Zastosowanie metody badania pokrycia przepływu danych w testowaniu programów obiektowych

Artur Rembiszewski
Wydział Elektroniki i Technik Informacyjnych
Politechnika Warszawska
A.Rembiszewski@stud.elka.pw.edu.pl

Streszczenie

Kryteria badania pokrycia kodu związane zarówno z przepływem sterowania, jak i te związane z przepływem danych pierwotnie zdefiniowane były dla programów proceduralnych. W chwili obecnej programowanie obiektowe stało się już powszechnie obowiązującym standardem i konieczne było dostosowanie do niego kryteriów testowania. Głównym problemem w dostosowaniu kryteriów pokrycia przepływu danych do programów obiektowych jest zidentyfikowanie instrukcji, które mają być traktowane jako definicje i użycia zmiennych. Do tej pory zaproponowano kilka modeli radzących sobie z tym zagadnieniem. Artykuł ten zawiera opis niektórych z nich, ze szczególnym uwzględnieniem kryteriów dla języka Java wraz z oceną ich użyteczności. W pracy zawarto także analizę możliwości implementacji narzędzia wspierającego testowanie programów obiektowych przy użyciu opisanych kryteriów.

1. Wstęp

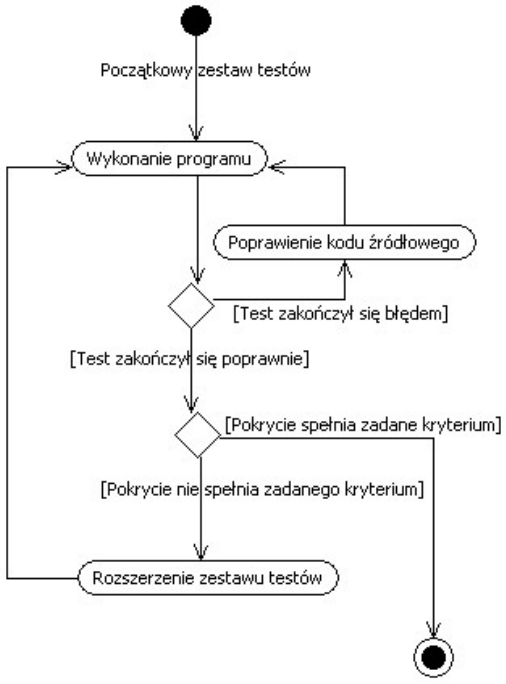
Jedną z popularnych technik testowania oprogramowania na poziomie jednostkowym jest dynamiczne testowanie strukturalne. Metoda ta polega na tworzeniu zestawów przypadków testowych w opraciu o znajomość kodu źródłowego testowanej aplikacji. Jakość tych zestawów może być mierzona przy pomocy różnych kryteriów pokrycia. Jedną ze stosowanych grup są kryteria pokrycia przepływu danych. Pierwotnie zdefiniowane były one dla programów proceduralnych. W tej pracy zaprezentowano różne koncepcje dostosowania kryteriów związanych z przepływem danych do programów obiektowych.

Dalsza część artykułu podzielona jest następująco: W rozdziale 2 omówiono model testowania strukturalnego z wykorzystaniem informacji o pokryciu kodu oraz wprowadzono pojęcia związane z mierzeniem pokrycia przepływu danych. Rozdział 3 zawiera ogólny opis dostosowania metody do odnajdywania łańcuchów o zasięgu całej klasy. Propozycja konkretnej implementacji metody przedstawionej w rozdziale 3 została umieszczona w rozdziale 4. Rozdział 5 prezentuje sposób określenia, które instrukcje definiują, a które używają zmiennych języka Java wykorzystywany w narzędziu JaBUTi [1]. Inne podejście – traktujące obiekt jako zbiór atrybutów widocznych z zewnątrz – zostało zaproponowane w rozdziale 6, a rozdział 7 zawiera analizę możliwości zaimplementowania tej metody dla języka Java. W rozdziale 8 streszczono wyniki badań pokazujących użyteczność przedstawionych metod. Pracę kończy krótkie podsumowanie.

2. Testowanie strukturalne

Testowanie strukturalne jest metodą opartą o znajomość kodu źródłowego. Schemat testowania oprogramowania wykorzystujący tę metodę przedstawiono na rysunku 1. Pierwszym etapem jest przygotowanie początkowego zestawu przypadków testowych. Następnie należy przeprowadzić testy pod kontrolą narzędzia pozwalającego określić ich

jakość. Miarą jakości zestawu testów jest to, w jakim stopniu pokrywają one kod źródłowy i różne możliwe ścieżki przepływu sterowania przez program.



Rys. 1 Testowanie z badaniem pokrycia kodu

Jeśli podczas przeprowadzania testów został wykryty błąd programu należy poprawić kod źródłowy i powtórzyć testy. Po poprawnym zakończeniu testów dostępna jest informacja o pokryciu kodu oraz fragmentów, które pozostały niepokryte. Na tej podstawie przygotowywane są kolejne przypadki testowe i procedura powtarzana jest aż do osiągnięcia satysfakcjonującego pokrycia kodu.

