AUTOMATYKA • 2007 • Tom 11 • Zeszyt 1–2

Dariusz Dereniowski*, Marek Kubale*

Model formalny dla problemu lokalizacji błędów w kodzie programu

1. Wprowadzenie

Testowanie oprogramowania jest ważnym i złożonym procesem technologicznym. Przez *test* w niniejszym artykule rozumiemy weryfikację poprawności fragmentu kodu programu. Weryfikacja ta polega na stwierdzeniu, czy dany fragment kodu zachowuje się zgodnie z jego specyfikacją. Będziemy rozważać model teoretyczny, zakładając, że program poddawany jest szeregowi testów.

Na jakość procesu testowania wpływa znacząco jakość wykonania pojedynczego testu. Istnieje wiele technik oceny jakości metody testowania. Pomiar jakości lub dokładności testu może polegać na wyznaczeniu stosunku liczby wykonanych fragmentów programu do liczby wszystkich fragmentów (*block coverage*), liczby różnych decyzji (dotyczy wyrażeń warunkowych) podjętych w trakcie wykonania programu do liczby wszystkich możliwych decyzji (*decision coverage*), czy analizie ścieżek wykonania programu, np. ścieżek łączących instrukcje prowadzące od definicji wartości zmiennej do jej użycia w kodzie [1]. W niniejszym artykule pomijamy aspekty wykonywania pojedynczego testu. Zakładamy jedynie, że jeśli badany fragment nie jest poprawny, to sytuacja ta zostanie wykryta.

Przez *blok* rozumiemy taki zbiór instrukcji (fragment programu), który może być osobno poddany testowi. Oznacza to, że do każdego bloku dołączony jest test, którego wykonanie pozwala stwierdzić, czy dany blok zawiera błąd. Naszym celem jest określenie, które testy oraz w jakiej kolejności należy wykonać, aby zlokalizować fragment programu zawierający błąd. Bloki nie muszą być rozłącznymi fragmentami programu, co zostanie dokładniej wyjaśnione w dalszej części pracy. Kryterium optymalizacyjnym jest liczba wykonanych testów, która odpowiada złożoności procesu testowania. Fakt, że w ogólności nie wszystkie testy muszą być wykonane w celu lokalizacji błędu, wynika z obserwacji, że jeśli po wykonaniu pewnego testu uzyskujemy informację, iż w przetestowanym fragmencie znajduje się błąd, to dalsze testowanie ograniczamy do tego fragmentu kodu. Z drugiej strony, jeśli jest on poprawny, to dalsze testowanie przeprowadzamy dla fragmentów kodu niezawierających się we fragmencie właśnie sprawdzonym.

* Katedra Algorytmów i Modelowania Systemów, Politechnika Gdańska

Możliwe jest reprezentowanie kodu programu w postaci grafu [1–3] i takie podejście jest stosowane w niniejszej pracy. Transformacja taka pozwala na precyzyjne sformułowanie problemu optymalizacyjnego: opis wejścia, wyjścia i kryteria optymalizacji.

W artykule przedstawiono: w rozdziale drugim wprowadzono podstawowe pojęcia z zakresu teorii grafów, w szczególności zdefiniowano jeden z modeli kolorowania grafu, który okazał się użyteczny podczas projektowania optymalnej strategii przeprowadzania testów. W rozdziale trzecim opisano teoretyczny model testowania oprogramowania. Podano precyzyjną definicję problemu, wyrażoną ostatecznie w terminologii teorii grafów. Rozdział czwarty zawiera opis procedury konstruującej algorytm wyszukiwania wraz z przykładem wykorzystania opisanej teorii. W ostatnim rozdziale podano pewne wnioski oraz sugestie dotyczące kierunków dalszych badań.

