General and Professional Education
3/2015 pp. 61-70
ISSN 2084-1469

TESTOWANIE OPROGRAMOWANIA: UZASADNIENIE POTRZEBY DYDAKTYKI ORAZ TWORZENIA STRATEGII TESTOWANIA WYKORZYSTUJĄC WNIOSKI Z TEORII „NO FREE LUNCH”

SOFTWARE TESTING: RATIONALE FOR TEACHING AND CREATING TEST STRATEGIES USING THE CONCLUSIONS DRAWN FROM THE “NO FREE LUNCH” TEORY

Marek Żukowicz
Rzeszów University of Technology
Faculty of Electrical and Computer Engineering
Department of Computer and Control Engineering
ul. W. Pola 2, 35-959 Rzeszów
e-mail: bobmarek@o2.pl

Abstract: The paper presents a mathematical justification of the problem of applying software testing for smaller class of problems. The *No Free Lunch* theory is shown, conclusions are drawn from this theory. The logic of software testing division is shown. Test optimization strategies to the problems already divided into equivalence classes are described. In the last chapter the educational impact of the article are discussed. The importantance of testing educational software is highlighted.

Keywords: software testing, *No Free Lunch* theory, optimization strategies, equivalence classes.

Wprowadzenie

W końcu XX wieku (1997) zostało opublikowane nowe twierdzenie o „ niemożności” nazwane „No Free Lunch” theorem. Pokazuje ono, że nie istnieje najlepszy uniwersalny algorytm optymalizacyjny (dla wszystkich zadań). Niezależnie od miary jakości algorytmu optymalizacyjnego, dowolne dwa różne algorytmy optymalizacyjne zachowują się „średnio” tak samo dla wszystkich zadań optymalizacyjnych. Twierdzenie to natychmiast stało się przedmiotem sporów i dociekań, a jego rozumienie i interpretacja jest dalej tematem badań i dyskusji. Jednym ze sposobów i prób obejścia *No Free lunch* jest konstrukcja algorytmów optymalizacyjnych naśladujących naturę. Zaowocowało to powstaniem algorytmów ewolucyjnych, których operatory mutacji, krzyżowania i selekcji działają na populacjach (multizbiorach).
Wiele dziedzin naukowych postuluje teorię niemożności. Na przykład, w matematyce istnieje twierdzenie Godla: Każdy niesprzeczny rozstrzygalny system formalny pierwszego rzędu, zawierający w sobie aksjomaty Peana, musi być niezupełny. Oznacza to, że żaden system formalny pierwszego rzędu nigdy nie "pokryje" w całości zbioru wszystkich twierdzeń arytmetyki. Nie oznacza to, że zbiór wszystkich twierdzeń arytmetyki nie istnieje, a jedynie, że nie może on być wygenerowany przez żaden system formalny.
W ekonomii istnieje twierdzenie Arrowa (o niemożności) – sformułowane w 1951 roku przez ekonomistę Kennetha Arrowa. Wykazał on, że, po przyjęciu pewnych założeń co do oczekiwanej racjonalności decyzji grupowych, skonstruowanie satysfakcjonującej (spełniającej te założenia) metody podejmowania grupowych decyzji jest niemożliwe. Można zatem wnioskować, że bez przyjęcia pewnych założeń dotyczących problemu optymalizacji, teoretycznie niemożliwe jest znalezienie odpowiedniej strategii postępowania w celu rozwiązania problemu optymalizacyjnego.

Testowanie oprogramowania jest także problemem, który nie ma jednej dobrej strategii postępowania. W praktyce bardzo często brakuje pomysłów, jak dobrać dane za pomocą których można optymalnie przetestować aplikację, zanim trafi ona do potencjalnego użytkownika. Z jednej strony dane testowe powinny być zależne od kontekstu. Z drugiej strony aplikacja musi zadziałać prawidłowo dla każdych danych wejściowych. Z problemem testowania spotyka się sporo firm programistycznych. Ostatnimi czasy, stało się to bardzo ważnym tematem. Jest to problem, który próbują na bieżąco rozwiązywać „potężne umysły", stosując coraz to nowsze podejście. Faktem jest, że żaden człowiek lub automat nie może przetestować całej aplikacji od początku do końca, ponieważ czas testów wielokrotnie przekraczałby długość życia oprogramowania lub byłoby to nieopłacalne. Nawet testy automatyczne nie są w stanie zapewnić pełnego pokrycia wszystkich możliwych kombinacji funkcjonalności oraz danych w programie. Pojawia się pytanie: Jak zaprojektować testy, dobrać dane testowe i jak napisać przypadki testowe, aby optymalnie sprawdzić, czy wszystkie funkcje w aplikacji działają prawidłowo? Odpowiedzią na to pytanie jest podział testów na pewne klasy problemów i konstruowanie takich strategii, które dla danej klasy problemów będzie najlepsze.