Pokrycie kodu może być mierzone przy pomocy różnych kryteriów. Najprostszym stosowanym kryterium jest pokrycie instrukcji. Dla danego fragmentu kodu badane jest czy każda jego instrukcja została wykonana. Jest to przykład kryterium związanego z przepływem sterowania. Inną grupą kryteriów pokrycia kodu są kryteria związane z przepływem danych. Po raz pierwszy kryterium pokrycia przepływu danych (*all-c-uses* oraz *all-p-uses*) zostało zdefiniowane w roku 1985 w pracy [2]. Autorzy opierają się na założeniu, że stwierdzenie czy pewna operacja przebiegła prawidłowo możliwe jest tylko w przypadku, gdy jej wynik zostanie użyty. Do badania pokrycia przepływu danych wykorzystuje się pojęcie grafu definicja-użycie DUG [3] (ang. def-use graph), który jest pewnym rozszerzeniem grafu przepływu sterowania. Każdy węzeł zawiera dodatkowo informację o definicji, lub użyciu zmiennych z nim związanych. Definicja zmiennej jest to instrukcja, w wyniku której do zmiennej zostaje przypisana wartość, użycie zmiennej to instrukcja, której wykonanie wymaga podstawienia wartości tej zmiennej. Kryteria związane z przepływem danych wymagają pokrycia pewnych ścieżek pomiędzy definicją a użyciem wszystkich zmiennych występujących w testowanym fragmencie kodu źródłowego. Dokładny opis kryteriów znajduje się w [2].

3. Dostosowanie kryteriów pokrycia przepływu danych do programowania obiektowego

W podstawowej wersji badanie pokrycia przepływu danych ograniczone jest do analizowania łańcuchów definicja-użycie w zasięgu jednej metody. W odniesieniu do testowania programów obiektowych należy rozszerzyć ten zasięg o inne metody wywoływane w ramach tej analizowanej oraz uwzględnić łańcuchy, które powstaną w wyniku wywołania pewnej sekwencji metod publicznych testowanego obiektu. Algorytmy służące do znalezienia łańcuchów w zasięgu całej klasy zostały opisane w [4]. Dla języków proceduralnych już wcześniej został opracowany algorytm PLR [5], który pozwala na znalezienie takich par definicja – użycie, w których definicja zmiennej występuje w jednej procedurze, a jej użycie w procedurze wywoływanej, lub tej która ją wywołała. Algorytm posługuje się interproceduralnym grafem przepływu sterowania (ang. interprocedural control flow graph), który powstaje w wyniku

połączenia grafów przepływu sterowania dla powiązanych ze sobą procedur. Metodę można stosować do zmiennych globalnych, lub atrybutów klasy oraz do powiązania parametrów przekazanych do metody przez referencję.

Dla programów obiektowych autorzy artykułu [4] zdefiniowali następujące poziomy testowania:

- *Intra-method* – testowanie w zasięgu jednej metody. Jest to model pierwotny zdefiniowany w [2]. Nie uwzględnia atrybutów klasy ani interakcji pomiędzy metodami.
- *Inter-method* – testowanie metody publicznej klasy wraz z metodami, które są bezpośrednio lub pośrednio przez nią wywoływane. Pozwala znaleźć pary definicja-użycie dla atrybutów klasy.
- *Intra-class* – uwzględnia łańcuchy powstałe w wyniku wywołania różnych sekwencji metod publicznych testowanej klasy. Liczba możliwych sekwencji wywołań metod publicznych jest w większości przypadków nieskończona, dlatego testowane są tylko niektóre z nich.

Zostały też zdefiniowane trzy typy par definicja-użycie: *intra-method*, *inter-mthod* oraz *intra-class*, o zasięgu odpowiadającym przedstawionym poziomom testowania.

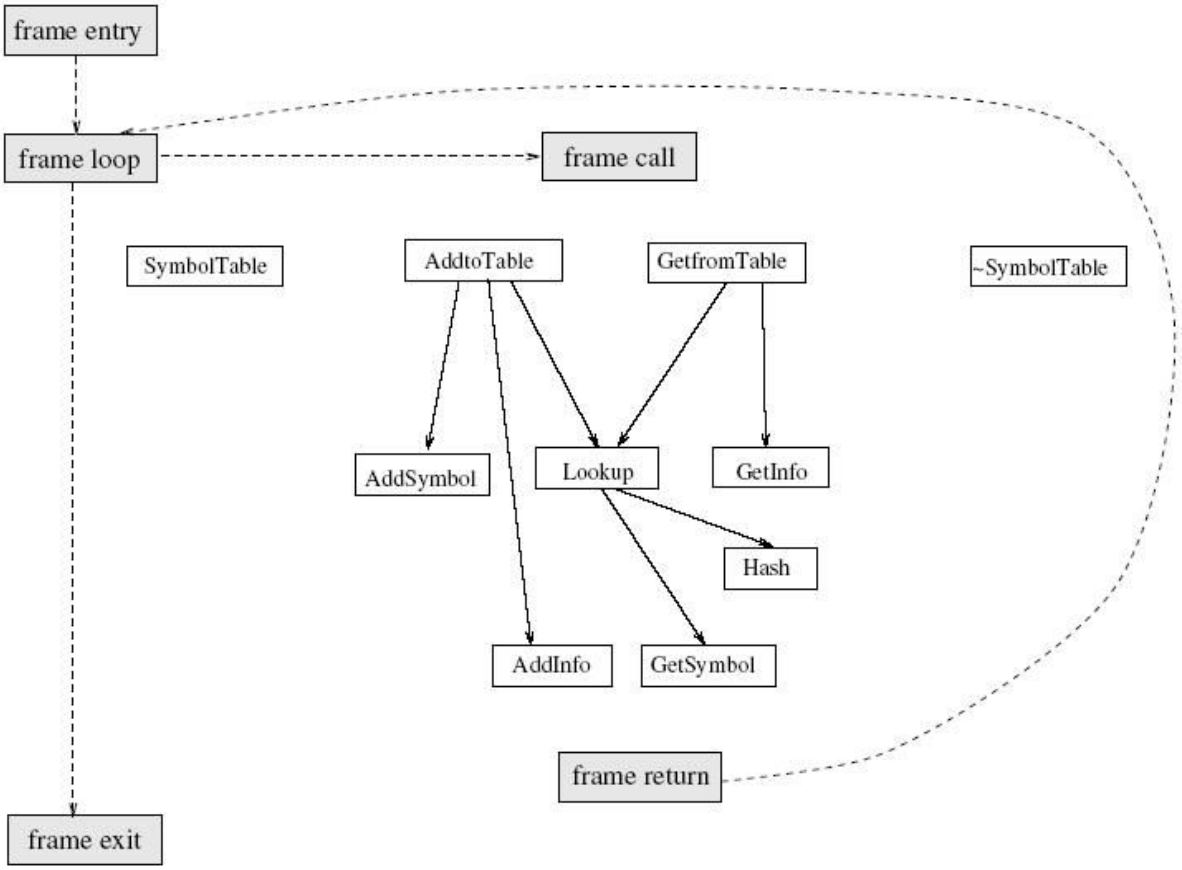
4. Realizacja testów dla programów obiektowych