2. Definicje podstawowe

2.1. Wybrane pojęcia z zakresu teorii grafów

Graf prosty G to uporządkowana para dwóch zbiorów V, E. Zbiór V nazywany zbiorem *wierzchołków,* natomiast E jest zbiorem *krawędzi*, tzn. elementów postaci $\{x, y\}$, gdzie $x, y \in V$. *Ścieżka* w grafie G łącząca wierzchołki x,y to dowolny zbiór parami różnych wierzchołków $v_1, ..., v_k$ takich, że $v_1 = x$, $v_k = y$ oraz $\{v_i, v_{i+1}\} \in E$ dla każdego $i = 1, ..., k - 1$. Graf jest *drzewem,* jeśli istnieje dokładnie jedna ścieżka pomiędzy każdą parą wierzchołków. Drzewo T nazywamy *ukorzenionym*, jeśli posiada wyróżniony wierzchołek r stanowiący jego korzeń. Definiujemy wówczas *poziom* wierzchołka x w ukorzenionym drzewie T jako długość ścieżki łączącej x i r. Graf jest *spójny*, gdy istnieje ścieżka pomiędzy każdą parą wierzchołków. Tak więc drzewo z definicji jest grafem spójnym. Jeżeli graf nie jest spójny, to przez *składową spójności* rozumiemy dowolny z jego spójnych podgrafów, maksymalnych w sensie liczby wierzchołków.

2.2. Uporządkowane kolorowanie grafów

Niech będzie dany graf prosty G. Funkcję $c: V \to \{1, ..., k\}$ nazywamy *uporządkowanym k-pokolorowaniem* grafu G, jeśli każda ścieżka łącząca wierzchołki x,y spełniające $c(x) = c(y)$ zawiera wierzchołek z taki, że $c(z) > c(x)$. Liczby z przeciwdziedziny funkcji c nazywamy zwyczajowo *kolorami*. Najmniejszą liczbę k, dla której powyższa funkcja c istnieje, nazywamy *uporządkowanym indeksem chromatycznym* grafu G i oznaczamy symbolem $\chi_r'(G)$.

Powyższy problem kombinatoryczny jest obliczeniowo trudny w przypadku ogólnym [4]. Jednakże istnieje liniowy algorytm, który wyznacza optymalne uporządkowane pokolorowanie drzewa [5]. Drugi z faktów będzie przydatny w dalszej części pracy. Będziemy również korzystać z następującego faktu.

Fakt 1. *Jeśli c jest uporządkowanym k-pokolorowaniem grafu G, to istnieje w G dokładnie jedna krawędź o kolorze k. Jeśli G jest drzewem, to po usunięciu tej krawędzi G posiada dokładnie dwie składowe spójności.*

3. Model teoretyczny

Rozpoczniemy od pewnych założeń wstępnych. Zakładamy, że w kodzie programu występuje dokładnie jeden błąd. Co prawda taka sytuacja ma miejsce rzadko, lecz założenie to nie zmniejsza ogólności rozważań. Kosztem wykonania jednego dodatkowego testu, któremu poddamy cały program, potrafimy stwierdzić, czy program zawiera błędy. Jeśli występuje więcej niż jeden błąd, to proces testowania pozwoli wykryć jeden spośród błędnych bloków. Rozważania teoretyczne poprzemy przykładem kodu pokazanym na rysunku 1a. Program jest przedstawiony w postaci pseudokodu, gdzie literami A, ..., Q oznaczone są poszczególne bloki programu. Każda z procedur może zawierać dowolne inne instrukcje oprócz tych, które należą do bloków wewnętrznych. Rysunek 1b pokazuje przykładową postać procedury zawierającej bezpośrednio dwa bloki wewnętrzne – może być to przykład bloku B z rysunku 1a. Procedura wyznacza medianę spośród tych elementów tablicy T, które spełniają kryterium f. Funkcja Przeksztalc usuwa z tablicy X te elementy, które nie spełniają kryterium f i jest ona ujęta w nawiasy będące asercją. Asercja poprzedzająca blok określa warunki, jakie spełniają dane wejściowe dla danego bloku, natomiast asercja zamykająca blok definiuje warunki, które powinny spełniać dane po realizacji bloku. Podobnie jest w przypadku drugiego bloku procedury Mediana – pierwsza z asercji określa, iż elementy tablicy X, o rozmiarze *m*, spełniają warunek f i są parami różne. Asercja zamykająca blok stawia wymaganie, aby elementy tablicy X były posortowane.

