Министерство науки и высшего образования Российской Федерации

НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ

ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ (НИ ТГУ)

Институт прикладной математики и компьютерных наук

КУРСОВАЯ РАБОТА

Конечно-автоматное модульное тестирование программных реализаций на языке Java

Разенков Семен Игоревич

Направление подготовки 10.05.01 Компьютерная безопасность

Направленность (профиль) «Анализ безопасности компьютерных систем»

Руководитель работы

канд. физ.-мат. наук

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_А. С. Твардовский

*подпись*

«\_\_\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_20\_\_\_г.

Автор работы

студент группы № 931825

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ С.И. Разенков

*подпись*

«\_\_\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_20\_\_\_г.

Томск – 2022

**ОГЛАВЛЕНИЕ**

Введение 3

1 Основные понятия и определения 4

1.1 Конечные автоматы 4

2Методы построения проверяющих тестов 4

3 Платформа JUnit5 4

Выводы

Список использованных источников и литературы

**ВВЕДЕНИЕ**

Задачи идентификации имеют долгую историю исследований, которые в последние десятилетия воплотились в ряд прикладных механизмов анализа и синтеза для реактивных дискретных систем. Примерами таких систем могут быть телекоммуникационные протоколы, серверные и клиентские реализации которых могут быть описаны конечным автоматом для дальнейшего анализа.

При разработке современного программного обеспечения много внимания уделяется контролю качества, в частности, тестированию, которому подвергается продукт перед выпуском. Когда цена ошибки в системе велика, и существует необходимость гарантировать обнаружение ряда критичных ошибок, может быть использовано тестирование на основе формальных моделей.

Конечно-автоматные методы широко используются в тестировании программного обеспечения, поскольку имеют гарантированную полноту покрытия неисправностей[1]. В основу таких методов лёг подход к распознаванию автоматов, предложенный Василевским, позволяющий обнаружить неконформные реализации автоматов.

Некоторые из модификаций метода Василевского реализованы, доступны на открытых платформах[2] и могут быть использованы для синтеза тестов по автоматному описанию системы. Однако, высокая сложность (длина) этих тестов подталкивает к автоматизации тестирования, что, в свою очередь, требует подбора подходящего инструментария.

Одним из объектов тестирования являются реализации на объектно-ориентированных языках программирования, для проверки которых может быть использовано модульное тестирование[3]. Целью модульного тестирования является проверка некоторой части функционала программы на корректное поведение при определенных сценариях работы.

Одним из наиболее популярных объектно-ориентированных языков программирования является Java. Причинами тому являются в первую очередь кроссплатформенность и относительно небольшое время разработки [4].

В данной работе мы решаем задачу автоматизации модульного тестирования автоматными методами для классов на языке Java.

**1 ОСНОВНЫЕ ПОНЯТИЯ И ОПРЕДЕЛЕНИЯ**

**1.1 Конечные автоматы**

Конечным автоматом называется пятерка *A* = (*S*, *I*, *O*, *TS*, *s*0), где *S* - конечное множество состояний c выделенным начальным состоянием *s0*, *I* - входной алфавит, *O* - выходной алфавит и *TS* *S* × *O* × *I* × *S* – отношение переходов[4]. Четверка (*s*, *i*, *o*, *s*`) *TS*, описывает переход в автомате *A* из состояния *s* в состояние *s*` под действием входного символа *i* с выходным символом *o*.

Автомат называется наблюдаемым, если для любой тройки (*s*, *i*, *o*) *S* × *I* × *O*, существует не более одного состояния *s*` *S* такого, что (*s*, *i*, *o*, *s*`) *TS*.

Если для каждой пары (*s*,*i*) *S* × *I* существует хотя бы один переход (*s*, *i*, *o*, *s*`) *TS*, то автомат называется полностью определенным. В противном случае он называется частично определенным.

