時相論理とProlog を用いた ゲート回路の効率的検証

藤田 昌宏　西山 智　田中 英彦　元岡 達
（東京大学　工学部）

1. はじめに

近年、計算機システムは益々大規模・複雑化されており、また社会の様々なところで重要な位置を占めるようになっている。このため、ハードウェアの設計においてもその信頼性が特に要求されており、実装段階だけでなく、論理設計の誤りを防ぐための計算機による支援が不可欠となっている。現在実用となっている論理設計検証手段は、シミュレーションによるものがほとんどである［1］。シミュレーションは誤設計の発見には有効であるが、それが存在しないことを示すことは、計算機スピード・スペース及び、見落しがでる等の点で難しい。シミュレーション専用ハードウェアを作成したり、適切なハードウェア記述言語を用いてうまいモデルを設定することで解決しようという試み［2、3］があるが、単なるシミュレーションである限り見落しがないとは言いきれない。

そこで、定式的に検証しようとする試みとして、定理証明のようにして検証する方法や、ソフトウェアの検証手法を応用する方法等幾つか研究されている。前者は設計記述を定理証明における仮定と考え、検証すべき仕様を目標の定理として、計算機支援のもとで証明していくものである。一階述語論理に対するproof checker を用いてＴＴＬ・ＩＣからできている回路を検証した例［4、5］がある。しかし現在のところ、ある程度人が証明手順を計算機に示してやらないとうまくいかない。また、後者は大きく分けて2つある。1つは、プログラムの最初と最後、それにループがある場合にはその中に少くとも1つの表明を記述し、それらの間を {pre-conditiond} execution {post-condition} の形でとらえ、それぞれ検証する方法である［6、7］。しかし、この方法は仕様の与え方が簡単ではなく、また並列動作をするハードウェアを検証するには何らかの工夫が必要となる。もう1つの方法は、検証すべき設計記述に対し記号シミュレーションを行ってその結果が仕様と一致したか否かを調べものである。実際には、レベルの異なる2つの記述（例えば、状態遷移図とゲート回路）の両方を同じ条件で記号シミュレーションを実行し、結果が等しいか否かを調べる［8］。これも仕様は通常の論理式で与えなければならないため並列動作の記述がやりにくく、また検証は2つの論理式の比較の問題となり計算機で円滑に行えるとは限らない。

これら以外に、並列に動作するものの記述を容易に行うために、時相論理（Temporal Logic）を用いて仕様記述を行い検証する手法が研究されている［9、10］。しかしこれは、もともと並行プロセスの検証のために研究されていたものの応用であり、ハードウェアに対するものは少く［11、12、13］、まだ進んでいない。

そこで我々は、まず仕様記述の方法として時相論理を用いて各信号間の時間関係を記述することを考え、また検証については時相論理の決定手続きを用いて背理法で証明する手法を提案し、合せてこの手法がProlog を用いて容易に実装できることを示した［14、15］。さらに、比較的小規模のハードウェアに対しては、仕様から状態遷移図の自動合成を現実的時間で行える手法についても示した［16］。しかしこの検証手法では、与えられた設計記述全体をそのまま用い、必要な全ての場合についてしらみつぶしに調べているため、ハードウェア規模に対し検証時間は最悪の場合には指数的に増大する。

ここでは、実用規模のハードウェアの検証を可能とするための効率化手法とそのProlog による実装法について述べる。まずハードウェアシステムを実際に論理・算術演算を行う演算部と、各演算間のデータ転送のタイミングを制御し同期をとる同期部に分け、演算部は述語論理の範囲で仕様を記述し定理証明法による検証を、また演算部については命題論理の範囲で記述し必要な全ての場合を自動的に調べることにする。一般に演算部では細かいタイミングはあまり問題とならず、定理証明法［4、5］で行う際にも設計者自身による証明の誘導も比較的容易である。一方同期部は並列に動くもの全体を考慮しなければならず、人にとっても誤りを起こしやすいため、自動検証が特に望まれる。

本論文ではゲート回路による設計記述に対し、その同期部の検証について考える。同期部の検証は、そのままでは回路規模に対し最悪指数的に増大するが、

①仕様記述の仕方を工夫する、
②与えられた回路そのままではなく、検証に対し真に必要な部分のみを抽出する、
③同じことを何度も処理しないように1度処理したことを記憶する、
④外部環境に対する条件をうまく利用して調べる場合の数を減らす、

等の工夫を行うことにより、調べなければならない場合の数はかなり抑えられる。ここでは、実際にＨＳＬ［18］で記述された回路に対する検証システムをProlog を用いて実装し、実用規模の回路に対しても現実的時間で検証ができることを示す。

2章では時相論理とそのハードウェア仕様記述への応用及び、検証や自動合成［16］を行い易くするような仕様記述法について述べる。3章ではもととなる検証手法及び、実用規模の回路の検証を可能とするための効率化手法について述べる。4章でHSLで記述された回路に対する検証システムについて説明し、5章で検証例を示す。最後に6章で結論を述べる。

2. 時相論理によるハードウェアの仕様記述

2.1. 同期部と演算部

一般にシステムは、実際に論理・算術演算を行なう演算部と、各演算間のタイミングを制御し同期をとる同期部に分けることができる［15］。例えば、ハンドシェィクを用いてデータ転送を行うシステム（図1［14］）について考えると、出力レジスタInfout から入力レジスタInfinに至るデータ転送路が演算部に相当し、データ転送の制御を行う制御信号発生部（Call 、Hear ）が同期部に相当する。

演算部では、主に与えられた入力データを用い、指定された時間で指定された計算が終了し得るかが問題となるのに対し、同期部では、主に指定された時刻に指定されたデータが送られてきているかが問題となる。同期部では並列に動作するもの全体を考えなければならず、誤設計を生じやすい。従って、設計支援が特に望まれる。本論文では、同期部の効率的な検証について述べる。

