Национальный исследовательский университет «Высшая школа экономики»
Факультет компьютерных наук
Системное программирование

Оплачко Николай Алексеевич
мСП21

Лабораторная работа 1

Формальные методы программной инженерии

Москва, 2021

Оглавление

1. Введение

Цели данной работы:

- Предложить систему и привести её неформальное описание
- Привести формальную спецификацию системы в виде конечного автомата
- Построить тестовые последовательности для автомата
- Программно реализовать предложенную систему и сгенерировать множество её мутантов
- Оценить полноту набора тестовых последовательностей относительно множества мутантов

2. Неформальное описание выбранной системы

Задача системы — посчитать количество чисел, поданных на вход, кратных трём.

1) На вход системе подаётся текст, состоящий из цифр и пробелов:
    - непрерывные группы цифр считаются числом, записанным в десятичной системе счисления.
2) Система считывает текст посимвольно, считая пробел отдельным символом.
3) При обнаружении числа, кратного трём, система возвращает команду «+1».

Примеры:

- ␣␣ → 0 чисел кратно трём
- 1␣2␣ → 0 чисел кратно трём
- 4␣␣33␣ → 1 число кратно трём

3. Описание программной реализации

В приложении 1 приведён исходный код программной реализации на языке C.

В файле `src/div.c` программно реализована процедура `scan_numbers_divisible_by_three`, которая принимает в качестве аргумента строку с трансформированным тестовым входом и выводит на каждый символ реакцию: 0 или 1 (аналогично автомату). Суть происходящего — к считываемому числу приписывается по одной цифре и пересчитывается остаток по модулю три. Если число считано целиком (встречен пробел), и остаток оказался равен нулю, программа выводит 1. В остальных случаях реакцией на символ будет 0.

Точка входа программной реализации находится в файле `src/main.c`: считывается строка и подаётся в качестве аргумента в процедуру `scan_numbers_divisible_by_three`.

Процесс сборки программной реализации описан в `CMakeLists.txt` и инициируется для заданного мутанта запуском скрипта `scripts/build_with_mutant.sh`.

Скрипт `scripts/transform_input.sh` осуществляет трансформацию входной тестовой последовательности. Пример: `123␣45␣` → `1203123`.

Скрипт `scripts/generate_mutants.sh` генерирует набор мутантов и оставляет только успешно проходящих процесс сборки.

Скрипт `scripts/run_tests.py` запускает исполняемый файл на заданном наборе тестовых последовательностей и завершается с одним из исходов:

- Тесты успешно пройдены
- Получен неправильный ответ на один из тестов
- Программа завершилась с ошибкой
- Программа превысила временное ограничение

Скрипт `scripts/test_mutants.sh` инициирует для каждого мутанта скрипт сборки, затем скрипт тестирования на заданном наборе тестовых последовательностей и сохраняет набор мутантов, успешно прошедших все тесты.

4. Автоматное описание системы, использованный метод построения тестовых последовательностей и построенное множество тестовых последовательностей

