表し、その上で等価性を検証する手法が提案されている。FSM をベースとした等価性検証では、一般的に、2 つの FSM 中のパスの等価性を調べるため、規模の大きな設計に対しては適用が難しい。そのため、文献 [5] のように動作合成環境から比較すべきパスの情報を得たり、文献 [3] のように対応する状態を探しながら検証を行う工夫が必要である。本稿では、逐次化された 2 つの動作記述の等価性検証を行う際に、差異部分のみの等価性を検証するため、多くのスケジューリング変更に対して、記述の一部のみを調べることによって検証を行うことができる。

また、文献 [2] では、動作を並列化したり、逐次化したりした際の等価性をあらかじめ定義された規則に基づいて検証する手法が提案されている。この手法では、関数単位で実行順序が変更されたり、並列化が行われたりした場合の等価性を高速に検証可能だが、個々の関数の中身が変更される場合には適用できない。本稿の提案手法では、スケジューリング変更のために、部分的な動作記述の変更 (関数の分割や関数内部の実行順序変更など) がある場合にも等価性を検証できる。

2.2　ILP ソルバーを利用した同期検証

本稿で行う逐次化手法は、文献 [6] で提案されている、システムレベル設計におけるデッドロックの有無のような同期設計の誤りを検証するための手法を応用したものである。この手法では、SpecC 設計記述中の各ステートメントが満たすべき順序関係の条件を線形式として表現し、全ての条件の間で矛盾がないかどうかを ILP ソルバーを用いて判定している。もし、矛盾があれば、実行されないステートメントがあることになり、

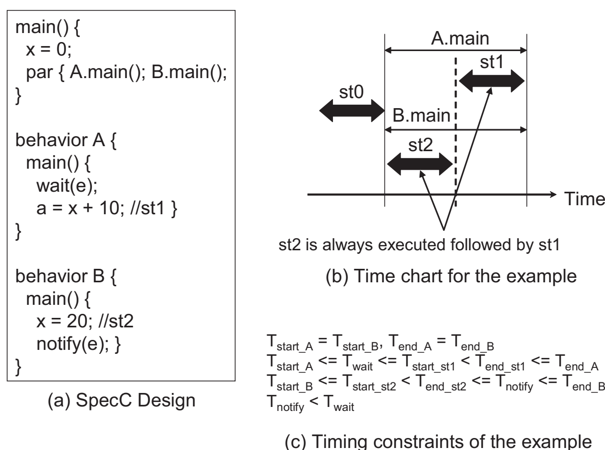


図 2　実行タイミングに対する条件式

デッドロックが起こり得ることが分かる。

図 2 は、文献 [6] の手法によって、デッドロックの有無を検証する例である。例題は、SpecC 言語で記述されており、並列実行と同期に関して、以下の記述を含んでいる。

- par によって記述されたビヘイビア (関数) は並列実行される
- $notify/wait$ 文は同期を実現するためのものであり、$wait(e)$ は $notify(e)$ が実行されるまで、そのプロセスの実行を止める。

図に示すように、文献 [6] の手法では、まず、与えられた SpecC 設計内の各ステートメントが満たすべき実行順序の条件と検証するプロパティ(この例では「デッドロックがない」) を線形式として表し、それらの条件間で矛盾がないかを ILP ソルバーを利用して調べる。図中の T_{start_X}, T_{end_X} は、ステートメント (または関数)X の開始時刻、終了時刻をそれぞれ表している。また、T_{notify}, T_{wait} は、$notify$ 文の実行時刻、$wait$ 文が $notify$ 文の実行を受けてプロセスの実行を再開する時刻をそれぞれ表している。デッドロックがないことは、$notify$ 文が $wait$ 文より先に実行される ($T_{notify} - T_{wait} < 0$) を与えればよい。デッドロックを含む場合には、ILP ソルバーはそれぞれの時刻を表す変数への代入値を発見することができず、与えた条件間で矛盾があることを示す。

本稿では、次節で述べるように、SpecC 設計記述から得られる条件式に加えて、依存関係のある並列実行部分の実行順序に関する条件を併せて与えることによって、実行順序の一意性を確認する手法を提案する。

2.3　記号シミュレーションによる等価性検証

逐次化された 2 つの動作記述の等価性検証には、文献 [7] で提案されている検証手法を用いる。この手法では、記述間で差異のある部分のみを局所的に記号シミュレーションすることによって、全体の等価性を検証するため、並列化やスケジューリング変更のように、動作が大きく変更される可能性の低い場合に、効率的に等価性を検証することができる。

