Piotr DZIURZAŃSKI, Mariusz KARPICKI
POLITECHNIKA SZCZECIŃSKA, WYDZIAŁ INFORMATYKI

Formalna weryfikacja automatycznego zrównoleglenia procesów

Dr inż. Piotr DZIURZAŃSKI

Ukończył studia na Wydziale Informatyki Politechniki Szczecińskiej w 2000 r. Na tym samym wydziale obronił pracę doktorską w 2003 r. Obecnie pracuje na stanowisku adiunkta w Instytucie Architektury Komputerów i Telekomunikacji WI PS. Jest członkiem organizacji IEEE i ACM. Jego zainteresowania to kosynteza sprzętowo-programowa, synteza wysokiego poziomu, synteza logiczna i formalna weryfikacja

e-mail: pdziurzanski@wi.ps.pl

Inż. Mariusz KARPICKI

Ukończył studia inżynierskie na Wydziale Informatyki Politechniki Szczecińskiej w 2004 r. Obecnie jest studentem drugiego stopnia na kierunku Informatyka Wydziału Informatyki Politechniki Szczecińskiej. Jego zainteresowania są związane z formalną weryfikacją systemów sprzętowo-programowych.

e-mail: mkarpicki@wi.ps.pl

Streszczenie

W artykule przedstawiono technikę formalnej weryfikacji systemów sprzętowo-programowych opisanych za pomocą języka opisu systemów SystemC. Formalnej weryfikacji dokonuje się z wykorzystaniem logiki temporalnej CTL i asercji. Przedstawiono formuły CTL dla systemu z jedną sekcją równoległą. Badania eksperymentalne wykazały liniowy wzrost liczby formuł i liniowy przyrost czasu działania programu automatycznie wstawiającego asercję, przez co prezentowane podejście nadaje się do zastosowań przemysłowych.

Słowa kluczowe: formalna weryfikacja, logika temporalna CTL, asercje, równoleglizowanie, SystemC

Formal verification of automatically parallelised processes

Abstract

In this paper, we present a formal verification technique of software/hardware systems given in the SystemC system description language. The verification is performed using temporal logic CTL and assertions. We enumerate the CTL formulas generated from a system with a single parallel section. Experimental results present a linear growth of a number of formulas and linear growth of the execution time of the developed tool that automatically inserts CTL assertions. Consequently, the proposed approach is suitable for industrial applications.

Keywords: formal verification, temporal logic CTL, parallelisation, assertion, SystemC

1. Wrowadzenie

Problem poprawności systemów sprzętowo-programowych jest analizowany przez naukowców i inżynierów od lat. Niewykryte błędy w systemie skutkują często wysokim kosztem i odpowiedzialnością prawną, jaką firmy ponoszą za wypuszczenie na rynek wadliwego projektu. Problem ten dotyczy zwłaszcza systemów wbudowanych, których poprawne działanie w określonych ramach czasowych często wpływa na życie ludzkie [1].

Obecnie używane narzędzia CAD lub EDA umożliwiają generację systemów, które nie mogą być przetestowane w rozsądnym czasie. W [2] podano, że 70% czasu projektowania spędza się na weryfikacji. Tradycyjne podejście nazwane funkcjonalną weryfikacją, traktujące system jako czarną skrzynkę i umożliwiające testowanie poprzez podawanie określonych wartości wejściowych i obserwowanie symulowanego wyjścia, jest niepraktyczne w przypadku nawet stosunkowo prostych projektów ze względu na wymaganą wykładniczą liczbę wektorów testowych. Niedoskonałość funkcjonalnej weryfikacji powoduje, że w 70% przypadków maski do układów cyfrowych VLSI trzeba wyprodukować przynajmniej dwukrotnie (koszt wykonania maski to przynajmniej 1 mln USD) [2].