a)

```
proc A:          proc D:
   call B;          call J;
   call C;          call K;
   call D;          call L;

proc B:          proc F:
   call E;          call M;
   call F;          call N;

proc C:          proc I:
   call G;          call O;
   call H;
   call I;       proc K:
                    call P;
                    call Q;
```

b)

```
proc Mediana(T: array[1..n] of integer,f: kryterium)
  linicjalizacja;
   X = T;

   {X[1],...,X[n]}
   Przeksztalc(X, f);
   {X[1],...,X[k]; f(X[i])=true, i=1,...,k}

   UsunDuplikaty(X);

   {X[1],...,X[m]; f(X[i])=true, i=1,...,m; X[i]<>X[j] dla i<>j}
   Sortuj(X);
   {X[1]<X[2]<...<X[m]}

   return X[1];
```

Rys. 1. Przykładowa struktura programu (a); przykład procedury o strukturze bloku B z rysunku 1a (b)

Ostatnia (podkreślona) instrukcja procedury na rysunku 1b nie jest poprawna, gdyż zwraca minimalny element tablicy X zamiast jej element środkowy. W dalszej części rozdziału będziemy kontynuować przykład pokazujący sposób lokalizacji tego błędu.

3.1. Wyszukiwanie elementów w zbiorach częściowo uporządkowanych

Niech będzie dany dowolny skończony zbiór V. Zakładamy, że dana jest relacja częściowego porządku $R \subseteq V^2$. Wprowadzamy następującą definicję [3]:

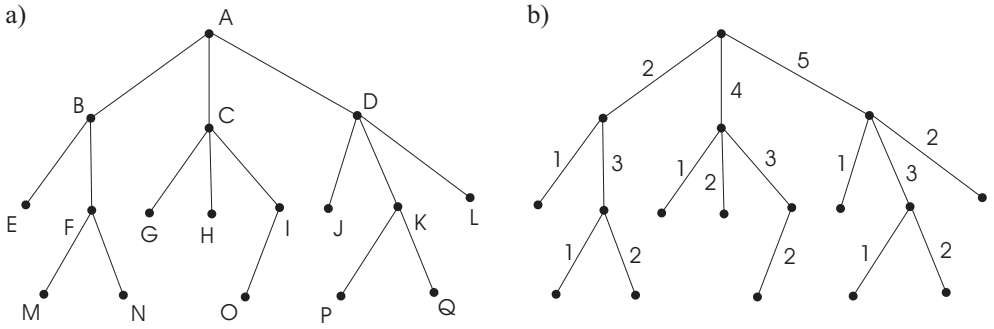
Definicja 1. *Algorytmem wyszukiwania A* elementu v w częściowym porządku (V, R) nazywamy uporządkowaną trójkę (x, A_1, A_2), gdzie $x \in V$, $|V| \neq 1$. A_1 jest algorytmem poszukiwania v w zbiorze $V_1 = \{y \in V: yRx\}$ i jest on wywoływany przez A w przypadku, gdy $v \in V_1$. A_2 jest algorytmem poszukiwania v w zbiorze $V_1 = V \backslash V_1$ i jest on wywoływany przez A w przypadku, gdy $v \notin V_1$. Jeśli $|V| = 1$, to algorytm kończy działanie, a poszukiwany element v został znaleziony.

Zakładamy, że element v należy do zbioru V. Definiujemy liczbę kroków s algorytmu A. Jeśli $|V| = 1$, to $s(A) = 0$ oraz dla $|V| > 1$ mamy

$$s(A) = \max(s(A_1), s(A_2)\} + 1 \tag{1}$$

3.2. Grafowy model programu

Dla dwóch bloków programowych u, v mówimy, że u *zawiera się* w v, gdy u jest zbiorem instrukcji, które bezpośrednio należą do v, lub u należy do bloku należącego do v. Definiujemy relację częściowego porządku R w zbiorze bloków programu w taki sposób, że uRv wtedy i tylko wtedy, gdy blok u należy do v. Podobnie jak w [3] zakładamy, że diagram Hassego relacji R jest ukorzenionym drzewem. Rysunek 2 pokazuje strukturę programu z rysunku 1a oraz uporządkowane pokolorowanie odpowiedniego drzewa.