Автомат называется детерминированным, если для любой пары (*s*,*i*) *S* × *I* существует не более одной пары (*o*, *s*`) *O* × *S* такой, что (*s*, *i*, *o*, *s*`) *TS*. Иначе автомат называется недетерминированным.

В данной работе будут рассматриваться детерминированные полностью определённые конечные автоматы.

Входной последовательностью автомата называется последовательность входных символов, в которой за символом следует [3]. Записывается следующим образом: .

Выходная последовательность автомата представляет собой последовательность выходных символов, в которой за символом следует . Записывается: .

Пусть . Параназывается входо-выходной последовательностью если и только если существует последовательность состояний таких, что , …, .

Состояния *s* и *p* автоматов A1 и A2 соответственно эквивалентны, если автомат A1 в состоянии *s* и автомат A2 в состоянии *p* под воздействием любой входной последовательности выдают одинаковые выходные последовательности. Если состояния не эквивалентны, то они различимы.

Автомат называется приведенным, если любые два состояния в нем различимы[5], т.е. для них существует такая последовательность входных символов, при подаче которой выходные последовательности автомата, находившегося изначально в каждом из этих двух состояний, будут разными.

Пример. Рассмотрим конечный автомат, описывающий поведение муравья. Входной алфавит автомата представляет собой множество всех 3-битных чисел. Каждый разряд числа, начиная со старшего, соответственно показывает истинность утверждений: «видит муравейник», «видит листик», «видит хищника». Выходной алфавит представляет собой множество 2-битных чисел, характеризующих скорость движения муравья. Множество состояний автомата следующее: «сидит в муравейнике», «ищет листик», «тащит листик», «убегает от хищника». Сопоставим каждому состоянию соответственно числа от 0 до 3. Тогда получим следующий конечный автомат (рис. 1):

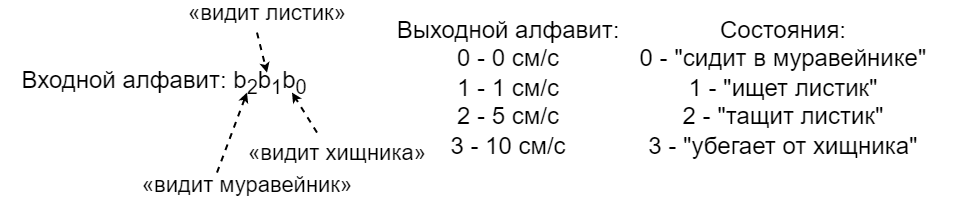
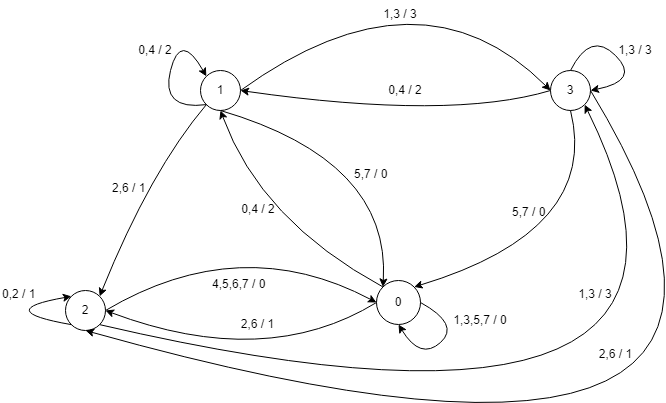


Рисунок 1. Диаграмма переходов конечного автомата.

Состояния на рисунке отмечены окружностями, представляющими собой вершины ориентированного графа, а вместе сдугами они показывают отношения переходов. Каждая дуга подписана входными значениями, вызывающими данный переход из одного состояния в другое, а также соответствующими выходными значениями.

