システムＬＳＩ設計技術　97－3
（２０００．９．７）

システム設計レベルにおける回路の性質検証のための整数データを処理可能なCTLモデル検査法の提案と実装

佐藤 友哉　　北道 淳司　　東野 輝夫

大阪大学
大学院基礎工学研究科

大阪大学
サイバーメディアセンター

大阪大学
大学院基礎工学研究科

〒 560-8531 豊中市待兼山町 1-3

TEL:(06)6850-6593
satou@ics.es.osaka-u.ac.jp

TEL:(06)6850-6072
kitamiti@ecs.cmc.osaka-u.ac.jp

TEL:(06)6850-6590
higashino@ics.es.osaka-u.ac.jp

概要　NTTが開発したハードウェア記述言語であるSFLを用いて設計された回路に対し，その回路上で充足すべき性質が満足されているかどうかを形式的に検証する方法を提案する．論理設計レベルで用いられるCTLモデル検査法に対し，システム設計レベルにおいて用いられるレジスタを整数とみなし，整数データのままでCTLモデル検査法を行うことを考える．本研究では扱う整数データのクラスとしてプレスブルガー算術と呼ばれるクラスを採用する．本研究では回路モデルとして，上下限のある整数データを保持できるレジスタを有する拡張有限状態機械EFSMを採用し，回路記述言語SFLからEFSMへの変換手順を示し，EFSMに対するCTLモデル検査アルゴリズムを提案する．アルゴリズムの実現においては，我々が開発したプレスブルガー算術処理用ライブラリを用いた．SFLで記述した例題回路に対して実現した検証系を用いて実際に検証を行った結果について報告する．

キーワード　システム設計レベル，CTLモデル検査法，プレスブルガー算術，拡張有限状態機械，

Proposal and Implementation of CTL Model Checking Algorithm Using Integer Data for Property Verification of Circuits on System Design Level

Tomoya SATOH　Junji KITAMICHI　Teruo HIGASHINO

Graduate School of Engineering Science, Osaka University

Cybermedia Center, Osaka University

Graduate School of Engineering Science, Osaka University

1-3, Machikaneyama-cho, Toyonaka-shi, Osaka, 560-8531 Japan

Abstract　For designed circuits described with SFL ; hardware description language developed by NTT, we propose a formal verification method which determines whether the properties that circuits should satisfy is actually true or not. Against CTL model checking in general applied on logic design level, we regard a register on system design level as an integer and execute CTL model checking using integer data. In our research, we adopt Presburger arithmetic, a class of integer. As a circuit model we introduce extended finite state machine(EFSM) with registers that can hold integer data, give the translation algorithm to EFSM from SFL, and propose CTL model checking algorithm on proposed EFSM. For implementation of the algorithm, the library we developed for the operation of Presburger arithmetic is used. Finally we describe the results of verification for example SFL circuits.

key words　System Design Level, CTL Model Checking, Presburger Arithmetic, Extended Finite State Machine

1 はじめに

近年のLSI製造分野では，システムLSIのような大規模回路の設計需要が非常に高まっており，ハードウェア記述言語を用いた抽象的な記述による設計を行うことが多くなっている．その一方で，設計した回路の正しさを検証する手法として，論理設計レベルではBDD[1](Binary Decision Diagram，二分決定グラフ)を用いた形式的な検証手法が報告されているが，ハードウェア記述言語のレベルでは，シミュレーション等の非形式的な手法が一般的である．

我々は，NTTが開発したハードウェア記述言語であるSFL[2]を用いて設計された回路に対し，その回路上で充足すべき性質が満足されているかどうかを形式的に検証する方法を提案する．設計回路上での性質の充足性を判定する手法としてCTL(Computation Tree Logic)モデル検査法[3]がある．本論文ではこのCTLモデル検査法に対し，レジスタを上下限のある整数とみなし，整数データを持つモデルの上でのCTLモデル検査法を行う形式的検証法を提案する．本研究では，扱う整数データのクラスとしてプレスブルガー算術[4](Presburger Arithmetic)と呼ばれるクラスを採用する．

本稿では，回路モデルとしては整数データを保持できるレジスタを有するEFSM(拡張有限状態機械)を採用する．提案するモデル検査法では，SFLで記述される状態間の遷移条件，レジスタの更新動作や入出力関係などをプレスブルガー算術で表現する．そしてEFSMに対して定義した基本アルゴリズムを用いて，拡張したCTL式が成立する状態空間を計算し，モデル検査を行う．EFSMを表現するためのデータ構造や基本アルゴリズムの実現には，我々の研究グループで開発したプレスブルガー算術処理用ライブラリ[4]を用いる．これは論理関数の効率的な表現として知られているBDDをプレスブルガー算術の表現のためのデータ構造およびその処理に拡張したものである．

SFLで記述した例題回路に対し提案手法を用いて評価実験を行った．加算器，カウンタ，排他的制御を行う回路などに対して回路で成立すべき性質を拡張したCTL式で表現し，それらが正しく成立する事を数秒以内の計算時間で証明する事が出来た．

以下，まず2章でプレスブルガー算術及びCTLについて説明する．3章ではSFLからEFSMへの変換方法，4章で提案するCTLモデル検査法の基本アルゴリズムについて述べる．5章で例題回路に対する評価を行う．

