Grzegorz Łabiak
UNIWERSYTET ZIELONOGÓRSKI, INSTYTUT INFORMATYKI I ELEKTRONIKI

Wykrywanie cykli zależności funkcyjnych sygnałów w językach HDL

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 referacie przedstawiono rozwiązanie problemu wyszukiwania cykli zależności funkcyjnych sygnałów w językach HDL. Cyklem zależności funkcyjnej jest taka sytuacja, w której sygnał, w sposób pośredni lub bezpośredni jest uzależniony od siebie samego na zasadzie sprzężenia zwrotnego. W klasycznych językach HDL ten rodzaj zależności jest wykorzystywany do projektowania m.in. asynchronicznych układów sekwencyjnych, podczas gdy w autorskim systemie HiCoS zjawisko tego rodzaju uznane jest za wadliwe.

Abstract

In the paper a solution to the problem of signal functional dependency cycle detection in HDL languages is presented. The signal functional dependency cycle is a situation where signal directly or indirectly depends on itself in feedback manner. In classic HDL languages this dependency type is commonly used in asynchronous sequential automata design, whereas in author's system, called HiCoS, is forbidden.

Słowa kluczowe: diagramy statecharts, HDL, sygnały, digrafy, cykle.

Keywords: statchart diagrams, HDL, signals, digraphs, cycles.

1. Wstęp

W językach HDL sygnały są podstawowymi środkami opisu sprzętu i stanowią fizyczny nośnik wartości funkcji logicznej. Funkcje logiczne z kolei mogą być definiowane za pomocą wyrażeń, w których operatory logiczne dokonują działań na sygnałach. W efekcie w modelowanym układzie występuje sieć wzajemnie powiązanych sygnałów, które mogą tworzyć pętle sprzężeń zwrotnych.

Najczęściej pojawienie się niezamierzonych sprzężeń zwrotnych jest traktowane jako zjawisko niepożądane, gdyż takie układy są trudne w analizie, cechują się możliwością wystąpienie takich niekorzystnych zjawisk jak wyścigi oraz mogą się wzbudzać. Ale sprzężenia zwrotne mogą też być wykorzystane z całym ich dobrodziejstwem, jakim jest szybkie przełączanie układzie. Ma to miejsce na przykład w realizacji asynchronicznych układów sekwencyjnych, lecz w tym przypadku nie należy mówić o nieplanowym wystąpieniu sprzężeń, gdyż projektowanie asynchronicznych układów sekwencyjnych przebiega właśnie według pewnego paradygmatu projektowego, gdzie sprzężenie zwrotne jest jego centralnym elementem.

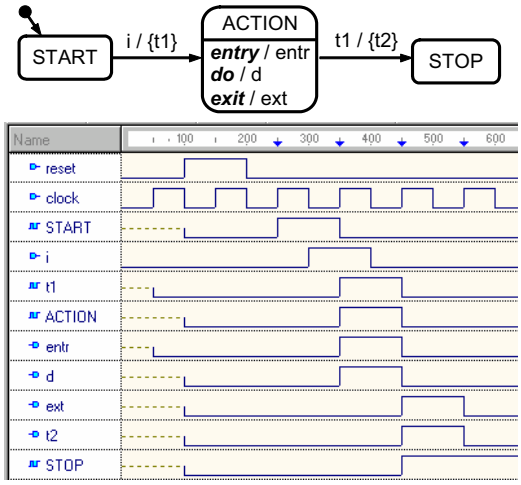
Popularne języki opisu sprzętu takie jak VHDL czy Verilog pozwalają, m.in. ze względu na możliwość projektowania układów asynchronicznych, na całkowitą dowolność w określeniu funkcji logicznych sygnałów, tak jak to jest możliwe w fizycznym układzie elektronicznym.

