И. Бурдонов, А. Косачев
Тестирование и верификация систем на основе формальных моделей. Лекция 1.
http://old.compsciclub.ru/courses/testingverification/
Уральский федельный университет, Computer Science клуб, Екатеринбург, 6,7,8 апреля 2012
45 слайдов

Тестирование и верификация систем на основе формальных моделей.

1. Формализация тестирования

Слайд 1. Читаю название

Слайд 2. Вот план доклада. Первые пять разделов посвящены теории конформности. Оптимизация тестирования – это уже переход к следующему разделу – практическому тестированию. Практическое тестирование будет рассмотрено позже. Тогда же в конце мы скажем несколько слов о дальнейшем развитии: что уже сделано и что еще предстоит сделать.

Слайд 3. 1. Формализация тестирования.

Введение: Что такое тестирование конформности?

Речь у нас пойдёт, главным образом, о тестировании конформности. По английски, Conformance Testing.

Конформность – это соответствие.

Тестирование соответствия – это разновидность тестирования вообще.

Тестирование – это разновидность верификации.

А верификация в самом общем виде понимается как проверка правильности.

То, правильность чего проверяется, обычно называют исследуемой системой или реализацией.

Чем отличается тестирование от других методов верификации?

Тестирование – это проверка правильности *в эксперименте*.

Этим оно отличается от аналитических методов *доказательства* правильности.

Слайд 4. 1. Формализация тестирования.

Введение: Что такое правильность?

Понятие конформности как раз и возникло в попытке найти наиболее адекватный ответ на этот вопрос.

Идея очень простая: не бывает абсолютно правильной реализации, правильность понятие относительное и определяется сравнением с эталоном – спецификацией.

Более строго: правильность – это соответствие требованиям.

Спецификация – это описание требований.

Требования – должны быть функциональными.

Слайд 5. 1. Формализация тестирования.

Введение: Что такое функциональность?

Функциональная спецификация отвечает на вопрос «ЧТО?», но не отвечает на вопрос «КАК?».

1►В качестве примера можно привести спецификацию функции вычисления квадратного корня. Она может быть записана в двух формах: $y^2=x$ *или* `y=алгоритм_`$\sqrt{(x)}$.

Первую форму обычно называют имплицитной, а вторую – эксплицитной. Наиболее наглядно идею спецификации демонстрирует имплицитная форма. Спецификация утверждает, что реализация правильная, если значение, возвращаемое функцией, будучи возведённым в квадрат, равно аргументу. Здесь нет никакого алгоритма вычисления квадратного корня, хотя именно такой алгоритм должна реализовывать реализация. Иными словами, спецификация говорит: не важно как это делает реализация, но результат должен быть вот таким.

2►Теперь понятно, что даже если спецификация эксплицитна, и в ней записан тот же самый алгоритм вычисления квадратного корня, что и в тексте реализации, смысл этих записей разный. Почему? Потому что требованиям спецификации удовлетворяет не только та

реализация, в которой используется такой же алгоритм, но и та, которая использует другой алгоритм, но возвращает такое же значение квадратного корня.

В этом месте *функциональность* – это слово, производное от слова *функция*. А в математике функция – это не то же самое, что алгоритм вычисления функции. Спецификация описывает саму функцию, неважно каким способом, а реализация реализует алгоритм вычисления функции.

3▶Говоря о том, что функциональность – это *что*, а не *как*, следует учитывать, что «лёд здесь тонок». Различие между *что* и *как* не абсолютное, а скорее относительное. Поэтому и функциональность – понятие относительное, оно зависит от того, что для нас существенно (вопрос *что*?), а что – второстепенно (вопрос *как*?).

4▶Характерным примером являются временные характеристики реализации. Обычно они считаются нефункциональными. Нам не важно, сколько времени реализация вычисляет квадратный корень, лишь бы она делала это правильно. Но иногда время оказывается настолько критически важным, что ограничение на него следует рассматривать как функциональное требование. Отвлекаясь в сторону, скажу, что в этом случае используется, например, не обычная автоматная модель реализации, а так называемые временные автоматы, которые умеют описывать и, тем самым, позволяют генерировать тесты, проверяющие время исполнения.

Слайд 6. 1. Формализация тестирования.

Введение: Формальные спецификации.

Итак, спецификация описывает функциональные требования к реализации. Это описание должно быть формальным. Зачем?

1▶Спецификацию пишет человек. Одно из назначений спецификации – служить «техническим заданием» разработчику, который будет создавать реализацию, удовлетворяющую этим требованиям. Если спецификация неформальна, это часто служит источником недопонимания между спецификатором, формулирующим требования, и разработчиком, создающим реализацию. Или между разработчиком системы и её пользователем. И, наконец, между разработчиком и тестировщиком, или между

заказчиком и тестировщиком. Поэтому первое, для чего нужна формальность спецификации, – это однозначность понимания человеком.

2▶Во-вторых, спецификация нужна для того, чтобы на её основе проверять правильность реализации. Конечная цель здесь – автоматическая верификация. Поэтому независимо от того, применяются ли аналитические методы доказательства правильности, или генерируются тесты, спецификация должна быть достаточно формальной, чтобы её мог понимать компьютер. К сожалению, верификация полностью автоматическая только в идеале, а на практике она всего лишь автоматизирована, то есть требует интеллектуальных усилий человека, хотя и не на всём пути, а лишь в некоторых критических точках. Поэтому для верификации также требуется понимание спецификации не только компьютером, но и человеком.

Слайд 7. 1. Формализация тестирования.

Введение: Математическая модель.

Для того чтобы спецификация была формальной, она должна быть сформулирована на формальном языке. В основе любого такого языка лежит математика как универсальный язык формализации. Иными словами, мы должны выбрать математическую модель спецификации и реализации, а также математически описать понятие правильности как соответствия между ними, то есть конформности.

Модель спецификации, или спецификационная модель, считается заданной спецификацией, которая, тем самым, понимается как описание этой модели. Способ такого описания нас пока не интересует.

1▶Что же касается реализации, то для неё используется, так называемая, *тестовая гипотеза* (G.Bernot). Она утверждает, что в любом эксперименте реализация ведёт себя так же, как некоторая модель. Важно подчеркнуть, что тестовая гипотеза утверждает лишь то, что реализационная модель существует, но не утверждает, что она известна. Тестирование применяется как бы к чёрному ящику, в котором скрыта реализация, а тестовая гипотеза утверждает, что с помощью эксперимента невозможно узнать, находится в ящике данная реализация или её реализационная модель.

Конечно, математическая модель – это абстракция, но абстракция полезная, при которой мы отвлекаемся от того, что считаем второстепенным (вопрос *как*?) и сосредоточиваем своё внимание на существенном (вопрос *что*?). Типов моделей может быть много, и выбор нужной модели – дело чрезвычайно важное. Оно определяется не только тем, какие мы формулируем требования, но и тем, какие у нас есть тестовые возможности и реализационные гипотезы.

2▶*Тестовые возможности* – это возможности проводить те или иные тестовые эксперименты. Они определяют, что мы можем наблюдать в эксперименте и как мы можем управлять экспериментом.

3▶*Реализационные гипотезы* – это гипотезы о реализации, предполагающие, что тестируемая реализация – не любая, а относящаяся к некоторому классу возможных реализаций. Такие предположения не проверяются при тестировании, лишь в некоторых случаях они могут контролироваться, да и то частично. Реализационная гипотеза – это предусловие тестирования.

4▶Теперь мы можем сказать, что конформность – это математическое соответствие, то есть подмножество декартового произведения множества реализационных моделей и множества спецификационных моделей. Таких соответствий может быть очень много. Первое, что мы дальше сделаем, – ограничим класс конформностей теми, которые имеют смысл при тестировании как эксперименте над реализацией.

Но сначала я хочу обратить внимание на очевидные проблемы, проистекающие из способа описания модели.

Слайд 8. 1. Формализация тестирования.

ПРОБЛЕМЫ: Введение: Извлечение модели.

Теперь обратим внимание не на сам факт того, что спецификация описывает модель, а на то, как она это делает. Понятное дело, что на практике спецификации пишутся не на языке математики, а на специальных языках спецификации или спецификационных расширениях языков программирования. И хотя в основе спецификации всегда лежит некоторая математическая

модель, а спецификация понимается как описание этой модели, тем не менее, спецификация и модель – не одно и то же.

Может быть, для разработки реализации по такой спецификации о модели можно не вспоминать, но методы проверки правильности (как доказательства, так и тестирования) опираются как раз на математическую модель.

Поэтому у нас возникает проблема: как извлечь модель из спецификации?

Нужно ещё учесть, что одной и той же спецификации может соответствовать несколько моделей разных типов. В известном смысле модель – это способ математической интерпретации спецификации. Вообще говоря, тот, кто пишет спецификацию, обычно имеет в голове модель некоторого типа, хотя часто это бывает неосознанно, а спецификация получается осмысленной просто потому, что хорошо продуман язык спецификации, не позволяющий человеку писать полную ерунду. Иными словами, тип модели заложен в самом языке спецификации. Однако в основе языка не обязательно лежит единственный тип модели. Чем язык универсальнее, тем больше свободы он предоставляет в выборе типа модели, которая подразумевается при создании спецификации.

