設 計 自 動 化 71－9
（1994. 6. 24）

アルゴリズミックデバッグ法を用いた VLSI 上位仕様記述の検証法

長沼 次郎　小倉 武　星野 民夫

NTT LSI 研究所
〒243-01 神奈川県厚木市森の里若宮 3-1
Phone : 0462 40 2139
E-mail : jiro@nttica.ntt.jp

あらまし

本論文では、アルゴリズミックデバッグ法を用いた構造化分析法による VLSI 上位仕様記述の検証 (誤り発見) 法を提案し、その有効性を 3 つの実チップレベルの大規模な例題を用いて評価する。対象システムの動作仕様は、構造化分析法を用いて上位のレベルで記述し、その中に含まれる仕様記述の誤りを、アルゴリズミックデバッガからの極めて少ない問い合わせに答えるだけで効率的に発見する。設計者とデバッガの協調作業による誤り発見手数は、従来のシミュレーションを用いた検証法と比較して、1/10 ～ 1/100 に短縮される。本手法は、効率的な VLSI の上位仕様記述の検証法において重要な役割を果たすだろう。

High-Level VLSI Design Specification Validation Using Algorithmic Debugging

Jiro NAGANUMA　Takeshi OGURA　Tamio HOSHINO

NTT LSI Laboratories
3-1, Morinosato Wakamiya, Atsugi, Kanagawa, 243-01 JAPAN
Phone : +81 462 40 2139
E-mail : jiro@nttica.ntt.jp

Abstract

This paper proposes a new environment for high-level LSI design validation using "Algorithmic Debugging" and evaluates its benefits on three significant examples. A design is specified at a high-level using the structured analysis (SA) method and some errors included in SA specifications are efficiently located by answering just a few queries from the debugger. The number of interactions between the designer and the debugger is reduced by a factor of ten to a hundred compared to conventional simulation based validation methodologies. This environment promises to be an important step towards efficient high-level LSI design validation.

1 はじめに

近年、VLSIの適用領域の拡大により、VLSIに要求される機能や規模が増々複雑化、大規模化している。これに伴い、その設計TAT(turn around time)も増大している。設計TAT増大の一因は、VLSI設計の流れの最上流に位置する仕様分析、仕様理解、仕様記述の不完全性、困難性にある。

一方、高性能な市販の論理合成システム [1] の登場により、ハードウェア記述言語(HDL)を用いたレジスタトランスファレベル(RTL)のVLSI設計が普及してきた。また、VHDL [2]、OVI [3]、UDL/I [4] などのハードウェア記述言語の標準化により、RTLを用いたVLSI設計が加速されている。

VLSI設計におけるRTLによる仕様記述は、ゲートレベルのものと比較して、明らかに理解し易くなってきている。しかし、大規模なVLSIの設計においては、以下のような極めて重要な問題が残されている。

- 仕様記述(speficication)：実現すべき完全な機能の仕様分析、仕様理解、仕様記述の困難性；
- 設計検証(design validation)：シミュレーションツールのみを用いた設計の検証作業(誤り発見)の困難性

構造化分析法による手法 [5]-[9] やグラフィカルHDLによる手法 [10]-[16] は、上記の仕様記述に関する問題を緩和している。構造化分析法は、ソフトウェア工学 [17][18] に起源を発するもので、データ/制御フロー図、状態遷移図などの形式化された図的な表現を用いて対象システムの動作仕様を階層的に記述するものである。大規模なリアルタイムのソフトウェアシステムの開発に広く用いられている。構造化分析法の大きな特徴は階層的な性質であり、システムの動作はプロセスの階層で表現され、上位のプロセスはより下位のサブプロセスに展開される。設計者はこのような階層により、下位のサブプロセスの誤りを上位のプロセスへ反映させることが容易に行える。SADE [8] や、Express-VHDL [9] は、グラフィカルで会話的なシミュレーション環境や、論理合成可能なRTL記述を生成する機能を有する構造化分析ツールである。グラフィカルHDLの各種ツールも同様な環境を提供する。SpecCharts [10] は、システムレベルでの仕様記述を可能とし、論理合成可能なRTL記述とそのソフトウェアを生成する機能も有している。これらのツールは、RTLをベースとした仕様記述、シミュレーション、論理合成をサポートするものの、設計検証、特に仕様記述の誤り発見のためのサポートは十分ではない。

設計検証の問題は、実現すべきVLSIを表現するRTL記述の規模に起因している。設計者は、最終的なレイアウトデザインを得るため、論理合成やレイアウトツールにRTL記述を入力する前に、数千行を越える誤りのない完全なRTL記述を書き上げなければならない。従来の典型的なRTL記述の検証法においては、設計者はシミュレータを起動し、その実行トレースを行うことにより、シミュレーション結果を観測し、その結果と期待値との比較を行い、人手作業により設計(仕様記述)の検証(誤り発見)を行う。RTL記述の規模も複雑さも増大しているため、実行トレースの規模も期待値の数も増大している。このため、人手による設計の検証作業は、最も労力を要し、また、設計者の注意力の限界を越えた作業である。

一方、抽象度レベルの異なる2つの仕様記述の比較検証には、Event Pattern Mappings [19] と呼ばれる非常に効率的な手法があるが、ある上位レベルの仕様記述それ自身を検証するものではない。定理証明 [20] や記号処理 [21] をベースにしたフォーマルな手法は、VLSIの仕様記述の完全性を示すかもしれない。これらは、設計者がVLSIの仕様記述の完全性を判断するには有効であるが、仕様記述に誤りがある場合はほとんど役に立たない。さらに、フォーマルな手法は、BDD [22] などのような最近の技術によって適用可能な規模は変わりつつあるが、まだ実チップレベルの大規模な設計への適用は困難である。

