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 5

2 Методы построения проверяющих тестов 8

2.1 Проверяющие тесты 8

2.2 Методы построения 9

3 JUnit-тесты 11

3.1 Платформа JUnit 5 11

3.2 Реализация JUnit-теста 11

4 Экспериментальные результаты 14

Выводы 16

Список использованных источников и литературы 17

**ВВЕДЕНИЕ**

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, могут быть использованы методы анализа на основе формальных моделей.

Задачи идентификации конечных автоматов имеют долгую историю исследований и 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.

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

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]. Соответственно, при реализации класса на языке Java, такой класс может быть с некоторой точностью описан соответствующим конечным автоматом. Далее, проверяющий тест строится на основе конечного автомата и подаётся на тестируемую реализацию.

В данной работе мы решаем задачу автоматизации модульного тестирования автоматными методами для классов на языке Java, а именно разрабатываем программное обеспечение для конвертации конечно-автоматных тестов в модульные тесты и их подачи на соответствующий класс в языке Java.

**1 ОСНОВНЫЕ ПОНЯТИЯ И ОПРЕДЕЛЕНИЯ**

Конечным автоматом называется пятерка *A* = (*S*, *I*, *O*, *TS*, *s*0), где *S* - конечное множество состояний c выделенным начальным состоянием *s*0, *I* – входной алфавит, *O* – выходной алфавит и *T*S*S*×*O*×*I*×*S* – отношение переходов [5]. Четверка (*s*, *i*, *o*, *s*`)*T*S, описывает переход в автомате *A* из состояния *s* в состояние *s*` под действием входного символа *i* с выдачей выходного символа *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 частичным.

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.

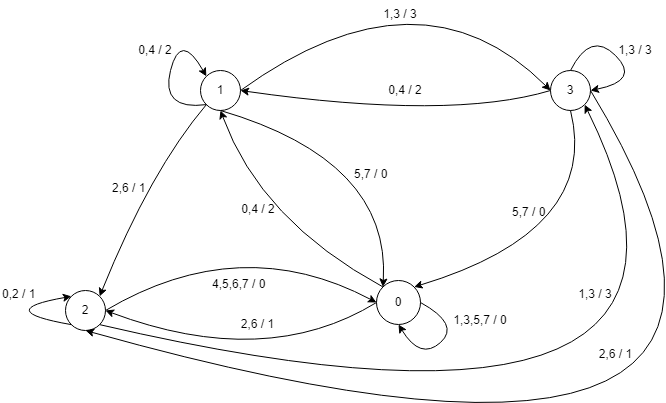
В данной работе будут рассматриваться детерминированные полностью определённые конечные автоматы.

Входной последовательностью автомата называется последовательность входных символов[6], которая может быть записана как .

Выходная последовательность автомата представляет собой последовательность выходных символов, которая может быть записана как .

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

Пример. Конечный автомат является удобной формой представления для простого (слабого) искусственного интеллекта. 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):



Изображение выглядит как стол

Автоматически созданное описание

Рисунок 1. Диаграмма переходов конечного автомата.

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

Рассмотрим пример вход-выходных последовательностей для автомата на рисунке 1. Пусть автомат находится в состоянии 0. Тогда (214,132) будет вход-выходной последовательностью, т.к. по представленным входным символам последовательно осуществляются переходы в состояния 2, 3, 1 с выходными реакциями 1, 3, 2 соответственно.

Состояния *s* и *p* автоматов *A*1 и *A*2 соответственно эквивалентны, если автомат *A*1 в состоянии *s* и автомат *A*2 в состоянии *p* под воздействием любой входной последовательности выдают одинаковые выходные последовательности. Если состояния не эквивалентны, то они различимы. Два конечных автомата эквивалентны, если эквивалентны их начальные состояния.

Например, состояния 1 и 3 в примере выше являются эквивалентными. По входным символам 2, 5, 6, 7 из них осуществляются переходы в одинаковые состояния с одинаковыми выходными символами, а по символам 0, 1, 3, 4 они переходят друг в друга, давая одинаковые выходные реакции.

Автомат называется приведенным, если любые два состояния в нем различимы[7], 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.

Заметим, что автомат а рисунке 1 не является приведённым, однако, может быть минимизирован (Рисунок 2) при помощи алгоритма из [6].

