Grzegorz ŁABIAK
INSTYTUT INFORMATYKI I ELEKTRONIKI, UNIWERSYTET ZIELONOGÓRSKI

Koncepcja iteratora symbolicznego w środowisku BDD

Dr inż. Grzegorz Łabiak

Dr inż. Grzegorz Łabiak pracuje w Instytucie Informatyki i Elektroniki Uniwersytetu Zielonogórskiego jako adiunkt. Zainteresowania naukowe koncentrują się wokół metod projektowania układów cyfrowych, technik programowania oraz formalnych metod weryfikacji systemów dyskretnych. Jest autorem i współautorem licznych publikacji o zasięgu krajowym i międzynarodowym.

e-mail: G.Labiak@iie.uz.zgora.pl

Streszczenie

W artykule przedstawiono koncepcję iteratora symbolicznego, którego zadaniem jest poruszanie się po zbiorze stanów, symbolicznie zapisanym funkcją logiczną. Całość zaimplementowana jest diagramami BDD z pakietu CUDD, który udostępnia zestaw funkcji i klas w języku C/C++. Zadaniem iteratora jest generowanie mintermów funkcji, co polega na przeglądaniu drzewa BDD funkcji i produkowaniu kostek. Dla każdej kostki algorytm buduje term składający się z brakujących zmiennych, tworząc w ten sposób pełny minterm. W pracy również przedstawiono klasę obiektu iteratora oraz przykładowe wykorzystanie iteratora.

Słowa kluczowe: iterator, BDD, metody symboliczne, stan, zbiór stanów

Idea for symbolic iterator in BDD environment

Abstract

In the paper the concept of symbolic iterator has been presented. The iterator has a task to proceed through a set of states symbolically represented by logic function. The whole is implemented in C/C++ language in CUDD BDD package [14]. Every state from the set corresponds to minterm from the function which is subsequently generated by the iterator. Minterm generation consists in searching BDD tree and producing cube terms covering function minterms [8, 11]. Next, for each cube, the iterator algorithm generates simple term built from variables which does not appear in the cube. Full function minterm is a product of cube and term. Missing term generation is carried out by dedicated special next term function (Fig. 5c) and consists in image computation according to traditional algorithm [4, 5]. Apart from minterm iterator class declaration (Fig. 3), in the paper has been also presented a piece of C++ code demonstrating usage of the symbolic iterator (Fig. 2).

Keywords: iterator, BDD, symbolic methods, state, set of states

1. Wprowadzenie i motywacje

Weryfikacja, synteza i optymalizacja układów cyfrowych [11] jest dziedziną nauki i techniki, gdzie swoje zastosowanie znajdują metody i algorytmy matematyki dyskretnej. Szczególne znaczenie ma pojęcie zbioru (stanów) oraz algebra Boola, która jest nie tylko podstawowym językiem opisu układu cyfrowego ale też stanowi środek specyfikowania jego dynamicznych aspektów.

Optymalizacja układu cyfrowego, rozumiana jako część syntezy, sprowadza się do procesu znalezienie równoważnej postaci układu cyfrowego, gdzie kryterium jest zmniejszenie zużycia dostępnych zasobów sprzętowych, bądź szybkość przełączania. Przykładami metod optymalizacji i ich zastosowań są dekompozycja funkcjonalna [9], synteza układów kombinacyjnych i kontrolerów [3, 15, 16], synteza kontrolerów opisanych sieciami Petriego [1, 2] czy diagramami statechart [8].

Wspólną cechą przedstawionych powyżej zagadnień jest to, że można je sprowadzić do problemów logicznych, gdzie, zwłaszcza w kontekście systemów stanowych, funkcja boolowska symbolicznie reprezentuje zbiór stanów, a same algorytmy określane są mianem metod symbolicznych. Tego rodzaju algorytmy znakomicie implementuje się w środowisku binarnych diagramów decyzyjnych [8, 10, 11]. Przykładem takiego środowiska jest pakiet CUDD [14, który oferuje programiście interfejs w postaci bibliotek funkcji języka C i klas języka C++.

