Analiza testów maksymalizujących pokrycia i wykrywalność mutacji programów

Anna Derezińska
Instytut Informatyki, Politechnika Warszawska,
ul. Nowowiejska15/19, 00-665 Warszawa
adr@ii.pw.edu.pl

Streszczenie

Praca dotyczy analizy jakości testów programów. Porównywano testy pod względem pokryć programów (pokryć bloków, rozejść decyzyjnych, użycia zmiennych w obliczeniach, i w wyrażeniach warunkowych) oraz wykrywalności wprowadzonych zmian – mutacji programów. Przedstawiono charakterystykę wykrywalności różnych rodzajów mutacji.

1. Wstęp

Zapewnienie wysokiego stopnia niezawodności działania programów wymaga zastosowania efektywnych strategii jego testowania. Testowanie programu jest dla dużej liczby testów procesem czasochłonnym. Ponadto po każdej modyfikacji programu pożądane jest ponowne przeprowadzenie testów pozwalające sprawdzić nie tylko poprawność wprowadzonych zmian, ale również podstawowe, uprzednio przetestowane działanie programu, które mogło ulec zakłóceniu. Z tego powodu istotną sprawą jest możliwość generacji jak najmniejszych zbiorów testów, które jednak możliwie efektywnie będą testować zadany program. Jednymi ze sposobów oceny jakości testów są: określenie stopnia pokrycia programu przez test [2,3] oraz określenie wykrywalności zmutowanych programów [4]. Stopień pokrycia programu wyznacza, jaka część kodu (np. instrukcji, danych – w zależności od typu pokrycia) była wykonywana w trakcie realizacji danego testu. Przyjmujemy, że dobry test zapewnia duże pokrycie programu, lub pokrycie tych części programu które nie mogą być

pokryte przez inne testy. Idea mutacji polega na modyfikacji kodu w określony sposób i porównaniu działania zmutowanego programu (mutanta) z działaniem oryginalnego programu. Wykonując testowanie programu spodziewamy się wykryć wprowadzone błędy. Test jest tym lepszy im więcej mutantów potrafi wykryć.

W niniejszej pracy przedstawiono wyniki analizy testów związane wyborem zbioru testów, które zapewniają zarówno wysoki stopień pokrycia programu jak i wysoką wykrywalność mutacji. Przeprowadzono badania 6 programów implementujących różne zagadnienia z diagnostyki systemowej [1 i referencje]. Programy (1-4) były napisane w języku C++, program (5) w języku C, a (6) w Perlu. Programy te zostały uruchomione i wstępnie przetestowane (strukturalnie i funkcjonalnie). Dalsze badanie tych programów było realizowane przez inną grupę osób, bez konsultacji z autorami programów.

Dla każdego z programów wyznaczono podstawowe miary charakteryzujące jego złożoność (tab. 1). Jako moduł programu rozumiana jest procedura, funkcja, metoda klasy (w tym konstruktor i destruktor klasy). Główne różnice pomiędzy programami związane są z zastosowaniem programowania obiektowego. Moduły implementowane w języku C (funkcje) są średnio biorąc bardziej złożone - dłuższe, z większa liczbą wyjść, niż moduły (metody klas) wykorzystywane w języku C++ oraz moduły w Perlu.

Tab. 1 Metryki złożoności programów

Metryki		Programy					
		1	**2**	**3**	**4**	**5**	**6**
Liczba linii kodu (wszystkich)		1866	1854	1595	4321	449	770
Liczba linii kodu (bez komentarzy i pustych linii), % od wszystkich linii		1427 76.5%	1347 72.6%	1344 84.3%	3135 72.6%	309 68.8%	623 80.9%
Liczba wykonywalnych instrukcji		888	1844	1709	2078	386	368
Liczba modułów		56	117	63	131	9	25
Średnia liczba	instrukcji w module	15.86	18.7	27.13	15.86	42.9	14.72
	punktów wyjścia z modułu	1.14	1.7	1.41	1.43	2.78	1.36
	wywołań modułów	1.4	7	3	3.4	1.1	3.4
Liczba zmiennych	globalnych	14	9	1	1	7	14
	lokalnych	123	67	160	345	28	49

