CZASOPISMO TECHNICZNE
WYDAWNICTWO POLITECHNIKI KRAKOWSKIEJ
z. 1-I/2008
ISSN 0011-4561
ISSN 1897-631X

STANISŁAW DENIZIAK, ADAM GÓRSKI*

KOSYNTEZA SYSTEMÓW SOC METODĄ ROZWOJOWEGO PROGRAMOWANIA GENETYCZNEGO

HARDWARE/SOFTWARE CO-SYNTHESIS OF SOC SYSTEMS USING DEVELOPMENTAL GENETIC PROGRAMMING

Streszczenie

W niniejszym artykule zaprezentowano nową metodę kosyntezy systemów wbudowanych specyfikowanych za pomocą grafów zadań, bazującą na metodzie programowania genetycznego. Przedstawiono propozycje reprezentowania procesu konstrukcji takiego systemu w formie drzewa stanowiącego tzw. genotyp. Następnie na skutek ewolucji (krzyżowania, mutacji, selekcji) generowane są kolejne „pokolenia" drzew, konstruujących systemy o coraz lepszych parametrach. Przedstawione wyniki wykonanych eksperymentów świadczą o dużych możliwościach metody RPG również w zakresie kosyntezy.

Słowa kluczowe: *programowanie genetyczne*, *kosynteza*

Abstract

This work presents a novel approach to hardware-software co-synthesis of distributed embedded systems, based on the developmental genetic programming. Unlike other genetic approaches where chromosomes represent solutions, in our method chromosomes represent system construction procedures. Thus, not the system architecture but the co-synthesis process is evolved. Finally a tree describing a construction of the final solution is obtained. The optimization process will be illustrated with examples.

Keywords: *genetic programming*, *hardware/software co-design*

* Dr hab. inż. Stanisław Deniziak, mgr inż. Adam Górski, Katedra Informatyki Technicznej, Wydział Inżynierii Elektrycznej i Komputerowej, Politechnika Krakowska.

Oznaczenia

α – współczynnik określający wielkość populacji w zależności od liczby zadań
β – współczynnik reprodukcji
ε – kryterium końca ewolucji
γ – współczynnik krzyżowania
δ – współczynnik mutacji
Φ – liczba rozwiązań generowanych przez reprodukcję
Ψ – liczba rozwiązań generowanych przez krzyżowanie
Π – wielkość populacji
Ω – liczba rozwiązań generowanych przez mutację
$b_{\mathrm{CL}i}$ – przepustowość i-tego kanału komunikacyjnego
c_i – i-te ograniczenie czasowe
A – całkowita powierzchnia systemu SOC
CL – kanał komunikacyjny
E – zbiór krawędzi grafu zadań
$d_{j,j}$ – wielkość transmisji związana z daną krawędzią grafu zadań
$e_{j,j}$ – krawędzie w grafie zadań
f_R – częstotliwość powtarzania zadań
G – graf zadań
HC – wyspecjalizowany moduł sprzętowy
PE – element obliczeniowy: procesor lub wyspecjalizowany moduł sprzętowy
PG – programowanie genetyczne
PP – rdzeń procesora uniwersalnego
RPG – rozwojowe programowanie genetyczne
s_i – powierzchnia układu scalonego wymagana do implementacji i-tego zadania
$S_{\mathrm{PE}i}$ – powierzchnia modułu PEi
SOC – system jednoukładowy (*System on Chip*)
$t_{j,j}$ – czas wykonania zadania v_i przez komponent PE_j
$tc_{i,j}$ – czas trwania transmisji danych pomiędzy zadaniami v_i i v_j
ts_i – czas rozpoczęcia wykonywania i-tego zadania
v_i, v_j – zadania, węzły w grafie zadań
V – zbiór węzłów grafu zadań

1. Wstęp

Współczesne systemy wbudowane charakteryzują się coraz większą złożonością, wynikającą z integracji różnorodnych funkcji w jednym systemie [1]. Z drugiej strony, rosną wymagania związane z poborem mocy, kosztem, szybkością działania oraz krótkim czasem projektowania. Aby sprostać tym wymaganiom, konieczne jest stosowanie komputerowych metod wspomagających projektowanie wyspecjalizowanych architektur systemów wbudowanych, a przede wszystkim opracowanie efektywnych metod optymalizujących projektowane systemy wbudowane pod względem ww. parametrów. Proces komputerowego projektowania architektury systemu nazywany jest syntezą systemową.

W przypadku systemów wbudowanych synteza dotyczy zarówno komponentów sprzętowych, jak i oprogramowania, dlatego wtedy mówi się o kosyntezie sprzętowo-programowej [2]. Efektem kosyntezy są coraz częściej wieloprocesorowe architektury, złożone z różnego rodzaju procesorów, wyspecjalizowanych modułów sprzętowych, pamięci itp., czyli mają charakter heterogeniczny i rozproszony. Architektury te najczęściej implementowane są jako jednoukładowe systemy typu SOC.

Kosynteza polega na automatycznej generacji architektury systemu wbudowanego na podstawie specyfikacji przedstawionej w formie współbieżnych procesów. Celem kosyntezy jest optymalizacja właściwości systemu, takich jak: koszt, czas wykonania lub pobór mocy. Większość współczesnych metod kosyntezy zakłada architekturę rozproszoną, złożoną z elementów obliczeniowych PE (*Processing Element*), które są komponentami sprzętowymi HC (*Hardware Core*) bądź *softwarowymi* PP (*Programmable Processor*).

Proces kosyntezy składa się z następujących zadań:

– alokacja zasobów: określa ilość i typy elementów obliczeniowych oraz kanałów komunikacyjnych dla architektury docelowej,
– przyporządkowanie zadań do zasobów: wybór elementów obliczeniowych wykonujących poszczególne zadania oraz transmisje pomiędzy nimi,
– szeregowanie zadań: określa czas rozpoczęcia wykonywania każdego z zadań.

