

# 重要な産業問題に対するコンパクト BDDライブラリの新たな応用

## Novel Applications of Compact Binary Decision Diagram Library to Important Industrial Problems

### あらまし

米国富士通研究所 (FLA) は、ParDDと呼ばれるプロジェクトのもとで、BDD (Binary Decision Diagram) によるブール関数処理技術の研究開発を長年にわたり行ってきた。FLAはこれまでに、ブール関数を分割してコンパクトに表現することで、超並列計算プラットフォーム上で大規模なブール関数を効率良く処理できる技術を開発した。この技術は、EDA (電子機器設計自動化) の分野において、様々な目的に応用されてきた。

近年、FLAは、このプロジェクトのもとで、新しいBDDライブラリ (NanoDD) を開発した。これは、BDDの個々のノードのデータ量が、BDDの全体の容量によってほぼ決まるというものである。このコンパクトなデータ構造により、BDDがこれまでほとんど適用されなかった分野への応用が可能となる。例えば、NanoDDを用いてコンパクトな転置インデックス (inverted index) を作成した。転置インデックスは、Webなどのデータベースにおける文書の索引化に用いられる行列である。また、Webクエリの満足度、インターネットルータの重要なコンポーネントであるアクセス制御リスト (ACL) をコンパクトに表現し、NanoDDの性能分析を行った。

### Abstract

Fujitsu Laboratories of America has, over the course of many years, worked to develop the frontier of binary decision diagram (BDD) technology under a project called ParDD. Our technology allows us to partition Boolean functions, represent them very compactly, and process them on a massively parallel computing platform. It has been used to create numerous applications in the field of electronic design automation. Recently under this project we have developed a novel BDD library (NanoDDs) where the storage requirement at each node closely tracks the total size of the stored representation. The compact nature of this data structure allows the solution of interesting problems to which BDDs have seldom been applied before. For example, we have used NanoDDs to create a compact inverted index, an essential matrix for indexing documents in any corpus, including the World Wide Web. We have also characterized the performance of NanoDDs for Web query satisfaction as well as for the creation of compact representations of access control lists, a core component of Internet routers.



Stergios Stergiou

米国富士通研究所 所属  
現在、形式検証と論理合成アルゴリズム、およびWeb関連技術の研究開発に従事。



Jawahar Jain

米国富士通研究所 所属  
現在、多様性領域における記号処理技術の応用研究に従事。

## まえがき

コンピュータサイエンスの問題の多くは、ブール関数を用いて定式化することができる。BDD (Binary Decision Diagram)<sup>(1)</sup>は、ブール関数をコンパクトに表現するために用いられる非環状の有向グラフである。BDDは終端ノードと非終端ノードから成る。終端ノードは、ブール関数1および0を表す。各非終端ノードは、部分関数  $f$  に対応し、ブール変数  $v$  によりラベル付けされ、外に向かう2本の枝を持つ。1-枝は、関数  $v \cdot f$  を表す部分BDDを指し、0-枝は関数  $\bar{v} \cdot f$  の部分BDDを指す。これら2本の枝は異なるノードを指す。

ROBDD<sup>(2)</sup>は二つの付加的な制限を持つBDDである。第一に、ルートノードから終端ノードへ至るすべての経路上の変数順は同一でなければならない。第二に、同型の部分グラフが存在してはならない。これらの制限により、ブール関数の一意な表現形式が得られる。

BDDは、効率的に操作することができる。二つのBDD間のいかなるブール演算も、各BDDの大きさの高々二次の時間で完了することができる。BDDは、実際の応用における大多数の関数をコンパクトに表現できる。

コンパクトであること、そして効率的であることから、シミュレーション、合成、検証、テスト生成、人工知能、データマイニング、ソフトウェアセキュリティ、フォールトトレラントコンピューティングなどの分野において、様々な応用が可能である。

ROBDDは、実用的な多数の関数を効率良く表現する。しかし、応用によってはROBDDのサイズが指数オーダーにしかならないものもある。このため、ROBDDが処理できる問題の複雑さが、制限されている。