2 プレスブルガー算術と拡張CTL

本章では，整数型の変数，加減算および大小比較などの演算を含む論理式を表現できるプレスブルガー算術と，CTLをプレスブルガー算術を扱う事が出来るように拡張した論理(以下，拡張CTL)について述べる．

2.1 プレスブルガー算術

本論文で扱うプレスブルガー算術の定義を行う．

定義1：プレスブルガー算術[1]

プレスブルガー算術を構成する終端記号は，
{ True, False, $\wedge, \vee, \neg, (,), +, -, \exists, =, <$, 0, 1, 2, $\ldots, x_1, \ldots, y_1, \ldots$ } である．ここで $x_1, \ldots$ は整数変数，$y_1, \ldots$ は論理変数とする．整数を値域とする項τは以下のように定義される．

$$\tau ::= x_1, \ldots, 0, 1, 2, \ldots, (\tau + \tau), (-\tau)$$

プレスブルガー算術εは以下のように定義される．ただしzは整数変数または論理変数，nは整数定数，$n|\tau$は項τが整数nで割り切れる事を表す．

$$\begin{aligned}\varepsilon \quad ::= \quad & y_1, \ldots, \mathrm{True}, \mathrm{False}, (\tau < \tau), (\tau = \tau), \\ & (n|\tau), (\varepsilon \wedge \varepsilon), (\varepsilon \vee \varepsilon), (\neg\varepsilon), (\exists z \varepsilon)\end{aligned}$$ □

以降では略記法として，整数変数xを用いた式$x+x$を，$2x$と表現する．また，$\alpha > \beta$及び$\alpha \leq \beta$を，それぞれ$\beta < \alpha$及び$\alpha - 1 < \beta$の代わりに用いる．さらに演算子を組み合わせる事により，$\forall$, $\oplus$(排他的論理和)，if ε then ε (else ε)という表現も可能であり，以降はこのような表現も用いる．

2.2 拡張CTL

CTLモデル検査法では，検証する性質を仕様記述言語CTLを用いて表現する．本稿ではCTLにおいてプレスブルガー算術を扱う事が出来るように拡張する．

定義2：拡張CTLのsyntax

- 定義1における
  $y_1, \ldots, \mathrm{True}, \mathrm{False}, (\tau < \tau), (\tau = \tau), (n|\tau)$
  はCTL式である．
- ϕ, ψがCTL式ならば，以下もCTL式である．
  $(\neg\phi), (\phi \vee \psi), (\phi \wedge \psi), (EX\phi), (EG\phi)$,
  $(E\phi U\psi), (\exists z\phi)$.
  ただしzは整数変数または論理変数である □

定義1における整数変数$x_1, \ldots$には整数定数である上下限値$sup_1, \ldots, low_1, \ldots$が設定される．以降，略記法として$AX\phi = \neg EX\neg\phi$,$EF\phi =$

[1]本来プレスブルガー算術には論理変数は含まれていないが，以下では，このように拡張された定義のもとで議論を進める．

$E(\text{True})U\phi$, $AF\phi = \neg EG\neg\phi$, $AG = \neg EF\neg\phi$, $A\phi U\psi = \neg(E\neg\psi U(\neg\phi \wedge \neg\psi)) \wedge \neg EG\neg\psi$, $\forall z\phi = \neg\exists\neg\phi$を用いる.

ここで扱う EFSM(拡張有限状態機械) のモデル K は, $\{C, C_{init}, S, S_{init}, \sum, \delta\}$ で定義される. ここで, Cは制御部レジスタの状態集合, C_{init}は制御部レジスタの初期状態, Sは制御部レジスタ以外のレジスタからなる状態集合, S_{init}は制御部レジスタ以外のレジスタの初期状態, $\sum$は入力記号の集合, δは遷移関係関数である. δは$\{C_{start}, cond, \lambda, C_{end}\}$の組の集合である. この組は, いわゆる状態遷移図の一遷移に相当する. C_{start}は状態遷移の始点の制御部レジスタの値, $cond$ はこの状態遷移の実行条件を表す論理式, λは遷移の前後での状態での制御部レジスタ以外のレジスタおよび入力の関係を表す論理式, C_{end}は状態遷移の終点の制御部レジスタの値である. $cond$ およびλは, プレスブルガー算術を用いて記述される. 以下では, 制御部レジスタの値を状態名と呼ぶ.

定義 3 : 拡張 CTL の semantics

拡張 CTL の semantics は以下のように定義される.