**2МЕТОДЫ ПОСТРОЕНИЯ ПРОВЕРЯЮЩИХ ТЕСТОВ**

**2.1 Проверяющие тесты**

Модель неисправности конечного автомата представляет собой тройку: эталон – конечный полностью определенный детерминированный приведённый инициальный автомат с конечным числом состояний; область неисправности – множество конечных полностью определенных детерминированных инициальных автоматов с тем же выходным алфавитом; отношение конформности – отношение эквивалентности. Отметим, что определение области неисправности может различаться в зависимости от используемого метода синтеза теста и будет более детально рассмотрено в последующих разделах.

Формально, модель неисправности может быть определена следующим образом: *FM* = <*S*, ≅, *FDm>*, где *S* – автомат спецификация, ≅ – отношение эквивалентности, *FDm* – область неисправности, содержащая множество конечных автоматов с такими же входным и выходным алфавитами как и у спецификации и не более чем *m* состояниями. Соответственно, реализация конформна спецификации, если соответствующие автоматы эквивалентны.

Задание модели неисправности для того или иного метода синтеза теста позволяет формально определить и доказать гарантированную полноту покрытия неисправностей. Область неисправности задаёт тот класс ошибок, который гарантированно обнаруживается простроенным тестом.

Полным проверяющим тестом называется конечное множество входных последовательностей конечной длины, по реакциям на которые можно отличить всякую неконформную реализацию из области неисправности. Если проверяемый автомат на последовательности теста реагирует так же, как эталонный автомат, то гарантируется, что на все остальные последовательности он будет реагировать, как эталон. Длиной теста называется сумма длин всех его последовательностей.

Можно различают ошибки двух видов: ошибки переходов и ошибки выходов. Первые характеризуются неправильным состоянием, в которое переходит автомат при некотором входном воздействии. Ошибкой выхода является присутствие в отношении перехода выходного символа, не совпадающего с символом, присутствующем в соответствующем отношении перехода эталонного автомата.

**2.2 Методы построения**

Примерами методов построения проверяющих тестов являются transition tour и метод Василевского (имеет много модификаций)[6].

Передаточной последовательностью для некоторого состояния эталонного автомата *s* является последовательность входных символов (возможно, пустая), по которой можно достигнуть состояния s из начального состояния.

Множество достижимости – это множество передаточных последовательностей для всех состояний эталонного автомата.

Множество различимости представляет собой множество, состоящее их входных последовательностей, по реакции на которые можно различить любую пару состояний.

Метод transition tour представляет собой метод построения теста обходом графа переходов для автомата. Порядок егопостроения следующий:

1. Формируется множество достижимости, т.е. для каждого состояния *s* автомата *A* cтроится передаточная последовательность α*s*
2. Для каждого состояния *s*, добавляем в тест все последовательности α*si*, где *i*∈*I.*

Однако, у данного алгоритма есть недостаток, заключающийся в большом количестве тестовых последовательностей при большой области неисправности. Данный метод гарантирует обнаружение ошибок выхода при отсутствии ошибок переходов.

Построение теста методом Василевского осуществляется в следующем порядке:

1. cтроится передаточная последовательность α*s* для каждого состояния *s* автомата*A*(т.е. множество достижимости)
2. Для каждого состояния *s*, добавляются в тест все последовательности α*si*, где *i*∈*I*∪ε, ε - пустой символ (покрытие переходов).
3. К каждой последовательности теста добавляется диагностическая (различающая) последовательность автомата *A* из множества различимости.

Тестовые последовательности могут быть построены в виде дерева, корнем которого является начальное состояние, ребрам соответствуют входные символы с соответствующими выходными символами из полученных в 1-3 последовательностей. Вершины обозначаются состояниями, в которые переходит автомат. Дерево строится сверху вниз. Если некоторая последовательность является началом другой, то вторая будет достраиваться в графе поверх ветки первой последовательности.

Этот метод является полным проверяющим тестом и позволяет обнаружить все ошибки из области неисправности *FDm*, т.е. обнаруживает все ошибки переходов и выходов при условии, что количество состояний у автоматов из области неисправности не больше, чем у эталонного автомата, и выходные алфавиты совпадают.