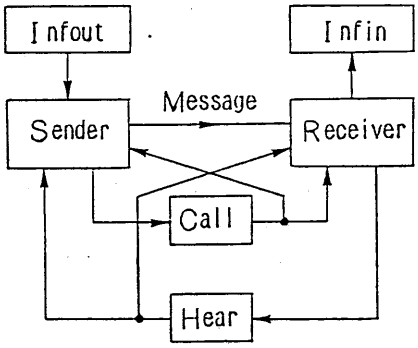

図1　ハンドシェィクを用いてデータ転送を行うシステム

2.2. 時相論理による同期部の仕様記述

本節ではまず時相論理について簡単に述べ、次に例を用いて検証や自動合成を効率よく行えるようなハードウェア同期部の仕様記述法について説明する。なお、時相論理についての詳細は参考文献［9、10］を参照されたい。

時相論理は、∧、∨、→、～等の古典論理の演算子に、○（next）、□（always）、▽（sometime）、Ｕ（until ）の4つの時相演算子を加えたものであり、各演算子は次のような意味をもつ。

○P：　次の時刻（同期回路では、次のクロック）にPが成り立つ

□P：　現在から将来ずっとPが成り立つ

▽P：　現在から将来の少くとも1時刻でPが成り立つ

P Ｕ Q：　現在から考えて、Qが成り立つまではPでありつづける。（ここでは、Qが必ずしも成立することを要求しないweak until［10］を用いる。）

時相論理を用いて通常タイムチャートで表されるような時間順序関係を表現することができる。

まず、『信号Pがactiveになると次の時刻に信号Qがactiveになる』は、

□（P→○Q）　…①

と表現できる。また、具体的に時刻は言えないが、将来少くともQがactiveになる場合は○を▽にかえて次のように表現できる。

□（P→▽Q）　…②

条件①や②では、『PがactiveになるとQがactiveになる』ことは保証するが、他の場合にもQがactiveになるかもしれない。これを『Qがactiveにへ変化するのはPがactiveになった次の時刻であり、かつその時に限る』とするには、次の条件を①に付け加える。

□（～Q→（（○～Q）　Ｕ　P））　…③

（以降、時相論理の式を並べたものは、各式の積（AND）を表わすものとする。）

信号の因果関係の表現は、①（又は②）と③を合せたものが基本となる。

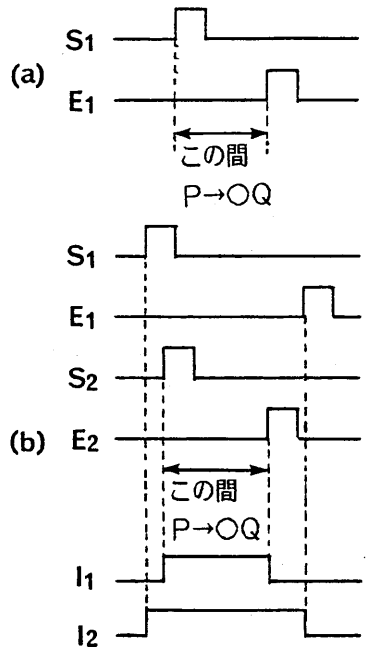

図2　タイムチャート

図2（a ）に示すタイムチャートのように、『スタート信号S1 とエンド信号E1 の間の時刻では信号Pがactiveになると次の時刻信号Qがactiveになる』は、次のようになる。

□（S1 →（（P→○Q） Ｕ E1 ））　…④

（ただし、S1 とE1 は図2のようにパルス状に外部から与えられ、かつS1 がE1 より先にくるとする。このことは、次のように時相論理で表現できる。

□（S1 →（（○～S1 ） Ｕ E1 ）），
（○～E1 ） Ｕ S1 ，
□（E1 →（（○～E1 ） Ｕ S1 ））　　）

さらに（b ）のように、スタート信号S1 とエンド信号E1 の間の時刻のさらにスタート信号S2 とエンド信号E2 の間の時刻においては、Pがactiveになると次の時刻Qがactiveになる』は次のようになる。

□（S1 →（（S2 →（（P→○Q） Ｕ E2 ））
Ｕ E1 ））　…⑤

このように複雑な仕様をそのまま記述すると時相演算子が次々とネスティングされる。しかし、スタート信号S1 とエンド信号E1 の間の時刻（interval）のみI1 という信号がactiveになり、I1 がactiveの時にスタート信号S2 とエンド信号E2 の間の時刻のみI2という信号がactiveになるというように記述すると簡単な条件の積で表現できる。

□（～I1 →（（～I1 ） Ｕ S1 ）），
□（S1 →I1 ），
□（I1 →（I1 Ｕ E1 ）），　　…⑥
□（E1 →（～I1 ）），
□（～I2 →（（～I2 ） Ｕ （I1 ∧S2 ））），
□（（I1 ∧S2 ）→I2 ），
□（I2 →（I2 Ｕ （I1 ∧E2 ））），
□（（I1 ∧E2 ）→（～I2 ）），
□（（I2 ∧P）→○Q）

これは最初4つの条件で信号I1 がactiveになる時刻をS1 とE1 の間の時刻のみにし、次の4つの条件で信号I2 がactiveになる時刻をI1 がactiveな時刻内のさらにS2 とE2 の間の時刻のみにし、そして最後の条件でI2 がactiveな時にPがactiveならば次の時刻Qがactiveになることを表現している。

一般に⑥のように適当な時間の幅（interval）［13］を考えることによって、複雑な仕様も簡単な条件の積で表現することができる。このI1 やI2 は外部には関係ない内部状態を表わすものであるが、これらを導入することにより仕様が記述しやすくなる。また、⑥から分かるように同じ形の条件式が多く、各々は単純な式となるため、後で述べるように検証に要する時間を低く抑えることができる。

