**Конспект по курсу**

**«Тестирование ПО» (2016)**

*Преподаватель: Кулямин Виктор Вячеславович*

*kuliamin@ispras.ru*

*Редактор: avasite*

*Сайт курса:* [*http://mbt-course.narod.ru/*](http://mbt-course.narod.ru/)

*У преподавателя есть хорошие лекции!*

**Содержание**

[1. Сдача курса 1](#_Toc468573071)

[2. Качество ПО (лекция кратко) 2](#_Toc468573072)

[3. Методы обеспечения и контроля качества ПО (лекция кратко) 2](#_Toc468573073)

[4. Цели, задачи и виды тестирования 3](#_Toc468573074)

[5. Организация тестовых наборов 4](#_Toc468573075)

[5.1. Машинная арифметика (лирическое отступление) 6](#_Toc468573076)

[6. Модели поведения систем 6](#_Toc468573077)

[7. Модели ситуаций. Критерий полноты тестирования 9](#_Toc468573078)

[8. Методы построения тестов 11](#_Toc468573079)

[9. Комбинаторные методы построения тестов 12](#_Toc468573080)

[10. Автоматные методы построения тестов 14](#_Toc468573081)

[11. Автоматные методы построения тестов. 15](#_Toc468573082)

[12. Технология UniTESK 16](#_Toc468573083)

1. Сдача курса

17.11, 24.11 – пары

1.11 – пары не будет

Контрольная работа – 8.12 (набор тестовых заданий на 45 минут)

Что-то простое (по мне так смахивает на IQ-test нужно будет быстро соображать)

Экзаменационная работа – 15.12 (набор задач на 1,5 часа)

Построить полный набор CDC

Задание на domain testing (есть некоторая функция, нужно нарисовать области и расставить там точки)

Задание на покрывающий набор (так чтобы все комбинации были накрыты)

Нарисован конечный автомат, и нужно либо найти различающую последовательность, либо сказать, что её нету, и с помощью W тестов нужно построить разбор.

Пользоваться можно чем угодно.

Если с первого раза – написано плохо (двойка), то можно будет переписать, но при переписывании вводится пенальти (и чтобы написать на ту же оценку, нужно будет уже постараться получше).

1. Качество ПО (лекция кратко)

Качество ПО: (**ISO 25010**)

1. Функциональность
2. Надёжность
3. Производительность
4. Удобство использования
5. Удобство сопровождения
6. Переносимость
7. Совместимость
8. Защищённость

Требования к ПО:

1. Стандарты
2. Нормы
3. Экспертиза
4. Закономерности
5. Пожелания и нужды

Выявление:

1. Источники
2. Извлечение
3. Согласование, систематизация и описание

ТЗ(RS):

1. Однозначность
2. Согласованность
3. Полнота
4. Адекватность
5. Проверяемость
6. Прослеживаемость
7. Модифицируемость
8. Систематичность

Ошибки:

1. defect
2. failure – сбой
3. fault – ошибка (место в коде, где что-то не так написано)
4. error – ошибка в понимании разработчика

Источники ошибок:

1. неправильное понимание задачи
2. неправильное решение
3. неправильная запись решения
4. Методы обеспечения и контроля качества ПО (лекция кратко)

Ошибки –

* Предотвращение
	+ Стандартизация
	+ Высокоуровневые языковые конструкции (Null Pointer Exception: 40-60% - ошибок) (необнуляемые ссылки, либо проверка на null)
* Обнаружение ошибок (контроль)
	+ Предварительная экспертиза
		- Валидация (человек проверяет код)
		- Верификация (сопоставление пар: требование-код, требование-тест)

Виды верификации:

* + - * Экспертиза людьми
			* Статический анализ (программа находит несоответствия)
			* Динамический анализ
			* Формальная верификация (спецификация, реализация)
				+ Дедуктивный анализ (theorem proving)
				+ Проверка моделей (model checking) (всё формализовано – граф, автомат, переходы)
		- **Тестирование** – проверка соответствия требованиям реальной работы системы на заранее выбранном наборе ситуаций
* Исправление

Тестовые ситуации (test suite)

Тесты = т.с. + проверка требований

T.S. 🡨test interface🡪 SUT

PCO (Points of Control & Observation)

1. Цели, задачи и виды тестирования

Цель – то, чего хотим достичь.

Задачи – то, что нужно сделать.