**3 JUNIT-ТЕСТЫ**

**3.1 Платформа JUnit5**

JUnit-тест представляет собой Java-класс, в котором реализованы тестовые методы, отвечающие за проверку интересующего функционала и вызываемые при запуске тестирования. Как правило, в них используется механизм Assertions, который при обнаружении некорректного поведения тестируемой системы, выводит ошибку, которая далее анализируется разработчиком.

JUnit 5 представляет собой программный комплекс для тестирования программного обеспечения на языке Java [7].Он состоит из следующих модулей: JUnit Platform, JUnit Jupiter и JUnit Vintage.

JUnit Platform предоставляет программный интерфейс для разработки тестов и предназначен для их запуска на виртуальной машине Java (JVM). В данном модуле есть дополнительное приложение для запуска тестов из командной строки, которое было использовано в данной работе .JUnit Platform поддерживается многими средами разработки ПО.

JUnit Jupiter предоставляет некоторые обновленные и дополнительные возможности для написания тестов. Например, дополнить методы JUnit-теста аннотациями, определяющими параллельное выполнение тестов, динамическую генерацию тестов, вспомогательные методы, параметризацию и т.д. JUnit Vintage позволяет запускать тесты, разработанные для более старых версий JUnit (JUnit 3 JUnit 4).

* 1. **Реализация JUnit-теста**

Предполагается, что тестируется реализация некоторой системы, работа которой представима в виде инициального конечного детерминированного конечного автомата, на языке программирования Java. JUnit-тесту в виде текстовых файлов (.csv, .txt) подаются на вход: название класса, реализующего систему (т.е. автомат), таблица соответствия входных символов автомата методам класса с соответствующими наборами аргументов, таблица соответствия выходных символов автомата значениям возвращаемых методами класса объектов, проверяющий тест для автомата, построенный произвольным методом. Ниже представлены примеры таких таблиц.

Таблица входной символ – вызов метода класса имеет следующий вид:

*0,returnSpeed,Boolean:false Boolean:true*

*1,returnSpeed,Boolean:false Boolean:false*

Здесь входному символу 0 соответствует вызов метода *returnSpeed* с тремя аргументами – булевыми значениями *false, true.* Количество строк в таблице соответствует мощности входного алфавита.

Таблица выходной символ – возвращаемое значение:

*0,Integer:2*

*1,Integer:0*

Выходному символу 1 автомата будет соответствовать значение 0 целочисленного типа Integer в языке Java.

Проверяющий тест для автомата:

*0/2 0/2 0/2 0/2*

*0/2 0/2 0/2 1/3*

*0/2 0/2 1/3 0/2*

*0/2 0/2 1/3 1/3*

Каждая строка представлена тестовой последовательностью. Например, первая тестовая последовательность содержит входную последовательность *0000*, под действием которой автомат должен выдавать выходную последовательность *2222*.

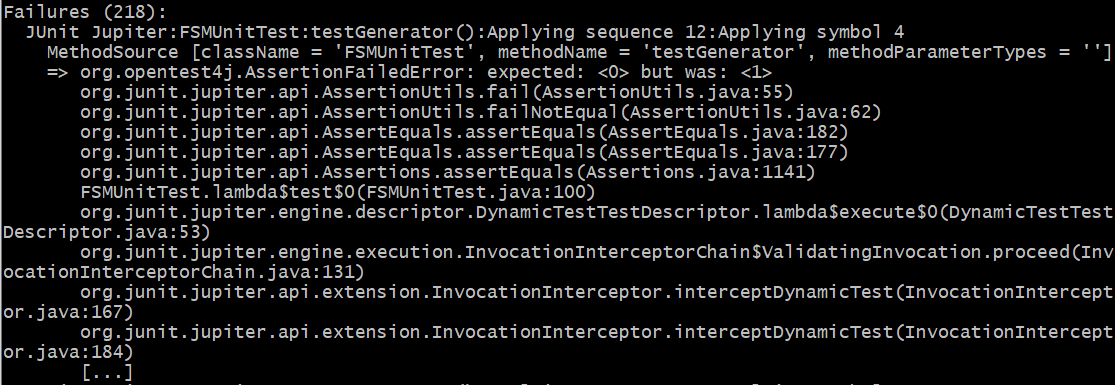
Кроме того, необходимо наличие скомпилированного Java-класса (файл .class), реализующего необходимый конечный автомат.