Zarówno szeregowanie, jak też przyporządkowanie zadań do zasobów to problemy NP--zupełne. Dlatego znalezienie najlepszej docelowej architektury dla danego systemu jest możliwe tylko z zastosowaniem skutecznych metod heurystycznych.

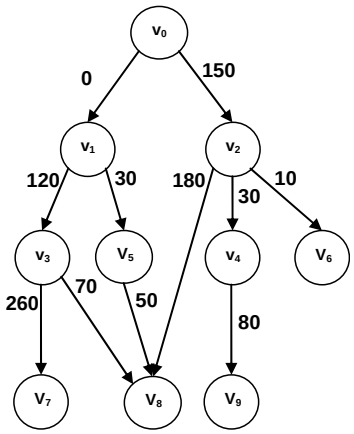
Większość istniejących algorytmów kosyntezy [3–5] są to algorytmy rafinacyjne. Metody te startują od rozwiązania suboptymalnego i starają się udoskonalić jakość systemu przez lokalne modyfikacje architektury. Jako rozwiązanie początkowe zwykle wybierana jest architektura zapewniająca wykonanie wszystkich zadań w najkrótszym czasie, gdzie każdy proces jest wykonywany przez inny PE. Modyfikacje polegają na przemieszczaniu zadań do innych PE, usuwaniu i dodawaniu PE itp. Niektóre algorytmy rafinacyjne są zdolne do opuszczania lokalnego minimum optymalizowanych parametrów [4], ale rezultaty wciąż są suboptymalne. Algorytmy konstrukcyjne budują system poprzez alokację nowych komponentów [6, 7]. Ponieważ takie podejście pozwala ocenić tylko lokalne efekty decyzji projektowych, używane są różne przybliżone miary, aby przewidzieć globalny efekt tych decyzji. Algorytmy konstrukcyjne mają niską złożoność obliczeniową, ale mają też tendencję do zatrzymywania się w lokalnych minimach optymalizowanych parametrów. Jakość uzyskanych wyników można poprawić poprzez cofanie się w konstrukcji rozwiązania [6], jednak stwarza to niebezpieczeństwo zapętlenia się algorytmu, a ponadto znacznie zwiększa jego złożoność obliczeniową. Algorytmy probabilistyczne mają zdolność do wydobywania się z lokalnych minimów. W szczególności dosyć dobre wyniki uzyskano z wykorzystaniem algorytmów genetycznych [8]. Metody genetyczne są też często używane do optymalizacji na wybranych etapach syntezy systemowej [9, 10].

W niniejszym artykule zaprezentowano nową metodę kosyntezy sprzętowo-softwarowej opartą na rozwojowym programowaniu genetycznym [11]. Istniejące, genetyczne metody kosyntezy definiują ewolucję przez rafinację architektury, w której chromosomy definiują rozwiązania. W prezentowanym algorytmie ewolucji podlega proces kosyntezy, a chromosomy reprezentują decyzje projektowe. Dzięki temu jako rezultat ewolucji otrzymujemy metodę konstruowania docelowego systemu. W następnym punkcie zostanie przedstawiony problem kosyntezy systemów wbudowanych implementowanych jako SOC.

Rozdział 3 zawiera opis rozwojowego programowania genetycznego. W rozdziale 4 przedstawiono proponowaną metodę kosyntezy. Ostatnie dwa rozdziały zawierają wyniki wykonanych eksperymentów oraz wnioski.

2. Problem kosyntezy systemów SOC

Zachowanie projektowanego systemu wbudowanego można opisać za pomocą grafu zadań $G = \{V, E\}$, który jest acyklicznym grafem skierowanym [2]. Każdy węzeł $v_i \in V$ reprezentuje zadanie, a krawędź $e_{i,j} \in E$ opisuje zależność między zadaniami v_i oraz v_j. Każda krawędź ma etykietę $d_{i,j}$ opisującą ilość danych, która musi być przesłana pomiędzy połączonymi zadaniami. Przykładowy graf zadań przedstawiono na rys. 1.



Rys. 1. Przykładowy graf zadań

Fig. 1. Sample task graph

Zakładamy, że istnieje baza danych elementów obliczeniowych i kanałów komunikacyjnych CL (*Communication Link*), zawierająca czasy wykonania zadań (t), powierzchnię modułów (s) dla implementacji w formie systemu SOC oraz przepustowość kanałów komunikacyjnych (b). Rozróżniamy dwa podstawowe typy PE: programowalne procesory uniwersalne (PP) oraz wyspecjalizowane moduły sprzętowe (HC). Każdy PP_j może wykonywać wszelkie zadania, które są z nim kompatybilne. Rozmiar zadania PP_i określa powierzchnię zajmowaną przez pamięć wymaganą przez dane zadanie, natomiast S_{PP_i} określa powierzchnię procesora PP_i. Moduły sprzętowe HC wykonują tylko jedno zadanie v_i, ale może istnieć wiele implementacji sprzętowych tego samego zadania. Szyny komunikacyjne są zdefiniowane poprzez przepustowość kanału oraz obszar s_{ij}, który zajmuje szyna podłączana do PE_i. W tabeli 1 pokazano przykładową bazę zasobów dla systemu opisanego przez graf zadań z rys. 1. Zadanie v_2 nie jest kompatybilne z elementem PP_1, natomiast v_5 ma tylko jedną implementację sprzętową. Każde z pozostałych zadań ma cztery alternatywne implementacje. Dostępne są dwie szyny komunikacyjne. CL_2 nie jest kompatybilna z procesorem PP_1.