Цели:

1. Поиск ошибок
2. Оценка качества
3. Контроль развития

Что проверяется:

1. Функциональность
2. Производительность
3. Надёжность
4. Совместимость
5. Переносимость
6. Защищённость
7. Удобство пользования
8. Соответствие (атестационное, сертификационное)
9. Регрессионное (тестирование во времени и антирегрессионность)

Источники информации:

1. Тестирование чёрного ящика
2. Тестирование белого ящика
3. Гипотезы об ошибках
	1. Дымовое тестирование (smoke) (тестирование базовых функций)
	2. Нагрузочное тестирование (load)
	3. Стрессовое тестирование (stress) (выход за границы спецификации, более мощное тестирование)

Уровень абстракции:

1. Unit (модульное) (тестируем кусок)
2. Компонентное (более крупные части)
3. Интеграционное (проверка взаимодействия компонент друг с другом)
4. Системное (тестируем систему целиком (через API, сообщения, GUI)) (смотрится какая-то конкретная характеристика)

Исполнитель:

1. Разработчик
2. Альфа-тестирование (разработчик играет роль пользователя)
3. Бета-тестирование (пользователи)
4. Независимое тестирование (специальная организация)
5. Приёмочное тестирование
6. Процедура проверки требований
7. Критерий полноты / модель ситуаций
	1. Воздействия + данные
	2. Состояния - сводятся к пункту a
	3. Параллелизм
	4. Внешняя среда - сводится к пункту a
8. Построение набора ситуаций
9. Организация тестового набора
10. Организация отчётов
	1. Полнота / покрытие
	2. Ошибки (summary)
	3. Какие требования нарушатся, локализация ошибок, короткий сценарий, детерминизм
11. Выполнение тестов
12. Анализ результатов
13. Организация тестовых наборов

Организация тестовых наборов состоит из следующих наборов:

1. **setup\init** (инициализация)

shared setup - бывает, что инициализация и ввод команд занимают много общего сорта действий, и это нужно делать автоматизировано, в таких случаях это называют shared setup и он занимается отдельной инициализация для test-case (тестового набора)

1. **exersize** (некоторая ситуация, которую нужно воссоздать)
2. **verify** (проверка, что программа повела себя в этой ситуации корректно)
3. **teardown\finallize** (подчищение за тестом, восстановление к состоянию, когда можно делать дальше другие тесты)

shared teardown – teardown для тестового набора

**test-case** – тестовые варианты, набор тестов

можно делать конфигурирование тестовых наборов (т.е. собирать их из нужных тестов для нужной цели)

**xUnit** – модули для тестирования, создаваемые в куче различных языков (в каждом адекватном языке такое должно быть).

Часто тестовый класс – отражает тестовый набор, и он содержит функции – тесты, а также функции, для **shared setup** и **shared teardown**.

Проблемы:

1. иногда тесты очень специфичны, и их setup очень сложный, но можно немного сократить количество тестов, сделав последовательно несколько exersize и verify. Это позволяет делать setup – реже.
2. Иногда, если setup одинаковый для всех тестов, но очень дорого стоит, то его имеет смысл вынести отдельно. Тогда порождается объект тестирования, после чего для него делается setup, а потом все тесты прогоняются, уже без нового заведения объекта и setup – это не очищает состояние объекта после предыдущего теста, что может привести к тому, что один тест поломает другой (пример такого фреймворка - TestNG). С одной стороны, теперь делать тесты нужно аккуратно, а с другой стороны – можно нарваться на какие-то хитрые недокументированные зависимости\возможности\баги.
3. Тестовый набор может стать большим и может сильно меняться в зависимости от той или иной потребности. Если вдруг изменится какое-то из требований, то это может привести к тому, что нужно пересмотреть все тесты и возможно что-то поменять в требованиях. Именно поэтому нужно, чтобы у всех тестов были связи с требованиями, на какие они нацелены и на каких они основаны.
4. Иногда функциональность сложна и хочется, чтобы test2 выполнился после test1, т.к. там есть какая-то связь – это противоречит исходной схеме, однако некоторые фреймворки, например, TestNG, позволяет указать зависимости (например, в виде атрибутов).
5. Если есть несколько тестов с общими кусками кода, то хочется создавать модули, делая тесты – модульными. (это приводит к тому, что всё разваливается на запчасти, у которых есть зависимости и связи с требованиями, и зависимости между ними самими)

Это хоть и тяжелее понимать, но удобнее сопровождать, внося различные изменения.

Основные типы модулей:

* 1. **генераторы данных**
		1. простая генерация (например, range (10, 100))
		2. коллекция данных вида пул (созданная заранее коллекция объектов)
		3. фильтрация (иногда тестовые данные должны иметь особенные свойства, но сгенерировать их сложно, для этого можно сгенерировать много объектов, а потом пройтись по сгенерированному набору и отфильтровать те объекты, которые не удовлетворяют свойствам) (годится, когда свойство легко проверить, но трудно создать)
		4. композиция
	2. **адаптеры** – нужны, если изменился интерфейс, но не изменились требования к работе. Обычно адаптер эмулирует старый интерфейс через новый в виде некоторого наследуемого класса. После этого старые тесты менять не нужно, и они останутся жить, хоть и работать будут через адаптер. (частенько такое случается в телекоммуникационных протоколах)
	3. **заглушки (tub / test double)** – допустим наш тест тестирует некоторый модуль, а этот модуль зависит от некоторой компоненты, которой почему-то ещё нету. Для этого там создаётся специальная заглушка, которая связана с прогоняемым тестом (ведь может быть зависимость по данным). Поэтому, когда тест запускается, с ним в паре идёт корректная заглушка.

