

Верификация программ

Верификация – это установление соответствия между программой и ее спецификацией описывающей цель разработки, т.е. проверка корректности программы.

Выделяют 6 уровней верификации:

- 1) Отсутствие синтаксических ошибок (находит компилятор)
- 2) Отсутствие ошибок в операциях (обнаруживаются компьютером и ЭВМ при выполнении: деление на нуль, переполнение, заикливание и тп.)
- 3) Выдача корректных результатов на контрольных примерах (уровень студентов) – Но этого не достаточно!
- 4) Для рациональной области исходных данных программа выдает корректные результаты (обеспечивается тестированием)
- 5) Для всех допустимых наборов исходных данных, которые определены спецификацией задачи, программа выдает корректные результаты (достигается методом верификации)
- 6) Тотальная корректность. Программа выдает корректный результат на всех допустимых наборах исходных данных и сообщает об ошибках на всех недопустимых исходных данных

Уровни 5 и 6 труднодостижимы. Рассмотрим один из методов верификации, применимый для 5 уровня

Верификация предполагает аналитическое исследование свойств программы по ее тексту (без выполнения на ЭВМ) путем математического доказательства соответствия программы ее спецификации.

Для проведения доказательства строится некоторая формальная система, в которой формально определяются свойства корректности(правильности) программы

МЕТОД ВЕРИФИКАЦИИ НА ОСНОВЕ ПРОГРАММНЫХ ФУНКЦИЙ

Правильность программы определяется как соответствие между ее программной и ее заданной функцией.

Пусть задана функция f с областью определения $D(f)$ и программа P , составленная для выполнения P .

Тогда задача верификации программы P состоит в проверке из следующих факторов:

- программа P полностью правильна, если $f=[P]$, т.е.

$\forall x \in D(f): f(x)=[P(x)]$

$\forall x \notin D(f): [P(x)]$ – не определена

- программа P достаточно правильна, если $f \subset [P]$, т.е.

$\forall x \in D(f): f(x)=[P(x)]$

$\forall x \notin D(f): [P(x)]$ – программа вычисляет некоторые значения

ВЕРИФИКАЦИЯ СТРУКТУРИРОВАННЫХ ПРОГРАММ

Пусть P не элементарная программа. Используя ее структуру, можно разложить P на элементарные составляющие.

Декомпозиция ведет к снижению трудоемкости доказательства.

Например:

Имеется f и программа F для ее вычисления

$$F = (\text{if } P \text{ then } G \text{ else } H \text{ fi})$$

где G, H – программы.

Вместо прямой верификации равенства

$$f = [\text{if } p \text{ then } G \text{ else } H \text{ fi}] \quad (1)$$

доказательство проводится в два этапа.

- 1) доказываемость правильность программ G и H на основе предположении об их программных функциях g и h :

$$g = [G] \text{ и } h = [H]$$

- 2) если первый этап доказательства успешный, то используя аксиому замещения, можно упростить задачу (1) путем сведения ее к

$$f = [\text{if } p \text{ then } g \text{ else } h \text{ fi}] \quad (2)$$

Если (2) будет доказана, то тем самым будет доказана полная правильность, т.е.

$$f = [F].$$

Если любая из трех программ удовлетворяет только достаточной правильности, то и F будет лишь достаточно правильной.

Аксиома замещения: пусть P простая подпрограмма программы Q . Заменяем P некоторой простой подпрограммой P^* с такой же программной функцией $[P] = [P^*]$. Это приведет к получению из Q некоторой новой программы Q^* . По аксиоме замещения программная функция Q при этом не изменяется, т.е. $[Q] = [Q^*]$.

Резюме: Корректность структурированной программы можно свести к последовательной верификации элементарных подпрограмм, из которых она составлена.

ПРАВИЛЬНОСТЬ ЭЛЕМЕНТАРНЫХ ПРОГРАММ

Если верифицируемая программа P является ациклической, то доказательство может быть проведено непосредственным построением $[P]$ по Е-схеме и сравнением $[P]$ с заданной функцией f .

Если P является циклической, то при доказательстве правильности вначале необходимо установить неизбежность ее окончания.

Условимся, что если P заканчивается для $\forall x \in D(f)$, то некоторый предикат (обозначим его $\text{fin}(f, P)$) принимает значение «истина».

Верификацию циклических программ типа

while-do

do-until

do-while-do

можно свести к доказательству их окончания и к доказательству правильности ациклических рекурсивных программ.

Лемма о сведении итерационных программ к рекурсивным

Пусть даны функции f, g, h и предикат p . Тогда справедливы 3 случая:

1) while-do

$$(f = [\text{while } p \text{ do } g \text{ od}]) \leftrightarrow (\text{fin}(f, \text{while } p \text{ do } g \text{ od}) \wedge f = [\text{if } p \text{ then } g; f \text{ fi}])$$

2) do-until

$$(f = [\text{do } g \text{ until } p \text{ od}]) \leftrightarrow (\text{fin}(f, \text{do } g \text{ until } p \text{ od}) \wedge f = [g; \text{if } \neg p \text{ then } f \text{ fi}])$$

3) do-while-do

$$(f = [\text{do1 } g \text{ while } p \text{ do2 } h \text{ od}]) \leftrightarrow (\text{fin}(f, \text{do1 } g \text{ while } p \text{ do2 } h \text{ od}) \wedge f = [g; \text{if } p \text{ then } h; f \text{ fi}])$$

МЕТОДИКА ВЕРИФИКАЦИИ ЦИКЛИЧЕСКИХ ПРОГРАММ ТИПА WHILE-DO, DO-UNTIL, DO-WHILE-DO

1. Формулируется гипотеза о программной функции (в частном случае гипотеза может быть задана).
2. На основе субъективного анализа программы устанавливается истинность условия ограничения: $\text{fin}(f, P) = \text{true}$.
3. С использованием леммы о рекурсивном представлении программа приводится к рекурсивной форме.
4. С помощью трассировочных таблиц строится программная функция рекурсивной программы.
5. Проводится сравнение полученной функции с гипотезой и делается вывод о правильности программы.

Трассировочные таблицы

Они предназначены для формальной записи вывода программных функций.

| Номер операторов | Операторы программы | Элемент данных |
|------------------|---------------------|----------------|
| | | |

Трассировочные таблицы позволяют путем систематического исключения всех промежуточных значений индексов, начиная с конечных значений, вывести функцию программы.

Пример 1.

$P = (\text{if } x \leq 0 \text{ then } x := x - y$
 $\text{else } y := y - x \text{ fi};$
 $\text{if } y \leq 0 \text{ then } y := x + y$
 $\text{else } x := y - x \text{ fi})$

$[P] = ?$

В программе есть 4 пути от входа к выходу. Поэтому строим 4 трассировочные таблицы.

| № | Операторы Предикаты | Значение предикатов | Данные | |
|---|------------------------|------------------------------|-------------------|-------------------|
| | | | X | Y |
| 1 | $x \leq 0$ | $(x_0 \leq 0) = \text{true}$ | $x_1 = x_0$ | $y_1 = y_0$ |
| 2 | $x := x - y$ | | $x_2 = x_1 - y_1$ | $y_2 = y_1$ |
| 3 | $y \leq 0$ | $(y_2 \leq 0) = \text{true}$ | $x_3 = x_2$ | $y_3 = y_2$ |
| 4 | $y := x + y$ | | $x_4 = x_3$ | $y_4 = x_3 + y_3$ |

Их таблицы выводим программную функцию для выбранного пути

- предикат пути

$$(x_0 \leq 0) \wedge (y_2 \leq 0) = (x_0 \leq 0) \wedge (y_1 \leq 0) = (x_0 \leq 0) \wedge (y_0 \leq 0) = \text{True}$$

- данные

$$x_4 = x_3 = x_2 = x_1 - y_1 = x_0 - y_0$$

$$y_4 = x_3 + y_3 = x_2 + y_2 = x_1 - y_1 + y_1 = x_0$$

- функция

$$(x \leq 0) \wedge (y \leq 0) \rightarrow x, y := x - y, x$$

| № | Операторы Предикаты | Значение предикатов | Данные | |
|---|------------------------|-------------------------------|-------------|-------------------|
| | | | X | Y |
| 1 | $x \leq 0$ | $(x_0 \leq 0) = \text{false}$ | $x_1 = x_0$ | $y_1 = y_0$ |
| 2 | $y := y - x$ | | $x_2 = x_1$ | $y_2 = y_1 - x_1$ |
| 3 | $y \leq 0$ | $(y_2 \leq 0) = \text{true}$ | $x_3 = x_2$ | $y_3 = y_2$ |
| 4 | $y := x + y$ | | $x_4 = x_3$ | $y_4 = x_3 + y_3$ |

Вывод программной функции:

- предикат пути

$$(x_0 > 0) \wedge (y_2 \leq 0) = (x_0 > 0) \wedge ((y_1 - x_1) \leq 0) = (x_0 > 0) \wedge (y_0 \leq x_0) = \text{True}$$

- данные

$$x_4 = x_3 = x_2 = x_1 = x_0$$

$$y_4 = x_3 + y_3 = x_2 + y_2 = x_1 + y_1 - x_1 = y_0$$

- функция

$$(x > 0) \wedge (y \leq x) \rightarrow x, y := x, y$$

Аналогично выводятся программные функции для двух других путей.

Объединяя функции путей, получаем функцию программы:

$$[P] = (((x \leq 0) \wedge (y \leq 0) \rightarrow x, y := x - y, x) \vee ((x > 0) \wedge (y \leq x) \rightarrow x, y := x, y) \vee ((x \leq 0) \wedge (y > 0) \rightarrow x, y := 2y - x, y) \vee ((x > 0) \wedge (y > x) \rightarrow x, y := y - 2x, y - x))$$

Пример 2

Дана функция $f = (k \leq n \rightarrow x := x \cdot k \cdot (k+1) \cdot \dots \cdot n = (k \leq n \rightarrow x := x \cdot \prod_{i=k}^n i))$, где k и n – целые.

Для выполнения f составлена программа P

$P = (\text{while } k \leq n \text{ do } x := x \cdot k;$
 $\quad \quad \quad k := k + 1 \text{ od})$

Доказываем правильность P

1) Гипотеза о программной функции в данном случае задана

2) Если начальное значение k : $k > n$, то P завершается сразу; $k \leq n$, то в ходе итераций $k := k + 1$ и станет $k > n$, т.е. P завершится. Следовательно, $\text{fin}(f, P) = \text{истина}$.