Celem prezentowanej pracy jest przekonanie czytelnika, że ciągle warto szukać nowych strategii optymalizacji testowania, dzielić problemy testowania na mniejsze klasy problemów oraz pokazanie, że wnioski z twierdzenia *No Free Lunch* istnieją również w testowaniu. Praca oprócz pokazania metod optymalizacji testów przedstawia informacje na temat *edukacji testowania*. Prawie każdy rozdział jest w pewnym stopniu wiedzą uświadamiającą czytelnikowi, jakie istnieją rodzaje oraz strategie testowania. Poza nauką artykuł posiada wartości edukacyjne, które powinien poznać każdy tester. Autor pragnie również w formie edukacji zasygnalizować, że samo testowanie nie jest oczywistą i prostą pracą. Treść zawarta w artykule stanowi połączenie doświadczenia autora, który pracuje jako tester, oraz wiedzy zawartej w publikacjach wymienionych w bibliografii.

Podstawy teorii „No Free Lunch"

Do rozwiania jest pewien problem *P*. Dane jest odwzorowanie $y=f(x)$, gdzie $x \in X$ jest pewnym rozwiązaniem danego problemu, natomiast y jest skalarną wartością miary jakości rozwiązania x. Zbiory X oraz Y są skończone. Celem poszukiwań jest wybranie takiego rozwiązania x, dla którego miara jakości y jest najlepsza (np. minimalizacja albo maksymalizacja wartości pewnej funkcji).

Zbiór F wszystkich możliwych funkcji, które mierzą jakość rozwiąznia danego problemu jest równa $|F|=|Y|^{|X|}$. Wiedząc, że zbiory X, Y oraz F są skończone, ich elementy można zapisać w następujący sposób:

$X=\{x_0,\ldots, x_{|X|-1}\}$,

$Y=\{y_0,\ldots, y_{|Y|-1}\}$,

$F=\{f_0,\ldots,f_{|F|-1}\}$.

Mając dany taki sposób zapisu elementów zbiorów *X, Y* oraz *F* można zdefiniować pewną macierz zwaną *P*-macierzą. *P*–macierz (*ang. Problem matrix*) definiowana jest następująco [8]: numer wiersza jest równy numerowi elementu ze zbioru *X*, numer kolumny jest równy numerowi elementu ze zbioru *F*, natomiast element $a_{ij} = f_j(x_i)$. Jeśli $|X| = 3$ oraz $|Y|= 2$, to $|F| = 8$. Mamy wtedy macierz przedstawioną na rys. 1.

P-macierz jest uogólnieniem klasy macierzy, które są nazywane macierzami liczącymi, dlatego podana jest następująca definicja [8]:

Definicja 1. Niech dana będzie para całkowitych liczb dodatnich *O* oraz *1*. Macierz o *I*-wierszach oraz O^I kolumnach nazywamy macierzą obliczeniową *C* (*counting matrix C*).

Na przykładu, jeśli *I=3, 0=2*, macierz *C* (wyznaczona przez reprezentację binarną zbioru *0*) wygląda jak na rys. 2.

	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7
x_1	y_0	y_1	y_0	y_1	y_0	y_1	y_0	y_1
x_2	y_0	y_0	y_1	y_1	y_0	y_0	y_1	y_1
x_3	y_0	y_0	y_0	y_0	y_1	y_1	y_1	y_1

Rys. 1. Macierz 1 (*P*-macierz)

C=[0 1 0 0 0 1 0 0 0   1 0 1 1 0 0 0 1 1   0 1 1 1 1 1]

Rys. 2. Macierz 2 (Macierz obliczeniowa)

Pewną ciekawą własność zdefiniowanej macierz, można przedstawić za pomocą lematu:
Lemat 1. (Podstawowy Lemat obliczeniowy) [8]. Dla macierzy liczącej z definicji 1 spełnione są warunki:
a) Każda liczba ze zbioru *0* pojawia się O^{I-1} razy;
b) Podmacierz *C'* macierzy *C*, utworzona w taki sposób, że eliminujemy jeden wiersz *i* oraz każdą kolumnę taką, że $c_{ik} \neq c_{ij}$ jest macierzą liczącą.
Prawdziwość punktów a) i b) nie jest bardzo prosta do wykazania. Dowód punktu a) może być np. taki:
Jeśli $|O|=m$, $|I|=n$ to oczywiste jest, że w macierzy *C* każda wartość w konkretnym wierszu powtarza się $\frac{n^m}{n} = n^{m-1} = O^{I-1}$ razy.
Dowód punktu b) może być przedstawiony indukcyjnie w następujący sposób:
Bierzemy macierz liczącą *C* o trzech wierszach i ośmiu kolumnach, usuwamy wiersz oraz kolumny zgodnie z punktem b), czyli otrzymujemy macierz przedstawioną na rys. 3.
Podmacierz, która powstanie po usunięciu wierszy i kolumn ma postać jak na rys. 4.