Другое использование заглушки – это использование её в качестве прослойки между тестируемой компонентой и ещё чем-нибудь, чтобы можно было наблюдать за тем, как они это пользуют.

Типы заглушек:

* + 1. **fake** - возвращает
		2. **dummmy** – бывает, что заглушка совсем пустая, и ничего не делает, но всё что происходит – возвращается какой-то не null объект – главное, чтобы подходил по типу, но не важно, что там внутри.
		3. **Stub** – связана с тестом и служит для мониторинга и корректных ответов тестируемому модулю.
			1. **spy** – проверяется, что вызывается тестируемым модулем и с какими параметрами, и вернуть ему нужные данные, а потом в конце теста можно глянуть, на протокол происходившего
			2. **mock** – также вызывается, тоже наблюдает и подставляет нужные возвращаемые параметры, но в случае ошибки – сразу дропает тест. (чтобы проверки происходили прямо в mock заглушке – есть специальные специфические фреймворки (jMock, EasyMock, Mockito))
	1. **зонд (probe)** – иногда реализация может содержать какую-то часть, а может и не содержать (например, некоторая опциональная часть стандарта (но если она присутствует, то она обязана работать по спецификации)). Зонд – пробегается по опциям и решает, какие опции есть в данной реализации, а каких нету, и уже в зависимости от этого будут запускаться не все тесты.
	2. **тестовый оракул** – проверяет результат, вычисленный тестируемым модулем

типы оракул:

* + 1. конкретно проверяют выходные данные для чётких входных данных
		2. другая реализация проверяемого модуля, позволяющая проверить, что новый модуль – корректен (например, выдал такой же ответ, или наоборот – исправил известную ошибку старого модуля)
		3. обратные функции (иногда обратная функция к прямой функции – легко и просто может вычисляться)
		4. проверка некоторых свойств выходных объектов.

Для написания оракул тоже есть фреймворки, но пользуются реже и поэтому мало развиты (для заглушек, например, развитость гораздо большая). Часто используются **BDD – Behaviour Driven Development**.

Если понадобилось выделить оракул в отдельный модуль, то к нему переходят связи по проверке требований, но добавляются зависимости его от теста. Но в таком случае часто у оракула появляются проблемы по проверке корректности результата относительно входных данных, которые он не контролировал (например, вычисление корня квадратного – операция не точная для не целых чисел, так же нельзя просто проверить погрешность, т.к. числа могут быть очень большими => нужно проверять относительную погрешность, но появилось деление => для входного параметра = 0, нужно отдельно проводить проверку)

* 1. Машинная арифметика (лирическое отступление)

*Для некоторых функций математики есть большие проблемы по их вычислению на компьютере, например, для tan(atan(x)) – дело в том, что пи\2 – представляется в виде ближайшего double числа, а для очень больших чисел, напр., x=10^100 – там как раз что-то близкое к пи/2, но оно округляется к ближайшему double, а после взятия tan – результат очень сильно отличается от x.*

*Double числа – состоят из бита знача + значащие биты + мантиса = (-1)s 2-(2^(k-1) -1) \* M/(2n-k-1)*

*Ноль бывает положительным и отрицательным, при нулевой мантисе значение будет значить +- бесконечность.*

*Для float n=32, k=8, m=23, для double n=64, k=11, m=52 (всё зависит от формата числа и архитектуры), extdouble n=80, k=15, m=64 (но у intel в реализации n=79, k=14, m=64, а один бит между мантисой и значащими битами задействован на различие между нормализованным и не нормализованным битом), quaddouble n=128, k=15, m=112.*

*При максимальном значении мантисы и любых значащих битах получается число, которому приписывается NaN (например, логарифм отрицательного числа, или деление на ноль, …).*

*Вычисления на вещественными числами всегда лишь приблизительны, есть несколько режимов округления: режим округления к ближайшему, к нулю и к бесконечности. (очевидно, что теперь задача проверки точности квадратного корня становится совсем сложной)*

1. Модели поведения систем

Модель поведения (модель требований) – т.е. не модель реального поведения, а модель должного поведения.

Модель ситуаций – (служит для выбора ситуаций для тестов).

**Модель поведения** – должна описывать, как система должна вести себя в целом.

1. **Логико-алгебраический вид** – ориентированы на внешне-наблюдаемые формы
	1. **Логические модели** – в основном важно принимает высказывание true или false, могут использоваться исчисления предикатов (предикаты первого порядка (т.е. только по переменным, непосредственно взятым из программы, а могут быть и предикаты второго порядка (там предикаты могут применяться к функциям от переменных)).
		1. **Лямбда-функции**. могут использоваться выражения с операторами первого и старшего порядка (например, лямбда от переменной, либо лямбда от функций). Могут использоваться типы (типизированные функции, абстракции по типам, лямбда операторы, кванторы, …)

Пример формулы из лямбда исчислений (формула более высокого порядка) – числа Чёрча (описывают все функции, которые могут быть определены на натуральных числах) = «(для любого) a : (a->a)->(a->a)».