3) Используем лемму для записи P в рекурсивной форме:

$P' = (\text{if } k \leq n \text{ then } x := x \cdot k;$
 $\quad \quad \quad k := k + 1;$
 $\quad \quad \quad (k \leq n \rightarrow x := x \cdot \prod_{i=k}^n i) \text{ fi})$

Для рекурсивной формы программы P' выводим программную функцию. Строим трассировочную таблицу.

| № | Операторы Предикаты | Значение предикатов | Данные | |
|---|--------------------------------|------------------------------|--------------------------------------|-----------------|
| | | | X | K |
| 1 | $k \leq n$ | $(k_0 \leq n) = \text{true}$ | $x_1 = x_0$ | $k_1 = k_0$ |
| 2 | $x := x \cdot k$ | | $x_2 = x_1 \cdot k_1$ | $k_2 = k_1$ |
| 3 | $k := k + 1$ | | $x_3 = x_2$ | $k_3 = k_2 + 1$ |
| 4 | $k \leq n$ | $(k_3 \leq n) = \text{true}$ | $x_4 = x_3$ | $k_4 = k_3$ |
| 5 | $x := x \cdot \prod_{i=k}^n i$ | | $x_5 := x_4 \cdot \prod_{i=k_4}^n i$ | $k_5 = k_4$ |

Вывод функции для выбранного пути:

-предикат

$$(k_0 \leq n) \wedge (k_3 \leq n) = (k_0 \leq n) \wedge (k_2 + 1 \leq n) = \dots = (k_0 \leq n) \wedge (k_0 + 1 \leq n) = (k_0 \leq n - 1) = \text{True}$$

-данные

$$x_5 = x_4 \cdot \prod_{i=k_4}^n i = \dots = x_2 \cdot \prod_{i=k_2+1}^n i = x_1 \cdot k_1 \cdot \prod_{i=k_1+1}^n i = x_0 \cdot k_0 \cdot \prod_{i=k_0+1}^n i = x_0 \cdot \prod_{i=k_0}^n i$$

-функция

$$(k \leq n - 1) \rightarrow x := x \cdot \prod_{i=k}^n i$$

| № | Операторы Предикаты | Значение предикатов | Данные | |
|---|------------------------|-------------------------------|-----------------------|-----------------|
| | | | X | K |
| 1 | $k \leq n$ | $(k_0 \leq n) = \text{true}$ | $x_1 = x_0$ | $k_1 = k_0$ |
| 2 | $x := x \cdot k$ | | $x_2 = x_1 \cdot k_1$ | $k_2 = k_1$ |
| 3 | $k := k + 1$ | | $x_3 = x_2$ | $k_3 = k_2 + 1$ |
| 4 | $k \leq n$ | $(k_3 \leq n) = \text{false}$ | $x_4 = x_3$ | $k_4 = k_3$ |

Вывод функции для выбранного пути:

-предикат

$$(k_0 \leq n) \wedge (k_3 > n) = (k_0 \leq n) \wedge (k_2 + 1 > n) = \dots = (k_0 \leq n) \wedge (k_0 + 1 > n) = (k_0 = n) = \text{True}$$

-данные

$$x_4 = x_3 = x_2 = x_1 \cdot k_1 = x_0 \cdot k_0 = x_0 \cdot n$$

-функция

$$((k = n) \rightarrow x := x \cdot n) = ((k = n) \rightarrow x := x \cdot \prod_{i=k}^n i)$$

| № | Операторы Предикаты | Значение предикатов | Данные | |
|---|------------------------|-------------------------------|-------------|-------------|
| | | | X | K |
| 1 | $k \leq n$ | $(k_0 \leq n) = \text{false}$ | $x_1 = x_0$ | $k_1 = k_0$ |

Вывод функции для выбранного пути:

-предикат

$$(k_0 > n) = \text{True}$$

-данные

$$x_1 = x_0$$

-функция

$$(k > n) \rightarrow x := x$$

Объединяем функции путей, получаем функцию программы

$$[P'] = ((k \leq n - 1) \rightarrow x := x \cdot \prod_{i=k}^n i) \vee ((k = n) \rightarrow x := x \cdot \prod_{i=k}^n i) \vee ((k > n) \rightarrow x := x) = (k \leq n) \rightarrow$$

$$x := x \cdot \prod_{i=k}^n i = f$$

Отсюда следует, что составленная программа обладает полной правильностью.

ВЕРИФИКАЦИЯ ПРОГРАММЫ FOR-DO

Общий вид:

$P = (\text{for } i: \in L(1:n) \text{ do } g \text{ od})$

Рассмотрим вывод программной функции.

Если число элементов в индексном списке $L(1:n)$ мало и заранее известно, то функцию можно получить по трассировочной таблице.

Если n велико или не задано, то построение трассировочной таблицы невозможно.

В этом случае необходимо сформулировать гипотезу о программной функции и доказать ее методом математической индукции.

Индукция может производиться по верхнему значению параметра n цикла или по числу элементов в списке цикла.

Пример.

$P = (\text{for } i: \in 1 \text{ to } n \text{ by } 1 \\ \quad \text{do } x := x \cdot i; \text{ od})$

Сформулируем гипотезу о программной функции $[P]$. Трассировочная таблица:

| № | Шаг | Операторы программы | Данные | |
|---------------------------------------|-----|---------------------|-------------|-----------------------|
| | | | i | x |
| 1 | 1 | $i := 1$ | $i_1 = 1$ | $x_1 = x_0$ |
| | 2 | $x := x \cdot i$ | $i_2 = i_1$ | $x_2 = x_1 \cdot i_1$ |
| Программная функция: $x := x \cdot 1$ | | | | |

| № | Шаг | Операторы программы | Данные | |
|--|-----|---------------------|-------------|-----------------------|
| | | | i | x |
| 2 | 1 | $i := 1$ | $i_1 = 1$ | $x_1 = x_0$ |
| | 2 | $x := x \cdot i$ | $i_2 = i_1$ | $x_2 = x_1 \cdot i_1$ |
| | 3 | $i := 2$ | $i_3 = 2$ | $x_3 = x_2$ |
| | 4 | $x := x \cdot i$ | $i_4 = i_3$ | $x_4 = x_3 \cdot i_3$ |
| Программная функция: $x := x \cdot 1 \cdot 2$... | | | | |

Из анализа этой таблицы следует предположение:

$$f(x) = (x := x \cdot n!)$$

Теперь проведем доказательство методом математической индукции по верхней границе списка цикла n .

При $n=1$ имеем программу:

$P_1 = \text{for } i: \in 1 \text{ to } 1 \text{ by } 1 \text{ do } x := x \cdot i \text{ od}$

Из трассировочной таблицы следует

$$[P_1] = x \cdot 1 = x \cdot 1!$$

Т.е. высказанная гипотеза справедлива. Пусть теперь гипотеза справедлива при $n=k$, т.е. для

$P_k = \text{for } i: \in 1 \text{ to } k \text{ by } 1 \text{ do } x := x \cdot i \text{ od}$

имеет место

$$[P_k] = (x := x \cdot k!)$$

Докажем, что

$$[P_{k+1}] = (x := x \cdot (k+1)!)?$$

Программа P_{k+1} имеет вид:

$P_{k+1} = \text{for } i: \in 1 \text{ to } k+1 \text{ by } 1 \text{ do } x := x \cdot i \text{ od}$

Представим ее как составную программу:

$P_{k+1} = (\text{for } i: \in 1 \text{ to } k \text{ by } 1 \text{ do } x := x \cdot i \text{ od};$
 $x := x \cdot (k+1))$

Первый оператор этой программы есть программа P_k , ее функция по предложению равна $x := x \cdot k!$. Тогда по аксиоме замещения можно записать

$P_{k+1} = (x := x \cdot k!; x := x \cdot (k+1))$

Для этой программы легко получить программную функцию

$[P_{k+1}] = (x := x \cdot k! \cdot (k+1)) = (x := x \cdot (k+1)!)$

Следовательно, высказанная гипотеза программной функции верна.

МЕТОД ИНДУКТИВНЫХ УТВЕРЖДЕНИЙ

Самый распространенный метод.

Он полужормальный. Он разработан Флойдом и Науром в 1966-67 гг. Составляет основу большинства автоматических систем верификации.

Он представляет формулировку и доказательство ряда теорем (называемых условиями верификации).

Метод состоит из ряда шагов.

1 ШАГ.

Необходимо записать утверждения о входных ($P(\bar{x})$) и выходных результатах программы $Q(\bar{x}, \bar{y})$.

Эти утверждения обычно формулируются в некоторой формально-логической системе. Как правило это исчисление предикатов первого порядка.

Пример

Задача: $Z = A^B$, $B > 0$ и целое число.

Алгоритмы (2 способа):

- 1) $Z := A * A * A * \dots * A$
- 2) Вычисляется A^{2^n} , $n=1,2,4,\dots$ A, A^2, A^4, A^8, \dots

Отбираются нужные степени, используя двоичное представление B .

Если $B = 13_{10} = 1101_2$ тогда $Z = A^8 * A^4 * A^1$.

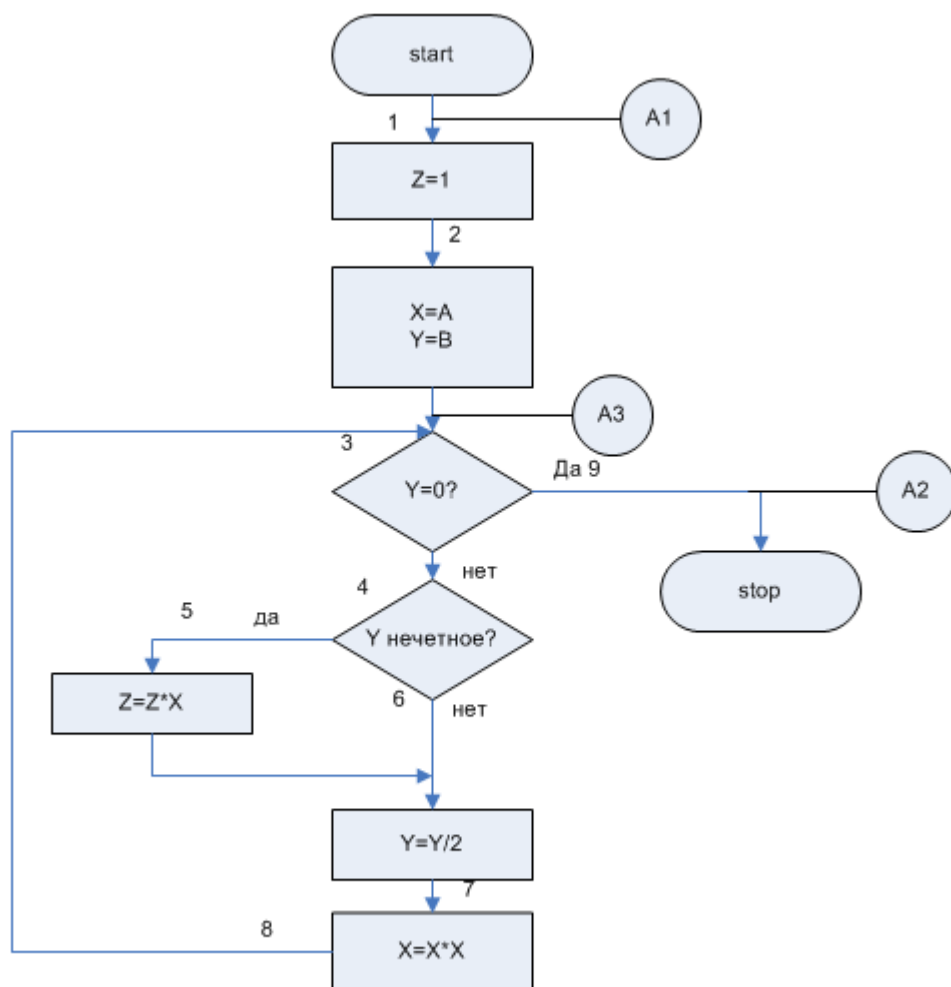
Программа затратит 7 умножений вместо 13.

Входное утверждение относится к точке $A1$, а выходное к точке $A2$.

Первое должно описывать все необходимые входные условия для программы, а второе – ожидаемый результат.

$A1: (B \geq 0) \wedge (B \text{-целое})$

$A2: (Z = A^B) \wedge (A, B \text{ не изменены})$



2 ШАГ.

Формулируются теоремы (или условия верификации), подлежащие доказательству.

Способ 1

Предположим, что A1 истинно в т. 1, далее воспользуемся семантикой оператора между т. 1 и 2 для преобразования A1 в промежуточное утверждение в т.2.

Так продолжаем, пока промежуточное утверждение, выведенное из A1, не будет построено для выхода – т.9.

Таким образом для т. 9 имеется 2 утверждения :

первое – выведенное,

второе – A2, т.е. выходное утверждение.

Теперь можно сформулировать теорему:

Из выведенного в точке 9 утверждения логически следует A2.

Способ 2

Начинаем с выходного утверждения и движемся по программе в обратном направлении до тех пор, пока не получится выведенное утверждение для т. 1. теперь теорема формулируется так:

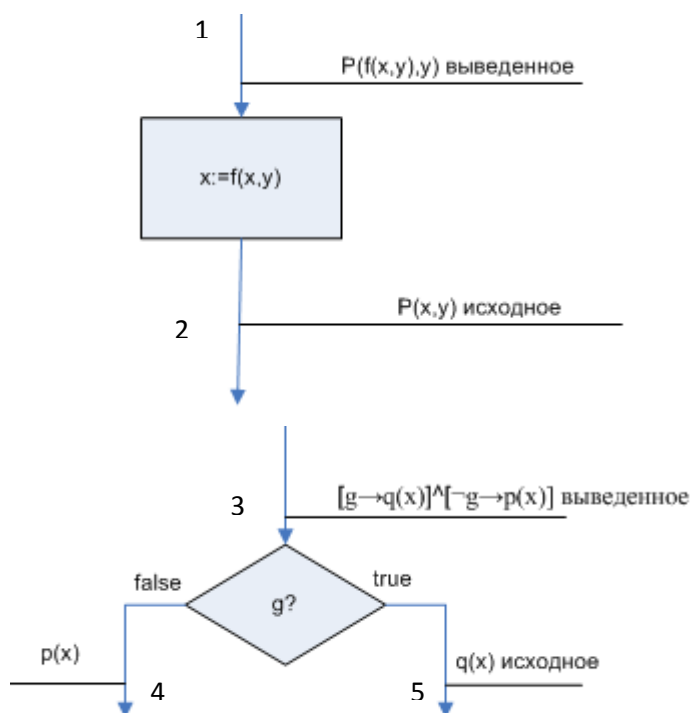
Из A1 следует утверждение, выведенное для т.1.