Z każdym zadaniem v_i może być związane wymaganie czasowe c_i, oznaczające chwilę, do której zadanie musi zostać wykonane. Wszystkie wymagania czasowe muszą być dotrzymane przez architekturę docelową. Niech ts_i będzie czasem rozpoczęcia wykonywania zadania v_i. Docelowy system jest poprawny wtedy i tylko wtedy, gdy spełnione są następujące warunki

$$\bigvee_{e_{i,j}} ts_j \geq ts_i + t_{i,r_i} + tc_{i,j} \tag{1}$$

$$\bigvee_{i} ts_i + t_{i,r_i} \leq c_i \tag{2}$$

gdzie

$$tc_{i,j} = \left\lceil \frac{d_{i,j}}{b_{\mathrm{CL}_{i,j}}} \right\rceil \tag{3}$$

r_i oznacza typ PE wykonującego zadanie v_i, $\mathrm{CL}_{i,j}$ jest typem szyny komunikacyjnej. Jeśli zadania v_i oraz v_j są przyporządkowane do tego samego PE, wówczas $tc_{i,j} = 0$. Warunek (1) zakłada poprawne uporządkowanie zadań, podczas gdy (2) zapewnia spełnienie wszystkich wymagań czasowych.

Tabela 1

Baza danych zasobów

	PP_1 $S = 100$		PP_2 $S = 200$		$\mathrm{HC1}_i$		$\mathrm{HC2}_i$	
	t	s	t	s	t	s	t	S
V_0	30	3	10	2	3	50	4	10
V_1	50	5	20	4	6	80	5	20
V_2	–	–	10	3	3	60	5	20
V_3	10	3	8	1	1	20	2	5
V_4	30	3	15	2	4	70	10	30
V_5	30	5	30	3	5	110	–	–
V_6	40	3	15	2	10	70	12	15
V_7	30	3	15	2	5	50	8	18
V_8	8	3	5	1	2	30	3	10
V_9	10	3	5	1	3	40	4	12
CL_1 $B = 8$	$s = 2$		$s = 1$		$s = 10$			
CL_2 $B = 16$	$s = 3$		$s = 4$		$s = 15$			

Niech architektura docelowa dla systemu opisanego grafem zadań zawierającym n procesów składa się z m programowalnych procesorów, p szyn komunikacyjnych, wtedy całkowita powierzchnia zajmowana przez system SOC wyraża się wzorem

$$A = \sum_{i=1}^{m} S_{\mathrm{PE}_i} + \sum_{j=1}^{n} s_j + \sum_{k=1}^{p} \sum_{l=1}^{P_k} s_{\mathrm{CL}_k,\mathrm{PC}_l} \tag{4}$$

gdzie: CL_k jest typem k-tej magistrali podłączonej do P_k i PC_l jest typem l-tego PE podłączonego do CL_k. Celem kosyntezy jest znalezienie architektury z najmniejszym A zapewniającym zachowanie warunków (1) i (2).

3. Rozwojowe programowanie genetyczne

Programowanie genetyczne (PG) [11] jest metodą optymalizacji polegającą na generacji programów za pomocą algorytmu genetycznego. Optymalizacja polega na ewolucji drzewa struktury programu, w taki sposób, aby uzyskać jak najlepszy program, który dla zadanych zestawów danych wejściowych generuje oczekiwane wyniki. Metoda PG okazała się skuteczna dla generacji efektywnych programów wykonujących obliczenia matematyczne, algorytmów sterowania robotami, rozpoznawania tekstów, analizy danych i wielu innych dziedzin. Znanych jest 36 problemów, dla których metodą PG znaleziono rozwiązania porównywalne lub lepsze od rozwiązań znalezionych ręcznie [12].

Rozszerzoną wersją PG jest rozwojowe programowanie genetyczne (RPG) [13], w którym zamiast programów (czyli gotowych rozwiązań) generowane są algorytmy konstruujące rozwiązanie danego problemu. Algorytmy te reprezentowane są przez drzewa, w których węzły odpowiadają poszczególnym decyzjom projektowym, a krawędzie opisują ich kolejność. Opracowano skuteczne metody RPG do rozwiązywania wielu problemów z zakresu konstrukcji liniowych układów elektronicznych (filtry, wzmacniacze), syntezy układów logicznych i innych. Jednak jak do tej pory nie przebadano możliwości zastosowania metody RPG w kosyntezie systemów wbudowanych.

Metoda RPG jest szczególnie efektywna dla optymalizacji problemów silnie ograniczonych (a takim jest problem kosyntezy). Wtedy większość rozwiązań, generowanych losowo (jak w tradycyjnym podejściu genetycznym) lub deterministycznie (jak w metodach rafinacyjnych), nie spełnia wymagań (funkcjonalnych lub czasowych). Powoduje to konieczność narzucenia ograniczeń sterujących procesem optymalizacji w taki sposób, aby rozpatrywane były tylko poprawne rozwiązania. Jednak ograniczenia te jednocześnie zawężają przestrzeń przeszukiwań, również uniemożliwiając znalezienie rozwiązań najlepszych, jeśli są one efektem rafinacji lub ewolucji rozwiązań niepoprawnych. Problemy te nie występują w RPG, gdzie ewolucja przebiega bez żadnych ograniczeń, natomiast poprawność rozwiązań jest zapewniona przez odpowiednie odwzorowanie genotypu (procedury konstrukcyjnej) w fenotyp (rozwiązanie). Taka cecha metody RPG umożliwia w wielu przypadkach uzyskanie dużo lepszych wyników niż w dotychczas stosowanych metodach.