Rys. 2. Diagram Hassego (a) dla programu z rysunku 1a; uporządkowane pokolorowanie drzewa odpowiadającego diagramowi Hassego (b)

Proces testowania programu przebiega następująco. Załóżmy, że dany jest blok u, w którym znajduje się błąd. Wybieramy dowolny blok kodu $u' \subseteq u$ i wykonując test dla u' uzyskujemy informację o tym, czy w u' znajduje się szukany błąd. Jeśli odpowiedź jest pozytywna, to poszukiwanie błędu ograniczamy do u', natomiast jeśli test nie stwierdził błędu w u', to poszukiwanie kontynuujemy pośród bloków należących do u, lecz nienależących do u'.

Schemat testowania pokazany na rysunku 2możemy zapisać w formalnej postaci: proces testowania programu jest algorytmem wyszukiwania A, który na wejściu pobiera u, a następnie dla ustalonego (dla tego algorytmu) u' wykonuje porównanie $v\mathrm{R}u'$. Jeśli powyższa relacja zachodzi, to realizowany jest algorytm A_1 z wejściem u', natomiast w przeciwnym wypadku wykonujemy A_2 dla $u \backslash u'$. Czas działania algorytmu dla początkowego programu r mierzymy liczbą wykonanych porównań i wynosi on $s(A)$.

4. Konstrukcja algorytmu wyszukiwania

W poprzednich rozdziałach wskazaliśmy równoważny opis problemu lokalizacji błędu w kodzie, w którym program został odwzorowany na graf. W niniejszym rozdziale podajemy opis konstrukcji algorytmu wyszukiwania, zakładając, że dane jest ukorzenione drzewo T, które opisuje strukturę programu. Kontynuujemy również przykład z rysunku 1.

4.1. Algorytm

Zakładamy, że c jest uporządkowanym pokolorowaniem jego krawędzi. Procedura wyznaczania optymalnego algorytmu przeszukiwania jest następująca:

procedure KonstrukcjaAlgorytmu
  Wejście: drzewo T, uporządkowane pokolorowanie c;
  Wyjście: algorytm wyszukiwania A;
begin
  if $|V(T)| = 1$ **then begin**
    v := wierzchołek drzewa T;
    A:= "return v";
  end else begin
    $\{u,v\}$:= krawędź tż. $c(\{u,v\}) \geq c(e)$ dla $e \in E(T)$; u jest ojcem v w T;
    A_1 := KonstrukcjaAlgorytmu($T[v]$, $c|_{T[v]}$);
    A_2 := KonstrukcjaAlgorytmu($T - T[v]$, $c|_{T-T[v]}$);
    $A := (v,A_1,A_2)$;
  end
  return A;
end

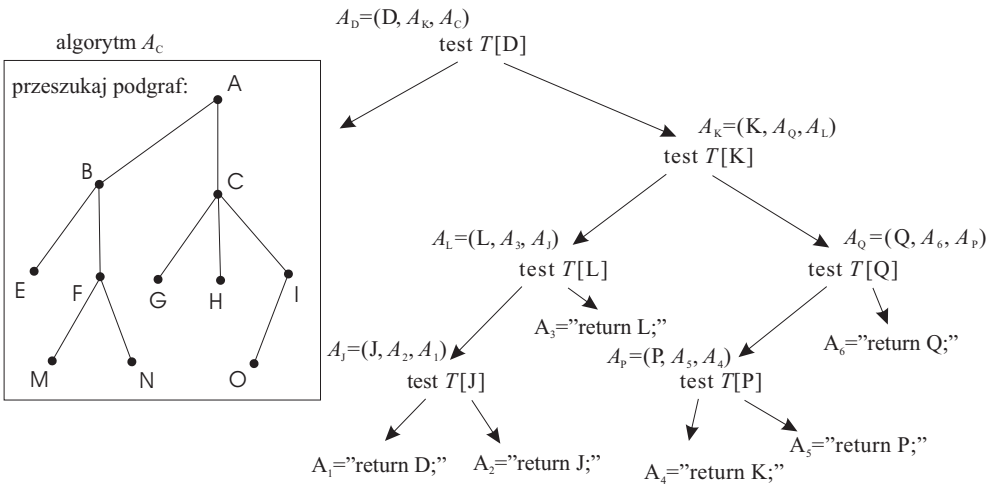
Czas działania powyższej procedury jest liniowy względem liczby wierzchołków drzewa T, a tym samym względem liczby bloków programu, dla których zostały utworzone testy. Fakt ten wynika stąd, iż uporządkowane pokolorowanie c można utworzyć w liniowym czasie [5], natomiast posiadając funkcję c, potrafimy w stałym czasie wyszukać krawędź o najwyższym kolorze.

