Цель работы

Приобретение навыков статического анализа ПС на предмет готовности к устойчивому функционированию с помощью подсчета метрик Холстеда и цикломатической сложности (метрики МакКейба) по исходным текстам программ.

Формулировка задания

1. Вычислить метрики Холстеда для выбранного и согласованного с преподавателем фрагмента исходного текста свободно распространяемого ПО.
2. Получить у преподавателя вариант *(Приложение 1. Метрики Холстеда).* Реализовать процедуру, указанную в задании. После того как программа создана провести тщательное тестирование и зафиксировать количество ошибок, которое обнаружено в ходе тестирования.
3. Для созданной программы вычислить метрики Холстеда.
4. Сопоставить значения расчетных метрик (п. 3) со значениями, зафиксированными на практике (п. 2).
5. Оценить цикломатическую сложность для указанных в задании *(Приложение 1. Метрика МакКейба)* учебной и реальной программ. С этой целью для каждой программы необходимо построить граф потока управления и по его счетным элементами рассчитать значение цикломатической сложности.
6. Для этих же программ рассчитать цикломатическую сложность по точкам выхода и принятия решения.
7. Сопоставить полученные результаты и сделать выводы о сложности двух фрагментов.
8. Выбрать из исходного теста свободно распространяемого ПО, использованного в п.1, целую функцию или модуль объемом не более 100 строк. Для этого программного блока построить граф потока управления и рассчитать значение цикломатической сложности. Для построения графа управления рекомендуется использовать общедоступные средства автоматизации анализа исходных текстов.

9. Оценить применимость метрики МакКейба для оценки сложности больших и небольших фрагментов кода. Узнать и аргументировать ее недостатки.

Вариант задания – 8. Процедура, реализующая сортировку INT массива кучей (пирамидальную сортировку).

Ход работы

Для вычисления метрик был выбран проект *https://github.com/compiler-dept/speck/blob/master/speck.c*. Фрагмент кода для анализа представлен в *приложении 1*.

Статистика исходного текста представлена в *таблицах 1* и *2*.

№	Оператор	Количество
1	while	1
2	*	2
3	If	1
4	->	6
5	+	2
6	++	1
7	=	10
Итого	$\boldsymbol{n_1 = 7}$	$\boldsymbol{N_1 = 23}$

Таблица 1 – Статистика по операторам

№	Оператор	Количество
1	fp	2
2	suite	7
3	line	5
4	NULL	2
5	0	3
6	linelen	2
7	test_count	5
8	temp	3
9	c_file	1
10	tests	6
Итого	$\boldsymbol{n_2 = 10}$	$\boldsymbol{N_2 = 36}$

Таблица 2 – Статистика по операндам

Значения метрик Холстеда:

- Словарь $N = N_1 + N_2 = 23 + 36 = 59$;
- Объем программы: $V = Nlog_2(n_1 + n_2) = 59 * \log_2 17 = 236$;
- Сложность программы: $D = \frac{n_1 N_2}{2n_2} = \frac{7*36}{2*10} = 12{,}6$;

- Трудоемкость программы: $A = VD = 2973{,}6$;
- Количество дефектов для «модели Холстеда 1»: $E = \frac{V}{3000} = 0{,}079$;
- Количество дефектов для «модели Холстеда 2»: $E = \frac{A^{\frac{2}{3}}}{3000} = 0{,}069$;
- Время написания программы: $T = \frac{A}{18} = 165{,}2$ сек

Далее реализована пирамидальная сортировка *(приложение 2)*. Написание программы заняло 25 минут. В *таблицах 3-4* представлена статистика по операторам и операндам.