このように、シミュレーションツールを用いたRTL記述の検証法、異なる2つの仕様の比較検証法、フォーマルな検証法などの手法は、上記の設計検証に関する問題に対しては、満足のいく解決法を与えていない。このため、RTLよりも抽象度の高いより上位のレベルでの仕様記述、設計検証をサポートする設計環境が不可欠である。

この設計検証の問題、特に仕様記述に含まれる誤り発見の問題を解決するため、我々は、自動デバッグ法 [23][24] と呼ばれるソフトウェア工学の有望な技術を、VLSIの上位設計の分野に導入する。自動デバッグの中でも特に成功している論理型言語 [25] のアルゴリズミックデバッグ法 [23] 用いる。アルゴリズミックデバッグ法は、設計者の検証(誤り発見またはデバッグ)作業を、デバッガとの協調作業により効率的にサポートする。

本論文では、SPEED (SPEcification drivEn VLSI Design) と呼ぶ、構造化分析法とアルゴリズミックデバッグ法を融合したVLSIの上位仕様記述と上位仕様検証のための新しい設計環境を提案し、その有効性を示す。まず、第2節では構造化分析法による仕様記述ツールと仕様検証ツールからなる本設計環境の概要を示す。第3節では構造化分析法による仕様記述法とそのアルゴリズミックデバッグ法を示し、第4節では本設計環境を3つの実チップレベルの大規模な例題に適用した結果を示す。

2 システム概要

SPPEDの設計環境は、図1に示すように、以下の3つの部分から構成されている。

1. 対象システムの動作の仕様記述は、Hatley [18] による構造化分析法に基づいて上位のレベルで記述される(図1左上)。
2. 仕様記述の誤りは、アルゴリズムデバッガにより、上位のレベルで効率的に行われる。誤り発見手数は、従来の検証法と比較して、1/10〜1/100に短縮される(図1右上)。
3. 検証された構造化分析法に基づく仕様記述は、論理合成可能なRTL記述に自動変換される(図1下)。

構造化分析法とアルゴリズミックデバッグ法は、第3節で詳細に説明する。構造化分析法に基づく仕様記述から、論理合成可能なハードウェア記述言語(RTL 記述) [4][26] への変換は、既に開発されている [27]。構造化分析法に基づく仕様記述は、RTL 記述生成ツールの入力として用いられる。また、生成された RTL 記述は、論理合成ツール [28][29] により、FPGA(図 1)、ゲートアレイ、スタンダードセルなどにマップされたゲートレベルの記述 (ネットリスト) に論理合成することができる。

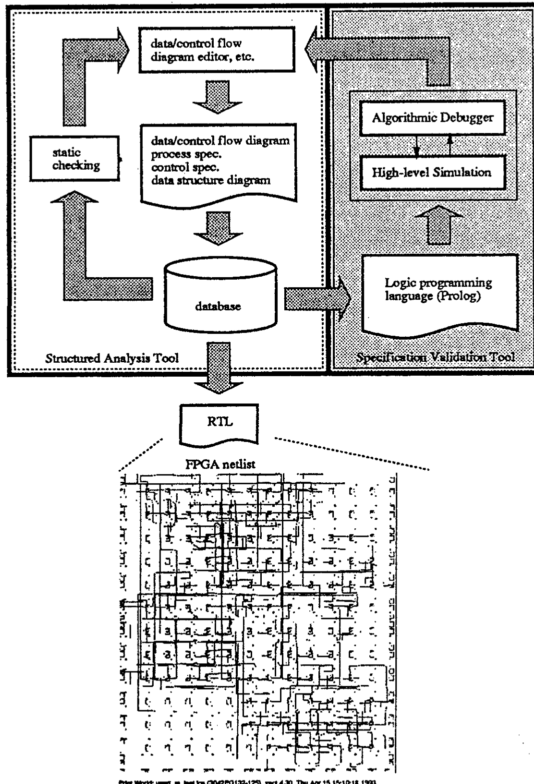


図 1: 仕様記述・検証システムの全体構成

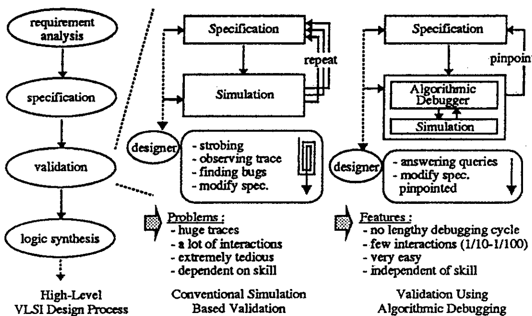


図 2: SPEED 設計環境の特徴

SPEED の設計環境には2つの大きな利点がある。

- アルゴリズミックデバッグ法は図 2に示すように、(1) 実行トレースの観察、(2) バグの分析究明、(3) 仕様の修正、といった従来のデバッグサイクルを不要にする。さらに大規模な仕様記述の中から誤りの位置を特定できる。
- アルゴリズミックデバッグ法はシミュレーションに基づく手法なので、対象システムの仕様記述の規模の制限はほとんどない。実際のチップまたはシステムレベルの仕様記述の検証に適用できる。