Zastosowanie testowania metodą badania pokrycia przepływu danych dla całej klasy, wymaga znalezienia wszystkich par definicja-użycie – zarówno *intra-method* jak i *inter-method* oraz *intra-class*. Jedna z metod odnajdywania par definicja-użycie dla klasy została zaproponowana w artykule [4]. Wykorzystywany jest tu graf przepływu sterowania dla całej klasy – CCFG (ang. Class Control Flow Graph).

Algorytm konstrukcji CCFG dla klasy C wygląda następująco [4] :

1. G = graf wywołań metod dla klasy C (konstrukcja grafu została opisana w dalszej części rozdziału)
2. Dodaj do G wierzchołki sterownika testu (ang. frame)
3. Dla każdej metody M w C:
   - Zastąp wierzchołek odpowiadający metodzie M w G grafem przepływu sterowania metody M.
4. Dla każdego wierzchołka S grafu G reprezentującego wywołanie metody M z klasy C:
   - Zastąp wierzchołek S wierzchołkami reprezentującymi przejście i powrót z metody M. Połącz je odpowiednio z wierzchołkiem startowym i końcowym podgrafu reprezentującego metodę M.
5. Dodaj do G odpowiednie krawędzie łączące części odpowiadające poszczególnym metodom z wierzchołkami sterownika testu.

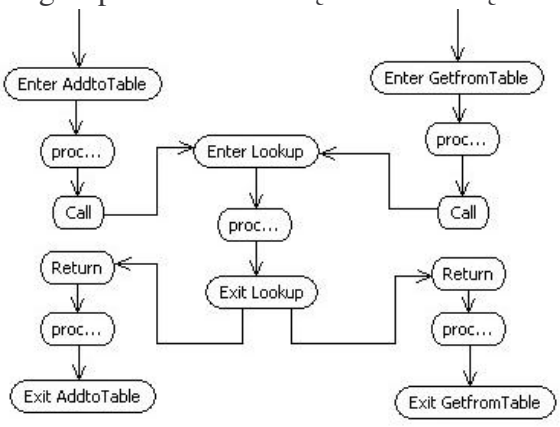
W pierwszym kroku algorytmu konstruowany jest graf wywołań dla klasy (ang. class call graph). Jego wierzchołki reprezentują poszczególne metody klasy. Krawędź z wierzchołka M1 do M2 istnieje, jeśli metoda M1 wywołuje metodę M2. W kroku 2 do grafu G dodawane są wierzchołki sterownika: *frame entry*, *frame loop*, *frame call*, *frame return* oraz *frame exit*. Umożliwiają one stworzenie ścieżki reprezentującej wywołanie sekwencji metod publicznych klasy. Dodatkowe wierzchołki sterownika oraz graf wywołań dla klasy zostały przedstawione na rysunku 2. W kroku 3 algorytmu wierzchołki odpowiadające metodom zastępowane są grafami przepływu sterowania tych metod, a następnie w kroku 4 wszystkie wierzchołki reprezentujące wywołania metod z klasy C zastępowane są parą wierzchołków reprezentujących przejście i powrót z odpowiednich metod. Wierzchołki reprezentujące przejście do metody M łączone są krawędzią z wierzchołkiem startowym podgrafu reprezentującego metodę M, a wierzchołek końcowym tego podgrafu łączony jest z odpowiednim wierzchołkiem



Rys. 2 Konstrukcja CCFG, źródło: [4]

reprezentującym powrót z metody. Przykładowy fragment grafu przedstawiono na rysunku 3. Metody `AddtoTable` oraz `GetfromTable` wywołują metodę `Lookup`. W ostatnim kroku dodawane są krawędzie łączące wierzchołek *frame call* z wierzchołkami startowymi metod publicznych oraz ich wierzchołki końcowe z wierzchołkiem *frame return*.