Zatem, w celu efektywnej programowej implementacji algorytmów weryfikacji, syntezy i optymalizacji logicznej, istnieje potrzeba dostarczenia programiście takiego obiektu, który umożliwiałby przegląd każdego elementu symbolicznego zbioru stanów, bez potrzeby znajomości jego wewnętrznej struktury. W dziedzinie inżynierii oprogramowania wzorcem projektowym takiego obiektu jest iterator [6]. W niniejszym artykule, ze względu na specyficzne zastosowanie i drobne uproszczania, dla odróżnienia od klasycznego wzorca projektowego, opisywany iterator zwany jest iteratorem symbolicznym.

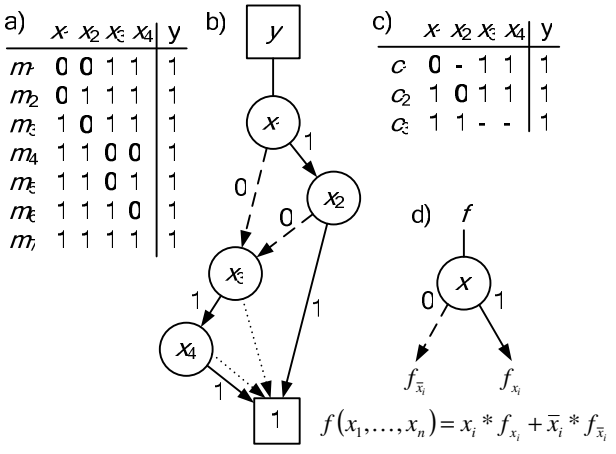
2. Przedstawienie problemu

Funkcja logiczna symbolicznie reprezentuje zbiór stanów, gdzie każdy jej minterm identyfikuje jeden stan. Przykładowo w systemie, który jest interpretowaną siecią Petriego, z każdym miejscem w sieci związana jest jedna zmienna logiczna, a jej wartość 1 symbolicznie oznacza aktywność miejsca. Wówczas funkcja zbudowana z sumy takich mintermów jest funkcją charakterystyczną zbioru stanów systemu [12], a gdy mintermy funkcji obejmują wszystkie możliwe przypadku aktywności sieci Petriego, funkcja taka jest funkcją charakterystyczną przestrzeni stanów globalnych (indykatorem przestrzeni stanów).

Jednym z efektowniejszych sposobów reprezentowania funkcji logicznych w pamięci komputera są binarne diagramy decyzyjne [10, 11, 14]. Binarny diagram decyzyjny jest drzewem binarnych decyzji, w którym z każdym węzłem związana jest zmienna boolowska, a krawędzie odpowiadają decyzjom (1 lub 0) będącym wartościowaniem zmiennej w węźle. Formalnie, każdą funkcję można rozwinąć w następujący szereg, zwany rozwinięciem Shannona [10, 11, 13]:

$$f(x_1, x_2, \ldots, x_n) = x_i * f_{x_i} + \overline{x}_i * f_{\overline{x}_i} \qquad (1)$$

gdzie $f_{x_i} = f(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n)$ i $f_{\overline{x}_i} = f(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n)$ są zwane, odpowiednio, pozytywnym oraz negatywnym dopełnieniem algebraicznym funkcji f ze względu na zmienną x_i. Równaniu (1) odpowiada węzeł drzewa BDD pokazany na rys. 1d.



Rys. 1. Trzy postaci funkcji logicznej $f=x_1*x_2+x_3*x_4$: a) zbiór mintermów, b) drzewo BDD, c) zbiór kostek i d) węzeł BDD
Fig. 1. Three forms of the logic function f=x1*x2+x3*x4: a) a set of minterms, b) a BDD tree, c) a set of cubes and d) BDD node

Stosując rekursywnie rozwinięcie Shannona, można stworzyć diagram BDD reprezentujący dowolną funkcję logiczną. Zredukowanym uporządkowanym binarnym diagramem decyzyjnym (*ang*. Reduced OBDD) jest diagram OBDD, z którego usunięte

zostały węzły nadmiarowe. Do zamiany diagramu OBDD na diagram ROBDD stosuje się następujące reguły redukcji [10, 11]:

- węzły których krawędzie wychodzące wskazują na ten sam węzeł potomny – są usuwane,
- podgrafy izomorficzne – są współdzielone.