Комплекс для тестирования состоит из ConsoleLauncher для JUnit Platform, скомпилированного JUnit-теста, а также двух вспомогательных Java-классов, хранящих во время тестирования информацию из таблиц входной символ – вызов метода класса и выходной символ – возвращаемое значение.

Программа тестирования запускается из командной строки с помощью shell-скрипта, что делает её кроссплатформенной. Сначала она с помощью системного загрузчика классов ClassLoader[6] путем вызова метода loadClass динамически, т.е. непосредственно после запуска JUnit-теста, подгружает скомпилированный заранее Java-класс, реализующий автомат. Затем из упомянутых выше текстовых файлов считывает информацию, необходимую для тестирования реализации.

JUnit позволяет добавлять генераторы тестов (аннотация @TestFactory), которые создают unit-тесты (или контейнеры с множеством unit-тестов) динамически, во время выполнения JUnit-теста. В данной работе этот функционал использован для генерации контейнеров unit-тестов, каждый из которых соответствует некоторой тестовой последовательности. Каждый тест внутри контейнера соответствует элементу тестовой последовательности. Для каждого контейнера (т.е. каждой тестовой последовательности) создается новый объект – экземпляр тестируемого автомата. В случае несоответствия значения возвращаемого объекта нужному выходному символу автомата программа выдаст ошибку и информацию о том, в какой тестовой последовательности и на каком входном символе было обнаружено несовпадение поведения реализации автомата и эталона. На Рисунке 1 сообщается о 218 ошибках при выполнении теста, одна из которых возникает при подаче 4 символа 12 входной последовательности.

Рисунок 1. Выводимая JUnit-тестом информация об ошибках



В случае отсутствия ошибок вывод программы будет пуст. **4 ЭКСПЕРИМЕНТАЛЬНЫЕ РЕЗУЛЬТАТЫ**

Разработанный инструмент был апробирован на реализации конечного автомата, представленного в параграфе 1. Реализация представляет собой класс, полем которого является целочисленная переменная, в которой хранится текущее состояние автомата. Кроме того, реализован один метод, принимающий на вход три булевы переменные (соответствующие цифрам двоичного представления входного символа автомата) и возвращающий целое число (выходной символ).

В первую очередь, было написано текстовое описание автомата в особом формате, который принимается на вход сервисом *fsmtestonline.ru* [2], следующего вида:

*F 0 /\* фиксированный заголовок \*/*

*s 4 /\* количество состояний, в данном случае 4 \*/*

*i 8 /\* количество входных символов, в данном случае 8 \*/*

*o 4 /\* количество выходных символов, в данном случае 4 \*/*

*n0 0 /\* начальное состояние, в данном случае 0 \*/*

*p 32 /\* количество переходов \*/*

*0 0 1 2 /\* переход (s = 0, i = 0, s’=1, o = 2)\*/*

*... /\* остальные переходы, по одному в каждой строке \*/*

В итоге получен текстовый файл с тестовыми последовательностями, аналогичный описанному в п.3.2. Тестирование на полученных последовательностях исходной реализации автомата было успешно пройдено. При попытке внести в реализацию ошибку выхода, указывая неверное возвращаемое значение (т.е. выходной символ), либо ошибку перехода, неправильно меняя значение поля объекта автомата (т.е, неверное следующее состояние), то JUnit-тест в обоих случаях начинает выдавать ошибку.