Zastąpienie grafu DUG w metodzie badania pokrycia przepływu danych grafem CCFG poszerzonym o informacje o definicjach i użyciach zmiennych w poszczególnych wierzchołkach pozwala na zbadanie pokrycia przepływu danych dla całej klasy (testowanie o zasięgu intra-class). Należy jednak zwrócić uwagę na pewne ograniczenie podczas wyszukiwania par definicja-użycie przy pomocy tak skonstruowanego grafu. Wierzchołek startowy podgrafu reprezentującego metodę wywoływaną przez kilka innych metod będzie miał kilka krawędzi wchodzących. Podobnie wierzchołek końcowy tego podgrafu będzie miał kilka krawędzi wychodzących. Podczas przechodzenia przez graf dozwolone są tylko te ścieżki, w których krawędź wychodząca z podgrafu dla każdej metody prowadzi do tego samego podgrafu, z którego prowadzi krawędź wchodząca.



Rys. 3 Fragment grafu CCFG

Dla przykładowego fragmentu pokazanego na rysunku 3 klasyczny algorytm mógłby wyszukać parę, w której definicja zmiennej wykonywana jest w metodzie

`AddtoTable` przed wywołaniem metody `Lookup`, a jej użycie w metodzie `GetfromTable`, po wywołaniu metody `Lookup`. W praktyce taki przepływ sterowania nie jest jednak możliwy. Implementacja algorytmu przechodzącego po grafie CCFG w celu wyszukania par definicja-użycie musi w specjalny sposób traktować wierzchołki, które odpowiadają wywołaniom metod, oraz wykorzystywać stos wywołań w celu wybrania odpowiedniej krawędzi wychodzącej z wierzchołka reprezentującego wyjście z metody (w przykładzie na rys. 3 wierzchołek *Exit Lookup*).

5. Określenie definicji i użyć zmiennych w programach obiektowych

W dostosowywaniu kryteriów związanych z przepływem danych do programów obiektowych poza znalezieniem łańcuchów definicja-użycia o zasięgu *inter-class* problemem jest także określenie, które instrukcje powinny być traktowane jako definicje, a które jako użycia zmiennych. Jedno z możliwych podejść zostało zaproponowane w artykule [3]. Autorzy skupili się na kryteriach dla języka Java. Podstawowa definicja grafu DUG (ang. def-use graph) zaproponowana przez Zhao w [6] przewiduje rozszerzenie zwykłego grafu przepływu sterowania o informacje dotyczące zmiennych przekazanych jako argumenty do rozpatrywanej metody, oraz zmiennych zdefiniowanych w jej ciele. Autorzy artykułu [3] proponują rozszerzenie tej metody o zmienne będące atrybutami klasy, do której metoda należy z włączeniem zmiennych statycznych tej klasy. Założono, że definicja tych zmiennych wykonywana jest w węźle startowym grafu. Ponadto w grafie DUG zostały rozróżnione dwa typy krawędzi: regularne – reprezentujące kolejne przejścia podczas normalnego wykonania programu, oraz gałęzie związane z wyjątkami (ang. exceptions).

W języku Java typy zmiennych dzielą się na dwa podstawowe rodzaje: typy proste (np. `int`, `boolean`) oraz referencje do obiektów. Typy agregacyjne (np. `int[]`) można potraktować jako referencje. Określenie, które instrukcje mają być traktowane jako definicja, a które jako użycia zmiennych dla typów prostych jasno wynika z pierwotnej definicji badania pokrycia przepływu danych [2]. Dla typów referencyjnych określono następujące reguły [3]:

1. Typy agregacyjne traktowane są jako obiekty zawierające referencje do kolejnych obiektów. Definicja (użycie) któregokolwiek elementu składowego traktowana jest jako definicja (użycie) całej zmiennej. W przykładzie „`a[i]=a[j]+1`” występuje definicja oraz użycie zmiennej `a[]`.
2. Dla zmiennej typu `a[][]` dostęp do elementu jest traktowany jako definicja (użycie) zmiennej `a[][]`. W przykładzie „`a[i][j]=10`” występuje definicja zmiennej `a[][]`, natomiast w przykładzie „`a[i]=new int[10]`” występuje definicja zmiennej `a[]`.
3. Każda definicja (użycie) niestatycznego pola innej klasy (lub elementu typu agregacyjnego) traktowane jest jako użycie referencji do obiektu oraz definicja (użycie) tego pola.
   Przykład: `ref_1` oraz `ref_2` są referencjami do obiektów klasy C zawierającej pola x oraz y. W wyrażeniu „`ref_1.x = ref_2.y`” występują: użycie zmiennych `ref_1` i `ref_2`, użycie zmiennej `ref_2.y` oraz definicja zmiennej `ref_1.x`.
4. W przypadku pól statycznych klas odwołanie traktowane jest jako definicja (użycie) tylko tego pola. W przykładach „`C.x=10`” oraz „`ref_1.x=10`”, gdzie `ref_1` jest referencją do obiektu klasy C, a x jest polem statycznym występuje tylko definicja zmiennej `C.x`.