3 構造化分析法とアルゴリズミックデバッグ法

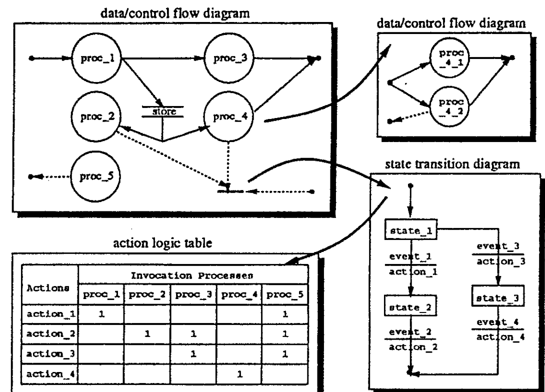
3.1 構造化分析法に基づく仕様記述

Hatley [18] の構造化分析法では、(1) プロセス間のデータと制御の関係を記述するデータ / 制御フロー図、(2) 逐次的に実行されるプロセスの動作を記述する状態遷移図、(3) 並列的に実行されるプロセスの動作を記述するアクションロジック表を用いて、対象システムの動作仕様を階層的に記述する。各プロセスは同様な方法で記述される下位のサブプロセスから構成される。各プロセスの基本モデルを付録 A の図 7に示す。

データ / 制御フロー図: データフロー (実線矢印)、制御フロー (破線矢印)、プロセス (○シンボル)、制御スペックバー (横棒)、ストア (平行線) からなる。プロセスは処理本体、データ / 制御フローはプロセス、ストア等のリソース間のデータと制御の流れを示す (図 3左上)。

逐次的な動作: 逐次的な動作は状態の遷移に伴って順次起動するアクション (複数のプロセス) として表現される。各状態遷移は、実行可能なイベントを持つアクションが実行される (図 3右下)。

並列的な動作: 並列的な動作は、同一の発火規則に持つアクションロジック表上の複数のプロセスとして表現される。各アクションは、起動マークを持つ複数のプロセスを並列に起動する (図 3左下)。



Actions	Invocation Processes				
	proc_1	proc_2	proc_3	proc_4	proc_5
action_1	1				1
action_2		1	1		1
action_3			1		1
action_4				1	

図 3: 構造化分析法の概要

3.2 構造化分析法の図的な仕様記述から Prolog への変換

逐次 / 並列動作を含み階層的に記述された対象システムの動作仕様は、1 対 1 に対応付けられた Prolog のソースコードに変換される。構造化分析法のプロセスの基本モデルと Prolog での表現、および、図的な仕様記述と対応する Prolog への変換規則を付録 A、付録 B にそれぞれ示す。このような 1 対 1 に対応付けられた完全な変換により、構造化分析法の仕様記述の完全なセマンティクスを与えるだけでなく、アルゴリズミックデバッグ法を用いて構造化分析法の仕様記述の検証 (誤り発見) を可能にする。特に、この 1 対 1 の双方向の対応付けにより、アルゴリズミックデバッグ法による Prolog ソースコードのエラーメッセージを構造化分析法の仕様記述の誤り発見に反映することできる。

この変換後は、対象システムを記述した構造化分析法の仕様記述は、Prolog の節の集合として表現される。Prolog の節は、ヘッド部とボディ部から構成されており、ヘッド部は、節の名前 (述語名) と引数から構成される。述語名は、固有の識別子で、プロセス名、状態遷移図の構成要素の種類、アクション名などに対応している。ヘッド部の引数は、プロセスのストアなどの内部状態、入出力などに対応している。またボディ部は、下位の節の呼び出しから構成されており、下位のサブプロセスに対応している。

3.3 アルゴリズミックデバッグ法を用いた構造化分析法の仕様記述の検証

前節で示した構造化分析法の仕様記述と Prolog ソースコードの 1 対 1 の対応付けにより、典型的な Prolog 処理系 [30] 上に作成された標準的なアルゴリズミックデバッガ [23] を用いることにより、構造化分析法で記述された対象システムの動作の仕様記述を検証することができる。アルゴリズミックデバッグ法を用いた構造化分析法の仕様記述の検証 (誤り発見) の概念を図 4に示す.

構造化分析法の仕様記述の検証 (誤り発見) は、仕様記述のトップの節にその引数の値を与え、それを具体化することにより、すなわち、アルゴリズミックデバッガに Prolog のゴール (問い合わせ) を与えることにより実現される。

アルゴリズミックデバッグ法は、図 4左側に示すように、構造化分析法のプロセスなどの図的な構成要素の動作を表している節を階層的に実行することにより適用される。アルゴリズミックデバッガは、全体の中から特定の節の実行を選択し、その入出力を設計者に問い合わせる。この問い合わせは、全体の計算における部分計算の結果と見ることができる。

アルゴリズミックデバッグ法 (“divide-and-query”) [23] の基本的な動作は、図 4右側に示すように、Prolog のゴールリダクション過程を示す計算木上での誤った節の実行部分を 2 分探索法でサーチすることである。