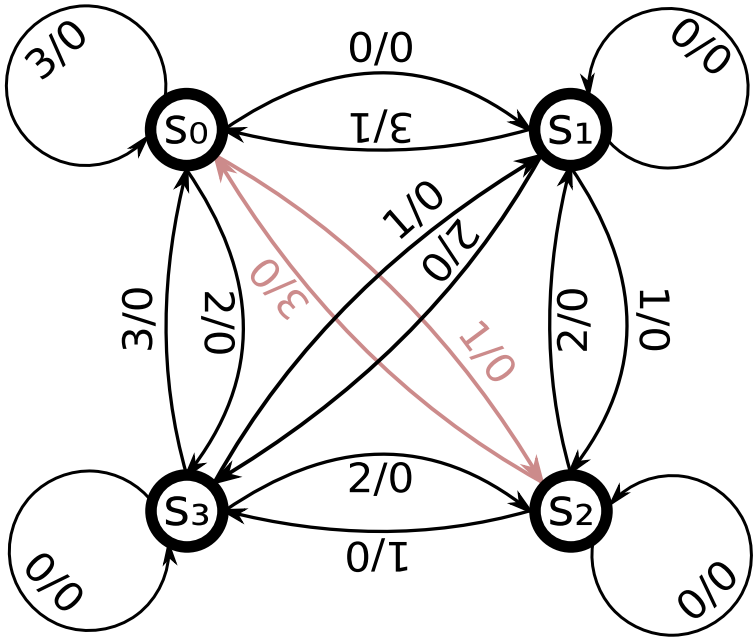

Рис. 1: Автоматное описание системы. Начальное состояние s_0

С целью упрощения формализации в виде автомата предложенной системы, её вход преобразуется следующей подстановкой:

$$0 \mid 3 \mid 6 \mid 9 \rightarrow 0$$
$$1 \mid 4 \mid 7 \rightarrow 1$$
$$2 \mid 5 \mid 8 \rightarrow 2$$
$$\text{<space>} \rightarrow 3$$

Суть этого преобразования в том, что мы считаем равнозначными цифры, дающие одинаковый остаток при делении на три. На рис. 1 графически изображен автомат, формально описывающий предложенную систему. Листинг 1 содержит описание этого автомата в формате FSM. Его реакцией на каждый допустимый входной символ являются 1, если было обнаружено очередное число, кратное трём, и 0 иначе. Данный автомат по сути считает сумму цифр в числе по модулю три, переходя в соответствующее значению остатка состояние. Корректность преобразования и работы автомата следует

из свойств взятия модуля от арифметических выражений:

$$(a_0 + a_1 \cdot 10 + \cdots + a_n \cdot 10^n) \ \% \ 3 =$$
$$((a_0 \ \% \ 3) + (a_1 \ \% \ 3) \cdot 10 + \cdots + (a_n \ \% \ 3) \cdot 10^n) \ \% \ 3 =$$
$$((a_0 \ \% \ 3) + (a_1 \ \% \ 3) + \cdots + (a_n \ \% \ 3)) \ \% \ 3$$

Для генерации множества тестовых последовательностей был использован метод «Black box/W» с заданными максимальными размерами конечного автомата 4 и 5, реализованный в системе `FSM Tester`[1]. В листингах 2 и 3 приведены сгенерированные тестовые последовательности.

Листинг 1: FSM-описание автомата

```
1  F 0
2  s 4
3  i 4
4  o 2
5  n0 0
6  p 16
7  0 0 1 0
8  0 1 2 0
9  0 2 3 0
10 0 3 0 0
11 1 0 1 0
12 1 1 2 0
13 1 2 3 0
14 1 3 0 1
15 2 0 2 0
16 2 1 3 0
17 2 2 1 0
18 2 3 0 0
19 3 0 3 0
20 3 1 1 0
21 3 2 2 0
22 3 3 0 0
```

[1] `http://www.fsmtestonline.ru`

Листинг 2: Тестовые последовательности для размера 4

```
1  0/0 0/0 0/0 3/1
2  0/0 0/0 1/0 3/0
3  0/0 0/0 3/1
4  0/0 1/0 0/0 3/0
5  0/0 1/0 1/0 3/0
6  0/0 1/0 3/0
7  0/0 2/0 0/0 3/0
8  0/0 2/0 1/0 3/1
9  0/0 2/0 3/0
10 0/0 3/1 0/0 3/1
11 0/0 3/1 1/0 3/0
12 0/0 3/1 3/0
13 1/0 0/0 0/0 3/0
14 1/0 0/0 1/0 3/0
15 1/0 0/0 3/0
16 1/0 1/0 0/0 3/0
17 1/0 1/0 1/0 3/1
18 1/0 1/0 3/0
19 1/0 2/0 0/0 3/1
20 1/0 2/0 1/0 3/0
21 1/0 2/0 3/1
22 1/0 3/0 0/0 3/1
23 1/0 3/0 1/0 3/0
24 1/0 3/0 3/0
25 2/0 0/0 0/0 3/0
26 2/0 0/0 1/0 3/1
27 2/0 0/0 3/0
28 2/0 1/0 0/0 3/1
29 2/0 1/0 1/0 3/0
30 2/0 1/0 3/1
31 2/0 2/0 0/0 3/0
32 2/0 2/0 1/0 3/0
33 2/0 2/0 3/0
34 2/0 3/0 0/0 3/1
35 2/0 3/0 1/0 3/0
36 2/0 3/0 3/0
37 3/0 0/0 3/1
38 3/0 1/0 3/0
39 3/0 3/0
```

