設計自動化 72－24
(1994. 10. 28)

論理シミュレーションをベースとした プロセサ制御の効率的検証手法

中田 恒夫, 古渡 聡, 岩下 洋哲, 広瀬 文保
富士通研究所 ＣＡＤ研究部
〒211 川崎市中原区上小田中 1015
E-mail: nakata@flab.fujitsu.co.jp

あらまし 近年，論理シミュレーションを用いたプロセサの論理検証の高速化技術が次々と開発されているが，入力である検証プログラムの開発は，人手やランダム生成に頼っていた．我々は，この検証プログラムを自動生成する手法を開発した．本手法では，プロセサの数学的モデルとして有限状態マシンを考え，特定の制御に対する検証プログラム生成問題を，状態集合への到達可能性問題にマッピングした．検証したい制御に関する仕様を抽出しモデル化しているため，形式的検証で問題となる状態数の増加は緩やかである．本手法の下で，パイプライン制御やキャッシュ制御に対する検証プログラム自動生成を実現しているが，レジスタ転送レベルで記述されたハードウェアの検証データ生成にも適用可能である．

和文キーワード プロセサ, 論理シミュレーション, 論理検証, 有限状態マシン

Effective Verification Method for Processor Controls Based on Logic Simulation

Tsuneo Nakata, Satoshi Kowatari, Hiroaki Iwashita, and Fumiyasu Hirose
CAD Laboratory, Fujitsu Laboratories Ltd.
1015 Kamikodanaka, Nakahara-ku, Kawasaki 211, Japan
E-mail: nakata@flab.fujitsu.co.jp

Abstract This paper presents an efficient verification method for processor controls using logic simulation. Currently, the verification program that is the input sequence for logic simulation, is developed mainly by hand or by weighted random generators. Our method uses a finite state machine as processor model. The problem of generating the verification program is then mapped into the reachability problem for the finite state machine, which can be solved efficiently by using BDD's. We have succeeded in applying this method to the pipeline control and cache control for microprocessors, and we hope it can also be expanded to the verification data generation for hardware systems in register transfer level.

英文 **key words** Processor, Logic Simulation, Logic Verification, Finite State Machine

1 はじめに

プロセサの性能は1.5年で2倍[4]という高い水準で向上している．これは素子の高速化とともに，プロセサの制御技術の進歩によるところが大きい．

プロセサの開発者は，高度化する設計に対処する一方で，設計期間短縮を要求される．そのためには設計期間の4割を占める設計検証期間[5]の短縮が不可欠である．設計検証で主要な役割を演じるのは論理シミュレーションである．この処理自体は，専用マシンやエミュレータによって，大幅に高速化されている．反面，シミュレーションの入力である検証プログラムの開発効率化は，ほとんど手を付けられていないのが実状である．

本論文は，検証プログラム開発効率化を目的とした，プロセサ制御のモデル化技法と，そのモデルの下で検証プログラムを自動生成する手法について述べる．

2 検証プログラム

2.1 検証プログラムとは

検証プログラムとは，プロセサの設計が仕様を満足しているかどうかを検証するために，設計モデルまたは実設計に流すプログラムであると定義する．

検証プログラムはプロセサの仕様に記された項目を確認する形で作られ，各項目が短いコードからなるブロックに対応する．一つのブロックは，条件設定部，実行部，実行結果の確認部の三つの部分から成る．例えば，分岐命令が遅延スロットを持ち，次の命令の実行後に分岐が起こるという仕様を考える．対応するブロックでは，分岐を起こす条件を設定し分岐命令を置く．遅延スロットには，実行したかどうかで結果が異なる命令を選ぶ．SPARC[7]の命令列を使って，これを表現すると図1のようになる．

```
        addcc   %g0,%g0,%g1     ! g1 = g0 + g0 = 0
        brz     %g1,out         ! if (!g1) then goto out
        mov     -1,%g1          ! g1 = -1
        ...
out:
        st      %g1,addr        ! Mem[addr] = %g1
```

図1: SPARCアセンブラによる検証プログラムの例

2.2 検証プログラム開発の問題点

従来，検証プログラムは実機への適用を前提として作られていたが，プロセサが複雑になると，論理的な設計誤りが見過ごされる可能性が高くなり，設計初期段階に検証プログラムを適用する必要が出てきた．ところが，検証プログラムへの要求事項が設計段階によって異なるため，実機用と設計初期段階用を分けて考えねばならない．