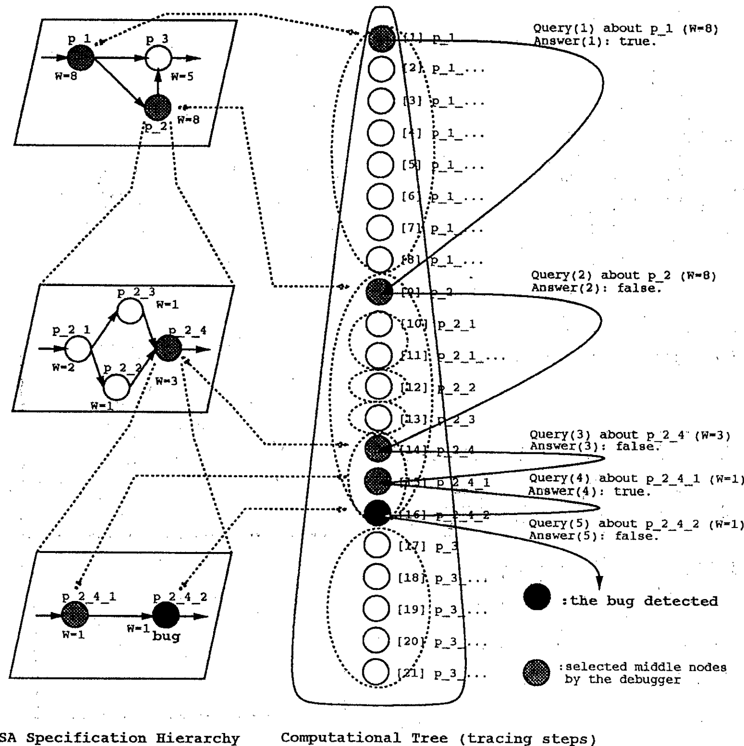


図 4: アルゴリズミックデバッグ法を用いた構造化分析法の仕様記述の検証

計算木上の各ノードには、それ以下の部分木の大きさを反映したリダクションの重さが定義されている。アルゴリズミックデバッガは、まず、ゴール(G)の実行をシミュレートするとともに、その全体の重み(w)を計算する。次にデバッガは、計算木の部分木の中から、その重みが$w/2$以下で最も重いノードQ (その重さw_Q)を選択し、その部分計算の結果が"真"か"偽"かを設計者に問い合わせる。例えば、デバッガは、まずトップの階層で、最も重いプロセス(w_Q=8)である"p_1"を一回目の問い合わせとして選択した。もし、設計者の答えが"真"ならば、デバッガは、同一階層の計算木の他の部分木を調べる。例えば、設計者は一回目の問い合わせ"p_1,"で"真"と答えたので、デバッガは、同一の階層(トップ)で、次に最も重いプロセス(w_Q=8)である"p_2"を二回目も問い合わせとして選択した。もし、設計者の答えが"偽"ならば、デバッガは、そのQをルートとする計算木の内部の部分木を調べる。例えば、設計者は二回目の問い合わせ"p_2,"で"偽"と答えたので、デバッガは、下位の階層(ミドル)で、最も重いプロセス(w_Q=3)である"p_2_4"を三回目の問い合わせとして選択した。三回目、四回目の問い合わせと答えも同様にして行われる。デバッガは、重さ(w_Q)が"1"の時に"偽"となった節を、誤りを含んだ節として指摘する。例えば、設計者は四回目の問い合わせ、すなわち重さ(w_Q)が"1"のプロセス"p_2_4_2"で"偽"と答えたので、デバッガは"p_2_4_2"の中に誤りを発見した。このようにして、デバッガは、仕様記述の誤りを設計者との協調作業により、2分探索法により効率的に探索する。

デバッガから設計者の発せられる問い合わせの数は、計算木の長さを"N"とすると、"$O(logN)$"であることが知られている[23]。

4 仕様記述の検証(誤り発見)の評価

4.1 構造化分析法による仕様記述

構造化分析法による仕様記述の例題として、3つの実チップレベルの大規模なシステムを取り上げた。

(1) Intel's 8251 チップとして実現されている通信制御プロセッサ(USART) [31])、

(2) 2ステージ(フェッチと実行)のパイプライン化された8ビットCPU (8bit CPU) [32]、

(3) 階層的なパイプライン処理と連想メモリを用いたバックトラック処理を実現したPrologマシン(ASCA) [33]。

3つの例題の仕様記述のサイズを表1に示す。また、Xウィンドウ上での実際の構造化分析ツールを用いた仕様記述の入力イメージ(USART)を付録Cに示す。

表1: 仕様記述量と変換されたProlog節の数

	number of hierarchical processes	number of bottom processes	number of transformed Prolog clauses
USART	6	21	109
8-bit CPU	29	28	160
ASCA	8	17	191

4.2 対象システムのシミュレーション

対象システムの状態遷移レベルの動作シミュレーションは、構造化分析法の仕様記述から変換されたPrologソースコードをProlog処理系で実行させることにより実現できる。上記3つの例題における変換されたPrologの節の数を表1に合わせて示している。変換されたPrologソースコードを用いたシミュレーションにおける状態遷移レベルの実行速度は、ひとつの状態遷移を4つのPrologのLI (logical inference)で実現しているため、Prolog処理系の実行速度の単位であるLIPS (logical inference per second)値の約1/4となる。ワークステーション上の1000 KLIPS (kiro logical inference per second)のProlog処理系を用いると、1秒間当たり、約250,000回の状態遷移がシミュレーションできる。このレベルの実行速度性能は、上記3つの例題のような実チップレベルの大規模な対象システムのシミュレーションを1秒以内に実行することができる。

4.3 仕様記述の検証(誤り発見)