Листинг 3: Тестовые последовательности для размера 5

```
1   0/0 0/0 0/0 0/0 3/1
2   0/0 0/0 0/0 1/0 3/0
3   0/0 0/0 0/0 3/1
4   0/0 0/0 1/0 0/0 3/0
5   0/0 0/0 1/0 1/0 3/0
6   0/0 0/0 1/0 3/0
7   0/0 0/0 2/0 0/0 3/0
8   0/0 0/0 2/0 1/0 3/1
9   0/0 0/0 2/0 3/0
10  0/0 0/0 3/1 0/0 3/1
11  0/0 0/0 3/1 1/0 3/0
12  0/0 0/0 3/1 3/0
13  0/0 1/0 0/0 0/0 3/0
14  0/0 1/0 0/0 1/0 3/0
15  0/0 1/0 0/0 3/0
16  0/0 1/0 1/0 0/0 3/0
17  0/0 1/0 1/0 1/0 3/1
18  0/0 1/0 1/0 3/0
19  0/0 1/0 2/0 0/0 3/1
20  0/0 1/0 2/0 1/0 3/0
21  0/0 1/0 2/0 3/1
22  0/0 1/0 3/0 0/0 3/1
23  0/0 1/0 3/0 1/0 3/0
24  0/0 1/0 3/0 3/0
25  0/0 2/0 0/0 0/0 3/0
26  0/0 2/0 0/0 1/0 3/1
27  0/0 2/0 0/0 3/0
28  0/0 2/0 1/0 0/0 3/1
29  0/0 2/0 1/0 1/0 3/0
30  0/0 2/0 1/0 3/1
31  0/0 2/0 2/0 0/0 3/0
32  0/0 2/0 2/0 1/0 3/0
33  0/0 2/0 2/0 3/0
34  0/0 2/0 3/0 0/0 3/1
35  0/0 2/0 3/0 1/0 3/0
36  0/0 2/0 3/0 3/0
37  0/0 3/1 0/0 0/0 3/1
38  0/0 3/1 0/0 1/0 3/0
39  0/0 3/1 0/0 3/1
40  0/0 3/1 1/0 0/0 3/0
41  0/0 3/1 1/0 1/0 3/0
42  0/0 3/1 1/0 3/0
43  0/0 3/1 2/0 0/0 3/0
44  0/0 3/1 2/0 1/0 3/1
45  0/0 3/1 2/0 3/0
46  0/0 3/1 3/0 0/0 3/1
47  0/0 3/1 3/0 1/0 3/0
48  0/0 3/1 3/0 3/0
49  1/0 0/0 0/0 0/0 3/0
50  1/0 0/0 0/0 1/0 3/0
51  1/0 0/0 0/0 3/0
52  1/0 0/0 1/0 0/0 3/0
53  1/0 0/0 1/0 1/0 3/1
54  1/0 0/0 1/0 3/0
55  1/0 0/0 2/0 0/0 3/1
56  1/0 0/0 2/0 1/0 3/0
57  1/0 0/0 2/0 3/1
58  1/0 0/0 3/0 0/0 3/1
59  1/0 0/0 3/0 1/0 3/0
60  1/0 0/0 3/0 3/0
61  1/0 1/0 0/0 0/0 3/0
62  1/0 1/0 0/0 1/0 3/1
63  1/0 1/0 0/0 3/0
64  1/0 1/0 1/0 0/0 3/1
65  1/0 1/0 1/0 1/0 3/0
66  1/0 1/0 1/0 3/1
67  1/0 1/0 2/0 0/0 3/0
68  1/0 1/0 2/0 1/0 3/0
69  1/0 1/0 2/0 3/0
70  1/0 1/0 3/0 0/0 3/1
71  1/0 1/0 3/0 1/0 3/0
72  1/0 1/0 3/0 3/0
73  1/0 2/0 0/0 0/0 3/1
74  1/0 2/0 0/0 1/0 3/0
75  1/0 2/0 0/0 3/1
76  1/0 2/0 1/0 0/0 3/0