W przypadku specjalizowanego języka SSF (ang. *Statechart Specification Format*) w systemie HiCoS [4, 6, 9], przeznaczonego do tekstowego opisu diagramów statechart, wystąpienie zamkniętych zależności sygnałowych jest z założenia zabronione. Założenie takie bierze się z faktu, że aby opisać diagram statechart nie ma konieczności posługiwać się konstrukcjami ze sprzężeniem zwrotnym, zatem, zrezygnowano z możliwości tworzenia takich przypisań, których wynikiem są cykliczne zależności funkcyjne i które ze swej natury są trudne w analizie. Stąd płynie wniosek dla projektanta kompilatora języka SSF, że aby odrzucić specyfikację zawierającą wadliwe cykle zależności funkcyjnych sygnałów, najpierw w kompilowanym kodzie źródłowym należy je wykryć i następnie zasygnalizować to projektantowi informacją o błędzie.

2. Diagramy statechart i system HiCoS

Diagramy statecharts [3] zostały opracowane jako wizualny formalizm modelowania złożonego zachowania systemów reaktywnych. Szczególnie dobrze nadają się do modelowania zachowania cyfrowych układów sterownia binarnego [4, 6], gdzie układ sterujący odpowiada na dyskretne pobudzenia zgodnie z zaprogramowanym algorytmem.

System HiCoS jest przykładem środowiska projektowego, które dokonuje transformacji zachowania sterownika dyskretnego wyspecyfikowanej diagramami statechart na język VHDL. Wejściem do systemu jest tekstowa reprezentacja diagramów zapisana w formacie SSF.



Rys. 1. Podstawowy cykl pracy układu w systemie HiCoS
Fig. 1. Basic working cycle of HiCoS controller

Działania cyfrowego sterownika specyfikowanego diagramami [6] stymulowane jest zdarzeniami przychodzącymi z otocznia. Przyjmuje się, że każde zdarzenie (przychodzące, wychodzące oraz wewnętrzne) w układzie fizycznie reprezentowane jest sygnałem i jest związane z dyskretną dziedziną czasu. Sterownik reaguje na zbiór dostępnych zdarzeń w systemie poprzez odpalenie zbioru gotowych do realizacji tranzycji – zwanych mikrokrokiem. Ze względu na możliwość wystąpienia oddziaływań zwrotnych, realizacja mikrokroku może pociągnąć za sobą wygenerowanie zdarzeń, które następnie mogą być przyczyną wykonania kolejnego mikrokroku. Taka sekwencja mikrokroków realizowana do mementu, gdy w układzie nie będzie już gotowych do realizacji tranzycji, zwana jest krokiem. Przyjmuje się, że w trakcie realizacji kroku, do układu nie przychodzą zdarzenia z zewnątrz. Zdarzenia wygenerowane w czasie mikrokroku, nie oddziałują na bieżąco realizowane tranzycje, lecz mogą wpływać na działanie układu, dopiero przy najbliższym takcie sygnału zegarowego, czyli przy następnym mikrokroku.

Rysunek 1 przedstawia krok wraz z przebiegiem czasowym, który składa się z dwu prostych mikrokroków. Po wykonaniu całego kroku, układ znajduje się w stanie STOP. Podsumowując, o dynamice realizacji sprzętowej można powiedzieć, że:

- system jest synchroniczny
- system reaguje na zbiór dostępnych zdarzeń przez realizacje gotowych tranzycji,

- generowane zdarzenia wewnętrzne, są dostępne dla systemu podczas następnego taktu sygnału zegarowego.

Przykład z rysunku 1 oraz przebieg czasowy ilustrują omawiane pojęcia i przyjęte założenia. Gdy tranzycja t_1 jest odpalona (T=350), zdarzenie t_1 jest rozgłaszane i staje się dostępne dla systemu podczas następnej chwili dyskretnego czasu (*T=450*). Aktywność jest przekazywana ze stanu START do stanu ACTION. W tym momencie tranzycja t_2 staje się tranzycją gotową do realizacji. Jej stan źródłowy jest aktywny, predykat nałożony na tranzycję (zdarzenie t_1) jest spełniony, tak więc w momencie T=450 system przekazuje aktywność to stanu STOP oraz rozgłasza zdarzenie t_2, które już nie oddziałuję na żadną tranzycję w systemie. Krok jest zakończony.