米国富士通研究所 (FLA) のParDDプロジェクトにおいて、Partitioned-ROBDD (POBDD) と呼ばれる、さらに効率の良い表現形式が開発された<sup>(3)-(6)</sup>。これは、とくに大規模な設計に有効である。この方法では、ブール空間を複数の領域に分割し、分割ごとに異なる変数順序付けを行うことができ、一度に一つの領域に対応したBDDをメモリ上に保持すれば十分である。

産業における大規模で複雑な設計を取り扱うため

に、FLAは、ROBDDを動的に分割する技術を開発した。これは、領域の分割数を動的に変化させることで、メモリの爆発的増大を回避する技術である。この方法による表現は、ROBDD、および領域の分割数が一定の方法に比べて指数オーダーでコンパクトであることが理論的に検証されている<sup>(5),(6)</sup>。

FLAは、この動的分割技術を応用した、不変条件検証 (invariant checking) に基づく到達可能性 (reachability)<sup>(7)</sup>判定、また、CTL<sup>(8),(9)</sup>のサブセットに対するモデル検証 (model checking) を開発した。また、この技術により、数学モデルの効率的な表現方法を確立し<sup>(10),(11)</sup> また、設計の誤りを検出するfalsificationという分野において、対称型マルチプロセッサ (SMP) アーキテクチャ<sup>(12)</sup>や大規模計算グリッド<sup>(13)</sup>を用いることで、並列度を超える (superlinear) 効率化を可能とする技術を開発することができた。

## NanoDDライブラリ

### ● NanoDDライブラリ

著者らは、ノードのデータ構造が固定化されたBDDではなく、必要となるブックキーピング情報を、OBDDの大きさの関数として、可能な限りコンパクトな形で管理する手法の開発を進めてきた。これにより、メモリ容量が大幅に節約される。また、BDDの演算に対する従来方式よりも効率の良い方式を考案した。さらに、著者らは、ZDD (Zero-Suppressed Binary Decision Diagram: 零抑制二分決定図)<sup>(14)</sup>と呼ばれるBDDの変形をサポートするために、ライブラリの設計および実装を新規に行った。ZDDは、ON集合が比較的疎なブール関数をより少ないノード数で表現できる。

変数の数を  $n$ 、BDDのノード数を  $d$  とすると、 $s_n = \lceil \log(n) \rceil$  ビットあれば、変数を表現できる。その上、ノードをメモリに連続的に配置する場合、 $s_d = \lceil \log(d) \rceil$  ビットあれば、その位置を特定できる。

上述の考察に基づき、NanoDD上の各ノードのデータ構造は次のようなる。

変数: $s_n$ -ビット	1-枝: $s_d$ -ビット	0-枝: $s_d$ -ビット
----------------	-----------------	-----------------

### ● 転置インデックス

転置インデックスは、文書の収集に用いられる

データ構造であり、特定のキーワードを含む文書の部分を効率的に特定するのに用いられる。転置インデックスは、リストの集合として保存することができ、各リストは一意のキーワード  $w_i$  に対応し、 $w_i$  を含む文書の識別子を含む。

転置インデックスの大きさは非常に大きくなる場合があり、必要となる記憶容量およびアクセス時間に直接影響する。したがって、多くの場合、素早くかつインクリメンタルに圧縮解除できるように、各リストを圧縮して保存する。

本稿では、NanoDDを用いた転置インデックスの表現方法について考察する。以下は、主流を占めるリスト圧縮法およびBDDの基礎的な情報である。

● NanoDDによる転置インデックス表現

各リストについて、対応するブール関数を構成し、そのBDDを従来のBDDパッケージ（とくに、zero-suppressed BDD）を用いて構築する。BDDによるリストの表現には二つの側面がある。第一の側面は、リスト要素のブール関数へのマッピングである。

(1) ブール関数としてのリスト

リスト [23, 33, 37, 54] をブール関数として表現する。2進表現では、リストの要素は [010111, 100001, 100101, 110110] である。