3．　検証手法とその効率化手法

3．1．　検証手法

本節では、我々が既に示した検証手法について簡単に説明する。詳細については参考文献［14、15］を参照されたい。検証は時相論理の決定手続き［10］を基礎においており、時間を先に進めながら検証していく順方向推論（Forward Reasoning）と、時間を前に戻しながら検証していく逆方向推論（Backward Reasoning）が考えられる。また、処理系は全てProlog で記述されている。

ここでは、

□（A→（B Ｕ C））　…⑦

［但し、A、B、Cには時相演算子が含まれていないものとする］

を例にとって検証手法を具体的に説明する。

全ての場合について、（A→（B Ｕ C））を背理法によって検証することで⑦を検証する。まず否定をとって、～（A→（B Ｕ C））≡（A∧～（B Ｕ C））となる。従って、（A∧～（B Ｕ C））を満すパス（状態遷移の列）があるかないか調べ、もしあれば反例として示せばよい。

＜1＞□P＝P∧○□P
＜2＞▽P＝P∨（～P∧○▽P｛P｝）
＜3＞P1 Ｕ P2＝
P2∨（P1∧～P2∧○（P1 Ｕ P2））
＜4＞～□P＝～P∨（P∧○（～□P）｛～P｝）
＜5＞～▽P＝～P∧（P∧○（～▽P））
＜6＞～（P1 Ｕ P2）＝（～P1∧～P2）
∨（～P2∧○（～（P1 Ｕ P2））｛～P1｝）

表1　時相演算子の展開規則　ただし、P、P1、P2は任意の時相論理の式

次に、（A∧～（B Ｕ C））を時相論理の決定手続き（Decision Procedure［10］）によって、次の時刻毎に展開して状態遷移図を作る。各時相演算子には表1のような性質がある。表1は時相論理の公理から得られるもので例えば、＜1＞は『ずっとPであることは、現在Pであり、かつ、次の時刻からみてもずっとPである』ということを表している。また＜2＞は、『いつかPであるということは、現在Pであるか、または、現在はPでなくかつ、次の時刻からみていつかPである』ということを意味する。しかし、＜2＞においていつも～P∧○▽Pを選択していると、ずっと～Pとなってしまい▽Pを満さない。したがって、～P∧○▽Pは『いつかはPを選択する』という条件付きで選択しなければならない。これはeventuality と呼ばれるもので、～P∧○▽Pの後ろの｛P｝はこのeventuality を示している。

この表を用いることによって、任意の時相論理の式を現在に対する条件と次の時刻以降に対する条件に展開できる。

例として、（A∧～（B　U　C））を展開してみる。まず表1の<6>から次の式が成立する。

A∧～（B　U　C）

＝　（A∧～B∧～C）　　…⑧

　∨（A∧～C∧○（～（B　U　C））｛～B｝）

～（B　U　C）

＝　（～B∧～C）　　…⑨

　∨（～C∧○（～（B　U　C））｛～B｝）

⑧、⑨から図3のような状態遷移図が得られる。図3の状態2で＊の遷移を取り続けると｛～B｝のeventuality が満されず、A∧～（B　U　C）を満さないものとなるため、検証においてはこのような状態遷移列は反例とはみなさない。

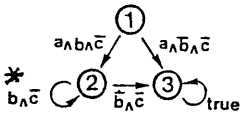

図3　～（A→（B　U　C））に対する状態遷移図

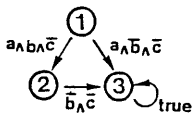

図4　図3を逆方向推論用に直したもの

仕様の否定を展開して得られた図3のような状態遷移図に従って、検証すべき設計記述を時間軸に対して順方向、又は逆方向に推論することで検証する。Prolog を用いた順方向推論は次のようになる。

① 初期状態1から始め、仕様の否定の状態遷移図の条件に従って設計記述の次の状態Nを得る。

② 仕様の否定の状態遷移図の条件に従って設計記述の次の状態Nを求めるという操作を同じ状態（仕様の否定の状態遷移図の状態名と設計記述の状態）が現れれループになるまで続ける。

③ ループになったら、そのループがeventuality を満すか否か調べ、満さなければ別のパスを調べるためにバックトラックをかけ②へ行き、満せばこのパスは反例なので印刷し、終了する。

逆方向推論の場合は、上を逆に辿っていけばよく次のようになる。

① 任意の状態から始め、仕様の否定の状態遷移図の条件に従って設計記述の1つ前の状態Bを求める操作をループになるまで繰り返す。

② ループになったら、そのループがeventuality を満すか否かを調べ、満さなければ別のパスを調べるためにバックトラックをかけ①へ戻り、満せば③へ行く。

③ 再び1つ前の状態を求める操作を繰り返し初期状態1に到達可能か否か調べる。不可能の場合はバックトラックをかけ①へ戻り、可能の時はこのパスは反例となるので印刷し、終了する。

以上から分かるように逆方向推論の方が、順方向推論よりも手間が多い。これは、順方向推論ではループになるパスをみつける操作のみであるが、逆方向推論ではさらに、そのパスが実際に初期状態に到達可能か否かを判定する操作が加わるからである。そこで、仕様の否定の状態遷移図においてループになり得る状態から始め、ループにならない遷移についてのみ上の③を調べるようにすれば、検証すべき条件はきつくなるが、手間をかなり減らすことができる。例えば、図3の場合では、図4のような状態遷移図に直して、状態3から始めて状態1に到達可能か否かをみるようにする。このようにすると、正しい設計に対しても反例がでる可能性があるが、まちがった設計を正しいと言うことはなく、5章の例からも分かるように実用上は十分である。

以上の検証は全てProlog を用いて行われる。ゲート回路、状態遷移図のProlog による表現法やProlog による検証プログラムについては4章及び、参考文献［14、15］を参照されたい。与えられた仕様に対して変化するのは図3のような仕様否定の状態遷移図のみであり、検証プログラムはいつも同じでよい。