Dodatkowym problemem przy weryfikacji funkcjonalnej jest stopień, w jakim testy pokryły funkcjonalność systemu (ang. *coverage*). Znane są takie metryki, jak liczba linii czy liczba stanów w odpowiadającym automacie skończonym. Korelacja między tymi miarami z pokryciem rzeczywistej funkcjonalności jest wciąż przedmiotem badań [3].

We wcześniejszych badaniach [4] jeden z autorów współuczestniczył w tworzeniu systemu do syntezy opisu systemu w języku SystemC z algorytmu zadanego w języku programowania ANSI C. Podejście to transformowało szeregowo wykonywany kod programu w specyfikację sprzętu przeznaczoną do wykonania równoległego. Przy transformowaniu kodu szeregowego do jego równoległego odpowiednika mogą jednakże wystąpić naruszenia zależności danych. Stąd konieczne było stworzenie systemu dokonującego weryfikacji poprawności opracowanego narzędzia.

Istnieją różne sposoby dokonywania weryfikacji poprawności oprogramowania. Nie wspominając o ręcznym testowaniu projektu, obecnie najczęściej praktykowane jest przeprowadzanie symulacji, za pomocą specjalnie skonstruowanych symulatorów, wykorzystujących automaty skończone lub sieci Petri [5]. Symulatory sprawdzają zachowanie się modelu testowanego projektu dla losowych bądź ustalonych danych wejściowych. Przy stosowaniu tej metody pojawiają się następujące problemy. Pierwszy to problem stworzenia odpowiedniej kombinacji danych wejściowych, które będą dostateczne do wykrycia jakichkolwiek nieprawidłowości w działaniu i drugi, jakim jest znajomość danych wyjściowych systemu, które niezbędne są do porównania z uzyskanymi poprzez symulację. Kolejnym sposobem weryfikowania poprawności jest metoda sprawdzająca w sposób formalny, poprzez dowodzenie, czy model zgodny jest ze specyfikacją systemu dla podawanych danych wejściowych. Metoda ta nazywa się formalną weryfikacją.

W artykule zostanie przedstawione zastosowanie logiki temporalnej CTL przy weryfikowaniu poprawności modelu programów współbieżnych (równoległych). Przedmiotem badań będzie kod programów skonstruowany przy użyciu języka przeznaczonego do opisu sprzętu – języka SystemC [6].

W niniejszym artykule jest wykorzystana technika weryfikacji przez model z wykorzystaniem asercji (ang. *assertion based verification*, ABV). Polega ona na (1) wyznaczeniu zbioru stanów modelu, (2) sprawdzeniu, czy każdy stan spełnia zadaną właściwość [7]. Problemy rozwiązywalne w tej technice muszą być opisywane za pomocą skończonej liczby stanów. Możliwe jest wówczas utworzenie automatu skończonego (ang. *finite state machine, FSM*) opisującego te stany i przejścia między nimi. Istnieje możliwość automatyzacji metody weryfikacji przez model i stąd jest ona przedkładana nad inne metody [8].

Asercje są wyrażeniami logicznymi, które mogą wyrażać dopuszczalne lub niedopuszczalne zachowanie systemu. Standardowa technika ABV polega na porównaniu asercji wyznaczonych na

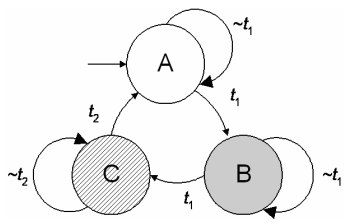
etapie projektowania z faktycznym działaniem systemu poprzez zapisanie tych asercji wewnątrz wykonywalnego modelu. W przypadku naruszenia jakiejkolwiek asercji, sytuacja ta jest natychmiast zgłaszana użytkownikowi [7]. Asercje są często wykorzystywane w formalnej weryfikacji, która wykorzystuje statyczną analizę kodu do sprawdzenia, czy dana asercja zachodzi dla wszystkich możliwych danych wejściowych, czy też istnieje kontrprzykład [1].