Zastosowanie reguł redukcji sprawia, że pamięciowa zajętość diagramu staje się mniejsza. Przykładowo na rys. 1a przedstawiono funkcję $f=x_1*x_2+x_3*x_4$ i jej zredukowane i uporządkowane drzewo BDD (rys. 1b) o porządku zmiennych (x_1,x_2,x_3,x_4).

Jak widać funkcja reprezentująca zbiór stanów określona jest przez 7 mintermów (rys. 1a), co w pamięci komputera odpowiada drzewu BDD zbudowanemu z 5 węzłów (rys. 1b). Należy mieć na uwadze fakt, że postać diagramu BDD, a w tym ilość węzłów czyli zajętość pamięci w komputerze, silnie zależy od uporządkowania zmiennych, a przypadek z rys. 1b jest jednym z 4!=24 możliwych permutacji 4 zmiennych i jest to uporządkowanie optymalne. Interpretacja drzewa jest następująca: ścieżki prowadzące od korzenia do węzła z **1** wyznaczają zbiór kostek, których suma wyznacza zbiór mintermów funkcji. Na rysunku krawędzie w drzewie BDD wyrysowane linią ciągłą, związane z decyzją 1, oznaczają, że zmienna w węźle, z którego wychodzi krawędź, do kostki wchodzi w postaci prostej, natomiast krawędzie wyrysowana linią przerywaną oznaczają, że zmienna z węzła początkowego krawędzi, wchodzi do kostki w postaci zanegowanej. Krawędzie kropkowane, są to tzw. krawędzie negujące, co oznacza że wyrażenie logiczne związane ze wskazywanym poddrzewem przez taką krawędź, wchodzi do wyrażenia reprezentowanego przez drzewo w postaci zanegowanej. W przypadku diagramu z rysunku 1b zanegowane krawędzie pokazują na węzeł końcowy **1**, czyli w drzewie bez krawędzi zanegowanych dochodziłyby do końcowego węzła z **0**. Wprowadzenie do diagramu BDD krawędzi zanegowanej powoduje redukcję liczby węzłów w diagramie; w tym przypadku usunięty został 1 węzeł – węzeł końcowy z **0**. Jak widać przykładowemu diagramowi odpowiada zbiór 3 kostek (rys. 1c).

Zadaniem iteratora symbolicznego jest nie tylko przeglądanie drzewa BDD, ścieżka po ścieżce i generowanie kostek, ale też kolejne wygenerowanie wszystkich mintermów pokrywanych przez kostkę i następnie udostępnianie ich programowi wywołującemu. I tak kostka c_1 pokrywa mintermy m_1 i m_2, c_2 pokrywa tylko minterm m_3, a c_3 pokrywa m_4, m_5, m_6 i m_7.

3. Klasa iteratora

Iterator symboliczny ma za zadanie przeglądać cały zbiór mintermów funkcji i każdy kolejny minterm ma udostępniać w programie w miejscu gdzie są wywoływane jego funkcje. Chodzi o to, aby można było uzyskać efekt pokazany na rys. 2 (linie 10 i 11), czyli funkcja operatorowa (i cała klasa iteratora) musi być tak skonstruowana, aby stan obiektu iteratora był pamiętany pomiędzy jej kolejnymi wywołaniami.

```
1  #include "cuddObj.h"
2  #include "MintermIterator.h"

3  Cudd cuddMgr;
4  BDD x1 = cuddMgr.bddVar();
5  BDD x2 = cuddMgr.bddVar();
6  BDD x3 = cuddMgr.bddVar();
7  BDD x4 = cuddMgr.bddVar();
8  BDD y  = x1*x2+x3*x4;
9  CMintermIterator MIterator(y, cuddMgr);

10 while(MIterator++) {
       // udostepnianie mintermu
11     MItertor.GetMinterm().PrintMinterm();
   }
```