Для формулировки условий верификации необходимо иметь аксиоматическое описание семантики языка программирования.

В нашей программе (см. схему) встречаются операторы двух типов: присваивания и ветвления. Поэтому необходимо рассмотреть, как должны изменяться утверждения при переходе через эти операторы.

Отдадим предпочтение обратному методу формулировки теорем, т.к. он дает новую точку зрения на выполнение программы.

Аксиомы преобразования утверждений при обратном просмотре будут иметь следующий вид:



Эти аксиомы должны отвечать на вопрос: если некоторое утверждение истинно после оператора программы, то какое утверждение должно быть истинным перед оператором?

Например.

В т.2 $p(x, y): (y = x + 4) \wedge (x \geq 0)$

Оператор: $x := x - 2$

В т. 1 будет выведено утверждение $(y = x + 2) \wedge (x \geq 2)$

Резюме: следует отменить присваивание и заменить в исходном утверждении все вхождения переменной, указанной слева от $:=$, выражением справа.

Теперь может показаться, что у нас достаточно информации для построения условий верификации. Но есть одна проблема: Как быть с циклом?

Число повторений – различное (степень В). Поэтому необходимо третье утверждение для точки внутри цикла (называется индуктивным утверждением или инвариантом цикла). Тогда можно применить для доказательства правильности метод математической индукции.

Индуктивное утверждение должно описывать все свойства программы, которые инвариантны для выбранной точки в цикле.

Для нашей программы инвариант в т.3 будет такой:

$$A3: (Z * X^y = A^B) \wedge (Y \geq 0) \wedge (Y - \text{целое}) \wedge (A, B \text{ не изменены})$$

Общеизвестно, что это самый трудный шаг в процессе доказательства. Теперь мы имеем 3 утверждения (может быть больше, если есть несколько циклов).

Вместо построения одного подлежащего доказательству условия верификации мы построим несколько таких условий. Для этого рассмотрим граф программы и все утверждения.

Выделим в программе все пути, которые начинаются и заканчиваются утверждениями и не содержат утверждений внутри. Такие пути называются выделенными путями. Для нашего примера:

1. от A1 к A3
2. от A3 к A2
3. от A3 к A3 для Y нечетного
4. от A3 к A3 для Y четного

Для каждого пути должны быть построены условия верификации, их доказательство подтверждает правильность программы.

Для построения условия верификации (теоремы) следует взять утверждение в конце выделенного пути и трансформировать его, возвращаясь к началу.

Рассмотрим третий выделенный путь. Утверждение в т.8 совпадает с утверждением в т.3, которое является A3. Между точками 3 и 8 нет операторов.

В т.7, заменяя X на $X * X$, получаем

$$(Z * X^{2y} = A^B) \wedge (Y \geq 0) \wedge (Y - \text{целое}) \wedge (A, B \text{ не изменены})$$

В т.6 заменяя Y на $Y/2$ имеем

$$(Z * X^{2(y/2)} = A^B) \wedge (Y/2 \geq 0) \wedge (Y/2 - \text{целое}) \wedge (A, B \text{ не изменены})$$

Двойки в показателе степени не сокращаются, потому что целое деление в машине – не то же самое, что в математике:

$$2 * (Y/2) \text{ не всегда } = Y$$

В т.5 выведенное утверждение получает вид

$$(Z * X^{1+2(y/2)} = A^B) \wedge (Y/2 \geq 0) \wedge (Y/2 - \text{целое}) \wedge (A, B \text{ не изменены})$$

В т.4 имеем

$$(Y \text{ нечетно}) \rightarrow [\text{утверждение для т.5}]$$

В т.3 имеем

$$(Y \neq 0) \rightarrow \{ (Y \text{ нечетно}) \rightarrow [\text{утверждение для т.5}] \}$$

Итак, мы получили условие верификации, согласно которому утверждение A3 влечет за собой написанное выше утверждение.

3 других условия верификации могут быть получены аналогично.

4 условия верификации:

$$1. [(B \geq 0) \wedge (B - \text{целое})] \rightarrow [(1 * A^B = A^B) \wedge (B \geq 0) \wedge (B - \text{целое}) \wedge (A, B \text{ не изменены})]$$

$$2. [(Z * X^y = A^B) \wedge (Y \geq 0) \wedge (Y - \text{целое}) \wedge (A, B \text{ не изменены})] \rightarrow \{ (Y = 0) \rightarrow [(Z = A^B) \wedge (A, B \text{ не изменены})] \}$$

$$3. [(Z * X^y = A^B) \wedge (Y \geq 0) \wedge (Y - \text{целое}) \wedge (A, B \text{ не изменены})] \rightarrow \{ (Y \neq 0) \rightarrow \{ (Y \text{ нечетно}) \rightarrow [(Z * X^{1+2(y/2)} = A^B) \wedge (Y/2 \geq 0) \wedge (Y/2 - \text{целое}) \wedge (A, B \text{ не изменены})] \} \}$$

$$4. [(Z * X^y = A^B) \wedge (Y \geq 0) \wedge (Y - \text{целое}) \wedge (A, B \text{ не изменены})] \rightarrow \{ (Y \neq 0) \rightarrow \{ (Y \text{ четно}) \rightarrow [(Z * X^{2(y/2)} = A^B) \wedge (Y/2 \geq 0) \wedge (Y/2 - \text{целое}) \wedge (A, B \text{ не изменены})] \} \}$$

ШАГ 3.

Доказательство всех 4ех утверждений.

Для этого можно воспользоваться правилами вывода в исчислении предикатов. Но обычно достаточно неформальных рассуждений.

Все 4 теоремы имеют вид

$$p \rightarrow q$$

Обычно метод доказательства таких теорем сводится к тому, чтобы показать, что q истинно каждый раз, когда истинно p .

Рассмотрим доказательство 3-его условия (оно посложнее остальных).

Утверждение для $Y/2$, A и B очевидны. Остается показать, что

$$(Z * X^Y = A^B) \rightarrow (Z * X^{1+2(Y/2)} = A^B), \text{ когда } Y - \text{нечетно.}$$

Условимся, исходя из действующего в большинстве языков соглашения, что округление до целого производится с недостатком, если результат деления – дробный: $3/2=1$.

Поэтому $2(Y/2)=Y-1$, т.к. Y -нечетно. Следовательно, $Y=1+Y-1$.

Доказательство окончено.

4ое доказательство проводится аналогично.

Мы показали правильность программы по отношению к входному и выходному утверждению. Для частичной корректности необходимо доказать, что программа не циклится.

Общего метода доказательства завершаемости программы нет, но бывает достаточно неформальных рассуждений.

В нашем примере программа заканчивается только тогда, когда $Y=0$.

Согласно входному утверждению, Y первоначально неотрицательно ($Y \geq 0$). Он изменяется в одном месте ($Y:=Y/2$), т.е уменьшается при каждом повторении цикла:

$$B, B/2, B/4, \dots, 1, 0.$$

Т.о. неформально доказывается завершаемость программы для $\forall B (B \geq 0)$.

Этапы доказательства методом индуктивных утверждений

1. Постройте блок-схему программы
2. Напишите входные и выходные утверждения
3. Найдите все циклы и сформулируйте индуктивные утверждения для каждого из них
4. Составьте список выделенных путей
5. Используя семантику операторов программы, постройте условия верификации
6. Докажите каждое условие верификации. Если это не удастся, то либо Вы недостаточно изобретательны, либо одно из ваших утверждений неполно, либо Вы обнаружили ошибку на выделенном пути
7. Докажите, что выполнение программы закончится

Реплики:

1. Может показаться, что этот метод рассчитан на численные программы – это не так!
2. Скептики видят в этом методе причудливую маскировку обычных процессов чтения текста программы или сквозного контроля.

Степень формализации!

3. Доказательство бывает длиннее самой программы.

Нужны верификаторы

4. А как быть с модульными программами (процедурами)?
Необходимо доказать правильность модуля за модулем (снизу вверх).
Прежде всего определить входные и выходные утверждения для каждого модуля. Когда внутри модуля встречается вызов другого модуля(процедуры), тогда используют утверждения в качестве семантики оператора вызова.

Недостатки метода:

1. Программы, правильность которых доказана, могут содержать ошибки.
2. Доказательства сложны и утомительны.
3. Некоторые конструкции языков программирования не поддаются доказательству.
4. Правильность доказывается только по отношению к входным и выходным утверждениям.
5. Доказательство ничего не говорит о поведении программы в случае, если реальные входные данные не удовлетворяют входным утверждениям.
6. Ошибки в многомодульных интерфейсах обычно не обнаруживаются при доказательстве.
7. Само доказательство может быть не правильным.
8. Можно неправильно интерпретировать семантику языка.
9. Системные и машинные ограничения(ошибки условного округления, переполнения) обычно опускаются в доказательствах.

Достоинства:

1. При верификации могут быть обнаружены новые ошибки.
2. Попытка доказать правильность заставляет программиста очень детально исследовать программу и спецификации, а также формализовать свое понимание программы.
3. Доказательство приносит и косвенную пользу (возникает стремление к точности программирования).
4. Математические доказательства оказывают влияние на разработку новых языков программирования.

Тестирование программ

Тестирование (испытание) – это установление соответствия программы заданным требованиям и программным документам.

Испытание программы является одним из этапов работ общего процесса разработки программы. Цель тестирования состоит в выявлении наличия ошибок в программе. Тестирование способно обнаружить ошибки, но не способно доказать их отсутствие. Затраты на тестирование достигают 50% от общих затрат на разработку программы.

Тестирование определяет качество программного продукта.

Кроме того, тестирование определяет стоимость и длительность разработки ПО.

Резюме: Необходимо создавать методы и средства, позволяющие достигать максимального качества программ при реальных ограничениях на длительность тестирования и связанные с этим затраты. Тестирование – проблема экономическая!

Тестирование – необычный процесс. Этот процесс разрушительный, т. к. цель тестовика- заставить программу ошибаться.

Этапы тестирования

-проектирование тестов

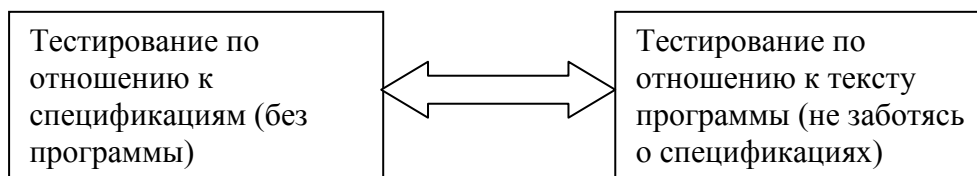
Тест – это набор значений входных данных и соответствующий им набор значений всех выходных данных.

-выполнение тестов программой на компьютере

-анализ результатов тестирования

Главное – это проектирование тестов!!!

Имеется широкий спектр подходов к проектированию тестов.



Сторонники левой крайности: проверяют все возможные комбинации значений на входе программы. Сторонники правой крайности: проверяют каждый путь, каждую ветвь программы.

Методы тестирования модулей

Основные подходы (стратегии) к проектированию тестов:

-ФУНКЦИОНАЛЬНЫЙ (принцип «черного ящика»)

-СТРУКТУРНЫЙ (принцип «белого ящика»)

Функциональные методы тестирования базируются на рассмотрении программы как черного ящика. При этом внутренняя структура программы не учитывается. Тесты составляются по внешним спецификациям программы.

В структурных методах для построения тестов используется информация о структуре (блок-схемы) программы.