Chociaż logiki temporalne używa się najczęściej do określania i dowodzenia własności osadzonych systemów sprzętowych i programowych, prowadzi się także badania nad wykorzystaniem logik temporalnych w robotyce [9].

2. Logika drzew obliczeń

Logiki temporalne są rozszerzeniem rachunku predykatów do systemu logicznego umożliwiającego wnioskowanie z użyciem czasu [8]. W logikach temporalnych wartości zmiennych zmieniają się w czasie, dzięki czemu istnieje możliwość wyrażenia zależności logicznych między zmiennymi, które zachodzą w różnych chwilach czasu, przez co można formalnie wyrażać wpływ jednego zdarzenia na inne, zachodzące później. Jedną z najpopularniejszych i najczęściej używanych logik temporalnych jest logika drzew obliczeń (ang. *computational tree logic, CTL*) [7].

Przy weryfikacji formalnej z wykorzystaniem asercji pierwszym etapem jest stworzenie automatu skończonego opisującego dany problem (a ściślej jego modyfikacji - struktury Kripke [1], zawierającej funkcję etykietującą, która przypisuje stanom zbiór predykatów atomowych w nim prawdziwych). Przykład struktury Kripke *M* opisującego system świateł ulicznych został przedstawiony na rys. 1.



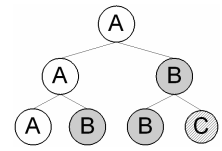
Rys. 1. Automat skończony opisujący światła uliczne
Fig. 1. Finite automate describing street lights

Na tym rysunku stała t_1 określa czas, w jakim świeci się światło zielone lub czerwona, a t_2 - żółte. Wielkie litery na stanach oznaczają formuły atomowe w nich prawdziwe, natomiast nazwy stanów symbolizowane są przez odpowiednie zakreskowanie wnętrza stanu. Stan biały oznacza świecenie się światła czerwonego, stan szary - zielonego, a stan wypełniony linią ukośną - żółtego.

Drzewo obliczeń (ang. *computation tree*) jest typem drzewa (czyli acyklicznego grafu skierowanego), który powstaje ze struktury Kripke opisującego dany system. Korzeń tego drzewa odpowiada stanowi początkowemu automatu (oznaczonego strzałką na rys. 1), a liczba jego poziomów jest nieskończona. Każdy wierzchołek drzewa obliczeń jest nazwany tak, jak odpowiadający mu stan w M.

Drzewo obliczeń jest budowane w następujący sposób: korzeń drzewa jest nazwany tak, jak stan startowy automatu, często oznaczany w literaturze jako q_0. Węzeł ten jest połączony z węzłami pierwszego poziomu, które odpowiadają stanom, z jakich można dojść ze stanu q_0 w jednym ruchu. Na trzecim poziomie drzewa występują stany, z których można dojść ze stanów poziomu drugiego itd.

Trzy pierwsze poziomy dla drzewa obliczeń zbudowanego z automatu z rys. 1 zostały przedstawione na rys. 2.



Rys. 2. Drzewo obliczeń zbudowane ze struktury Kripke opisującej światła uliczne (rys. 1)
Fig. 2. Computation tree built from the Kripke structure describing street lights (Fig. 1)

W procesie weryfikacji na drzewie obliczeń sprawdzane są asercje zadane w postaci formuł CTL. Formuły te są zbudowane z (1) predykatów atomowych (odpowiadającym zmiennym w modelu), (2) spójników logicznych (AND, OR, XOR, NOT) i (3) operatorów temporalnych.

Operatory temporalne składają się z dwóch części

1. kwalifikatora ścieżki:
- A - własność zachodzi dla wszystkich ścieżek wychodzących z bieżącego stanu,
- E - własność zachodzi przynajmniej dla jednej ścieżki wychodzącej z bieżącego stanu,

2. temporalnej modalności (ang. *temporal modality*):
- Fφ - φ zachodzi przynajmniej raz w przyszłości (Future)
- Gφ - φ zachodzi zawsze (Global),
- Xφ -φ zachodzi w następnym stanie (neXt),
- φUψ - φ zachodzi dopóki (Until) ψ nie zachodzi.