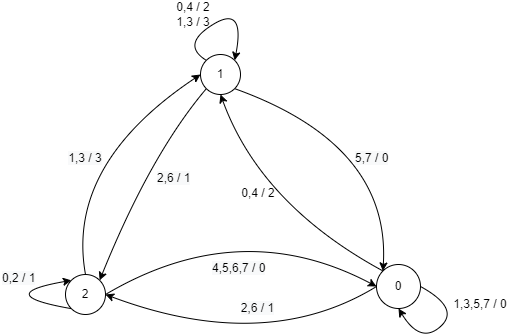


Рисунок 2. Диаграмма переходов приведенного конечного автомата

**2 МЕТОДЫ ПОСТРОЕНИЯ ПРОВЕРЯЮЩИХ ТЕСТОВ**

**2.1 Проверяющие тесты**

Модель неисправности конечного автомата представляет собой тройку: эталон (спецификация) – конечный полностью определенный детерминированный приведённый инициальный автомат; область неисправности – множество конечных полностью определенных детерминированных инициальных автоматов с тем же выходным алфавитом; отношение конформности – отношение соответствия между эталоном и корректной реализацией, например, отношение эквивалентность. 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 построенным тестом.

Полным проверяющим тестом называется конечное множество входных последовательностей конечной длины, по реакциям на которые можно отличить всякую неконформную реализацию из области неисправности от спецификации. Если проверяемый автомат реагирует на последовательности теста 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.

При тестировании на основе конечных автоматов, чаще всего выделяют ошибки двух видов: ошибки переходов и ошибки выходов. Первые характеризуются неправильным состоянием, в которое переходит автомат при некотором входном воздействии. Ошибкой выхода называется несовпадение выходного символа в некотором отношении перехода у проверяемого и эталонного автоматов.

Например, если для автомата на рисунке 2, построить неконформную реализацию, которая будет переходить в состояние 1 при подаче входного символа 2 в состоянии 0, то внесённая неисправность является ошибкой перехода. Если построить неконформную реализацию путём изменения выходной реакции в переходе (0,2,1,2), поменяв его на (0,2,0,2), то такая неисправность будет ошибкой выхода.

Отметим также, что существуют неисправности, которые заключаются во внесении дополнительных состояний в реализацию, но анализ таких ошибок остаётся за рамками данной работы.

**2.2 Методы построения**

Примерами методов построения проверяющих тестов являются метод обхода графа переходов и метод Василевского, который, в свою очередь, имеет много модификаций[8].

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 для автомата *A* – это множество передаточных последовательностей для всех состояний автомата *A*.

Множество различимости для автомата *A* представляет собой множество входных последовательностей, по реакциям на которые можно различить любую пару состояний автомата *A*. Множество может как состоять из одной диагностической последовательности, различающей любую пару состояний автомата, так и включать отдельную последовательность для каждой пары состояний. Существуют также и другие способы формирования множества различимости, такие как идентификаторы состояний, но они не рассматриваются в данной работе.

Обход графа переходов представляет собой метод построения теста путем обхода всех переходов автомата. Алгоритм его построения 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: обход графа переходов гарантирует обнаружение ошибок выходов лишь при отсутствии ошибок переходов. Тем не менее, такой тест имеет небольшую длину относительно метода Василевского и может быть использован для обнаружения выходных неисправностей.

Построение теста методом Василевского осуществляется следующим алгоритмом:

1. Строится 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, где 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 (Рисунок 3), корнем которого является начальное состояние, ребрам соответствуют входные символы с соответствующими выходными символами из полученных в пункте 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. В итоге будут получены тестовые последовательности, количество которых равно количеству листьев дерева.

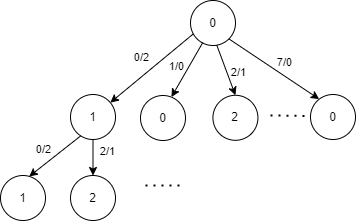


Рисунок 3. Построение дерева тестовых последовательностей

Метод Василевского позволяет получить полный проверяющий тест и 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, а входные алфавиты реализации и спецификации совпадают.

**3 JUNIT-ТЕСТЫ**

**3.1 Платформа JUnit 5**

JUnit 5 представляет собой программный комплекс для тестирования программного обеспечения на языке Java [9].

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 выводит ошибку при обнаружении некорректного поведения тестируемой системы, позволяя разработчику проанализировать обнаруженную неисправность.

JUnit 5 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 (ConsoleLauncher), которое было использовано в данной работе. JUnit Platform поддерживается многими средами разработки ПО.

JUnit Jupiter предоставляет некоторый обновленный и дополнительный функционал для написания тестов. Например, возможность дополнить методы JUnit-теста аннотациями, определяющими вспомогательные методы, параметризацию, параллельное выполнение тестов, динамическую генерацию тестов и т.д.

JUnit Vintage позволяет запускать тесты, разработанные для более старых версий JUnit (JUnit 3 и JUnit 4).

* 1. **Реализация JUnit-теста**

В настоящей работе предполагается, что тестируемая реализация является некоторым классом, написанным на языке программирования Java. Более того, работа такой реализации представима в виде инициального детерминированного конечного автомата частично или полностью.

