Grzegorz ŁABIAK, Tomasz MAKOWSKI
UNIWERSYTET ZIELONOGÓRSKI, INSTYTUT INFORMATYKI I ELEKTRONIKI

Binarne Diagramy Decyzyjne w technologii .NET

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

Mgr inż. Tomasz MAKOWSKI

Absolwent Wydziału Elektrotechniki, Informatyki i Telekomunikacji Uniwersytetu Zielonogórskiego. Zainteresowania naukowe koncentrują się wokół nowoczesnych technik programowania z wykorzystaniem języków obiektowych. Obecnie pracuje w kierowanej przez siebie firmie GRT STUDIO (grt.com.pl), gdzie zajmuje się projektowaniem kompleksowych rozwiązań IT dla przedsiębiorstw.

e-mail: tmakowski@grt.com.pl

Streszczenie

W referacie przedstawiono sposób adaptacji istniejącego pakietu BDD (CUDD), napisanego w języku C/C++ dla środowiska UNIX, na platformę .NET. Opracowane przejście opiera się w głównej mierze na wykorzystaniu bibliotek DLL. Uzyskane wyniki potwierdzają dobrą jakość opracowanej transformacji, czego dowodem jest uzyskanie porównywalnego rezultatu dekompozycji funkcji boolowskiej dla funkcji znanych z literatury.

Słowa kluczowe: BDD, DLL, Platform Invocation Service, .NET, C#.

Binary Decision Diagrams in .NET Technology

Abstract

The paper presents a method of implementation of BDD package in .NET platform, originally designed for UNIX operating system. Described transformation consists in using mainly Dynamic Link Library. Obtained results proved its usefulness, what is confirmed by the outcome for literature example of Boolean function decomposition, which turned out to be comparable.

Keywords: BDD, DLL, Platform Invocation Service, .NET C#.

1. Wstęp

Binarne Diagramy Decyzyjne są popularnym i bardzo wydajnym sposobem reprezentowania wyrażeń boolowskich w pamięci komputera. Pozwalają na bardzo efektywne wykonywanie wielu działań na wyrażeniach zarówno pod względem pamięciowym jak i czasowym, a także dzięki opracowaniu wielu bibliotek programistycznych, możliwe jest szybkie implementowanie algorytmów przetwarzających funkcje logiczne. Ze względu na naturę zastosowaniach dotychczasowymi platformami implementacyjnymi diagramów był język *C*/*C++*, a realizacje w środowiskach typu *Java* czy *.NET* praktycznie niespotykane.

2. Binarne diagramy decyzyjne

Binarne Diagramy Decyzyjne (*ang*. Binary Decision Diagrams) [5, 6] są acyklicznym zorientowanym grafem z jednym wyróżnionym węzłem zwanym korzeniem, do którego nie dochodzą żadne krawędzie oraz z dwoma węzłami końcowymi zawierającymi wartości zero i jeden, odpowiadającymi boolowskim funkcjom 0 i 1. Każdy węzeł nieterminalny posiada numer, który służy do skojarzenia węzła ze zmienną logiczną z reprezentowanego wyrażenia logicznego oraz od każdego takiego węzła odchodzą dwie krawędzie zaetykietowane zerem i jedynką.

Uporządkowanym *BDD* (*ang*. Ordered *BDD*) jest diagram, w którym zmienne boolowskie we wszystkich ścieżkach poprowadzonych od korzenia do węzła końcowego, występują w tym samym porządku i żadna zmienna w ścieżce nie pojawia się więcej niż jeden raz.

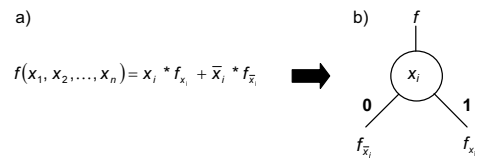
Każdą funkcję logiczną można rozwinąć w następujący szereg, zwany rozwinięciem Shannona [5, 6]:

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

gdzie

$$f_{x_i} = f(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n)$$
$$\text{i } f_{\overline{x}_i} = f(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n)$$

są zwane odpowiednio pozytywnym oraz negatywnym dopełnieniem algebraicznym funkcji f ze względu na zmienną x_i. Równanie (1) może zostać przedstawione w postaci *BDD* w następujący sposób:



Rys. 1. Idea *BDD*: a) rozwinięcie Shannona, b) węzeł *BDD*
Fig. 1. *BDD* idea: a) Shannon expansion, b) *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:

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

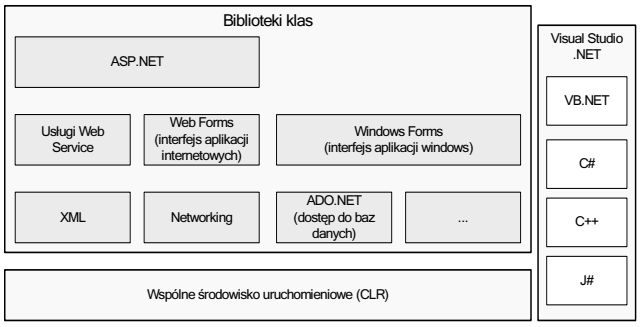
Jedną z bardzo udanych implementacji diagramów *BDD* jest pakiet *CUDD* (CU Decision Diagram) [8], który został stworzony prze Fabio Somenzi na wydziale Elektroniki i Inżynierii Komputerowej na Uniwersytecie Kolorado **(**Department of Electrical and Computer Engineering University of Colorado at Boulder**).** Pakiet ten dostarcza funkcji pozwalających na manipulacje drzewami *BDD* (w tym również zredukowanymi *BDD*). Pakiet ten pierwotnie został zaprojektowany dla systemu operacyjnego *UNIX*, jako zestaw funkcji w języku *C* i klas w języku *C++*.

3. Platforma .NET i środowisko programistyczne

Platforma *.NET* stanowi środowisko uruchomieniowe (*Runtime Environment*) dla aplikacji pisanych w różnych języka, cechujących się przenoszalnością niezależną do sprzętu i systemu operacyjnego, i stanowi odpowiedź firmy *Microsoft* na sukces technologii *Java*.

Podstawową zaletą technologii *.NET* jest niezależność od języka programowania i platformy, na której program ma być uruchomiony. Jest to związane ze zmianą sposobu wykonywania kodu. Kod źródłowy kompilowany jest do standardowego Języka

Pośredniego Microsoft w skrócie zwanym *MSIL* (*Microsoft Intermediate Language*). Podczas pierwszego uruchomienia aplikacji kod pośredni kompilowany jest przez wspólne środowisko uruchomieniowe *CLR* (*Common Language Runtime*) na kod maszynowy procesora na którym została uruchomiona aplikacja [9].



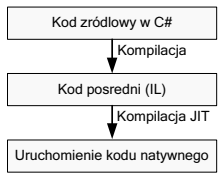
Rys. 2. Struktura platformy *.NET* [9]
Fig. 2. *.NET* platform structure [9]

Główne składniki platformy *.NET* pokazane na rysunku 2 to wspólne środowisko uruchomieniowe *CLR* oraz nowe hierarchicznie zorganizowane biblioteki klas.

Microsoft Visual Studio .NET to pełne środowisko programistyczne zawierające narzędzia graficzne, ułatwiające tworzenie kodu, śledzenie krokowe programu i kompilowanie kodu oraz organizację prac programistycznych. Dużą zaletą *Visual Studio .NET* jest możliwość korzystania w nowo stworzonym projekcie z kodu wcześniej już napisanego, nie zależnie od systemu operacyjnego, języka programowania i stosowanego modelu obiektowego.

Język *C#* jest językiem w pełni obiektowym. Łączy on w sobie najlepsze cechy języka *Java* i *C++*. Poprzez podobieństwo składni do obu tych języków łatwo pozwala programiście na zmianę języka programowania oraz łatwą konwersję kodu napisanego w *C++* lub *Java*.

Kompilator *C#* znajduje się w środowisku uruchomieniowym *.NET Framework*. W wyniku kompilacji otrzymujemy kod w języku pośrednim zwanym *IL*, który umieszczany jest w module binarnym. Podczas uruchamiania modułu uruchamiany jest kompilator *JIT* (*Just-In-Time*), w wyniku czego tworzony jest kod natywny systemu operacyjnego, a następnie jest on uruchamiany jak zwykła aplikacja. Schemat całego procesu jest pokazany na rysunku 3. Dzięki takiemu sposobowi uruchamiania prędkość działania aplikacji napisanej w *C#* porównywalna jest z aplikacją napisaną w *C++*.



Rys. 3. Schemat uruchomienia programu w systemie operacyjnym
Fig. 3. Program executing scheme under operating system

4. BDD na platformie .NET

Autorzy w swych pracach skupili się nad popularnym pakietem *CUDD*, który został zaprojektowany na platformę systemu operacyjnego *UNIX* z interfejsem w języku *C/C++*. Aby można było korzystać na platformie *.NET* z binarnych diagramów decyzyjnych z pakietu *CUDD* w warunkach systemu operacyjnego *Windows*, konieczne jest stworzenie biblioteki *dll* [7] eksportującej funkcje operujące na diagramach i następnie zaimportowanie tychże funkcji w środowisku *Visual Studio .NET*.