Ponadto każdy z programów (1-6) badano według następującego schematu:

- Testowanie funkcjonalne
  - Na podstawie otrzymanej specyfikacji i dokumentacji programu został opracowany zbiór podstawowych testów funkcjonalnych.
  - Określono jakie jest poprawne zachowanie się programu (w tym jakie są poprawne zbiory danych wyjściowych) dla poszczególnych podstawowych testów funkcjonalnych.
  - Przetestowano program z użyciem tych testów.

- Badanie pokryć
  - Dla zbioru podstawowych testów funkcjonalnych zmierzono pokrycia dla czterech typów pokryć: bloków, decyzji, c-use, p-use [2].
  - Wyznaczono fragmenty kodu nie pokryte przez testy podstawowe.
  - Na podstawie analizy kodu opracowano zbiór dodatkowych testów, w celu zwiększenia stopnia każdego typu pokrycia programu.
  - Wybrano minimalny podzbiór testów, dla którego pokrycia wszystkich czterech typów są takie same jak pokrycia uzyskane dla wszystkich testów podstawowych i testów dodatkowych dla pokryć.
- Badanie mutacji
  - Wybrano kilka rodzajów mutacji i wyznaczono zbiór mutanów programu.
  - Sprawdzono, czy podstawowe testy funkcjonalne wykrywają te mutanty.
  - Po analizie kodu programu opracowano zbiór dodatkowych testów, w celu zwiększenia wykrywalności mutantów.
  - Wybrano minimalny podzbiór testów, dla którego wykryto wszystkie mutanty, które udało się wykryć za pomocą wszystkich testów podstawowych i testów dodatkowych dla mutacji.
- Porównanie dodatkowych testów
  - Zmierzono pokrycia programu dla dodatkowych testów zwiększających wykrywalność zadanych mutantów.
  - Określono wykrywalność mutacji za pomocą dodatkowych testów zwiększających pokrycia.
  - Wyznaczono ostateczny minimalny podzbiór testów maksymalizujący i pokrycia programów i wykrywalność mutantów. Pozwala on uzyskać takie same pokrycia wszystkich czterech typów oraz wykryć wszystkie takie same mutanty, jak zbiór wszystkich testów dla danego programu.

Przygotowanie jednego testu oznaczało przygotowanie jednego zestawu danych wejściowych, określających parametry diagnozowanego systemu, który był badany przez program (1-6), np. liczbę węzłów, topologię połączeń komunikacyjnych, graf diagnostyczny, rozkład uszkodzeń węzłów w systemie, scenariusz uszkodzeń i napraw węzłów w czasie, parametry symulacji (rzędu 20÷40 wartości całkowitych, dla zakresu od kilku wartości do około 650). Realizacja jednego testu wymagała jednego przebiegu wykonania programu.

2. Pokrycia programów

Dla każdego z programów wyznaczane były 4 typy pokryć [1,2]:

- pokrycie bloków,
- pokrycie rozejść decyzyjnych,

- pokrycie użycia zmiennych w obliczeniach (c-use),
- pokrycie użycia zmiennych w wyrażeniach warunkowych (p-use).

Pomiar pokryć był realizowany przez półautomatyczną inspekcję kodu, z ewentualnym wykorzystaniem grafu odwzorowującego wykonanie programu. Dla przykładu, grafy utworzone dla wybranych 12 modułów programu (3) składały się z następującej liczby węzłów i krawędzi:

węzły	48	45	45	33	17	16	16	13	9	3	3	1
krawędzie	93	76	70	49	24	32	30	24	14	6	6	0

Każdy węzeł grafu odpowiada pojedynczemu blokowi w sensie pokryć. W innych programach wykorzystano makra ustawiające w tablicy znaczniki użycia kodu (2), obliczano liczniki wykonań linii dla sformatowanego kodu (4) lub wstawiano automatycznie za pomocą skryptu odwołania do procedur (6) [2]. Dla programów wieloprocesowych (2,6) podawane będą pokrycia optymistyczne, tzn. uznajemy element programu za pokryty, jeśli został wykonany w przynajmniej jednym procesie.