№	Оператор	Количество
1	If	2
2	Else	2
3	Else if	1
4	For	5
5	while	1
6	=	18
7	return	1
8	Непрямое обращение (*)	2
9	&&	1
10	==	1
11	<	1
12	>=	2
13	Обращение к элементу массива	7
14	!	1
15	++	3
16	+	2
17	--	2
18	-	6
19	/	1
20	Умножение (*)	7
21	<=	1
Итого	$\boldsymbol{n_1 = 21}$	$\boldsymbol{N_1 = 67}$

Таблица 3 – Статистика по операторам

№	Оператор	Количество
1	numbers	16
2	root	12
3	bottom	3
4	done	3
5	maxChild	8
6	temp	4

7	array_size	4
8	I	22
9	a	5
Итого	$\boldsymbol{n_2 = 9}$	$\boldsymbol{N_2 = 77}$

Таблица 4 – Статистика по операндам

Значения метрик Холстеда:

- Словарь $N = N_1 + N_2 = 67 + 77 = 144$;
- Объем программы: $V = Nlog_2(n_1 + n_2) = 144 * \log_2 30 = 706.59$;
- Сложность программы: $D = \frac{n_1 N_2}{2n_2} = \frac{21*77}{2*9} = 89{,}83$;
- Трудоемкость программы: $A = VD = 63472{,}98$;
- Количество дефектов для «модели Холстеда 1»: $E = \frac{V}{3000} = 0.24$;
- Количество дефектов для «модели Холстеда 2»: $E = \frac{A^{\frac{2}{3}}}{3000} = 0.53$;
- Время написания программы: $T = \frac{A}{18} = 58{,}77$ мин

Далее была оценена цикломатическая сложность для учебного и реального кода, представленных в *приложении 3*.

Анализируя учебный код, получаем следующий граф потока управления *(см. рис. 1)*.

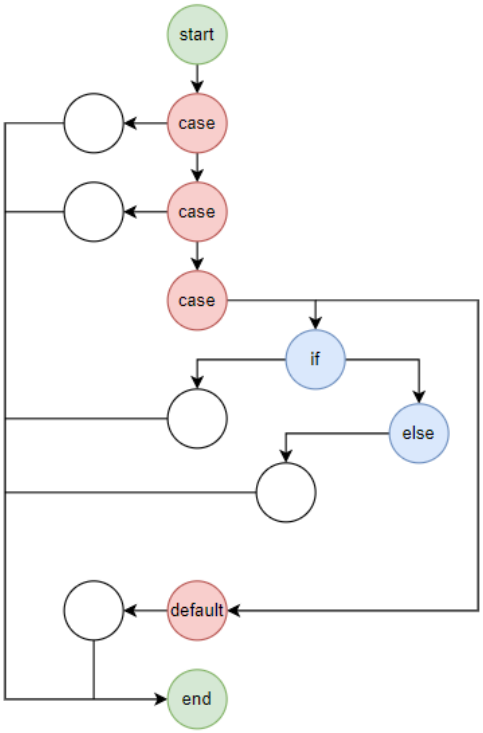

Рисунок 1 – Граф потока управления учебной программы

Анализируя *рисунок 1*, получаем следующее:

- Количество ребер $E = 16$;
- Количество вершин $N = 13$;
- Количество точек принятия решений: $W = 6$;
- Количество точек выхода: $X = 1$;
- Количество связанных компонент (независимые части): $P = 1$;
- Цикломатическая сложность (метрика МакКейба) по счетным элементам: $M = E - N + 2P = 16 - 13 + 2 * 1 = 5$;
- Цикломатическая сложность (метрика МакКейба) по точкам принятия решений: $M = W + 1 = 6 + 1 = 7$;
- Цикломатическая сложность (метрика МакКейба) по точкам выхода: $M = W - X + 2 = 6 - 1 + 2 = 7$.

Анализируя реальный код, получаем следующий граф потока управления *(см. рис. 2)*.