W ostatnim przypadku pierwsza własność zachodzi dla każdego stanu w ścieżce aż do stanu (włącznie albo wyłącznie), w którym druga własność zachodzi. W ścieżce musi być stan, dla którego druga własność zachodzi.

Weryfikacja przez model w logice CTL jest prostsza niż w innych modelach, takich jak LTL [7], gdyż umożliwia przeprowadzenie analizy bezpośrednio na podstawie jednego automatu skończonego, czy też skojarzonego z nim drzewa obliczeń [8].

3. Język opisu sprzętu SystemC

Języki opisu sprzętu (ang. *Hardware Description Language, HDL*) stały się jednym z najważniejszych sposobów opisu systemów sprzętowych i sprzętowo-programowych. Języki HDL wprowadzają konstrukcje językowe na wysokim poziomie abstrakcji umożliwiające projektantom szybkie opisywanie dużych systemów.

Projektowane systemy wykonują operacje równolegle na bardzo niskim poziomie abstrakcji, dlatego wiele języków umożliwia przedstawienie równoległości działania przez zastosowanie współbieżnych procesów. Języki HDL umożliwiają modelowanie układów w postaci modeli strukturalnych (sprzętowych) i behawioralnych (funkcjonalnych).

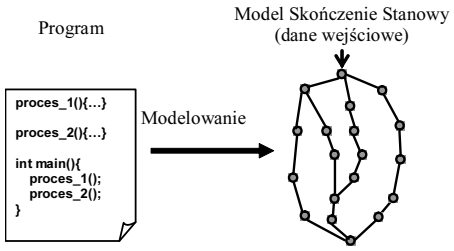
Język SystemC jest biblioteką stworzoną przez organizację Open SystemC Initiative (OSCI). W roku 2005 został uznany jako standard przez organizację IEEE. Język SystemC zaprojektowany został w celu modelowania systemów i układów cyfrowych w języku programowania C++. Tak jak inne języki opisu sprzętu doskonale nadaje się do konstruowania modeli współbieżnie wykonujących wiele procesów. Modele systemów cyfrowych wykonywane w języku SystemC są programami wykonywalnymi [6].

4. Proponowane podejście

Zazwyczaj statycznej analizy kodu w języku ANSI C lub SystemC nie dokonuje się bezpośrednio na opisie tekstowym, a na modelu zbudowanym na podstawie tego kodu. Dlatego pierwszym

etapem wykonywania procesu formalnej weryfikacji jest modelowanie, czyli przekształceniem systemu w formalny model.

Programy weryfikujące oparte na tej metodzie działają na modelach o skończonej liczbie stanów skonstruowanych z testowanych programów. Model reprezentowany jest przez graf (diagram przejść automatu skończonego). Składa się on ze zbioru wierzchołków przedstawiających stany systemu (określających np. wartości zmiennych) oraz krawędzi będących przejściami ze stanu do stanu (rys. 1). Ideę modelowania systemu przedstawiono na rys. 3.



Rys. 3. Idea procesu modelowania
Fig. 3. Idea of modeling process

Drugim etapem procesu jest weryfikacja. Podczas tego etapu sprawdzane jest, czy reprezentacja modelowa programu spełnia wymagania stawiane i dostarczane przez specyfikację. Algorytmicznie sprawdzane jest to, czy formuły są prawdziwe w modelu. Realizacja sprawdzenia dokonana może być bezpośrednio, realizując przejście po automacie dla danej formuły bądź poprzez konstruowanie automatu akceptującego wszystkie modele dla formuły i sprawdzenie, czy badany model zawiera się w zbiorze.