* + 1. **Модальные логики** – выражают то, что должно быть верно. Расширители: Квадратик – должно быть, Ромбик – возможно. Расширители могут быть параметризованными, например, «в результате таких-то действий, должно быть выполнено то-то»

В реальной жизни – модальные логики очень

Частные случаи модальных логик:

* + - 1. темпоральная логика (LTL, CTL, …)
			2. логики с мю-оператором
			3. оператор неподвижной точки (описывает формулу таких состояние, где что-то там (??) выполнено при какой-то там замене функций (??))
			4. Динамические логики – тут может быть программа, а после неё условие.
			5. тройки Хорра.
	1. **Алгебраические модели** –

Примеры:

* + 1. формат хранения знаковых целых чисел в компьютере + операции над ними
		2. формат хранения вещественных чисел + модель содержит в себе такие значения, как INVALID (если где-то по центру был NaN), DIVIZION\_BY\_ZERO (бесконечный результат на конечных операндах, например, деление на ноль, или логарифм нуля), OVERFLOW (если вылетели за возможные границы представления числа в большую сторону), UNDERFLOW, INEXACT (если пришлось округлять результат, т.к. не получается представить точный математический результат в машинном представлении) + процедура проведения операций с значениями (как с числовыми, так и специальными выделенными значениями)

Алгебраические модели бывают:

* + 1. через **реляционную алгебру** (например, набор списков) (выражается через некоторые отношения)
		2. **Аксиоматическое описание абстрактных типов данных** – описание через множество аксиом, которым модель удовлетворяет.

Пример: есть список

 add (список, добавляемое число) -> новый список

 remove (список, индекс удаляемого объекта) -> новый список

 get (список, индекс) -> элемент

 empty() -> новый список

пример аксиом:

 [i=j] get(add(l, i, 0), j) = 0

 [i<j] get(add(l, i, 0), j) = get(l, j-1)

 [i>j] get(add(l, i, 0), j) = get(l, j)

 add(add(empty, 0,o1), 1, o2)

* + 1. ProcessAlg – есть некоторые значки (события), к которым можно применять элементарные операции (последовательная, параллельная композиция, …) – это позволяет задавать циклические процессы, подобно неподвижным точкам.
1. **Исполнимый вид (property-based модели)** – ориентированные на внутренне-выполняемые свойства.

Пытаются описывать, как система ведёт себя во времени (похоже на программу, но, чтобы отличаться от программ, то делается обычно на основе примитивного языка, который может быть выполнено примитивными виртуальными машинами)

* 1. **Операционные модели (state-based models)** – некоторого рода автоматы.

Конечные автоматы (входной алфавит, выходной алфавит, алфавит состояний, отображения из входного алфавита x алфавита состояний -> выходной алфавит x алфавит состояний). Можно нарисовать бесчисленное множество автоматов, которые делают одинаковые действия, но внутри устроены по-разному.

Автоматы могут расширяться различными способами:

1. **LTS** – система конечных переходов – здесь переходы автомата могут быть помечены только одним входным алфавитом, или состоянием, … в итоге автомат может цепочки преобразовывать к другим цепочкам других размеров
2. **EFSM** – у данного автомата, в каждом состоянии могут быть различные внутренние переменные, и переходы в автомате могут быть параметризованы этими переменными.

Переход может иметь охранное условие, зависящее от внутренних переменных. Самы переменные могут меняться при переходе через некоторый переход.

1. **CFSM - взаимодействующие автоматы** – есть набор автоматов, и между ними есть способ взаимодействия (выход одного автомата подаётся некоторому другому автомату) (могут добавляться буфера, которые накапливают приходящий вход, этот буфер может выдавать данные потом пачками, или целенаправленно задерживать данные на какой-то промежуток времени, …)
2. **HFSM - иерархические автоматы –**

Некоторые автоматы организуются в группу и их общий переход идёт некоторому другому автомату, но если кто-то подаёт данные в группу, то либо указывает, кому именно нужно передать данные, либо без указания, и тогда данные попадают тому в группе, кто выбран там для этой цели по умолчанию.