Twierdzenie 1. *Procedura* KonstrukcjaAlgorytmu *wyznacza algorytm wyszukiwania A, który w czasie $w(A) = \chi_r'(T)$ znajduje węzeł drzewa T odpowiadający błędnemu fragmentowi programu.*

Dowód. Niech x będzie wierzchołkiem odpowiadającym fragmentowi kodu, który zawiera błąd. Dowodzimy indukcyjnie ze względu na liczbę wierzchołków drzewa T, że jeśli $x \in V(T)$, to x zostanie znaleziony przez algorytm wyszukiwania A. Jeśli $|V(T)| = 1$, to jedynym elementem $V(T)$ jest x, co oznacza, że twierdzenie jest prawdziwe, gdyż A nie wykona żadnego testu oraz $\chi_r'(T) = 0$. Jeśli $|V(T)| > 1$, to $A := (v, A_1, A_2)$. Zgodnie z definicją, A dokonuje testu fragmentu kodu odpowiadającemu podgrafowi $T[v]$, gdzie v jest zdefiniowany w ciele procedury KonstrukcjaAlgorytmu. W przypadku gdy test nie wykryje błędu, to $x \in V(T[v])$ i z założenia indukcyjnego wiemy, że x zostanie znaleziony w $\chi_r'(T[v])$ krokach. Jeśli w wyniku przeprowadzenia testu $T[v]$ algorytm A stwierdza, że $x \in V(T - T[v])$, to ponownie z założenia indukcyjnego wiemy, iż x zostanie odnaleziony w $\chi_r'(T - T[v])$ krokach. Zależności $\chi_r'(T) = \max\{\chi_r'(T[v]), \chi_r'(T - T[v])\}$ oraz (1) dowodzą tezy.

4.2. Przykład

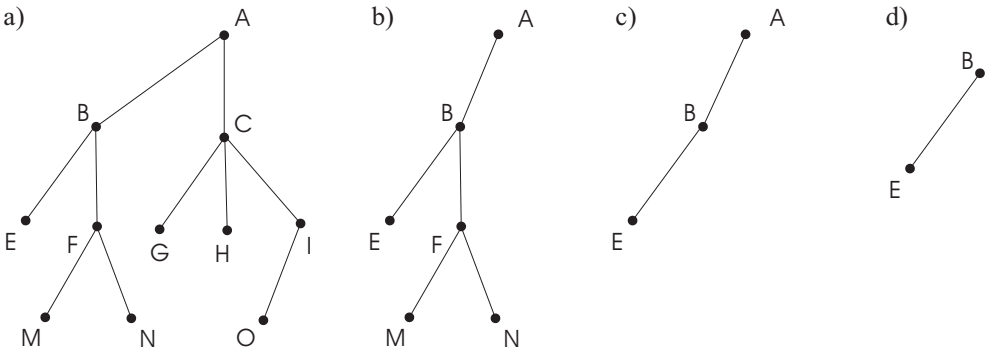
Poniżej kontynuujemy przykład konstrukcji algorytmu wyszukiwania dla kodu przedstawionego na rysunku 1. Odpowiednie drzewo T pokazano na rysunku 2a. Zakładamy, że optymalne uporządkowane pokolorowanie c drzewa T ma postać taką jak na rysunku 2b.