Как решать проблему извлечения модели из спецификации? Здесь приходится балансировать, если можно так выразиться, между желаниями человека и компьютера (или другого человека). Тот, кто пишет спецификации, хочет иметь универсальный язык, хотя бы потому, что ему лень изучать много разных языков. Ещё он хочет, чтобы спецификация была ему самому понятной. А для автоматического извлечения модели желательно, чтобы тип этой модели как можно более явно отражался в конструкциях языка, делая его, тем самым, менее универсальным. И здесь важна не понятность для человека, а формальное соответствие типу модели. На практике часто приходится делать выбор между понятностью спецификации и явностью отражения в ней модели. И всё равно самый первый этап генерации тестов делается вручную как раз потому, что на этом этапе извлекается модель. Кроме того, эта модель часто ещё и преобразуется для удобства тестирования.

1►Можно отметить, что для методов доказательства, в отличие от тестирования, требуется явное задание не только

спецификационной, но и реализационной модели. А эта проблема на порядок сложнее. Понятно, что если у нас нет исходного кода реализации, то извлекать модель просто не из чего, и проблема становится неразрешимой. Здесь можно применять только тестирование, но не аналитическую верификацию.

Однако даже если такой программный код есть, он совсем не похож на код спецификации. Это и понятно: спецификация специально предназначена для описания функциональных требований, а реализация пишется на языке программирования, и её задача – «разворачивать» эти требования в конкретные алгоритмы, структуры данных и т.п. Иными словами, код реализации гораздо больше «замусорен» второстепенными деталями, чем код спецификации. Пример: функция извлечения квадратного корня. Понятно, что имплицитная спецификация очень короткая, а алгоритм вычисления квадратного корня гораздо сложнее.

Тем не менее, существуют способы автоматического извлечения модели из текста программы. Однако, во-первых, их применимость сталкивается с целым рядом проблем, что вызывает целый ряд ограничений на программы. Проблемы создают конструкции языка типа указателей, преобразования типа – casting, объединения – union и т.п. Другие проблемы связаны с неформализованностью семантики языка, например, порядок вычислений в некоторых выражениях языка Си. Третий род проблем – это выделение интерфейса программы, что необходимо, например, для построения автоматной модели. Но, может быть, самое главное – это взрыв состояний: получающаяся модель слишком велика и сложна для практического использования. Она не показывает смысл программы, что, по идее, и должна делать модель, поскольку оказывается «замусоренной» также, как исходная программа. Поэтому-то извлечение модели из программы чаще всего применяется для model checking, когда требуется не столько сама модель, сколько проверка выполнения некоторых её свойств, для чего иногда строится лишь частичная, сокращённая модель. Но даже в тех случаях, когда приемлемую модель можно извлечь автоматически, нет гарантии, что она правильная, поскольку работа программы зависит также от компилятора, в котором могут быть ошибки, и операционной среды.

Всё это оказывается одной из главных причин, по которой область применения тестирования гораздо шире, чем область

применения аналитической верификации. Правда, у этой медали, как обычно, есть и другая сторона: доказательство, если оно возможно и имеется правильная модель реализации, даёт гарантированно правильный результат за конечное время, в то время как тестирование – это процесс, про который почти никогда нельзя сказать, что оно закончено. Эту проблему мы рассмотрим позже.

Слайд 9. 1. Формализация тестирования.

Введение: Цикл разработки и тестирования.

Прежде, чем двигаться дальше, имеет смысл рассмотреть место тестирования конформности в общем процессе создания программного продукта.

Всё начинается с заказчика, который формулирует требования к системе. Эти *требования неформальные*.

1▶До появления идеи формальных спецификаций, а на практике во многих случаях и сегодня, эти требования непосредственно используются разработчиком для создания *реализации*.

2▶Если мы хотим тестировать реализацию на соответствие спецификации, то нам нужно сначала эти спецификации создать. Источником служит исходный код самой реализации. Нужно этот код изучить и понять, это называется инспекцией кода. Такую инспекцию часто делают безотносительно к составлению формальных спецификаций. Но, когда при изучении у вас есть цель создать формальные спецификации, это сильно дисциплинирует и систематизирует изучение. Многие ошибки в реализации находятся уже на этом этапе. Результаты такого изучения записываются формально как спецификация.

Иногда, правда, есть промежуточный этап, когда создаётся документация, которую можно использовать для спецификации. Эту документацию создаёт разработчик одновременно с написанием кода или потом кодоинспектор одновременно с изучением кода. Однако, во-первых, такая документация не всегда есть, во-вторых, она почти всегда не полна, и, в-третьих, часто недостоверна. Поэтому код остаётся единственным надёжным источником, а документация лишь помогает понять код.

3▶Получив спецификацию, мы можем создавать тесты и тестировать соответствие реализации этой спецификации.

4▶Более современной и более правильной в методологическом смысле является другая последовательность. Сначала тот, кого можно назвать *архитектором*, на основе неформальных требований создаёт формальные спецификации. Этот процесс можно понимать как уточнение требований и их формализацию.

5▶Спецификации служат одновременно источником как для разработчика, так и для *тестировщика*, которые создают, соответственно, реализацию и тесты. Эти процессы становятся независимыми и могут происходить параллельно. За счёт этого можно получить большой выигрыш по времени, правда, с учётом потери времени на спецификацию.

6▶Частично и реализация и тесты могут генерироваться из спецификации автоматически. Для реализации это означает создание прототипа. Тесты также могут генерироваться автоматически, хотя доля ручного труда всё равно остаётся.

7▶Когда реализация и тесты готовы, начинаются тестовые эксперименты.

8▶Тест выносит вердикт: найдена ошибка или нет. После этого происходит *анализ* найденных ошибок. Кроме вердикта, результатом тестирования являются разного рода оценки полноты тестирования, то есть, если ошибок не найдено, то какова вероятность, что они всё же остались, и какого типа могут остаться ошибки.

9▶Цель тестирования – найти ошибки в реализации.

10▶После этого разработчик исправляет ошибки и снова происходит тестирование и анализ его результатов.

11▶Однако, поскольку и при создании спецификаций, и при создании тестов тоже принимает участие человек, ошибки могут быть найдены и в спецификации, и в тесте. Фактически, какое-то время тестирование одновременно является отладкой тестов и спецификаций.

12▶Если ошибка в тесте, тестировщик меняет тест и снова происходит тестирование и анализ его резльтатов.

13▶Понятно, что наиболее неприятна ошибка в спецификации.

14▶Такая ошибка вызывает переделку спецификации и, как следствие, реализации и теста. Поэтому эта часть работы особенно важна и критична. К тому же спецификация делается почти полностью вручную. Попытки автоматизировать какую-то часть работы, конечно, предпринимаются. Но, в общем, работы в этом направлении пока находятся в зачаточном состоянии.

В идеале после какого-то периода отладки основным циклом становится цикл разработки и тестирования, когда все обнаруживаемые ошибки – это ошибки реализации. Здесь важно отметить, что тесты, созданные по формальным спецификациям, годятся для любой реализации, пока не изменяются сами требования и, тем самым, спецификация. Поэтому развитие системы, её модернизация, изменение алгоритмов или структур данных, если эти изменения нефункциональны, а также добавление новых возможностей, перенос на другие аппаратные или программные платформы и т.п. – всё это происходит с одними и теми же тестами, которые, тем самым, повторно используются много раз.

15▶Наконец, когда ошибок не обнаруживается, система считается протестированной и работа закончена. До обнаружения ошибок уже во время эксплуатации – это если тестирование было недостаточно полным. Или до тех пор, пока не потребуется модификация системы, то есть создание новой её версии.

Слайд 10. 1. Формализация тестирования.

Моделирование

Задача верификации – проверка правильности исследуемой системы.

1▶Под «правильностью» понимается соответствие системы заданным требованиям.

2▶Для того, чтобы это отношение формализовать, используются формальные модели. В модельном мире система отображается в реализационную модель, которую мы будем называть реализацией, а требования – в спецификационную модель, которую мы будем называть спецификацией.

3▶Соответствие исследуемой системы требованиям отображается в отношение конформности.

4▶Отношение конформности – это бинарное отношение, то есть подмножество декартового произведения множества всех реализаций и множества всех спецификаций.

5▶Система соответствует требованиям ⇔ реализация конформна спецификации.

6▶Этот вывод основан на предположении, что моделирование требований спецификацией, системы реализацией и соответствия конформностью «правильное». Правильность моделирования – вне рамок нашего рассмотрения. Мы предполагаем, что моделирование «правильное».

7▶Спецификация всегда задана.

Слайд 11. 1. Формализация тестирования.

Моделирование

Существование реализации (как модели реальной системы) предполагается (тестовая гипотеза). Если реализация также задана явно, верификация конформности может быть выполнена аналитически.

1▶Что делать, если устройство реализации неизвестно? То есть она представляет собой «чёрный ящик»? Бывает и так, что реализация в принципе может быть известна, но, как часто бывает при автоматическом извлечении модели из программы, слишком сложна для анализа.

2▶Тогда приходится применять тестирование как проверку конформности в процессе тестовых экспериментов.

3▶Разумеется, в этом случае требования должны быть выражены в терминах взаимодействия системы с окружающим миром.