ここでは，設計初期段階用の検証プログラム開発を支援する方式を考える．設計初期段階用の検証プログラムに対しては，プログラムが短く，仕様変更に速やかに対処できることが要求される．

開発法から検証プログラムを分類する．

1. 設計者がチェック用に開発したプログラム
2. 検証プログラム専門の技術者が開発したプログラム
3. ランダムに生成したプログラム

最初のものは，設計者が設計に追われつつ開発しており，十分な検証能力を期待できない．また，設計者が想定しない状況でのみ露見する設計誤りには無力である．

次は，人間が仕様書から検証すべき項目を抽出し，それを実現する命令列を組み合わせたものである．例えば，図1の例では，分岐命令の仕様の中から「分岐命令は1命令分の遅延スロットを持ち...」という記述に着目し，「分岐命令の次の命令が実行されるかどうか」という検証項目を抽出して，図にある命令列を作る．本プログラムの検証能力は高いが，開発工数が大きいという問題を抱えている．更に，設計仕様は設計初期の段階で頻繁に変わることが多く，その度にプログラムの手直しが必要になるという欠点もある．

最後のランダムなプログラムは，単純なランダム命令列ではなく，発生させる命令の頻度やデータ，アドレスの範囲を制御するとともに，特定の機能を持った命令列ブロックを組み合わせた，ある程度意味のある命令列である．この生成方式は，ほぼ自動的に命令列を生成できるだけでなく，設計者が見落としていた状況を発生させる可能性があり，かなりの検証能力を持つ．難点としては，ランダム列の制御や命令列ブロックの選択に，プロセサ固有のノウハウがあり，新規開発のプロセサでは，命令列発生システムの開発コストがかかる点がある．

以上，それぞれの長所と短所を論じたが，この他に共通の問題がある．検証の網羅性を保証できない点である．実際のプロセサ開発の場では，設計誤りの発見件数を時間に沿ってプロットし，件数が落ち着いたら，次の設計段階に移るという形を取っている．この方法では，設計誤りを見過ごしている可能性が残るばかりでなく，きちんと検証した部分としていない部分の切り分けができない．今後のプロセサ設計では，テスト生成における故障検出率のような検証に関する明確な尺度を設定し，それを基にして検証の質の制御を行なうことが，不可欠である．

2.3 本研究のアプローチ

本研究は，論理検証用プログラムにおける，開発工数と網羅性の問題を解決するために，仕様から検証項目を抽出して，それを実現する検証プログラムを作る処理を自動化することを目指す．これにより，開発工数を削減するとともに，仕様変更への対処も極めて容易になる．

検証プログラム自動生成には仕様の明確化が不可欠である．設計者に対して，明確化のための負担を強いることになるが，非明示的な暗黙の仕様が設計誤りの大きな原因であることから，設計品質を向上させるための新しい設計方法論として受け入れられると期待している．仕様が明確にされれば，検証項目の母数を求めることができ，検証プログラムの網羅率を求めること，網羅率100%の検証プログラムを生成することも可能となる．

3 プロセサ制御のモデル化

3.1 プロセサのモデル

プロセサで多くの物量を占めるのはメモリと演算器だが，検証の観点からはさほど重要ではない．これらは規則的な構造を持っており，かつユニット単独の検証で十分だからである．これに対し，制御部は物量は少ないが，不規則な構造から成る論理の深い順序機械で構成され検証が難しい．検証プログラム生成のターゲットは，検証が難しい制御部とする．検証プログラム生成に限らず，プロセサの検証ではデータパス部と制御部の分離が必須である．

次に制御部を機能別に分類する．プロセサの重要な制御として，パイプライン，キャッシュ，仮想記憶，割り込みの各制御がある．これらはデータパス部の各ユニットを制御し，データパス部からのデータを受け，他の制御部と信号をやりとりする．

3.2 制御部のモデル

各制御部は論理の深い順序機械で構成される．このような装置の抽象的なモデルとして，ここでは，決定的有限状態マシンを採用する．

検証プログラムを生成するには検証項目を制御モデルの下で表現する必要がある．ここで人手開発の手順を思い起こしてみると，仕様に記された検証項目をチェックするとは，項目に対応する状況を作り正しく動作するかを調べることに相当する．すなわち，検証項目とは制御モデルにおける状態集合のうち，次に正しい状態遷移が起こるかが疑わしいもの，あるいは，制御モデルの有限状態マシンにおける「危険な状態」の集合であると言える．