3．2　効率化手法

前節のようにして、ハードウェア論理設計の検証を行うことができる。しかし、この手法は与えられた仕様について全ての場合を調べているので、検証に要する時間は回路規模、回路の状態数に対し指数的に増大する。従って、実用的規模の回路を検証対象とするためには何らかの方法で検証の効率化を行い、検証時間を抑える必要がある。実用規模の回路の検証を可能とする効率化手法として、『状態遷移の記憶』『外部条件の利用』『回路の絞り込み』が考えられる。以下にこれらの手法について述べる。

・状態遷移の記憶

3．1で述べたように、検証は時相論理で与えられる仕様の否定を各時刻毎に展開して状態遷移図を作り、検証対象の回路がこれを満さないことを調べることにより行う。この時、単純にProlog のバックトラック機構のみを用いていたのでは、同じ状態（仕様の否定の状態遷移図の状態名とフリップ・フロップの内部状態）が何度も現れ、同じ状態遷移を何度も処理することになる。たとえば、図5に示す状態遷移図による設計に対して、図3の仕様を検証する場合には、2つ合せて図6のような状態遷移図を調べることになる。この時、

1・D1 →2・D1 →2・D2 →2・D2　と

1・D2 →2・D1 →2・D2 →2・D2　は下線部で同じことを処理している。従って、一度計算した状態遷移を記憶しておき、以前に現れていない状態遷移についてのみ処理することにより検証の効率化が図れる。これはある程

度大きな回路に対しては非常に有効である。

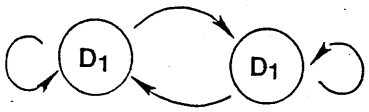

図5　状態遷移図による設計例

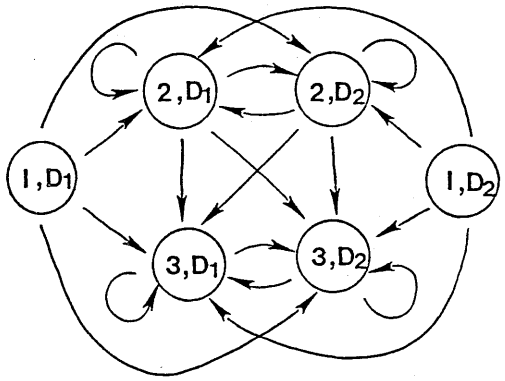

図6　仕様の否定（図3）と設計（図5）を合せた全体の状態遷移図

・外部条件の利用

検証対象の回路はそれ自身のみで動作することはなく、外部に何らかの回路がある場合が多い。この時、回路自身では分からないが、外部回路との接続によって入力の条件が定められる場合がある。これらを外部環境の条件として、時相論理式によって記述し、これを元に回路の各端子の値の変化を押えることにより、調べる場合の数を減らすことができ、検証の効率化が図れる。

・回路の絞り込み

以上に示した2つの手法により、いずれも検証時間は小さくなるが、回路規模に対してはやはり最悪の場合指数的に増大する。そこで、回路規模に検証時間ができるだけ依存しないような効率化手法が必要となる。2章でみてきたように、Interval の導入により、仕様は幾つかの簡単な時相論理の式の積で表現できる。検証はこれらの式を1つずつ調べていくが、1つの式に現れる出力端子の数は小さい。そこで、1つの仕様の式の検証に真に必要な回路のみを抽出して計算を行うことが考えられる。仕様に現れる出力端子から回路を出力から入力へ逆に辿っていくことにより、実際に影響を及ぼす範囲を絞り込むことができる。これをここでは『回路の絞り込み』と呼ぶ。例えば、2．1のハンドシェイクによるデータ転送システムのReceiver全体の回路は図7のようになるが、このうち仕様の1つである［14］

□（Call →▽Hear ）　…⑩

について絞り込むと図7中の点線内になる。前述のように、システムは同期部と演算部に分けられる。本検証システムは同期部が対象であり、また仕様は単純な条件式の積で表現されるため、1つの条件式の検証に真に必要な回路は同期部の一部でシステム全体から比べるとかなり小さい部分にすぎない。また、回路規模が大きくなっても絞り込んだ後の回路規模は大きくならないと考えられる。5章で示すように、『回路の絞り込み』に要する時間は検証時間に対して短く、かつ回路規模に対して比例した時間で済む。回路規模の増大に対し仕様中の条件式の数がほぼ比例して増大するため、検証時間も回路規模に対してほぼ比例した程度の増大で抑えられると考えられる。

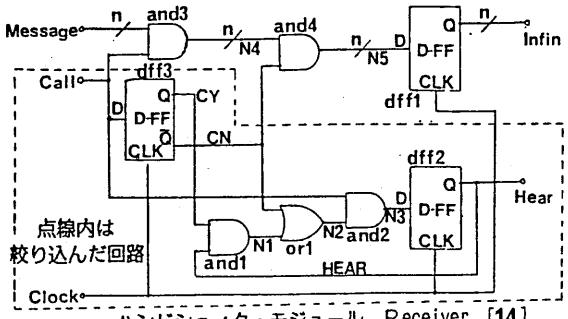

図7　ハンドシェイク・モジュール　Receiver［14］

4．　検証システムの実装

4．1　C－Prolog による実装

VAX11／730・UNIX上のC－Prolog［17］、及び一部にC言語を用いて効率化手法も含めて検証システムを実装した。C－Prolog はEdinburgh大学で開発された標準的な文法を持つProlog である。代表的な論理型言語であるProlog は強力なパターン照合機能と自動バックトラック機構を持つ。これらの機能を論理シミュレータや時相論理式の状態遷移表現への展開に利用することにより、検証システムの実装が容易になる。C－Prolog を用いて論理回路は次のように表現できる。（以下では組み合せ回路には遅延はないものとする。）NOT、AND、OR等の基本ゲートは、入力と出力の関係を真理値表に対応する形で、フリップ・フロップは、入力と出力の他に現在の状態と次の時刻の状態も含めた関係で図8のように表現される。モジュールは外部端子の値、現在の各フリップ・フロップの状態、及び次の時刻の状態を引数とし、ボディに各ゲートの記述をAND並列した形で表現できる。一例として、Receiver の回路（データ幅4bit ）をC－Prolog で記述したものを図9に示す。