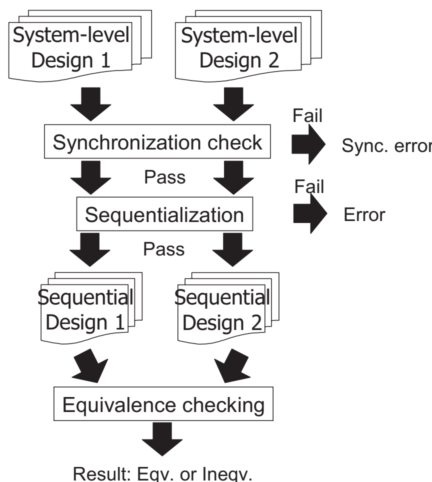


図 3　検証の流れ

3. 提案する検証手法

提案する検証手法の流れを図 3 に示す。まず、与えられた 2 つのシステムレベル設計記述に対してそれぞれ逐次化を行う。このとき、デッドロックの有無についても、文献 [6] の手法に基づいて検証を行い、デッドロックが検出されれば、そこで検証を終了する。次に、逐次化された 2 つの設計記述の等価性を記号シミュレーションを適用することによって行う。なお、提案手法で検証する等価性は、等価な入力パタンを与えたときに、等価な出力パタンが得られることとする。ただし、設計中に状態変数がある場合には、対応する変数は等価であることを前提として検証を行う。また、設計記述中の代入文・条件評価は、ある時点において 1 つだけ実行されるものとする。つまり、並列実行する 2 つのプロセスがあった場合、ある一瞬のみを考えれば、どちらかのプロセスの代入文・条件評価のみが実行されることとする。

3.1　逐　次　化

並列実行を含む設計記述を等価な逐次実行記述へと変換する場合、変換前の設計記述は、以下の条件を満たさなかればいけない。

- デッドロックがない
- 並列プロセスのあり得る全ての実行順序に対して、等価な実行結果が得られる

なお、第 1 点については、第 2 節で述べた文献 [6] の手法を適用して検出する。第 2 点については、以下の 2 つの事項を調べればよい。

- 並列プロセス間で共有されている変数に対して、2 つのプロセスで代入が行われる順序が常に同じである (ただし、その変数を 1 度も使わない場合には考慮しなくてよい)
- あるプロセスで共有変数が代入され、同じ変数が他のプロセスで使われる場合、その実行順序は常に同じである (代入してから使用するか、使用してから代入するか、のどちらかしか起こらない)

実際には、これらが満たされたなかっとしても、実行結果が変わらない場合がある。例えば、共有変数への代入が同期を取ら

ずに行われたとしても、代入されている値が常に等価であれば、結果的に、その変数を使用したときの変数値は等価となる。本研究では、一般的には、上述の 2 点を満たさない設計は誤りである可能性が高いため、同期設計に誤りがあることを報告して終了し、等価性は検証しない。

上述した 2 点に該当する箇所がある場合、その代入・使用が行われる 2 つのベーシックブロックをそれぞれ $BB1, BB2$ として、共有変数の代入・使用に対する実行順序の一意性を調べるために、次の 2 つのプロパティを ILP ソルバーを用いて検証する。

プロパティ1：$T(BB1_{start}) > T(BB2_{end})$

プロパティ2：$T(BB1_{end}) < T(BB2_{start})$

$T(BB_{start}), T(BB_{end})$ は、それぞれ、BB の開始時刻と終了時刻を表す。なお、ここでのベーシックブロックは、条件分岐と $notify/wait$ 文を含まない一連のステートメントとして定義する。それぞれのプロパティに対する検証結果は、$pass$ または $fail$ であるため、(プロパティ1 の結果, プロパティ2 の結果) によって、以下の 4 通りに分類される。

- **(pass, pass)** $BB1$ は $BB2$ より先に実行される場合も、後に実行される場合もある
- **(pass, fail)** $BB2$ は常に $BB1$ より先に実行される
- **(fail, pass)** $BB1$ は常に $BB2$ より先に実行される
- **(fail, fail)** 起こり得ない

そのため、2 つのプロパティの検証結果が共に pass であった場合には、実行順序が一意でないため、共有変数へのアクセスにおいて、正しく同期が取れていないことを報告して、検証を終了する。

以上の手法を用いて、逐次化は以下の手順で行われる。

（1） デッドロックの有無を検証する。もし、デッドロックがあれば、エラーを出力して終了する

（2） 設計記述中の全ての par 構文をその深さとともに集合 SET_{par} の要素とする。par 構文の深さは、階層的に par 構文が記述されている場合の最上位の par 構文からの深さである (階層的でない場合は、0 となる)