4. Kosynteza metodą programowania genetycznego

Zgodnie z zasadą rozwojowego programowania genetycznego w opracowywanym algorytmie ewolucji podlega drzewo (genotyp) opisujące konstruowanie projektowanego systemu. Korzeń określa konstrukcję systemu embrionicznego, węzły odpowiadają funkcjom budującym system. Aby zapewnić implementację wszystkich zadań, struktura

genotypu jest drzewem odwzorowującym graf zadań. Każdy węzeł drzewa genotypu implementuje zadanie odpowiadające danemu węzłowi w grafie zadań. Struktura tego drzewa jest zachowana po mutacji i krzyżowaniu.

4.1. System embrionalny

Embrion implementuje pierwsze zadanie z podanego grafu zadań, a zatem liczba możliwych embrionów jest równa liczbie elementów PE w bibliotece zasobów. Dla każdego rozwiązania embrion jest generowany losowo. Dla przykładu z rys. 1 każdy z czterech PE z tab. 1 może być embrionem.

4.2. Funkcje konstruujące system

Każdy węzeł w drzewie genotypu reprezentuje funkcję implementującą jedno zadanie z grafu zadań. Każda funkcja składa się z następujących kroków:

1. Alokacja nowego PE: ten krok jest opcjonalny. Krok ten jest również zawsze wykonywany, jeśli nie jest możliwe wykonanie kroku 2 bez alokacji nowego PE.
2. Przyporządkowanie zadania do PE: ten krok zawsze musi być wykonany. Jeśli poprzedni krok został wykonany, wtedy zadanie jest przyporządkowane do nowego PE, w przeciwnym razie do dowolnego PE z aktualnej architektury.
3. Alokacja nowego CL: ten krok jest opcjonalny. Krok ten jest również zawsze wykonywany, jeśli nie jest możliwe wykonanie kroku 4 bez alokacji nowego CL.
4. Przyporządkowanie transmisji do CL: kroki 3 i 4 są powtarzane dla każdej krawędzi wchodzącej do węzła odpowiadającego implementowanemu zadaniu.
5. Szeregowanie zadań: ten krok jest wykonywany tylko wtedy, gdy istnieje przynajmniej jeden PP z więcej niż jednym przyporządkowanym zadaniem.

Najpierw generowane jest pokolenie początkowe złożone z losowo wygenerowanych genotypów. Wielkość pokolenia początkowego definiuje parametr α i wynosi: $\Pi = \alpha \cdot n \cdot e$ (n jest liczbą zadań, natomiast e – liczbą możliwych embrionów). Dla każdej funkcji wszystkie kroki są generowane losowo zgodnie z tab. 2. Ostatnia kolumna przedstawia prawdopodobieństwo wyboru danej opcji. Jeżeli alokowany jest nowy PE lub CL, o wyborze decydują szybkość, powierzchnia lub najmniejszy iloczyn czasu obliczeń i powierzchni. Największe prawdopodobieństwo ma jednak opcja „żaden" niepowodująca alokacji żadnego nowego PE lub CL. Opcja „najrzadziej alokowany" gwarantuje, że wszystkie zasoby z bazy zasobów będą brane pod uwagę podczas konstrukcji systemu.

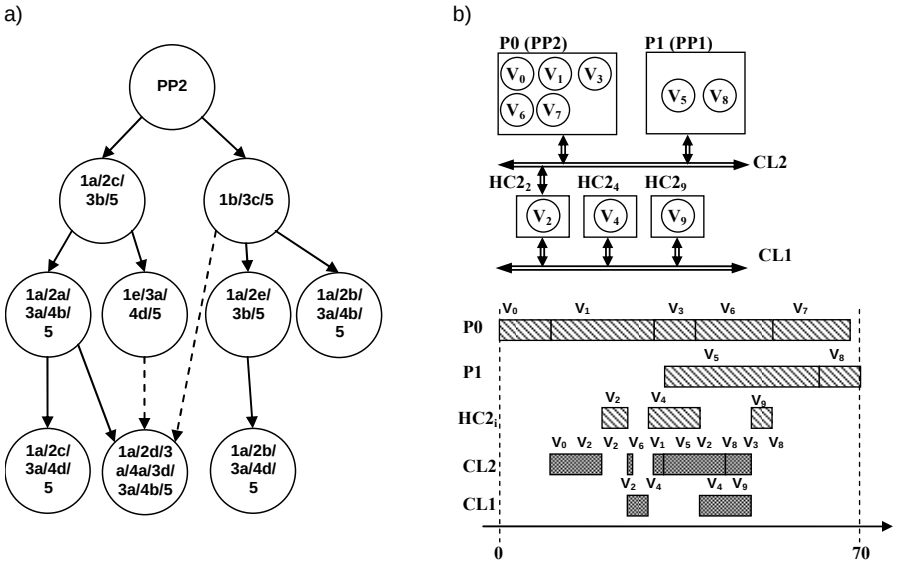
System jest konstruowany przez wykonywanie funkcji wg poziomu węzła w drzewie genotypu. Węzły znajdujące się na tym samym poziomie są wykonywane od lewej do prawej. Zawsze wybierane są tylko opcje zapewniające spełnienie ograniczeń czasowych. Na rysunku 2a) przedstawiono przykładowy genotyp dla grafu zadań z rys. 1. Zakładając, że maksymalny czas zakończenia zadań wynosi $c_5 = 70$, wtedy system jest konstruowany następująco:

- PP2: procesor P0 typu PP2 jest embrionem i wykonuje zadanie V_0,
- 1a/2c/3b/5: funkcja nie alokuje nowego procesora, zatem zadanie V_1 jest również przydzielone do P0, ponieważ zadania V_0 i V_1 są przyporządkowane do tego samego procesora – nie ma potrzeby alokacji kanału komunikacyjnego i przyporządkowania transmisji, jedyne możliwe uszeregowanie zadań to $V_0 \rightarrow V_1$,