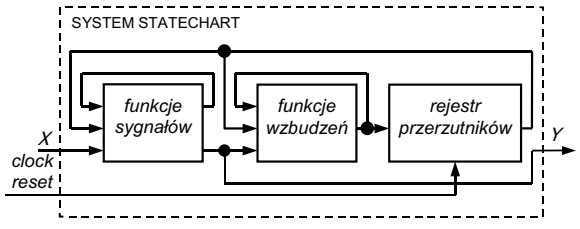
Rysunek 2 przedstawia przykładowy kod w języku SSF, który jest równoważny diagramowi z rysunku 1. Ten prosty diagram składa się z jednego automatu sekwencyjnego, nazwane SC, trzech stanów: START, ACTION, STOP; jednego zdarzania wejściowego i; trzech zdarzeń dostępnych dla świata zewnętrznego: t2, entr, d, ext; jednego zdarzeni o zasięgu lokalnym t1. Dwóm przejściom ze stanów START i ACTION odpowiadają dwie instrukcje określające funkcję przejścia tf wraz z predykatami, odpowiednio i oraz t1. Dla inżyniera projektanta postać tekstowa jest zdecydowanie mniej poglądowa niż diagramy, lecz dla celów badawczych zdecydowano się na mniej atrakcyjną pośrednią formę wprowadzania danych, ale za to tańszą w realizacji i łatwiejszą do szybkiej modyfikacji. W opinii autora, realizacja prostego programu kompilatora jest znacznie mniej pracochłonna niż realizacja złożonego edytora graficznego.

```
controller Demo;
port(i: in; t2: out; entr: out; d: out;
ext: out);

signal t1;
SC = ({START,ACTION,STOP}, START, tf, ef){
    tf(START, i) => (ACTION, {t1});
    ef(ACTION) = entry / {entr};
    ef(ACTION) = do / {d};
    ef(ACTION) = exit / {ext};
    tf(ACTION, t1) => (STOP, {t2});
};
```

Rys. 2. Tekstowa postać w SSF diagramu z rysunku 1
Fig. 2. Textual form in SSF of the diagram from figure 1

Plik wyjściowy z systemu HiCoS, zapisany w języku VHDL, przedstawia sobą system powiązanych przerzutników, funkcji wzbudzeń i funkcji sygnałów (rys. 3). Jak widać to na rysunku w przypadku funkcji sygnałów, funkcje te są od siebie uzależnione, co w pewnych sytuacjach może prowadzić do sprzężeń zwrotnych. Tego rodzaju sytuacje stanowią przypadek niekorzystny, który powinien zostać przez system HiCoS wykryty, a specyfikacja zawierająca taki przypadek uznana za wadliwą.

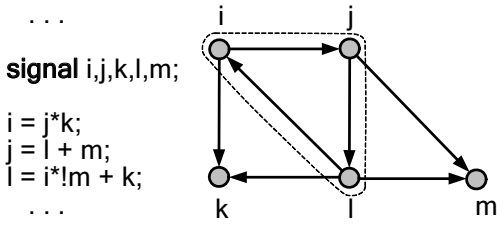


Rys. 3. Model implementacyjny sterownika w systemie HiCoS
Fig. 3. Implemental model of HiCoS controller

3. Cykle zależności funkcyjnych sygnałów

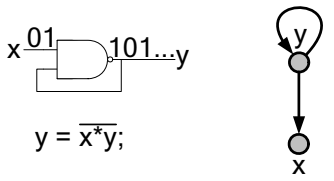
Diagramy statechart zasadniczo nie oferują możliwości przypisywania wyrażeń algebraicznych sygnałom, ale ze względu na wygodę w projektowaniu cyfrowych sterowników binarnych, w języku SSF taka możliwość została wprowadzona. Z jednej strony mechanizm ten ułatwia tworzenie złożonych wyrażeń algebraicznych rozwiązujących konflikty między tranzycjami [5], lecz z drugiej strony może prowadzić do wystąpienia pewnych sytuacji niepożądanych. Poniżej zamieszczono trzy przykładowe takie przypadki.