Из каких-то автоматов выход может подаваться на несколько автоматов сразу. Предполагается, что эти автоматы работают после этого одновременно, и по времени каждый следующий переход каждого их этих автоматов происходит «одновременно» (т.е. по сути отслеживается внутренне выполнение автоматов по времени)

1. **Расширенные взаимодействующие автоматы с иерархией** (в UML называется statechart) – помесь всех возможных автоматов.
2. **Временные автоматы –** специфические автоматы с таймерами. Таймер после этого считает тики, и его значение можно использовать в условиях автомата, можно сбрасывать таймер на переходах (подобные автоматы используются для real-time систем).
	1. **Сети Петри** – граф, в котором состояния бывают 2-х типов, и один из видов может иметь несколько входов, и из обоих входов данные переправляют по какому-то из переходов-выходов куда-то дальше.
3. Бывают помеси логико-алгебраических и исполнимых видов.
	1. **ASM – формализм машинно-абстрактных состояний**

Состояниями системы являются множество универсальных алгебр. Между состояниями происходят переходы и на некоторых переходах операции в множестве меняются.

* 1. **Программный контракт** – описываются пред- и пост- условия, помимо этого могут быть переменные связанные с состояниями.

На практике далеко не все модели поведения используются, часто пользуются: программные контракты, автоматы FSM, ELTS.

Для тестирования модель даёт ситуации для тестирования + по моделям видны хитрые ситуации, которые следует обработать тестами (например, максимальные и минимальные числа при работе с целыми числами).

1. Модели ситуаций. Критерий полноты тестирования

Модель поведения – описывает как система должна себя вести во всех случаях.

Модель ситуаций – описывает, как система должна поступать в той или иной ситуации.

При модели ситуаций – выделяется некоторое разбиение на ситуации (наборы классов эквивалентности) (иногда приходится делать грубое разбиение, чтобы уложиться в ресурсы (деньги на проект)), которые нужно проверить тесты. (Внутри каждой ячейки разбиения ситуация в принципе тоже бывает нескольких видов, и теоретически можно разбивать ситуации и дальше, но ... деньги)

Ситуации, находящиеся вдоль границ разбиения на классы эквивалентности могут рассматриваться с дополнительной тщательностью.

Правильного разбиения – нету, однако есть некоторые «правила»/шаблоны, которых придерживаются при разбиении, чтобы оно было более «правильным».

Может рассматриваться «набор видов ситуаций» (в отличие от классов эквивалентности – эти могут пересекаться).

Для каждого типа ситуаций включается монитор, который отслеживает свою ситуацию, и при прогонке тестов, они присматривают за своей ситуацией, и потом мы анализируем их лог, а не лог тестов. И можно не смотреть на то, как они там друг с другом.

***Источники критериев:***

Способы задания:

1. Графы
2. Логические выражения
3. Грамматики

Источники информации:

1. Структура данных (это могут быть входные данные, результаты, или внутренние структуры данных)
2. Структура (кода) системы
3. Структура требования (или модель поведения)
4. Дополнительные гипотезы об ошибках

Покрытия:

1. Покрытие вершин
2. Покрытие рёбер
3. Покрытие пар рёбер (k-путей)
4. Покрытие простых путей (max)

Пути покрывать можно, пытаясь делать самые разные пути в графе поведения системы.

Покрытие ветвлений – подразумевает, что у каждого if, есть опция true и false (причём если else отсутствует, то вполне можно покрыть весь код, пройдя по ветке true, но при этом не покрыть «путь» в программе без захода в true.).

*Пример*:

year = ORIGINALYEAR;

while (days > 365){

 if (isLoopYear(year)){

 if (days > 366){

 days -=366;

 year++;

 }

 } else {

 days -= 365;

 year++;

 }

}

в высокостный год цикл зациклился и устройства подвисли

Другой способ покрытия:

**Использование потоков данных**: - когда определяем на графе новую разметку (обычно за счёт переменных в коде)

all\_uses (пройти через все места графа)

all\_dupair (пройти всеми путями в каждой возможной точке (во всех ветвлениях))

all\_du-paths (пройти всеми путями в графе (при этом при проходе какой-то точки имеет значение – как мы прошли до этого в графе (становится важна история)))

*Пример*:

if (x>0){

 return 0;

}

else{

 return 1;

}

Для покрытия всех использований (по переменной) – тут достаточно пройти только по ветке true.

При проходе через граф начинают появляться guardians – спецификация при каких условиях можно зайти в тот или иной путь.

Критерии полноты, основанные на гипотезах об ошибках: - определяют такие вещи как:

1. А что если по циклу пройти слишком много раз
2. А что если переполнить какие-то данные

Метод «Мутантов». – берётся код и вставляются туда мутации, (например, если забыть всю ветку else, и тому подобное). При мутациях требуется сохранение компилируемости и более-менее разумного поведения программы.

Каждый раз предполагается только одна мутация программы, но каждый раз в разных местах.

Среди мутантов смотрится, нету ли эквивалентных основной программе мутантов (например, такое легко случится, если в конструкторе как-то определяются параметры, а потом они сразу переопределяются). Таким образом, эквивалентных мутантов – нужно выкинуть. А оставшихся мутантов нужно проверить, различает ли их тестовый набор. Так вот тестовый набор имеет хорошее покрытие, если все мутанты различимы.