(2) 2進符号化

最小個数の変数によりリストを表現するブール関数は、各変数を各ビットに単に割り当てることで得られる。例えば、上のリストは関数

$$f = \bar{x}_1x_2\bar{x}_3x_4x_5x_6 + x_1\bar{x}_2\bar{x}_3\bar{x}_4\bar{x}_5x_6 + x_1\bar{x}_2\bar{x}_3x_4\bar{x}_5x_6 + x_1x_2\bar{x}_3x_4x_5\bar{x}_6$$

に対応する。

(3) 線形符号化

各文書IDに異なる変数を割り当てると、別の表現が得られる。しかし、文書の数はとても大きくなることがあるので、この表現は実用的ではない。また、複数のリストを一つのブール関数で表現しない限り、ノードを共有することもできない。

(4) ベース  $2^k$  符号化

$2^k$  ベースでリスト要素を表現すると、線形符号化および2進符号化を組み合わせることができる。ベース  $2^k$  のそれぞれの桁に、 $2^k$  の異なる変数を用いて、ワンホットで表現する。

例えば、数54を符号化する場合を考える。この

数はベース4では312である。各桁はワンホット符号化され、1000 : 0010 : 0100を得る。したがって、要素54は次のように符号化される。

$$g = x_1\bar{x}_2\bar{x}_3\bar{x}_4\bar{x}_5\bar{x}_6x_7\bar{x}_8\bar{x}_9x_{10}\bar{x}_{11}\bar{x}_{12}$$

この方式における変数の個数の増加は、非効率に見えるかもしれないが、実際は、とくにZDDを用いる場合は、より優れた共有とよりコンパクトな表現が得られる。

● 性能特性

(1) データベース (corpus)

NanoDDライブラリの性能評価を行うために、オンラインで入手可能なWebページの最大の集合について転置インデックスを作成した。このWebページの集合は、スタンフォード大学のWebBaseプロジェクトからダウンロードしたもので、9400万以上のWebページを含む。これに対して、最初のGoogleの実装は2500万のWebページを含むに過ぎない。

Webページごとに、テキストの用語を抽出した。50文字までのすべてのシーケンスを保存した。全部でほぼ220億の単語を処理した。1億1400万以上の一意的な単語ごとに、それが含まれるページの集合を計算した。すべての一意的な用語に対するすべてのページの集合が転置インデックスである。完全な転置インデックスの大きさは、従来の非圧縮リストの実装において、163 Gバイトである（表-1）。

(2) NanoDD転置インデックス

上記のWebページに対して、NanoDDベースの完全な転置インデックスを計算した。計算時間は25時間未満であった。これに対して、データベースの解析およびリストベースの転置インデックスの生成には、ほぼ4日間を要した。したがって、NanoDDによる転置インデックスの構築に要する時間は、無視することはできないが、問題ではないと考える。また、生成されたNanoDDによる転置インデックスの大きさは、従来のリストベースの方法で得られるものに比べて25%以下であった。

この結果が示すように、転置インデックスの保存に

表-1 転置インデックスの統計

処理を行ったページ数	9400万
一意的な単語	1億1400万
処理を行った単語	220億
リストに基づく転置インデックス	163 Gバイト

要するディスク空間が減少するのは大きな利点である。

### (3) 既存のBDDパッケージとの比較

NanoDDライブラリの性能を最新のBDDパッケージ (CUDD : Colorado University Decision Diagram) と比較するために、CUDDライブラリを用いてZDDベースの転置インデックスを生成するツールの実装も行った。

計算時間については、CUDDによる実装は、著者らのNanoDDの方法よりもほぼ8倍遅く実用的でないことが分かった。メモリ容量については、生成された転置インデックスは、著者らのNanoDDの方法に比べ6倍以上大きかった。

生成されたZDDの構造は二つの方法で同じであるため、メモリ量の増加はZDDの構造によるものではなく、ノードの構造によることが分かった。CUDDの場合、ノードあたりのサイズは少なくとも16バイトであり、生成されるZDDのノード数に依存しない。NanoDDの場合、ノードあたりのサイズは、生成されるZDDのノード数によって変わり、2から8バイトの間であった。