C=[0 1 0 0 0 1 0 ~~0 0   1 0 1 1 0 0 0 1 1   0~~ 1 1 1 1 1]

Rys. 3. Macierz 3 (Macierz obliczeniowa przed redukcją kolumn)

C'=[0 1 0 0 0 1 1 1]

Rys. 4. Macierz 4 (Macierz obliczeniowa po zredukowaniu kolumn)

Macież *C'* jest również macierzą liczącą.
Schemat przeprowadzania dowodu daje następujący wniosek:
Wniosek 1. W macierzy liczącej sumy wszystkich elementów oraz średnie arytmetyczne dla każdego wiersza są równe.
Oznacza to iż: Każda strategia rozwiązująca wszystkie klasy problemów optymalizacyjnych jest średnio taka sama, jak inne strategie.
Łatwo zauważyć, że *P*–macierze są strukturalnie analogiczne jak macierz z lematu 1. Rozumując analogicznie jak w dowodzie lematu, wyciągamy kolejny wniosek:
Wniosek 2. *P*–macierze spełniają lemat liczący dla dowolnie dużego, ale skończonego zbioru *O* oraz dla dowolnie skończonego zbioru *I*.
Analizując *P*–macierz można przyjąć, że wiersze tej macierzy są wszystkimi możliwymi strategiami testowania, a kolumny możliwymi problemami optymalizacji testów. Uogólniając problem, można stwierdzić, że każda strategia może być poddawana ocenie jakości za pomocą pewnego odwzorowania. Ponieważ wszystkie średnie arytmetyczne wierszy *P*-macierzy są równe, prowadzi to do następującego wniosku, nazywanego teorią *No Free Lunch*: *Uśredniając wszystkie strategie oraz wszystkie problemy, mają one tą samą jakość. Inaczej, nie jest możliwe skonstruowanie takiego algorytmu (strategii), który jest uniwersalny i lepszy niż wszystkie inne algorytmy rozwiązujące problem optymalizacyjny.*
Najważniejszym faktem jest „równość wszystkich wierszy”. Natychmiast możemy wyciągnąć następujące wnioski w kontekście rozwiązania problemu testowania oprogramowania:
a) Warto dzielić testowanie na klasy oraz projektować nowe strategie optymalizacji testów;
b) Nie ma jednej optymalnej strategii testowania, która zoptymalizuje wszystkie rodzaje testów;
c) Mając strategię testowania, która nie nadaje się do pewnej klasy problemów, nie należy jej od razu odrzucać, ponieważ może zaistnieć problem, dla którego ta strategia rozwiązań będzie dobra lub nawet najlepsza;

d) Wprowadzenie pewnych założeń co do testowania ułatwia znalezienie optymalnego sposobu testów;
e) Niezbędna jest edukacja testowania, ale taka, która pozwoli testerowi być kreatywnym.
Analizując wyciągnięte wnioski nietrudno zauważyć, jak duża jest potrzeba wprowadzenia dydaktyki oraz metod testowania oprogramowania. Kolejne rozdziały stanowią pewne strategie oraz wprowadzają elementy edukacji testowania.

Edukacja testowania w Polsce

Samo wprowadzenie edukacji oraz szkoleń z zakresu testowania to krok do przodu w przypadku tego rodzaju problemów. Okazuje się, że w państwie polskim brakuje dydaktyki testowania. Wiele uczelni proponuje kierunki informatyczne, lecz nieliczne oferują elementy edukacji testowania. Uczelnia to często zamknięta forteca. Wiele mówi się o otwartości uczelni na pomysły i pomoc ze strony przedsiębiorców, ale niestety mało słyszy się o edukacji testowania na uczelniach wyższych. Zawodowi testerzy często muszą na własną rękę zwiększać swoją wiedzę oraz kwalifikacje. W Polsce np. portal *testerzy.pl* zajmuje się popularyzowaniem testowania i jego członkowie prowadzą liczne szkolenia z tego zakresu. Wiąże się to jednak z kosztami, odległością (dla niektórych osób) i nie każdy tester jest w stanie odbyć takie szkolenie. Okazuje się, że dobrym źródłem wiedzy dotyczącej testowania jest Internet. Znając język angielski, człowiek jest w stanie sporo dowiedzieć się na temat testowania. Ciekawym źródłem wiedzy są również publikacje naukowe związane z testowaniem. Można w nich znaleźć ciekawe strategie postępowania w niektórych przypadkach. Jednak niektóre z nich są nieco abstrakcyjne i niekoniecznie znajdą zastosowania w praktyce.