При построении конечного автомата по классу, входным символам ставятся в соответствие вызовы методов класса с определенными значениями аргументов, а состояния – значениям полей класса. Если поля класса не могут быть отражены в состояниях автомата, то они могут быть исключены из рассмотрения.

Разработанное программное обеспечение представляет собой инструмент для конвертации конечно-автоматных тестов, построенных над абстрактными алфавитами автомата в JUnit-тесты, подачи таких тестов на тестируемый класс и вынесении вердикта о конформности реализации спецификации. Далее разработанный инструмент будем называть JUnit-тест.

JUnit-тесту в виде текстовых файлов (.csv, .txt) подаются на вход: название класса, реализующего систему, 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 двумя 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 содержит тестовую последовательность. 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 нужный конечный автомат.

Программа 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, по сути, является обычной программой на языке Java и запускается из командной строки с помощью shell-скрипта, что делает её кроссплатформенной. С помощью системного загрузчика классов ClassLoader[10], путем вызова метода loadClass, динамически, т.е. непосредственно после запуска, JUnit-тест подгружает скомпилированный заранее Java-класс, который необходимо протестировать. 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, т.е. 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 с эталоном. На Рисунке 4 приведен пример вывода JUnit-теста. В нем сообщается о наличии 218 ошибок при выполнении теста, одна из которых возникает при подаче 4 символа 12 входной последовательности.

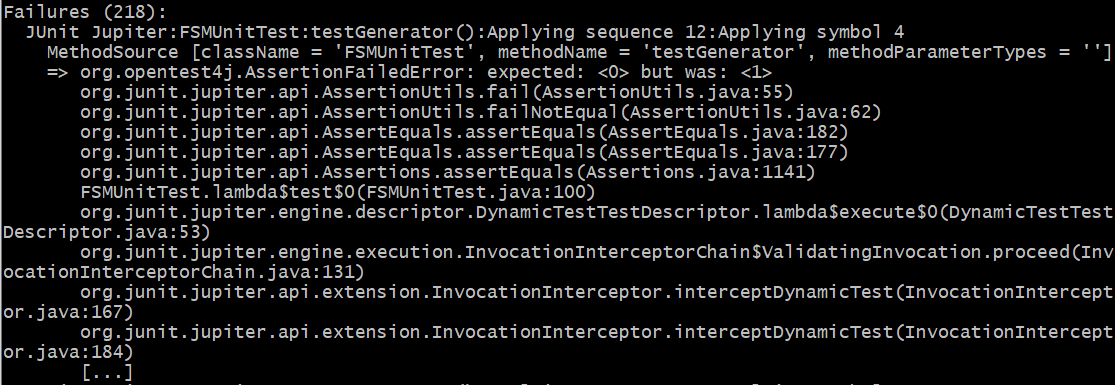


Рисунок 4. Выводимая JUnit-тестом информация об ошибках

В случае отсутствия ошибок вывод программы будет пуст. При необходимости, вывод JUnit-теста можно записать в xml-файл для дальнейшего анализа.

**4 ЭКСПЕРИМЕНТАЛЬНЫЕ РЕЗУЛЬТАТЫ**

Разработанный инструмент был апробирован на реализации конечного автомата, представленного в разделе 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, в данном случае 32\*/*

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

*... /\* остальные31 переходов, по одному в каждой строке \*/*

Сервисом генерируется текстовый файл с тестовыми последовательностями, имеющий вид, аналогичный описанному в п.3.2. Для метода Василевского получается тест, состоящий из 352 последовательностей, а для метода обхода гораздо более короткий – из 3 последовательностей.

При построении теста методом Василевского есть возможность указать максимальное количество (m) состояний в автомате из области неисправности (*FDm*). Кроме того, если автомат не приведенный, то сервис автоматически предложит его минимизировать.

Тестирование на полученных последовательностях исходной реализации автомата было успешно пройдено. При попытке внести в реализацию ошибку выхода, указывая неверное возвращаемое методом значение (т.е. выходной символ), либо ошибку перехода, неправильно меняя значение поля объекта автомата (получая неверное следующее состояние) внутри метода, JUnit-тест выдал сообщение об ошибке (Рисунок 4).

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 параллельная подача тестовых последовательностей позволила 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 сокращение времени 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 методов класса, 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 тестирования автоматных систем. Например, функционал может быть заключен не целиком 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 сократить время тестирования.

Модульное тестирование широко распространено во многих языках программирования, и даже с недавнего времени стало активно 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 для дальнейших исследований.

Следующие этапы данной работы могут заключаться в использовании более сложных методов синтеза тестов и переносе разработанного инструмента на другие языки.

**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/>