5. Wywołania metod traktowane są jako użycie zmiennej będącej referencją obiektu w kontekście, którego wywoływana jest metoda. W przykładzie „`ref_1.foo(e1,e2)`” występuje użycie zmiennej `ref_1`. Zasady dotyczące zmiennych `e1` i `e2` pozostają takie jak opisano w punktach powyższych.
6. Wywołanie metody z tej samej klasy traktowane jest jako wywołanie przy użyciu zmiennej `this`. Definicja zmiennej `this` występuje w węźle startowym grafu DUG.

6. Zewnętrzne testowanie obiektu

Metody przedstawione w poprzednich rozdziałach skupiają się na testowaniu wewnętrznej struktury obiektu. Przedstawione algorytmy mają na celu odnalezienie łańcuchów pomiędzy zmiennymi występującymi w różnych metodach testowanej klasy. Inne podejście zostało zaproponowane w artykule [7]. Chen i Kao zdefiniowali dwa kryteria testowania odnoszące się do modelu obiektowego. Pierwsze z nich - *all-du-pairs* – mierzy pokrycie przepływu danych traktując obiekt jako atomową jednostkę. Pojęcia definicji i użycia odnoszą się tu do całego obiektu, a nie jego poszczególnych składowych. Drugie kryterium - *all-bindings* – bierze pod uwagę występowanie dziedziczenia oraz polimorfizm i pozwala znaleźć metody, które powinny być ponownie przetestowane w kontekście klasy pochodnej oraz zidentyfikować wiązania (ang. binding) dynamiczne, które należy przetestować. W celu dokładniejszego opisania wymienionych kryteriów zdefiniowano następujące pojęcia:

Stan obiektu – pewna kombinacja wartości zmiennych będących atrybutami obiektu. Przez wartość dla atrybutu będącego innym obiektem rozumiany jest jego stan.

Definicja obiektu – Instrukcja w wyniku, której stan obiektu jest inicjalizowany lub zmieniany. Sytuacja taka zachodzi, gdy:

1. Wywoływany jest konstruktor obiektu.
2. Wywoływana jest instrukcja będąca definicją jednego z atrybutów obiektu.
3. Wywoływana jest metoda obiektu w wyniku, której zmienna będąca atrybutem obiektu jest inicjalizowana lub modyfikowana.

Użycie obiektu – Występuje w jednym z przypadków:

1. Zmienna będąca atrybutem obiektu jest używana bezpośrednio przez instrukcję znajdującą się na zewnątrz klasy.
2. Wywoływana jest metoda obiektu, która używa zmiennych będących atrybutami obiektu.
3. Obiekt przekazywany jest jako parametr wywołania funkcji.

Wiązanie – Skojarzenie obiektu z konkretną klasą w czasie wykonania programu.

Używając przedstawionych pojęć zdefiniowano następujące kryteria pokrycia kodu:

All-bindings – Wymagane jest, aby każde możliwe wiązanie dla każdego obiektu wystąpiło co najmniej raz w instrukcji będącej definicją obiektu, oraz co najmniej raz w instrukcji będącej użyciem obiektu.

All-du-pairs – Wymagane jest, aby dla każdej definicji obiektu wykonana została co najmniej jedna wolna od definicji ścieżka do każdego osiągalnego użycia tego obiektu.

W odróżnieniu od kryteriów zdefiniowanych w rozdziale 5 według kryterium *all-du-pairs* instrukcja `ref_1.foo(e1,e2)` nie zostanie potraktowana jako użycie zmiennej referencyjnej `ref_1`, ale jako definicja obiektu `ref_1` jeśli metoda `foo` zmienia stan obiektu lub jako użycie obiektu `ref_1`, jeśli metoda `foo` używa zmiennych będących atrybutami obiektu.

Autorzy artykułu [7] zaproponowali algorytm pozwalający znaleźć pary-definicja użycie dla zadanych obiektów o zasięgu *inter-method*. W celu zmniejszenia liczby koniecznych do przetestowania par definicja-użycie wprowadzono dodatkowe ograniczenie. Sekwencja metod:

```
ref_1.set_state();
ref_1.use_state();
```
jest parą definicja-użycie obiektu `ref_1` tylko wtedy, gdy metoda `use_state` używa tego samego atrybutu obiektu `ref_1`, który jest definiowany w wyniku wywołania metody `set_state`.
Przykład:
Niech klasa `C` ma 3 atrybuty: `x,y,z`. Niech metoda `set_state` definiuje atrybuty `x` oraz `y`. Jeśli metoda `use_state` używa atrybutu `x` lub `y` przedstawiona sekwencja jest parą definicja-użycie dla obiekty `ref_1`. Jeśli natomiast używany jest tylko atrybut `z`, przedstawiona sekwencja nie jest parą def-use.