Rysunek 4 przedstawia fragment kodu w języku SSF stanowiący nietrywialną sekwencję przypisań wyrażeń algebraicznych do sygnałów. W wyniku przypisań zostały ustalone zależności między sygnałami, które są zobrazowane przy pomocy grafu zorientowanego umieszczonego obok. Z tego rysunku widać, że sygnał i jest uzależniony od sygnałów j oraz k, sygnał j zależy od sygnałów l i m, a sygnał l jest zależny od sygnałów k, i oraz m. Istotnym wnioskiem z tego grafu jest fakt, że sygnał i zależy od sygnału j, j zależy od l, a l zależy znowu od i – co w efekcie tworzy cykl. W klasycznych języku HDL jak VHDL [7] czy Verilog taki rodzaj przypisań nie prowadzi do błędów kompilacji. O tym jakie będzie zadziałanie takiego układu zdecyduje symulator bądź docelowa technologia implementacyjna i nie jest wykluczone, że uzyskany efekt będzie odbiegał od zamierzonego.



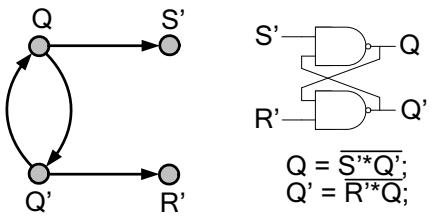
Rys. 4. Przykład cyklicznej zależności niejawnej
Fig. 4. An example of cyclic dependency

Rysunek 5 zawiera przykład najprostszej jawnej zależności cyklicznej. Wyjście bramki nand podłączone jest do jednego ze swoich wejść. W przypadku gdy do wejścia x podłączona jest wartość 0 na wyjściu y będzie 1. Gdy na wejściu pojawi się sygnał 1, bramka wejdzie w stan niestabilny, na wyjściu będą miały miejsce zmiany z 0 na 1 i 1 na 0 o okresie równym dwóm czasom propagacji bramki. Kompilacja tego prostego układu w języku VHDL przebiegła bez problemu, lecz w przypadku próby symulacji został zgłoszony błąd: „# KERNEL: Delta count overflow – stopped (…)". Sprawcą błędu symulacji jest oczywiście cykliczna zależność sygnału y od siebie samego, co na grafie jest przedstawione pętlą.



Rys. 5. Przykład cyklicznej zależności jawnej
Fig. 5. An example of explicit cyclic dependancy

Na rysunku 6 zamieszczono cykliczną zależność sygnałów, która jest jak najbardziej zamierzona. Jest to bowiem klasyczny przerzutnik RS i zarazem egzemplifikacja najprostszego sensownego asynchronicznego układu sekwencyjnego.



Rys. 6. Przykład cyklicznej zależności zamierzonej
Fig. 6. An example of intentional cyclic dependency

W języku SSF w systemie HiCoS, w odróżnieniu od klasycznych języków HDL, zdecydowano się zabronić występowania cyklicznych zależności sygnałów. Po pierwsze w systemie HiCoS nie ma potrzeby definiowanych własnych asynchronicznych układów sekwencyjnych, a po drugie przyjęto za bardzo niekorzystną sytuację, gdy na skutek wystąpienia zależności cyklicznych system HiCoS generuje kod w języku VHDL poprawny składniowo, lecz interpretacja jego wyników symulacji i działania sprawiają trudności.