図10に検証システムの構成を示す。ハードウェア記述（4．2参照）はHSL［18］で行われ、HSLトランスレータ（4．3参照）によりC－Prolog の記述（設計データベース）に変換される。仕様は時相論理の式をリスト表現にした形で記述され、TLトランスレータにより状態遷移表現の形のC－Prolog の記述に変換される。これは自動合成のためのプログラム（約1200行）を流用し

```
'FDEPE'([D,0,1,1,Q,Qn],Q,Q):-'FNOT'([Q,Qn]).
'FDEPE'([D,1,1,1,Q,Qn],Q,D):-'FNOT'([Q,Qn]).
'FDEPE'([D,Clk,0,1,1,0],Q,1).
'FDEPE'([D,Clk,1,0,0,1],Q,0).
'FDEPE'([D,Clk,1,0,1,1],Q,1).
```

Dinput Clock Preset Clear　　Q Qinv State Next-State

図8　基本ゲートのProlog による記述

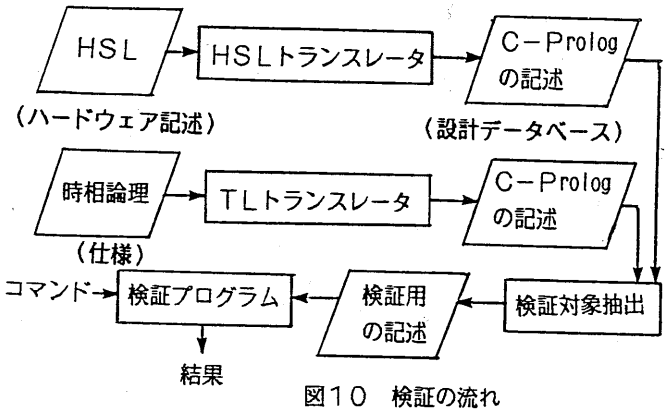

図10　検証の流れ

```
'RECEIVER'([_MESSAGE_3,_MESSAGE_2,_MESSAGE_1,_MESSAGE_0,_CALL,_CLOCK,_INFIN_3,_INFIN_2,
           _INFIN_1,_INFIN_0,_HEAR],
          [_DFF2,_DFF1_3,_DFF1_2,_DFF1_1,_DFF3,_DFF1_0],
          [__DFF2,__DFF1_3,__DFF1_2,__DFF1_1,__DFF3,__DFF1_0]):-
        'FAND2'([_CY,_HEAR,_N1]),
        'FAND2'([_MESSAGE_3,_CALL,_N4_3]),
        'FAND2'([_MESSAGE_2,_CALL,_N4_2]),
        'FAND2'([_MESSAGE_1,_CALL,_N4_1]),
        'FDEPE'([_CALL,_CLOCK,_DFF3_PS,_DFF3_PC,_CY,_CN],_DFF3,__DFF3),
        'FAND2'([_MESSAGE_0,_CALL,_N4_0]),
        'FOR2'([_CN,_N1,_N2]),
        'FAND2'([_N4_3,_CN,_N5_3]),
        'FAND2'([_N4_2,_CN,_N5_2]),
        'FAND2'([_N4_1,_CN,_N5_1]),
        'FAND2'([_N4_0,_CN,_N5_0]),
        'FAND2'([_CALL,_N2,_N3]),
        'FDEPE'([_N5_3,_CLOCK,_DFF1_3_PS,_DFF1_3_PC,_INFIN_3,_DFF1_3_QB],_DFF1_3,__DFF1_3)
        'FDEPE'([_N5_2,_CLOCK,_DFF1_2_PS,_DFF1_2_PC,_INFIN_2,_DFF1_2_QB],_DFF1_2,__DFF1_2)
        'FDEPE'([_N5_1,_CLOCK,_DFF1_1_PS,_DFF1_1_PC,_INFIN_1,_DFF1_1_QB],_DFF1_1,__DFF1_1)
        'FDEPE'([_N5_0,_CLOCK,_DFF1_0_PS,_DFF1_0_PC,_INFIN_0,_DFF1_0_QB],_DFF1_0,__DFF1_0)
        'FDEPE'([_N3,_CLOCK,_DFF2_PS,_DFF2_PC,_HEAR,_DFF2_QB],_DFF2,__DFF2).
```

図9　Receiver （データ幅4ビット）のProlog による記述

ている。詳細については参考文献［16］を参照されたい。検証対象部分抽出プログラムは検証に必要な回路を抽出し、検証用の記述に変換する。効率化の手法の内『回路の絞り込み』はこの時に行われる。これについては4．4で述べる。検証プログラムは、ハードウェアの検証用記述と仕様の状態遷移表現のC－Prolog の記述を受け取り、3章で示した手法に従って、それらをいっしょに状態遷移させることにより検証を行う。効率化の手法の内『状態遷移の記憶』『外部条件の利用』はこの時に行われる。これらについては4．5で述べる。検証の結果、反例を見つけるとその反例を表示して終了する。反例が出ないと、そのハードウェア記述は仕様を満すことが検証されたことになる。

4．2　階層記述言語HSLによるハードウェア記述

本検証システムは階層記述言語HSLによるハードウェア記述を設計入力とする。HSL［18］は電々公社武蔵野通信研究所が開発したハードウェア記述言語であり、スタンフォード大学で開発されたSDL［19］とほぼ同じである。HSLには次のような特徴がある。

① 階層構造が自由に取扱える。
② さまざまな設計フェーズ（論理設計、回路設計、レイアウト設計、テストパターン発生等）のデータが記述できる。
③ 仕様の追加、拡張に柔軟に対処できる。