Zastosowanie zaprezentowanej metody wymaga znajomości struktury wewnętrznej każdej z używanych klas. W artykule [7] został zaproponowany algorytm pozwalający znaleźć wszystkie takie pary definicja-użycie. W algorytmie zostało użyte pojęcie grafu OCFG (ang. Object Control Flow Graph). Graf OCFG jest konstruowany dla całego programu. Wierzchołki Sn grafu reprezentują poszczególne metody i funkcje. Każdy wierzchołek Sn zawiera podgraf będący grafem przepływu sterowania dla metody, którą reprezentuje *Sn* uzupełnionym o informacje o użyciach i definicjach zmiennych. Wierzchołki reprezentujące metody w grafie są oznaczane jako sn_i (ang. super node), natomiast wierzchołki reprezentujące instrukcje są oznaczane jako n_i. W grafie istnieją dwa typy krawędzi: krawędzie (n_i, sn_j) oznaczające wywołanie innej metody oraz krawędzie (sn_i, sn_j) oznaczające, że w metodzie reprezentowanej przez sn_i występuje definicja pewnej zmiennej, która jest używana w metodzie reprezentowanej przez sn_j. Algorytm znajdywania par definicja-użycie dla całego programu z ograniczeniem wprowadzonym w [7] składa się z następujących kroków:

1. Dla każdej metody *i* znajdź zbiory def_i i use_i zmiennych definiowanych i używanych wewnątrz tej metody. Zbiory nie zawierają zmiennych lokalnych metod.
2. Utwórz graf OCFG z pominięciem krawędzi (sn_i, sn_j).
3. Wykorzystując zbiory znalezione w kroku 1 znajdź pary definicja-użycie o zasięgu *intra-method*. (n_i, n_j) jest parą definicja-użycie jeśli w wierzchołku n_i występuje definicja zmiennej `x` lub istnieje krawędź (n_i, sn_k) i $x \in def_k$, w wierzchołku n_j występuje użycie zmiennej `x` lub istnieje krawędź (n_j, sn_m) i $x \in use_m$, oraz istnieje ścieżka z n_i do n_j.
4. Dodaj do OCFG krawędzie (sn_i, sn_j). Krawędź (sn_i, sn_j) jest dodawana jeśli istnieje para definicja-użycie (n_i, n_j) oraz krawędzie (n_i, sn_i) i (n_j, sn_j).
5. Wykorzystując krawędzie dodane w kroku 4 znajdź pary definicja-użycie o zasięgu *inter-method*. (n_i, n_j) jest parą definicja-użycie jeśli spełnione są następujące warunki:
   a. wierzchołek $n_i \in sn_k$
   b. wierzchołek $n_j \in sn_l$
   c. w wierzchołku n_i występuje definicja zmiennej `x` lub istnieje krawędź (n_i, sn_a) i $x \in def_a$
   d. w wierzchołku n_j występuje użycie zmiennej `x` lub istnieje krawędź (n_j, sn_b) i $x \in use_b$
   e. istnieje krawędź (sn_i, sn_j)
6. Jeśli w kroku 5 dodano nowe pary definicja-użycie powróć do kroku 4, w przeciwnym wypadku zakończ algorytm.

7. Testowanie oparte o stan obiektu dla języka Java

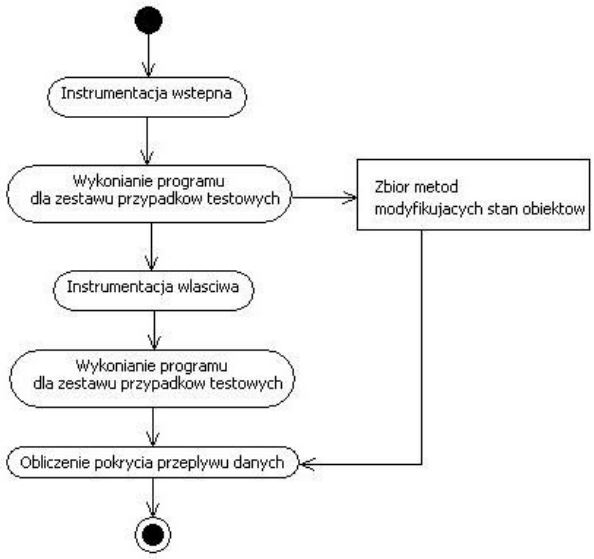
Algorytm zaprezentowany w rozdziale 6 pozwala zbadać przepływ danych dla programu obiektowego traktując obiekt jako pewną spójną jednostkę. Takie podejście jest bliższe paradygmatowi programowania obiektowego od podejścia, w którym przepływ danych mierzony jest osobno dla poszczególnych atrybutów obiektu. Zaproponowana metoda wymaga jednak znajomości kodu źródłowego klasy,

której obiekty są używane. W praktyce bardzo często wykorzystywane są gotowe biblioteki oferujące zestawy klas, dla których kod źródłowy jest niedostępny. Zbadanie przepływu danych dla fragmentu programu, w którym używane są obiekty takich klas jest możliwe po przyjęciu pojęć definicji i użycia obiektu opartych o jego stan – tak jak przedstawiono w rozdziale 6. Głównym problemem jest wtedy określenie, które metody modyfikują stan obiektu, a które jedynie go odczytują. W ogólnym przypadku bez znajomości kodu źródłowego używanych metod może być to niemożliwe. Pewne języki programowania udostępniają konstrukcje, dzięki którym taka informacja może być dostępna. Np. w języku C++ metody w deklaracji klasy mogą być oznaczone atrybutem `const`, który oznacza, że metoda ta nie modyfikuje pól klasy [8]. Poniżej przedstawiono przykład takiej deklaracji:

```
class Class1 {
public:
  int get_value() const;
};
```

W czasie kompilacji klasy sprawdzane jest czy metoda oznaczona atrybutem `const` nie zawiera instrukcji modyfikujących pola klasy. Jeśli zawiera, kompilacja kodu zostanie przerwana z komunikatem o błędzie.