4. Wykrywanie zależności cyklicznych

Problem znajdowania cyklicznych zależności funkcyjnych sygnałów sprowadza się do problemu znajdowania cykli w grafie zorientowanym (digraf) [8].

Podstawowym elementem prawidłowo działającego algorytmu jest struktura danych na której algorytm operuje. W przypadku algorytmów grafowych typowymi reprezentacjami grafów w pamięci komputera są macierz sąsiedztw i lista sąsiedztw [1, 2]. W rozpatrywanym problemie zdecydowano się na zastosowanie listy sąsiedztw, gdyż ta reprezentacja ekonomiczniej wykorzystuje pamięć komputera, mianowicie zajętość pamięci jest liniowo zależna od sumy liczby wierzchołków w grafie i liczby krawędzi [1]. Istotą listowej reprezentacji grafu jest to, że z każdym wierzchołkiem grafu G związana jest lista wierzchołków (w algorytmie przedstawionym na rys. 7 zwana jest `ListąIncydencji`), do których z danego wierzchołka poprowadzona jest krawędź. Oprócz listy sąsiedztw każdemu wierzchołkowi przyporządkowano dwie zmienne pomocnicze `Visited` i `InPath`. Zmienna `Visited` przyjmuje wartość true gdy dany wierzchołek został odwiedzony, a zmienna `InPath` przyjmuje wartość true, gdy wierzchołek przynależy do ścieżki [8], która jest testowana na zawartość cyklu.

```
 1  try {
 2     for każdy wierzchołek u in G
 3        P.clear()
 4        P.push_back(u)
 6        _SearchCycle()
 7  }
 8  catch(v) {
 9     Sciezka P zawiera cykl
10  }
11  _SearchCycle() {
12     u = P.back()
13     if not u.Visited
14        u.Visited ← true
15        u.InPath ← true
16     else
17        return
18     for v in u.ListaIncydencji
19        if v.InPath throw v
20        if not v.Visited
21           P.push_back(v)
22           _SearchCycle()
23           P.pop_back()
24     u.InPath ← false
25  }
```

Rys. 7. Algorytm wyszukiwania cykli w digrafie
Fig. 7. An algorithm of cycle detection in digraf

Działanie algorytmu, który jest przedstawiony na rysunku 7, jest pewną modyfikacją klasycznego algorytmu przeszukiwania grafu na zasadzie przeszukiwania w głąb [2] i jest przerywane w momencie napotkania pierwszego cyklu. Ponieważ rozpatrywany graf jest grafem zorientowanym, zatem należy dla każdego wierzchołka grafu dokonać sprawdzenia, czy przypadkiem z tego wierzchołka nie jest osiągalny cykl. To sprawdzanie jest przeprowadzane na zasadzie przeszukiwania w głąb. Warto również zauważyć, że dotarcie w przeszukiwaniu do wierzchołka już odwiedzonego nie wymaga dalszego przeszukiwania związanego z odwiedzonym wierzchołkiem, gdyż gdyby odwiedzony wierzchołek należał do cyklu, to zostałoby to wcześniej wykryte i spowodowałoby przerwanie działania algorytmu. Powyższa konstatacja prowadzi do wniosku, że w najgorszym przypadku w przeszukiwanym grafie każdy wierzchołek zostanie odwiedzony co najwyżej dwa razy, zatem złożoność obliczeniowa algorytmu jest liniowa i wynosi $O(n+m)$, gdzie n oznacza liczbę wierzchołków w grafie a m liczbę krawędzi.