ФУНКЦИОНАЛЬНЫЕ МЕТОДЫ ТЕСТИРОВАНИЯ

МЕТОД ФУНКЦИОНАЛЬНЫХ ДИАГРАММ

Основывается на использовании спецификации программы, написанных на естественном языке.

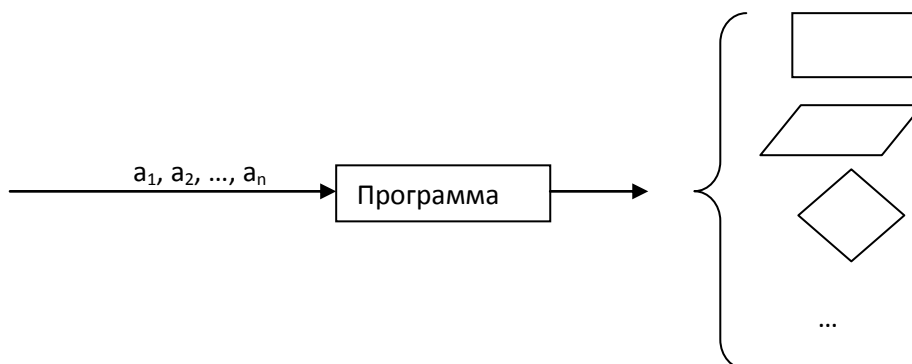
Метод дает неплохие тесты, не страдающие избыточностью. Кроме того, обнаруживается неполнота и неоднозначность в исходных данных и спецификациях.

Идея метода:

Метод предполагает анализ содержания внешних спецификаций и перевод их на язык логических отношений между входными данными(ситуациями) и выходными данными(эффектами), представленными в форме логической диаграммы(«и-или» графа) называемой функциональной диаграммой.

ПРИМЕР ИСПОЛЬЗОВАНИЯ

Программа: вводит 8 чисел, первые 4 – длины сторон четырехугольника, а остальные – углы. программа передает сообщение о том, какой этот четырехугольник: квадрат, ромб, прямоугольник, параллелограмм.



ситуация- это условие или комбинация условий, которые могут возникнуть на входе

эффект –это есть видимый результат(действие)

| СИТУАЦИЯ | ЭФФЕКТ |
|--|---|
| 1. Сумма углов !=360 | Напечатать сообщение «Это не 4х угольник» |
| 2. Условие 1 не выполняется, все углы = 90 | Напечатать сообщение «Это квадрат» |
| ... | ... |

Шаги метода

1 шаг. Тщательный анализ спецификаций. Определение причин и следствий.

Причины:

1. сумма углов =360
2. все углы = 90
3. все стороны равны между собой
4. противоположные стороны равны
5. противоположные углы равны

Следствия:

- 91. Печатать «Это не 4угольник»
- 92. «Прямоугольник»
- 93. «Квадрат»
- 94. «Параллелограмм»
- 95. «Ромб»
- 96. «Разносторонний 4угольник»

2 шаг. Изобразить функциональную диаграмму («и-или» граф)

Элементы диаграммы:

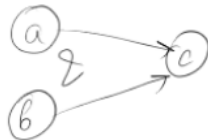
Если а то b:



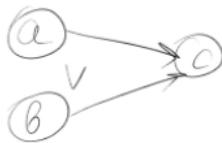
Если не а то b:



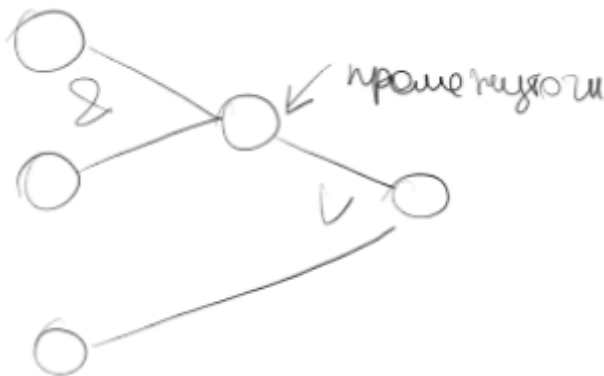
Если а и b то c:



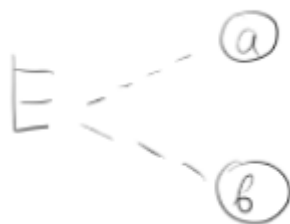
Если а или b то c:



Можно использовать промежуточные вершины:

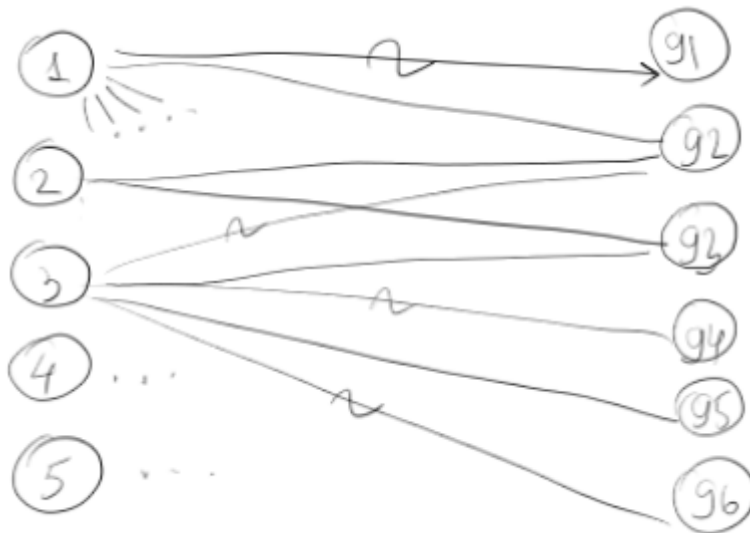


Ограничение. а и b не могут одновременно иметь место. Е – тип ограничения



Причины

Следствия



Шаг 3. Генерация таблицы решений с ограниченным входом

| | 1 | 2 | 3 | 4 | 5 | 6 |
|----|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 1 | 1 | 1 |
| 2 | | 1 | 1 | | | |
| 3 | | 0 | 1 | 0 | 1 | 0 |
| 4 | | 1 | | 1 | | |
| 5 | | | | 1 | 1 | |
| 91 | 1 | 0 | 0 | 0 | 0 | 0 |
| 92 | 0 | 1 | 0 | 0 | 0 | 0 |
| 93 | 0 | 0 | 1 | 0 | 0 | 0 |
| 94 | 0 | 0 | 0 | 1 | 0 | 0 |
| 95 | 0 | 0 | 0 | 0 | 1 | 0 |
| 96 | 0 | 0 | 0 | 0 | 0 | 1 |

1 – причина имеет место

0-причина отсутствует

Шаг 4. Преобразование таблицы решений в тесты

Тест 1: 10, 20, 75, 40, 120, 90, 70, 200

Тест2: 25, 15, 25, 15, 90, 90, 90, 90

Тест3: 13, 13, 13, 13, 90, 90, 90, 90

Тест4: 20, 10, 20, 10, 60, 120, 60, 120

Тест5: 40, 40, 40, 40, 80, 100, 80, 100

Тест6: 80, 21, 50, 70, 85, 100, 95, 80

АНАЛИЗ МЕТОДА:

Функциональная диаграмма может приводить к большому числу тестов. Надо проставить приоритеты к тестам и отбросить тесты с низким приоритетом.

Искусство тестирования в том и состоит, чтобы знать, когда остановиться.

Недостатки:???

хорош для задач у которых множество комбинаций на входе???

МЕТОД ЭКВИВАЛЕНТНОГО РАЗБИЕНИЯ

Исходные данные – внешние спецификации программы.

Цель – разработать как можно меньше тестов, но образующих полный набор для всесторонней проверки программы.

Идея:

Если набор тестов обнаруживает один тип ошибок в программе, то этот набор определяет один класс, и в разрабатываемый набор тестов достаточно включить 1 тест из класса.

$$X = \left\{ \begin{array}{l} x1 \\ x2 \\ x3 \\ x4 \\ x5 \\ x6 \\ \vdots \\ xn \end{array} \right\} \begin{array}{l} T3 \\ T2 \\ Tn \end{array} \Rightarrow \text{Программа} \Rightarrow Y$$

Этапы разработки тестов:

- 1) Выделение классов эквивалентности
- 2) Составление тестов

I Этап. Классы эквивалентности записываются в виде таблицы

| Функциональные хар-ки программы | Правильные классы эквивалентности | Неправильные классы эквивалентности |
|------------------------------------|--------------------------------------|--|
| | | |

Функциональные характеристики программы – все ф-ии программы, условия, свойства, описывающие входные данные

Правильные классы - входные данные, удовлетворяющие этим характеристикам.

Неправильные классы – входные данные, не удовлетворяющие этим характеристикам, т.е. все другие данные.

Выделенные классы произвольно нумеруются

Рекомендации по выделению классов:

1) Если характеристика задает область значений (например, переменная Z меняется от 100 до 500), то выделяется один прав класс($100 \leq Z \leq 500$) и два неправ класса($Z < 100$, $Z > 500$)

2) Если характеристика описывает множество значений, каждое из которых программа обрабатывает по своему (напр «строчная переменная S задает метод решения и принимает значение МГАУСС, МПИ, МПН»), то определяется правильный класс эквивалентности каждого элемента этого множества и один неправильный класс (например, «S принимает значение ABC»)

3) Если характеристика описывает ситуацию типа «должно быть» (напр, «имя файла должно начинаться с букв NC»), то определяется 1 прав класс эквивалентности (имя начинается с NC) и один неправ класс(имя не с NC)

II Этап. Построение тестов

| Для правильных классов | Для неправильных классов |
|---|--|
| Составляются тесты, покрывающие одновременно максимально возможное число правильных классов. Полученное множество тестов должно покрывать все правильные классы | Для каждого класса составляется отдельный тест |

ПРИМЕР

Протестировать программу осуществляющую синтаксическую проверку записи идентификатора.

Внешние спецификации: Идентификатор – это строка, состоящая из букв и цифр, начинающаяся с буквы или символа #, длиной не более 8 символов. Идентификатор не может состоять из одного символа #.

Выделяем классы эквивалентности

| Функциональные хар-ки программы | Правильные классы эквивалентности | Неправильные классы эквивалентности |
|--|--|-------------------------------------|
| Символы идентификатора, кроме первого | Буквы(1) Цифры(2) Буквы и цифры(3) | Другие символы(4) |
| Первый символ | Буква(5) #(6) | Другие символы(7) |
| Длина идентификатора | ≤ 8 (8) | > 8 (9) |
| Длина идентификатора, начальный символ # | $1 < \text{длина} \leq 8$ (10) | 1 (11) |

Тесты, покрывающие правильные классы эквивалентности

| Тесты | Номера классов |
|-------|----------------|
| NAME | 1,5,8 |
| #123 | 2,6,10 |
| X2 | 3 |

Тесты, покрывающие неправильные классы эквивалентности

| Тесты | Номера классов |
|--------------|----------------|
| N\$Me | 4 |
| \$ad1 | 7 |
| Var123456789 | 9 |
| # | 11 |

Для проверки программы получено 7 тестов

МЕТОД АНАЛИЗА ГРАНИЧНЫХ УСЛОВИЙ (ЗНАЧЕНИЙ)

Метод прост и дает хорошие тесты

Идея: анализ граничных условий! Граничные условия – это ситуации, возникающие непосредственно НА, ВЫШЕ и НИЖЕ границ входных и выходных классов эквивалентности.

1. Тесты разрабатываются так, чтобы проверялась каждая граница класса эквивалентности.

2. Рассматриваются не только входные условия (пространство входов), но и пространство выходных данных, т.е. входные классы эквивалентности.

Применение метода тем успешнее, чем больше изобретательности (творчества, интеллекта) проявит программист.

Следует применять следующие рекомендации:

1. Если функциональная характеристика задает область значений входных данных, то надо построить тесты с неправильными входными данными

$-1.0 \leq X \leq +1.0$

будет 4 теста: $x = -1.0$ $x = +1.0$ $x = -1.01$ $x = +1.01$

2. Использование правила 1 для каждой выходной характеристики.