W tab. 2 zebrano wartości pokryć, uzyskane dla zbioru podstawowych testów funkcjonalnych oraz po zastosowaniu dodatkowych testów. Podano też liczby wykonywanych testów podstawowych i dodatkowych. Przyjmujemy, że liczba testów dla danego programu jest przybliżoną miarą złożoności procesu testowania. Można zauważyć, że około połowy rozpatrywanych testów wystarczy, żeby osiągnąć takie same stopnie wszystkich czterech typów pokryć jakie uzyskano dla wszystkich testów.

Niepokryte bloki i rozejścia decyzyjne były związane tylko w niewielkim stopniu (1-2.5%) z bezpośrednią realizacją zadań w poszczególnych programach. W większości niepokryte bloki dotyczą kodu służącego do obsługi sytuacji wyjątkowych lub nieużywanych metod ogólnych, pomocniczych klas należących do programu (np. klasy obsługi łańcuchów, stosu, grafu), i dlatego są bardzo trudne lub niemożliwe do pokrycia przez testy funkcjonalne. Dla przykładu bloki nie pokryte w programie (1) należą do następujących kategorii kodu programu:

6.1% - kod niedostępny przez żadne odwołanie z programu,
1.6% kod niewykonywany przy prawidłowym działaniu programu,
1.2% destruktory klas,
2.4% realizacja funkcji

(liczby określają procent bloków danej kategorii względem ogólnej liczby bloków). Niepokryty pozostał na ogół kod związany z obsługą błędów wynikających z braku zasobów w systemie operacyjnym (brak pamięci, niemożność utworzenia procesu potomnego), oraz ustawieniem zmiennych systemowych (np. zmiennej PATH dla programu (6)). Również niepokryty pozostał kod obsługujący wewnętrzne błędy programu, - np. kod asercji, których warunki nigdy nie powinny być spełnione; kod zapisany w gałęziach **default** dla instrukcji **switch** (3). Niepełne pokrycie może też wynikać z dużej liczby możliwości podania błędnych danych, powodujących natychmiastowy koniec programu. Dla każdego

takiego przypadku potrzebny jest osobny test. Na przykład, żeby pokryć niepokryte bloki i rozejścia decyzyjne związane z błędami kończącymi działanie programu (3), należałoby wykonać 32 dodatkowe testy. Warto zauważyć, że niepokryty kod służący do obsługi błędów, oraz kod niedostępny przez wywołania z programu, mógł być pokryty w zmutowanych programach.

Tab. 2 Pokrycia programów

		Programy					
		1	**2**	**3**	**4**	**5**	**6**
Liczba podstawowych testów		11	10	6	10	6	7
Pokrycia na postawie podstawowych testów [w %]	bloków	71,3	85	87.9	95,3	70.8	82
	rozejść warunkowych	88.0	100	84.2	95,3	74.5	96
	c-use	100	84	100[1)]	76	48.6	98
	p-use	95.0	78	100[1)]	67	66.7	83
Liczba dodatkowych testów zwiększających pokrycia		3	2	2	5	1	0
W sumie liczba testów dla pokryć		14	12	8	15	7	7
Minimalna liczba testów dla maksyma-lizacji pokryć	bloków	5	-	-	2	2	3
	rozejść warunkowych	5	-	-	2	2	1
	c-use	2	-	-	7	2	-
	p-use	2	-	-	3	2	-
Minimalna liczba testów dla wszystkich pokryć		6	6	5	8	3	3
Liczba testów dodatk. z mutacji poprawiających pokrycie c-use		0	0	0	1	2	0
Liczba testów (w sumie)		14	12	8	16	9	7
Minimalna liczba testów maksyma-lizujących pokrycia (po mutacjach)		6	6	5	6	4	3
Maksymalne uzyskane pokrycie [w %]	bloków	88,7	-	93.1	95.3 100[2)]	89.6	82.0
	rozejść warunkowych	92.8	-	90.6	95.3 100[2)]	91.5	96
	c-use	100	-	100	92 100[3)]	75.7 81.1[3)]	98
	p-use	95.0	-	100	78	93.9	83