4.1. Eksport funkcji pakietu CUDD

Stworzenie biblioteki łączonej dynamicznie (**.dll*) w środowisku *Visual Studio* polega na utworzeniu odpowiedniego projektu, który wygeneruje odpowiednie ustawienia dla programu kompilatora i programu łączącego. W takim projekcie można wykorzystać albo wcześniej przygotowane, w oparciu o pakiet *CUDD*, biblioteki łączone statycznie (**.lib*) lub też można wykorzystać pliki źródłowe z pakietu *CUDD*, z których bezpośrednio zostanie wygenerowana biblioteka *dll*. Przeniesienie plików źródłowych z oryginalnego pakietu zostało zrealizowane w ramach projektu *Chios* [1, 3]. Trzeba tu zaznaczyć, że jedynymi komplikacjami w tej czynności, było zgłoszenie przez kompilator 4 ostrzeżeń *C4244* i 4 ostrzeżeń *C4146*, które w ogólności odnoszące do niedopasowania typów zmiennych. Liczne serie testów symulacyjnych, w ramach projektu *HiCoS*, wykazały, że nie miało to wpływu na poprawne funkcjonowanie pakietu w nowym środowisku *Windows*. Kolejną czynnością jest specyfikacja w pliku źródłowym biblioteki dynamicznej eksportowanych funkcji operujących na diagramach. Przykładowa deklaracja funkcji iloczynu wygląda następująco:

```
__declspec(dllexport) DdNode* Cudd_bddAnd_exp
        (DdManager *dd, DdNode *f, DdNode *g);
        {
                return Cudd_bddAnd (dd, f, g);
        }
```

Każda z eksportowanych funkcji składa się z przedrostka *__declspec(dllexport)*, który jest informacją dla programu łączącego, aby funkcja ta została dodana do tablicy eksportowanych funkcji pliku *dll*. Następnie umieszczony jest typ wartości zwracanej przez funkcję, poczym podana jest nazwa funkcji taka sama jak nazwa funkcji z pakietu *CUDD* plus przyrostek *_exp*. W ciele funkcji wywoływana jest funkcja, którą jest eksportowana. Następnie nazwę eksportowanej funkcji dodawana jest do pliku o rozszerzeniu *def* zawierającego nazwy eksportowanych funkcji. Dla powyższej funkcji taka deklaracja wygląda następująco:

```
EXPORTS
Cudd_bddAnd_exp @1
. . .
```

4.2. Import funkcji w środowisku Visual Studio

Aby w programach tworzonych w środowisku *Visual Studio* można było korzystać z funkcji umieszczonych w bibliotekach *dll*, trzeba skorzystać z usług technologii *Platform Invocation Services* zwanej w *.NET P/Invoke*. *P/Invoke* to zbiór usług *CLR*, które pozwalają na korzystanie z dowolnych języków programowania zgodnych z *.NET* poprzez udostępnianie wspólnych usług. *P/Invoke* pośredniczy więc przy wywołaniu natywnego kodu z aplikacji pracującej pod kontrolą wspólnego środowiska uruchomieniowego. Główne zadania *P/Invoke* to odnalezienie i załadowanie do pamięci pliku biblioteki z deklaracją danej funkcji oraz, po odnalezieniu adresu funkcji w pamięci, umieszczenie jej argumentów na stosie i przekazanie jej sterowania.

Aby móc skorzystać z wcześniej zaimplementowanych i wyeksportowanych funkcji trzeba je odpowiednio zaimportować. Deklaracja taka w języku *C#* wygląda następująco:

```
[DllImport("mcf.dll", CallingConvention = CallingConvention.Cdecl, ExactSpelling
= true)]
public static extern IntPtr Cudd_bddAnd_exp(IntPtr dd, IntPtr f, IntPtr g);
```

Przed deklaracją funkcji znajduje się atrybut *DllImport,* który zawiera niezbędne informacje dla *P/Invoke*. Obowiązkowym atrybutem jest tylko nazwa biblioteki z której chcemy korzystać, reszta atrybutów jest opcjonalna. Do dyspozycji mamy takie atrybuty jak:

- *CharSet* – odpowiada za przekazywanie łańcuchów miedzy zarządzanym a nie zarządzanym kodem.