77  1/0 2/0 1/0 1/0 3/0
78  1/0 2/0 1/0 3/0
79  1/0 2/0 2/0 0/0 3/0
80  1/0 2/0 2/0 1/0 3/1
81  1/0 2/0 2/0 3/0
82  1/0 2/0 3/1 0/0 3/1
83  1/0 2/0 3/1 1/0 3/0
84  1/0 2/0 3/1 3/0
85  1/0 3/0 0/0 0/0 3/1
86  1/0 3/0 0/0 1/0 3/0
87  1/0 3/0 0/0 3/1
88  1/0 3/0 1/0 0/0 3/0
89  1/0 3/0 1/0 1/0 3/0
90  1/0 3/0 1/0 3/0
91  1/0 3/0 2/0 0/0 3/0
92  1/0 3/0 2/0 1/0 3/1
93  1/0 3/0 2/0 3/0
94  1/0 3/0 3/0 0/0 3/1
95  1/0 3/0 3/0 1/0 3/0
96  1/0 3/0 3/0 3/0
97  2/0 0/0 0/0 0/0 3/0
98  2/0 0/0 0/0 1/0 3/1
99  2/0 0/0 0/0 3/0
100 2/0 0/0 1/0 0/0 3/1
101 2/0 0/0 1/0 1/0 3/0
102 2/0 0/0 1/0 3/1
103 2/0 0/0 2/0 0/0 3/0
104 2/0 0/0 2/0 1/0 3/0
105 2/0 0/0 2/0 3/0
106 2/0 0/0 3/0 0/0 3/1
107 2/0 0/0 3/0 1/0 3/0
108 2/0 0/0 3/0 3/0
109 2/0 1/0 0/0 0/0 3/1
110 2/0 1/0 0/0 1/0 3/0
111 2/0 1/0 0/0 3/1
112 2/0 1/0 1/0 0/0 3/0
113 2/0 1/0 1/0 1/0 3/0
114 2/0 1/0 1/0 3/0
115 2/0 1/0 2/0 0/0 3/0
116 2/0 1/0 2/0 1/0 3/1
117 2/0 1/0 2/0 3/0
118 2/0 1/0 3/1 0/0 3/1
119 2/0 1/0 3/1 1/0 3/0
120 2/0 1/0 3/1 3/0
121 2/0 2/0 0/0 0/0 3/0
122 2/0 2/0 0/0 1/0 3/0
123 2/0 2/0 0/0 3/0
124 2/0 2/0 1/0 0/0 3/0
125 2/0 2/0 1/0 1/0 3/1
126 2/0 2/0 1/0 3/0
127 2/0 2/0 2/0 0/0 3/1
128 2/0 2/0 2/0 1/0 3/0
129 2/0 2/0 2/0 3/1
130 2/0 2/0 3/0 0/0 3/1
131 2/0 2/0 3/0 1/0 3/0
132 2/0 2/0 3/0 3/0
133 2/0 3/0 0/0 0/0 3/1
134 2/0 3/0 0/0 1/0 3/0
135 2/0 3/0 0/0 3/1
136 2/0 3/0 1/0 0/0 3/0
137 2/0 3/0 1/0 1/0 3/0
138 2/0 3/0 1/0 3/0
139 2/0 3/0 2/0 0/0 3/0
140 2/0 3/0 2/0 1/0 3/1
141 2/0 3/0 2/0 3/0
142 2/0 3/0 3/0 0/0 3/1
143 2/0 3/0 3/0 1/0 3/0
144 2/0 3/0 3/0 3/0
145 3/0 0/0 0/0 3/1
146 3/0 0/0 1/0 3/0
147 3/0 0/0 3/1
148 3/0 1/0 0/0 3/0
149 3/0 1/0 1/0 3/0
150 3/0 1/0 3/0
151 3/0 2/0 0/0 3/0
152 3/0 2/0 1/0 3/1
153 3/0 2/0 3/0
154 3/0 3/0 0/0 3/1
155 3/0 3/0 1/0 3/0
156 3/0 3/0 3/0
```