4▶Тестирование основано на том, что тест подменяет собой окружение и, взаимодействуя с системой, наблюдает её поведение.

5▶Поэтому само отношение конформности и его тестирование основаны на той или иной семантике взаимодействия.

Слайд 12. 1. Формализация тестирования.

Моделирование

Прежде всего, по спецификации генерируются тесты. Тест, взаимодействуя с реализацией, в конце своей работы выдает вердикт: ***pass*** или ***fail***. Определяется отношение «реализация проходит тест». Оно означает, что при любом прогоне этого теста всегда выдаётся вердикт ***pass*.**

1▶Набор тестов полный, если реализация конформна спецификации тогда и только тогда, когда она проходит каждый тест из набора.

2▶Эти тесты находятся в модельном мире. Нужно получить из них реальные тесты, то есть тестовые программы в реальном мире. Это делается с помощью трансляции модельных тестов в реальные тесты. Реальные тесты взаимодействуют уже с реальной исследуемой системой и тоже выдают вердикт ***pass*** или ***fail***. Определяется отношение «система проходит тест». Оно также означает, что при любом прогоне этого теста всегда выдаётся вердикт ***pass*.**

3▶Утверждается, что целевая система проходит реальный тест тогда и только тогда, когда её модель, то есть реализация, проходит модельный тест.

4▶Правильность этого утверждения основаны на том, что генерация и трансляция тестов правильные.

5▶На практике существует целый ряд важных ограничений. Сейчас имеет смысл указать пока только на одно из них: каждый тест должен заканчиваться за конечное время.

Слайд 13.1. Формализация тестирования.

Моделирование

Суммируя, можно сказать, что тестирования конформности основано на трёх базовых положениях.

1▶Во-первых, моделирование выполнено правильно: спецификация – правильная модель требований, реализация – правильная модель исследуемой системы, отношение конформности между реализацией и спецификацией – правильная модель отношения «исследуемая система соответствует требованиям».

2▶Во-вторых, должен существовать полный набор тестов, основанный на соответствии между отношением конформности и отношением «реализация проходит тест». Нужно отметить, что для произвольной конформности полный набор тестов может и не существовать. В конце этих лекций мы будем говорить о симуляции, для которой полный набор тестов не всегда существует. Но пока мы будем иметь дело с конформностью, для которой хотя бы один полный набор тестов существует для каждой спецификации.

3▶В-третьих, моделирование самого взаимодействия правильное. Это значит, что модельное отношение «реализация проходит тест» правильно моделирует реальное взаимодействие исследуемой системы и реального теста. Разумеется, здесь также предполагается, что трансляция модельного теста в реальную тестовую программу происходит правильно.

4▶Если эти предположения верны, то верно и утверждение о том, что исследуемая система соответствует требованиям тогда и только тогда, когда она проходит набор тестов, транслированных из полного набора модельных тестов.

Слайд 14. 1. Формализация тестирования.

Этапы проверки конформности

Для того, чтобы тестировать конформность, мы должны, в первую очередь, выбрать модель взаимодействия. Мы должны понять, какие тестовые воздействия на реализацию мы можем осуществлять и какие возможны наблюдения над поведением реализации.

1▶Во-вторых, нужно выбрать тип модели для спецификации и для реализации.

2▶Далее нужно выбрать отношение конформности, то есть ответить на вопрос: как следует понимать спецификацию?

3▶Потом нужно определить, для каких спецификаций мы собираемся генерировать тесты и какие реализации собираемся тестировать. Такого рода ограничения бывают очень полезны для выполнения практических требований, когда в общем случае акие требования выполнены быть не могут.

4▶Наконец, когда всё готово, создаётся система автоматической генерации тестов по спецификации.

5▶После того, как модельный тест создан, его нужно транслировать в реальный тест. Основная проблема здесь – это согласование интерфейсов в модельном и реальном мире. Эту функцию выполняет часть тестовой системы, которая называется медиаторами. Они преобразуют модельные тестовые воздействия в реальные, чтобы подавать их на исследуемую систему. Также они преобразуют реальные наблюдения над поведением системы в модельные наблюдения.

6▶Наконец, когда реальные тесты готовы, они прогоняются на целевой системе.

7▶На этих лекциях мы будем заниматься, в основном, первыми пятью пунктами, то есть тем, что происходит в модельном мире.

Слайд 15. 1. Формализация тестирования.

Машина тестирования

Теперь нам нужно формализовать само понятие тестового эксперимента, то есть формально описать семантику взаимодействия.

Формализация взаимодействия – это ключевой момент в тестировании конформности. От того, какой вид взаимодействия мы рассматриваем, непосредственно зависит отношение конформности и допустимые классы реализаций и спецификаций.

Семантика взаимодействия формализует имеющийся набор тестовых возможностей по управлению и наблюдению за поведением тестируемой системы. При тестировании мы можем наблюдать только такое поведение реализации, которое, во-первых, «спровоцировано» тестом (управление) и, во-вторых, наблюдаемо во внешнем взаимодействии. Интересующие нас отношения конформности – это те отношения, для проверки которых необходимо и достаточно тестовых возможностей, описываемых данной семантикой взаимодействия.

1▶Эту семантику удобно описывать с помощью, так называемой, машины тестирования. Она представляет собой «чёрный ящик», внутри которого находится реализация. Реализация нам не видна, но ящик снабжён разного рода индикаторами для наблюдения

и кнопками или переключателями для управления. Тем самым, фиксируется *интерфейс* между реализацией и тестом.

Сначала рассмотрим наблюдение.

2▶Для наблюдения за поведением реализации машина снабжается дисплеем.

Работа реализации понимается как последовательность дискретных действий, которые она совершает. Некоторые из этих действий – это внешние или наблюдаемые действия. Иными словами, мы должны уметь разбить внешнее, наблюдаемое поведение машины на отдельные действия.

3▶Считается, что задан алфавит внешних действий **L**. Важно отметить, что внешний, наблюдаемый символ a – это абстракция: в машине может быть много различных действий, которые мы видим как a; для нас эти действия неразличимы между собой, но отличимы от тех действий, которые мы видим как символ b. Выбор подходящей абстракции и, тем самым, определение алфавита действий – это важная часть определения семантики взаимодействия.

4▶Когда машина совершает действие a из алфавита **L**, мы можем это действие наблюдать на экране дисплея.

5▶Кроме внешнего, наблюдаемого, поведения, машина может иметь *внутреннюю активность*. Это та часть поведения машины, которая не наблюдаема. Поэтому в описании семантики взаимодействия её нет смысла разбивать на отдельные и, тем более, различные действия. Внутреннюю активность принято обозначать одним символом τ.

6▶Тем не менее, иногда мы можем наблюдать сам факт того, что машина имеет какую-то активность: внешнюю или внутреннюю. Мы видим, что реализация что-то делает, а что именно неизвестно. В таком случае машина снабжается зелёной лампочкой, которая горит, пока реализация работает, и гаснет, когда реализация останавливается.

Следует отметить, что зелёная лампочка – это дополнительная тестовая возможность. Её может и не быть. Например, если тест и реализация работают в одном компьютере, то часто можно отслеживать, занимает ли реализация процессорное время или нет. Однако если реализация удалена и находится на другом конце канала

связи, то, как правило, приходится считать, что зелёной лампочки у нас нет.

7▶Если тестирование ограничивается только наблюдением, то его называют мониторингом или пассивным тестированием. Оно тоже полезно, но мы будем рассматривать общий случай тестирования, которое позволяет как-то управлять поведением реализации.

Слайд 16. 1. Формализация тестирования.

Машина тестирования

Управление сводится к тому, что оператор машины, выполняя тест (понимаемый как инструкция оператору), осуществляет тестовое воздействие, нажимая кнопки на клавиатуре машины, тем самым «разрешая» реализации выполнять те или иные действия, которые могут им наблюдаться.

1▶Первую машину тестирования придумал Милнер, она называется реактивной машиной. Потом Ван Глаббек придумал генеративную машину тестирования. Чем они отличаются?

2▶Машина Милнера работает «по приказу»: для каждого внешнего действия в ней имеется кнопка, нажатие которой разрешает выполнять только это действие. До выполнения внешнего действия реализация может иметь внутреннюю активность, которая не запрещается нажатием кнопки. После выполнения внешнего действия машина *приостанавливается*, в том числе приостанавливается внутренняя активность, до нажатия следующей кнопки.

3▶В машине Ван Глаббека каждому внешнему действию соответствует переключатель, который имеет два положения: *on* (разрешение действия) и *off* (запрещение действия). Можно устанавливать любые переключатели в любые положения. Машина работает, выполняя разрешённые внешние действия до тех пор, пока такие действия у неё есть. Между действиями может быть внутренняя активность, которая не запрещается никакими положениями переключателей.

Если переключатели разрешают реализации выполнять несколько действий, то выбор выполняемого действия происходит, вообще говоря, недетерминированным образом.

4▶На самом деле эти машины оказываются эквивалентными, если сделать в них некоторые модификации. В реактивной машине нужно разрешить нажимать сразу несколько кнопок, предоставляя машине самой выбирать, какое из разрешённых действий ей выполнять. А в генеративной машине нужно ввести переключатель, приостанавливающий машину, то есть блокирующий внутреннюю активность и выполнение внешних действий. Поэтому в дальнейшем я буду ссылаться, как правило, на машину Ван Глаббека.