Rozwiązanie przedstawione dla języka C++, nie może być jednak zastosowane dla języka Java, ze względu na elementu składniowego, który jednoznacznie określa czy metoda może modyfikować stan klasy. W dalszej części rozdziału zaproponowano mechanizm, który pozwala na dynamiczne wyszukanie zbioru metod modyfikujących stan obiektu dla każdej klasy w czasie wykonania programu. Schemat całego procesu testowania przedstawiono na rysunku 4. Konieczne jest dwukrotne wykonanie zestawu przypadków testowych dla testowanego programu. Podczas pierwszego uruchomienia generowany jest zbiór metod modyfikujących stan obiektów, który następnie wykorzystywany jest do obliczenia pokrycia po drugim wykonaniu.



Rys. 4 Dwustopniowe badanie pokrycia przepływu danych.

7.1 Określanie zbioru metod modyfikujących stan obiektu przy użyciu metody `hashCode`

W języku Java każda klasa dziedziczy z klasy `Object`. Jedną z metod przez nią udostępnianych jest metoda `hashCode()`. Metoda jako wynik działania zwraca liczbę typu `int` (liczba 32-bitowa ze znakiem) wyliczaną na podstawie elementów składowych obiektu oraz innych dostępnych o nim informacji. Założenia metody są następujące [9]:

- Każde wywołanie metody w kontekście tego samego obiektu, musi zwrócić tą samą wartość (o ile obiekt nie był modyfikowany). Wartość nie musi być taka sama po ponownym uruchomieniu programu.
- Jeśli dwa obiekty są równe (wg założeń projektowych danej klasy) wywołanie metody dla obydwu musi zwrócić tą samą wartość.
- Nie jest konieczne, aby wyniki zwrócone przez metodę dla dwóch różnych obiektów były różne.

Głównym zastosowaniem funkcji `hashCode()` jest pobieranie wartości funkcji mieszającej w celu umieszczenia

obiektu w tablicy mieszającej (ang. hash table). Kompilator języka Java nie wymusza implementacji tej metody, jednak zaleceniem oraz dobrą praktyką programistyczną jest jej nadpisywanie (ang. override) i własna implementacja dla własnych obiektów. Większość standardowych klas Java oraz klas udostępnianych w popularnych bibliotekach zawiera własną implementację metody `hashCode()` zgodną z wyżej przedstawionymi założeniami. W oparciu o nie można przyjąć, że jeśli stan klasy uległ zmianie, wartość zwracana przez metodę `hashCode()` również uległa zmianie. Na podstawie tego stwierdzenia można zaproponować algorytm pozwalający znaleźć zbiór metod modyfikujących stan, dla każdej z użytych klas. Zbiór ten może być określany dynamicznie w trakcie działania programu po wcześniejszym przeprowadzeniu odpowiedniej instrumentacji kodu. Instrumentacja polega na otoczeniu każdego wywołania każdej z używanych metod parą instrukcji pobierającą wartość funkcji `hashCode()` obiektu, przed oraz po wywołaniu sprawdzanej metody. Jeśli pobrane wartości są różne oznacza to, że metoda zmodyfikowała stan obiektu. Przykład instrumentacji został przedstawiony w tabeli 1. Linie oznaczone kursywą zostały dodane w wyniku instrumentacji.

```
{
int hashValue;
...
hashValue = obj.hashCode();
obj.method1();
if (hashValue!=obj.hashCode())
 setDef(obj.getClass(),"method1");
...
}
```

Tabela 1. Instrumentacja kodu

Sprawdzaną metodą jest `method1()` wywoływana w kontekście obiektu `obj`. Metoda `setDef` dopisuje nazwę sprawdzanej metody, do zbioru metod modyfikujących stan danej klasy. Tak przygotowany zbiór udostępnia informacje konieczne do zmierzenia pokrycia przepływu danych dla programu obiektowego.

7.2 Wady algorytmu opartego o metodę `hashCode`

Podstawową wadą sprawdzania, czy metoda modyfikuje stan obiektu w czasie wykonania programu jest fakt, że pewne metody mogą modyfikować stan obiektu warunkowo – w zależności od stanu, w jakim obiekt się znajdował przed wywołaniem metody, lub w zależności od argumentów z jakimi metoda została wywołana. Jeśli w trakcie wykonania programu warunek nie zostanie spełniony metoda zostanie błędnie sklasyfikowana jako niemodyfikująca. Analogicznie pewne metody obiektu mogą dla danego zestawu przypadków testowych w ogóle nie zostać wykonane, co nie pozwoli na ich sklasyfikowanie. Skutkiem tego może być pominięcie pewnych par definicje-użycie, które powinny być pokryte.

Drugą wadą algorytmu jest używanie metody `hashCode` do określania stanu obiektu. Przede wszystkim definicja metody nie mówi, że powinna ona rozróżniać obiekty o różnych stanach, ani też, że jej wartość powinna być określana w oparciu o wartości jej atrybutów. Ponadto w praktyce często zdarza się, że metoda ta nie jest zaimplementowana podczas tworzenia nowej klasy. Niezdefiniowanie metody skutkuje wywołaniem w tracie wykonania programu domyślnej implementacji z klasy `Object`, która nie uwzględnia atrybutów klasy podczas obliczania zwracanej wartości. W takim przypadku zmiana stanu klasy nie zostanie wykryta. Implementowanie metody `hashCode` jest jednak zalecane, dlatego pomimo przedstawionych wad przyjmuje się, że dla większości przypadków przedstawiony algorytm da wyniki zgodne z oczekiwaniami.