5. Процедура генерации и описание построенного множества мутантов

Для генерации множества мутантов был использован проект с открытым исходным кодом `universalmutator`[2], основанный на регулярных выражениях. В листинге 4 приведён список правил для языка C, используемый мутатором для генерации программ-мутантов.

Листинг 4: Список правил-подстановок мутатора для языка C

```
(^\s*)(\S+[^{}]+.*)\n ==> \1/*\2*/\n
if (\(.*\)) ==> if (!\1)
if(\(.*\)) ==> if (!\1)
if (\(.*\)) ==> if (0)
if(\(.*\)) ==> if(0)
if (\(.*\)) ==> if (1)
if(\(.*\)) ==> if(1)
while (\(.*\)) ==> while (!\1)
while(\(.*\)) ==> while(!\1)
else ==>
\|\|.*\) ==> || 0
\|\|.*\s ==> || 0
\(.*\|\| ==> 0 ||
\s.*\|\| ==> 0 ||
&&.*\) ==> && 1
&&.*\s ==> && 1
\(.*&& ==> 1 &&
\s.*&& ==> 1 &&
// ==> SKIP_MUTATING_REST
```

Генерация мутантов и фильтрация успешно компилируемых среди них производился запуском соответствующего скрипта:

```
$ ./scripts/generate_mutants.sh
```

[2]Исходный код `universalmutator`: `https://github.com/agroce/universalmutator`.

6. Описание процедуры тестирования программной реализации и найденные ошибки

Прогон сгенерированных мутантов на тестовых набор был произведён следующими командами:

```
$ ./scripts/test_mutants.sh ./tests/tests_4.txt
   passing_mutants_4.txt
$ ./scripts/test_mutants.sh ./tests/tests_5.txt
   passing_mutants_5.txt
```

Было получено 240 мутантов, которые мы разделим на следующие группы:

1) 56 мутантов проходят тесты для автомата размера 4
    - из них 53 мутанта также проходят тесты для автомата размера 5
        - 24 мутанта при компиляции дают оригинальный ассемблер
        - 23 мутанта семантически эквивалентны изначальной программной реализации при заданных ограничениях на вход
        - 6 мутантов эквивалентны изначальной программной реализации при заданных ограничениях на вход и на компилятор
2) 148 мутантов выводят неправильный ответ на один из тестов
3) 13 мутантов завершаются с ошибкой исполнения программы
4) 23 мутанта не выводят ответ на тест за установленное временное ограничение в 1 секунду

Три мутанта из первой группы сразу показывают неполноту множества тестовых последовательностей для автомата размера 4. Суть каждого из них — проигнорировать числа, состоящие целиком из цифр 2 (не установить флаг, означающий считывание числа). Минимальным контрпримером к программной реализации с такой ошибкой является тестовый пример `2/0 2/0 2/0 3/1`, который отсутствует среди тестовых последовательностей для размера 4, но присутствует среди оных для размера 5.

Подмножество мутантов, проходящих все сгенерированные тесты

Некоторые мутанты не являются эквивалентными программами на уровне сгенерированного кода ассемблера, однако продолжают работать за счёт соблюдения некоторых ограничений при выполнении программы. Например, в листинге 5 приведены примеры, работающие по причине того, что остаток от деления 10 на 3 равен 1.