Logika podziału testów oprogramowania

Pierwszym krokiem w doborze optymalnej strategii testowania jest podział testów ze względu na ich przeznaczenie. Niżej przestawione są rodzaje testów ze względu na atrybuty, które należy przetestować:
Ze względu na funkcjonalność:
1) testy funkcjonalne (funkcje w programie):
- testy jednostkowe (testy podstawowych modułów, obiektów, klas),
- testy integracyjne małe (integracja klas, Modułów),
- testy integracyjne duże (integracja np. dwóch różnych systemów, typowa dla oprogramowania ERP)
- testy systemowe (testy systemu jako całości),
- testy akceptacyjne (potwierdzające, czy funkcje w oprogramowanie spełniają wymagania użytkownika);
2) testy niefunkcjonalne:
- użyteczność (rozumiana jako ilość czynności, które użytkownik musi wykonać, zanim wykona pewną funkcję),
- przenaszalność,
- bezpieczeństwo (ochrona danych w aplikacja webowych),
- wydajność (może być rozumiana jako natężenie użytkowników w przypadku aplikacji webowych)
- niezawodność (zdolność do pracy po wystąpieniu błędu).
Ze względu na dostęp testera do kodu:
1) testy biało-skrzynkowe (pełny dostęp do kodu),
2) testy czarno-skrzynkowe (brak dostępu do kodu),
3) testy szaro-skrzynkowe (częściowy dostęp do kodu lub środowiska do programowania).
Ze względu na rodzaj pokrycia testami:
1) testy pokrycia warunków logicznych,
2) testy pokrycia instrukcji.
Taki podział testów jak wyżej można utożsamiać z matematycznymi klasami równoważności, które zawierają dodatkowo mniejsze podklasy z pewnymi narzuconymi warunkami. Jeśli dwa przypadki testowe dotyczą tego samego problemu, to ich połączenie lub wzięcie części wspólnej również dotyczyło będzie tego samego problemu.

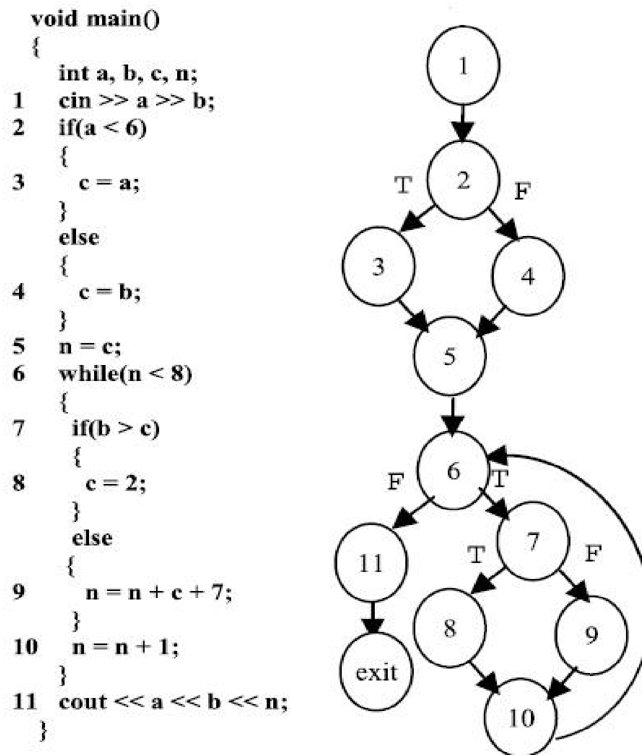
Priorytetyzacja przepadków testowych oraz zjawisko regresji

Podczas testowania bardzo często mówi się o testach regresji. Regresja to zjawisko utraty konkretnej funkcjonalności powstałe w nowej wersji programu i zwykle skutkujące komunikatem o błędzie, błędem logicznym lub brakiem działania. Do regresji dochodzi wskutek wprowadzania zmian w jakiejś części kodu programu. Skutkiem tych zmian jest błędne działanie innej funkcji programu, która w poprzednich wersjach działała prawidłowo. Aby wyłapać takie błędy, wprowadza się

testowanie regresyjne podczas stabilizacji wersji. Jest to rodzaj testów, które mają potwierdzić, że funkcje, który istniały w poprzednich wersjach nadal działają prawidłowo. Przy okazji testów regresji ujawnione zostają defekty, które powstały w wyniku modyfikacji kodu przez nieuwagę programistów.