5▶При тестировании могут быть дополнительные тестовые возможности, которые в машине задаются с помощью, так называемых, *лампочек меню*, соответствующих внешним действиям. Лампочка для действия горит, когда это действие определено в реализации. Если машина активна, лампочки могут все время мигать и трудно сказать, какие действия определены в текущий момент времени. Поэтому обычно считается, что состояние лампочек достоверно только тогда, когда машина стоит, то есть у неё нет ни внутренней, ни внешней активности. Лампочки меню позволяют в момент остановки машины определить *множество готовности* (*ready set*) – множество всех внешних действий, определённых в текущем состоянии, то есть готовых к выполнению. Последовательности наблюдений, включающие как действия, так и множества готовности, называются трассами готовности – ready traces.

Обычно такая тестовая возможность отсутствует, но бывают и исключения.

6▶Например, в графических интерфейсах такое меню определённых действий может появляться на экране, иногда в виде списка кнопок для всех (или части) действий. При этом действиям, которые сейчас не могут выполняться, или отсутствуют в меню или им соответствуют «бледные кнопки».

В дальнейшем мы будем рассматривать семантики взаимодействия без лампочек меню. Хотя некоторые вещи переносятся на семантики с лампочками готовности без особого труда: определение конформности и генерация тестов.

Слайд 17.1. Формализация тестирования.

Машина тестирования

Машина тестирования Ван Глаббека имеет один недостаток. Дело в том, что в её устройстве никак не регламентировано какое множество переключателей можно установить в положение “on”, а какое нельзя. Формально любые переключатели я могу устанавливать в любые положения. На практике, однако, это далеко не всегда так. Поэтому приходится как-то «сбоку» от самой машины указывать соответствующие ограничения. Это неудобно, поскольку машина тестирования как раз и предназначена для полного описания семантики взаимодействия. Эти ограничения должны быть заложены в самом её устройстве. Примеры я потом покажу.

А сейчас я расскажу о машине тестирования, которую мы придумали. Она лишена этого недостатка. Для этого машина параметризуется семейством **P** подмножеств алфавита внешних действий. Это семейство должно покрывать весь алфавит.

Каждому множеству из семейства **P** соответствует своя кнопка.

Одновременно можно нажимать только одну кнопку.

Но, поскольку на кнопке написано не одно действие, а множество действий, нажатие такой кнопки разрешает реализации выполнять любое действие из этого множества.

Действие, которое будет выполняться реализацией, выбирается, вообще говоря, недетерминированным образом.

Внутренняя активность считается всегда разрешённой, как в машине Ван Глаббека.

Зелёной лампочки и лампочек действий у нас нет.

1▶Вот мы нажимаем кнопку A больше и получаем на экране наблюдение a маленькое из множества A большое. Кнопка автоматически отжимается. Дальше мы можем снова нажать ту же самую или другую кнопку.

2▶Например, мы нажимаем кнопку B большое и наблюдаем на экране действие b маленькое из множества B большое.

Слайд 18. 1. Формализация тестирования.

Машина тестирования. Пример

А теперь примеры. Первый пример – это функция вычисления квадратного корня.

Действие заключается в том, что мы передаём функции аргумент x≥0 и получаем от неё вычисленное ею значение y. Такое действие задаётся парой чисел (x,y), где x≥0. Понятно, почему нужно указывать x – это то число, квадратный корень из которого мы хотим вычислить. Почему нужно указывать y? Потому что разные результаты для одного и того же аргумента – это для нас разные действия. Мы хотим их различать хотя бы для того, чтобы сказать, какой результат правильный, а какой нет.

Однако для каждого аргумента x в машине только одна кнопка “x”, которая разрешает выполнять все действия (x,y), где x фиксировано для этой кнопки. Иными словами, кнопка “x” означает, что мы передаём функции аргумент x и готовы принять от неё любой результат y.

1►Вот, например, мы нажимаем кнопку “4” и получаем наблюдение “+2”. Ну, формально, следовало бы писать на экране пару (4,2), то есть корень из 4 равен +2. Это правильный ответ.

2►Если мы нажимаем кнопку “4” и получаем наблюдение “+5”, то формально это действие (4,+5), то есть корень из 4 равен +5, что не верно.

Такого рода взаимодействие, когда действие – это пара «аргумент, результат», соответствует конечному автомату. Только в терминах конечных автоматов аргумент называется стимулом или входным символом, а результат – реакций или выходным символом.

Слайд 19. 1. Формализация тестирования.

Машина тестирования. Пример

На примере квадратного корня рассмотрим другой способ задания алфавита и машины тестирования. Идея в том, чтобы разделить пару (x,y) на два действия: действие x, понимаемое как передача в реализацию аргумента x, и действие y, понимаемое как получение от реализации результата y.

Соответственно, для каждого аргумента x≥0 имеем свою кнопку “x”. А для получения результатов y нас будет одна кнопка, которая здесь обозначена знаком квадратного корня. Эта кнопка разрешает получить любой результат.

1▶Теперь, когда мы нажимаем кнопку “4”, мы наблюдаем всего лишь передачу числа 4 в реализацию в качестве аргумента. А когда нажимаем кнопку квадратного корня, получаем одно из возможных наблюдения. Если это “+2”, то всё правильно.

2▶Но у числа 4, как известно, два значения квадратного корня: +2 и -2. Как получить все значения квадратного корня. В машине тестирования из предыдущего слайда это сделать или невозможно или слишком громоздко. А здесь нужно просто ещё раз нажать кнопку «корень», и машина выдаст второй корень “-2”. Разумеется, предполагается, что реализация выдаёт последовательно эти два корня.

Слайд 20. 1. Формализация тестирования.

Машина тестирования. Пример

Вот более интересный пример калькулятора. Здесь действия трёх типов. Первый тип – это передача аргумента в реализацию. Второй тип – это математическая операция, на слайде – сложить и умножить. Третий тип – это получение от реализации результата вычислений.

Каждому значению аргумента соответствует своя кнопка. На самом деле, калькуляторы устроены посложнее: так, что число можно набирать с помощью всего-навсего десяти цифр и запятой. Ну, легко догадаться, как сделать в соответствующей машине тестирования.

Каждому знаку математической операции также соответствует своя кнопка.

А вот для получения любого результата имеется одна единственная кнопка “равно”.

1▶Вот пусть нам нужно вычислить 2+3. Нажимаем кнопку “2”, потом кнопку “+”, потом кнопку “3” и, наконец, кнопку “=”. В конечном счете наблюдаем на экране ответ 5.

Слайд 21. 1. Формализация тестирования.

Машина тестирования. Пример

Если эти числа нужно перемножить, аналогично нажимаем кнопку “2”, потом кнопку “x”, потом кнопку “3” и, наконец, кнопку “=”. В конечном счете наблюдаем на экране ответ 6.

Слайд 22.1. Формализация тестирования.

Машина тестирования. Пример

Такого рода взаимодействие является частным случаем общего взаимодействия ввода-вывода. Действия делятся на стимулы и реакции. Стимул – это сообщение, передаваемое из окружения в реализацию, а реакция – сообщение, передаваемое в обратную сторону: от реализации в окружение.

В нотации CCS стимулы обозначаются с префиксным вопросительным знаком, а реакции – в восклицательным.

1►Вот мы нажимаем кнопку, означающую передачу стимула a, потом нажимаем кнопку приёма всех реакций, и получаем реакцию y.

Такое взаимодействие, когда за стимулом обязательно следует реакция, всё начинается со стимула и заканчивается реакцией, соответствует конечному автомату. Но это только частный случай машины ввода-вывода.

2►В общем же случае мы можем нажать послать несколько стимулов подряд и получить несколько реакций подряд.

Слайд 23.1. Формализация тестирования.

Приоритеты

Я уже говорил, что если кнопка разрешает выполнять несколько действий, то выбор выполняемого действия происходит, вообще говоря, недетерминированным образом. Если есть внутренняя активность, то выбор между выполнением внешнего действия и внутренней активностью также происходит, вообще говоря, недетерминированным образом.

Что значит «вообще говоря»?

Говорят, что машина не имеет приоритетов, если в данной текущей ситуации любое определенное и разрешённое действие выполнимо.

Это означает, что выполнимость определённого в реализации действия *не зависит* от того, какие ещё действия разрешает нажимаемая кнопка и какие ещё действия определены в машине в этот момент. времени.

Иными словами, выполнимость действия не зависит от состояния машины и от нажимаемой кнопки при условии, что в этом состоянии лействие определено и кнопка разрешает это действие.

1►В машине без приоритетов равноприоритетны все внешние действия и внутренняя активность.

Слайд 24.1. Формализация тестирования.

Приоритеты

Очевидно, что, если каждая кнопка, кроме пустой, разрешает ровно одно действие, а τ-активность всегда выполнима, то приоритетов нет.

1►Особый случай, когда кнопка может разрешать несколько действий, но каждое внешнее действие разрешается *не более чем одной кнопкой*, то есть семейство «кнопочных» множеств является разбиением алфавита внешних действий.