$K, s_0 \models \phi$	状態 s_0 で $\phi = \text{True}$
$K, s_0 \models \neg\phi$	状態 s_0 で $\phi = \text{False}$
$K, s_0 \models \phi \vee \psi$	$K, s_0 \models \phi$ または $K, s_0 \models \psi$
$K, s_0 \models \phi \wedge \psi$	$K, s_0 \models \phi$ かつ $K, s_0 \models \psi$
$K, s_0 \models EX\phi$	$K, s_1 \models \phi$ である Kの遷移系列 $s_0, s_1, \ldots$ が存在
$K, s_0 \models EG\phi$	任意の $i \geq 0$ に対し $K, s_i \models \phi$ である K の遷移系列 $s_0, s_1, \ldots$ が存在
$K, s_0 \models E\phi U\psi$	ある i に対し $K, s_i \models \psi$, かつ任意の $0 \leq j < i$ に対し $K, s_j \models \phi$である K の遷移系列 $s_0, s_1, \ldots$ が存在
$K, s_0 \models \exists z\phi$	zのある値に対し $K, s_0 \models \phi$. ただし zが整数変数の場合, その値が設定した上下限内にあること

□

$K, s_0 \models \phi$は, EFSM のモデル Kの状態 s_0においてCTL 式ϕが満足される事を表す. Kの状態は, 全レジスタの状態 (ただし整数レジスタは上下限の範囲内の値を取り得る) の組である. $K, s_0 \models \phi$ と $K, s_0 \models \neg\phi$におけるϕは, 定義 2 の論理変数, 論理定数, $(\tau < \tau), (\tau = \tau), (n|\tau)$ 及び制御レジスタ $= state_i \in C$の形の式であり, ϕの真偽はその状態における $S, \sum$の値及び整数変数の上下限値 $sup_1, \ldots, low_1, \ldots$ により決定される. 提案する CTL モデル検査法では, S及び$\sum$の整数変数の値が一つでも設定した範囲外にある場合のϕの真偽を考えないように, アルゴリズムを定義する. また EX, EG, EUにおける遷移系列 $s_0, s_1, \ldots$とは, s_iにおいて $C = C_{start}$, s_{i+1}において $= C_{end}$, かつ s_iでの $S, \sum$の値により $cond$ が真となるようなδが存在する $\{s_i, s_{i+1}\}(0 \leq i)$ の系列をいう.

3 SFL から EFSM への変換法

本章では, SFL から EFSM モデルへの変換方法について述べる.

3.1 モジュール

SFL 記述の基本単位はモジュールと呼ばれ, 加算器やマルチプレクサのような部品を表現する. モジュールは, 他のモジュール (サブモジュール) を使うことにより, 階層的な設計が可能である.

モジュール内では, モジュール内共通動作と制御端子による動作, 及びステージの定義を行う. 一つのステージは一つの制御部を記述する事ができ, 一つの EFSM によって表現できる. モジュール内共通動作と制御端子による動作は, モジュール内の全てのステージに共通の動作である. ステージが複数存在する場合は, 各ステージの EFSM の直積マシンを生成する事により, 一つのモジュールを一つの EFSM に変換する. またモジュールが複数定義されている時は更に各モジュールに関する EFSM の直積マシンを求める.

3.2 ステージ

ステージは, 複数の状態とその状態内での動作, 及び状態間の状態遷移で表現した制御部と考えられる. ステージでは状態ごとに動作を記述する. ステージ内の全ての状態に共通の動作はステージ内共通動作として記述される. 状態ごとの動作定義では, その中に記述されている動作から, 遷移ごとに状態遷移関数δ をプレスブルガー算術で表現し, このステージを表現する EFSM を得る.

3.3 動作

SFL の動作は, 複数の単位動作が集まった動作ブロックである. 動作ブロックは, par ブロック, alt ブロック, any ブロック, if ブロックである.

par ブロック

ブロック内にある動作が全て並行に動作する事を記述する.

par { 動作$_1$; ...; 動作$_n$}

この記述は,

$$\bigwedge_{i=1}^{n} \text{動作}_i$$

というプレスブルガー算術により表現する.

alt ブロック

優先度付きの条件をもつ動作を記述する.

alt { 条件$_1$: 動作$_1$; ...; 条件$_{n-1}$: 動作$_{n-1}$;

else : 動作$_n$}

条件$_1$から条件$_{i-1}$が全て成立せず，かつ条件$_i$が成立する時は動作$_i$を実行する．else の動作は条件$_1$から条件$_{n-1}$が全て成立しない時実行される．この記述は

$$\bigwedge_{i=1}^{n}\{\text{if}(\bigwedge_{k=1}^{i-1}(\neg\text{条件}_k)\wedge\text{条件}_i)\text{then 動作}_i\}$$

というプレスブルガー算術により表現する．

any ブロック

優先度のない条件付き動作を記述する．

any { 条件$_1$: 動作$_1$; ... ; 条件$_{n-1}$: 動作$_{n-1}$;

else : 動作$_n$}

他の条件の成否に関わらず，条件$_i$が成立する時は動作$_i$を実行する．else の動作は条件$_1$から条件$_{n-1}$が全て成立しない時実行される．この記述は

$$\bigwedge_{i=1}^{n-1}\{(\text{if}(\text{条件}_i)\text{then 動作}_i)\wedge \text{if}(\bigwedge_{k=1}^{n-1}(\neg\text{条件}_k))\text{then 動作}_n\}$$

というプレスブルガー算術により表現する．

if ブロック

一つの条件付き動作である．この記述は

if(条件)then 動作

というプレスブルガー算術により表現する．

3.4 単位動作

単位動作は，データ端子への出力，レジスタの更新，状態遷移，制御端子の起動である．またモジュール内の制御端子による動作についても述べる．

データ端子への出力

式の値をデータ端子に代入する事を記述する．

データ端子名 = 式;