Systemy weryfikujące oparte na metodzie weryfikacji modelowej są w stanie wychwycić błędy logiczne i funkcjonalne. Mogą być to błędy dotyczące wielowątkowości i wielozadaniowości. Wykryte błędy mogą być różnego typu, włączając blokady, warunki wykonań, problemy pierwszeństwa, problemy zakleszczeń, związki między wykonywanymi procesami, naruszanie granic systemu, niekompatybilność ze specyfikacją, martwe miejsca w kodzie, problemy logiczne w tym brak relacji czasowych [8].

Przedstawiony przykład zastosowania weryfikacji przez model z zastosowaniem logiki CTL ilustrować będzie weryfikowanie poprawności kolejności wykonania procesów w programach języka SystemC. Badaniom poddany został zrównoleglony algorytm mnożenia macierzy. W rozważanym przykładzie przyjęto, iż zrównoleglony program do dyspozycji ma cztery procesy mogące wykonywać się równolegle a wynikowa macierz *mac3* o wymiarach 5x5 powstaje z przemnożenia macierzy *mac1* o wymiarach 5x4 oraz macierzy *mac2* o wymiarach 4x5. Algorytm programu przedstawiono na rys. 4 w postaci pseudokodu.

```
for(k=0;k<5;k++){
    for(i=0;i<5;i++){
        for(j=0;j<4;j++){
         mac3[k][i]=mac3[k][i]+mac1[k][j]*mac2[j][i];
        }
    }
}
```

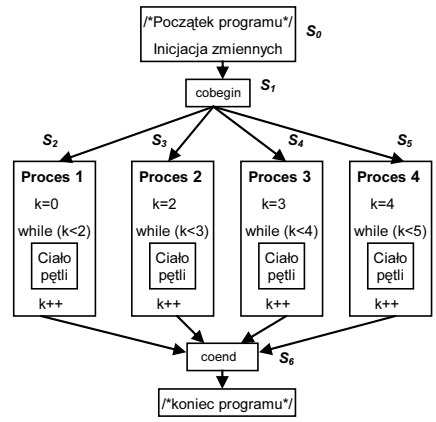
Rys. 4. Algorytm mnożenia macierzy
Fig. 4. Matrices multiplication algorithm

Podczas procesu zrównoleglenia programu, kolejne wykonania zewnętrznej pętli przydzielone zostały procesom. Podczas przeprowadzania badań znaczenie będą miały następujące stany:

- *Init* – stan inicjujący, blok zawierający deklaracje zmiennych,
- *Accept* – stan akceptujący, ostatni stan, jaki zaistnieje w programie,
- *Cobegin* – rozpoczęcie sekcji współbieżnej w programie,
- *Parallel* – blok kodu wykonywanego współbieżnie,
- *Coend* – koniec sekcji współbieżnej w programie.

Wykonanie zrównoleglonego programu wraz z nazwami stanów, w jakich się znajduje, przedstawia rys. 5.

Proces tworzenia formuł logicznych CTL, służących jako specyfikacji programu dotyczy zmiennych: *threads_num* będącej liczbą całkowitą określającą liczbę procesów oraz zbioru zmiennych stanu $\{S_0, ..., S_n\}$.



Rys. 5. Zrównoleglony pogram z przydzielonymi nazwami stanów przebiegu
Fig. 5. Parallelised program with states assigned with names

Poprawność programu charakteryzują dwie własności *żywotności* (ang. *liveness*) oraz *bezpieczeństwa* (ang. *safety*) i własności poprawności dla poszczególnych stanów.