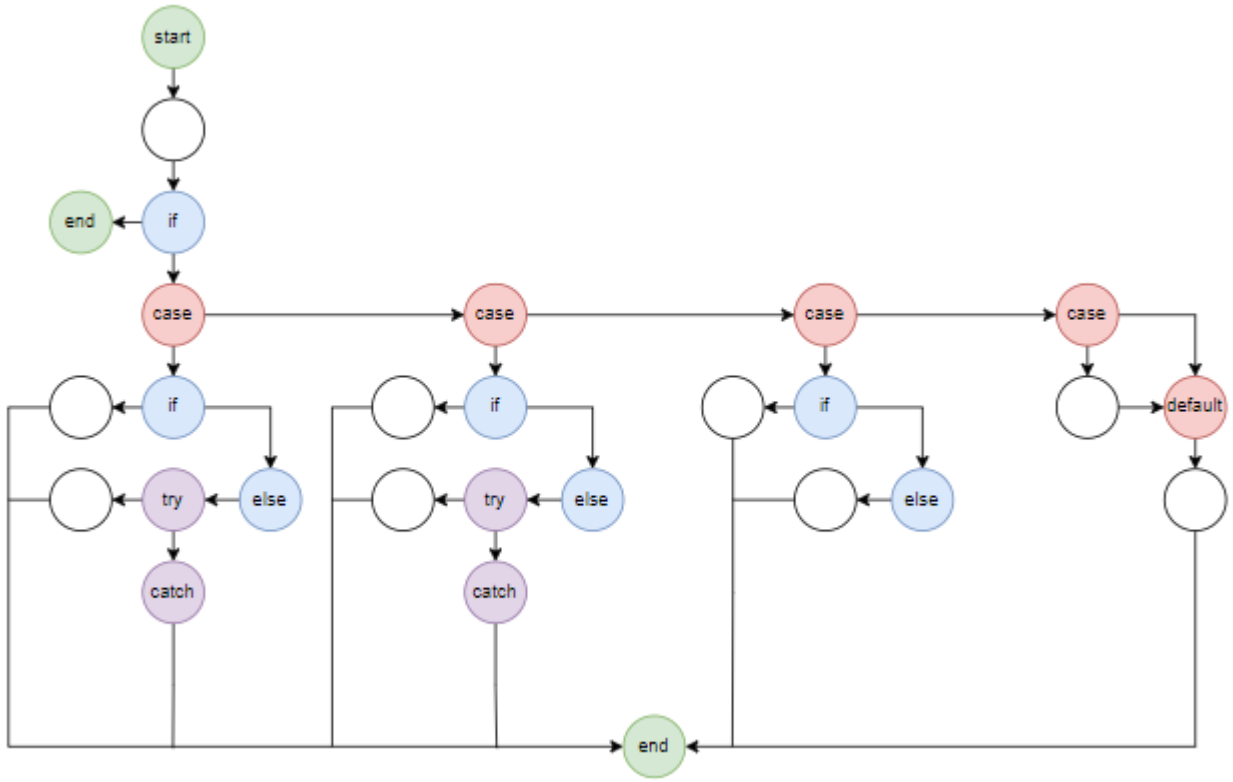

Рисунок 2 – Граф потока управления реальной программы

Анализируя рисунок 2, получаем следующее:

- Количество ребер $E = 36$;
- Количество вершин $N = 28$;
- Количество точек принятия решений: $W = 16$;
- Количество точек выхода: $X = 2$;
- Количество связанных компонент (независимые части): $P = 1$;
- Цикломатическая сложность (метрика МакКейба) по счетным элементам: $M = E - N + 2P = 36 - 28 + 2 * 1 = 10$;
- Цикломатическая сложность (метрика МакКейба) по точкам принятия решений: $M = W + 1 = 16 + 1 = 17$;
- Цикломатическая сложность (метрика МакКейба) по точкам выхода: $M = W - X + 2 = 16 - 2 + 2 = 16$.

Результаты не совпали, поэтому можно сделать вывод, что цикломатическая сложность достаточно велика, и стоит тщательнее проверять программу.

Построим граф для ранее выбранного фрагмента ПО *(приложение 1)* и также рассчитаем для него значения.