この記述は

端子名の整数変数 = 式

というプレスブルガー算術により表現する．

レジスタの更新

式の値をレジスタに代入する事を記述する．

レジスタ名:=式;

ただし，SFL ではレジスタに式の値が実際に代入されるのは 1 サイクル後 (次状態) であるため，次状態のレジスタを表す変数を新たに設け，これを用いて次のプレスブルガー算術で表現する必要がある．以降では'(ダッシュ) が付加した変数を，次状態を表す変数とする．この記述は

レジスタ名の整数変数$'$ = 式

というプレスブルガー算術により表現する．

状態遷移

状態間の遷移を記述する．

goto 状態名;

goto の動作が定義されている状態名と，goto の引数として指定されている状態名より，遷移の始状態，終状態は判明するが，この遷移が起こるための条件は，この goto が出現するまでの実行条件のネスト状況より決定する．

制御端子の起動

制御端子を起動する事を記述する．

制御端子名 ([引き数, 引き数]);

制御端子は各動作を制御するための 1,0 の値を取る端子であり，論理変数として表現する．制御端子の起動とは論理変数の値を 1 にすることであり，

制御端子名の論理変数

というプレスブルガー算術により表現する．また制御端子には引数を取るように宣言されているものがあり[2]，その場合引数についてのデータ転送に関する関係式が必要になる．ただし別の動作中で同じ引数に別の値をデータ転送させている記述も考えられるので，転送される値とその転送が起きる条件について考慮する必要がある．よって引数の関係式を

if(動作が起きる条件) then 引数の関係式

というプレスブルガー算術により表現する．

制御端子による動作

制御端子が起動された時の動作を記述する．

instruct 制御端子名 動作;

これは，if ブロックでの動作の条件を制御端子の起動として表されたものと考え，if ブロックと同様に

if(制御端子名の論理変数)then 動作

というプレスブルガー算術により表現する．

3.5 式

SFL における式をプレスブルガー算術表現に変換する際には，プレスブルガー算術で定義されている演算子を用いればそのまま表現できるものが多い．SFL の式中で使用できる演算子のそれぞれの表現方法について，表 1及び表 2に示す．加算 (インクリ

[2]例えば，引数としてサブモジュールの入力を宣言し，その入力値を与えてサブモジュールが起動されるようにする使用法が可能である

Syntax	Semantics	プレスブルガー算術での表現
単項式 | 式	論理和	"単項式 ∨ 式"というプレスブルガー算術で表現．ここで単項式，式は論理式．
単項式 @ 式	排他的論理和	"単項式 ⊕ 式"というプレスブルガー算術で表現．ここで単項式，式は論理式．
単項式 & 式	論理積	"単項式 ∧ 式"というプレスブルガー算術で表現．ここで単項式，式は論理式．
単項式 || 式	連結	式のビット幅を n とすると，"$2^n \times$ 単項式 + 式"というプレスブルガー算術で表現．ここで単項式，式は算術式．
単項式 + 式	加算	"if 単項式 + 式 $< 2^n$ then 単項式 + 式 else 単項式 + 式 -2^n"というプレスブルガー算術で表現(オーバーフローを考慮)．ここで単項式，式は算術式．n は大きい方のビット幅．
単項式 - 式	減算	"if 単項式 − 式 >= 0 then 単項式 − 式 else 単項式 − 式 $+2^n$"というプレスブルガー算術で表現(アンダーフローを考慮)．ここで単項式，式は算術式．n は大きい方のビット幅．
単項式 >> 式	ビット右シフト	単項式のビット幅を a，式の値を b (式として定数のみ許す) とすると，シフトのオペランド(単項式) を a-b ビットと b ビットに分割した 2 つの変数 $(a1, a2)$ を新たに設ける．シフトの結果としては "$a1$" というプレスブルガー算術が返される．ここで $a1, a2$(単項式も) は整数変数である．また "単項式 $= 2^b \times a1 + a2$" という関係式を与える．
単項式 << 式	ビット左シフト	単項式のビット幅を a，式の値を b (式として定数のみ許す) とすると，シフトのオペランド(単項式) を b ビットと a-b ビットに分割した 2 つの変数 $(a1, a2)$ を新たに設ける．シフトの結果としては "$2^b \times a2$" というプレスブルガー算術が返される．ここで $a1, a2$(単項式も) は整数変数である．また "単項式 $= 2^{a-b} \times a1 + a2$" という関係式を与える．
単項式 == 式	一致判定	"単項式 = 式"というプレスブルガー算術で表現．= の表現は，両者が算術式の時は=でよいが，論理式の時は $\neg\oplus$ による．
単項式 != 式	不一致判定	"¬(単項式 = 式)"というプレスブルガー算術で表現．≠ の表現は，両者が算術式の時は¬ =でよいが，論理式の時は $\oplus$ による．
単項式 > 式	大小比較	"単項式 > 式"というプレスブルガー算術で表現．単項式，式は算術式．
単項式 >= 式	同上	"単項式 >= 式"というプレスブルガー算術で表現．単項式，式は算術式．
単項式 < 式	同上	"単項式 < 式"というプレスブルガー算術で表現．単項式，式は算術式．
単項式 <= 式	同上	"単項式 <= 式"というプレスブルガー算術で表現．単項式，式は算術式．
単項式	表 2 参照	表 2 参照