Мутанты частично используют грамматики.

Примеры случаев с грамматиками:

a::=(a)?(b|c)

b::=c(d)\*

rules(...|d|ad|...)

alternatives

Пример тестирования основанный на структуре данных double: в нём есть знак, мантиса и значащие биты – можно пытаться переполнить отдельно кого-то, …

*Пример*:

(a&&b)||c

(!a&c)||!b

a,b,c - ?

Покрытие предикатов(predicates)/покрытие решений(decision coverage)

Покрытие условий (condition coverage)

CDC - Condition-desicon coverage (нужно сделать и для каждого условия и для каждого предиката, чтобы он был и true и false)

MCC - Multiple-condition coverage (выписываются всем комбинации условий, и для каждой из комбинации должен быть построен тест)

Проблемы начинаются, когда условия не простые и зависят друг от друга и multiple-condition тяжело сделать.

MCDC – Modified condition-decision coverage – для каждого предиката и для каждого условия, что если условие действительно влияет на значение предиката (так чтобы всё не поменялось, а данный предикат изменился), то такое нужно проверить тестом.

К проверке важно знать всё эти 3 штуки:

CDC – 2n

MCDC – 2n

MCC – 2n

1. Методы построения тестов

Методы построения тестов:

1. Вероятностные (пытаются нагенерировать случайные данные)

Минусы: Чтобы аккуратно оценить полноту и качество – нужны другие методы.

Плюсы: Очень легко применяются

Чтобы нагенерировать вероятностные тесты, нужно распределение ситуаций.

Можно попытаться предположить, где самые большие риски напортачить (например, пользователь часто задаёт данные около нуля), тем нужно больше нагенерировать тестов в точку, где повышены риски. Однако статистику рисков собрать – не просто (например, если пользователи сбиваются в какие-то странные группы).

1. Комбинаторные (пытаются нагенерировать тесты, для получения каких-то комбинаций определённого вида)

Существенно проще автоматизировать, чем нацеленные, но не до конца дают хорошее покрытие, хотя и лучше, чем вероятностные.

Методы:

* 1. Построение комбинаций факторов
	2. Автоматные методы (берётся модель того, как система должна работать, и комбинаторное построение теста пытается обойти эту модель)
	3. Алгебраические
1. Нацеленные (строим тесты специально для попадания в ту или иную ситуацию)

Минусы: Применять очень трудно (почти не автоматизируется, нужно думать головой)

Плюсы: Дают ровно ту полноту, на которую рассчитываешь

Нужно много знать о системе, чтобы аккуратно строить тесты. Для построения теста, нужно на каждом шаге думать, что делать, чтобы попасть в нужную тестовую ситуацию.

Основные техники:

* 1. Критерий полноты (нужно определить, откуда берётся список ситуаций, в которые нужно попасть для проверки)

Часто критерии полноты нужно дополнять теми ситуациями, где повышены риски, например, границы экстремумов.

Сценарии:

* + 1. Простые, основанные на usecase's
		2. Длинные сценарии использования (например, если пользователь поменял пароль, потом пожил и снова зашёл под старым паролём – сможет ли он зайти в систему?)

**Domain testing** – доменное тестирование. Означает, что в качестве критерия полноты подразумевается некоторая структура данных. В общем же смысле это означает, что понимается некоторое построение тестов на основе числовых данных (например, если цвета кодировать числами, то близкие цвета будут закодированы похожими числами, и тогда кодирование данных системы - разумно).

Выделяется пространство параметров, на котором в разных областях может быть по-разному всё работает. Лучше всего, если границы этих областей – линейны от заданных числовых параметров. (в случае многомерного пространства – у границ могут быть свои границы). Дальше смотрится включена ли граница в область, если да – то берём тест на границе, а если нет, то нужно взять тест, как можно ближе к границе. Есть некоторая теорема, что если так брать тесты, то все ошибки, связанные с границами (индексами) обнаружатся.

Фильтрация (чем больше параметров, и типов данных – тем сложнее их фильтровать)

Оптимизация (напр., эволюционный алгоритм)

SMT solver

(SAT – satisfiability (подразумевает набор boolean переменных, нужно найти решения булевой формулы))

DART - Directed automated random testing (или concole testing) – суть в том, чтобы ???

1. Комбинаторные методы построения тестов

**Classification tree** - метод на основе дерева классификаций.

*Пример:*

Для объекта автомобиль:

Погода (некоторый фактор), набор значений, которые будут участвовать в этом факторе:

1. Сухо
2. Дождь
3. Оттепель
4. гололёд

Покрытие

1. асфальт
2. Гравий
3. песок

Шины

1. старые
2. новые

Педаль тормоза

1. нажата – приводит к важности другого фактора:

тормозная жидкость

* 1. много
	2. мало
1. отпущена

газ

1. нажат
2. отпущен

Тестовое покрытие строится за счёт построения возможных комбинаций значений факторов.