Листинг 5: Мутанты, использующие модульную арифметику

```
1  mutants/div.mutant.7.c
2  6c6
3  < const int BASE = 10;
4  ---
5  > const int BASE = 1;
6  mutants/div.mutant.106.c
7  33c33
8  <                 number_rem = (number_rem * BASE + 0) % MOD;
9  ---
10 >                 number_rem = (number_rem % BASE + 0) % MOD;
```

Шесть мутантов среди проходящих все тесты используют переполнение (примеры в листинге 6). Программа продолжает корректно работать за счёт ограничений на входную последовательность, а также поскольку 0 и -1 для типа unsigned имеют одинаковый остаток по модулю 3: $2^{32} - 1 \equiv 0 \ (3)$. Переполнение не относится к undefined behavior, и при соответствии компилятора стандарту языка C на большинстве распространённых систем программа будет продолжать работать. Однако стандарт C гарантирует лишь то, что тип unsigned занимает как минимум 16 бит[3], и если его ширина будет вида, отличного от $2^n, \ n \geqslant 4$, то программа работать перестанет. Поскольку на используемой для тестирования операционной системе ширина типа unsigned равна 32, никакие тесты не в состоянии опровергнуть таких мутантов, и в итоговом зачёте мы их учитывать не будем.

Листинг 6: Мутанты, использующие переполнение типа unsigned

```
1  mutants/div.mutant.104.c
2  33c33
3  <                 number_rem = (number_rem * BASE + 0) % MOD;
4  ---
5  >                 number_rem = (number_rem - BASE + 0) % MOD;
6  mutants/div.mutant.76.c
```

[3] https://en.cppreference.com/w/c/language/arithmetic_types

```
26c26
<         unsigned number_rem = 0;
---
>         unsigned number_rem = -1;
```

Подмножество мутантов, не укладывающихся во временное ограничение

В листинге 7 приведён пример мутанта, который вместо увеличения значения позиции в строке оставляет её неизменной, что приводит к зацикливанию на одном символе.

Листинг 7: Мутант, вызывающий зацикливание

```
mutants/div.mutant.43.c
18c18
< void emit(size_t* pos, char c) { ++*pos; putchar(c); }
---
> void emit(size_t* pos, char c) { +-*pos; putchar(c); }
```

Подмножество мутантов, завершающихся с ошибкой исполнения

Примеры мутантов, завершающихся с ошибкой, приведены в листинге 8: в первом примере мутатор переставил местами аргументы, подав на место указателя значение 0, и программа обратилась в память, в которую ей обращаться запретило ядро операционной системы; во втором примере мутатор заменил добавление нуля делением на него, и возникло исключение.

Листинг 8: Мутанты, завершающийся с ошибкой

```
mutants/div.mutant.97.c
32c32
<                 emit(&i, OUT_ZERO);
---
>                 emit( OUT_ZERO,&i);
mutants/div.mutant.101.c
33c33
<                 number_rem = (number_rem * BASE + 0) % MOD;
---
>                 number_rem = (number_rem * BASE / 0) % MOD;
```

Подмножество мутантов, дающих ошибочный ответ на одну из тестовых последовательностей

Бóльшая часть мутантов дают ошибочную реакцию на некоторые тестовые последовательности. Некоторые примеры таких мутантов приведены в листинге 9. Процесс их тестирования, содержащий тестовые последовательности, на которые мутанты дали неверные ответы, находится в листинге 10.

Листинг 9: Мутанты, выдающих ошибочную реакцию

```
mutants/div.mutant.38.c
16c16
< #define OUT_ONE '1'
---
> #define OUT_ONE '0'
mutants/div.mutant.58.c
22c22
<     while (s[i] != IN_END) {
---
>     if (s[i] != IN_END) {
mutants/div.mutant.182.c
43c43
<                 number_rem = (number_rem * BASE + 2) % MOD;
---
>                 /*number_rem = (number_rem * BASE + 2) % MOD;*/
mutants/div.mutant.44.c
18c18
< void emit(size_t* pos, char c) { ++*pos; putchar(c); }
---
> void emit(size_t* pos, char c) { --*pos; putchar(c); }
```

Листинг 10: Тестирование приведённых примеров ошибочных мутантов

```
$ ./scripts/build_with_mutant.sh mutants/div.mutant.38.c
$ python3 ./scripts/run_tests.py -v -e ./build/div tests/tests_4.txt
Failed on input: 0003
Program output: 0000
Expected output: 0001
$ ./scripts/build_with_mutant.sh mutants/div.mutant.58.c
$ python3 ./scripts/run_tests.py -v -e ./build/div tests/tests_4.txt
Failed on input: 0303
Program output: 01
Expected output: 0101
$ ./scripts/build_with_mutant.sh mutants/div.mutant.182.c
$ python3 ./scripts/run_tests.py -v -e ./build/div tests/tests_4.txt
Failed on input: 0203
Program output: 0001
Expected output: 0000
$ ./scripts/build_with_mutant.sh mutants/div.mutant.44.c
$ python3 ./scripts/run_tests.py -v -e ./build/div tests/tests_4.txt
Failed on input: 0003
Program output: 0
Expected output: 0001
```