Rys. 2. Przykład wykorzystania iteratora symbolicznego
Fig. 2. The example of symbolic iterator usage

W linii 1 dołączany jest plik nagłówkowy z pakietu CUDD zawierający deklaracje klas menadżera BDD (`Cudd`) oraz zmiennej logicznej (`BDD`). W linii 2 dołączany jest plik z deklaracją klasy iteratora symbolicznego (`CMintermIterator`). W liniach od 3 do 7 ustanawiane są obiekty menedżera i zmiennych. W linii 8 utworzona jest zmienna `y`, będąca wyjściem funkcji, wraz z przypisaniem jej wyrażenia algebraicznego. Dzięki mechanizmowi przeciążania operatorów, znaczenie operatorów `*` i `+` w odniesieniu do obiektów klas `BDD` jest takie same jak ich znaczenie w algebrze Boola w odniesieniu do zmiennych logicznych. W linii 9 zdefiniowany jest obiekt iteratora (`MIterator`), który zainicjowany jest funkcją logiczną `y` i menadżerem `cuddMgr`. W linii 10 i 11 ma miejsce posługiwanie się iteratorem. Najważniejszą funkcja klasy iteratora jest funkcja przeciążająca operator przyrostkowy `++`. Każde wywołanie tej funkcji powoduje wygenerowanie kolejnego nowego mintermu i zwrócenie wartości `true`. W sytuacji gdy wszystkie mintermy już zostały wygenerowane, ponowne użycie funkcji operatorowej zwraca wartość `false`, co przerywa pętlę `while` (linia 10). Sam minterm dostępny jest poprzez wywołanie funkcji składowej `GetMinterm()` i w linii 11 zostaje on wyświetlony. Wnętrze pętli `while` jest miejscem dla algorytmów, które operują pojedynczym stanem (np. [8]), symbolicznie reprezentowanym przez wyrażenie logiczne.

Fragment deklaracji klasy przedstawiony jest na rys. 3.

```
1  class CMintermIterator {
2     BDD m_BDDCube;
3     BDD m_BDDMIterator;
4     BDD m_BDDTerm;
5     BDD m_BDDFunction;
          . . .
6     BDD _FirstCubeTerm();
7     BDD _NextCubeTerm();
8     BDD _FirstCube();
9     BDD _NextCube();
   public:
10    CMintermIterator();
11    BDD GetMinterm();
12    bool operator++(int);
          . . .
   };
```

Rys. 3. Fragment deklaracja klasy `CMintermIterator` w języku *C++*
Fig. 3. Piece of `CMintermIterator` class declaration in *C++* language

4. Algorytm iteracji

Działanie iteratora polega na przeglądaniu wszystkich ścieżek drzewa BDD prowadzących od korzenia do węzła z **1**, celem generowania kolejnych kostek. Następnie dla każdej kostki, odpowiadającej takiej ścieżce, generowane są kolejno wszystkie mintermy pokrywane przez tę kostkę, gdzie minterm m jest iloczynem kostki c i termu t: $m=c*t$. Całość zaimplementowana jest w funkcji operatorowej `operator++` i przedstawiona na rys. 4.