表 1: 式の表現

メント) 及び減算 (デクリメント) については，SFL ではオーバー (アンダー) フローが起きると，桁上げを考慮せず，2^n(n はビット幅) を法とする剰余を結果の値とする言語仕様になっている．演算結果が変数の上限値より大きくなった (下限値より小さくなった) 時は，演算結果より上限値を減ずる (加える) ことをプレスブルガー算術で表現する．

4 EFSM 上の CTL モデル検査アルゴリズム

本章では，EFSM 上で充足すべき性質が満足されているかどうかを形式的に検証する方法について提案する．従来の CTL モデル検査法は Kripke 構造で表現される回路と CTL 式で表された性質を入力とし，その手順は

1. 基本アルゴリズムによって CTL 式が成立する状態空間を計算
2. 求めた状態空間に初期状態空間が包含されることを示す事で，任意の回路動作でのその性質の充足性を証明

というものであった．本稿では EFSM で表現される回路と拡張 CTL 式で表された性質をモデル検査の入力とし，上下限のあるレジスタを持つ EFSM 上での基本アルゴリズムを述べる．アルゴリズムの特徴として，EFSM の制御部レジスタの値ごとに CTL 式が成立する状態空間を計算する．これによって，EFSM のある一つの制御部状態に注目した性質の検証を行いたい時には，その制御部状態についての状態空間計算のみを行う．また遷移関係関数を制御部状態ごとに分割しており，注目する制御部状態に関係しない遷移関係関数を考慮する必要がない．これらによって，モデル検査の効率化を図る事が出来る．また範囲を限定された整数を扱う上で，計算される状態空間はその範囲内にあるものでなければならない．提案手法では，存在限定子の処理中に整数変数の範囲を考慮すること，及び単項式における演算結果が整数変数の範囲となることでそれを保証する事ができる．

4.1 基本アルゴリズム

基本アルゴリズムは，与えられた CTL 式が成立する状態空間を求めるためのものである．一つの CTL 式に対してその式が成立する状態空間を求めるためには，その CTL 式の部分式のそれぞれについて式が成立する状態空間を求め，これを再帰的に繰り返す事によって与えられた CTL 式が成立する状態空間を計算する．基本アルゴリズムは，その部分式についての再帰的な状態空間計算のためのアルゴリズムである．以降では，CTL の部分式について，その式が成立する状態空間を求めるため，基底段階に対する処理，論理演算子の式に対する処理，時相演算子の式に対する処理，限定子が施されている式に対する処理の四つのアルゴリズムについて述べる．

なお，CTL 式が成立する状態空間を表現するため，及び EFSM の情報を格納するために，以下のようなデータ構造を用いる．

```
状態空間 charfunc_d {
        state_name;  制御部レジスタの値
        cond;        CTL 式が成立するために，その
                     制御部状態で成立すべき条件
    }

EFSM(遷移) strf_d {
        s_state;  始点の制御部状態名
        e_state;  終点の制御部状態名
        trf;      遷移関係関数
```

Syntax	Semantics	プレスブルガー算術での表現
¬単項式	否定(論理) 反転(算術)	単項式が論理式の場合，"¬ 単項式"というプレスブルガー算術で表現．算術式の場合， 単項式のビット幅をnとすると，"$2^n - 1 -$ 単項式"というプレスブルガー算術で表現．
/|単項式	桁方向の or	全てのbitが0のとき0，そうでない時1なので，"if (単項式 = 0) then False else True"というプレスブルガー算術で表現．ここで単項式は算術式．
/@単項式	桁方向の eor	単項式のビット幅をnとすると，全て1bitに切り出して$a1, \ldots, an$という変数を新たに設ける．演算結果としては"$a1 \oplus a2 \oplus \ldots \oplus an$"というプレスブルガー算術を返す．ここで$a1, \ldots, an$(単項式も)は整数変数である．また"単項式 = $2^{n-1} \times a1 + 2^{n-2} \times a2 + \ldots + an$"という関係式を与える．
/&単項式	桁方向の and	全てのbitが1のとき1，そうでない時0なので，"if (単項式 = $2^n - 1$) then True else False"というプレスブルガー算術で表現．ここで単項式はnビットの算術式．
/単項式	デコード	"if 単項式 = 0 then 2^0 else if 単項式 = 1 then 2^1 else if ...(単項式のmaxまで)"というプレスブルガー算術で表現．ここで単項式は算術式．
\ 単項式	エンコード	"if 単項式 = 0 then $2^{\log_2 n+1}$ else if ($2^0 \leq$ 単項式 $< 2^1$) then 0 else if ($2^1 \leq$ 単項式 $< 2^2$) then 1 else if ...(単項式のmaxまで)" というプレスブルガー算術で表現．ここで単項式はnビットの算術式．
結果の桁数 ‖ 単項式	符号拡張	符号拡張はある値をbit幅の異なる表現に変換するだけであり，値そのものは変化しない．よって"単項式"というプレスブルガー算術で表現．
要素 <最上位桁位置[:最下位桁位置]>	ビット切り出し	要素のビット幅をn，最上位桁位置をa，最下位桁位置をbとすると，要素を$n - a - 1$ビット，$a - b + 1$ビット，bビットに分割した3つの変数($a1, a2, a3$)を新たに設ける．切り出しの結果としては"$a2$"というプレスブルガー算術を返す．ここで$a1, a2, a3$(要素も)は整数変数である．また"要素 = $2^{a+1} \times a1 + 2^b \times a2 + a3$"という関係式を与える．
単項式 ++	インクリメント	"if 単項式 + 1 < 2^n then 単項式+ 1 else 0"というプレスブルガー算術で表現(オーバーフローを考慮)．ここで単項式は算術式．単項式のビット幅がn．
単項式 –	デクリメント	"if 単項式 − 1 > 0 then 単項式 − 1 else 2^n"というプレスブルガー算術で表現(アンダーフローを考慮)．ここで単項式は算術式．単項式のビット幅がn．
要素		データ端子，レジスタ，制御端子など