- *EntryPoint* – definiuje nazwę funkcji w bibliotece *dll* w postaci jej nazwy bądź #numer funkcji. Jest to przydatne, gdy nie chcemy używać długich nazw funkcji.
- *SetLastError* – w wypadku ustawienia na true nakazuje *CLR* przechowanie wartości zwróconego ewentualnego błędu przez funkcję, który można pobrać przy pomocy metody *Marshall. GetLastWin32Error*.
- *CallingConvention* – wskazuje, w jaki sposób *CLR* ma odkładać elementy na stosie. Domyślną wartością jest *CallingConvention. WinApi*.
- *ExactSpelling* – wskazuje czy *P/Invoke* ma wyszukać w bibliotece *dll* funkcji o nazwie identycznej z nazwą tworzonego prototypu.

Samą deklarację funkcji rozpoczynamy obowiązkowo od słów kluczowych *static* i *extern*. Modyfikator *static* oznacza, że nie istniej żadna taka instancja klasy, która posiadałaby taką metodę, zaś *extern* informuje kompilator o tym, że metoda ta została napisana w innym języku niż *C#*. Następnie znajdują się typ zwracany przez funkcję oraz jej deklaracja. Na samym początku występuje również słowo kluczowe *public*, które informuje o tym, iż metoda ta jest metodą publiczną. Przed deklaracją samych funkcji trzeba jeszcze zdefiniować przestrzeń nazw *System.Runtime.Interop Services*, w której znajdują się klasy i metody niezbędne przy korzystaniu z *P/Invoke*. Tak zdefiniowane funkcje (które w kontekście klasy są metodami) można teraz używać tak samo jak metody zaimplementowane w kodzie zarządzanym. Na uwagę zasługuje jeszcze fakt, iż w deklaracji metody należy wpisać zarządzane typy danych, a importowane metody zawarte w bibliotece *dll* korzystają z typów danych nie zarządzanych. Tłumaczeniem i odpowiednim szeregowaniem (*marshaling*) danych zajmuje się *P/Invoke*. *P/Invoke* podczas odbierania i przekazywania wartości przy wywołaniu nie zarządzanego kodu odpowiada za odpowiednie tłumaczenie wartości na podstawie wskazówek programisty jak i własnych wbudowanych mechanizmów. Wiążę się z tym zjawiskiem dość sporo problemów. Jak wiadomo w kodzie zarządzanym typy referencyjne przechowywane są na zarządzanej stercie, która jest nadzorowana przez *Garbage Collector* i w ramach potrzeb nie używane referencje są usuwane ze sterty. W kodzie nie zarządzanym pamięć nie jest zwalniania. Dodatkowo typy referencyjne są przenoszone w pamięci fizycznej podczas działania programu, a typy zaimplementowane w kodzie nie zarządzanym nie. Po przesłaniu niektórych obiektów do kodu niezarządzanego, mogłaby już nie istnieć żadna referencja odnosząca się do obiektu. Wówczas takie obiekty mogłyby być usunięte bądź przeniesione przez *Garbage Collector*, w czasie gdy są przetwarzane przez kod natywny. Aby tak się nie stało obiekty takie są „przypinane" (ang. *pinned*) przez *P/Invoke* i w skutek czego *Garbage Collectora* ich nie usunie. Ponadto, w środowisku zarządzanym sposób organizacji w pamięci obiektów może ulec zmianie, natomiast w środowisku nie zarządzanym pozostaje on stały.

Ważną rzeczą przy przekazywaniu parametrów do nie zarządzanych funkcji jest ich typ. Typy używane w języku C mają swoje odpowiedniki w *CTS* (*Common Type System*), których przykładowe zestawienie pokazane jest w tabeli 1. *P/Invoke* posiada domyślny system tłumaczenia parametrów wejściowych i wyjściowych funkcji, jednak dzięki odpowiednim atrybutom możemy podpowiedzieć *P/Invoke* jak ma tłumaczyć parametry.