Тогда выполнимость разрешенного действия при наличии приоритетов зависит только от состояния машины.

В одних состояниях оно отсутствует и, естественно, не выполнимо.

В других определено и выполнимо.

В третьих определено, но не выполнимо.

Понятно, что первый и третий случай, фактически, совпадают: в обоих случаях это действие в этом состоянии никогда не выполняется. Это действие можно просто удалить из состояния третьего типа.

Тогда для отсутствия приоритетов нужно только, чтобы τ-активность была *всегда выполнима*.

Слайд 25.1. Формализация тестирования.

Приоритеты

Сейчас я дам только два примера систем с приоритетами.

Первый пример: прерывание внутренней активности внешним воздействием.

Мы можем заметить, что компьютер чего-то делает, даже если мы ему никаких заданий не давали, или если он это задание уже выполнил.

1▶Для этого достаточно вызвать диспетчер задач и посмотреть раздел «Быстродействие». Вы увидите, что иногда имеется внутренняя активность – загрузка процессора ненулевая. Но если вы кликните мышкой или нажмете кнопку клавиатуры, система сразу же реагирует на это и выполняет действие, разрешённое этим вашим тестовым воздействием.

Это означает, что внешние действия имеет приоритет над внутренней активностью.

Слайд 26. 1. Формализация тестирования.

Приоритеты

Второй пример: прерывание цепочки действий – операция «отменить».

Нашим тестовым воздействием мы можем запустить выполнение цепочки действий в реализации. Некоторые из этих действий могут быть и внешними, если мы в нужные моменты времени будем их разрешать нажатием соответствующей кнопки.

Однако есть такие внешние воздействия, которые отменяют выполнение этой цепочки. Это операция «отменить».

1▶Например, вы задали длинную перепись файлов. На экране высвечивается окошко, в котором показывается ход выполнения переписи, то есть вы видите наблюдаете некоторые действия машины. И тут же имеется кнопка «Отмена». Если вы ее нажмете, перепись прекратится.

Это означает, что внешнее действие «Отмена» имеет приоритет над другими внешними (наблюдаемыми) действиями и над внутренней активностью.

2▶Пока мы будем рассматривать машины без приоритетов.

Нужно сказать, что все так делаютю Правда, проблема приоритетов сегодня уже осознана, но в литературе нам не удалось найти её решение.В конце лекций, если получится, мы расскажем о том, как мы предлагаем работать с приоритетами.

Слайд 27.1. Формализация тестирования.

Практические предположения

Прежде чем двигаться дальше, сформулируем основное практическое требование к тестам:

Каждый тест должен заканчиваться за конечное время.

Понятно, что для этого сам тест должен быть конечным, то есть заканчиваться после конечного числа нажатий кнопок.

А кроме этого, нужны следующие практические предположения:

1) Конечная последовательность любых действий выполняется конечное время, и только бесконечная последовательность действий выполняется бесконечное время.
2) «Передача» тестового воздействия в реализацию и наблюдения от реализации выполняется за конечное время.

Слайд 28.1. Формализация тестирования.

Дивергенция

Дивергенция – это бесконечная внутренняя активность. Она обозначается символом Δ.

В частности, зацикливание программы, что обычно рассматривается как ошибка.

Прежде всего, нужно подчеркнуть, что дивергенция – это вовсе не обязательно ошибка зацикливания программы. Мы уже видели, что это может быть нормальный режим работы машины. Она просто делает какое-то своё внутреннее дело, то есть «думает», и может это делать бесконечно долго. Ничего страшного в этом нет. Плохо, если она при этом не реагирует на внешние воздействия. Если внешнее действие имеет приоритет над внутренней активностью, то, как мы уже видели, всё будет хорошо: выполнение внутренней активности прервётся, и реализация начнёт выполнять наше задание. Но для

этого нам нужны системы с приоритетами. А пока что мы рассматриваем системы без приоритетов.

Дивергенция – это характерный пример нереализуемого на практике наблюдения. Даже если у нас есть зелёная лампочка, которая показывает внутреннюю активность, мы не можем сказать, погаснет лампочка через какое-то время, что означает конечную внутреннюю активность, или будет гореть бесконечно долго, что означает дивергенцию.

Но хуже то, что дивергенция препятствует другим наблюдениям. Вот мы нажимаем кнопку и ожидаем внешнего действия. Предположим, что хотя бы одно из действий, разрешённых нажатой кнопкой, определено в реализации. Тогда, если дивергенции нет, то любая её внутренняя активность конечна, то есть исчезает через конечное время (зелёная лампочка, если она есть, гаснет). И мы наблюдаем внешнее действие. Но если есть дивергенция, мы можем ждать этого внешнего действия бесконечно долго, потому что при отсутствии приоритетов внутренняя активность может выполняться бесконечно долго.

Вообще говоря, после нажатия кнопки не обязательно должно наблюдаться разрешённое ею действие. Если такие действия в реализации не определены, то они не будут выполняться и мы их не будем наблюдать. При отсутствии дивергенции рано или поздно произойдёт остановка машины, зелёная лампочка погаснет, а на экране не появится внешнее действие. Но при наличии дивергенции мы не знаем, погаснет зелёная лампочка или нет.

При отсутствии зелёной лампочки мы вообще не различаем дивергенцию и остановку машины.

1►Поэтому, как правило, на реализацию налагают дополнительное ограничение: в ней не должно быть дивергенции. Но опасна ведь не сама по себе дивергенция, а попытка выхода из неё – тестовое воздействие во время дивергенции. Это первая опасность во время тестирования. Всего их будет три.

Тем самым, мы предлагаем более либеральный подход к дивергенции: запрещается не сама дивергенция, а тестовые воздействия во время дивергенции. В этом случае мы должны закончить выполнение теста. После этого можно снова запустить этот же или другой тест. Потом мы посмотрим, как определяется

возможность или невозможность дивергенции в реализации в той или иной ситуации.

А пока для нас важно следующее: попытка выхода из дивергенции опасна! Если дивергенция возможна, то сколько бы времени мы не ждали наблюдения после нажатия кнопки, мы не знаем: то ли наблюдение будет и нужно ещё подождать, то ли наблюдения не будет, поскольку бесконечно будет выполняться внутренняя активность.

Слайд 29. 1. Формализация тестирования.

Отказы

После нажатия кнопки наблюдается какое-нибудь внешнее действие, разрешаемое этой кнопкой, при выполнении двух условий:

1. В реализации в данный момент времени нет дивергенции.
2. В реализации определено хотя одно действие, разрешаемое нажатой кнопкой.

Если выполнено только первое условие, через конечное время возникает deadlock: внутренняя активность закончилась, и реализация не может выполнить ни одного из разрешённых действий. Машина останавливается. Эта ситуация называется отказом. Формально, отказ P – это остановка машины при нажатой кнопке P. Он определяется множеством действий P (*refusal set*).

Если есть зелёная лампочка, отказы можно наблюдать. Если после нажатия кнопки P зелёная лампочка погасла (или не горела), а экран пуст, то это и есть отказ P.

Другой способ основан на ограничении на время выполнения конечной внутренней активности и внешнего действия. У нас уже было практическое предположение о том, что время выполнения конечной активности и время выполнения внешнего действия конечны. Здесь же речь идёт о том, что это время ограничено сверху известной константой. В этом случае вводят соответствующий тайм-аут и, если после нажатия кнопки P тайм-аут истекает, а экран пуст, то это и означает отказ P.

Иногда можно считать, что реализация сама посылает машине тестирования «уведомление об отказе» выполнять разрешённые

действия. Тогда машина может известить об этом оператора каким-нибудь сигналом.

Слайд 30. 1. Формализация тестирования.

Отказы

Мы абстрагируемся от способа наблюдения отказа, но разделяем все отказы на два типа: одни наблюдаемые, а другие – нет. Поскольку отказы взаимно-однозначно соответствуют кнопкам параметризованной машины тестирования, семейство кнопок **P** разбивается на два семейства: **R** – кнопки с наблюдаемыми отказами, и **Q** – кнопки с ненаблюдаемыми отказами. Такую семантику мы называем ***R*/*Q***-семантикой.

Будем предполагать, что отказы не путаются с действиями, семейства **R** и **Q** не пересекаются и покрывают в совокупности алфавит внешних действий.

Будем считать, что при возникновении наблюдаемого отказа на экран высвечивается специальный символ θ. Зная нажатую кнопку P и наблюдая символ θ, мы фиксируем наблюдение отказа P.

1►Теперь у нас для каждой **R**-кнопки имеется два типа наблюдения: наблюдение внешнего действия и наблюдения отказа.

2►Для **Q**-кнопок имеется только один тип наблюдения: внешние действия. Если возникает **Q**-отказ, то он ненаблюдаем.

3►Ненаблюдаемый отказ – это вторая опасность при тестировании. Если мы нажимаем Q-кнопку и возможен ненаблюдаемый отказ, то сколько бы времени мы не ждали наблюдения после нажатия кнопки, мы не знаем: то ли будет выполнено внешнее действие и его нужно ещё подождать, то ли никаких действий не будет, поскольку возник отказ, а мы этого не знаем – ведь отказ ненаблюдаемый.

Слайд 31. 1. Формализация тестирования.