表 2: 単項式の表現

}

基底段階

基底段階とは，2.2節のCTLのsyntaxの定義の1.に現れる論理変数，論理定数，$(\tau < \tau), (\tau = \tau), (n|\tau)$，制御部レジスタ$= state_i$のいずれかである．基底段階が成立するための条件とは，基底段階の式自身である．よって，CTL式中の基底段階(ϕとする)について，制御部レジスタの値sにおいてCTL式が成立する状態空間の表現は，以下のアルゴリズムで実現できる．

各制御部レジスタの値sについて
{state_name = s;
cond = ϕ;}
である charfunc_d によって状態空間を表現．

論理演算子

本稿でいうCTLの論理演算子とは¬, ∨, ∧である．例えば¬のアルゴリズムは以下のように実現する．∨(∧)に関しては，cond = {CTL1->cond ∨ (∧)CTL2->cond}とする．

¬ CTL1

各制御部レジスタの値sについて
{state_name = s;
cond = ¬(CTL1->cond);}
である charfunc_d によって状態空間を表現．

時相演算子

CTL式をオペランドとする時相演算子の式に対し，その式が成立する状態空間の表現方法について述べる．EX,EG,EUの内，EUについて述べる．EX,EGについての元のアルゴリズムは[3]に書かれており，EUと同様に拡張している．ここでCTL1,CTL2はそれぞれ，オペランドとなるCTL式が成立する状態空間charfunc_dを表している．

E(CTL1)U(CTL2)

```
1 while (CTL2 が収束しない限り)
2     各制御部レジスタの値 s について {
3         for (始点が s である各遷移 t について) {
4             state_name = t->e_state である CTL2->
5                 cond について，次時刻変数へ置換した
6                 ものを用意 (CTL2->cond'とする);
7             α = t->trf ∧ CTL2->cond';
8             β = ∃v'α ∧ [現時刻を表す整数変数の範囲];
9             γ = [state_name = t->s_state である
10                CTL1->cond] ∧ β;
11        }
12        δ = [始点が s である各遷移 t についての⋁(γ)];
13        δ ∨ [state_name = s である CTL2->cond]
14            が，s で成立すべき条件;
15    }
16    CTL2 = 求めた状態空間;
17 }
```

アルゴリズムではまず4~10行目で，ある状態名sからの一つの遷移tについて，次時刻でCTL2が成立する可能性が存在し，かつsでCTL1が成立するために，sで成立しなければならない条件を計算している．また13行目で最初にCTL2がsで成立している場合を考慮している．

1行目の収束判定アルゴリズムは次の通り．

Uv が元の状態空間，Tv が一回計算後の状態空間とすると
for (EFSM の各状態 s ごとに)
{(Tv->cond → Uv->cond)
が全変数に対して真であるかどうかを判定 }

8行目の限定子の処理ではクーパーのアルゴリズム [4] が適用される．クーパーのアルゴリズムとは直観的には，$\exists x F(x)$(x は整数変数) に対して $F(x)$ が真となる可能性のある x の候補を抽出し，それぞれを x に代入した全ての論理和を求め，$F(x)$ と等価で x の現れない関数を求めるアルゴリズムである．提案手法では，8行目で整数変数の範囲を付加する事により，整数変数の場合その候補が設定した上下限の範囲内に収まり，一回の EU の計算によって求まる状態空間は整数変数に設定した範囲によって区切られる有限の空間になる．収束判定アルゴリズムで Tv → Uv の一方向のみを行っているのは，EU の繰返し計算によって状態空間は単調増加し，かつ求まる状態空間が有限である事が保証されているからである．

論理積などの演算では，計算結果が設定した範囲外に及ぶものであっても，その式に時相演算子が施されて次時刻の値を考える時や限定子が施される時には整数変数の範囲が考慮され，かつ最後の初期状態空間の包含判定の時に整数の範囲を前提条件とした判定を行う．加減算では剰余を演算結果とする仕様をプレスブルガー算術で表現するためオペランドの整数の範囲を考慮しているが，他の演算では演算ごとに整数の範囲を考慮する必要はない．