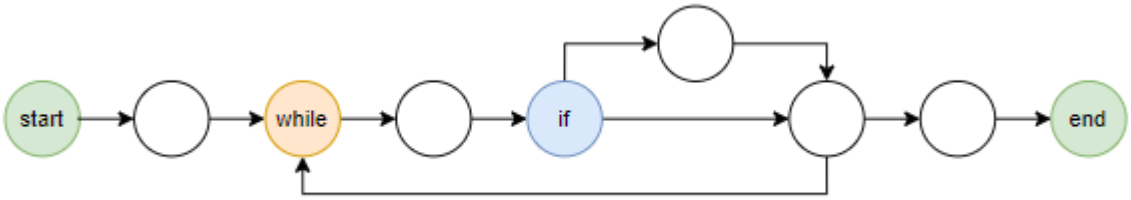

Рисунок 3 – Граф потока управления выбранного фрагмента ПО

- Количество ребер: $E = 10$;
- Количество узлов: $N = 9$;
- Количество точек принятия решений: $W = 2$;
- Количество точек выхода: $X = 1$;
- Количество связанных компонент (независимые части): $P = 1$;
- Цикломатическая сложность (метрика МакКейба) по счетным элементам: $M = E - N + 2P = 10 - 9 + 2 * 1 = 3$;

- Цикломатическая сложность (метрика МакКейба) по точкам принятия решений: $M = W + 1 = 2 + 1 = 3$;
- Цикломатическая сложность (метрика МакКейба) по точкам выхода: $M = W - X + 2 = 2 - 1 + 2 = 3$.

Полученные результаты совпали, что означает простоту выбранного фрагмента и отсутствие необходимости в его упрощении.

Для больших фрагментов кода использование метрики МакКейба является затруднительным, особенно если анализируемый участок имеет большое количество ветвлений.

Недостаток метрики заключается в том, что не учитывается сложность линейных участков кода, которые могут нести большое количество вычислений, значительно усложняя программу.

Контрольные вопросы

1. На чем основаны статические методы анализа ПС?

Они основаны на подсчете количества ошибок в программе.

2. Какие характеристики программы входят в набор метрик Холстеда?

Количество операторов, операндов и машинных инструкций.

3. Что такое словарь программы? Укажите формулу для его расчета.

Словарь программы – это общее количество операторов и операндов программы; вычисляется по формуле $\mathrm{N} = \mathrm{N}_1 + \mathrm{N}_2$.

4. Почему применяют две различные оценки числа дефектов в ПС по Холстеду?

Применяются две ошибки, так как первая обычно используется для объектно-ориентированных языков, вторая – для не объектно-ориентированных языков.

5. Зачем необходим расчет времени T написания программы в модели Холстеда?

Для понимания того, сколько на самом деле необходимо времени для написания ПО.

6. Что влияет на сложность ПС в модели МакКейба?

На сложность ПО влияет количество и взаимное расположение точек принятия решения.

7. Как влияет на сложность ПС количество независимых подпрограмм при расчете цикломатической сложности?

При увеличении количества независимых подпрограмм сложность ПО увеличивается, поскольку увеличивается количество возможных путей их прохождения.

8. Какие топологический параметры графа потока управления влияют на цикломатическую сложность программы?

На сложность ПО влияет количество узлов графа, его рёбер и число связных компонент.

9. Как вычисляют цикломатическую сложность программы, у которой одна точка входа и два варианта завершения работы?

$M = E - N + 2P$, где E – число рёбер, N – число узлов, P – число связных компонент, или $M = W - X + 2$, где W – число точек принятия решения, X – число точек выхода, равное в данном случае 2.

10. Какие недостатки существуют у метрики МакКейба?

Недостатками модели МакКейба является смешивание циклов и условных операторов, а также рассмотрение только потока управления в отрыве от прочего кода, выполняющегося линейно.

Вывод

В результате выполнения работы было проведено ознакомление с детерминированной методикой оценки надежности ПО с использованием программных метрик Холстеда, полученных на основе анализа исходного текста программы. Так же в данной работе была рассмотрена и применена метрика МакКейба.