Tabela 2

Opcje wyboru dla konstruowanego systemu

Krok	Opcja	P
1	a) żaden	0,6
	b) najmniejsza powierzchnia	0,1
	c) najszybszy	0,1
	d) najmniejsze $t \cdot s$	0,1
	e) najrzadziej alokowany	0,1
2	a) najmniejsza powierzchnia	0,2
	b) najszybszy	0,2
	c) najmniejsze wykorzystanie	0,2
	d) najdłużej bezczynny	0,2
	e) tak jak dla poprzednika	0,2
3	a) żadna	0,5
	b) najmniejsza powierzchnia	0,2
	c) największe b	0,2
	d) najrzadziej alokowana	0,1
4	a) najmniejsza powierzchnia	0,3
	b) najszybsza	0,3
	c) najmniejsze wykorzystanie	0,2
	d) najdłużej bezczynna	0,2
5	szeregowanie listowe	1,0



Rys. 2. Przykładowy genotyp (a) i odpowiadający mu fenotyp (b)

Fig. 2. Sample genotype (a) and the corresponding phenotype (b)

– 1b/3c/5: dla zadania V_2 alokowany jest PE o najmniejszej powierzchni, tzn. $HC2_2$, alokowana jest magistrala komunikacyjna o największej przepustowości (CL2) do transmisji pomiędzy zadaniami V_0 i V_2,
– 1a/2a/3a/4b/5: zadanie V_3 jest przyporządkowane do P0 (najmniejszy wzrost powierzchni), ponieważ zadania V_1 i V_3 są przyporządkowane do tego samego procesora – nie ma potrzeby wykonywania transmisji i krok 4 nie jest wykonywany, jedyne możliwe uszeregowanie zadań przyporządkowanych do P0 to $V_0 \rightarrow V_1 \rightarrow V_3$,
– 1e/3a/4d/5: dla zadania V_5 alokowany jest procesor P1 typu PP1 (najrzadziej alokowany), transmisja $V_1 \rightarrow V_5$ jest przydzielona do CL1 i uszeregowana po transmisji $V_0 \rightarrow V_2$ (V_0 kończy się wcześniej niż V_1),
– 1a/2e/3b/5: zadanie V_4 jest implementowane jak poprzednik, czyli implementacja sprzętowa o najmniejszej powierzchni ($HC2_4$). Do transmisji alokowany jest kanał o najmniejszej powierzchni (CL1),
– 1a/2b/3a/4b/5: zadanie V_6 jest przyporządkowane do najszybszego dotychczas alokowanego procesora (P0), a transmisja $V_2 \rightarrow V_6$ zostaje przyporządkowana do łącza CL2, zadania są uszeregowane w kolejności $V_0 \rightarrow V_1 \rightarrow V_3 \rightarrow V_6$, natomiast transmisje zostaną uszeregowane w kolejności $V_0 \rightarrow V_2$, $V_2 \rightarrow V_6$, $V_1 \rightarrow V_5$,
– 1a/2c/3a/4d/5: zadanie V_7 powinno być przyporządkowane do procesora P1 (jako najmniej obciążonego), jednak wtedy zostanie naruszone ograniczenie czasowe, nawet przy najszybszej implementacji pozostałych zadań, dlatego takie rozwiązanie nie jest brane pod uwagę. Kolejną możliwością jest przyporządkowanie zadania do procesora P0, wtedy transmisja $V_3 \rightarrow V_7$ nie jest potrzebna i krok 4 jest pominięty,
– 1a/2d/3a/4a/3d/3a/4b/5: zadanie V_8 zostaje przyporządkowane do procesora P1, transmisja $V_3 \rightarrow V_8$ zostanie przyporządkowana do kanału CL2 (nie powoduje wzrostu powierzchni), natomiast dla transmisji $V_5 \rightarrow V_8$ powinien być alokowany nowy kanał, ale nie ma takiej potrzeby (oba zadania są przyporządkowane do tego samego procesora), transmisja $V_2 \rightarrow V_8$ zostanie przyporządkowana do CL2 jako do najszybszego łącza,
– 1a/2b/3a/4d/5: zadanie V_9 powinno być przyporządkowane do procesora P0 (najmniejszy wzrost powierzchni), ale to naruszy ograniczenie czasowe, przyporządkowanie zadania do P1 też naruszy ograniczenia, w związku z tym dla tego zadania zostanie alokowany nowy moduł sprzętowy $HC2_9$, a transmisja $V_4 \rightarrow V_9$ do łącza CL1 (najdłużej bezczynnego).

Implementacja zadań V_7 i V_9 pokazuje, w jaki sposób następuje odwzorowanie tylko w poprawne fenotypy. Podczas konstrukcji rozwiązania podejmowane są tylko takie decyzje projektowe, które prowadzą do rozwiązania spełniającego zadane wymagania czasowe. Funkcje konstruujące system uwzględniają tylko takie implementacje zadań, które zapewniają uzyskanie systemu poprawnego dla najlepszego przypadku implementacji pozostałych zadań. Na rysunku 2b) przedstawiono architekturę systemu oraz uszeregowanie zadań i transmisji, skonstruowane przez genotyp przedstawiony na rys. 2a). Ostatecznie otrzymano system wykonujący wszystkie zadania w czasie 70 i o koszcie 433.