Была дополнительно опробована экспериментальная на момент выполнения данной работы возможность распараллеливать выполнение контейнеров unit-тестов (или просто unit-тестов), реализуемая с помощью аннотации *@Execution(ExecutionMode.CONCURRENT)*. Во время выполнения тестирования параллельное тестирование последовательностей позволило сократить время выполнения тестов на 10%, что позволяет сделать вывод о пользе данной возможности для проведения тестирования программных реализаций конечных автоматов. При тестировании более сложных автоматных систем выигрыш может быть более существенным.

**ВЫВОДЫ**

Разработан и протестирован способ автоматизации конечно-автоматного тестирования программных реализаций на языке Java с помощью комплекса JUnit 5 и возможности динамической загрузки классов Java.

Созданная программа позволяет обнаруживать все неисправности, доступные для используемого при этом метода построения проверяющих тестов. Предполагается, что автоматное поведение заключено в одном Java-классе, поля которого характеризуют состояние автомата. Отношения переходов при этом реализованы с помощью функций, принимающих в виде набора аргументов входной символ и осуществляющих переход в следующее состояние путем изменения значений полей класса. Возвращаемый методом объект соответствует выходному символу автомата.

Однако, используя те же инструменты, можно реализовать и другие, более сложные реализации автоматных систем. Например, функционал может быть заключен не целиком одном в одном Java-классе, а в нескольких. Тогда потребуется динамическая загрузка не одного, а нескольких Java-классов, чтобы создать объект-экземпляр автомата. Кроме того, возвращаемое значение может не возвращаться методом, а присваиваться переменной, чей указатель в качестве аргумента подан функции. При каждом отдельном способе реализации конечного автомата появляются свои особенности автоматизации модульного тестирования.

Выполнение тестовых последовательностей в параллельном режиме позволяет ускорить тестирование.

Модульное тестирование широко распространено во многих языках программирования, и даже с недавнего времени стало использоваться в языках описания аппаратуры [11], с помощью которых описываются цифровые схемы, практически всегда представляющие собой некоторые конечные автоматы. Следовательно, автоматизация конечно-автоматных методов тестирования в таких системах является актуальной задачей.

**СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ**

1 Chow T. S. Test software design modeled by finite state machine / T. S. Chow // IEEE Transactions on Software Engineering. – 1978. – Vol. 4, № 3. – P. 178–187.

2 Test generation for FSM and EFSM. URL: <http://fsmtestonline.ru/>

3 Osherove, R. The Art of Unit Testing: with examples in C# / R.Osherove - 2nd ed. – Manning Publications – 2014.

4 Эккель Б. Философия Java. 4-е полное изд. – СПб.: Питер, 2016 – 1168 с.

5 Евтушенко Н.В. Недетерминированные автоматы: анализ и синтез. Ч. 2: Решение автоматных уравнений: учеб. пособие / Н. В. Евтушенко, М. В. Рекун, С. В. Тихомирова. – Томск: Томский государственный университет, 2009. – 111 с.

6 Гилл А. Введение в теорию конечных автоматов / А. Гилл. – М. : Издательство Наука, 1966. – 272с.

7 Агибалов Г. П. Лекции по теории конечных автоматов / Г. П. Агибалов, А. М. Оранов. – Томск : Издательство ТГУ, 1984. – 185 с.

8 M. P. Vasilevskii, "Failure diagnosis of automata," Kibernetika (Transl.), no.4, pp. 98-108, July-Aug. 1973.

9 JUnit 5 User Guide – URL: <https://junit.org/junit5/docs/current/user-guide/#writing-tests-dynamic-tests>

10 Java™ Platform Standard Edition 8 API Specification – URL: <https://docs.oracle.com/javase/8/docs/api/java/lang/ClassLoader.html>

11 VUnit: a test framework for HDL. URL: <https://vunit.github.io/>