Legenda: [1)] - tylko wybrane zmienne
[2)] - po poprawieniu wykrytego błędu w programie
[3)] – po wykorzystaniu testów dodatkowych wykrywających mutacje

W tabeli 3 przedstawiono ocenę efektywności testów dla poszczególnych programów i typów pokryć, określoną na podstawie wszystkich testów, dla których były wyznaczane pokrycia. Dla każdego typu pokrycia w wierszu min podano pokrycie testu, dla którego wartość tego typu była najmniejsza. Analogicznie w wierszu max podano wartość pokrycia otrzymaną dla jednego testu, dla którego pokrycie tego typu było największe. Podano też średnią arytmetyczną pokryć danego typu uzyskanych od poszczególnych testów. W

większości przypadków widoczna jest duża rozpiętość wyników pokryć uzyskiwanych dla poszczególnych testów (różnice pomiędzy min i max). Z drugiej strony wykonując nawet jeden test można osiągnąć stosunkowo duże pokrycia. Oznacza to, że opłaca się tworząc testy funkcjonalne wybierać je również pod względem spodziewanego pokrycia programu.

Tab. 3 Efektywność testów pokrywających programy

Pokrycie przez pojedyncze testy [w %]		Programy					
		1	**2**	**3**	**4**	**5**	**6**
bloków	min	30.36	70.0	10.1	-	27.1	49.0
	średnio	57.1	77.5	56.6	-	48.3	63.6
	max	67.6	85.0	76.2	-	85.4	75.0
rozejść warunkowych	min	33.7	42	7.9	-	34.0	62.0
	średnio	63.5	69.8	49.1	-	53.9	78.1
	max	78.3	100	73.0	-	85.1	96.0
c-use	min	46.2	68	100	0	13.5	54.0
	średnio	70.7	75.3	100	12.4	34.1	74.3
	max	84.6	84	100	56.0	67.6	98.0
p-use	min	20	61	50	0	36.4	53.0
	średnio	65.8	69.1	83.3	7.6	53.0	64.9
	max	92.5	78	100	55.6	81.8	80.0

3. Mutacje programów

Proste transformacje kodu są wprowadzane do programu za pomocą operatorów mutacji otrzymując zmutowany program (mutanta). Każdy mutant jest programem, w którego kodzie dokonano tylko jednej zmiany. Mutant zostaje wykryty ("zabity") przez test, jeśli po wykonaniu zmutowanego programu dla określonych danych testowych, zachowuje się on inaczej niż oryginalny program np. otrzymamy "zmienione" wyjście tego programu. Mogą istnieć mutanty, zwane mutantami równoważnymi, które dla każdego zbioru danych wejściowych mają identyczne zachowanie jak dla niezmutowanego programy i nie mogą być wykryte przez żadne testy.

Ze względu na złożoność wykrywania mutantów, dla każdego programu badano po kilka rodzajów mutacji modyfikujących wybrane instrukcje programu. Zastosowane mutanty, wyznaczone przez rodzaje mutacji oraz miejsce ich wystąpienia, zostały zaproponowane uwzględniając instrukcje występujące w wybranym fragmencie kodu, odpowiedzialnego za realizację algorytmów diagnostyki systemowej. Rozpatrywane są tylko takie mutacje, które nie są wykrywane przez kompilację zmutowanego programu. W badaniach wykorzystano następujące rodzaje operatorów mutacji [1,5]:

- zmiany zmiennych i stałych Oxx (OVV, OVC, OCV, OCC, OPP, OVI),

- modyfikacji instrukcji sterujących Sxx (SRN, SOI, SRI, SWR, SCA, SGL),
- modyfikacji warunków w instrukcjach warunkowych Cxx (CDE, CCO),
- modyfikacji wyrażeń arytmetycznych i logicznych Exx (EAI, ENI, EUI, EUR, EOR, ELR, ERR).