При этом некоторые факторы могут быть несовместимы (например, нельзя нажимать одновременно тормоз и газ)

Комбинаций становится много, но не все нужны, поэтому используют:

**Covering arrays** (покрывающие наборы) – позволяет, например, выбирать – все возможные пары значений конкретных комбинаций, … Но не все возможные комбинации.

*Пример*: печать веб-страницы

Размер (1,2,5 страниц) – рисунок (есть/нет) – OC (XP, Vista, Android, ubuntu) – Browser (IE, Mozilla, Chrome) – Принтер (Xerox, Cannon, HP) – размер бумаги (A4, A5)

Всего комбинаций = 3\*2\*4\*3\*3\*2 = 432, - рост комбинаций очень много.

Covering arrays – позволяет выбрать лишь те наборы, в которых встречается каждая двойка значений, а вот для любой тройки может и не существовать теста.

Тут количество

CA (t, n1, n2, …, nk) – множество покрывающи наборов глубины или силы t, с параметрами имеющими n1, n2, … nk значений. Это приводит к матрице (столбцы – это параметры значений, стр – существующие тесты), на пересечениях – значение параметра, который используется в данном тестовом наборе. И в этой матрице для каждой комбинации i1, … it и каждой v1, v2, … vt, есть некоторая строка, что в стб n\_i1, … n\_ik – стоят соответственно значения v1, v2, … vt.

Однородный набор – такой набор, что n1 = n2 = … = nk = n

CA (t, k, n)

Количество покрывающих набора тогда будет = C\_{m\*n}^k

*Пара частных случаев*:

t = 2

CA (2, k, 2)

Нужно найти N чтобы k <= C\_{N-1}^{ceil(N/2)}

N = 10 k = 126

N = 20 k = 92378

k = 15 N = 7

CA(2, m, p^k=n) = A (матрица, N строк) =>

CA(2, n\*m +1, p^k = n)

N <= (1 + \eps)\* t \* n^t \* log K

Получается, что при большом количестве параметров K, использовать покрывающие наборы – выгодно.

Построить минимальный покрывающий набор – задача NP-полная.

Поэтому используют рекурсивные техники, позволяющие построить достаточно компактный набор (растёт по логарифму), хоть он и не минимальный.

<http://www.pairwise.org/>

на этом сайте много ссылок на инструменты, которые умеют строить наборы автоматически.

Использование **последовательностей деБроина**

Там предположение, что есть система, на поведение которой влияет k последних действий.

(например, наборные замки, которые открываются после 4-х правильно нажатых кнопок)

Есть алфавит действий n

Хочется построить наиболее компактный проверяющий тест.

Последовательность строится так, что она содержит в себе все возможные последовательности символов длинны k.

Например, aabba – для k = 2 и n = 2. Длинна n^k + (k-1)

Для любых n и k можно пострить последовательность длинные n^k + (k-1) – и это самый компактный способ, т.е. никакая не повторяется дважды, и все последовательности присутствуют.

Пример для n = 3, k = 2: aabbaccbca

Как строить последовательность деБройна систематически:

Для этого нужно построить граф деБройна – в нём вершины - это всевозможные последовательности длинны k-1. А дуги устроены так, что метка на ней – это конкатенация меток инцидентных вершин, но с учётом наложения конца первой и начала второй, метка дуги по длинне = n.

Граф – всегда Эйлеров, а значит в нём есть Эйлеров цикл, он то нам и нужен!

В результате метки рёбер Эйлерова цикла порождает последовательность деБройна.

1. Автоматные методы построения тестов

Автомат - (S, s0, I, R, T)

T модмножество SxIxRxS

<Тут была лекция про автоматы с множеством формул>

**Полностью определённые автоматы** – автоматы, в котором в любом состоянии можно подать любой входной символ.

**Детерминированный автомат** – когда для любого входа и состояния – существует не более одного единственного перехода.

**Наблюдаемый автомат** – если для любого входа, состояния и любой реакции существует не более 1 перехода по состоянию для 1 реакции. (такой автомат не обязан быть детерминированным (т.к. получив реакцию, всегда можно понять, в каком состоянии находишься))

Для любого автомата, можно найти наблюдаемый автомат, который эквивалентен по поведению.

Для детерминированных автоматов есть понятие **минимального автомата** – среди всех эквивалентных автоматов, существует автомат с минимальным числом вершин (такое есть для любого автомата, но для детерминированных автоматов – все минимальные автоматы будут изоморфны с точностью до перенумерации вершин)

При построении тестов, предполагается, что есть некоторый конечный автомат, который определяет, как система себя ведёт (например, взято из спецификации) – этот автомат как правило нам полностью известен.

Есть автомат, соответствующий реализации – этот автомат нам особо не известен.