（3） SET_{par} の要素の par 構文のうち、最も大きな深さを持つ要素を取り出し、以下を行う

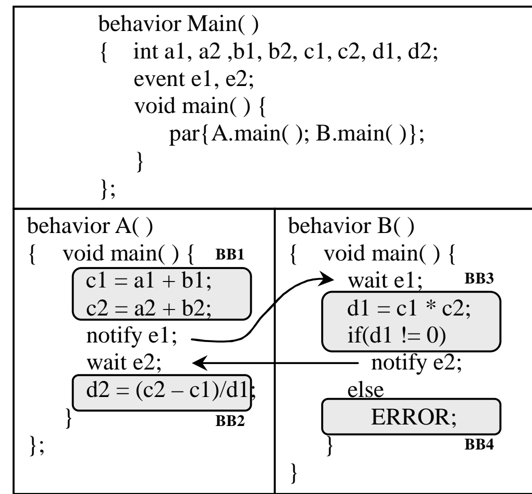
（a） 同一の共有変数に対する代入を行っているベーシックブロック $BB1, BB2$ が異なるプロセスにある場合。$BB1$ と $BB2$ の実行順序の一意性を調べる。一意でなければ、エラーを出力して終了

（b） 同一の共有変数の代入と使用を行っているベーシックブロック $BB1, BB2$ が異なるプロセスにある場合。$BB1$ と $BB2$ の実行順序の一意性を調べる。一意でなければ、エラーを出力して終了

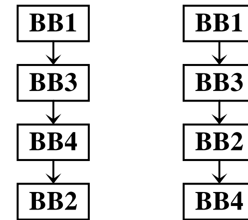
（c） 同期関係と前項で調べた実行順序を満たすように、各ベーシックブロックを並べ、この par 構文を逐次化する

逐次化の例

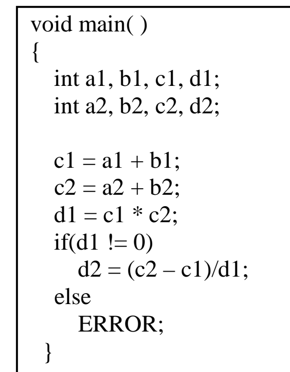
ここでは、本節で述べた逐次化の例を図 4,5 を用いて述べる。



(a) Two parallel behaviors with synchronization



(b) Possible execution orders. No dependence between BB2 and BB4 so either of them is ok for sequentialization.



(c) Sequentialized version of (a)

図 4　逐次化が可能な例

紹介する例題は、SpecC 記述を用いて記述されており、並列実行と同期に関する構文については、第 2 節で述べた通りである。

図 4 に示す例では、共有変数の代入・使用があるため、変数 $c1, c2$ について $BB1$ と $BB3$、また、変数 $d1$ について $BB2$ と $BB3$ の実行順序が常に同じである必要がある。この例題では、同期が行われているため、許される実行順序は図の (b) に示すものしかない。これより、常に、$BB1$ の後に $BB3$、また、$BB3$ の後に $BB2$ が実行されるため、それぞれの共有変数 $c1, c2, d1$ への代入結果は、実行順序によらず必ず等価になることが保証されるため、逐次化することができる。

図 5 に示す例では、同期以外は図 4 と同じ記述となっている。しかし、共有変数 $c1, c2$ に対する代入と使用が含まれる $BB1$

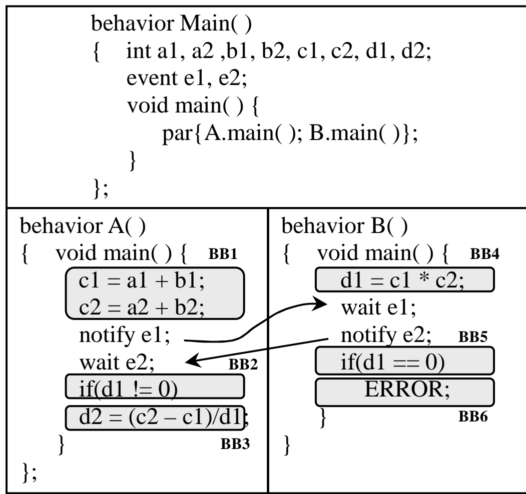


図 5 逐次化が不可能な例 (依存関係のある並列実行部分の実行順序が一意でない場合)