W tab. 4 dla każdego rodzaju mutacji i każdego programu podano liczbę analizowanych mutantów (kolumna M) oraz liczby wykrytych mutantów (kolumna Z). W kolumnie Z podano kolejno liczbę mutantów wykrytych przez podstawowe testy funkcjonalne oraz jeśli istnieją, liczbę wykrytych tylko przez testy dodatkowe dla mutacji i liczbę wykrytych przez testy dodatkowe dla pokryć. Wartości dla poszczególnych operatorów mutacji można znaleźć w [1]. W programach (1) i (5) zidentyfikowano część mutantów jako równoważne. W innych programach (zwłaszcza w 4), w których pozostały niewykryte mutanty przypuszcza się, że niektóre z nich też są mutantami równoważnymi. Nie udało się jednak wykazać ich równoważności. Określenie równoważności mutantów okazało się trudne i wymagające nie dającej się automatyzować analizy poszczególnych przypadków. Ogólnie, liczba niewykrytych i nierównoważnych mutanów w programach (1,2,5,6) jest bardzo mała w porównaniu do liczby wszystkich mutantów, a i w pozostałych programach (3,4) wykryto zdecydowaną większość mutantów.

Tab. 4 Wykrywalność różnych rodzajów mutacji

Rodzaje mutacji	Liczby mutantów i wykrytych m. w programach 1 - 6											
	1		**2**		**3**		**4**		**5**		**6**	
	M	Z	M	Z	M	Z	M	Z	M	Z	M	Z
Oxx	15	13,2	3	3	10	8	12	7			6	6
Sxx			11	10	1	1	11	9			6	6
Cxx			6	5, 1	32	23,-,3						
Exx	9	6			4	3	32	20	346	153,97	6	6
Mut. równoważnych	3		0		0		0		91		0	

Charakterystykę zbiorów testów, przy pomocy których wykrywano mutanty podano w tab. 5. Liczności ostatecznych zbiorów testów maksymalizujących zarówno pokrycia jak i wykrywalność mutacji są dla poszczególnych programów większe od zbiorów testów maksymalizujących tylko pokrycia lub tylko wykrywalność mutacji. Zbiory te są jednak często znacznie mniejsze od zbiorów wszystkich rozpatrywanych testów, oraz są zawsze mniejsze od zbiorów podstawowych testów funkcjonalnych.

Kolejne porównanie testów dotyczy efektywności wykrywania poszczególnych rodzajów mutacji (tab. 6). Wartość 100% w i-tej kolumnie oznacza, że dla i-tego programu wszystkie testy wykrywały wszystkie mutanty z rodzaju określonego w danym wierszu. W przypadku dwóch wartości, pierwsza jest określona z pominięciem mutantów z danego rodzaju uznanych za równoważne. Efektywność testów jest bardzo zróżnicowana. Niektóre testy

funkcjonalne nie wykrywały żadnego, lub co najwyżej jednego mutanta. Inne testy potrafiły wykryć wszystkie mutanty dla danego programu. Dla operatorów mutacji OVV, SRI, SCA, CCO, prawie wszystkie testy wykrywały jakiegoś mutanta. Natomiast najmniej testów wykrywało jakiegokolwiek mutanta z operatorami OVI, SWR, EAI, ENI.