Pomocny przy testach regresji bardzo często jest odział na testy biało-, czarno- oraz szaro-skrzynkowe. Taki podział wynika z tego, że w przypadku testów jednostkowych wymagany jest dostęp do kodu, a w przypadku testów funkcjonalnych taki dostęp nie jest potrzebny. W dużych systemach informatycznych jedna osoba nie jest w stanie jednocześnie pisać testów jednostkowych i wykonywać testów funkcjonalnych. Testy jednostkowe, czyli testy biało-skrzynkowe może pisać po prostu programista, programujący funkcje dla których należy te testy należy albo osoba, która pracuje jako programista testów. Przy testach biało-skrzynkowych mówi się o *analizie statycznej (testy bez wykonywania kodu).* Takimi testami mogą być np. przestrzeganie pewnych standardy kodowania, czyli *przeglądy kodu* lub sygnalizowanie przez środowisko do programowania o błędzie w składni kodu. Do testów szaro-skrzynkowych nie jest wymagana umiejętność programowania, dlatego takie testy mogą wykonywać testerzy, którzy nie programują, natomiast potrafią obsługiwać środowisko do programowania. Z testami czarnoskrzynkowymi tester ma do czynienia, jeśli po prostu nie ma dostępu do kodu i baz danych, sprawdza działanie samych funkcji w systemie. Testy biało-, czarno- oraz szaro-skrzynkowe można stosować zarówno w testach regresji jak i w testach nowych funkcjonalności.

Bardzo dobrym sposobem na zrozumienie zależności pomiędzy komponentami lub funkcjami systemu jest rozrysowanie głównych zależności za pomocą grafu np. zależność pomiędzy funkcjami, graf przepływu danych lub ścieżki i pętle. Zdarza się, że graf ma ciekawe własności (graf Hamiltona lub graf Eulera), które mogą być wykorzystane jeszcze w fazie projektowania systemu. Na rys. 5 przestawiono fragment kodu razem z grafem.



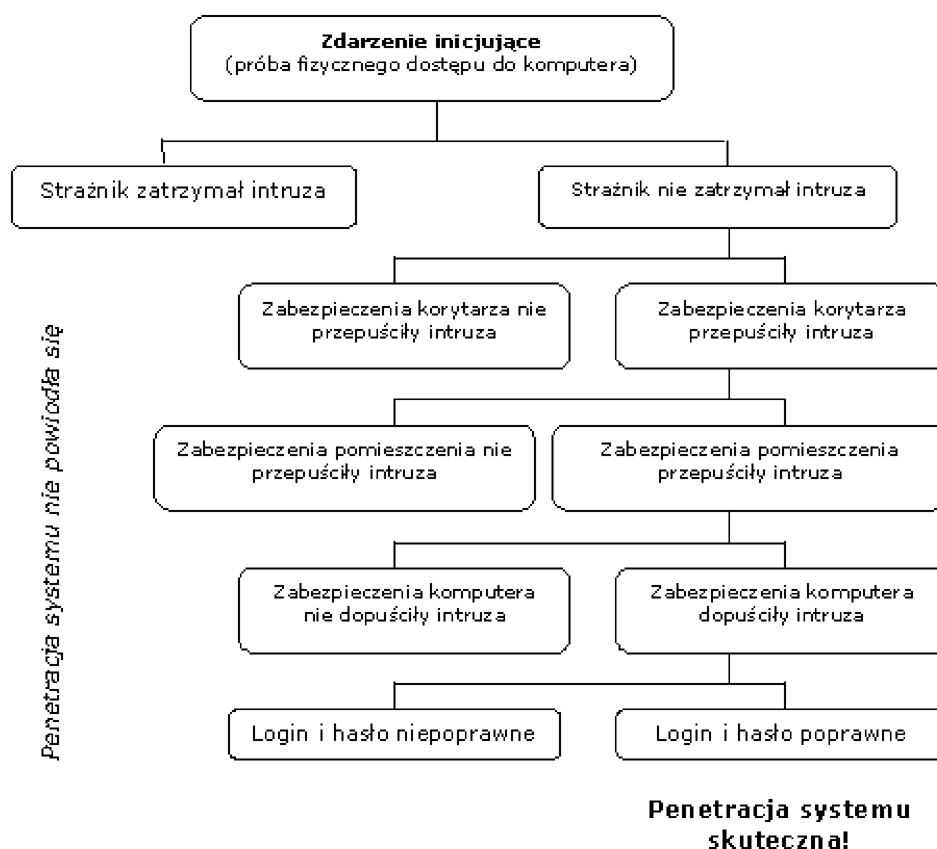
Rys. 5. Przykładowy kod testowanego programu wraz z grafem

Posiadanie takiego grafu ułatwia dobranie danych testowych, poprawne napisanie testów pokrycia instrukcji oraz testów pokrycia warunków logicznych.