4.3.1 誤った仕様記述

仕様記述の検証(誤り発見)を行うため、3つの例題の仕様記述に以下のような誤りを挿入した。

(1) USART　初期化プロセスの中のコマンド書込みプロセスにおいて:
正: 入力8ビットをコマンドレジスタに書き込む、
誤: 入力8ビットをモードレジスタに書き込んだ。

(2) CPU8　オペランドフェッチプロセスにおいて:
正: プログラムカウンタ値のプログラムRAMの内容をオペランドレジスタに格納する、
誤: プログラムカウンタ値をオペランドレジスタに格納した。

(3) ASCA　ユニフィケーションプロセスにおいて:
正: callerとcalleeの両方の変数のdereference処理を行う、
誤: callerの変数のみのdereference処理を行った。

4.3.2 アルゴリズミックデバッグ法の実際例

3つの誤り含む仕様記述をアルゴリズミックデバッガを用いて検証(誤り発見)する。図5は、USARTの初期化プロセスのアルゴリズミックデバッグの過程を示している。

先の3.3節で示したように、デバッガは、Prologの節の実行を行い、その部分計算の結果を設計者に問い合わせる。この例では、図5に示すように、"c_spec_wait_mode"、"c_spec_complete_mode"、"write_command"プロセスの問い合わせがデバッガによって選択されている。

設計者はデバッガからの部分計算の結果の問い合わせに対して、それが"真(期待通り)"か"偽(期待外れ)"かを答える。有限回の問い合わせの後に、デバッガは特定の誤りを含んだPrologの節をレポートする。図5に示す例では、"write_command"プロセスが誤ったPrologの節として指摘されている。アルゴリズミックデバッグ法により発見された

誤りを含むPrologの節は、構造化分析法による仕様記述とPrologソースコードの1対1の対応付けにより、容易にオリジナルの構造化分析法による仕様記述の図的な構成要素に関連付けられる。

4.3.3 実験結果と考察

先節に示した例では、全実行トレース数(計算木の長さ)は“12”であるが、わずか“3”回の問い合わせだけで、デバッガは誤りを含むPrologの節を発見している。他の例題における、全実行トレース数と仕様記述の中の誤りを発見するまでにデバッガが発する問い合わせ数の関係を表2に示す。また、アルゴリズミックデバッグ法のより大規模な例題に対する特性を評価するため、ASCAの例題で結合するリスト長(Ln)を変化させて、全実行トレース数を増加させ、より大規模な例題を模擬した。この場合の全実行トレース数と誤り発見までの問い合わせ数の関係を表3に示す。