Приложение 1

```
void get_tests(struct suite* suite)
{
    FILE* fp = fopen(suite->c_file, "r");
    char* line = NULL;
    size_t linelen = 0;
    ssize_t len;
    int test_count = 0;
    while ((len = getline(&line, &linelen, fp)) > 0) {
        char* temp = str_match(line, len);
        if (temp) {
            suite->tests = realloc(suite->tests, (test_count + 1) * sizeof(char*));
            suite->tests[test_count++] = temp;
        }
        free(line);
        line = NULL;
    }

    suite->tests = realloc(suite->tests, (test_count + 1) * sizeof(char*));
    suite->tests[test_count] = NULL;

    free(line);
    fclose(fp);
}
```

Приложение 2

```
// Функция "просеивания" через кучу - формирование кучи
void siftDown(int* numbers, int root, int bottom)
{
   int maxChild; // индекс максимального потомка
   int done = 0; // флаг того, что куча сформирована
   // Пока не дошли до последнего ряда
   while ((root * 2 <= bottom) && (!done))
   {
      if (root * 2 == bottom)     // если мы в последнем ряду,
         maxChild = root * 2;    // запоминаем левый потомок
       // иначе запоминаем больший потомок из двух
      else if (numbers[root * 2] > numbers[root * 2 + 1])
         maxChild = root * 2;
      else
         maxChild = root * 2 + 1;
      // если элемент вершины меньше максимального потомка
      if (numbers[root] < numbers[maxChild])
      {
         int temp = numbers[root]; // меняем их местами
         numbers[root] = numbers[maxChild];
         numbers[maxChild] = temp;
         root = maxChild;
      }
      else // иначе
         done = 1; // пирамида сформирована
   }
}
// Функция сортировки на куче
void heapSort(int* numbers, int array_size)
{
   // Формируем нижний ряд пирамиды
   for (int i = (array_size / 2) - 1; i >= 0; i--)
      siftDown(numbers, i, array_size - 1);
   // Просеиваем через пирамиду остальные элементы
   for (int i = array_size - 1; i >= 1; i--)
   {
      int temp = numbers[0];
      numbers[0] = numbers[i];
      numbers[i] = temp;
      siftDown(numbers, 0, i - 1);
   }
}
```

Приложение 3

```
// Учебный код
class SwitchDemo2
{
   public static void main(String[] args) {
      int month = 2;
      int year = 2000;
      int numDays = 0;
      switch (month) {
      case 12:
         numDays = 31;
         break;
      case 11:
         numDays = 30;
         break;
      case 2:
         if (((year % 4 == 0) && !(year % 100 == 0))
            || (year % 400 == 0))
            numDays = 29;
         else
            numDays = 28;
         break;
      default:
         System.out.println(«Invalid month.»);
         break;
      }
      System.out.println(<< Number of Days << +numDays);
   }
}

// Реальный код
private void invoke(final InvokationType type, final ActionEx action)
{
   ActionControllerMethod method = action.getMethod();
   if (!method.isValid())
      return;
   switch (type) {
   case AsyncSwing:
      if (EventQueue.isDispatchThread()) {
         directInvoket(action);
      }
      else {
         try {
            Swingutilities.invokeLater(new Task(action));
         }
         catch (Throwable th)
         {

         }
      }
      break;
   case SyncSwing:
      if (EventQueue.isDispatchThread())
      {
         directInvoke(action);
      }
      else {
         try {
            Swingutilities.invokeAndWait(new Task(action));
         }
         catch (Throwable th)
         {

         }
      }
      break;
```

```
    case SeparatedThread:
        if (m_pool != null)
            m_pool.execute(new Task(action));
        else
            new Thread(new Task(action)).start();
        break;
    case Direct:
        new Thread(new Task(action)).start();
    default:
        directInvoke(action);
        break;
    }
}
```