この比較においては、中間的に生成されるキャッシュに必要となるメモリ量は考慮されていない。なぜならば、キャッシュは結果の一部としては保存されないからである。

また、ACM/SIGSA 組合せ回路を用いて、NanoDDとCUDD<sup>(15)</sup>およびCal<sup>(16)</sup>との比較も行った。変数の順序は固定とし、回路を深さ優先探索して得られるものを用いた。いずれかのライブラリにより1800秒以内に計算できる自明でない回路のみを示す。結果を表-2に示す。

### NanoDDによるWebサーチの性能特性

NanoDDによる転置インデックスをWebサーチに用いることの最大の利点は、操作性の維持にある。

これを実現するために、Webサーチの処理により適した特別な操作方式を開発した。また、NanoDDの記憶方法を変更した。

### ● 演算の背景

K順序付けされたリストに実行される基本的演算は論理積であり、K重マージとして実装される。具体的には、リストの先頭から始まりすべての共通の要素が検出されるまで1回に一つずつ要素を読み出す。

例えば、つぎの二つのリストについて、共通の要素を検出する場合を考える。

リスト1 : 10, 20, 23, 36, 47, 52

リスト2 : 16, 18, 23, 47

リストの要素を指すポインタを  $p_1$ ,  $p_2$  とする。最初、ポインタはそれぞれ「10」と「16」を指す。 $p_1$ が指す要素は $p_2$ が指す要素よりも小さいので、 $p_1$ は要素「20」へ進む。すると、 $p_2$ が指す要素のほうが小さくなるので、 $p_2$ は「18」へ進む。「18」は「20」よりも小さいので、 $p_2$ は「23」へ進む。今度は、 $p_1$ が「23」へ進み、共通の要素として「23」を出力する。この処理を続けると、両ポインタはそれぞれ要素「47」へ進む。ここで要素「47」が出力される。 $p_2$ はリスト2の末尾に到達しており、これ以上要素がないので、処理を終了する。

リストのトレースにおいては、要素を一つずつ取り出す処理を行う。ここでは、リストLにおいて、値がelement以上である次の要素を取り出す処理として、`get_next_element_greq (L, element)`を実装する。

#### (1) `get_next_element_greq (L, element)`

NanoDDをトレースする際にアップデートされる変数への値の割当ての配列  $A$  を記録する。NanoDDに記録されている最初の要素は、深さ優先探索 (depth-first traversal) によりルートノードから終端ノード1まで0-枝を先にたどることで得られる。探索したノードごとに、変数IDおよび探索した

表-2 ACM/SIGDA回路に対するNanoDDとCUDD 2.4.2およびCal 2.1との比較

回路 名前	回路		NanoDD		CUDD					Cal				
			時間 (秒)	メモリ (Mバイト)	時間 (秒)	メモリ (Mバイト)	$R_t$ (%)	$R_m$ (%)	$R_p$ (%)	時間 (秒)	メモリ (Mバイト)	$R_t$ (%)	$R_m$ (%)	$R_p$ (%)
C3540	50	22	0.49	64.3	0.59	64	120	100	120	1.05	52.1	214	81	174
i10	257	224	0.5	63.2	0.62	74.2	124	117	146	1.19	64.7	238	102	244
C6288	32	32	225.17	5149.9	402.36	6312.6	179	123	219	468.67	6278.2	208	122	254
C2670	233	140	9.05	285.6	21.17	402.6	234	141	330	22.79	564.2	252	198	497

$R_t$  : 時間比

$R_m$  : 空間比

$R_p = R_t * R_m$

枝のidを記録する。ルートから終端ノード1までの経路に現れない変数には、最初に値0を割り当てる。

`get_next_element_greq (L, element)` が呼び出されると、配列  $A$  に対して `element` の2進表現と共通な割当てを持つ変数をルートから探索する。このアルゴリズムは、割り当てられた値が最初に異なる変数までバックトラッキングを行い `element` が指定する残りの割当てに従って NanoDD をトレースする。