限定子

本稿でいう CTL の限定子は∃である．CTL 式をオペランドとして変数に限定子を施した式に対し，その式が成立する状態空間の表現は，以下のアルゴリズムで実現できる．ここで CTL1 はオペランドとなる CTL 式が成立する状態空間 charfunc_d を表しており，これらは既に求まっているものとする．また v は変数である．整数の範囲は，全ての整数変数に関する範囲の条件の論理積をプレスブルガー算術で表現したものである．

∃ v CTL1

各制御部レジスタの値 s について

{state_name = s;

cond = ∃v (CTL1->cond ∧ 整数の範囲);}

である charfunc_d によって状態空間を表現．

4.2 初期状態空間の包含判定

与えられた CTL 式を充足する状態空間を求めれば，求めた状態空間に初期状態空間が包含されるか否かを判定する．判定アルゴリズムは以下の通り

Uv が与えられた CTL 式を充足する状態空間，Tv が初期状態空間とすると，

初期状態について，

(整数の範囲 → (Tv->cond → Uv->cond))

が全変数に対して真であるかどうかを判定

整数の範囲は，全ての整数変数に関する範囲の条件の論理積をプレスブルガー算術で表現したものである．また，初期状態空間 Tv->cond は次のようにして求める事が出来る．

Tv->cond = (全レジスタの初期値に関する関係式)

4.3 アルゴリズムの実現

以上のアルゴリズムに従い検証システムを実現した．実現には我々の研究グループで開発したプレスブルガー算術処理ライブラリ及びデータサイト社の開発した SFL パーサを用いている．

一般に BDD では，根からの変数の順序により同じ関数の表現でもそのノード数に大きな差が生じる可能性があるという事が知られており，我々のプレスブルガー算術処理用のデータ構造でも同様である．今回のシステム実装においては [5] の実験結果に従い，入力端子の局所計算性及び出力端子に対する入力端子の影響力を考慮して，SFL 記述中に出現する順序を根からの変数順として採用し，SFL 記述からプレスブルガー算術への変換を行っている．また，全ての変数が冠頭に存在する限定子によって束縛されているプレスブルガー算術にクーパーのアルゴリズムを適用する場合，$\forall x_1 \forall x_2 ... \forall x_n F$ を $\forall x_n \forall x_2 ... \forall x_1 F$ として変数を処理する順番を変更しても式の値は変わらない．この順番を考慮する事により計算時間が大幅に変わる事が知られている．これは EU 計算時の $\exists v'$ の処理でも同様であり，変数を消去する順番により計算時間が大きく異なる．本手法では [5] の実験結果に基づき，EFSM の出力端子に対する依存度が大きい変数を先に消去する方針を取り，制御端子を表す論理変数を先に，データ端子を表す整数変数を後に消去している．

5 例題回路での評価

実装した検証システムに対し，SFL で記述した例題回路によって評価を行った．使用計算機は PentiumIII 600MHz，メモリ 512MB のものを用いた．結果を表 3に示す．回路名について，MPX はマルチプレクサ，DMPX はデマルチプレクサ，adder8 は 8bit 加算器，PENC はプライオリティエンコーダ，counter16 は 16bit カウンタ，mutex1 は 2 つのプロセスについて排他制御を行う調停回路である．仕様は拡張 CTL により整数を用いた表現で記述される．各仕様の検証に必要とした時間及び領域計算量は表の 3 列目，4 列目の通りである．領域計算量は検証に必要としたプレスブルガー算術処理ライブラリが用いたノード数で表現している．

検証の結果，例題回路のほとんどの仕様について，回路で正しく成立する事を 1 秒以内で検証する事が出来た．一方では，検証できなかった仕様や計算量が非常に大きくなっている仕様も見られる．これら

回路名	仕様	時間 (s)	領域
MPX	AG(if SEL = 0 then OUT_D = IN1 else if SEL = 1 then OUT_D = IN2 else if SEL = 2 then OUT_D = IN3 else if SEL = 3 then OUT_D = IN4);	0.016	941
DMPX	AG(if SEL = 0 then (OUT_D1 = IN1 and OUT_D2 = 0 and OUT_D3 = 0 and OUT_D4 = 0));	0.033	1130
	AG(if SEL = 1 then (OUT_D2 = IN1 and OUT_D1 = 0 and OUT_D3 = 0 and OUT_D4 = 0));	0.033	1115
	AG(if SEL = 2 then (OUT_D3 = IN1 and OUT_D1 = 0 and OUT_D2 = 0 and OUT_D4 = 0));	0.033	1091
	AG(if SEL = 3 then (OUT_D4 = IN1 and OUT_D1 = 0 and OUT_D2 = 0 and OUT_D3 = 0));	0.033	1029
adder8	AG(if (A_IN + B_IN + C_IN > 255) then ((S = A_IN + B_IN + C_IN - 256) and C = 1));	0.000	504
	AG(if (A_IN + B_IN + C_IN ≤ 255) then ((S = A_IN + B_IN + C_IN) and C = 0));	0.033	1989
PENC	AG(if (256 > IN1 and IN1 ≥ 128) then S=7);	0.016	1095
	AG(if (128 > IN1 and IN1 ≥ 64) then S=6);	0.016	1094
	AG(if (64 > IN1 and IN1 ≥ 32) then S=5);	0.016	1091
	AG(if (32 > IN1 and IN1 ≥ 16) then S=4);	0.016	1088
	AG(if (16 > IN1 and IN1 ≥ 8) then S=3);	0.016	1086
	AG(if (8 > IN1 and IN1 ≥ 4) then S=2);	0.016	1084
	AG(if (4 > IN1 and IN1 ≥ 2) then S=1);	0.016	1082
	AG(if (2 > IN1 and IN1 ≥ 0) then S=0);	—	mem out
counter16	AG(if RESET then AX(COUNT_TMP = 0));	0.283	5727
	AG(forall(n: if (not RESET and LOAD and DATA = n) then AX(COUNT_TMP = n)));	540.200	6147042
	AG(forall(n: if (not RESET and not LOAD and INC and COUNT_TMP = n) then AX(COUNT_TMP = n + 1)));	1.666	49212
	AG(forall(n: if (not RESET and not LOAD and not INC and COUNT_TMP = n) then AX(COUNT_TMP = n)));	1.150	32348
mutex1	not EF(c0 and c1);	0.050	1195
	AG(if (t0) then AF(c0));	0.266	3179
	AG(if (t1) then AF(c1));	0.283	3280