と $BB4$ の実行順序が一意に決められないため、$BB4$ での変数 $d1$ の代入結果が異なる可能性があり、逐次化することができない。

逐次化可能な並列・同期設計

提案手法では、条件分岐の条件の評価を行わないため、逐次化できる記述は、どの分岐を実行した場合であっても、共有変数に対する代入・使用の実行順序が同じである場合に限られる。実行されるパスによって実行順序が入れ換わるような設計に対しては、提案手法では、共有変数に対する代入・使用の順序が変わると捉えるため、実際には全ての場合に対して等価な 1 つの逐次実行があったとしても、逐次化することができない。ただし、そのような並列・同期設計が行われることは非常に希であると考えられる。

3.2 記号シミュレーションによる等価性検証

逐次化された記述は、文献 [7] で提案されている記号シミュレーションによる等価性検証手法によって、等価性を検証する。この手法では、記述間の差異部分のみを局所的に記号シミュレーションすることによって、効率性を向上させ、大規模な記述の検証を可能にしている。記述間の差異を有効に利用するために、逐次化された設計記述間で、関数ごとの対応を取る。この対応取りによって、スケジューリング変更や並列化の前後で、記述上、関数の順番などが変わった場合にも、効果的に差異を特定することができる。次に、それぞれの関数に差異があれば、その等価性を検証する。このとき、全く差異のない関数は、uninterpreted に扱うことによって、高速化を図ることができる。一般的に、スケジューリング変更の前後では、多くの関数は変更されることがないため、等価性検証は短時間で行うことができると考えられる。

4. 実験結果

4.1 実 装

デッドロックの検証と等価性検証については、文献 [6], [7] の手法を実装したプロトタイプツールを使用した。逐次化については、並列プロセス中における共有変数の代入・使用を把握するために、依存グラフを利用した。使用した依存グラフは、GrammaTech 社の CodeSurfer [9] によって生成されるシステム依存グラフ [8] である。ILP ソルバーとしては、ILOG 社 CPLEX を用いた。

表 2 等価性検証の結果

Sequentialized Benchmarks		LOC of *diff*	# of Eqv. check	Runtime (sec)
seq_IDCT1	seq_IDCT2	156	18304	< 0.1
seq_IDCT1	seq_IDCT3	148	18304	< 0.1
seq_IDCT3	seq_IDCT4	224	11691	< 0.1
seq_VocSpec	seq_VocArch	25	22	< 0.1
seq_VocArch	seq_VocSched	0	-	-
seq_MP3DEC1	seq_MP3DEC2	478	120425	< 0.1

4.2 例 題

ここでは、以下の例題を用意して、実験を行った。例題は、逆離散コサイン変換 (IDCT) と Vocoder、MP3 デコーダに対して、並列化やスケジューリング変更を行った結果として得られたものである。全ての例題は、SpecC 言語で記述されている。各 SpecC 記述の行数、記述中の含まれる $behavior, channel, par$ 構文の数、同期の数を表 1 に示す。

IDCT1 は IDCT の仕様記述であり、行計算と列計算を行うモジュールを 1 つずつ持ち、逐次実行の設計である。**IDCT2** は **IDCT1** に対して、行計算と列計算を行うモジュールをそれぞれ 2 つずつに増やし、一部を並列化した設計である。**IDCT3** は **IDCT1** に対して、行計算と列計算を行うモジュールを 8 個ずつ用意し、並列性をさらに高めた設計である。**IDCT4** は **IDCT3** の行計算・列計算モジュール内での並列性を高めるために、明示的に並列に実行されるようにスケジューリングした設計である。

VocSpec は Vocoder の仕様として与えられた記述であり、全ての動作を 1 つの HW 上で行うことを仮定している。また、依存関係のない関数は並列に実行するように設計してある。**VocArch** は **VocSpec** に対して、動作の一部を DSP を用いて行うようにアーキテクチャを変更し、それに伴って、プロセッシングエレメントの割り当てを行った設計である。**VocSched** は **VocArch** に対して、各プロセッシングエレメント (HW と DSP) 内でスケジューリングを行った設計記述である。

MP3DEC1 は MP3 デコーダの SpecC 仕様記述であり、ソフトウェアと 1 つの追加 HW によって実行される。**MP3DEC2** は **MP3DEC1** に対して、計算負荷の大きい DCT 計算に対して、2 つの追加 HW を割り当てた設計である。

4.3 検証結果

表 1 は、逐次化を行った際に発見されたデッドロック数、ILP ソルバーの実行回数、デッドロック検証を含む全体の実行時間を示している。並列プロセス間で共有変数を用いている **IDCT4** と **MP3DEC2** においてのみ実行順序の確認のための ILP ソルバーの実行が必要であった。その他の例題では、並列プロセス間で依存関係がなかったため、同期による条件のみに従って