Szczegółowe działanie algorytmu jest następujące. W liniach od 2 do 6 w bloku dozorowanym `try` dla każdego wierzchołka grafu jest wywoływana funkcja `_SearchCycle()` sprawdzająca czy z danego wierzchołka jest osiągalny cykl. Każdy wierzchołek jest umieszczany na testowej ścieżce `P`. W razie napotkania cyklu funkcja zgłasza wyjątek przekazując wierzchołek, który na ścieżce `P` poprowadzonej od rozpatrywanego wierzchołka pojawił się po raz drugi, co oznacza wystąpienie cyklu. Obsługa wyjątku umiejscowiona jest w lini 9. Funkcja `_SearchCycle()` zaczyna od pobrania ze ścieżki testowej P ostatnio dodanego wierzchołka (linie 4 i 21) i sprawdzenia czy nie ma do czynienia z już odwiedzonym wierzchołkiem (linie 13-17) i zapisania, że wierzchołek został odwiedzony i należy do ścieżki testowej (linie 14, 15). Następnie jest sprawdzane dla każdego wierzchołka sąsiedniego jego przynależność do ścieżki testowej `P`, co sygnalizowane jest zgłoszeniem wyjątku (linia 19) i oznacza znalezienie cyklu i przerywa działania algorytmu. Jeżeli wierzchołek sąsiedni nie został odwiedzony i nie należy do ścieżki testowej, wówczas jest dodawany do ścieżki testowej `P` (linia 21) i dla tego wierzchołka funkcją `_SearchCycle()` (linia 22) sprawdzana jest osiągalność cyklu. Jeżeli nie znaleziono cyklu osiągalnego z sąsiada, wówczas wierzchołek ten jest usuwany ze ścieżki testowej `P` (linia 23). Jeżeli dla rozpatrywanego wierzchołka, dla żadnego z jego sąsiadów nie znaleziono cyklu wówczas wierzchołek jest usuwany ze ścieżki testowej `P`, co polega na nadaniu zmiennej pomocniczej `InPath` wartości false (linia 24). Warto odnotować, że algorytm, który w swej istocie polega na jawnej rekurencji, został zaprojektowany w taki sposób, że istotna dla algorytmu ścieżka testowa `P` jest niezależna od stosu programu.

5. Podsumowanie

Cykliczne zależności funkcyjnye sygnałów w językach HDL z jednej strony dostarczają silnego środka ekspresji w modelowaniu sprzętu, lecz jednocześnie mogą sprawiać pewne problemy projektowe. W autorskim systemie HiCoS ten środek specyfikacja został uznany za wadliwy, z czym wiąże się konieczność wykrywania w specyfikacji wejściowej tego rodzaju konstrukcji. Wykrywanie zależności cyklicznych polega na wykrywaniu cykli w grafie zorientowanym, który stanowi formalne zobrazowanie zależności między sygnałami. Złożoność obliczeniowa algorytmu liniowo zależy od sumy liczby sygnałów i liczby krawędzi.

6. Literatura

[1] Banachowski L., Diks K., Rytter W.: Algorytmy i struktury danych, WNT, Warszawa 2001
[2] Cormen Thomas H., Leiserson Charles E., Rivest Ronald L.:Wprowadzenie do algorytmów, WNT, Warszawa 1998
[3] Harel D. (1987): Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming 8, 231–274.
[4] Łabiak G.: From UML Statecharts to FPGA – the HiCoS Approach, FDL ss. 354-363, Frankfurt 2003
[5] Łabiak G.: Transition Conflicts Detection in Binary Modular Statechart Diagrams, PDS, ss. 192-197, Kraków 2004
[6] Łabiak G.: Wykorzystanie hierachicznego modelu współbieżnego automatu w projektowaniu układów cyfrowych, Prace Naukowe z Automatyki i Informatyki, Oficyna Wydawnicza Uniwersytetu Zielonogórskiego 2005
[7] Perry Douglas L.: VHDL, McGraw-Hill, Inc, New York 1991
[8] Wilson Robin J.: Wprowadzenie do teorii grafów, PWN, Warszawa 2007
[9] http://www.uz.zgora.pl/~glabiak strona domowa systemu HiCoS

Title: Detection of signal functional dependency cycle in HDL.

Artykuł recenzowany