図-1の単純なBDDで `get_next_element_greq` がどのように動作するかを示す。

図-1のBDDは関数  $f = x_1x_3x_4 + x_1\bar{x}_3\bar{x}_4$  を表しており、リスト [8, 11, 12, 15] を符号化している。

最初の要素は図-2に示すトレースから得られる。変数の割当ては  $(x_1, x_2, x_3, x_4) = (1, 0, 0, 1)$  であり、リストの最初の要素「8」が得られる。

リストの次の要素にアクセスする場合は、`get_next_element_greq (L, 9)` により「8」よりも大きな次の要素を検索する。 $(1, 0, 0, 0)$  と  $(1, 0, 0, 1)$  の最初の三つの変数の割当てが同じなので、変数  $x_3$  に対するバックトラックを行ったのち、引き続き次の経路を探索する (図-3)。その

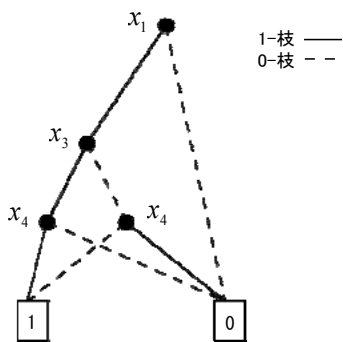


図-1 単純なBDDの例  
Fig.1-Example of simple BDD.

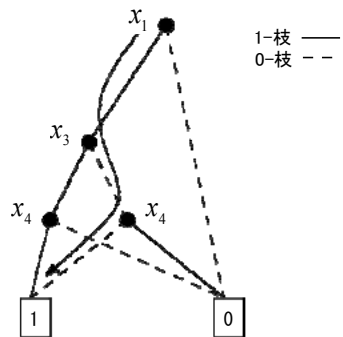


図-2 最初の要素を求める  
Fig.2-First element is obtained.

結果、つぎの割当て  $(1, 0, 1, 1)$  が求められ、要素「11」が得られる。同様にして、リストの残りの要素を得ることができる。

検索にBDDを用いる利点は、リスト中の不要な要素に対する検索をスキップできるということである。

例えば、二つのリスト [8, 11, 12, 15] と [7, 13, 15] の積集合を求める場合を示す。まず、双方のリストの最初の要素「8」、「7」を得る。「8」は「7」よりも大きいので、第二のリストで「8」以上の次の要素を検索し、要素「13」を得る。つぎに、`get_next_element_greq (L, 13)` を第一のリストに適用する。このとき、 $(1, 1, 0, 1)$  («13」に対応) と  $(1, 0, 0, 0)$  («8」に対応) では、最初の変数の割当てのみが一致するため、変数  $x_1$  から改めて、指定された割当て  $(1, 1, 0, 1)$  よりも小さくならないように NanoDD を探索し、最終的に  $(1, 1, 1, 1)$  を得る。

### ● 記憶方法

単一の NanoDD ノードには  $2s_d + s_n$  ビット必要である。1-枝の前に0-枝をたどる場合、深さ優先横断がノードを訪問する順番で、ノードはメモリまたはディスクに連続的に記憶される。このようにして、ディスク上の NanoDD の情報をインクリメンタルに取り出すことができる。終端ノードには仮想的な固定値を割り当てることで、終端ノードを明示的に記憶する必要はない。

### ● ParDDWebサーチクエリの性能特性

#### (1) AOLクエリデータ

最近、AOLは、検索エンジン (本質的にGoogleのフロントエンド) で実行された検索クエリの匿名情報を公開した。これは実際のユーザクエリであるので、著者らの演算性能を検証するための良い題材となる。

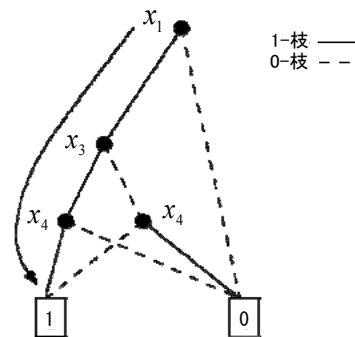


図-3 変数  $x_3$  へのバックトラック  
Fig.3-Backtrack to variable  $x_3$ .