Rys. 3. Algorytm wyszukiwania A utworzony na podstawie uporządkowanego pokolorowania drzewa podanego na rysunku 2b

Na rysunku 3 przedstawiony został algorytm wyszukiwania, ograniczony do poddrzewa T[D]. Prawy potomek każdego węzła prowadzi do algorytmu wyszukiwania odpowiedniego poddrzewa, uruchamianego w sytuacji, gdy wynik testu jest pozytywny, tzn. stwierdzono, że dany fragment kodu zawiera błąd. Lewy potomek stanowi algorytm wyszukiwania, który jest realizowany, gdy w wyniku przeprowadzenia testu stwierdzono, że badany fragment kodu nie zawiera błędu. Zgodnie z definicją, algorytm wyszukiwania dla liścia pokazanego drzewa jest instrukcją „return", gdyż przeszukiwana domena jest ograniczona

do grafu zawierającego jeden wierzchołek. Algorytmy odpowiadające liściom zostały oznaczone symbolami A_1, ..., A_6 na rysunku 3. Dla pozostałych węzłów algorytm wyszukiwania jest uporządkowaną trójką, przy czym wywołanie rekurencyjne zależy od wyniku odpowiedniego testu.



Rys. 4. Kolejne etapy działania algorytmu poszukiwania błędu dla przykładu z rysunku 1
Objaśnienia w tekście

W przykładzie na rysunku 1b znajduje się błędna instrukcja „return X[1]". Uruchamiamy algorytm *A* dla grafu *T*. Realizacja instrukcji „test *T*[D]" pozwala stwierdzać, że *T*[D] nie zawiera błędu. Domena poszukiwania po wykonaniu pierwszego testu jest pokazana na rysunku 4a. Zgodnie z pokolorowaniem podanym na rysunku 1b kolejna decyzja podejmowana jest po realizacji testu podprogramu odpowiadającego poddrzewu *T*[C], co prowadzi do ograniczenia domeny przeszukiwania do postaci pokazanej na rysunku 4b, gdyż kod odpowiadający *T*[C] nie zawiera błędnych instrukcji. Kolor najwyższy w poddrzewie z rysunku 4b jest przydzielony krawędzi {B, F}, co oznacza, że kolejnym etapem algorytmu będzie wykonanie testu *T*[F]. Odpowiedź będzie negatywna, co ograniczy nasz podgraf do postaci z rysunku 4c. Kolejna faza działania algorytmu to „test *T*[B]", co ogranicza domenę do krawędzi {B, E} (rys. 4d). Ostatecznie algorytm *A* wykona jeszcze jeden test, mianowicie podprogramu odpowiadającemu *T*[E], co ograniczy obszar poszukiwania do wierzchołka B. Oznacza to, że błąd programu znajduje się wśród instrukcji procedury, z wyłączeniem tych, które znajdują się w obrębie asercji, czyli E oraz F.

5. Wnioski

Opisana powyżej metoda przedstawia model formalny, pozwalający na zaplanowanie serii testów prowadzących do lokalizacji błędu w kodzie programu. Domyślnym założeniem jest to, że wykonanie pojedynczego testu jest operacją czasochłonną. Oznacza to, że liniowy czas działania algorytmu projektującego strategię wyszukiwania sprawia, iż powyższa metoda jest bardzo wydajna. Jednym z dalszych kierunków pracy może być wpro-

wadzenie kryteriów pozwalających na oszacowanie czasów przeprowadzenia poszczególnych testów lub oszacowanie prawdopodobieństw wystąpienia błędów w poszczególnych wierzchołkach.

Literatura

[1] Horgan J.R., London S., Lyu M.R.: *Achieving software quality with testing coverage measures.* Computer, 27, 1994, 60
[2] Lipman M.J., Abrahams J.: *Minimum average cost testing for partially ordered components.* IEEE Transactions on Information Theory 1995, 41, 287
[3] Ben-Asher Y., Farchi E., Newman I.: *Optimal search in trees.* SIAM J. Comp., 6, 1999, 2090
[4] Lam T.W., Yue F.L.: *Edge ranking of graphs is hard.* Discrete Appl. Math., 85, 1998, 71
[5] Lam T.W., Yue F.L: *Optimal edge ranking of trees in linear time.* Proceedings of the ninth annual ACM-SIAM Symposium on Discrete Algorithms, 1998, 436