W stanie początkowym zmienna `m_BDDCube` ma wartość 0. W pierwszym bloku `if` (linie 1-8) generowany jest pierwszy minterm, a w linii 2 pierwsza kostka funkcji. Jeżeli funkcja ma wartość 0 działanie algorytmu się kończy (3,4). W przeciwnym razie, generowany jest pierwszy term, składający się ze zmiennych brakujących w pierwsze kostce, który jest częścią pierwszego mintermu (5). Term zapamiętywany jest dodatkowo w zmiennej pomocniczej `m_BDDTerm2` (6). Sam minterm jest iloczynem termu i kostki (7). W następnych wywołaniach funkcji operatora `++`, `m_BDDCube` jest różne od zera, wykonywana jest instrukcja 9, gdzie jest tworzony następny term zbudowany ze zmiennych brakujących w kostce. Funkcja `_NextCubeTerm()`, dla danej kostki generuje wszystkie termy. Gdy dla danej kostki term się powtórzy (10), w linii 11 trzeba wygenerować następną kostkę (lub gdy była to kostka ostatnia, linie 12 i 13, trzeba przerwać działanie) i następnie dla tej nowej kostki jest generowany pierw-

szy term (14) i tworzony minterm (16). Gdy dla danej nowej kostki z linii 9, term wygenerowany jest termem nowym, w linii 18 generowany jest minterm.

```
 1  if(m_BDDCube == 0) {
 2     m_BDDCube = _FirstCube()
 3     if(m_BDDCube == 0)
 4        return false;
 5     m_BDDTerm = _FirstCubeTerm();
 6     m_BDDTerm2 = m_BDDTerm;
 7     m_BDDMIterator = m_BDDCube*m_BDDTerm;
 8     return true;
    }
 9  m_BDDTerm = _NextCubeTerm()
10  if(m_BDDTerm == m_BDDTerm2) {
11     m_BDDCube = _NextCube;
12     if(m_BDDCube == 0)
13        return false;
14     m_BDDTerm = _FirstCubeTerm();
15     m_BDDTerm2 = m_BDDTerm;
16     m_BDDMIterator = m_BDDCube*m_BDDTerm;
17     return true;
    }
    else {
18     m_BDDMIterator = m_BDDCube*m_BDDTerm;
19     return true;
    }
```

Rys. 4. Algorytm iteracji zaimplementowany w funkcji operatorowej ++
Fig. 4. Iteration algorithm implemented in operator ++ function

Osobnego omówienia wymaga opis działania funkcji generujących kostki i termy. Funkcje `_FirstCube()`, `_NextCube()`, `_FirstCubeTerm()` swoje termy generują na zasadzie prostego przeglądania drzewa BDD funkcji. Inaczej jest w przypadku funkcji `_NextCubeTerm()`. Term jest tutaj określony jako iloczyn tych zmiennych funkcji, które są nieobecne w bieżącej kostce. Można zauważyć, że problem generowania termu następnego sprowadza się do skonstruowania funkcji logicznej $\mathbf{Y}_T$, będącej permutacją [12] z jednym cyklem [7], o tylu wejściach i wyjściach ile liczą zmienne w termie, która realizuje wzajemnie jednoznaczne przekształcenie zbioru termów na zbiór termów. Liczba wszystkich takich funkcji określona jest liczbą Stirlinga pierwszego rodzaju i wynosi $\begin{bmatrix} 2^n \\ 1 \end{bmatrix}$ [7], gdzie n oznacza liczbę zmiennych w termie. Wówczas problem obliczenia termu następnego, sprowadza się do obliczenia obrazu funkcji $\mathbf{Y}_T$ dla bieżącego termu. Wszystkie te działania łatwo są implementowane w środowisku BDD.

Zarówno dobór samej funkcji $\mathbf{Y}_T$ jak i algorytmu obliczania obrazu funkcji przekładają się na złożoność obliczeniową działania iteratora. W niniejszej pracy zagadnienie złożoności obliczeniowej nie było istotnym kryterium. W przypadku funkcji termu następnego $\mathbf{Y}_T$, przyjęto funkcję cechującą się pewną regularnością, a w przypadku algorytmu obliczania obrazu funkcji przyjęto znany algorytm podany niezależnie przez Burch'a [4] i Coudert'a [5].