4.3. Ewolucja

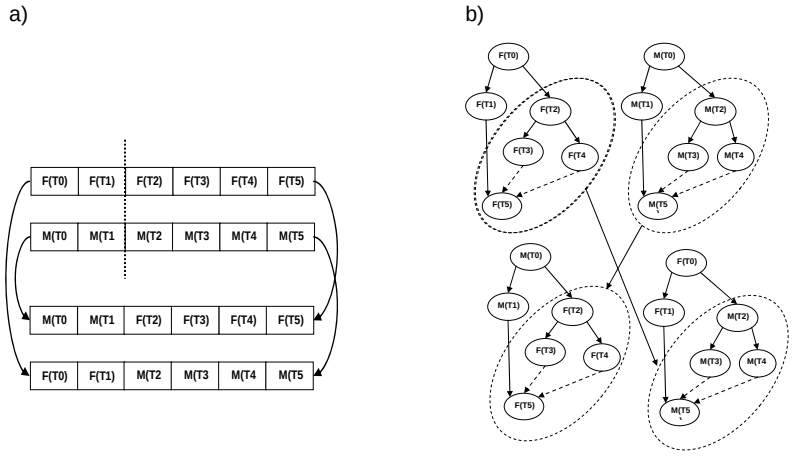
Na drodze ewolucji za pomocą operatorów genetycznych: reprodukcji, krzyżowania i mutacji, generowane są nowe pokolenia rozwiązań. Liczba rozwiązań w każdej populacji zawsze wynosi Π. Ewolucja jest kontrolowana przez parametry β, γ i δ w następujący sposób:

– $\Phi = \beta \cdot \Pi$: liczba rozwiązań uzyskana przez użycie reprodukcji,
– $\Psi = \gamma \cdot \Pi$: liczba rozwiązań uzyskana przez użycie krzyżowania,
– $\Omega = \delta \cdot \Pi$: liczba rozwiązań uzyskana przez użycie mutacji,
– $\beta + \gamma + \delta = 1$: w każdej populacji jest taka sama liczba osobników.

Dla każdego pokolenia dokonywany jest ranking rozwiązań (od najtańszego do najdroższego). Reprodukcja kopiuje Φ rozwiązań z aktualnej populacji. Rozwiązania są wybierane losowo, ale z różnym prawdopodobieństwem, które jest uzależnione od pozycji r w rankingu

$$P = \frac{\Pi - r}{\Pi} \tag{5}$$

Krzyżowanie losowo wybiera dwa rozwiązania, następnie pojedynczy punkt przecięcia jest wybierany losowo dla obu rodziców. Na rysunku 3 przedstawiono przykładowe krzyżowanie dwóch rozwiązań. Mutacja wybiera losowo jedno rozwiązanie, następnie losowo wybiera węzeł i podmienia go, generując nowe opcje (z tab. 2). Algorytm zatrzymuje się, jeśli najlepsze rozwiązanie nie zostało znalezione w ε kolejnych krokach.



Rys. 3. Krzyżowanie genotypów reprezentowanych przez łańcuch (a) oraz ta sama operacja wykonana na grafach (b)

Fig. 3. Crossover of genotypes represented with strings (a), and the same operation for genotypes represented with graphs (b)

5. Wyniki wykonanych eksperymentów

Ze względu na dużą złożoność problemu kosyntezy znalezienie najlepszych rozwiązań jest możliwe tylko dla prostych grafów (do kilkunastu węzłów), dlatego jedyną możliwością oceny skuteczności zaprezentowanej metody jest wykonanie eksperymentów i porównanie wyników z wynikami uzyskanymi za pomocą innych metod. Ponieważ nie ma

standardowych benchmarków, do takiej oceny najczęściej stosowane są grafy zadań wygenerowane losowo. W eksperymentach wykonano syntezę dla grafów złożonych z od 10 do 110 węzłów. Parametry zadań były generowane w taki sposób, aby odpowiadały parametrom rzeczywistych zadań. Analogiczne eksperymenty wykonano dla algorytmów rafinacyjnych: Yen–Wolf [3] oraz EWA [4], przy czym algorytm Yen–Wolf został przystosowany do syntezy systemów SOC. Algorytm EWA okazał się skuteczniejszy niż algorytm MOGAC [4], dlatego porównanie z tym algorytmem powinno dać odpowiedź na pytanie, czy i na ile podejście RPG jest skuteczniejsze od klasycznego podejścia genetycznego. W tabeli 3 przedstawiono wyniki syntezy, kolejne kolumny zawierają: liczbę węzłów grafu, ograniczenie czasowe oraz dla każdego algorytmu przedstawione są czas wykonania wszystkich zadań oraz koszt architektury. W algorytmie RPG występuje czynnik losowy, powodujący, że przy kolejnych wykonaniach algorytm generuje różne rozwiązania. Dlatego aby uniknąć przypadkowości, wykonano po 30 prób dla każdego grafu. W tabeli 3 przedstawiono średnią z tych prób, a także najlepsze wyniki. We wszystkich przypadkach metodą RPG uzyskano lepsze wyniki niż w algorytmie Yen–Wolf. Natomiast metoda RPG jest porównywalna z algorytmem EWA pod względem efektywności. Jednak biorąc pod uwagę najlepsze wyniki, widać, że metodą RPG można, w większości przypadków, uzyskać wyniki lepsze. Ostatni wiersz tabeli 3 zawiera sumę kosztów uzyskanych rozwiązań, pozwala to na sumaryczne porównanie efektywności tych metod.