例としてハンドシェイク・モジュールReceiver （図7）［14］のHSLによる記述例を図11に示す。

```
IDENT    :HANDSHAKE;
VERSION  :1.0;
DATE     :83/12/11;
AUTHOR   :M. FUJITA;
PROJECT  :VERIFIER;
COMMENT  :SAMPLE PROGRAM FOR DEMO AND TEST;
NAME     :RECEIVER;
PURPOSE  :LOGSIM ;
LEVEL    :MODULE;
EXT      :MESSAGE<0:3>, CALL, CLOCK, HEAR, INFIN<0:3>;
INPUTS   :.MESSAGE<0:3>, .CALL, .CLOCK;
OUTPUTS  :.HEAR, .INFIN<0:3>;
TYPES    :AND2,OR2,DEPE;
AND2     :AND1,AND2,AND3<0:3>,AND4<0:3> ;
OR2      :OR1 ;
DEPE     :DFF1<0:3>,DFF2,DFF3 ;
MESSAGE<0:3>     =FROM(.MESSAGE<0:3>)     TO(AND3<0:3>.1);
CALL             =FROM(.CALL)             TO(AND3<0:3>.2,AND2.1,
                                             DFF3 .D);
CLOCK            =FROM(.CLOCK)            TO(DFF1<0:3>.CLK, DFF2.CLK, DFF3.CLK);
HEAR             =FROM(DFF2.Q)            TO(.HEAR, AND1.2);
INFIN<0:3>       =FROM(DFF1<0:3>.Q)       TO(.INFIN<0:3>);
CY               =FROM(DFF3.Q)            TO(AND1.1);
CN               =FROM(DFF3.QB)           TO(OR1.1, AND4<0:3>.2);
N1               =FROM(AND1.3)            TO(OR1.2);
N2               =FROM(OR1.3)             TO(AND2.2);
N3               =FROM(AND2.3)            TO(DFF2.D);
N4<0:3>          =FROM(AND3<0:3>.3)       TO(AND4<0:3>.1);
N5<0:3>          =FROM(AND4<0:3>.3)       TO(DFF1<0:3>.D);
END;
CEND ;
```

図11　Receiver（データ幅4ビット）のHSLによる記述

```
ident('HANDSHAKE').
version('1.0').
date('83/12/11').
project('VERIFIER').
extern_name('RECEIVER',['MESSAGE',[0,3]]).
extern_name('RECEIVER',['CALL']).
extern_name('RECEIVER',['CLOCK']).
extern_name('RECEIVER',['HEAR']).
extern_name('RECEIVER',['INFIN',[0,3]]).
gate_interface('RECEIVER',[['MESSAGE',[0,3]],['CALL'],['CLOCK']],[['HEAR'],['INFIN',[0,3]]]).
type_def('RECEIVER',[   [['AND1'],'AND2'],
                        [['AND2'],'AND2'],
                        [['AND3',[0,3]],'AND2'],
                        [['AND4',[0,3]],'AND2'],
                        [['OR1'],'OR2'],
                        [['DFF1',[0,3]],'DEPE'],
                        [['DFF2'],'DEPE'],
                        [['DFF3'],'DEPE']]).
all_net_def('RECEIVER',[['MESSAGE',[0,3]],['MESSAGE',[0,3]],[[['AND3',[0,3]],['1']]]]).
all_net_def('RECEIVER',[['CALL'],['CALL'],[[['AND3',[0,3]],['2']],[['AND2'],['1']],[['DFF3'],['D']]]]).
all_net_def('RECEIVER',[['CLOCK'],['CLOCK'],[[['DFF1',[0,3]],['CLK']],[['DFF2'],['CLK']],[['DFF3'],['CLK']]]]
all_net_def('RECEIVER',[['HEAR'],[['DFF2'],['Q']],[['HEAR'],[['AND1'],['2']]]]).
all_net_def('RECEIVER',[['INFIN',[0,3]],[['DFF1',[0,3]],['Q']],[['INFIN',[0,3]]]]).
all_net_def('RECEIVER',[['CY'],[['DFF3'],['Q']],[[['AND1'],['1']]]]).
all_net_def('RECEIVER',[['CN'],[['DFF3'],['QB']],[[['OR1'],['1']],[['AND4',[0,3]],['2']]]]).
all_net_def('RECEIVER',[['N1'],[['AND1'],['3']],[[['OR1'],['2']]]]).
all_net_def('RECEIVER',[['N2'],[['OR1'],['3']],[[['AND2'],['2']]]]).
all_net_def('RECEIVER',[['N3'],[['AND2'],['3']],[[['DFF2'],['D']]]]).
all_net_def('RECEIVER',[['N4',[0,3]],[['AND3',[0,3]],['3']],[[['AND4',[0,3]],['1']]]]).
all_net_def('RECEIVER',[['N5',[0,3]],[['AND4',[0,3]],['3']],[[['DFF1',[0,3]],['D']]]]).
```

図12　図11からHSLトランスレータにより得られるProlog の記述

4.3　HSLトランスレータ

HSLトランスレータは、本検証システム中唯一C、LEX、及びYACCを用いて、約3000行で記述されている。これはHSLが文脈自由文法であり、C、LEX、YACCにより容易にトランスレータを作成できるためである。本トランスレータのみC言語で記述してあるため、他のシステムに対しバッチ的な使用しか行えないが、設計データベースを作成する目的からみて十分である。本トランスレータはゲート・レベルの検証システム用に作成されたものであり、HSLの文法は完全に受け付けるが、ゲート・レベルより低いレベル（素子レベル等）の記述を無視する。