8. Efektywność testowania programów obiektowych

W celu zbadania efektywności kryteriów pokrycia danych dla programów obiektowych zostały przeprowadzone pewne testy [7]. Dla trzech dużych systemów napisanych w języku C++ spośród wszystkich znanych błędów zostały wydzielone błędy związane z obiektowością, a więc głównie błędy spowodowane stosowaniem takich mechanizmów jak dziedziczenie oraz polimorfizm. Stanowiły one od 20% do 35% wszystkich błędów występujących w systemach. Następnie każdy system był testowany przy użyciu trzech strategii: I – testowanie przy użyciu kryteriów pokrycia instrukcji oraz pokrycia warunków, II – testowanie oparte na badaniu stanów poszczególnych klas oraz przejść pomiędzy stanami – metoda dokładniej opisana w [7], III – testowanie przy użyciu kryteriów *all-bindings* oraz *all-du-pairs*. Po przeprowadzeniu testów zmierzono jaki procent wszystkich znanych błędów udało się zlokalizować przy użyciu poszczególnych strategii. Dla błędów związanych z obiektowością wyniki są następujące – strategia I – 22%, strategia II – 44%, strategia III – 88%. Wyniki potwierdzają, iż klasyczne kryteria stosowane do programów proceduralnych słabo sprawdzają się w testowaniu programów obiektowych. Dla testowanych w tym eksperymencie programów użycie kryteriów specyficznych dla programów obiektowych pozwoliło zidentyfikować 4 razy więcej błędów niż użycie klasycznych kryteriów pokrycia instrukcji oraz pokrycia warunków.

9. Podsumowanie

Ze względu na bardzo dużą popularność programowania obiektowego konieczne jest dostosowanie do niego technik testowania. W tym artykule zaprezentowano różne podejścia do tego problemu oraz zaproponowano metodę testowania dla języka Java. Do tej pory powstała duża liczba narzędzi wspomagających testowanie dla programów w tym języku. Wiele z nich pozwala na mierzenie pokrycia kodu przez zestawy przypadków testowych. Można je podzielić na takie, które działają na podstawie kodu źródłowego, np.: PiSCES [10], TCAT/Java [11], JCover [12], JTest [13] oraz takie, które potrafią analizować programy już skompilowane do postaci plików `class`, np.: Emma [14], JaBUTi [1]. Większość narzędzi pozwala jednak jedynie na mierzenie pokrycia kodu związanego z przepływem sterowania. Spośród wymienionych jedynie JaBUTi pozwala mierzyć pokrycie przepływu danych. Ogranicza się ono jednak tylko do przepływu wewnątrz pojedynczej metody (*intra-method*). Jak wykazały badania przedstawione w rozdziale 8 użycie technik opisanych w tym artykule dało bardzo dobre efekty dla programów w języku C++, dlatego narzędzie dla Javy o podobnych możliwościach mogłoby okazać się bardzo przydatne. W ramach swojej pracy magisterskiej zamierzam podjąć próbę implementacji takiego narzędzia.

Literatura

1. JaBUTi Homepage - http://jabuti.incubadora.fapesp.br/ (dostęp: 12.2007)

2. S. Rapps, E.J. Weyuker, - „Selecting software test data using data flow information.”, IEEE Transactions on Software Engineering 11, 367–375, 1985

3. A.M.R. Vincenzi, J.C. Maldonado, W.E. Wong, M.E. Delamaro – „Coverage testing of Java programs and components”, Science of Computer Programming 56, 211-230, 2005

4. M.J. Harrold, G. Rothermel – „Performing data flow testing on classes”, Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering, 154-163, 1994

5. M.J. Harold, M.L. Soffa – „Interprocedural data flow testing", Procedings of the Third Testing, Analysis, and Verification Symposium, 158 - 167, 1989.

6. J. Zhao – „Dependence analysis of Java bytecode", 24th IEEE Annual International Computer Software and Applications Conference, COMPSAC'2000, IEEE Press, Taipei, Taiwan, 486–491, 2000.

7. Mei-Hwa Chen, H.M. Kao – „Testing Object-Oriented Programs An Integrated Approach", Proceedings of the 10th International Symposium on Software Reliability Engineering, 73-83, 1999

8. S.B. Lippman, J. Lajoie – „Podstawy języka C++", Wydawnictwa Naukowo-Techniczne, Warszawa, 2003

9. Java 2 Platform Standard Edition 5.0 API Specification - http://java.sun.com/j2se/1.5.0/docs/api/ (dostęp: 04.2008)

10. A. Binns, G. McGraw – „Building a Java software engineering tool for testing applets", IntraNet 96 NY Conference, New York, USA, 1996.

11. TCAT for Java/Windows—version 1.2, http://www.soft.com/ (dostęp 04.2008)

12. Java Code Coverage Analyzer – Jcover - http://www.mmsindia.com/JCover.html (dostęp 04.2008)

13. Parasoft JTest - http://www.parasoft.com (dostęp 04.2008)

14. EMMA: a free Java code coverage tool - http://emma.sourceforge.net/ (dostęp 04.2008)