Funkcja termu następnego została dobrana w ten sposób, że wartość liczbowa wektora termu następnego jest o 1 większa od termu bieżącego, za wyjątkiem termu t_8 (rys. 5a). Funkcje składowe wektora funkcyjnego $\mathbf{Y}_T$ (rys. 5d) przyjmują wartość 1, gdy: albo najwyższa zmienna istotna w wyrażeniu ma wartość 1, albo gdy pozostałe zmienne mają jednocześnie wartość 1 (rys. 5b). Dzięki takiemu doborowi funkcji, formuła uogólniona na wartość funkcji składowej (rys. 5c) przedstawia się w postaci łatwej do skalowania na dowolną liczbę zmiennych.

Obliczenie obrazu funkcji termu następnego ($\mathbf{Y}_T$) dla bieżącego termu t realizowane jest według następujących formuł [4, 5]:

$$t^{next} = \exists_x \left(t * \prod_{i=1}^{n} \left[x_i' \odot (t * y_i(x)) \right] \right) \tag{2}$$

$$t^{next} = t^{next} \langle x' \leftarrow x \rangle \tag{3}$$

gdzie y_i oznacza i-tą funkcję składową wektora funkcyjnego $\mathbf{Y}_T$, x_i' jest pomocniczą zmienną odpowiadającej i-tej zmiennej termu, n jest liczbą zmiennych w termie, a t^{next} jest termem następnym. Symbol $\odot$ oznacza operator XNOR, a symbol $\exists_x$ oznacza operację wygładzania funkcji, co nieformalnie odpowiada usunięciu zmiennej z wyrażenia. W wyniku operacji wygładzania w wyrażeniu na term następny występują tylko zmienne pomocnicze (w równaniu (2) oznaczone znakiem prim). Wyrażenie (3) oznacza zamianę zmiennych, po której term następny t^{next} zależy tylko od tych samych zmiennych co term bieżący t.

a)

	x_3	x_2	x_1	y_3	y_2	y_1
t_1	0	0	0	0	0	1
t_2	0	0	1	0	1	0
t_3	0	1	0	0	1	1
t_4	0	1	1	1	0	0
t_5	1	0	0	1	0	1
t_6	1	0	1	1	1	0
t_7	1	1	0	1	1	1
t_8	1	1	1	0	0	0

b) $y_1 = x_1 \oplus 1$

$y_2 = x_2 \oplus x_1$

$y_3 = x_3 \oplus x_1 * x_2$

c) $y_n = x_n \oplus \prod_{i=1}^{n-1} x_i$

d) $\mathbf{Y_T} = [y_1, \ \ldots \ , y_n]$

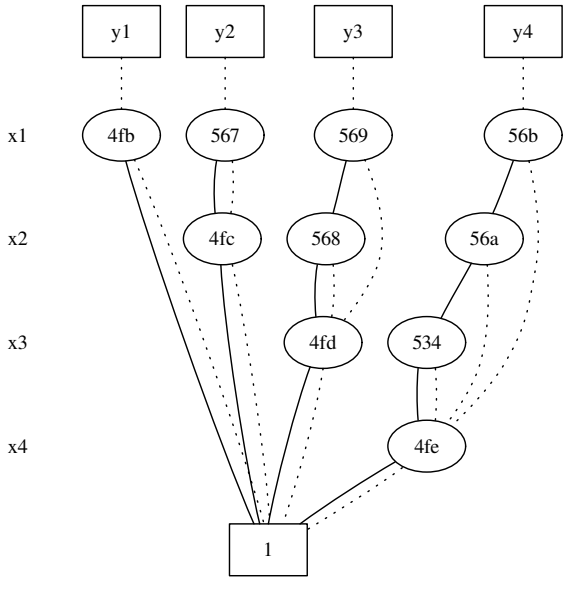
Rys. 5. Funkcja termu następnego 3 zmiennych: a) tabela prawdy, b) równania składowych, c) formuła ogólna, d) wektor funkcyjny
Fig. 5. Next term function for 3 variables: a) truth table, b) component equations, c) general formulae d) functional vector

5. Dyskusja na temat porządku zmiennych w funkcji stanu następnego

Powszechnie znanym faktem jest, że dla danej funkcji mogą istnieć różne diagramy BDD, różniące się porządkiem zmiennych w diagramie. Te różne postacie funkcji mogą znaczącą się różnić pod względem wielkości. Stąd, wydaje się interesującym pomysłem przebadanie wpływu uporządkowania zmiennych w diagramie najważniejszej funkcji iteratora symbolicznego – funkcji stanu następnego.