Receiver のHSL記述を本トランスレータでC－Prolog の記述（設計データベース）に変換した例を図12に

示す。

4．4　検証部分抽出プログラムと回路の絞り込み

検証すべき仕様から実際に検証に必要な回路が定まる。この部分を設計データベースから抽出して検証プログラムの必要とする形にするのが検証部分抽出プログラムである。検証部分抽出プログラムは『回路の絞り込み』『Rank 付け』『検証プログラム用の記述への変換』の大きく3つの部分に分けることができる。

4．4．1　『回路絞り込み』プログラム

HSLトランスレータから生成されたC－Prolog の設計データベースは、HSLの特徴である階層的記述、及び配列表現による記述がなされている。検証を行うためには、これをHSLの標準ゲートとして定義されているゲート、及びフリップ・フロップのレベルまで展開し、同時に配列表現をも展開する必要がある。この時に、真に検証に必要な部分のみを抽出して展開することにより『回路の絞り込み』を行うことができる。実装した回路の絞り込みのアルゴリズムは次のようなものである。

```
program　回路絞り込み
  procedure　端子絞り込み（端子）
    begin
    端子を出力から入力へ逆方向にネットを溯る　；
    そのネットを必要なネットとして記憶する　；
    if（外部端子に抜ける）　then　return
    else　if（一度通ったゲートに当る）
      then　return
    else　begin　｛初めてのゲートに当ったので｝
      そのゲートを記憶する　；
      そのゲートの全ての入力端子を絞り込む
      べき端子として登録する　；
    end
  end　；

  begin
    仕様から検証に必要な全ての外部出力端子を抽出し、
    絞り込むべき端子として登録する　；
    while（絞り込むべき端子がある）　do
    ”端子絞り込み”（端子）　；
  end .
```

4．4．2　『Rank 付け』のプログラム

本検証システムでは、4．4．3節で述べる検証プログラム用の記述への変換の際に、各ゲートのRank を用いて最適化された記述を生成している。

Rank 付けのアルゴリズムは次のようになる。

```
procedure　Rank 付け
  begin
    全ての外部入力端子、フリップ・フロップの
    出力端子のネットに対し、Rank を1とする　；
    while（Rank の付いていないゲートがある間）　do
    for（全てのRank の付いていないゲートについて）
      if（そのゲートの全ての入力端子にRank が付いて
          いたら）then　begin
        入力端子中、最大のRank をそのゲートのRank
        とする　；
        ゲートの出力端子のネットに対し、
        ゲートのRank ＋1をRank として付ける；
      end
  end .
```

4．4．3　『検証プログラム用の記述への変換』プログラム

検証プログラムは4．1節で述べた回路記述を用いて検証を行う。従って、絞り込んだ回路をこの形に変換する必要がある。この際、各ゲートの記述をAND並列する順序に各ゲートのRank を参考にすることにより、回路記述内でのバックトラックをなくし、検証時間の減少を図ることができる。

本プログラムでは必要とする回路記述を生成するためにはアトムからユニークな名前の変数を生成する必要がある。C－Prolog にはこのような機能はないため、一度外部ファイルに文字列として出力し、その外部ファイルをC－Prolog の記述として読みこむことにより、この処理を行っている。

以上のプログラムは説明の都合上、3つ部分に分けたが、実際は約1000行の一本のまとまったプログラムとなっている。本プログラムによってReceiver （図7）の回路を仕様⑩を元に、設計データベース（図12）から絞り込んだ検証用の記述を図13に示す。

4．5　検証プログラムの効率化

『外部条件の利用』による効率化は、現在のところ、利用できる条件式を人手によって与え、それを状態遷移表現に展開し、検証条件として加えることで行っている。

状態遷移の記憶はC－Prolog により簡単に実現できる。ある状態遷移の処理が終了すれば、その状態遷移をプレディケイトとしてassertしておけば、同じ状態になったかはそのプレディケイトがパターン照合するかを見ればよい。検証プログラムの大きさは約600行である。

```
'RECEIVER'([_CALL,_CLOCK,_HEAR],[_DFF2,_DFF3],[__DFF2,__DFF3]):-
        'FAND2'([_CY,_HEAR,_N1]),
        'FDEPE'([_CALL,_CLOCK,_DFF3_PS,_DFF3_PC,_CY,_CN],_DFF3,__DFF3),
        'FOR2'([_CN,_N1,_N2]),
        'FAND2'([_CALL,_N2,_N3]),
        'FDEPE'([_N3,_CLOCK,_DFF2_PS,_DFF2_PC,_HEAR,_DFF2_QB],_DFF2,__DFF2).
```

図13　図7を仕様⑩を元に絞り込んだ回路に対するProlog の記述（図9参照）

5.　検証例

5．1　簡単な例

簡単な小規模の回路の例として、ハンドシェイク・モジュルReceiver （図7）について、仕様［14］

□（Reset→□（Call →▽Hear ））　…○

を検証した。検証方法は順方向推論を用い、『回路の絞り込み』を行う場合・行わない場合、『状態遷移の記憶』を行う場合・行わない場合について比較を行った。『外部条件の利用』については、検証対象の回路が簡単なため適用可能な外部端子がなく、比較を行わなかった。

検証結果を表2に示す。また、仕様○を元にReceiverの回路を絞り込んだ回路は図7の点線内の回路である。表から回路の絞り込みを行わないと、検証時間はすぐに実用範囲を越えること、及び、状態遷移の記憶を行うと、回路規模に対する処理時間の増大率を抑えられることが分かる。

	状態の記憶あり	状態の記憶なし
絞り込んだ回路	0．93	0．87
演算部データ幅1bit	4．28	9．88
〃　2bit	28．25	204．80
〃　3bit	253．30	5645．80

表2　□（Call →▽Hear ）の検証に要するCPU時間
（単位：秒、VAX11／730上のC－Prolog による）

5．2　複雑な例

複雑な回路の検証例として、（a ）我々の研究室で現在開発中の高並列推論エンジンPIEのUnify　Processorのマイクロプログラム・シーケンサ制御部［21］、（b ）富士通のミニコンピュータU－300のDMA制御回路［22］について検証を行った。