Например: Программа начисляет зарплату Z

$0 \leq Z \leq 10000 \text{ грн}$

Тесты:

T1 = {Исходные данные, вызывающие начисление зарплаты 0 грн}

T2 = {Исходные данные, вызывающие начисление зарплаты 10000 грн}

T3 = {Исходные данные, вызывающие начисление отрицательной зарплаты}

T4 = {Исходные данные, вызывающие начисление зарплаты больше 10000 грн}

Не всегда можно построить тесты, выводящие выходные данные за допустимые границы

3. Если входные и выходные данные программы – упорядоченное множество (линейный список, таблица), то надо разрабатывать тесты, связанные с 1ым и последним элементами этого множества.

ПРИМЕР 1

Протестировать программу сортировки.

Функциональные характеристики программы:

- обращение к программе имеет вид $\text{SORT}(A, B, N, K)$

- идентификатор A – имя исходного неупорядоченного массива целых чисел

- Идентификатор B – имя массива, в котором расположены упорядоченные элементы

- размерность массива A и B равна N (целая переменная или константа в диапазоне от 0 до 1000)

- целая переменная или константа K указывает вид сортировки

K=0 – массив A по возрастанию

K=1 – по убыванию

Проведем анализ граничных условий!

Первое граничное условие связано с N(размерностью массива). Из него получаем 4 теста

| № теста | Тестовые данные | | |
|---------|-----------------|---|------------------------------|
| | N | K | A |
| 1 | 1 | 0 | Любой неуп массив A[1] |
| 2 | 10000 | 1 | Любой неуп массив A[1..1000] |
| 3 | 0 | 1 | Любой неуп массив A |
| 4 | 1001 | 1 | Любой неуп массив A[1..1001] |

Второе граничное условие связано с множеством значений K. из него получают тесты 2,5,6

| № теста | Тестовые данные | | |
|---------|-----------------|---|----------------------------|
| | N | K | A |
| 5 | 10 | 0 | Любой неуп массив A[1..10] |
| 6 | 5 | 2 | Любой неуп массив A[1..5] |

Третье граничное условие связано с перемещением всех элементов при сортировке (тесты 7,8) и с отсутствием перемещения элементов (9,10,11,12)

| № теста | Тестовые данные | | |
|---------|-----------------|---|---|
| | N | K | A |
| 7 | 5 | 0 | Упоряд по убыванию, элементы – различны |
| 8 | 5 | 1 | Упоряд по возраст, элементы – различны |
| 9 | 5 | 0 | Элементы одинаковы |
| 10 | 5 | 1 | Элементы одинаковы |
| 11 | 5 | 0 | Упоряд по возраст |
| 12 | 5 | 1 | Упоряд по убыванию |

ПРИМЕР 2

Протестировать программу, которая распознает вид 4угольника.

Из анализа выявлены следующие граничные условия.

1: $a_i > 0$

2: $0 < \alpha_i < 180$

3: Число ввода чисел = 8

Появляются следующие новые тесты:

-одна или несколько сторон = 0

- все стороны > 0 (уже покрывается)

- один или неск углов = 0

- все углы $0 < \alpha_i < 180$ (покрывается уже)

- один или неск углов ≥ 180

-число вводимых чисел = 8(уже покрывается)

- число вводимых чисел > 8

Получаем тесты:

Тест 7: 0,20,30,50,80,100,30,150

Тест8: 10,20,10,20,0,140,60,160

Тест9: 10,50,30,50,180,100,40,40,70

Тесты10: 10,10,10,10,90,90,90,10

Резюме

1. Метод анализа граничных значений – один из самых полезных методов.
2. Однако граничные условия могут быть едва улавливаемые, следовательно, их обнаружение связано с большими трудностями.

СТРУКТУРНЫЕ МЕТОДЫ ПРОЕКТИРОВАНИЯ ТЕСТОВ

Структурные методы базируются на применении “белого ящика” т.е используют известную логическую структуру программы.

В основе этого класса методов лежит некоторый критерий обеспечивающий соответствующую полноту тестирования программы.

МЕТОД ПОКРЫТИЯ ОПЕРАТОРОВ

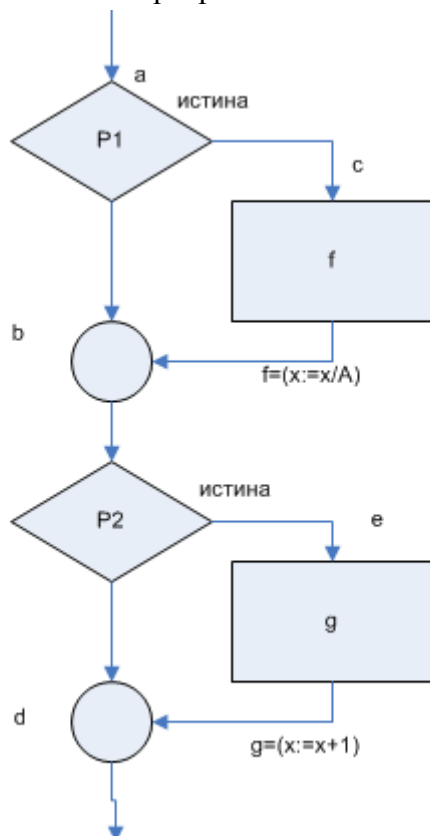
Критерий метода: каждый оператор программы реализуется хотя бы один раз.

Это слабый критерий

Он является необходимым, но не достаточным условием для хорошего тестирования.

Рассмотрим формульную методику построения тестов.

ПРИМЕР программы



$$P1=(A>1)\&(B=0)$$

$$P2=(A=2)\vee(X>1)$$

Критерию покрытия операторов удовлетворяет путь aсed

Найдем предикаты пути, т.е. условия прохождения этого пути.

Применим трассировочную таблицу.

| № | Оператор | Значение предиката | Данные |
|---|----------|------------------------|------------|
| 1 | P1 | $P1(X0)=\text{истина}$ | $X1=X0$ |
| 2 | F | | $X2=f(x1)$ |
| 3 | P2 | $P2(x2)=\text{истина}$ | $X3=X2$ |
| 4 | g | | $X4=g(X3)$ |

Предикат пути:

$P1(X0) \& P2(X2) = \text{true}$

$P1(X0) \& P2(f(x1)) = P1(X0) \& P2(f(x0)) = \text{true}$

Получаем систему уравнений:

$$\begin{cases} P1(X0) = \text{истина} \\ P2(f(x0)) = \text{истина} \end{cases}$$

$X0$ -набор исходных данных

Любое решение системы дает тест, удовлетворяющий критерию покрытия операторов

Пусть:

$P1=(A>1) \& (B=0)$

$P2=(A=2) \vee (X>1)$

$f=(X:=X/A)$

$g=(X:=X+1)$

$$\begin{cases} (A0 > 1) \wedge (B0 = 0) = \text{истина} \\ (A2 = 2) \vee (X2 > 1) = \text{истина} \\ (A0 > 1) \wedge (B0 = 0) = \text{истина} \\ (A0 = 2) \vee (X0/A0 > 1) = \text{истина} \end{cases}$$

решением может быть набор

ТЕСТ = ($A0=2$, $B0=0$, $X0=4$)

Этот тест обеспечивает прохождение пути ascd

Этот метод плохо находит ошибки в предикатах (слабо чувствителен к такому виду ошибок)

Например. Ошибка1 – в P1 вместо символа & набита операция \vee

Ошибка2 – в P2 набрано $X>2$.

Эти ошибки не обнаружатся

МЕТОД ПОКРЫТИЯ ПЕРЕХОДОВ

Критерий метода: каждый переход в программе реализуется хотя бы 1 раз, т.е. каждый предикат в предикатной вершине принимает как истинное, так и ложное значение.

Этому критерию удовлетворяют 2 пары путей (см.блок-схему). ascd и abd или ascd и abed

Из трассировочной таблицы можно найти предикаты допустимые для первой пары:

Путь aced

$$P1(X0) \& P2(x2) \rightarrow \begin{cases} P1(X0) = TRUE \\ P2(f(X0)) = TRUE \end{cases}$$

Путь abd

$$\neg P1(X0) \& \neg P2(x2) \rightarrow \begin{cases} \neg P1(X0) = TRUE \\ \neg P2(X0) = TRUE \end{cases}$$

Из каждой системы получается по одному тесту:

$$\begin{cases} (A0 > 1) \wedge (B0 = 0) = \text{истина} \\ (A0 = 2) \vee (X0/A0 > 1) = \text{истина} \end{cases}$$

$$T1 = (A0=2, B0=1, X0=0)$$

$$\begin{cases} (A0 > 1) \wedge (B0 = 0) = \text{ложь} \\ (A2 = 2) \vee (X0 > 1) = \text{ложь} \end{cases}$$

$$T2 = (A0=3, B0=1, X0=0)$$

Аналогично из второй пары найти тесты. Этот метод плохо находит ошибки в условиях, образующих предикат.

Ошибка 1: в P1 набрано (A>-10). Эту ошибку T1 и T2 не найдут.

МЕТОД ПОКРЫТИЯ УСЛОВИЙ

Критерий метода: каждый возможный результат каждого условия в предикатах, записанных предикатных вершинах, выполняется хотя бы 1 раз.

Пусть предикаты P1 и P2 зависят от 2 условий:

$$P1 = P1(P11, P12)$$

$$P2 = P2(P21, P22)$$

Для нашего примера условиями будут:

$$P11 = (A > 1) \quad P12 = (B = 0)$$

$$P21 = (A = 2) \quad P22 = (X > 1)$$

Тогда для каждой предикатной вершины можно записать по 4 системы, а всего будет 8:

$$\begin{cases} P11(X0) = TRUE \\ P12(X0) = TRUE \end{cases} \quad (1)$$

$$\begin{cases} P11(X0) = TRUE \\ P12(X0) = FALSE \end{cases} \quad (2)$$

$$\begin{cases} P11(X0) = FALSE \\ P12(X0) = TRUE \end{cases} \quad (3)$$

$$\begin{cases} P11(X0) = FALSE \\ P12(X0) = FALSE \end{cases} \quad (4)$$

$$\begin{cases} P21(X0) = TRUE \\ P22(X0) = TRUE \end{cases} \quad (5)$$

$$\begin{cases} P21(X0) = TRUE \\ P22(X0) = FALSE \end{cases} \quad (6)$$

$$\begin{cases} P21(X0) = FALSE \\ P22(X0) = TRUE \end{cases} \quad (7)$$

$$\begin{cases} P21(X0) = FALSE \\ P22(X0) = FALSE \end{cases} \quad (8)$$

где f(X0) - составное поле данных перед вычислением предикатов P2 то есть в точке B.

Критерию покрытия условий будут удовлетворять тесты, полученные из систем:

- 1,4,5,8

- 2,3,6,7

Рассмотрим 1-ый вариант.

$$\begin{cases} P11 = (A0 > 1) = TRUE \\ P12 = (B0 = 0) = TRUE \end{cases} \quad (9)$$

$$\begin{cases} P21 = (Ab = 2) = TRUE \\ P22 = (Xb > 1) = TRUE \end{cases} \quad (10)$$

$$\begin{cases} P11 = (A0 > 1) = FALSE \\ P12 = (B0 = 0) = FALSE \end{cases} \quad (11)$$

$$\begin{cases} P21 = (Ab = 2) = FALSE \\ P22 = (Xb > 1) = FALSE \end{cases} \quad (12)$$

Выводим тесты из (9)÷(10)

$P11 \& P12 \& P21 \& P22 = TRUE$

$(A0 > 1) \& (B0 = 0) \& (Ab = 2) \& (Xb > 1) = (A0 > 1) \& (B0 = 0) \& (A0 = 2) \& (X0/A0 > 1) =$
 $(A0 > 1) \& (B0 = 0) \& (A0 = 2) \& (X0 > A0) = TRUE$

Получаем системы равенств и неравенств.

$$\begin{cases} A0 > 1 \\ B0 = 0 \\ A0 = 2 \\ X0 > A0 \end{cases} \quad (13)$$

Из нее получаем тест

$T1 = (A0 = 2, B0 = 0, X0 = 6)$

соответствующий пути ascd.

Выводим тесты из 11, 12.

$\neg P11 \& \neg P12 \& \neg P21 \& \neg P22 = (A0 \leq 1) \& (B0 \neq 0) \& (Ab \neq 2) \& (Xb \leq 1) =$
 $(A0 \leq 1) \& (B0 \neq 0) \& (A0 \neq 2) \& (X0 \leq 1) = TRUE$

Имеем систему

$$\begin{cases} A0 \leq 1 \\ B0 \neq 0 \\ A0 \neq 2 \\ X0 \leq A0 \end{cases} \quad (14)$$

Из системы получаем тест

$T2 = (A0 = 1, B0 = 1, X0 = 1)$

соответствующий пути abd.