Tab. 5 Statystyka testów wykrywających mutanty

	Programy					
	1	**2**	**3**	**4**	**5**	**6**
Liczba podstawowych testów	11	10	6	10	6	7
Liczba dodatkowych testów	2	2	2	1	7	0
W sumie testów , w tym testów	13	12	8	11	13	7
maksymalizujących wykrywalność m.	4	3	3	3	4	1
Liczba dodatkowych testów z pokryć	-	-	2	2	1	-
Ostateczny zbiór testów dla mutacji	4	3	4	5	4	1
Łącznie liczba wszystkich testów	16	12	8	19	14	7
Liczba testów maksymalizująca i pokrycia i wykrywalność mutantów	**7**	**6**	**5**	**8**	**4**	**5**

Tab. 6 Efektywność testów wykrywających mutanty

Rodzaj mutacji	Liczba testów wykrywających mutacje danego rodzaju [średnio w % do liczby wszystkich testów]					
	1	**2**	**3**	**4**	**5**	**6**
Oxx	40.5	69.7	58.3	26.9		42.9
Sxx		76.0	87.5	15.4		35.7
Cxx		59.1	36.5			
Exx	44.0 (*29.4*)		43.8	25.0	52.7 (*39.2*)	64.2

W tab. 7 przedstawiono charakterystykę wykrywalności mutantów dla poszczególnych rodzajów mutacji. Dla wykrytych mutantów podano jaki procent mutantów był wykryty w wyniku jednego z następujących sposobów zakończenia zmutowanego programu (kolumny w-s):

w – program generuje błędne wyniki (ale kończy działanie),
k – program wykrył błąd i wysłał komunikat,
z – program pętli się, nie generuje wyników,
s – inny błąd, w tym błędy zgłoszone przez system: np. segmentation fault, core dump, bus error.

W przypadku, gdy program skończył działanie i wygenerował wyniki, określenie jego wykrywalności mogło być realizowane w prosty sposób przez porównanie tych wyników z określonymi uprzednio wynikami poprawnymi (co jednak nie rozwiązuje problemu mutantów równoważnych). Oprócz wyników końcowych istnieją jednak jeszcze inne czynniki, np. czas działania programu. Dla jednego z mutantów z operatorem SCA (modyfikującego switch), czas

działania mutanta był około 10 razy dłuższy niż czas wykonania tych samych danych testowych przez niezmutowany program (2). Ponieważ wygenerowane dane były poprawne i ograniczenie czasu działania programu nie było określone w jego wymaganiach, to uznano, że mutant ten nie został wykryty przez testy. Z drugiej strony istniały mutanty które działały około 20 minut (dla czasów niezmutowanych programów nie przekraczających minuty), które uznano za wynik zapętlenia się programu. W sumie 12% przypadków wykrycia mutantów wynikało z uznania braku stopu w programie (kolumna z). Dopiero sprawdzenie zmutowanego kodu i ewentualnie rozważenie skutków propagowania się danej mutacji pozwoliły uzasadnić słuszność takiej decyzji. Niestety w ogólności rozstrzygnięcie tego, czy program zapętlił się może być trudne do ustalenia. Dla programów z określonymi wymaganiami maksymalnych czasów reakcji, wydłużony czas działania można by uznać za wykrycie mutanta (nawet gdy nie jest związany z zapętleniem się programu).

Tab. 7 Sposoby wykrywania mutantów

Rodzaj mutacji	Liczba mutantów	Mutanty równo-ważne	Mutanty niewykryte	Mutanty wykryte				
				liczba	wg sposobu wykrycia [w %]			
					w	**k**	**z**	**s**
Oxx	46	0	7	39	60.0	11.3	13.0	15.7
Sxx	29	0	3	26	19.8	5.6	36.5	38.1
Cxx	38	0	6	32	37.3	18.3	32.0	12.4
Exx	397	94	18	285	70.3	1.0	8.9	19.8
Łącznie	**510**	**94**	**34**	**382**	**65.1**	**2.8**	**12.0**	**20.1**