（a ）の回路については回路規模が大きく、絞り込みを行わない回路については検証時間が長過ぎて測定が不可能であり、絞り込んだ回路のみについて測定を行った。また、『外部条件の利用』は行っていない。検証方法は順方向推論及び逆方向推論の両方を用い、『状態の記憶』を行う場合・行わない場合について比較を行った。検証の際、逆方向推論において検証を開始する状態（状態遷移図でループとなる状態）は人間が状態遷移図から判断して与え、計算機はループのチェックを行っていない。仕様は出力端子の違いから、4つずつ2種類に分けられるため、絞り込みも2種類行っている。仕様については参考文献［16］を参照されたい。

また（b ）の回路については、順方向推論とループのチェックを省略した場合の逆方向推論の両方について参考文献［20］に示す仕様について、『回路の絞り込み』『状態遷移の記憶』『外部条件の利用』全部の組み合せについて検証に要する時間の比較を試みた。

紙面の都合上、結果の詳細については省略するが、検証結果から次のことが分かった。なお、結果の詳細については参考文献［23］を参照されたい。

（1）「状態遷移の記憶』『外部条件の利用』を行うと、検証時間は1～2桁短くなる。また、回路規模に対する増加率も抑えられる。このため、大型計算機を用いれば100ゲート程度の回路の検証は十分行える。なお、実際に記憶される状態遷移の数は、数十ゲート程度の回路で10～200程度であった。

（2）『回路の絞り込み』を行うと、仕様が単純な式なため、もとの回路規模にほぼ関係なく数十ゲートの大きさに絞り込める。また、絞り込みに要する時間はもとの回路規模に比例する程度の時間であり、（a ）（b ）の例では効率化された検証時間の数倍程度であった。

6.　おわりに

処理系にC－Prolog を用いたゲート回路に対する論理検証システムについて述べた。各種効率化手法、特に『回路の絞り込み』により、同期部については実用規模の回路の検証も十分実用的時間で行うことができる。

今後は状態遷移レベルの設計記述の検証や演算部の検証を本システムに組み込み込んで行きたい。

参考文献

[1] M.A. Breuer and A.D. Friedman, A. Isoupovicz, "A Survey of the State of the Art of Design Automation", Computer, Vol.14, No.10, pp.58-75 October 1981.

[2] G.F. Pfister, "The YORKTOWN SIMULATION ENGINE: INTRODUCTION", ACM IEEE 19th DA Conference, June 1982.

[3] T. Sasaki, N. Koike, K. Omori and K. Tomita, "HAL: A Block Level Hardware Logic Simulator", ACM IEEE 20th DA Conference, June 1983.

[4] T.J. Wangner, "Hardware Verification", Dept. of Computer Science, Stanford Univ., Report STAN-CS-77-632, 1977.

[5] A.S. Wojcik, "Formal Design Verification of Digital Systems", ACM IEEE 20th DA Conference, June 1983.

[6] V. Pitchuman and E.P. Stabler, "A Formal Method for Computer Design Verification", ACM IEEE 19th DA Conference, June 1982.

[7] J.A. Darringer, "The Application of Program Verification Techniques to Hardware Design", ACM IEEE 16th DA Conference, June 1979.

[8] W.C. Carter, W.H. Joyner Jr. and D. Brand, "Symbolic Simulation for Correct Machine Design", ACM IEEE 16th DA Conference, June 1979.

[9] Z. Manna and A. Pnueli, "Verification of Concurrent Programs, Part 1: The Temporal Framework", Dept. of Computer Science, Stanford Univ. Report STAN-CS-81-836, June 1981.

[10] P. Wolper, "Temporal Logic Can Be More Expressive", 22nd Annual Symposium on Foundation of Computer Science, October 1981.

[11] G.V. Bochmann, "Hardware Specification with Temporal Logic: An Example", IEEE Trans. Computer, Vol.C-31 No.3 pp.223-231, March 1982.

[12] B. Mishra and E.M. Clarke, "Automatic and Hierarchical Verification of Asynchronous Circuits Using Temporal Logic", Dept. of Computer Science, Carnegie-Mellon Univ. Report CMU-CS-83-155, September 1983.

[13] Moszkowski, "Resoning about Digital Circuit", Dept. of Computer Science, Stanford University, Report No.STAN-CS-83-970, 1983.

[14] 藤田, 田中, 元岡, "時相論理によるハードウェア仕様記述とProlog を用いたゲート回路の検証", 情報処理学会論文誌, Vol. 25, No.2, 1984.

[15] 藤田, 田中, 元岡, "ハードウェア状態遷移表現のProlog による検証", 情報処理学会論文誌, 採録済み.

[16] 藤田, 田中, 元岡, "時相論理を用いたハードウェア同期部の記述と状態遷移表の自動合成", 電子通信学会, 電子計算機研究会資料, EC83-59, 1984.

[17] F. Pereira, "C-Prolog Users Manual Version 1.2a"

[18] 杉山, 須藤, 唐津, "VLSI設計システム", 情報処理学会, 電子装置設計技術研究会資料, 7-2, 1980.

[19] W.M. VanCleemput, "A Hierarchical Language for the Structural Description of Digital Systems", ACM IEEE 14th DA Conference, June 1977.

[20] 藤田, 中田, 田中, 元岡, "時相論理を用いた論理設計検証システムの実際的使用について", 電子通信学会, 電子計算機研究会資料, EC-83-29, 1983.

[21] M. Yuhara, H. Koike, H. Tanaka and T. Moto-oka, "A Unify Processor Pilot Machine for PIE", The Logic Programming Conference '84, Tokyo, Japan, March 1984.

[22] "User Device Design Manual for PANAFACOM U-series", 09HS-0080-1, Fujitsu Ltd.

[23] 西山, "Prolog を用いたハードウェア同期部検証の効率化", 東京大学工学部電気工学科卒業論文, 1984.