Если бы системы 13, 14 были несовместны, то для получения тестов надо рассмотреть другие пары системы (1)÷(8).

Внимание: Тесты полученные этим критерием не всегда обеспечивают выполнение критерия покрытия перехода.

Например, пусть в программе используется предикат $P = A \& B$.

Тесты $T1 = (A = \text{истина}, B = \text{ложь})$.

$T2 = (A = \text{ложь}, B = \text{истина})$

удовлетворяют критерию покрытия условий но не удовлетворяют критерию покрытия переходов (они не покрывают переход для $P = \text{истина}$)

МЕТОД КОМБИНАТОРНОГО ПОКРЫТИЯ УСЛОВИЙ

Критерий метода: все возможные комбинации результатов условий в каждом предикате выполняется хотя бы 1 раз

Этот метод устраняет все недостатки, рассмотренные ранее, метод создает столько тестов, сколько необходимо для выполнения по крайней мере одного раза всех возможных комбинаций результатов условий в предикатах всех переходов в программе. Для построения тестов необходимо использовать все системы уравнений (1)÷(8), т.е. системы (9)÷(12) и (2),(3),(6),(7):

$$\begin{cases} P11 = (A0 > 1) = TRUE \\ P12 = (B0 = 0) = FALSE \end{cases} \quad (15)$$

$$\begin{cases} P21 = (Ab = 2) = TRUE \\ P22 = (Xb > 1) = FALSE \end{cases} \quad (16)$$

$$\begin{cases} P11 = (A0 > 1) = FALSE \\ P12 = (B0 = 0) = TRUE \end{cases} \quad (17)$$

$$\begin{cases} P21 = (Ab = 2) = FALSE \\ P22 = (Xb > 1) = TRUE \end{cases} \quad (18)$$

Из 8 систем можно получить в общем случае 8 тестов, но т.к. P2 вычисляется после P1, то достаточно 4.

$$T1=(A0=2,B0=0,X0=4) <- \{9,10\}$$

$$T2=(A0=2,B0=1,X0=1) <- \{15,16\}$$

$$T3=(A0=1,B0=0,X0=2) <- \{17,18\}$$

$$T4=(A0=1,B0=1,X0=1) <- \{11,12\}$$

ОБЩАЯ СТРАТЕГИЯ ПРОЕКТИРОВАНИЯ ТЕСТОВ

Каждый рассмотренный метод обеспечивает создание определенного набора тестов, но ни один из них сам по себе не может дать полный набор тестов.

Рекомендации:

1.Если спецификации содержат комбинацию входных условий, то начать надо с применения метода функциональных диаграмм.

2.Всегда надо использовать анализ граничных условий.

3.Метод эквивалентного разбиения.

4.Для получения дополнительных тестов можно использовать метод предположения об ошибке.

5.Применяя методы структурного тестирования отобрать тесты не входящие в ранее построенные.

Реализация этой стратегии весьма трудоемка. Она не гарантирует что все ошибки будут найдены.

ТЕСТИРОВАНИЕ МОДУЛЬНЫХ ПРОГРАММ

Процесс тестирования больших программных систем (с модульной структурой) состоит из двух этапов:

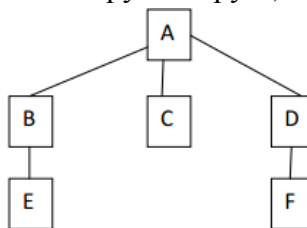
- тестирование отдельных модулей системы;
- тестирование всей системы.

Различают 2 подхода к комбинированию модулей:

- монолитное тестирование;
- пошаговое тестирование.

МОНОЛИТНОЕ ТЕСТИРОВАНИЕ МЕТОДОМ «БОЛЬШОГО УДАРА»

Это традиционный подход. В начале рассмотренными методами тестируется каждый модуль в отдельности (независимо друг от друга).



Для тестирования отдельного модуля надо:

- модуль «драйвер»
- модуль «заглушка»

Схема полигона для тестирования ниже:

В заключении оттестированные модули собираются в единую программу, которая тестируется как отдельный модуль. Монолитное тестирование – трудоемко, но ускоряет проверку за счет распределения работ.



ПОШАГОВОЕ ТЕСТИРОВАНИЕ

Модули тестируются не изолированно друг от друга, а подключаются поочередно к ранее отестированному набору. Возможно направление тестирования

-сверху,

-снизу.

При нисходящем тестировании проверка начинается с головного (верхнего) модуля:

А (первый уровень), В,С,D (второй уровень) Е,F (третий уровень)

Рекомендации по порядку подключения очередного тестируемого модуля.

1) модули, включающие операции ввода-вывода, подключаются как можно раньше

2) если в программе есть критические в каком-либо смысле части (сложный модуль, новый алгоритм и т.д.) то целесообразно включить эти части как можно раньше.

Например, если модуль F содержит операторы ввода, то пошаговая последовательность может быть следующей:

А,D,F,B,C,E

Для подключаемого тестируемого модуля создаются модули заглушки, которые имитируют функции вызова модулей.

Создание заглушек – задача не тривиальная, их задача – вернуть вызывающему модулю правильные данные. Могут создаваться разные заглушки индивидуально для каждого теста. При этом все тесты и сопутствующую информацию записывают во внешней памяти и заглушка их только читает и передает тестируемому модулю.

Недостатки нисходящего тестирования:

- может оказаться невозможным передать модулю необходимый набор данных, например из f в E.

- получить входной набор, соответствующих тесту для E – трудная интеллектуальная задача.

При восходящем тестировании проверка начинается с терминальных модулей (в начале E,C,F, потом B,D и в конце A).

Для каждого модуля – свой драйвер (для нашего примера 5 драйверов). Иногда драйверы разработать легче чем заглушки. На последовательность также влияет критичность модуля.

Выводы: при пошаговым тестировании обнаруживаются ошибки в интерфейсах. Отладка программы при пошаговом тестировании легче (ошибки локализируются в текущем контексте).

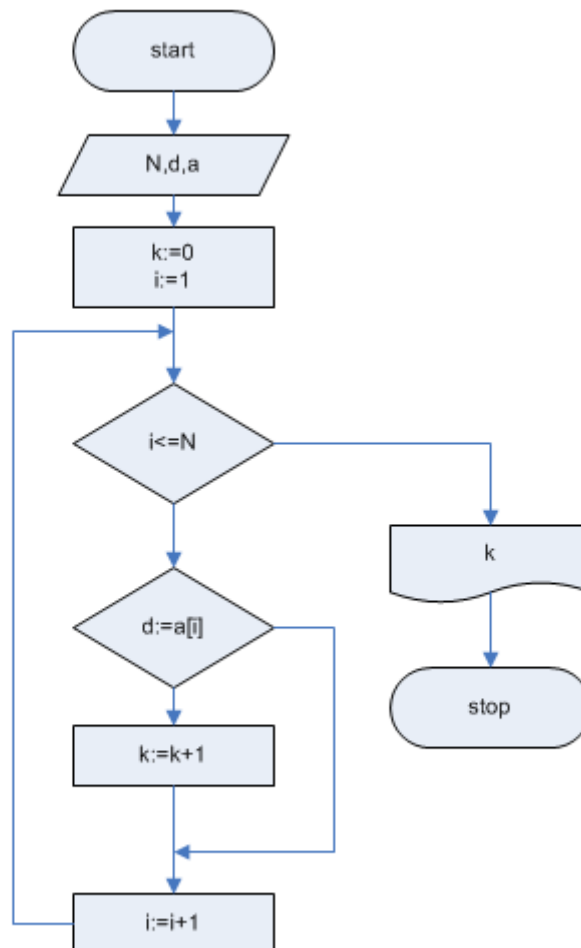
ТЕСТИРОВАНИЕ ПОТОКОВ ДАННЫХ

В этом способе используется принцип “белого ящика”. Но анализу подвергается не управляющая структура, а информационная структура программы.

Работа программы проверяется с позиции обработки потоков данных, передаваемых от входа к выходу.

Рассмотрим суть метода на примере программы, которая определяет сколько раз в массиве встречается заданный элемент.

Управляющая структура программы



Подсчитать количество повторения элементов в массиве $a[i:N]$, где

a – массив чисел

N – размер массива

d – заданный элемент

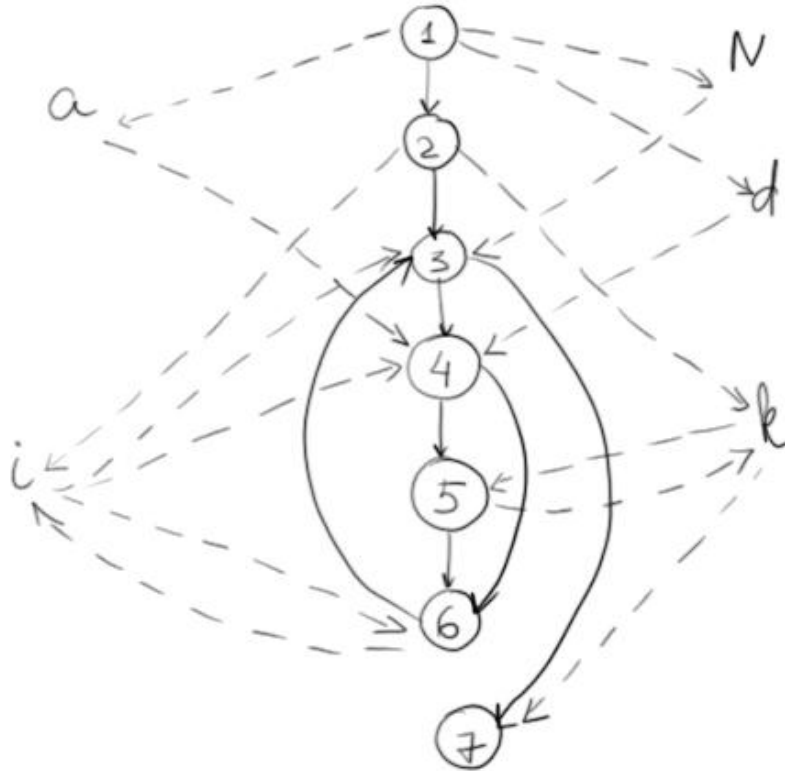
k – число повторения d в a

Построим потоковый граф программы, на котором изображены важные для нас данные: в каких блоках программы они определяются и в каких блоках они используются.

В нем:

- сплошные дуги – это связи по управлению между операторами в программе.
- пунктирные дуги отмечают информационные связи (связи по потокам данных).

Граф программы с управляющими и информационными связями



Информационные связи на данном графе обозначают следующее:

- в вершине 1 определяется значение переменных N, d, a
- значение переменной d и a используется в вершине 4
- значение переменной i определяется в вершинах 2 и 6, а используется в вершинах 3, 4 и 6
- и т.д.

В общем случае для каждой вершины графа можно записать:

-множество определений данных

$$DEF(i) = \{x \mid i\text{-я вершина содержит определение } x\}$$

-множество использования данных

$$USE(i) = \{x \mid i\text{-я вершина использует } x\}$$

Под определением данных понимают действия, изменяющие элемент данных, т.е. имя элемента стоит в левой части оператора присваивания $x := f(\dots)$.

Использование данных – это применение элемента в выражении, где происходит обращение к элементу данных, но не его изменение (т.е. имя элемента стоит в правой части оператора присваивания):

$$? := f(x)$$

Назовем ДИ-цепочкой (цепочкой определения-использования) конструкцию $[x, i, j]$, где i, j – имена вершин, x определена в i -ой ($x \in DEF(i)$) и используется в j -ой ($x \in USE(j)$) вершине.