7. Оценка полноты теста относительно множества построенных мутантов

Мутатором было построено множество из 240 мутантов, из которых 53 было отнесено к эквивалентным при некоторых описанных ограничениях. Среди оставшихся 187 мутантов — 3 мутанта проходят все сгенерированные тесты для автомата размера 4, но не проходят для размера 5. Таким образом:

- Полнота теста для размера 4 — $\frac{187-3}{187} \cdot 100\% \approx 98.4\%$
- Полнота теста для размера 5 — 100%

8. Заключение

В настоящей работе произведено следующее:

- предложены неформальное и автоматное описание системы
- произведена программная реализация системы, исходный код которой находится
- построены два множества тестовых последовательностей методом «Black box/W» по автоматному описанию системы
- сгенерировано множество мутантов программной реализации
- полученные мутанты были протестированы и классифицированы
- было показано, что одно из множеств тестовых последовательностей не является полным относительно сгенерированного множества мутантов, другое же оказалось полным

1. Приложение: исходный код

Листинг 11: scripts/transform_input.sh

```
#!/bin/bash
tr 0 0 | tr 3 0 | tr 6 0 | tr 9 0 | \
tr 1 1 | tr 4 1 | tr 7 1 | \
tr 2 2 | tr 5 2 | tr 8 2 | \
tr ' ' 3
```

Листинг 12: src/div.c

```
#include "div.h"
#include <stdbool.h>
#include <stddef.h>
#include <stdio.h>

const int BASE = 10;
const int MOD = 3;

#define IN_ZERO '0'
#define IN_ONE '1'
#define IN_TWO '2'
#define IN_SPACE '3'
#define IN_END '\0'

#define OUT_ZERO '0'
#define OUT_ONE '1'

void emit(size_t* pos, char c) { ++*pos; putchar(c); }

void scan_numbers_divisible_by_three(char* s) {
    size_t i = 0;
    while (s[i] != IN_END) {
        while (s[i] == IN_SPACE) {
            emit(&i, OUT_ZERO);
        }
        unsigned number_rem = 0;
        bool number_start = false;
        bool number_end = false;
        while (!number_end) {
            switch (s[i]) {
            case IN_ZERO:
                emit(&i, OUT_ZERO);
                number_rem = (number_rem * BASE + 0) % MOD;
                number_start = true;
                break;
            case IN_ONE:
                emit(&i, OUT_ZERO);
                number_rem = (number_rem * BASE + 1) % MOD;
                number_start = true;
                break;
            case IN_TWO:
                emit(&i, OUT_ZERO);
                number_rem = (number_rem * BASE + 2) % MOD;
                number_start = true;
                break;
            default:
```

```
                number_end = true;
            }
        }
        if (s[i] != IN_END) {
            if (number_start && number_rem == 0) {
                emit(&i, OUT_ONE);
            } else {
                emit(&i, OUT_ZERO);
            }
        }
    }
}
```

Листинг 13: src/main.c

```c
#define  _GNU_SOURCE
#include "div.h"
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int EXIT_CODE = 0;

int main() {
    char* input = NULL;
    size_t n = 0;
    if (getline(&input, &n, stdin) == -1) {
        printf("%s\n", strerror(errno));
        EXIT_CODE = 1;
        goto free_input;
    }
    scan_numbers_divisible_by_three(input);
free_input:
    free(input);
    return EXIT_CODE;
}
```

Листинг 14: scripts/generate_mutants.sh

```
#!/bin/bash
mutate ./src/div.c --mutantDir ./mutants --cmd "./scripts/build_with_mutant.sh MUTANT"
```