3.3 汎用検証プログラム生成アルゴリズム

検証プログラム生成の入力は，有限状態マシンの形式の制御モデル，および検証項目に相当する危険な状態の集合である．このとき，検証プログラム生成のアルゴリズムは，像計算，逆像計算 [6] と呼ばれる有限状態マシンに対する処理を用いて容易に表すことができる．

有限状態マシンの状態集合に対して，次の時刻にそこから／そこに到達可能な状態集合を求める演算が像計算／逆像計算である．この演算を効率的に行なうには，有限状態マシンを特性関数や遷移関係関数で表現し [2]，BDD[1] を用いて計算する方法が知られている．

検証プログラムは，制御モデルの初期状態から危険な状態に到達させる命令列である．汎用アルゴリズムを図2に示す．ここでは，最初に初期状態から到達可能な危険な状態を像計算によって予め求めている点である．単に検証プログラムを求めるならば，各危険な状態から初期状態に至るまで逆像計算を繰り返せばよい．ところが，現実には初期状態から到達不可能な危険な状態が多く含まれており，ここに示したアルゴリズムの方が効率的である．

```
procedure VerificationProgramGeneration {
  Given a FSM: M;
  Generate characteristic functions F_i (1 ≤ i ≤ n);
    (F_i corresponds to the i−th "critical" states)
  F'(x) := 0; F(x) := F_I(x); (I:Initial states of M)
  while (F(x) ≠ F'(x)) F'(x) := Image(F(x));
  foreach i  (1 ≤ i ≤ n) {
    F_i(x) := F_i(x) · F(x);
    if (F_i(x) ≠ 0) {
      Trace :=< F_i(x) >;
      do {
        F'(x) := Image^{-1}(F_i(x));
        Trace := concat(F'(x), Trace);
      } while (F'(x) ∩ F_I(x) ≠ φ);
      Collect an input sequence Seq from Trace;
      Register Seq as a verification program for F_i;
    }
  }
}
```

図 2: 汎用検証プログラム生成アルゴリズム

3.4 パイプライン制御への適用

3.4.1 仕様と検証項目

本手法では仕様をモデル化しており，状態数を抑えるには検証に必要な仕様だけを抽出することが有効である．これは形式的検証における設計抽象化に近い考え方であるが，それよりも容易かつ系統的に検証仕様を抽出できる．

検証仕様の抽出には，検証項目を定義する必要がある．ここでは，パイプライン制御で最も重要なインタロック機構 [4] の検証を考える．インタロック機構は，ハードウェア資源の競合やデータ依存性の破壊が起きないようにステージを制御する．検証プログラムは，上記の状況が起こり得る直前の状態に到達させる命令列である．次の時刻で競合が起きないよう各ステージが制御されていれば，インタロック機構の設計が正しいと考えてよい．このとき，検証仕様として必要十分な情報は次の通りである．

- ステージの接続関係
- 命令が使用するステージとそのタイミング
- 命令が各ステージで利用するハードウェア資源

ここに示した仕様は，パイプラインを設計する上で最も基本的な情報であり，設計者が容易に提供できるものである．設計者側はこれらの情報を列挙すればよく，システム側で複数の命令が組み合わされた状況を作り出して，問題となる状態を列挙する．図 3に仕様の例を示す．

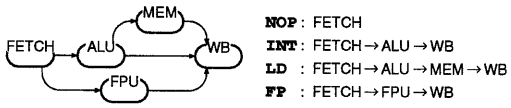

図 3: パイプライン仕様の例

3.4.2 有限状態マシンの生成

次にパイプラインを有限状態マシンに対応させる．パイプラインの状態は，各ステージにある命令とオペランドの組み合わせで定められる．今，簡単のため命令の組み合わせだけを考える[1]．図 3の例では，4 命令，5 ステージであり，$(4+1)^5 = 3125$ 状態となる．ところが，この作り方では状態数が爆発的に増大する．そこで，検証上区別の必要がない状態をマージさせる．パイプライン仕様の観点から同じ性質を持つ命令群，例えばフラグを立てない整数演算命令は同一の命令とみなせる．これにより，対象とすべき命令の数を大幅に削減でき，実際のプロセサでは，50 程度の命令を考えれば十分である．