W przypadku, gdy przeznaczone do testowania oprogramowanie posiada bardzo dużo funkcjonalności i duży poziom szczegółowości, przetestowanie takiego oprogramowania w wyznaczonym czasie może być utrudnione lub niemożliwe, ponieważ testerom po prostu zabraknie czasu. Ale wiedząc, że konkretny klient lub przyszły użytkownik danego oprogramowania nie użyje wszystkich ogólnodostępnych funkcjonalności, można takie funkcje oznaczyć jako *funkcje niskiego ryzyka.* Natomiast takie, które na pewno zostaną wykorzystane przez potencjalnego użytkownika, oznacza się jako *czynności (ścieżki) krytyczne*. Dzięki temu można pominąć testowanie funkcjonalności z niskim ryzykiem w początkowej fazie testów. Wówczas po przeprowadzeniu testów wszystkich ścieżek krytycznych i stwierdzeniu, że te testy „przechodzą poprawnie" oprogramowanie lub jego nowa wersja może trafić do klienta bez utraty jego zaufania. Natomiast po wykonaniu wszystkich testów i usunięciu błędów wersję zawsze można zaktualizować, ponieważ obecnie nie stanowi to problemu. Przy pomocy *priorytetyzacji przypadków testowych* pomocna jest również *analiza drzewa usszkodzeń*, opisana w dalszej części.

Analiza drzewa uszkodzeń

Analiza drzewa uszkodzeń (rys. 6) to metoda używana do analizy przyczyn uszkodzeń (defektów). Technika modeluje za pomocą grafu wizualnie związki logiczne pomiędzy awariami, błędami człowieka i zewnętrznymi zdarzeniami mogą powodować powstawanie specyficznych defektów. Zaletą tej metody jest możliwość dodania bramek logicznych do drzewa uszkodzeń, dzięki czemu przy pomocy specjalistów z konkretnej dziedziny możemy oszacować prawdopodobieństwo wystąpienia uszkodzenia.



Rys. 6. Przykładowe drzewo uszkodzeń przewidujące możliwość włamania się do systemu informatycznego

Analiza drzewa uszkodzeń może być wykorzystana nie tylko w informatyce, również w przemyśle. Na świecie istnieje wiele systemów, które kontrolują produkcję lub pewne procesy w fabrykach. Dla takich systemów również można wykonać drzewo uszkodzeń, co pozwoli na zaprojektowanie poprawnych testów integracji urządzeń oraz systemu informatycznego.

Metoda redukcji par danych wejściowych w testowaniu konfiguracji (pariwise testing)

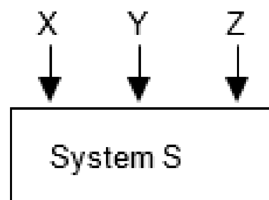
Testowanie par (*ang. pairwise testing*): Czarnoskrzynkowa technika projektowania przypadków testowych w której przypadki testowe są projektowane tak, aby wykonać wszystkie możliwe dyskretne kombinacje każdej pary parametrów wejściowych. Celem takiego projektowania testów jest potwierdzenie, że różne kombinacje zmiennych są prawidłowo obsługiwane przez system lub wykrycie defektów, które są wynikiem kombinacji par zmiennych w konfiguracji lub par zmiennych wejściowych. W wielu przypadkach zdarza się, że wystąpienie awarii jest związane z kombinacją dwóch parametrów, a nie ich większej ilości. Jest tak dlatego, że prawdopodobieństwo wystąpienia błędu spowodowanego kombinacją dwóch par jest o wiele większe, niż prawdopodobieństwo wystąpienia błędu spowodowanego kombinacją trzech parametrów lub funkcji. Zastosowanie takiej metody przedstawia dalej przedstawiony przykład.

Do przetestowania jest system *S*, który ma trzy wejścia jak na rys. 7.

Zbiór $D=X\times Y\times Z$ jest iloczynem kartezjańskim danych wejściowych, gdzie $D(X)=\{1, 2\}$, $D(Y)=\{Q, R\}$, $D(Z)=\{5, 6\}$.

Liczba przypadków testowych potrzebnych do przetestowania wszystkich kombinacji wynosi $2\times2\times2=8$.

Stosując metodę *pairwise testing* można ograniczyć liczbę przypadków do 4, tak jak w tabeli 1.



Rys. 7. Schemat testowanego systemu z trzema wejściami

Tabela 1. Przypadki testowe wygenerowane metodą *pairwise testing* dla systemu *S*

ID testu	We X	We Y	We Z
1	1	Q	5
2	1	R	6
3	2	Q	6
4	2	R	5

Upraszcza to znacznie testowanie i skraca czas testów. Metoda doskonale sprawdza się przy testach konfiguracji, ale w przypadku, gdy parametry są od siebie zależne.

Zastosowanie algorytmów ewolucyjnych w procesie testowania

Wraz z końcem XX wieku zaczęły pojawiać się artykuły naukowe, dotyczące zastosowania algorytmów genetycznych w procesie

testowania. W testach regresji mówi się o zjawisku nazwanym *paradoks pestycydów*. Paradoks pestycydów to taka negatywna cecha testów, w wyniku której zaprojektowane testy regresji nie znajdują usterek, czego powodem jest niezmienność danych testowych (wartości, które wprowadzane są do pół lub funkcji). Dane testowe powinny być zmieniane co jakiś czas.