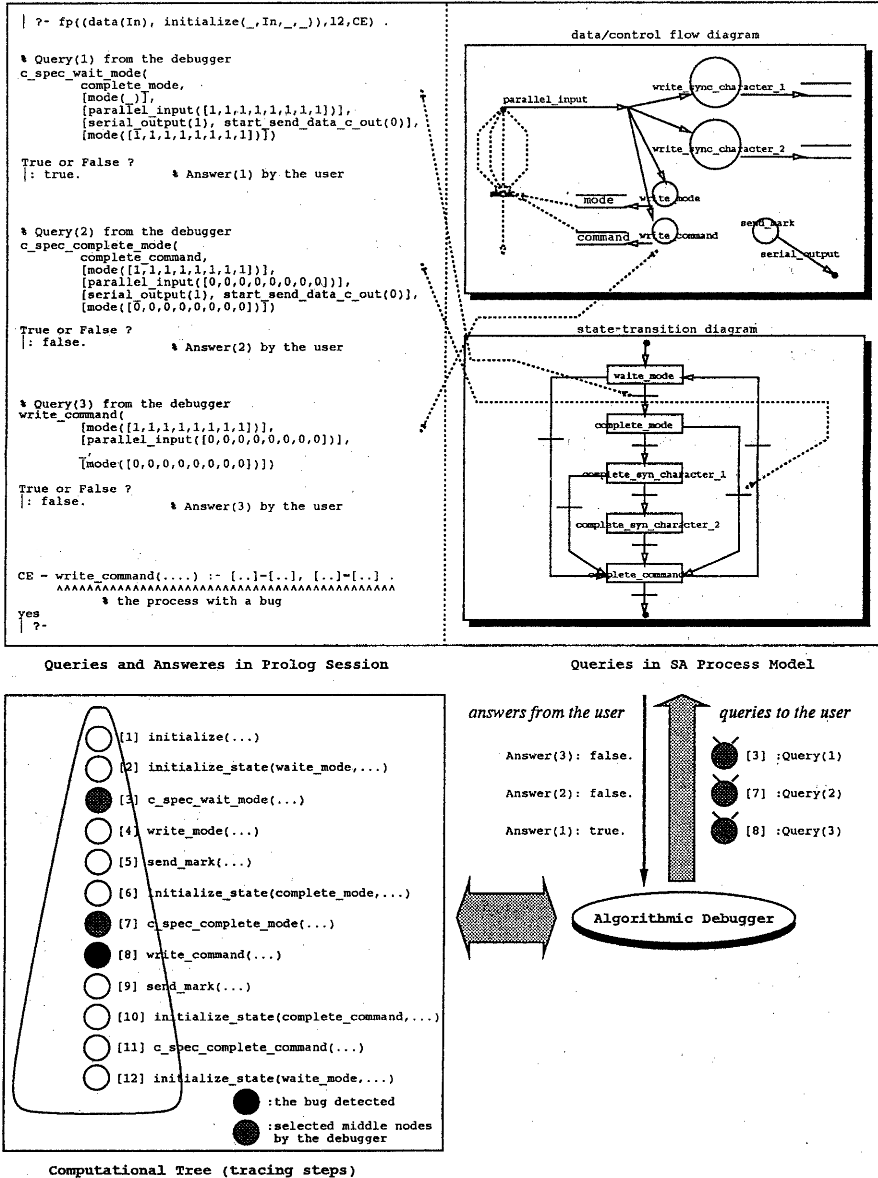


図5: USARTの初期化プロセスの検証(誤り発見)例

誤った動作が発見された位置を示す計算木上の深さ(誤り発見深さ)を表 2と表 3に合わせて示している。この誤り発見深さは、従来のシミュレーションの実行トレースを用いた検証(誤り発見)法において、仕様記述の誤りを発見するまでに必要な最小ステップ数を示している。

この誤り発見深さは(従来の手法)、仕様記述の構造と誤りの存在する位置に大きく依存しており、計算木の長さ(全実行トレース数)を"N"とすると"$O(N)$"となる。これに対し、アルゴリズミックデバッグ法における問い合わせ数は、仕様記述の構造や誤りの存在する位置にはあまり依存せず、理論的に"$O(logN)$"となる。このような従来と我々の手法における実行トレース規模と誤り発見手数の関係を明らかにするため、表2と表3の同一のデータを用いたグラフを図6に示す。X軸は全実行トレース数、Y軸は誤り発見手数を示しており、両軸は対数のスケールである。誤り発見手数として、従来の手法は誤り発見深さを、我々の手法は全問い合わせ数を示している。図 6に示すように、実際の実験結果は、理論的な背景を良く反映している。

このように、各例題に含まれていた構造化分析法の仕様記述の誤りは、デバッガからの極めて少ない問い合わせに答えるだけで、効率良く発見することができる。設計者とデバッガの協調による誤り発見手数は、従来のシミュレーションを用いた検証法と比較して、1/10 〜 1/100 に短縮される。

我々の手法と従来の手法の最も重要な違いは、実行トレースにおいて期待しない誤った動作を見つけてから、全体の仕様記述の中から誤りの部分を正確に特定する (pinpoint) 方法である。従来の手法では、設計者にとって、大規模な仕様記述の中から誤り部分を特定することは難しい作業である。誤った動作を見つけてから仕様記述の誤りを正確に特定することは、設計者のスキルに大きく依存する。

これに対し、アルゴリズミックデバッガは特定の誤ったPrologの節を指摘し、その発見された誤ったPrologの節は、(a) 対象システムの動作を階層的に表した計算木と (b) 構造化分析法による仕様記述とPrologソースコードとの1対1の対応付けにより、容易にオリジナルの構造化分析法による仕様記述の図的な構成要素に関連付けることができる。

表 2: 全実行トレース数と問合せ数

	total tracing steps	bug-detection depth	number of queries
USART(ini)	12	8	3
8-bit CPU	73	15	9
ASCA(L30)	746	668	7

表 3: より大規模な例題における特性

ASCA(Ln)	total tracing steps	bug-detection depth	number of queries
L5	146	118	5
L10	266	228	6
L20	506	448	7
L30	746	668	7
L50	1226	1108	8
L70	1706	1548	9
L100	2426	2208	9

"goal: ?-append([1,2,3,..,Ln],[],X)."

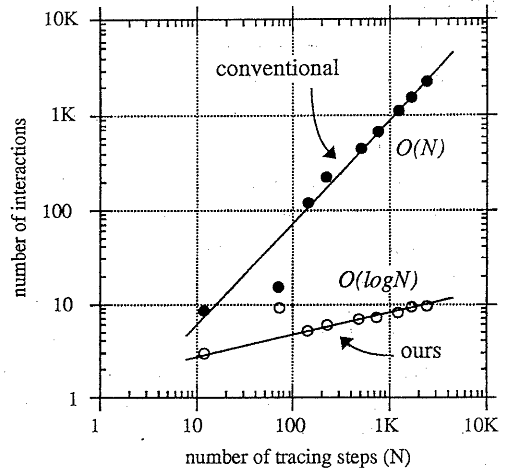


図 6: 誤り発見手数の実行トレース規模の関係

このように、SPEEDの設計環境は、VLSI上位仕様記述の検証(誤り発見)作業を効率的にサポートする。設計者は、労力を要するRTLによる仕様記述の検証(誤る発見)作業から解放され、対象のVLSI設計自身に専念することができる。

5 おわりに

本論文では、SPEEDと呼ぶ構造化分析法とアルゴリズミックデバッグ法を融合したVLSIの上位仕様記述と上位仕様検証(誤り発見)のための新しい設計環境を提案し、その有効性を3つの実チップレベルの大規模な例題を用いて評価した。3つの大規模な例題に含まれる誤りは、デバッガからの極めて少ない問い合わせに答えることにより極めて容易に発見された。設計者とデバッガの協調作業による誤り発見手数は、従来のシミュレーションを用いた検証法と比較して、1/10 〜 1/100 に短縮された。検証された構造化分析法による仕様記述は、論理合成可能なRTL記述に自動変換できる。SPEEDの設計環境は、設計者のLSI上位仕様記述の検証(誤り発見)作業を効率的にサポートする。今後、このSPEED設計環境をベースとして、構造化分析法を中心としたVLSI上位仕様からの性能評価法 [34]、ハード / ソフトコデザイン環境の検討を進めていく。

謝辞

本検討を進めるにあたり、終始御指導、御助言頂いた酒井保良 NTT LSI研究所長、唐津 修 設計システム研究部長に感謝致します。また、初期の論文の構成に関して有益なコメントを頂いたDr. Benoit A. Gennart、および、種々有益な御討論を頂いた安達 徹 リーダ, 武田 和光 リーダをはじめとした設計システム研究部の皆様に感謝致します。

参考文献

[1] "Design Compiler Reference Manual (Version 3.0)," Synopsys, Inc., Dec., 1992.