表 1　例題の性質と逐次化の結果

Benchmark	LOC (original)	LOC (sequentialized)	# of Behaviors	# of Channels	# of *par*	# of Sync.	# of Deadlock	# of ILP run	Runtime (sec)
IDCT1	300	300	4	1	0	0	0	0	0.7
IDCT2	314	298	6	1	8	0	0	0	0.8
IDCT3	256	251	4	1	2	0	0	0	0.7
IDCT4	390	360	18	1	4	10	0	48	1.0
VocSpec	9165	9165	102	4	10	4	1	0	39.0
VocArch	10178	10164	144	14	15	14	1	0	48.5
VocSched	10139	10132	144	14	2	14	1	0	42.0
MP3DEC1	8580	8580	44	6	4	6	1	0	160.2
MP3DEC2	8576	8560	46	6	5	10	1	18	162.2

逐次化が可能であった。なお、Vocoder と MP3 デコーダの例題では、*channel* 部分にデッドロックが検出されたが、これを修正して実験を続けた。なお、ILP ソルバーの実行時間の総和は、全ての例題で 1 秒以下であった。それ以外の実行時間は、デッドロック検証と前処理に費されている。

逐次化された記述に対する等価性検証の結果を表 2 に示す。表では、差異の行数 (両記述の和)、検証ツール内部で行われた等価性判定の回数、検証時間を示している。実験では、全ての例題に対して等価であることが示された。逐次化された **VocArch** と **VocSched** との間には差異がなかったため、この例題に対しては、記号シミュレーションによる検証を行う必要はなかった。これは、**VocArch** と **VocSched** への変更では、単純にスケジューリングのみが行われたためである。このような場合は、文献 [2] の手法によっても検証することができる。その他の例題では、一部の関数に差異があったため、記号シミュレーションを行って等価性を検証する必要があった。並列化やスケジューリングの前後では、実行の順序が変化するだけのことが多く、複雑な最適化は行われないため、非常に高速に記号シミュレーションによって等価性を検証することができている。

5.　まとめと今後の課題

本稿では、システムレベル設計における並列化やスケジューリング変更の前後の記述に対する等価性検証手法を提案した。提案手法では、与えられた設計記述を逐次化してから等価性を検証するため、検証時に並列プロセス間に存在する interleave を考慮せずに検証が可能である。逐次化として、並列プロセスが共有する変数に対する代入・使用の実行順序が常に同じであるかどうかを ILP ソルバーを利用して判定することによって、元の並列記述と等価な逐次記述を得る手法を提案した。2 つの逐次化された記述の等価性検証は、記号シミュレーションに基づく手法を適用した。提案した検証手法は、実際の並列化やスケジューリング変更の例題を用いた実験を通して、短時間で等価性を検証することができることが示された。

提案する検証手法では、逐次化の際に、1 つの逐次実行に変換することができない場合を対象としていなかった。これは、主に、そのような設計では、同期設計に誤りがある可能性が高いためである。しかし、等価性を検証するという立場からは、そのような場合であっても、等価であるかどうかを判定すべきであると言える。そこで、1 つの逐次実行に変換することができない場合の検証手法を検討していく必要がある。

また、本稿での提案手法を用いて、高位合成前後における等価性検証の手法を今後の課題として検討する。高位合成では、スケジューリングの他にも、様々な変換や最適化が行われている。そこで、それらの最適化を検証するための前処理として本稿で提案した逐次化を用いることが有効であるかどうかを検討する予定である。

文　献

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

[2] S. Abdi and D. Gajski, “Functional Validation of System Level Static Scheduling,” *Proc. of Design, Automation and Test in Europe*, pp.542–547, Mar. 2005.

[3] C. Karfa, C. MAndal, D. Sarkar, S. R. Pentakota, C. Reade, “A Formal Verification Method of Scheduling in High-Level Synthesis,” *Proc. of International Symposium on Quality Electronic Design*, pp.71–76, Mar. 2006.

[4] P. Ashar, A. Raghunathan, A. Gupta, and S. Bhattacharya, “Verification of Scheduling in the Presence of Loops Using Uninterpreted Symbolic Simulation,” *Proc. of International Conference on Computer Design*, pp.458–466, 1999.

[5] 竹中, 向山, 若林, 中田, 前川, 山際, “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 年 4 月.

[6] T. Sakunkonchak, S. Komatsu, and M. Fujita, “Synchronization Verification in System-Level Design with ILP Solver,” *Proc. International Conference on Formal Methods and Models for Codesign*, pp.121–130, July 2005.

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

[8] S. Horwitz, T. Reps, and D. Binkley, “Interprocedural Slicing Using Dependence Graphs,” *ACM Trans. on Programming Languages and Systems*, Vol.12, No.1, pp.26–60, 1990.

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