Ważną zaletą algorytmów genetycznych jest to, że za ich pomocą można w sposób *losowy* wygenerować dane wejściowe oraz wprowadzić funkcję dopasowania dla każdej takiej danej. W teorii ewolucji istnieją takie zjawiska jak krzyżowanie (jednopunktowe lub wielopunktowe) oraz mutacja osobników w danej populacji. Te dwa zjawiska pozwalają na tworzenie nowych danych testowych z danej populacji w taki sposób, żeby uzyskać „dobre" dopasowanie *nowoutworzonych danych wejściowych* do konkretnej funkcji. Ogólna zasada działania algorytmu genetycznego jest niezmienna. Natomiast takie procesy jak krzyżowanie, mutacja oraz zaprojektowanie *funkcji dopasowania* są zależne od danych wejściowych i sposobu w jaki te dane powinny być generowane. Zaprojektowanie funkcji dopasowania jest łatwiejsze, jeśli twórca ma do dyspozycji Graf Przepływu Danych *(ang. Control Flow Graph*). Zwykle krzyżowane są chromosomy o takiej samej długości, a mutacja dotyczy takich genów, które przedstawić można za pomocą liczb. Natomiast Autor pozycji [5] przedstawia krzyżowanie, które dopuszcza różną długość chromosomów. Z kolei algorytmy ewolucyjne można zastosować w testach funkcjonalnych czarno-skrzynkowych, testach biało-skrzynkowych, testach regresji, testach pokrycia węzłów oraz ścieżek w grafie przepływu danych.

Testowanie przeglądowe oraz „intuicyjne"

Niekiedy w firmach programistycznych ma miejsce taka sytuacje, że oprogramowanie jest już napisane, jednak nie testerzy lub kierownik testów nie mają specyfikacji oprogramowania lub opisu funkcjonalności dostępnych w oprogramowaniu. Brak specyfikacji funkcjonalnej może wynikać np. z tego, że projekt jest „nieduży" (zawiera mało funkcji), a przez to twórcy wyszli z założenia, że nie ma potrzeby pisania specyfikacji funkcjonalnej oprogramowania. Często ma miejsce również taka sytuacja, że nie wszystko jest dobrze opisane, ale ktoś musi przeprowadzić lub zaprojektować testy. W takim przypadku dobrze sprawdzają się testy przeglądowe, które polegają na tym, że testerzy sprawdzają po kolei, co stanie się po kliknięciu przycisków lub sprawdzają jak zadziała pewna funkcjonalność po wpisaniu dowolnych wartości. Takie podejście jest związane z intuicją osoby, która testuje dane oprogramowanie.

Testy intuicyjne to testy, które wymagają pewnego rodzaju doświadczenia od testera. Osoba, która przetestowała sporo systemów czy projektów dzięki doświadczeniu potrafi tak przeprowadzić testy, żeby przekonać się czy pewne czynności nie wygenerują awarii systemu.

Analiza wartości brzegowych i podział danych testowych na klasy równoważności

Klasa równoważności (również w testowaniu) jest to zbiór danych o podobnym sposobie przetwarzania w oprogramowaniu dla konkretnej funkcjonalności, używanych do przeprowadzenia testu. Wykonanie testu z użyciem kilku elementów zbioru, powoduje uznanie całej klasy za poprawną i zwalnia osobę testującą od testowania wszystkich elementów w np. 100-elementowym zbiorze.

Rozwinięciem testów z użyciem klas równoważności jest *testowanie wartości brzegowych*. Wartość brzegowa to wartość znajdująca się wewnątrz, pomiędzy lub tuż przy granicy danej klasy równoważności. Analizę wartości brzegowych przedstawia następujący przykład:

Testowany system: czajnik z elektronicznym czujnikiem temperatury (zakres pomiaru: od 0,0^{o}C do 120,0^{o}C, z dokładnością do 0,2^{o}C), emitujący dźwięk w przypadku przekroczenia temperatury 100,0^{o}C. Poniżej przedstawiono procedurę wyznaczania wartości brzegowych jako danych testowych

Krok 1. Dla każdej danej wejściowej wyznaczono klasy równoważności, poprawne i niepoprawne:

1) [0,0; 100,0] (Poprawna),
2) [100,1; 120,0] (Poprawna),
3) Klasa < 0,0 (Niepoprawna)
4) Klasa > 120,0 (Niepoprawna).

Krok 2. Dla każdej granicy klasy równoważności utworzono dane testowe umieszczone możliwie najbliżej tej granicy, po obu jej stronach.