[2] "IEEE Standard 1076-1988, VHDL Language Manual," IEEE 1988.

[3] "Verilog HDL Language Reference Manual (LRM) Version 1.0", Open Verilog International, Nov., 1991.

[4] T. Hoshino, "UDL/I Version Two: A New Horizon of HDL Standards," in proc. CHDL93, Apr., 1993. "UDL/I Language Reference (Version One)," UDL/I Committee, JEIDA, Mar. 1990.

[5] T. Tikkanen et al., "Structured Analysis and VHDL in Embedded ASIC Design and Verification," in proc. EDAC90, Mar. 1990.

[6] M. P. J. Stevens and F.P.M. Budzelar, "System Level VLSI Design," North-Holland, Microprocessing and Microprogramming 30, pp. 321-330, 1990.

[7] . W. Glunz and G. Venzl, "Using SDL for Hardware Design," in Proc. 5th SDL Forum, Elsevier Science Publishers, 1991.

[8] J. Lahti et al., "SADE: A Graphical Tool for VHDL-based System Analysis," in Proc. ICCAD'91, pp. 262-265, Nov. 1991.

[9] D. Harel et al, "Statecharts: A Working Environment for the Development of Complex Reactive Systems," IEEE Trans. Software Engineering, vol. SE-16, no. 4, pp. 403-413, Apr. , 1990. "Express-VHDL Reference Manual," i-Logix Inc. 1992.

[10] S. Narayan et al., "System Specification and Synthesis with the SpecCharts Language," in Proc. ICCAD'91, pp. 266-269, Nov. 1991.

[11] "N.2 User's Manual," TD Tech. Inc. 1991.

[12] "flow-HDL:," Knowledge Based Silicon Corporation, 1992.

[13] 黒木他, "機能設計支援システム OZ における階層設計手法," 第45回情処全大, 2K-4, pp. 6-23, Oct. 1992.

[14] 松本他, "グラフィカル入力による動作機能設計の一手法," 情処研報, 93-ARC-98, vol. 93, no. 6, pp. 73-80, Jan. 1993.

[15] "Visual HDL," SEE Technologies, 1993.

[16] "SPeeDCHART," SPeeD Electronic, Inc. 1993.

[17] T. DeMacro, "Structured Analysis and System Specification," Yourdon Press, N.Y., 1985.

[18] D. J. Hatley and I. A. Pirbhai, "Strategies for Real-Time System Specification," Dorset House Publishing Co., Inc., 1988.

[19] B. A. Gennart and D. C. Luckham, "Validating Discrete Event Simulations Using Event Pattern Mappings," in Proc. 29th Design Automation Conference (DAC), pp. 414-419, June, 1992.

[20] M. Gordonr, "Why higher order logic is a good formalism for specifying and verifying hardware," G. Milne and P. A. Subrahmanyan (Eds), Formal Aspects of VLSI Design, North-Holland, pp. 153-177, 1986.

[21] J. R. Burch et al., "Sequential circuit verification using symbolic model checking," in Proc. 27th Design Automation Conference (DAC), pp. 46-51, June, 1990.

[22] R. E. Bryant, "Graph-based Algorithms for Boolean Function Manipulation," IEEE Trans. Comput. vol. C-35, no. 8, pp. 667-691, Aug., 1986.

[23] E. Y. Shapiro, "Algorithmic Program Debugging," The MIT Press, 1983.

[24] B. Korel, "PELAS – Program Error-Locating Assistant System," IEEE Trans. Software Engineering, vol. 14, no.9, pp. 1253-1260, Sep, 1988.

[25] W. F. Clocksin and C. S. Mellish, "Programming in Prolog," Springer-Verlag, 1981.

[26] Y. Nakamura, "An Integrated Logic Design Environment Based on Behavioral Description," IEEE Trans. CAD, vol. CAD-6, no.3, Mar., 1987.

[27] 長沼他, "構造化分析法を用いた LSI 仕様記述・検証法の検討," 情報処理学会、DA シンポジウム '92, pp. 81-84, Aug. 1992.

[28] "UDL/I Synthesizer (Version One)," UDL/I Committee, JEIDA, Sep. 1993.

[29] K. Oguri et al., "Evaluation of Behavior Description Based CAD System Used in Prolog Machine Design," in Proc. ICCAD'86, pp. 116-119, Nov. 1986. "Parthenon User's Manual," NTT Data, 1990.

[30] M. Carlsson and J. Widen, "SICStus Prolog User's Manual (version 0.7)," Swedish Institute of Computer Science, July 1990.

[31] "Microcommunications Handbook," Intel, 1988.

[32] "論理設計 CAD に関する調査報告書," 日本電子工業振興協会, 1986.

[33] J. Naganuma et al., "High-Speed CAM Based Architecture for a Prolog Machine (ASCA)," IEEE Trans. Comput. vol. 37, no. 11, pp. 1375-1383, Nov. 1988.