Tabela 3

Wyniki syntezy dla losowych grafów zadań

N	T_{max} [ms]	Yen–Wolf		Ewa		**RPG-średnia**		**RPG – min.**	
		czas	koszt	czas	koszt	**czas**	**koszt**	**czas**	**koszt**
10	400	315	1573	287	1517	**395**	**1552**	**395**	**1545**
20	450	196	3046	196	3046	**396**	**2665**	**396**	**2649**
30	500	499	4213	488	4361	**473**	**3988**	**484**	**3823**
40	800	794	5204	779	5188	**782**	**4959**	**795**	**4595**
50	1100	1099	6017	1092	5967	**1079**	**5289**	**1090**	**5046**
60	1400	1360	8218	1386	7316	**1337**	**8086**	**1376**	**7784**
70	1600	1548	6859	1590	6657	**1554**	**5832**	**1599**	**5607**
80	1900	1893	11 692	1878	8662	**1745**	**10 243**	**1854**	**9918**
90	2000	1917	13 184	1995	8257	**1943**	**7012**	**1986**	**6599**
100	2150	2115	10 800	2140	7240	**2098**	**8524**	**2133**	**7941**
110	2200	2167	12 171	2199	9030	**2142**	**9098**	**2193**	**8499**
Suma			**82 977**		**6 7241**		**67 248**		**64 006**

Ograniczenia czasowe zawężają przestrzeń przeszukiwań, a zatem im to ograniczenie jest ostrzejsze, tym większa liczba rozwiązań jest niepoprawna. Aby sprawdzić, w jaki sposób wpływa to na skuteczność algorytmu RPG, wykonano eksperymenty dla grafu o 50 węzłach, wybierając różne ograniczenia czasowe. Wyniki przedstawiono w tab. 4. W porównaniu z metodami rafinacyjnymi metoda RPG jest efektywniejsza dla przypadków z silnymi ograniczeniami. Potwierdza to tezę, że metoda ta może być skuteczna dla pro-

blemów silnie ograniczonych. Wtedy metody rafinacyjne, ze względu na niewielką swobodę w rafinacji rozwiązań, szybko zatrzymują się w lokalnych minimach. Natomiast metoda RPG jest mniej skuteczna dla przypadków ze słabszymi ograniczeniami, zapewne wynika to ze znacznie większej przestrzeni możliwych rozwiązań. Aby poprawić skuteczność tej metody, prawdopodobnie należałoby wtedy zwiększyć liczebność populacji rozwiązań, czyli uzależnić wielkość populacji od ograniczenia czasowego.

Tabela 4

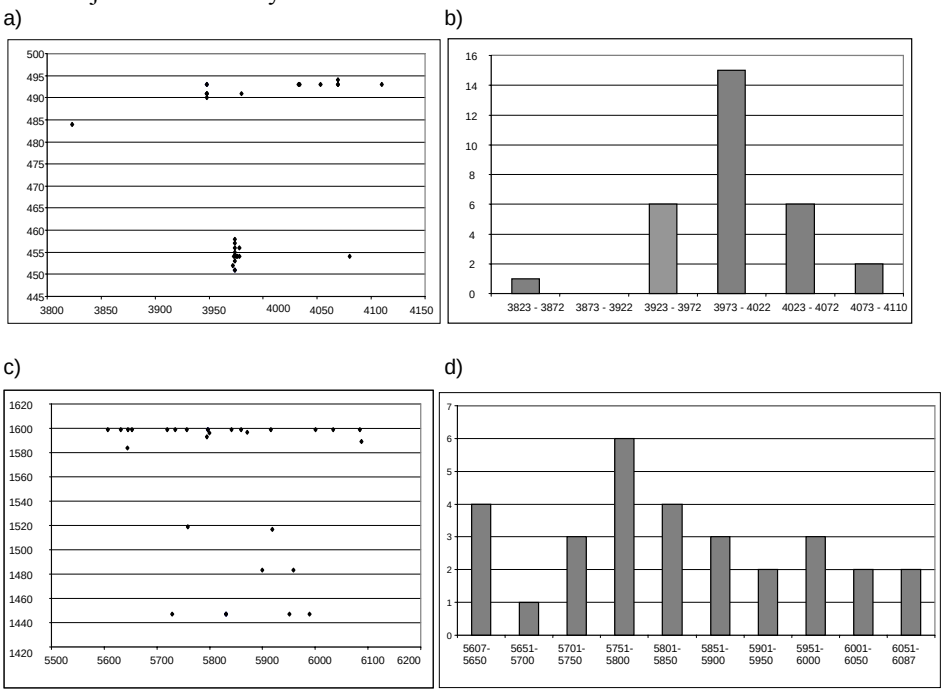
Wyniki syntezy dla różnych ograniczeń czasowych

T_{max} [ms]	Yen–Wolf		Ewa		**RPG – średnia**		**RPG – min.**	
	czas	koszt	czas	koszt	**czas**	**koszt**	**czas**	**koszt**
720	708	7099	717	7134	**700**	**7050**	**649**	**6601**
780	777	6708	772	6609	**763**	**6870**	**773**	**6121**
840	831	6889	836	6397	**819**	**6615**	**823**	**6171**
900	883	6251	895	6304	**877**	**6382**	**895**	**5805**
1000	999	7457	1000	6017	**952**	**6116**	**961**	**5575**
1200	1172	6182	1179	5606	**1180**	**5496**	**1176**	**4923**
1500	1445	5835	1489	4603	**1412**	**5132**	**1398**	**4829**
2000	1942	4783	1959	4191	**1873**	**4792**	**1808**	**4461**
2500	2456	3754	2436	3498	**2351**	**4458**	**2500**	**3974**
3000	2914	3429	2926	3163	**2836**	**3861**	**2914**	**3329**

Zarówno uśrednione parametry znalezionych rozwiązań, jak i wynik najlepszy nie pokazują w pełni skuteczności algorytmu RPG. W przypadku dużych grafów nawet uzyskanie wyników w pobliżu średniej może wymagać wielokrotnego powtarzania obliczeń. Dlatego konieczne jest zbadanie rozrzutu wyników, aby określić prawdopodobieństwo uzyskania wyniku w pobliżu lub poniżej średniej, a także aby oszacować, o ile można poprawić uzyskany wynik poprzez powtarzanie obliczeń. Na rysunku 4 przedstawiono wykresy pokazujące parametry wszystkich uzyskanych wyników (z 30 prób) dla grafów o 30 (rys. 4a) i b)) i 70 (rys. 4c) i d)) węzłach. W obu przypadkach pokazano położenie rozwiązania w przestrzeni, czas obliczeń (oś Y) oraz koszt (oś X), a także wykresy słupkowe ilustrujące liczbę rozwiązań w poszczególnych przedziałach kosztu. Podobne wykresy wykonano dla pozostałych grafów. Na podstawie otrzymanych wykresów można zauważyć, że:

- wszystkie wyniki znajdują się w przedziale: koszt średni ±5%, a zatem rozrzut wyników nie jest zbyt duży,
- najwięcej rozwiązań zlokalizowanych jest wokół średniej, przy czym przy tej samej liczbie powtórzeń zależność ta jest coraz mniej wyraźna wraz ze wzrostem wielkości grafu, a zatem aby prawdopodobieństwo uzyskania dobrego wyniku było takie samo, należałoby zwiększać liczbę powtórzeń algorytmu,
- w wielu przypadkach nawet najgorszy uzyskany wynik jest lepszy od wyniku uzyskanego za pomocą innych metod kosyntezy.

Obserwacje te pozwalają na stwierdzenie, że uzyskane wyniki nie są przypadkowe, a metoda RPG jest bardzo efektywna.



Rys. 4. Rozkład wyników dla badanych grafu grafów: a) rozwiązania znalezione dla grafu *G*30, b) statystyka rozwiązań uzyskanych dla grafu *G*30, c) rozwiązania znalezione dla grafu *G*70, d) statystyka rozwiązań uzyskanych dla grafu *G*70

Fig. 4. Distribution of results obtained for each graph: a) solutions found for the *G*30 graph, b) statistics of solutions obtained for *G*30, c) solutions found for the *G*70 graph, d) statistics of solutions obtained for *G*70

6. Wnioski

W artykule przedstawiono metodę kosyntezy systemów specyfikowanych za pomocą grafu zadań i implementowanych w technologii SOC. Metoda oparta jest na rozwojowym programowaniu genetycznym. Wedle naszej wiedzy jest to pierwsze użycie algorytmu RPG do problemu kosyntezy. Podstawową różnicą w odniesieniu do innych podejść genetycznych jest to, że ewolucji nie podlegają rozwiązania, ale metoda konstruująca te rozwiązania. Okazało się, że metoda RPG jest również skuteczna dla problemu kosyntezy. Już we wstępnych eksperymentach uzyskano rozwiązania lepsze niż w dotychczas stosowanych metodach.

Dalsze prace będą miały na celu badanie innych metod reprezentacji problemu kosyntezy dla potrzeb RPG. Przede wszystkim planuje się opracowanie i ocenę bardziej zaawansowanych metod konstrukcji systemu, odpowiadających węzłom w drzewie genotypu.

Zostaną przebadane również alternatywne wersje operatorów genetycznych i wpływ parametrów algorytmu na jakość uzyskanego rozwiązania.

Literatura

[1] Wolf W., *High-Performance Embedded Computing: Architectures, Applications, and Methodologies*, Morgan Kaufman, St. Louis, USA 2006.

[2] Yen T.-Y., Wolf W., *Hardware-Software Co-synthesis of Distributed Embedded Systems*, Springer, Heidelberg, Germany 1997.

[3] Yen T.-Y., Wolf W.H., *Sensivity-Driven Co-Synthesis of Distributed Embedded Systems*, Proc. of the Int. Symposium on System Synthesis, Canes, France 1995, 4-9.

[4] Deniziak S., *Cost-efficient synthesis of multiprocessor heterogeneous systems*, Control and Cybernetics, Vol. 33, No. 2, Warszawa 2004, 341-355.

[5] Oh H., Ha S., *Hardware-Software Cosynthesis of Multi-Mode Multi-Task Embedded Systems with Real-Time Constrains*, Proc. of the Int. Workshop on Hardware/Software Codesign, Estes Park, USA 2002, 133-138.

[6] Dave B.P., Lakshminarayana G., Jha N.K., *COSYN: Hardware-Software Co-Synthesis of Embedded Systems*, Proc. of the Design Automation Conference, Anaheim, USA 1997, 703-708.

[7] Bianco L., Auguin M., Pegatoquet A., *A Path Analysis Based Partitioning for ime Constrained Embedded Systems*, Proc. of the Int. Workshop on Hardware/Software Codesign, Seattle, USA 1998, 85-89.

[8] Dick R.P., Jha N.K., *MOGAC: A Multiobjective Genetic Algorithm for the Co-Synthesis of Hardware-Software Embedded Systems*, IEEE Trans. on CAD, Vol. 17, No. 10, Piscataway, NJ, USA 1998.

[9] Purnaprajna M., Reformat M., Pedrycz W., *Genetic Algorithms for hardware-software partitioning and optimal resource allocation*, Journal of Systems Architecture, No. 53, North Holland 2007, 339-354.

[10] Grewal G.W., Wilson T.C., *An enhanced genetic algorithm for solving the high-level synthesis problems of scheduling, allocation and binding*, International Journal of Computational Intelligence and Applications 1 (2001), Paris, France, 91-110.

[11] Koza J.R., *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, MIT Press, Cambridge, MA, USA 1992.

[12] Koza J.R., Poli R., *Genetic Programming*, [in:] E. Burke, G. Kendal (eds.). *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*, Chapter 5, Springer, New York, USA 2005.

[13] Keller R.E., Banzhaf W., *The evolution of genetic code in genetic programming*, Proc. of the Genetic and Evolutionary Computation Conference, Orlando, USA 1999, 1077-1082.

[14] Koza J.R., Keane M.A., Streeter M.J., Mydlowec W., Yu J., Lanza G., *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*, Kluwer, Norwell, USA 2003.