При построении тестов, нужно их взять такими, чтобы если все тесты давали одинаковые результаты для этих 2-х автоматов, то можно было утверждать, что для любой другой последовательности автоматы также будут вести себя одинаково. В общем случае такого добиться конечно нельзя (т.е. автомат реализации может быть очень сложным и всегда может случиться так, что какую-то особенную последовательность он будет обрабатывать иначе)

У реалистичных систем получаются обычно очень сложные автоматы (даже для простых вещей, например, турникета в метро)

Автомат спецификации строится минимальным, детерминированным, полным, определённым.

Предполагается, что автомат связный (чтобы можно было свободно ходить по нему тестами)

На этих автоматах должны быть следующие действия: reset, перейди в такое-то состояние.

Тестовые последовательности:

Статическая разрешающая последовательность – обычная фиксированная последовательность.

Адаптивная последовательность (начинается применяться последовательность кодов, но в зависимости от реакции выбирается свой способ продолжения последовательности)

Может быть пример автомата, такого, что адаптивная различающая последовательность есть, а статической разрешающей последовательности – нету.

Адаптивная последовательность не длиннее, чем квадрат числа состояний, и строится она за O от квадрата состояний (и можно будет сказать, что она существует, либо что она не существует).

1. Автоматные методы построения тестов.

J – детерминированный, вполне определённый, сильно связный. <= N

S – минимальный = n

Ij вложено в Is

Rj эквивалентно Rs

RCIW:

Reset

C – последовательность ???

I – произвольный входной символ

W- произвольная последовательность из ???

RCIW (произносится «double-U метод») тестирование основано на том, чтобы покрыть все переходы.

RCI^{N-n+1}W

RCI^{N-n+1}{d} – d-метод

RCI^{N-n+1}u\_s – не подходит в качестве теста, а вот для всех u: RCI^{N-n+1}{u\_s} - подходит

Обычно сложность по количеству тестов O(p\*n3) (p – мощность Is)

Wp-метод – 2-х шаговый

1. RCW –

С - последовательность покрывающего множества (нужно для того, чтобы проверить, что в спецификации должны быть состояния, которые отличают), а потом всё W

1. Для каждого из оставшихся переходов, применяем соответствующую последовательность из С, чтобы попасть в начало перехода, потом делаем переход, а потом проверяем в каком состоянии мы оказались – для этого нужно использовать идентифицирующее множество Ws.

(объединение всех Ws = W)

1. Проверка всех пар стимулов, последовательности, которых не были проверены на первых 2-х шагах. – Итерационно, пока не будет полностью проверен автомат реализации на основе автомата спецификации.

UIOv-метод

Можно вычеркнуть повторяющиеся под наборы из тестов из полученного при помощи Wp-метода.

Бывают автоматные методы, не использующие операцию Reset (например, в случае если он очень дорогой).

LTS

Переходы по стимулам и по реакции:

Есть некоторый недетерминизм в том, нужно выдать в данный момент некоторую реакцию, или же пройти по стимулу.

Если окружение быстрее, то система будет идти по стимулам, если медленнее, то по реакции. Чтобы на это воздействовать – можно вводить те или иные задержки.

Если модель описывается расширенными автоматами (т.е. есть переменные, от которых что-то зависит).

Обычно всё начинается с развёртки расширенных автоматов.

Редуцированный expansion: допустим есть переход, который ориентируется на то, что мы должны пройти хотя бы несколько раз по петле, тогда давайте мы пройдя через эту петлю несколько раз, потом просто сходим по ней очень много раз и пропустим промежуточные состояния (пару раз до этого проверили, что действия нормально работают)

В общем случае для расширенного автомата построить тесты эквивалентности нельзя, но если переменные – счётчики, то в целом вполне можно редуцировать.

1. Технология UniTESK

Тесты строятся на основе формальных моделей требований.

Метод.

Архитектура тестового набора – позволяет представить через набор модулей, и можно понять, какие модули получатся автоматических из формальных моделей, а какие нужно ещё дорабатывать руками.

Инструменты (+ языки)

1. Определение задач и рамок SUT.
	1. Определение задач и рамок SUT
	2. Определение интерфейса
	3. Определение функций и проверяемых свойств
	4. Определение важности функции, свойства, компонент.
	5. Первичная декомпозиция
	6. Определение техник тестирования
2. Определение требований к SUT.
	1. Определение источников
	2. Извлечение требований
	3. Систематизация требований
	4. Уточнение и согласование требований
	5. Финальная декомпозиция
	6. Формализация требований
3. Определение требований к полноте тестов.
	1. Выделить требования к полноте
	2. Формализация критериев полноты
4. Разработка тестов (+ выполнение и отладка)
	1. Определение способа организации тестов и техники тестирования
	2. Построение генераторов тестовых данных
	3. Построение автоматной модели теста
	4. Построение адаптеров
	5. Отладка тестов
	6. Получение результатов тестов
5. Анализ результатов

Для тестирования асинхронной работы компонент или параллелизма, используется особая техника.