В нашем примере получается следующая цепочка:

| | | | | |
|---------|---------|---------|---------|---------|
| [a,1,4] | [N,1,3] | [i,2,3] | [d,1,4] | [k,2,5] |
| | | [i,2,4] | | [k,2,7] |
| | | [i,2,6] | | [k,5,5] |
| | | [i,6,3] | | [k,5,7] |
| | | [i,6,4] | | |
| | | [i,6,6] | | |

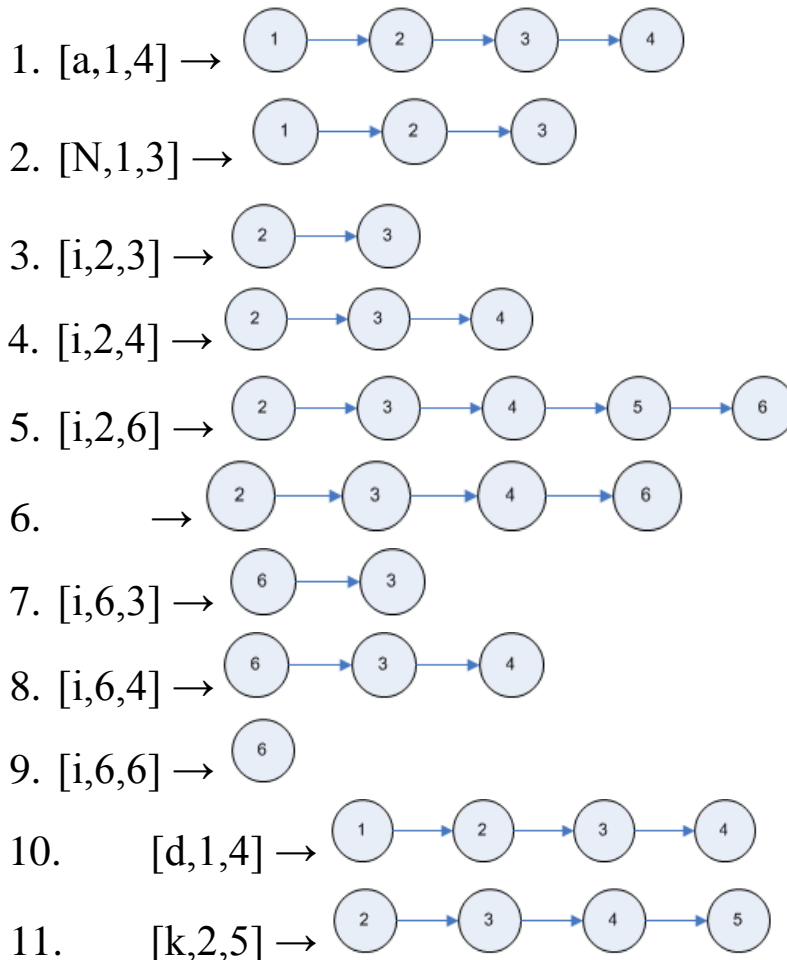
Всего получилось 13 ДИ-цепочек.


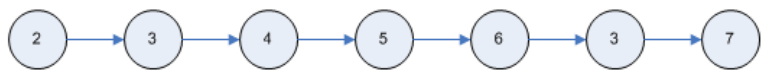
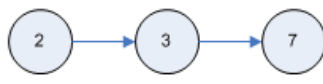
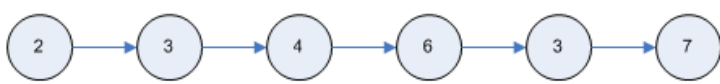
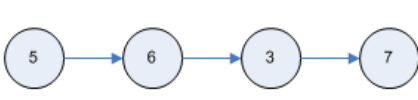
Способ ДИ тестирования требует охвата всех ДИ-цепочек программы, т.е., тесты должны разрабатываться исходя из анализа «жизни» всех данных программ (в частном случае – наиболее важных данных).

Следовательно, для составления тестов требуется выделение маршрутов-путей выполнения программы на управляющем графе программы. Критерий для выбора пути – покрытие максимального количества ДИ-цепочек.

Шаги ДИ-тестирования:

- 1) Построение управляющего графа программы
- 2) Построение информационного графа
- 3) Формирование полного набора ДИ-цепочек
- 4) Формирование полного набора отрезков путей в управляющем графе (отображение набора ДИ-цепочек информационного графа в отрезки путей управляющего графа):



12. $[k, 5, 5] \rightarrow$ 
13. $[k, 2, 7] \rightarrow$ 
14. \rightarrow 
15. \rightarrow 
16. $[k, 5, 7] \rightarrow$ 

5) Построение полных путей (маршрутов) на управляющем графе программы, каждый из которых покрывает максимальный набор отрезков путей:

1-ый полный путь выполнения программы:



Он покрывает следующие отрезки путей:

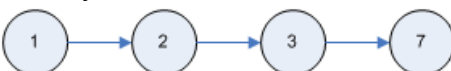
1,2,3,4,5,7,9,10,11,12,13,16

2-ой путь:



Он покрывает: 6,15

3-ий путь:



покрывает: 14

4-ый путь:



покрывает: 8

6) Составление тесов. Их можно получить из трассировочной таблицы этих путей

T1: N=1; a=[7]; d=7; Эталон k=1

T2: N=1; a=[7]; d=8; Эталон k=0

T3: N=0; a=[.]; d=5; Эталон k=0

T4: N=2; a=[8,3]; d=8; Эталон k=1

Достоинства ДИ-тестирования:

+ простота необходимого анализа операционно-управляющей структуры программы

+ возможность автоматизации метода

Недостатки:

- трудности в выборе минимального количества максимально эффективных тестов

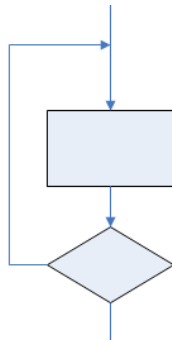
Область использования:

Программы с вложенными условными операторами и операторами цикла.

ОСОБЕННОСТИ ТЕСТИРОВАНИЯ ЦИКЛОВ

Тестирование циклов производится по принципу «белого ящика».

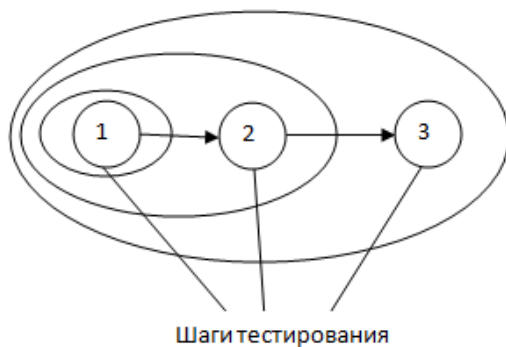
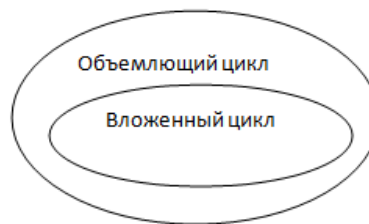
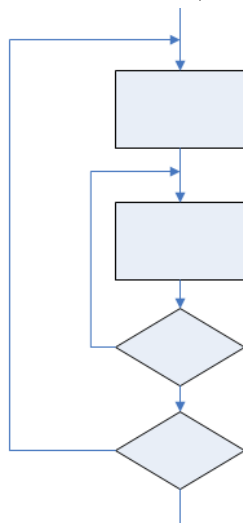
1. Простые циклы



Для проверки простых циклов с количеством повторений n может использоваться один из следующих наборов тестов:

- 1) прогон всего цикла
- 2) только один проход цикла
- 3) 2 прохода цикла
- 4) m проходов цикла, где $m < n$
- 5) $n-1$, n , $n+1$ проходов цикла

2. Вложенные циклы



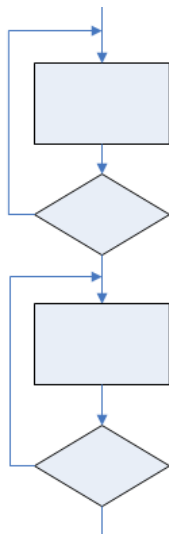
С увеличением уровня вложенности циклов количество возможных путей резко возрастает. Это приводит к нереальному количеству тестов.

Для сокращения количества тестов применяется следующая методика, в которой используется такие понятия как объемлющий и вложенный циклы. Порядок тестирования вложенных циклов показан на рисунке.

Шаги тестирования:

- 1) Выбирается самый внутренний цикл. Устанавливаются минимальные значения параметров всех остальных циклов.
- 2) Для внутреннего цикла проводятся тесты простого цикла. Добавляются тесты для исключенных значений и значений, выходящих за пределы рабочего диапазона.
- 3) Переходим в следующий по порядку объемлющий цикл. Выполняется его тестирование. При этом сохраняются минимальные значения параметров для всех внешних (объемлющих) циклов и типовые значения для всех вложенных циклов.
- 4) Работа продолжается до тех пор, пока не будут протестированы все циклы.

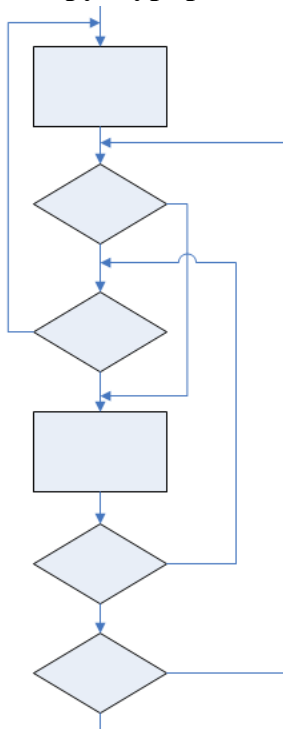
3. Объединенные циклы



Если каждый из циклов независим от других, то используется техника тестирования простых циклов.

При наличии зависимости (например, конечное значение счетчика первого цикла используется как начальное значение счетчика второго цикла) используется методика для вложенных циклов.

4. Неструктурированные циклы



Неструктурированные циклы тестированию не подлежат.

Этот тип циклов должен быть преобразован одним из методов в структурированную форму.

ТЕСТИРОВАНИЕ ПРАВИЛЬНОСТИ

Проводится после того, как закончится тестирование программной системы!

Цель – подтвердить, что функции, описанные в спецификации требований к ПС соответствуют ожиданиям заказчика.

Подтверждение правильности ПС выполняется с помощью тестов «черного ящика», демонстрирующих соответствие требованиям.

При обнаружении отклонений от спецификаций требований составляется список недостатков. Как правило, отклонения и ошибки, выявленные при подтверждении правильности, требуют изменения сроков разработки ПС.

Кроме самой программы другим важным элементом подтверждения правильности является проверка конфигурации ПС. Конфигурацией ПС называют совокупность всех элементов информации, вырабатываемых в процессе разработки ПС.

Минимальная конфигурация ПС включает следующие базовые элементы:

- 1) системную спецификацию
- 2) план программного проекта
- 3) спецификацию требований к ПС
- 4) предварительное руководство пользователя
- 5) спецификация проектирования
- 6) листинги исходных текстов программ
- 7) план и методику тестирования (тесты+результаты)
- 8) руководство по работе и инсталляции
- 9) ехе-код выполняемой программы
- 10) описание БД
- 11) руководство пользователя по настройке
- 12) документы сопровождения
- 13) стандарты и методики конструирования ПС

Проверка конфигурации минимизирует проблемы, которые могут возникнуть на этапе сопровождения ПС.

Разработчик не может предугадать, как заказчик будет реально использовать ПС. Поэтому для обнаружения ошибок, которые способен найти только конечный пользователь, предусматривается **альфа- и бета-тестирование**.

Альфа-тестирование проводится Заказчиком в организации Разработчика. Фиксируются все ошибки ПС и проблемы исполнения.

Бета-тестирование проводится конечным пользователем в организации Заказчика. Разработчик в этом процессе участия не принимает. Тестирование проводится в течении определенного срока (около года). Если обнаружены ошибки, то Разработчик их устраняет, изменяя ПС.

ТЕСТИРОВАНИЕ ОБЪЕКТНО-ОРИЕНТИРОВАННЫХ ПРОГРАММ

Тестирование этого класса программ отличается от процедурно-ориентированного тестирования. Хотя многие понятия, подходы и способы у них общие.

Особенности объектно-ориентированных систем вносят определенные изменения как в последовательность этапов, так и в содержание этапов тестирования.