[34] 長沼他, "構造化分析法による LSI 上位仕様からの性能評価法の検討," 第48回情処全大, 5B-6, pp. 6-79, Mar. 1994.

[35] 磯田他, "設計情報とコードの一体管理方式に基づくソフトウェア開発支援システム (Soft DA/SA)," NTT R&D, Vol.38, No.11, 1989.

付録 A:
構造化分析法の基本モデルと Prolog での表現

構造化分析法では、逐次 / 並列動作を含む対象システムの動作仕様を、構造化分析法の複数のプロセスを用いて階層的に記述する。構造化分析法の個々のプロセスの基本モデルと Prolog での表現を図 7と図 8に示す。

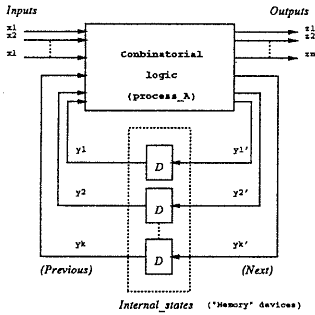


図 7: 構造化分析法のプロセスの基本モデル

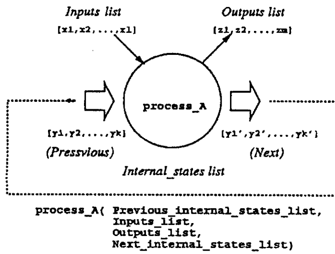


図 8: Prolog での表現

付録 B:
Prolog への変換規則

構造化分析法での動作の制御を記述する状態遷移図とアクションロジック表とそれらの Prolog への変換規則を図 9と図 10に示す。

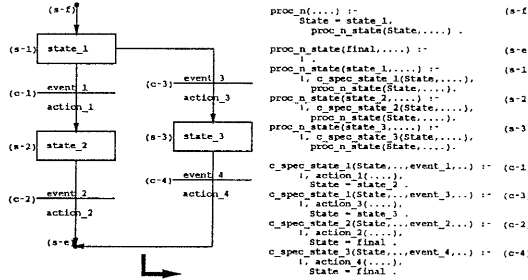
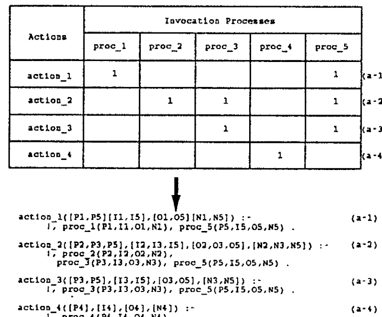


図 9: 状態遷移図



Actions	Invocation Processes					
	proc_1	proc_2	proc_3	proc_4	proc_5	
action_1	1				1	(a-1)
action_2		1	1		1	(a-2)
action_3			1		1	(a-3)
action_4				1		(a-4)

```
action_1([P1,P5][I1,I5],[O1,O5][N1,N5]) :-                         (a-1)
     !, proc_1(P1,I1,O1,N1), proc_5(P5,I5,O5,N5) .

action_2([P2,P3,P5],[I2,I3,I5],[O2,O3,O5],[N2,N3,N5]) :-          (a-2)
     !, proc_2(P2,I2,O2,N2),
       proc_3(P3,I3,O3,N3), proc_5(P5,I5,O5,N5) .

action_3([P3,P5],[I3,I5],[O3,O5],[N3,N5]) :-                       (a-3)
     !, proc_3(P3,I3,O3,N3), proc_5(P5,I5,O5,N5) .

action_4([P4],[I4],[O4],[N4]) :-                                   (a-4)
     !, proc_4(P4,I4,O4,N4) .
```

図 10: アクションロジック表

付録 C:
構造化分析法の仕様記述の入力イメージ

Hatley の構造化分析法をサポートする入力ツールとして、Soft DA/SA ツール [35] を用いている。X ウィンドウ上での本入力ツールを用いた USART の初期化プロセスの入力イメージを図 11に示す。データ / 制御フロー図、状態遷移図、アクションロジック表の各入力エディタ群が、"xeyes"、"login" ウィンドウなどの X ウィンドウのユーティリティとともに表示されている。

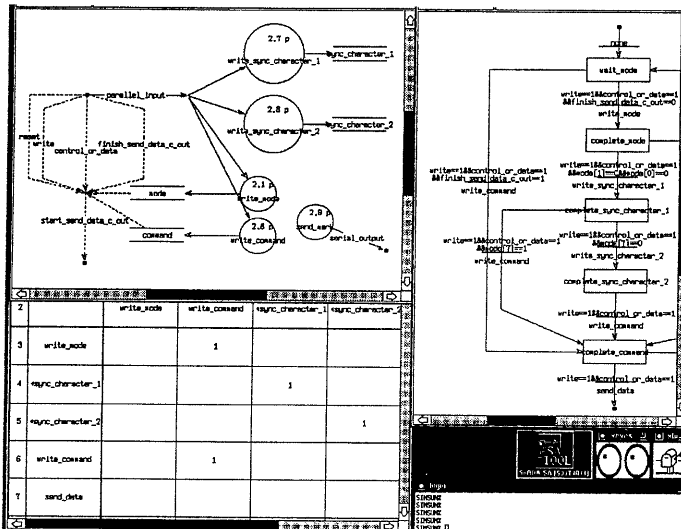


図 11: X ウィンドウ上での構造化分析法の仕様記述の入力イメージ (USART)