Wartości graniczne:
1) Spoza zakresu: -0,1; 120,1;
2) W obrębie zakresu: [0.0; 120,0] oraz dodatkowo:
- Próg emisji dźwięku (1): 100,1 (emisja dźwięku),
- Próg emisji dźwięku (2): 100,0 (brak emisji dźwięku).

Automatyzacja testów

Dwie grupy testów, które warto (ale nie zawsze) automatyzować to: testy wydajnościowe i testy regresji. Automatyzacja testów wydajnościowych jest oczywista, ponieważ prościej i taniej jest stworzyć automaty symulujące pracę dużej ilości użytkowników, niż zebrać i zarządzić odpowiednio dużą grupą ludzi do wykonania testu. Natomiast automatyzacja testów regresji pozwala na uruchamianie całego zestawu testów, weryfikującego poprawność działania witalnych funkcji biznesowych oprogramowania względem nowo wprowadzonych zmian. Warto również automatyzować wszystkie testy, które są powtarzalne np. jednostkowe, funkcjonalne. Automatyzacja testów nie oznacza jednak, że przetestowane zostanie wszystko, co tego wymaga. Twórcy oprogramowania do testów automatycznych nie są w stanie nadążyć za ciągle zmieniającymi się technologiami. Nie jest możliwe, żeby zautomatyzować 100 % testów. W praktyce, jeśli uda się wprowadzić ok. 20 % testów automatycznych, to jest to dobry wynik. Przed rozpoczęciem automatyzacji należy przeprowadzić analizę kosztów (program do testów automatycznych, sprzęt, ludzie) i sprawdzić, czy koszt automatycznych testów nie przewyższy potencjalnego zysku płynącego z automatyzacji.

Podsumowanie

Prezentowana praca ma charakter edukacyjny. Celem jej było zaznaczenie, że testowanie oprogramowania jest bardzo obszernym oraz niełatwym do zastosowania w praktyce problemem. Dodatkowo istnieje matematyczne uzasadnienie potrzeby szukania rozwiązań dla problemów optymalizacji, a okazuje się, że testowanie jest takim właśnie problemem. Można wyróżnić wiele rodzajów testów, co widać w rozdziale *Logika podziału testów oprogramowania.* W każdym rodzaju testów oprogramowania z pewnością można znaleźć pewną klasę lub schemat dla których istnieje lepsza jakościowo strategia od tej, która była wcześniej przyjęta.
Edukacja oraz popularyzowanie testowania jest bardzo ważnym zagadnieniem, ponieważ testowanie idzie zawsze w parze z programowaniem. Dzieje się tak dlatego, że informatyka istnieje niemal w każdej dziedzinie życia oraz przemysłu, dlatego warto pisać o testowaniu lub nawet podjąć próby rozszerzenia wiedzy związanej z testowaniem oprogramowania nie tylko od stronnych technicznej, ale również od strony optymalizacji oraz doboru strategii do rodzaju testu.

Bibliografia

1. Alavi R., Lotfi S., *The New Approach for Software Testing Using a Genetic Algorithm Based on Clustering Initial Test Instances,* 2011 International Conference on Computer and SoftwareModeling.
2. Bieniawski S., Wolpert D. H., Rajnarayan D., *Probability collectives in optimization,* Encyclopedia of Stastistics, 2012.
3. Chen, Y., Zhong Y., *Automatic path-oriented test data generation using a multipopulation genetic algorithm,* the 4th International Conference on Natural Computation, 2008.
4. Coppit D., Dugan J.B., Sullivan K.J., *Developing a Low-Cost High-Quality Software Toolfor Dynamic Fault-Tree Analysis,* IEEE TRANSACTIONS ON RELIABILITY, VOL. 49, NO. 1, MARCH 2000.
5. Ghiduk A. S., *Automatic generation of basis test paths using variable length genetic algorithm,* Elsevier B.V. 2014.
6. Ghiduk A.S, *Automatic generation of object-oriented tests with a multistage based genetic algorithm*, J. Comput., 2010.
7. Helmer G, Wong J., Slagell M., Honavar V., Miller L., Lutz., R., *A Software Fault Tree Approach to Requirements Analysis of an Intrusion Detection System,* 2002 Springer-Verlag London Limited.

8. Ho Y., Pepyne D. L., *Simple Explantation of No Free Lunch Theorem of Optimization,* IEEE, Conference on Decision and Control, Orlando, Florida USA, 2001,IPCSIT vol.14 (2011) © (2011) IACSIT Press, Singapore 225.
9. Pargas R. P., Harrold M. J., Peck R. R., *Test-Data Generation Using Genetic Algorithms,* Journal of Software Testing, Verification and Reliability, 1999.
10. Tai K., Lei Y., *A Test Generation Strategy For Pair Wise Testing,* IEEE Transactions on Software Enginiering, January 2002.