更に，ステージを通り得る命令を考慮に入れると状態数を一層削減することができる．図 3の例で，FPU ステージを通る命令は 1 種類であるから，ここの取り得る状態は命令の有無を含め 2 種類となる．以上の方法により，図 3 の状態数は 240 $(= 5 \times 3 \times 2 \times 2 \times 4)$ となる．

3.4.3 危険な状態の算出

仕様の中に記されたハードウェア資源の使用状況を基にして，危険な状態を算出する．これには，ある資源を利用する命令を列挙して，同時に使う状況を考えた上で，その直前の状況を求める．これは単に命令を一つ前のステージに動かせば良い．データハザードに関しては，ある命令がデコードされてからレジスタに書き込むまでの間，その命令が書き込み先のレジスタ資源を占有するという形で実現できる．

[1] ハザードのうち，データハザード，制御ハザードを無視して構造ハザードだけを対象とすることと等価である．

次に，このようにして列挙された危険な状態で，初期状態から到達可能なものだけを抽出する．これには像計算を用いる．インタロック機構の検証では，到達できない危険な状態がかなり多くなるため，列挙された危険な状態それぞれについて逆像計算による検証プログラム生成を適用するよりも，この方が効率的である．

図 3の例における危険な状態を図 4に示す．このうち，9–12 の状態は初期状態から到達不可能である．

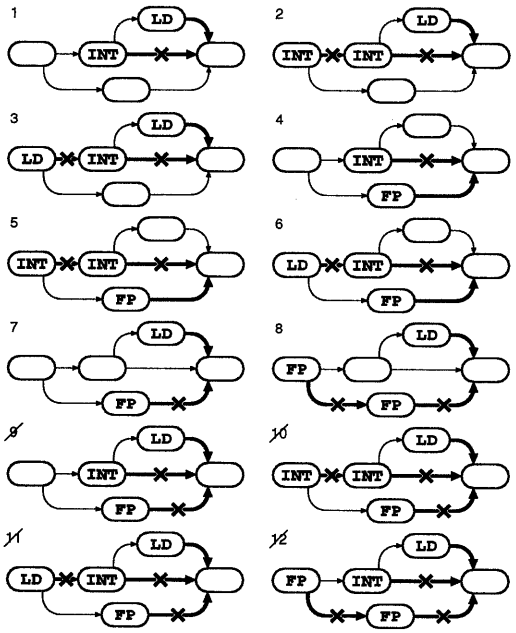

図 4: 危険な状態の列挙

3.4.4 検証プログラム生成

有限状態マシンと危険な状態の集合が求めた後，図 2のアルゴリズムで検証プログラムを生成する．現在のシステムは Perl と C を組み合わせた形で実装されており，パイプライン仕様記述用の独自言語から直接 BDD を用いた内部データに変換している．現在，有限状態マシン記述を間にはさむようにして，アルゴリズムのエンジン部分を汎用化する作業を進めている．表 1に 4 種類のパイプラインに対して検証プログラム生成を適用した結果を示す．このうち P1 は，図 3で示したものである．

表 1: 実験結果

例題	P1	P2	P3	P4
ステージ数	5	9	11	20
命令数	4	6	6	7
到達可能なケース	8	25	285	508
到達不可能なケース	4	36	212	3117
検証プログラム長	27	127	2516	6182
CPU 時間 （秒）	13	90	579	1028
到達可能な FSM 状態数	125	1.7e4	1.9e6	3.6e7

4 考察

4.1 形式的検証との対比

形式的検証手法に関する研究は，BDD[1] による論理関数の演算手法の劇的な改善のおかげで近年大きな進歩を遂げた．ところが，形式的検証をプロセサに適用した例は，キャッシュプロトコルの検証例 [3] を除き，皆無であった．プロセサでは内部レジスタの数が多く，状態数が膨大なことが原因と考えられる．これに対し，検証能力を落さずに状態数を減らす設計抽象化の研究が進められているが，大きな成果を挙げるに至っていない[2]．