Według pozycji [15] dobre uporządkowanie zmiennych w diagramie, cechuje się następującymi właściwościami:

- Obliczeniowość lokalna (*ang*. local computability): sygnały wejściowe, które są ze sobą blisko związane, w porządku zmiennych powinny być blisko siebie.
- Moc oddziaływania na wyjście (*ang*. power to control the output): sygnały wejściowe, które silnie wpływają na wyjście powinny być umieszczone w BDD na wyższych poziomach (bliżej korzenia).

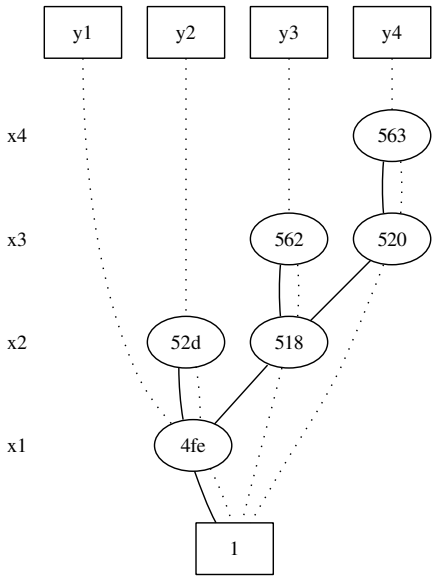


Rys. 6. Diagram BDD funkcji stanu następnego o porządku (x_1,x_2,x_3,x_4)
Fig. 6. BDD diagram of next term function for (x_1,x_2,x_3,x_4) order

Jednakże przedstawione powyżej dwie właściwości, dla danej funkcji, mogą być sprzeczne. Co więcej, w sytuacji gdy dwie lub więcej funkcji są reprezentowane w diagramie zarządzanym przez wspólnego menadżera BDD, tak jak to ma miejsce w przypadku iteratora symbolicznego, wówczas ich optymalne uporządkowanie może być dla każdej z nich inne. Wówczas, znalezienie wspólnego porządku jest pewnego rodzaju kompromisem.

W przypadku funkcji składowej y_4 dla funkcji stanu następnego 4 zmiennych (rys. 5c), zmienne x_1, x_2 i x_3 są ze sobą blisko powiązane i powinny być blisko siebie. O zmienna x_4 można powiedzieć, że silnie wpływa na wyjście funkcji, i jej właściwe miejsce znajduje się na początku porządku zmiennych. Jednakże w rzeczywistej programowej implementacji iteratora symbolicznego, przyjęty porządek zmiennych jest taki sam jaki jest w funkcji na której operuje iterator, stąd porządek ten jest następujący (x_1,x_2,x_3,x_4). Diagram BDD dla całego wektora funkcyjnego jest przedstawiony na rysunku 6. Całkowita ilość węzłów wynosi 11.

Diagram BDD dla najbardziej korzystnego uporządkowania, w którym zmienne znajdują się na swych zalecanych pozycjach jest przedstawiony na rysunku 7. Porządek ten jest następujący (x_4,x_3,x_2,x_1) a liczba węzłów obliczona przez funkcję `ReadNodeCount()` z pakietu CUDD wynosi 10. Niektóre z węzłów są liczone po kilka razy, np. węzeł nr 518 dwukrotnie a węzeł 4fe trzy razy.