Пустые отказы

Я хочу обратить ваше внимание на пустой отказ, то есть пустое множество действий. В машине тестирования может быть такая

кнопка, которая разрешает пустое множество действий, то есть не разрешает никаких внешних действий.

Пустой отказ возникает, когда исчезает внутренняя активность и машина останавливается, поскольку выполнение внешних действий запрещено нажатой пустой кнопкой.

Если это **Q**-отказ, то он не наблюдаем. При тестировании мы должны избегать появления **Q**-отказов. Однако пустой **Q**-отказ всегда возможен, если нет дивергенции. Действительно, в этом случае внутренняя активность конечная и, поскольку внешние действия запрещены, реализация не будет их выполнять и через конечное время внутренняя активность исчезнет, то есть возникнет пустой **Q**-отказ.

Поскольку при дивергенции мы не должны нажимать никакие кнопки, получается что всегда, когда мы могли бы нажать пустую кнопку, пустой **Q**-отказ возможен. Следовательно, нажатие пустой **Q**-кнопки всегда опасно.

Поэтому-то в дальнейшем будем считать, что у нас нет пустой **Q**-кнопки.

В то же время пустой **R**-отказ безопасен всегда, когда нет дивергенции. Это как раз наблюдение исчезновения внутренней активности в ситуации, когда все внешние действия запрещены.

Ван Глаббек обозначал такое наблюдение символом S.

Но, конечно, наличие или отсутствие пустой **R**-кнопки зависит от той семантики взаимодействия, которую мы моделируем с помощью машины тестирования.

Слайд 32. 1. Формализация тестирования.

Отказы

Если $\boldsymbol{Q}=\varnothing$ и $\boldsymbol{R}=2^{\boldsymbol{L}}$, то это хорошо известная *failure trace semantics*.

Еще один пример – это семантика популярного отношения ***ioco***, когда действия разбиваются на стимулы (input) – множество X и реакции (output) – множество Y. Есть только одна ***R***-кнопка приема всех реакций, а соответствующий отказ δ маленькая называется по-английски *quiescence*, то есть молчание. Мы обычно называем это

стационарностью. Каждый стимул посылается в реализацию с помощью одной ***Q***-кнопки, которая разрешает реализации только одно действие: приём стимула *x*.

1►Практический пример отказа у нас уже был. Это «бледные кнопки» в меню графического интерфейса. Их можно понимать как блокированные стимулы. Только в отличие от ***ioco***-семантики, где блокировки стимулов ненаблюдаемы, здесь они хорошо видны на экране, хотя и бледные.

Другой пример я приведу для того, чтобы показать различие между отказом и «пустым» выполнением внешнего действия. Что такое это «пустое» выполнение? Было время, когда многие люди не желали признавать наличие отказов, особенно блокировки стимулов. Иногда это встречается и сейчас.

Эти люди рассуждают так. «Что значит “реализация блокирует стимул”? Куда этот стимул девается? Разве он не попадает в реализацию? Не правильнее ли считать, что реализация принимает стимул, но ничего после этого не делает, в том числе, не меняется её состояние. При таком подходе отказы не нужны, поскольку они понимаются просто как особый вид обработки стимула».

Вот моё возражение. Отказ отличается от «пустого» выполнения тем, что происходит тогда, когда машина стоит. Поэтому повторное нажатие той же самой **R**-кнопки или **R**-кнопки, разрешающей меньшее, то есть вложенное, множество действий, гарантированно вызовет отказ. Это не нужно тестировать, это входит в саму семантику отказа. А вот «пустое» выполнение действия тестировать нужно. Откуда мы знаем, что оно «пустое»? Нам сообщается только, что реализация выполнила данное действие, разрешённое нажатой кнопкой. А вдруг она изменила своё состояние?

А пример наблюдаемого отказа и «пустого» выполнения действия такой.

2►В магазине вы протягиваете карточку Viza, ее у вас берут, что-то проверяют, а потом возвращают со словами: «извините, нет связи с банком» или «платеж не прошел» или «покупка не оплачена» или что-то в этом роде. По идее, состояние, то есть сумма на вашем счете, не должна измениться. Но гарантии нет – этот факт нужно тестировать, поскольку возможно мошенничество.

А вот если вы протягиваете карточку Viza, а у вас ее не берут, она остаётся у вас в руках, и вам говорят: «мы карточки Viza не принимаем» или «вообще карточки не принимаем, платите наличными», то у вас есть гарантия, что состояние, то есть сумма на вашем счете, не изменилась.

Первый случай: действие выполняется, но система не меняет своё состояние, точнее – не должна была бы менять. Второй случай – это отказ: состояние гарантированно не меняется.

Вот почему наблюдаемые отказы так важны.

Ну, а ненаблюдаемые отказы, как я уже говорил, опасны и их следует избегать при тестировании.

Слайд 33. 1. Формализация тестирования.

Разрушение

Наконец, рассмотрим последний, третий вид опасности при тестировании. Это разрушение. Оно означает любое нежелательное при тестировании поведение реализации. Нежелательное не в том смысле, что неправильное, ошибочное, а в том смысле, что мы его должны избегать при тестировании.

Причины такой нежелательности могут быть различные.

1►Возможно реальное разрушение: потеря данных, поломка, взрыв и тому подобное.

2►Эта часть системы ещё не сделана (не отлажена).

3►Поведение системы не определено (нарушение предусловия).

4►Нарушение конфиденциальности – несанкционированный доступ, которого следует избегать при тестировании.

5►Действия, запрещённые по тем или иным причинам.

6►Спецификация определяет не только желательное поведение реализации, но и те случаи, когда в реализации допускается нежелательное поведение (неважно, какое именно).

7►После разрушения любое наблюдение недостоверно.

8►Если окружение «правильно» (т.е. согласно спецификации) взаимодействует с системой, разрушение не достижимо. Поскольку мы тестируем реализацию, а не окружение, будем при тестировании избегать разрушения.

9►Разрушение считается действием, обозначается символом γ. Разрушение всегда разрешено, как и внутренняя активность. Поэтому, когда машина стоит, в ней нет не только внутренней активности, но и разрушения. Если разрушения нет с самого начала, то оно может возникнуть только после нажатия какой-нибудь кнопки, после которой наблюдается какое-нибудь внешнее действие, разрешённое этой кнопкой, а после него – разрушение.

Можно отметить, что разрушение аналогично понятию запрещенного состояния, которое иногда используется в model checking. Но разрушение как запрещенное действие лучше, поскольку мы не опираемся на понятие состояния, а имеем дело только с наблюдениями.

Слайд 34. 1. Формализация тестирования.

Безопасное тестирование

Тестирование будем называть безопасным, если в процессе тестирования не возникают те три опасности, о которых мы уже говорили:

1) Нет попыток выхода из дивергенции, то есть не нажимается кнопка, если в реализации возможна дивергенция.
2) Не возникают ненаблюдаемые отказы, то есть не нажимается кнопка, если может возникнуть ненаблюдаемый отказ.
3) Не возникает разрушение, то есть не нажимается кнопка, если она разрешает действие, после которого может быть разрушение.

Слайд 35. 1. Формализация тестирования.

Недетерминизм и глобальное тестирование

Мы уже несколько раз говорили о том, что реализация может быть недетерминированной. Как можно тестировать такую реализацию?

Для этого была предложена, так называемая, гипотеза о глобальном тестировании.

Она основана на следующем предположении о недетерминизме реализации: недетерминизм – это явление только того уровня абстракции, которое определяется семантикой взаимодействия. Иными словами, поведение реализации, на самом деле, детерминировано – оно однозначно определяется тем, какие кнопки нажимает оператор, а ещё некими не учитываемыми нами факторами. Они называются погодными условиями.

1▶Гипотеза о глобальном тестировании утверждает, что любое погодное условие может быть воспроизведено в тестовом эксперименте на любом его шаге.

Как это можно моделировать в формализме машины тестирования? Было предложено несколько способов.

2▶Первый способ – это специальная кнопка репликации. Её нажатие создаёт на данном шаге тестирования множество копий машины с одним и тем же текущим состоянием – по крайней мере по одной копии для каждого варианта погодных условий *на этом шаге*. После этого можно продолжать эксперимент независимо с каждой из копий.

3▶Второй способ – это кнопка Undo – откат. Её нажатие возвращает машину на шаг назад для продолжения эксперимента с новым вариантом погодных условий *на предыдущем шаге*. Предполагается, что таким способом можно перебрать все погодные условия на каждом шаге.

Различие между этими кнопками в том, что кнопка репликации не ограничивает число погодных условий, а кнопка отката такие ограничения налагает. Если её нажимать конечное число раз, каждый раз после тестового воздействия, то есть после нажатия какой-то кнопки управления A,B,C, то мы получим конечное число погодных условий. А при бесконечной последовательности нажатий – счётное множество погодных условий.

4▶Кнопки репликации и отката позволяют тестировать конформности типа симуляции. Это конформности, основанные не только на наблюдениях внешних действий и отказов, но и на соответствии между состояниями реализации и спецификации. Одному состоянию реализации будут соответствовать все копии машины тестирования, созданные репликацией на данном шаге тестирования. Хотя, конечно, копиям машины, созданным на разных шагах, могут соответствовать разные состояния реализации.