Листинг 15: scripts/run_tests.py

```python
import argparse
from typing import Iterator, Tuple
import subprocess
from sys import exit
from enum import Enum


def iter_test_samples(filename: str) -> Iterator[Tuple[str, str]]:
    with open(filename, "r") as test_file:
        for test in test_file.readlines():
```

```
            test_input, test_output = map(
                lambda x: "".join(x),
                zip(*map(lambda x: x.split("/"), test.strip().split())),
            )
            yield test_input, test_output


class TestingResult(Enum):
    SUCCESS = 0
    INCORRECT = 1
    TIMEOUT = 2
    ERROR = 3


def run_tests(*, tests_filename: str, executable: str, verbose: bool) -> TestingResult:
    for test_input, test_output in iter_test_samples(tests_filename):
        try:
            output = subprocess.check_output(
                [executable], input=test_input.encode(), timeout=1
            ).decode()
        except subprocess.TimeoutExpired:
            return TestingResult.TIMEOUT
        except subprocess.CalledProcessError:
            return TestingResult.ERROR
        assert type(output) is type(test_output)
        if output != test_output:
            if verbose:
                print("Failed on input:", test_input)
                print("Program output:", output)
                print("Expected output:", test_output)
            return TestingResult.INCORRECT
    return TestingResult.SUCCESS


if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("filename", type=str, help="Input FSM file")
    parser.add_argument(
        "-e", "--executable", type=str, help="Executable path", required=True
    )
    parser.add_argument(
        "-v", "--verbose", action="store_true", help="Whether to print fail case"
    )
    args = parser.parse_args()
    exit(
        run_tests(
            tests_filename=args.filename,
            executable=args.executable,
            verbose=args.verbose,
        ).value
    )
```

Листинг 16: scripts/build_with_mutant.sh

```
#!/bin/bash
MUTANT=${1:-./src/div.c}
rm -rf build
cmake -S. -Bbuild -DDIV_C=${MUTANT}
cmake --build build
```

Листинг 17: scripts/test_mutants.sh

```
#!/bin/bash
SCRIPT_DIR="$( cd -- "$( dirname -- "${BASH_SOURCE[0]}" )" &> /dev/null && pwd )"
TESTS_FILE=${1:-tests.txt}
PASSING_MUTANTS=${2-passing_mutants.txt}
> ${PASSING_MUTANTS}
for mutant in $(find mutants -type f -name "div.mutant.*.c"); do
    echo "***${mutant}***"
    sh ${SCRIPT_DIR}/build_with_mutant.sh ${mutant} &>/dev/null
    python3 ${SCRIPT_DIR}/run_tests.py -e build/div ${TESTS_FILE}
    exit_code=$?
    if [[ ${exit_code} = 0 ]]; then
        echo ${mutant} >>${PASSING_MUTANTS}
        diff --color ./src/div.c "${mutant}"
    elif [[ ${exit_code} = 2 ]]; then
        echo "***Timeout***"
    elif [[ ${exit_code} = 3 ]]; then
        echo "***Runtime error***"
    fi
done
```

Листинг 18: scripts/list_mutants_with_same_assembler.sh

```
#!/bin/bash
MUTANTS_LIST_FILE=${1:-passing_mutants.txt}

remove_first_line() {
    tail -n +2 "$1"
}

gcc -S -o div.s ./src/div.c

for mutant in $(cat "${MUTANTS_LIST_FILE}"); do
    gcc -S -o div_mutant.s ${mutant} &>/dev/null
    diff <(remove_first_line div.s) <(remove_first_line div_mutant.s) &>/dev/null && \
    { echo ${mutant}; diff --color ./src/div.c ${mutant} >&2; }
done

rm div.s div_mutant.s
```

Листинг 19: CMakeLists.txt

```
cmake_minimum_required(VERSION 3.13.4)
project(div LANGUAGES C)
set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -Wall -Wextra -pedantic -O2 -std=c99")

if (NOT DEFINED DIV_C)
    set(DIV_C src/div.c)
endif()

add_executable(div ${DIV_C} src/main.c)
target_include_directories(div PUBLIC include)
```