Dla połowy operatorów mutacji większość wykrywanych mutantów kończy swoje działanie z błędnym wynikiem (w). Mutacje związane ze zmianą stałych (OVC, OCV, OCC) były w całości wykryte i dawały zawsze zakończenie mutanta kontrolowane przez ten program (w lub k). W przypadku operatora mutacji SGL zmiana etykiety skoku w instrukcji GOTO powoduje całkowicie odmienne zachowanie programu. Wiele testów nie wykrywa takich mutantów, a gdy są wykrywane to na ogół w wyniku błędów zgłoszonych przez system (s). Część mutacji zmieniających operator relacji - ERR jest nieprzydatna z punktu widzenia oceny testów, gdyż jest wykrywana przez każdy wykonywany test. To samo dotyczy większości mutantów wymuszających jedno wykonanie pętli - SOI, które powodują przedwczesne zakończenie się programu. Zgłoszenie błędu systemu dla mutantów SOI jest uzasadnione, gdyż ograniczenie wykonań pętli może w istotny sposób zmienić sterowanie programu. Również zgodne z oczekiwaniami jest to, że mutant SRI (zmieniający kierunek wykonania pętli) spowodował zapętlenie się programu. Podobnie dla mutantów dotyczących wyrażeń arytmetycznych i logicznych nie zawsze zostały wygenerowane wyniki, gdyż zmienione wyrażenia określały sterowanie programem (np. wykonanie pętli). W szczególności, dla operatorów mutujących wyrażenia arytmetyczne EOR, a zwłaszcza EUR, w wielu przypadkach programy wykrytych mutantów zawiesiły się podczas działania.

Równoważne lub niewykryte pozostały głównie mutanty modyfikujące wyrażenia (EAI, EUR, EOR, ELR, ERR). Zachodzi to wówczas, gdy istotny jest wyłącznie pewien warunek określony przez dane wyrażenie, a nie jego dokładna postać. Dla danych wartości argumentów warunek ten jest taki sam przed i po mutacji, np. dla dwóch wartości logicznych TRUE ich koniunkcja i alternatywa (&& lub ||) da taki sam wynik, także ten sam warunek będzie spełniony dla wielu wartości przy zamianie relacji (np. < na <=) itd.

Analizując testy maksymalizujące pokrycia i wykrywalność mutacji stwierdzono, że już dla kilku testów (różnych zestawów danych wejściowych) można było osiągnąć pokrycia rzędu 80-100%, przy liczbie niewykrytych mutantów rzędu 0-19. Zwiększenie różnych typów pokrycia powyżej pewnej granicy jest związane często z wykonywaniem dodatkowych testów przeznaczonych do pokrycia tylko wybranych, niedużych fragmentów kodu. Jak stwierdzono, część pozostałego kodu jest niemożliwa do pokrycia przez testy funkcjonalne. Natomiast niektóre z pozostałych niewykrytych mutantów mogą być mutantami równoważnymi. Określenie równoważności mutantów stanowi jednak podstawowy problem, nawet dla niezbyt dużych programów. Ponadto nie zawsze można rozstrzygnąć jaki jest wynik działania programu, np. przy braku ograniczeń czasowych. Zamiast przyjętego tutaj wykonywania programu do końca i oceny efektów jego działania "strong mutation testing", możliwe jest badanie tylko zmiany stanu programu w wyniku mutacji programu "weak mutation testing". Jednak takie podejście może być trudne do zastosowania dla programów wieloprocesowych. Ponadto mutacje obejmowały tylko podstawowe operatory, które mogą być zastosowane do języków programowania ogólnego przeznaczenia. Dla programów (1-4) napisanych w C++ można by użyć również innych operatorów, które odnoszą się do obiektowych cech tego języka.

Bibliografia

[1]. A. Derezińska, Analiza testów maksymalizujących pokrycia i wykrywalność mutacji programów, Rap. Bad. II PW 7/2001

[2]. M. R. Lyu, J. R. Horgan, S. London, A coverage analysis tool for the effectiveness of software testing, IEEE Trans. on Reliability, vol. 43, no. 4, December 1994, pp 527-534

[3]. J. Sosnowski, Analiza pokrycia testów, Rap. Bad. II PW 6/2001

[4]. J. M. Voas, G. McGraw, Software fault injection – inoculation programs against errors, John Wiley & Sons, 1998