Własność żywotności dla programu przedstawiona jest w sposób:

$$AG (S_0 \rightarrow AF\ S_6),$$

a własność bezpieczeństwa:

$$AG (threads\_num <= 4).$$

Dla poszczególnych stanów wygenerowano formuły. Dla stanu inicjującego S_0:

$$S_0 \vDash AX (S_1),$$
$$S_0 \vDash AX (threads\_num = 1).$$

Dla stanu *Cobegin*:

$$S_1 \vDash EX\ (S_2) \wedge EX \wedge (S_3) \wedge EX\ (S_4) \wedge EX\ (S_5),$$
$$S_1 \vDash AX (threads\_num = 4),$$
$$S_1 \vDash EX (S_2\ AU\ S_6),$$
$$S_1 \vDash EX (S_3\ AU\ S_6),$$
$$S_1 \vDash EX (S_4\ AU\ S_6),$$
$$S_1 \vDash EX (S_5\ AU\ S_6),$$

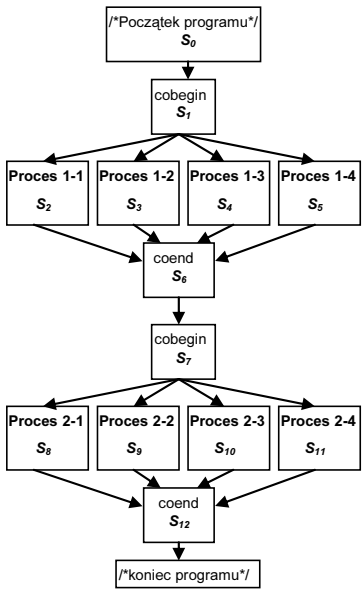
gdzie symbol $\vDash$ oznacza, że formuła zapisana po jego prawej stronie jest prawdziwa w stanie zapisanym po lewej stronie.

Dla stanów *Parallel* reprezentujących bloki wykonywane współbieżnie (formuły są analogiczne, przykład przedstawiono dla stanu S_2):

$$S_2 \vDash AX (S_6),$$
$$S_2 \vDash AX (threads\_num = 1).$$

Obserwacje nad liczbą możliwych formuł przeprowadzono również dla programu z dwoma sekcjami równoległymi (rys. 6). Spowodowało to, iż poza kolejną sekcją *Cobegin* oraz blokami

równoległymi *Parallel* powstały formuły dla bloku *Coend* pierwszej sekcji równoległej, który nie był już stanem akceptującym.



Rys. 6. Program z dwoma sekcjami równoległymi
Fig. 6. Program with two parallel sections

Jako, że następnym stanem występującym po stanie *Coend* jest jeden stan *Cobegin* formuły dla tego stanu wyglądają następująco:

$$S_{10} \vDash \mathrm{AX}\,(S_{11}),$$
$$S_{10} \vDash \mathrm{AX}\ \ (\mathit{threads\_num} = 1).$$

5. Badania eksperymentalne

W tabeli 1 przedstawiono porównanie liczby formuł wygenerowanych dla stanów programów zawierających jedną i dwie sekcje równoległe. W programie posiadającym dwie sekcje równoległe w obu przyjęto tę samą liczbę tworzonych procesów.

Tab. 1. Zestawienie wyników badań
Tab. 1. Experimental results

	Jedna sekcja równoległa (mnożenie macierzy)			Dwie sekcje równoległe		
#P	**#S**	**#F**	**t [ms]**	**#S**	**#F**	**t [ms]**
5	8	19	50	15	38	60
4	7	16	40	13	32	50
3	6	13	30	11	26	40
2	5	10	10	9	20	20

W tabeli użyto następujących oznaczeń: #P - liczba procesów, #S - liczba stanów, #F - liczba formuł, t - czas wykonani w milisekundach. Badania przeprowadzono na komputerze Intel Pentium 1,5 GHz z 512 MB pamięci operacyjnej RAM, pod systemem Windows XP pro, i wirtualną maszyną JAVA w wersji 1.5.0_06.

Ze względu na algorytm działania, liczba stanów i formuł nie zależy od konkretnej funkcjonalności algorytmu, lecz od liczby sekcji równoległych w programie. Zbliżone wyniki do przedstawionych w Tabeli 1 uzyskano dla wszystkich testowanych algorytmów z jedną bądź dwiema sekcjami równoległymi. W ramach pierwszej grupy przetestowano m.in. mnożenie macierzy i filtry FIR. Przykładem algorytmu z dwoma sekcjami równoległymi może być również przetestowany przez autorów popularny algorytm kryptograficzny DES.