О симуляциях мы ещё поговорим в конце этого курса лекций. А сейчас будем рассматривать нашу машину тестирования без кнопок репликации и отката. Но всё-таки как-то нам нужно перебирать погодные условия.

5▶Мы будем делать это с помощью операции Reset. Специальным ключом машина выключается и реализация сбрасывается в её начальное состояние. Потом машина снова включается. Предполагается, что последовательность включений Reset'ов перебирает все возможные погодные условия. Точнее, перебираются все возможные конечные последовательности погодных условий – по одному варианту погодных условий на каждом шаге. Если тестовый эксперимент состоит из десяти шагов, то есть десяти нажатий кнопок управления A,B,C, то ход эксперимента определяется последовательностью из десяти вариантов погодных условий.

Тестирование с помощью кнопок репликации и Undo – это один ветвящийся процесс. Статья Ван Глаббека, посвящённая этой теме, так и называлась «Линейное время – Спектр Ветвящегося Времени». При тестировании с помощью Reset'а у нас есть множество конечных тестов, каждый из которых мы прогоняем на машине тестирования несколько раз – столько, сколько нужно для перебора всех вариантов погодных условий. Между прогонами тестов выполняется Reset.

Разумеется, это чисто абстрактное моделирование погодных условий. Его цель – прояснить смысл гипотезы о глобальном тестировании. На практике возможны различные способы реализации такой гипотезы. Но об этом мы поговорим позже, когда речь пойдет о проблемах практического тестирования.

Слайд 36. 1. Формализация тестирования.

Истории и трассы

Итак, тестирование – это множество тестовых экспериментов. О числе таких экспериментов мы пока ничего не говорим. Но каждый тестовый эксперимент выполняется конечное время.

1▶Что является результатом такого конечного тестового эксперимента? Да просто запись всего того, что мы делали и что мы наблюдали. Это чередующаяся последовательность кнопок и наблюдений, начинающаяся кнопкой и заканчивающаяся наблюдением. Такую последовательность будем называть тестовой историей. Результат тестирования – это множество полученных историй.

2▶Наблюдения двух типов: действия и θ-наблюдения. θ-наблюдение после кнопки “A” интерпретируется как отказ A.

3▶Действие разрешается предшествующей кнопкой.

Отказ – это множество действий, разрешаемых предшествующей кнопкой, то есть само кнопочное множество.

4▶Подпоследовательность истории, состоящую из наблюдений, то есть с пропуском кнопок, называют трассой наблюдений, или просто трассой.

5▶Если приоритетов нет, то перед действием может быть любая кнопка, разрешающая это действие. Разумеется, с учётом безопасности тестирования. Поэтому по трассе восстанавливается множество всех историй, имеющих эту трассу в качестве подпоследовательности наблюдений. Поэтому до тех пор, пока мы рассматриваем системы без приоритетов, нам достаточно трасс. Результат тестирования – это множество полученных трасс наблюдений.

6▶Нам также нужно учесть опасности, которые могут возникать при тестировании: дивергенцию, ненаблюдаемые отказы и разрушение. Для этого мы будем также рассматривать трассы, заканчивающиеся **Q**-отказом, дивергенцией или разрушением. Из семантики отказа следует, что дивергенция и разрушение не могут следовать только после отказа.

Слайд 37. 1. Формализация тестирования.

Типы наблюдений в машине van Glabbeek'a

В заключении раздела о формализации тестирования я хочу рассказать о той классификации видов наблюдений, семантик взаимодействия и конформностей, которые сделал Ван Глаббек в двух свои статьях.

Соотношение между нашей параметризованной машиной тестирования и машиной Ван Глаббека в целом следующее.

С одной стороны, Ван Глаббек рассматривает такие типы наблюдений, которых у нас нет. В этом смысле у нашей машины меньше тестовых возможностей, чем у машины Ван Глаббека. Но, как нам кажется, мы реализуем в своей машине почти все практически значимые возможности. Исключение составляет, пожалуй, только лампочки действий, про которые я уже говорил. Эта возможность существует в графических интерфейсах. Другие типы наблюдений Ван Глаббека довольно экзотические и малопрактичные. Хотя теоретически они и важны.

С другой стороны, Ван Глаббек не рассматривает семантики, связанные с ограничениями на клавиатуру машины: какие множества действий можно разрешать, а какие нет, а также когда может наблюдаться отказ, а когда нет. Поэтому в классификацию Ван Глаббека не попала, например, такая популярная конформность как ***ioco***. Наша машина как раз ориентирована на параметризацию такими множествами разрушаемых действий – кнопками, с указанием, для каких кнопок отказы наблюдаемы, а для каких – нет.

Иными словами, классификация Ван Глаббека в значительной степени устарели. Однако, она остаётся интересной и полезной, поскольку это, насколько я знаю, единственная такая попытка.

Итак, Ван Глаббек насчитывает 30 типов наблюдений. Вот они все на слайде. Сразу нужно отметить, что не все возможные комбинации этих наблюдений соответствуют каким-то семантикам взаимодействия. Ван Глаббек устанавливает определённого рода зависимости между этими наблюдениями. Вот здесь 6 видов таких зависимостей.

Например, семантика должна обязательно содержать какие-то наблюдения. Или она должна содержать обязательно хотя бы одно наблюдения из некоторого множества наблюдений. Или не может содержать одно наблюдения без другого и так далее.

Я коротко скажу об этих наблюдениях и некоторых получающихся на них семантиках, чтобы вы имели представление о том, какие в теории бывают наблюдения.

1▶Первое наблюдение T означает выключение машины и конец тестового эксперимента.

2▶Есть наблюдение действия с возможной последующей тау-активностью или без неё.

3▶Возможная тау-активность без внешних действий.

4▶Наблюдение типа замыкания по тау-активности: перед каждым наблюдением может быть тау-активность. Исключение составляют отрицательные наблюдения.

5▶Вот набор этих наблюдений вместе с их дизъюнкцией определяет трассы как последовательности таких наблюдений без отказов. Соответствующая семантика называется трассовой семантикой.

6▶Далее наблюдение перехода реализации в терминальное состояние: все действия разрешены, а зелёная лампочка погасла.

7▶Это даёт возможность получать завершённые трассы, то есть трассы, заканчивающиеся в терминальных состояниях. Соответствующая семантика – это семантика завершенных трасс.

8▶Далее всевозможные отказы. Они различаются на конечные, то есть наблюдение отсутствия конечного множества действий, и бесконечные. А также на те, после которых тестирование может или не может продолжаться дальше.

9▶Так получаются две семантики с отказами в зависимости от того, могут ли отказы быть только в конце трасс или также и в середине трасс.

10▶ Есть наблюдение перехода реализации в стабильное состояние: когда внешние действия не разрешены, а зелёная лампочка погасла.

11▶Это наблюдения множеств готовности ready set. Опят же в зависимости от того, наблюдаются только конечные или также и бесконечные множества действий. И от того, можно ли продолжать тестирование после такого наблюдения или нет.

12▶Соответствующие трассы – это трассы готовности. И семантика трасс готовности.

13▶Далее идут наблюдения разного рода конъюнкций наблюдений, которые соответствуют разным типам репликаций.

14▶Дизъюнкция наблюдений как отдельный вид наблюдения.

15▶Бесконечное наблюдение. Имеется в виду возможность проводить тестирование бесконечное время.

16▶Различные виды отрицательных наблюдений. Это как бы наблюдение того факта, что какого-то наблюдения не бывает.

17▶Отдельное наблюдения – это дивергенция. Также есть наблюдение дивергенции или deadlock’a, когда мы их не можем различить.

18▶Наконец, замыкание наблюдений в связи с невозможностью наблюдения дивергенции или, соответственно, дивергенции или deadlock’a.

Слайд 38. 1. Формализация тестирования.

Конформности для машины van Glabbeek’a

На основе этих 30 типов наблюдений Ван Глаббек выделил 155 семантик взаимодействия и основанные на них конформности типа предпорядков и эквивалентностей.

Вот он изобразил это наглядно в виде такого графа.

2. Модели реализации и спецификации

Слайд 39. 2. Модели.

LTS-модель

Ну, а мы теперь переходим к моделям реализации и спецификации.

Наиболее распространенной моделью является система помеченных переходов – LTS (Labelled Transition System), которая определяется как ориентированный граф, вершины которого

называются состояниями, а дуги помечены внешними действиями или символами τ или γ и называются переходами.

Внутренняя активность моделируется как τ-действия.

1▶Переход из (пре)состояния *s* в (пост)состояние *s`* по символу *z* обозначается $s—z\rightarrow s`$. Выделяется начальное состояние, с которого реализация начинает работать при каждом рестарте.

2▶После рестарта реализация выполняет последовательность смежных переходов, то есть движение по маршруту, начинающемуся в начальном состоянии, каждый переход которого помечен действием *z*. Каждое такое действие *z* разрешается текущей нажатой кнопкой *P* машины тестирования. Разные действия, естественно, могут разрешаться разными кнопками. Действие *z* это действие из соответствующей кнопки P, внутреннее действие τ или разрушение γ.

3▶Отказ *P* в LTS порождается в *стабильном* состоянии, то есть состоянии, из которого не выходят τ- и γ-переходы, при условии, что из этого состояния не выходят также переходы по действиям из *P*.