表 3: 例題回路の検証結果

は AG の被演算子に対する状態空間計算の結果が恒真にならず，AG 計算中の存在限定子処理においてノード数が大きくなったためと考えられる．提案手法は遷移関係関数を EFSM の状態ごとに分割し，その分割の上で基本アルゴリズムにより状態空間の計算を行っているので，本来制御部を持つような回路 (mutex1) に対して有効であると考えられる．

加算器やマルチプレクサはビット幅 8 ビットの回路に対する検証結果であるが，ビット幅を 16 にしても，検証にかかる時間計算量，領域計算量に変化は見られなかった．従来の CTL モデル検査法ではビット幅を長くするにつれて端子を表現するための変数の数も増加し計算量が大きくなることが予想されるが，提案手法では整数変数を使用するのでビット幅が変化しても変数の数が変わらないためである．よって提案手法はビット列を用いるようなシステムレベルの記述の回路に対して有効であると考えられる．

6 まとめと今後の課題

本稿では，SFL を用いて設計された回路に対し，その回路上で充足すべき性質が満足されているかどうかを形式的に検証する方法を提案した．今後の課題としては，EFSM の表現，並びに基本アルゴリズムでの限定子処理における最適な変数順の決定が挙げられる．今回採用した変数順は組合せ回路の実験において有効だったものであり，それが本当に EFSM の検証においても有効であるかどうかを再考したい．実装した CTL モデル検査法に対する改良手法の導入も課題の一つであり，例えば Forward State Traversal[6] や変数の依存関係を考慮して関係のない変数をあらかじめ消去する手法 [7][8]，CTL クラスの制限 [9] などはその有力な候補である．以上のような改良手法を考慮し，今後はより規模の大きい EFSM の検証を行いたい．

参考文献

[1] Minato, S.: *Binary Decision Diagrams and Applications for VLSI CAD*, Kluwer Academic Publishers (1996).

[2] 日本電信電話株式会社: PARTHENON HOME PAGE, http://www.kecl.ntt.co.jp/parthenon/index_j.htm.

[3] 平石裕実, 浜口清治: 論理関数処理に基づく形式的検証手法, 情報処理, Vol. 35, No. 8, pp. 710–718 (1994).

[4] 景山洋行, 北道淳司, 船曳信生: プレスブルガー文真偽判定アルゴリズムのための BDD の応用とそれを用いた回路検証, 情報処理学会論文誌, Vol. 40, No. 4, pp. 1578–1586 (1999).

[5] 景山洋行, 北道淳司, 東野輝夫: 高位の回路設計のためのプレスブルガー算術処理系の開発およびそれを用いた同期式順序回路の形式的検証, 第 15 回パルテノン研究会資料集, pp. 89–96 (1999).

[6] Iwashita, H., Nakata, T. and Hirose, F.: CTL Model Checking Based on Forward State Traversal, *International Conference on Computer Aided Design (ICCAD'96)*, pp. 82–87 (1996).

[7] Clarke, E., Jha, S., Lu, Y. and Wang, D.: Abstract BDDs: A Technique for Using Abstraction in Model Checking, *10th IFIP WG10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'99)*, LNCS 1703, pp. 172–186 (1999).

[8] Yang, J. and Tiemeyer, A.: Lazy Symbolic Model Checking, *37th Design Automation Conference (DAC'2000)*, pp. 35–38 (2000).

[9] Déharbe, D. and Moreira, A. M.: Symbolic Model Checking with Fewer Fixpoint Computations, *World Congress on Formal Methods in the Development of Computing Systems (FM'99)*, LNCS 1708, pp. 272–288 (1999).

[10] Bultan, T., Gerber, R. and Pugh, W.: Symbolic Model Checking of Infinite State Systems Using Presburger Arithmetic, *Computer Aided Verification, 9th International Conference (CAV'97)*, pp. 400–411 (1997).

[11] McMillan, K. L.: *SYMBOLIC MODEL CHECKING*, Kluwer Academic Publishers (1993).