Z wyników przedstawionych w Tab. 1 wynika, że liczba stanów w strukturze Kripke rośnie liniowo wraz z liczbą procesów. W sposób liniowy rośnie również liczba formuł oraz czas działania programu. Podwojenie liczby sekcji równoległych zwiększa dwukrotnie liczbę stanów i formuł przy wzroście czasu działania programu o ok. 30%. Przy dalszym zwiększaniu liczby sekcji równoległych różnice te są podobne. Oznacza to, że prezentowane podejście skaluje się bardzo dobrze i nadaje się przy weryfikacji dużych systemów, syntetyzowanych obecnie w przemyśle.

6. Podsumowanie

W artykule opisano technikę formalnej weryfikacji systemów sprzętowo-programowych zadanych za wykorzystaniem języka opisu systemu SystemC. Prezentowane podejście wykorzystuje logikę temporalną CTL i weryfikację opartą o asercje.

W artykule wymieniono asercje dla przykładu algorytmu mnożenia macierzy z jedną sekcję równoległą. Zaprezentowano także wyniki badań eksperymentalnych uzyskanych z wykorzystaniem autorskiego narzędzia do automatycznego wstawiania asercji w kod systemu. Wyniki te wykazały się bardzo dobrą skalowalnością - zarówno liczba stanów w generowanym automacie, liczba asercji i czas ich generowania rosną liniowo wraz ze wzrostem liczby procesów równoległych. Natomiast podwajanie liczby sekcji równoległych powoduje liniowy wzrost liczby stanów i formuł, przy wzroście czasu działania programu o 30 procent.

Opracowane podejście można zastosować we wszystkich zadaniach, w których dokonuje się transformacji kodu źródłowego, np. w celu przyspieszenia jego wykonania. Zastosowania te mają przede wszystkim miejsce przy systemach wbudowanych czasu rzeczywistego.

7. Literatura

[1] E. M. Clarke, E. A. Emerson, A. P. Sistla: Automatic verification of finite-state concurrent systems using temporal logic specifications, ACM Transactions on Programming Languages and Systems (TOPLAS), vol 8, no. 2, 1986, pp. 244-263.
[2] J. Horgan, Assertion Based Verification: EDAWeekly Review, October 18 - 22, 2004.
[3] N. Jayakumar, M. Purandare, F. Somenzi: Simulation coverage and generation for verification: Dos and don'ts of CTL state coverage estimation, Proceedings of the 40th Design Automation Conference (DAC'03), 2003, pp. 292-295.
[4] K. Trifunović, P. Dziurzański, W. Bielecki: Sprzętowa implementacja pętli programowych, Pomiary, Automatyka, Kontrola, nr 7BIS, 2006, pp. 59-61.
[5] M. Varea, B. M. Al-Hashimi, L. A. Cortés, P. Eles, Z. Peng: System design methods: analysis and verification: Symbolic model checking of Dual Transition Petri Nets, Proceedings of the Tenth International Symposium on Hardware/Software Codesign, 2002, pp. 43-48.
[6] SystemC Version 2.0 User's Guide, www.systemc.org, 2002
[7] C. Kern, M. R. Greenstreet: Formal verification in hardware design: a survey, ACM Transactions on Design Automation of Electronic Systems (TODAES), vol. 4, no. 2, 1999, pp. 123-193.
[8] M. Ben-Ari: Logika matematyczna w informatyce, WNT, Warszawa, 2005.
[9] G. E. Fainekos, H. Kress-Gazit, G. J. Pappas: Temporal Logic Motion Planning for Mobile Robots, Proceedings of the 2005 IEEE International Conference on Robotics and Automation, Barcelona, Spain, April 2005.

Artykuł recenzowany