(2) 実験結果

クエリの複雑度とシステムの性能の相関を調べるため、 $k = [2..6]$  の場合に、 $k$  語以上のAOLクエリ集合からランダムに抽出した10 000個のクエリについて、平均のクエリ時間を求めた。10 000クエリの集合ごとに、NanoDDベースとリストベースの操作コードを実行する。性能向上の様子を表-3に示す。

● エネルギー分析

抽象的なレベルでは、検索エンジンの典型的な計算ノードのエネルギー消費は、二つの要素から成る。第一の要素はシリコン（CPU、チップセット、メモリ）に関するものであり、第二の要素はハードディスク駆動に関するものである。CPUについては60 W、ハードディスク駆動については8 Wの平均電力消費を想定する。

転置インデックスのサイズはリストベースに比べ25%以下になるので、リストベースの転置インデックスについては92 W、NanoDDベースについては68 Wの平均電力消費を想定する。したがって、NanoDDの電力消費はリストベースの74%である。

表-3に示す性能向上率を考慮すると、エネルギー消費量は、表-4に示すように低下することが分かる。冷却要件が軽減される可能性が高いので、ランニングコスト削減に関する効果も期待できる。

ParDD技術を用いたアクセス制御リストの圧縮

● 問題の定式化

インターネットルータアクセス制御の問題の概略

表-3 リストベースをNanoDDベースにしたときの性能向上

$k$	性能向上
2	115%
3	125%
4	130%
5	140%
6	150%

表-4 リストベースをNanoDDベースにしたときのエネルギー消費の低減

$k$	エネルギー消費 (NanoDDベース/ リストベース)
2	64%
3	59%
4	57%
5	53%
6	49%

を以下に示す。すなわち、ルータを通過することを許可されていないパケットの送信元と送信先のIPアドレスの対のリストを維持する。このリストを  $L = \{ \langle IP_s, IP_d \rangle_i \}$  とする。通常、IPアドレスは32ビット長の整数なので、各対は64ビットの数で示される。

各対  $i$  について、64個の変数による最小項 (minterm)  $m_i$  を構成する。例えば、対

1101111010101111001000000000111  
0010000000001101101111010101111

は最小項

$$x_1 x_2 \bar{x}_3 x_4 x_5 \cdots \bar{x}_{60} x_{61} x_{62} x_{63} x_{64}$$

により表される。引き続き次のように、関数  $f$  を構成する。

$$f(x_1, \dots, x_{64}) = \bigvee_{i=1}^{|L|} m_i$$

つぎに、 $f = \bigvee_{i=1}^{|L|} p_i$  の形で  $f$  の積和 (SOP : Sum-of-Product) 表現が得られるように、関数  $f$  についてのZDDを構築し、得られたZDDに対して深さ優先検索を実行する。それぞれの積項  $p_i$  は、64個の変数のサブセットから成り、三つの記号 (0, 1, X) を用いた位置表記で示すことができる。例えば、積項  $x_1 \bar{x}_4 x_6$  は次のように位置表記で示される。

1XX0X1XXX...XX

送信元、送信先IP組  $T$  を用いて、ACLサブシステムは、対応するパケットを転送するか否かを決定しなくてはならない。この処理は、 $T$  から得られる変数の割当てで関数  $f$  を評価することで行う。例えば、 $T$  が2進表記で10011...10であるとき、 $f(1,0,0,1,1,\dots,1,0)$  を計算する。