Rys. 7. Diagram BDD funkcji stanu następnego przy uporządkowaniu najlepszym
Fig. 7. BDD diagram of next term function for best ordering

Zaproponowana funkcja stanu następnego (rys. 5), gdzie term następny jest numeryczną wartością zwiększoną o 1, szczęśliwie nie jest silnie zależna od uporządkowania zmiennych. Dla przypadku funkcji o 4 zmiennych, najlepsze uporządkowanie zużywa 10 węzłów, podczas gdy najgorsze uporządkowanie wymaga zaledwie 11 węzłów. Wszystkich permutacji z jednym cyklem funkcji stanu następnego o 4 zmiennych jest $(2^n-1)!=15!$.

5. Podsumowanie

Opisany iterator symboliczny spełnia swoje zadanie poruszania się po zbiorze stanów, symbolicznie przedstawionym wyrażeniem logicznym. Środowisko BDD, z pakietu CUDD, z powodzeniem implementuje opisany algorytm, a zaproponowany interfejs programistyczny w postaci klasy języka C++ sprawia, że posługiwanie się operatorem jest bardzo proste. Złożoność obliczeniowa algorytmu przede wszystkim zależy od doboru funkcji termu następnego $\mathbf{Y}_T$ oraz od algorytmu obliczającego obraz tej funkcji.

6. Literatura

[1] M. Adamski, *Parallel Controller Implementation using Standard PLD Software*, W. R. Moore, W. Luk, (red.), FPGAs, Abingdon EE&CS Books, Abingdon, England, ss. 296-304, 1991
[2] K. Biliński, *Application of Petri Nets in parallel controllers design*, PhD Thesis, University of Bristol, Electrical and Electronic Engineering Department, Bristol, 1996
[3] A. Bukowiec, A. Barkalov, *Synteza logiczna skończonych automatów stanów z zastosowaniem wielokrotnego kodowania stanów*, Przegląd Telekomunikacyjny i Wiadomości Telekomunikacyjne 2008, nr 6, ss. 766-769 [CD-ROM]
[4] J. R. Burch, E. M. Clarke, K. L. McMillan i D. Dill, *Sequential Circuit Verification Using Symbolic Model Checking*, Proceedings of the 27th Design Automation Conference, ss. 46-51, June 1990
[5] O. Coudert, C. Berthet and J. C. Madre, Verification of Sequential Machines Using Boolean Functional Vectors, IMEC-IFIP Int. Workshop on App. Formal Methods for Correct VLSI Design, ss. 111-128, Nov. 1989
[6] E. Gamma, R. Helm, R. Johnson, J. Vlisslides, *Design Patterns*, Addison Wesley 1995
[7] W. Lipski, *Kombinatoryka dla programistów*, WNT, Warszawa 1989
[8] G. Łabiak, *From statecharts to FSM-description - transformation by means of symbolic methods*, Discrete-Event System Design - DESDes '06: a Proc. from the 3rd IFAC Workshop, Rydzyna, Polska, 2006
[9] T. Łuba, *Synteza układów logicznych*, Wyd. WSISiZ, Warszawa 2001
[10] S.-I. Minato: *Binary Decision Diagrams and Applications for VLSI CAD*, Kluwer Academic Publishers, Boston, 1996
[11] G. de Mitcheli, *Synteza i optymalizacja układów cyfrowych*, WNT '98
[12] H. Rasiowa, *Wstęp do matematyki współczesnej*, PWN 1998
[13] C. A. Shannon, *A symbolic analysis of relay and switching circuits*, Massachusetts Institute of Technology 1937
[14] F. Somenzi, *CUDD: CU Decision Diagram Package*, http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html
[15] M. Wiśniewska, R. Wiśniewski, M. Adamski, *Usage of hypergraph theory in decomposition of concurrent automata*, "Pomiary, Automatyka, Kontrola", 2007, nr 7, ss. 66-68
[16] R. Wiśniewski, A. Barkalov, A. Janiak, *Methods of designing of compositional microprogram control units with mutual memory*, "Pomiary, Automatyka, Kontrola", 2008, Vol. 54, nr 8, ss. 493-495

Artykuł recenzowany