Tab. 1. Przykładowe odpowiedniki typów *C* w *CTS*
Tab. 1. Examples of *C* types and *CTS* equivalents

Typy niezarządzane z *wtypes.h*	Niezarządzany typ jęzka *C*	Nazwa klasy zarządzanej	Opis
HANDLE	void*	System.IntPtr	32 bits
BYTE	unsigned char	System.Byte	8 bits
SHORT	short	System.Int16	16 bits
WORD	unsigned short	System.UInt16	16 bits
INT	int	System.Int32	32 bits
UINT	unsigned int	System.UInt32	32 bits
LONG	long	System.Int32	32 bits
BOOL	long	System.Int32	32 bits

Dodatkowo w deklaracji funkcji przed deklaracją parametrów funkcji można użyć atrybutu kierunkowego *[Out]*, *[In]*, *[In, Out]*, nadpisując domyślne zachowanie *P/Invoke.* Atrybut kierunkowy informuje, czy ma być szeregowany parametr funkcji do niej wchodzący oraz czy zmiany dokonane w danym parametrze podczas wykonywania funkcji mają być widoczne. Domyślnym atrybutem kierunkowym dla parametrów wejściowych jest *[In]*. Jedynym wyjątkiem jest *StringBuilder,* dla którego domyślnym atrybutem kierunkowym jest *[In, Out]*. Dla szybkiego działania aplikacji bardzo ważną rolę odgrywa prawidłowe deklarowanie kierunkowości przesyłu danych. Jeżeli przesyłany typ danych pomiędzy kodem zarządzanym i niezarządzanym ma taką samą reprezentację w pamięci, wówczas nie ma potrzeby konwersji pomiędzy nimi. Taki obiekt zostaje „przypięty" w pamięci, a do funkcji zostaje przekazany adres pod którym znajduje się dany obiekt. Do takich typów zaliczają się: typy całkowite, *byte*, *sbyte*, *IntPtr*, tablice jednowymiarowe złożone z wyżej wymienionych typów oraz struktury, których pola są typami wymienionymi wcześniej. Jeśli jednak przekazywane są napisy (*string*) lub tablice (*array*) inne niż wyżej wymienione, wówczas przed wywołaniem niezarządzanej funkcji zostaje stworzona kopia obiektu i do funkcji zostaje przekazany jej adres. Jeśli dodatkowo zdefiniowany parametr zadeklarowany jest z atrybutem *[Out]* lub *[In, Out]*, wtedy po wykonaniu funkcji jego nowa wartość zostanie skopiowana do zarządzanego obiektu. Czynności te mają duży wpływ na ilość instrukcji wykonywanych przez procesor. Przykładowo, samo wywołanie niezarządzanej metody bez szeregowania danych to około 10-30 operacji procesora x86 [1]. Tak dość duża ilość operacji, jakie musi wykonać procesor wynika z tego, że środowisko uruchomieniowe musi dokonywać konwersji przy każdym kopiowaniu.

5. Podsumowanie

Binarne diagramy decyzyjne z pakietu *CUDD* mogą być wykorzystywane na platformie *.NET*. Jest to możliwie, z jednej strony, dzięki wykorzystaniu bibliotek dynamicznych *DLL* oraz, z drugiej strony, dzięki usłudze *Platform Invocation Service*, dostępnej w *Microsoft Visual Studio*. W pracy [4] pokazano przykładową implementację algorytmu dekompozycji funkcji boolowskiej, a uzyskane wyniki syntezy potwierdzają użyteczność opracowanej techniki.

6. Literatura

[1] Jagocki A.:P/Invoke i iphlpapi.dll - czyli jak zarządzać nie zarządzanym kodem, http://codeguru.pl/article-232.aspx
[2] Łabiak G., Adamski M.: Hierarchiczny diagram stanów i jego odwzorowanie w strukturze logicznej FPGA, Reprogramowalne Układy Cyfrowe - RUC 2004, Mat VII Krajowej Konferencji. Naukowej. Szczecin, Polska, 2004 (www.uz.zgora.pl/~glabiak)
[3] Łabiak G.: Wykorzystanie hierarchicznego modelu współbieżnego automatu w projektowaniu sterowników cyfrowych, Oficyna Wydaw. Uniwersytetu Zielonogórskiego (oraz http://zbc.uz.zgora.pl), 2005
[4] Makowski T.: Projekt i realizacja obiektu dokonującego minimalizacji funkcji logicznych metodą dekompozycji, Praca magisterska, WEiT Uniwersytet Zielongórski, Zielona Góra 2006
[5] de Mitcheli G.: Synteza i optymalizacja układów cyfrowych, WNT, Warszawa 1998
[6] Minato S.-I.:Binary Decision Diagrams and Applications for VLSI CAD, Kluwer Academic Publishers, Boston, 1996
[7] Petzold Ch.: Programowanie Windows. Kompletny podręcznik Win32 API do Windows 95/98/NT, RM, wyd. V, Warszawa 1999
[8] Somenzi F.: CUDD: CU Decision Diagram Package, http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html
[9] http://www.microsoft.com/poland/developer/net/podstawy/wprowadzenie.mspx

Artykuł recenzowany