4▶В примере мы видим 5 отказов в трех состояниях.

5▶Дивергенция моделируется бесконечным τ-маршрутом, то есть маршрут, все переходы которого помечены символом τ. Состояние дивергентно, если в нём начинается такой бесконечный τ-маршрут, в частности τ-цикл. Состояние конвергентно, если оно не дивергентно.

6▶В примере имеется одно дивергентное состояние.

7▶Для определения трасс LTS в каждом стабильном состоянии добавляются виртуальные петли по порождаемым отказам, переходы по разрушению перенаправляются в терминальное состояние, а в дивергентных состояниях добавляются переходы по Δ в терминальное состояние. После этого трасса LTS определяется как последовательность пометок на переходах маршрута, начинающегося в начальном состоянии и, по построению, не продолжающегося после Δ- или γ-перехода, с пропуском символа τ.

Кроме того, иногда мы будем рассматривать также трассы, начинающиеся не в начальном состоянии, в каком-то другом. Через двойную стрелку будем обозначать тот факт, что имеется трасса сигма, которая начинается в состоянии s и заканчивается в состоянии

s`. Это значит, что в состоянии s начинается маршрут, который заканчивается в состоянии s` и имеет трассу сигма.

В общем случае в данном состоянии может начинаться несколько маршрутов с одной трассой, заканчивающихся в разных состояниях. Поэтому следует говорить о множестве состояний после трассы.

LTS детерминирована, если каждая её трасса заканчивается ровно в одном состоянии.

8▶Вот как в примере получается трасса.

9▶Состояние *достижимо*, если до него можно добраться из начального состояния по некоторому маршруту. Иными словами, состояние достижимо, если в нём заканчивается некоторая трасса. Но, вообще говоря, трасса заканчивается в нескольких состояниях, если LTS недетерминирована.

Слайд 40. 2. Модели.

Трассовая модель

Как было уже сказано, для взаимодействия, основанного на наблюдениях, и при отсутствии приоритетов важны только трассы.

Множество трасс LTS можно рассматривать как самостоятельную *трассовую модель*.

Эти свойства определяют трассовую модель для заданной **R**/**Q**-семантики. Здесь важно, что все отказы в трассах – это наблюдаемые отказы, то есть отказы из семейства **R**.

Трассу, в которой все отказы из семейства **R**, будем называть **R**-трассой.

Трассовую модель с отказами из семейства **R** будем называть **R**-моделью.

Можно дать и независимое от LTS определение **R**-модели как множество трасс, обладающее характеристическим набором из 5 свойств.

Характеристичность этого набора свойств означает, что, во-первых, для любого множества трасс с этим набором свойств существует LTS с этим множеством трасс, и, во-вторых, множество

трасс любой LTS обладает этим набором свойств. С этой точки зрения LTS-модель и трассовая модель эквивалентны.

1▶Детальное доказательство этого утверждения можно найти в моей докторской диссертации вот по такой ссылке.

Теория конформности (функциональное тестирование программных систем на основе формальных моделей). LAP LAMBERT Academic Publishing, Saarbrucken, Germany, 2011, ISBN 978-3-8454-1747-9, 428 стр. (содержание книги доступно по адресу:

http://www.ispras.ru/~RedVerst/RedVerst/Publications/TR-01-2007.pdf)

Там также показано, что все эти свойства независимы друг от друга.

Допустимость: Δ и γ только в конце трасс.

Согласованность: после отказов нет действий из этих отказов, нет Δ и нет γ.

Конвергентность: если в конце трассы и после нее нет Δ и γ, то трасса продолжается $\forall$ **R**-отказом или действием из него.

Замкнутость: множество трасс замкнуто по ***d***-операции: операции удаления отказа из трассы.

Полнота: замкнутость по ***i***-операции: операции вставки **R**-отказа в трассу после префикса, не продолжающегося действиями из этого отказа и заканчивающегося каким-нибудь отказом.

Слайд 41.2. Модели.

Машина тестирования и две модели

На этом слайде показано соотношение машины тестирования и двух моделей.

Доказано, что они эквивалентны в следующем смысле.

1. Множество трасс наблюдений на машине тестирования является трассовой моделью.
2. По трассовой модели можно построить LTS, имеющую то же множество трасс.
3. Если внутрь чёрного ящика машины тестирования поместить LTS, то будет наблюдаться множество трасс этой LTS.

Слайд 42.2. Модели.

Трассовая и LTS-модель

На этом слайде показано соотношение двух моделей: трассовой и LTS.

Трассовой модели достаточно для определения отношений безопасности и конформности, а также для генерации тестов.

LTS-модель более «подробная», она основана на понятии состояния и, тем самым, содержит избыточную, с точки зрения конформности, информацию.

Поэтому одной трассовой модели соответствуют много различных LTS-моделей.

Зато для LTS можно определить композицию – операцию сборки составной системы из компонентов, что невозможно сделать для трасс наблюдений. Ну, об этом попозже.

Слайд 43.2. Модели.

Трассовая и LTS-модель

На этом слайде я хочу показать место LTS-модели среди других моделей, которые мы в этом курсе лекций, как правило, не будем рассматривать.

На этом слайде изображены три типа моделей и соответствующие им проблемы тестирования по мере усложнения.

Самая простая модель – математическая функция как абстракция чистой процедуры. Основная проблема здесь – большое количество значений аргумента, которые нужно проверить для того, чтобы убедиться, что результат всегда правильный.

У нас в институте есть один наш коллега, Виктор Кулямин, он, по-моему, читал у вас в Клубе какую-то лекцию. Так вот он довольно много времени потратил на изучение тестирования математических функций в различных библиотеках программ. Ему удавалось довольно быстро находить такие изощрённые значения аргументов, что синус оказывался больше единицы и тому подобное.

Другой особо тяжёлый случай – это компиляторы. Как бы то же функция, но её аргумент – это программа на языке, то есть большие данные со сложной структурой.

Вторая модель – это автомат, взаимодействие с которым сводится к той же самой простой схеме «стимул-реакция», что и для функции. Но, в отличие от функции, реакция зависит не только от стимула, но и от предыстории взаимодействия, формализуемой в понятии состояний. Добавляется проблема перебора состояний. Сложность в том, что сами состояния во многих случаях не наблюдаемы при тестировании.

Третья модель – это система помеченных переходов, по-английски, Labelled Transition System, LTS. Для таких систем характерно сложное взаимодействие. Теперь в ответ на стимул может быть несколько реакций или ни одной. Можно подавать несколько стимулов подряд или получать реакции без стимулов. Для LTS характерны такие явления как недетерминизм, отказы, а также дивергенция.

Для систем последнего типа применяются и другие модели: сети Петри, алгебраические спецификации, в частности ASM – Abstract State Machine, которые придумал Юрий Гуревич и которые вместе с ним используются в Microsoft Research, а также контрактные спецификации, построенные на пред- и постусловиях.

Можно назвать три причины, по которым мы будет заниматься в основном LTS-моделью:

1. Распространённость этой модели.
2. Удобство генерации тестов по LTS.
3. И то, что на LTS определена композиция, моделирующая взаимодействие компонентов составной системы.

Для первых двух пунктов нам часто удобнее будет использовать трассовую модель.

Слайд 44. 2. Модели.

Трассовые модели спецификации и реализации

Для спецификации в **R**/**Q**-семантике мы будем использовать **R**-модель.

В реализации нужно ещё учитывать ненаблюдаемые отказы. Поэтому нужны трассы как с наблюдаемыми, так и с ненаблюдаемыми отказами, то есть **R∪Q**-модель.

1►Для *failure trace semantics,* в которой все подмножества алфавита внешних действий являются наблюдаемыми отказами, ***R***-модель назовем *полной* трассовой моделью, а ее трассы – *полными* трассами. Полные трассы, не содержащие дивергенции и разрушения, называются *failure traces.*

2►Для ***ioco***-семантики ***R***-трассы (без Δ и γ) называются *suspension traces.* Это трассы, в которых, кроме действий – стимулов и реакций, встречается только отказ δ – отсутствие реакций.

Трассовая модель наименее избыточна с точки зрения верификации конформности, поскольку при тестировании мы наблюдаем только трассы. В частности, разные LTS могут иметь одно и то же множество трасс.

В то же время LTS более удобна для практического использования, поскольку является компактным способом описания трассовой модели. В частности, бесконечная трассовая модель может быть задана конечной LTS аналогично тому, как регулярные множества последовательностей задаются конечными порождающими графами (автоматами).

3►С другой стороны, LTS обладает существенным неудобством, связанным с недетерминизмом. Недетерминизм в LTS проявляется как наличие τ-переходов и/или «веера» переходов $s \text{—} z \rightarrow s`$ из одного состояния *s* по одному и тому же внешнему действию *z*, ведущих в разные постсостояния *s`*. Из-за этого трасса в LTS заканчивается, вообще говоря, не в одном состоянии, а во множестве состояний. Этот недетерминизм в LTS, вообще говоря, неустраним. Это объясняется тем, что трасса может продолжаться как отказом, так и действием из этого отказа. Такая трасса не может заканчиваться только в одном состоянии, поскольку в одном состоянии не может быть и отказа и действия из этого отказа.