● 実験による評価

(1) ボーダーゲートウェイプロトコル

ボーダーゲートウェイプロトコル (BGP) はインターネットの核心をなすルーティングプロトコルである。自律システム (AS) 間のネットワーク到達可能性を表すIPネットワークまたは「プレフィックス」の一覧表を維持する。BGPは、経路ベクトルプロトコルとして記述される。BGPは、経路、ネットワークポリシー、規則群に基づいてルーティングの意思決定を行う。AT&Tのネットワークに含まれる200 000個の規則を収集して関数

$f$  を構築し、本手法を評価した。

## (2) 実験結果

NanoDDライブラリを用いて関数  $f$  の計算と保存を行った。圧縮率は400%以上であった。本手法は、BDDから情報を抽出するコストが、得られた圧縮の利得を上回るもので、主にハードウェア加速によるアクセス制御リストのサポートがない場合に有用である。

## む す び

FLAにおいて、ParDDプロジェクトは、最初にブール関数を分割し、コンパクトに表現し、超並列計算プラットフォームで効率的に処理を行う技術の開発に注力した。この技術は、電子機器設計自動化(EDA)における様々な問題に応用された。著者らは、EDA以外の分野でBDDの応用先を見つけるために、新しいBDDライブラリ(NanoDD)を開発した。これは、共有OBDDの大きさに合わせてノード構造を動的に調整するものである。このデータ構造はコンパクトなので、BDDがまれにしか適用されなかった興味深い問題へのソリューションが得られた。

ノードの大きさがコンパクトであることから、メモリの占める量が小さいことに加えて、NanoDDは、極めて高速なBDD操作を実現することが、ACM/SIGDAベンチマーク回路および富士通独自設計により検証された。BDDが様々な問題に使用されることを考えると、メモリ量と処理時間の双方の効率改善は、FLAのBDD技術力の高さを示す説得力ある証拠である。

## 参考文献

- (1) S. B. Akers : Binary decision diagrams. *IEEE Transactions on Computers*, Vol.C-27, No.6, p.509-516 (1978).
- (2) R. E. Bryant : Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, Vol.35, No.8, p.677-691 (1986).
- (3) J. Jain et al. : Functional partitioning for verification and related problems. In Proceedings of Brown/MIT VLSI Conference, 1992.
- (4) A. Narayan et al. : Reachability analysis using partitioned-robbdds. In Proceedings of IEEE/ACM International Conference on Computer-Aided Design, 1997.
- (5) A. Narayan et al. : Partitioned-robbdds - a compact, canonical and efficiently manipulable representation for boolean functions. In Proceedings of IEEE/ACM International Conference on Computer-Aided Design, 1996.
- (6) B. Bollig et al. : Partitioned bdds vs. other bdd models. In Proceedings of the International Workshop on Logic and Synthesis, 1997.
- (7) O. Coudert et al. : A unified framework for the formal verification of sequential circuits. In International Conference on Computer Aided Design, 1990, p.126-129.
- (8) E. M. Clarke et al. : Design and synthesis of synchronization skeletons using branching time temporal logic. In Proc. IBM Workshop on Logics of Programs, volume 131 of Lecture Notes in Computer Science, 1981, p.52-71.
- (9) K. L. McMillan : Symbolic model checking. In Kluwer Academic Publishers, 1993.
- (10) S. Iyer et al. : On partitioning and symbolic model checking. In Proceedings of the International Symposium of Formal Methods, 2005.
- (11) S. Stergiou et al. : Disjunctive transition relation decomposition for efficient reachability analysis. In Proceedings of the IEEE International High Level Design Validation and Test Workshop, 2006.
- (12) D. Sahoo et al. : Multi-threaded reachability. In Proceedings of the 42nd Design Automation Conference, 2005.
- (13) S. Iyer et al. : Under-approximation heuristics for grid-based bmc. In Proceedings of the 4th International Workshop on Parallel and Distributed Methods in Verification, 2005.
- (14) S. Minato : Zero-suppressed bdds for set manipulation in combinatorial problems. In Design Automation Conference, 1993, p.272-277.
- (15) F. Somenzi : CUDD : CU decision diagram package—release 2.4.2, 2009.
- (16) R. Ranjan : CAL : Binary decision diagram package—release 2.1, 1998.