Изменения проявляются в следующих трех направлениях:

- 1) расширение области применения тестирования
- 2) изменение методики тестирования
- 3) учет особенностей объектно-ориентированного ПО при составлении тестов

РАСШИРЕНИЕ ОБЛАСТИ ПРИМЕНЕНИЯ ТЕСТИРОВАНИЯ ПРО ООП

Разработка объектно-ориентированного ПО начинается с проектирования различных моделей (многомодульный подход): диаграмм классов и объектов, диаграммы переходов состояний объектов, временных диаграмм, диаграммы модулей.

Опыт показал, что на составление качественных моделей уходит большая часть временных затрат.

Если к этому добавить, что цена устранения ошибки растет с каждым витком «спирали», то логично тестирование надо начинать с объектно-ориентированных моделей анализа и проектирования!

Критерии тестирования моделей:

- правильность
- полнота
- согласованность

О правильности моделей судят, во-первых, по синтаксической правильности использования нотации применяемой технологии (например, нотации Г. Буча, язык UML), во-вторых, по семантике правильности моделей, т.е. насколько модели соответствуют предметным абстракциям.

Полнота моделей – это уровень отображения моделей реального мира. Уровень оценивается экспертами, имеющими знания и опыт в данной предметной области. Эксперты анализируют содержание классов, наследование классов, выявляя пропуски и неоднозначности. Проверяется соответствие намеченных отношений между классами реалиям физического мира.

О согласованности судят путем рассмотрения противоречий между элементами в модели. Несогласованная модель имеет в одной части представления, которые противоречат представлениям в других частях модели.

Для оценки согласованности нужно исследовать каждый класс и его взаимодействие с другими классами! Для упрощения исследования применяют модель Класс-Обязанность-Сотрудничество (модель CRC, Class-Responsibility-Collaboration). Основным элементом этой модели – CRC-карта.

По сути, CRC-карта – это небольшая таблица. Она помогает установить задачи класса и выявить его окружение (классы-собеседники). Для каждого класса создается отдельная карта.

В каждой CRC-карте указывается имя класса, его обязанности (операции) и его сотрудники (другие классы, в которые он посылает сообщения и от которых он зависит при выполнении своих обязанностей).

Сотрудничество подразумевает наличие ассоциаций и зависимостей между классами, которые фиксируются в диаграмме классов и объектов, а также во временной диаграмме.

CRC-карта намеренно сделана простой по структуре и маленькой по размеру.

Пример CRC-карты класса Банкомат.

| Имя класса: Банкомат | |
|----------------------|----------------|
| Обязанности | Сотрудники |
| Читать карту клиента | Карта клиента |
| Идентификация | БД |
| Проверка счета | БД счетов |
| Выдача денег | Блок денег |
| Выдача квитанции | Блок квитанции |
| Захват карты | Блок карт |

Цель: если список обязанностей и сотрудников не помещается на карте, то, наверное, данный класс надо разделить на несколько классов.

Для оценки модели (в основном диаграммы классов) с помощью CRC-карт рекомендуются следующие шаги:

- 1) Выполняется перекрестный просмотр CRC-карты и диаграммы объектов (временной диаграммы).

Цель – проверить наличие сотрудников, согласованность информации в этих 3-х моделях (CRC-карте, диаграмме объектов и временной диаграмме).

- 2) Исследуются обязанности CRC-карт.

Цель – определить, предусмотрены ли в карте сотрудники и обязанности, которые делегируются ему из данной карты.

Например, для CRC-карты Банкомат выясняем, выполняется ли обязанность *Читать карту клиента*, которая требует использования сотрудника *Карта клиента*. Это означает, что класс *Карта клиента* должен иметь операцию, которая позволяет ему прочитать карточку клиента.

- 3) Организуется проход по каждому соединению CRC-карты.

Цель – проверить корректность запросов, выполняемых через соединения.

Такая проверка гарантирует, что каждый сотрудник, предоставляющий услугу, получает обоснованный запрос. Например, если допущена ошибка и класс *БД клиентов* получает от класса Банкомат запрос на *Состояние счета*, то он не сможет его выполнить – ведь *БД клиентов* не знает состояние их счетов.

- 4) Определяется, требуются ли другие классы.

Цель – правильно (рационально) ли распределены обязанности по классам. Для этого используют проходы по соединениям, исследованным на шаге 3.

- 5) Определяется, нужно ли объединить часто запрашиваемые обязанности.

Цель – сократить число обязанностей. например, в любой ситуации используют пару обязанностей – *Читать карту клиента* и *Идентификация клиента*. Их можно объединить в новую обязанность *Проверка клиента*, которая подразумевает как чтение его карты, так и идентификацию клиента.

- 6) Шаги 1-5 применяются итеративно, к каждому классу и на каждом шаге эволюции объектно-ориентированной модели.

ИЗМЕНЕНИЕ МЕТОДИКИ ПРИ ОБЪЕКТНО-ОРИЕНТИРОВАННОМ ТЕСТИРОВАНИИ

Классическая методика тестирования предполагает следующий порядок действий:

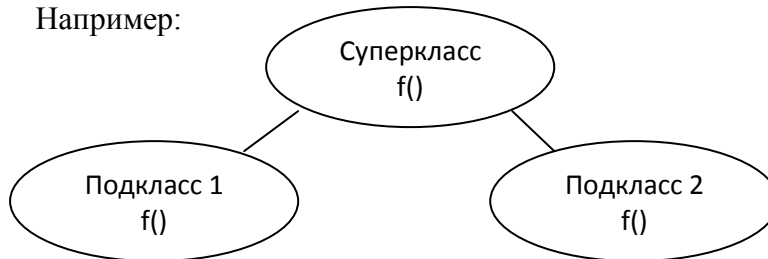
- 1) тестируются отдельные модули
- 2) тестируется интеграция модулей (система)
- 3) тестируется правильность (функциональные требования заказчика + конфигурация ПО)
- 4) системное тестирование (правильная реализация всех системных функций)

1. Особенности тестирования объектно-ориентированных «модулей»

При объектно-ориентированном тестировании меняется понятие и «модуля». Наименьшим тестируемым элементом теперь является класс (объект). Поскольку класс содержит несколько операций и свойств, то это сильно изменит содержание тестирования класса.

В данном случае нельзя тестировать отдельную операцию изолированно, как это принято в стандартном подходе к тестированию модулей. Любую операцию приходится рассматривать как часть класса.

Например:



Есть иерархия классов. Операция $f()$ определена в суперклассе и наследуется несколькими подклассами. Каждый подкласс использует $f()$, но она применяется в контексте его частных особенностей. Этот контекст меняется, поэтому операцию $f()$ надо тестировать в контексте каждого подкласса.

Т.е. изолированное тестирование операции $f()$, являющаяся традиционным подходом в тестировании модулей, не имеет смысла в ООП.

Выводы:

- Тестированию модулей традиционного ПО соответствует тестирование классов ООП.
- Тестирование традиционных модулей ориентировано на поток управления внутри модуля и поток данных через интерфейс модуля.
- Тестирование классов ориентировано на операции, инкапсулированные в классе, и состояния в пространстве поведения класса.

2. Тестирование объектно-ориентированной интеграции

Объектно-ориентированное ПО не имеет иерархической управляющей структуры, поэтому здесь не применимы методики как восходящей, так и нисходящей интеграции.

Даже классический прием интеграции (добавление по одной операции в класс) зачастую неосуществимо.

В свое время (1994 г.) американцы предложили 2 методики интеграции ОО Систем:

- 1) Тестирование, основанное на потоках;
- 2) Тестирование, основанное на использовании.

Первая методика. Объектом интеграции является набор классов, обслуживающий единственный ввод данных в систему. Другими словами, средства обслуживания каждого потока интегрируются и тестируются отдельно. Для проверки отсутствия ошибок (побочных эффектов) применяют регрессионное тестирование.

Вторая методика. В начале синтезируются и тестируются независимые классы. Далее переходят к первому слою зависимых классов (которые используют независимые классы), ко второму слою и т.д. В отличие от стандартной интеграции, везде, где возможно, избегают драйверов и заглушек.

Опыт показывает, что одним из шагов ОО Тестирования интеграции должно быть кластерное тестирование.

Кластер сотрудничающих классов определяется исследованием CRC-модели или диаграммы объектов. Тестовые варианты для кластера ориентированы на обнаружение ошибок сотрудничества.

3. Объектно-ориентированное тестирование правильности

Подтверждение правильности объектно-ориентированного ПО ориентировано на видимые действия пользователя и распознаваемые ими выводы из системы (аналогично структурному ПО). Здесь обычно применяют тестирование «черного ящика».

Для формирования тестов используют информацию из ТЗ, диаграммы объектов и временной диаграммы.

Проверяется конфигурация ПО.

ПРОЕКТИРОВАНИЕ ОБЪЕКТНО-ОРИЕНТИРОВАННЫХ ТЕСТОВ

Традиционные тесты ориентированы на проверку в отдельном модуле:

- последовательности: ввод исходных данных-обработка-вывод результатов;
- внутренней управляющей (информационной) структуры.

Объектно-ориентированные тесты проверяют состояние классов!

Получение информации о состоянии затрудняют следующие объектно-ориентированные характеристики:

1) Инкапсуляция

Информацию о состоянии класса можно получить только с помощью встроенных в него операций, которые возвращают значения свойств класса.

2) Полиформизм

При выводе полиморфной операции трудно определить, какая реализация будет проверяться.

Например, надо проверить вызов функции $y=f(x)$.

при стандартном тестировании достаточно рассмотреть одну реализацию этой функции.

В объектно-ориентированном варианте придется рассмотреть поведение реализаций:

- Базовый класс :: $f(x)$
- Производный класс :: $f(x)$
- Наследник производного класса : $f(x)$

Надо рассматривать разные поведения $f(x)$.

3) Наследование

Наследование также может усложнить проектирование тестов.

Например, пусть Родительский_класс содержит операции унаслед() и переопр(). Дочерний_класс переопределяет операцию переопр() по-своему. Очевидно, что реализация Дочерний_класс: переопр() должна повторно тестироваться, ведь ее содержание изменилось.

Но надо ли повторно тестировать операцию Дочерний_класс: унаслед()?

Возможен случай, когда операция Дочерний_класс:унаслед() вызывает операцию переопред().

Поскольку реализация операции переопр() изменена, операция Дочерний_класс: унаслед() может не соответствовать этой новой реализации. Поэтому нужны новые тесты, хотя содержание операции Унаслед() не изменено.

Выводы:

1. К операциям класса применимы классические способы тестирования «белого ящика», которые гарантируют проверку каждого оператора и их управляющих связей.
При большом количестве операций от тестирования по «белому ящику» приходится отказываться. Меньших затрат потребует тестирование на уровне классов.
2. Способы тестирования «черного ящика» также применимы к ОО Системам.

Построение тестов, основанное на ошибках

Цель такого тестирования – обнаруживать предполагаемые ошибки в программе. Разработчик выдвигает гипотезу о предполагаемых ошибках и для проверки его предположений составляются тесты.

Например, пусть имеется логическое выражение

$$if(x \text{ and } not\ y \text{ or } z)$$

Специально по тестированию выдвигают гипотезу о предполагаемых ошибках выражения:

- вокруг *not y or z* должны быть круглые скобки;
- вместо *and* должно быть *or*;
- и тд.

Для каждой предполагаемой ошибки проектируются тесты.

Эффективность этой методики зависит от правильности предположения об ошибке. Эта идея тестирования применима и при тестировании интеграции.

Построение тестов, основанных на сценариях

Тестирование, основанное на ошибках, оставляет в стороне 2 важных типа ошибок:

- некорректные спецификации (программа не выполняет то, что хочет заказчик);
- взаимодействия между подсистемами (поведения одной подсистемы создает предпосылки для отказа другой подсистемы).

Тестирование на сценариях ориентировано на действия пользователя, а не на действия ПС.

Это означает фиксацию задач, которые выполняет пользователь, а затем применение их в качестве тестов! Задачи пользователя получают из ТЗ и спецификаций ПС.

Этот метод более эффективен, чем тестирование на ошибках, т.к. тесты на сценариях более сложны и лучше отражают реальные проблемы предметной области.

Способы тестирования содержания классов

Они ориентированы на отдельный класс и операции, которые инкапсулированы классом.

1. Стохастическое тестирование класса.

Сами тесты генерируются случайным образом. Рассмотрим пример. Пусть имеется класс `Счет` со след. операциями, которые могут выполняться в разном сочетании...

~FIN~