我々のアプローチをこれらの手法と比べると，次のような相違点が見られる．

- 仕様だけをモデル化し，設計には手をつけない．
- シミュレーションを用い，完全性を問わない．
- 状態集合への到達可能性問題だけを解く．

一般に仕様は設計よりもコンパクトなため，状態数爆発の危険性は大幅に小さくなる．また，設計に手をつけないことから，設計検証の前処理がほぼゼロとなる．

一方完全性に関しては，生成された検証プログラムの持つ自由度をすべて網羅するような命令列を加えれば実現できるが，実際にはは不可能である．これは弱点ではあるが，多くのバグを設計に含む段階において，完全性を確かめる形式的検証を適用するのはコスト的に無駄であると考えている．我々の手法で十分にバグを取り，ある程度信頼できる設計データができた段階で，初めて形式的検証手法を適用するというのが現実的であろう．

[2]これに対してごく最近，記号シミュレーションを応用したプロセサ検証手法がいくつか提案された．

また，形式的検証システムの多くは，プロパティに時間の要素を含めることができるため，あることが無限回起こるとか，有限回しか起こらないといった条件を表すことができる．これは，プロトコル関係の重要な性質，例えばデッドロックの有無やライブネス，セイフティを調べる際に力を発揮する．これに対し我々の手法では，有限の状態遷移でチェックできるプロパティしか扱えない．また，形式的検証では，非決定的な振舞いをするシステムを扱えるが，検証プログラムを適用した際に所望の状態に遷移できるとは限らないため，検証能力に制限が付く．

4.2 本アプローチの可能性

現時点においては，本アプローチによりパイプライン制御におけるインタロック機構の検証を実現するとともに，マルチプロセサのキャッシュコヒーレンシプロトコルの検証への適用を検討しているに留まっている．しかし，有限状態マシンという汎用のモデルを確立したことで，プロセサの制御での検証項目が広がるとともに，それ以外の分野に適用できる見通しを得た．

実際の検証プログラム開発で，必ず検証項目として挙げられる以下の項目についても，注目した状況に対応する状態集合を求めることで，検証プログラムを生成できる．

- パイプラインに命令がある数以上詰まった状態
- リザベーションステーションが溢れた状態
- ストアバッファが溢れた状態
- バスで複数のユニットから同時に要求が来た状態

更に，レジスタ転送レベルの記述に対して，高位レベル合成の手法を適用して制御部を有限状態マシン，データパス部を適当なモジュールで表せたすると，元の記述で特定の条件が成り立つような状況に到達させる入力列を制限付で生成できる．制限とは，データパスの値を制御部で受けて条件判定に用いている場合に，その条件判定結果を外部から制御できる場合に限るというものである．制御部の内部で生成される値の場合は制限対象とはならない．応用としては，VHDL 記述の中の assert 文を活性化させる入力列生成などが考えられる．

5 おわりに

論理検証プログラム自動生成のための，制御部のモデル化技術とそれに基づく汎用検証プログラム生成アルゴリズムについて述べた．仕様を有限状態マシンにマッピングして，状態の到達可能性問題を解くという形で，これを実現している．本手法は極めて汎用性が高く，プロセサの制御機構の検証だけでなく，ハードウェア検証全般に渡って適用することが可能である．

形式的検証手法と比較して，本手法は完全性を保証できないものの，必要な情報はプロセサの検証仕様のみであり，有限状態マシンでモデル化した際に状態数爆発が起こりにくい．更に，設計にかかわらず検証プログラムを作れ，設計変更にすぐ対処できるのが強みである．

現在，パイプライン制御，キャッシュ制御に対する検証プログラム自動生成を実プロセサに適用する作業を続けている一方で，ここで示した汎用のモデル化技術を生かして，応用範囲を広げるべく検討を進めている．

参考文献

[1] R. E. Bryant. Graph based algorithm for boolean function manipulation. *IEEE Trans. Comput.*, C-35(8):677–691, 1986.

[2] 平石, 浜口. 論理関数処理に基づく形式的検証手法. 情報処理, 35(8):710–718, 1994.

[3] K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[4] D. A. Patterson and J. L. Hennessy. *Computer Organization & Design: The Hardware/Software Interface.* Morgan Kaufmann Publishers, 1994.

[5] R. Peck and J. Patel. Design methodology for a MIPS compatible embedded control processor. In *Proceedings of the IEEE International Conference on Computer Design*, pp. 324–328, 1991.

[6] H. J. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using BDD's. In *Proceedings of the International Conference on Computer-Aided Design*, pp. 130–133, 1990.

[7] D. L. Weaver and T. Germond. *The SPARC Architecture Manual.* Prentice-Hall, 1994.

盛光印刷所