И. Бурдонов, А. Косачев
Тестирование соответствия (conformance testing)
ИСПРАН
239 слайдов

О чём пойдёт речь?

Мы *не* будем рассказывать о:

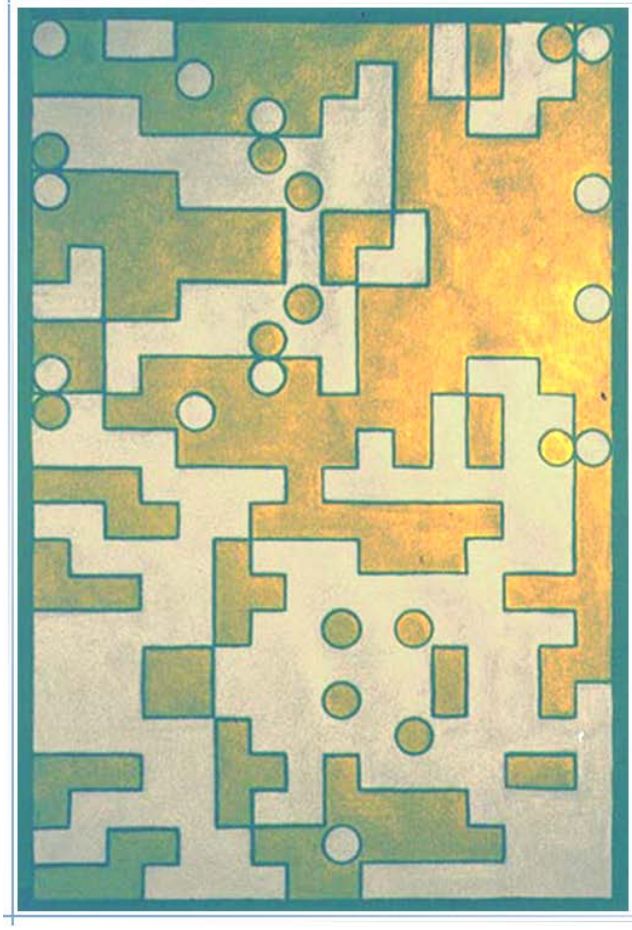
конкретных тестовых системах, инструментах, алгоритмах, языках спецификации, теориях тестирования с их формальными определениями и теоремами и т.п.

О чём пойдёт речь?

Мы хотим рассказать об основных *идеях*, лежащих в основе и общих как для теории, так и для практики тестирования соответствия.

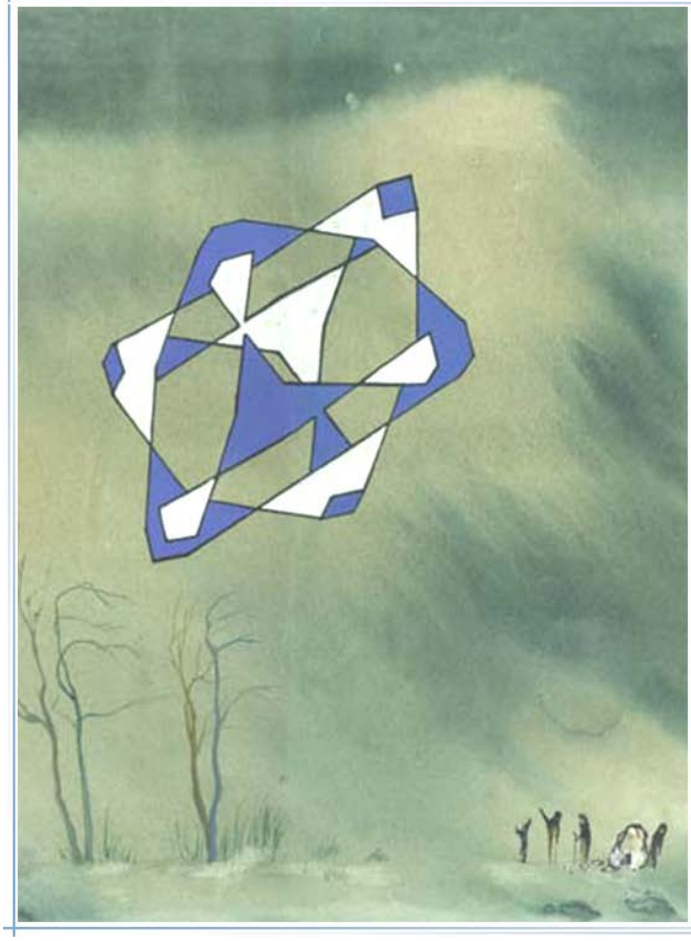
Основное внимание будет уделено возникающим здесь *проблемам*: как уже решённым, так и находящимся в процессе решения, то есть частично решённым, а частично нет, а также совсем нерешённым проблемам.

Также мы постараемся рассказать хотя бы о некоторых *методах решения* этих проблем: как тех, что уже используются, так и тех, что только предлагается использовать, или даже предполагаемых решениях, которые подсказывает интуиция, но про которые совсем не понятно, как их осмыслить, как их использовать и можно ли это вообще сделать.



ОГЛАВЛЕНИЕ

- Введение
- Автомат Мили
- Асинхронный автомат
- Асинхронное тестирование
- Композиционное тестирование
- Приоритеты



ВВЕДЕНИЕ

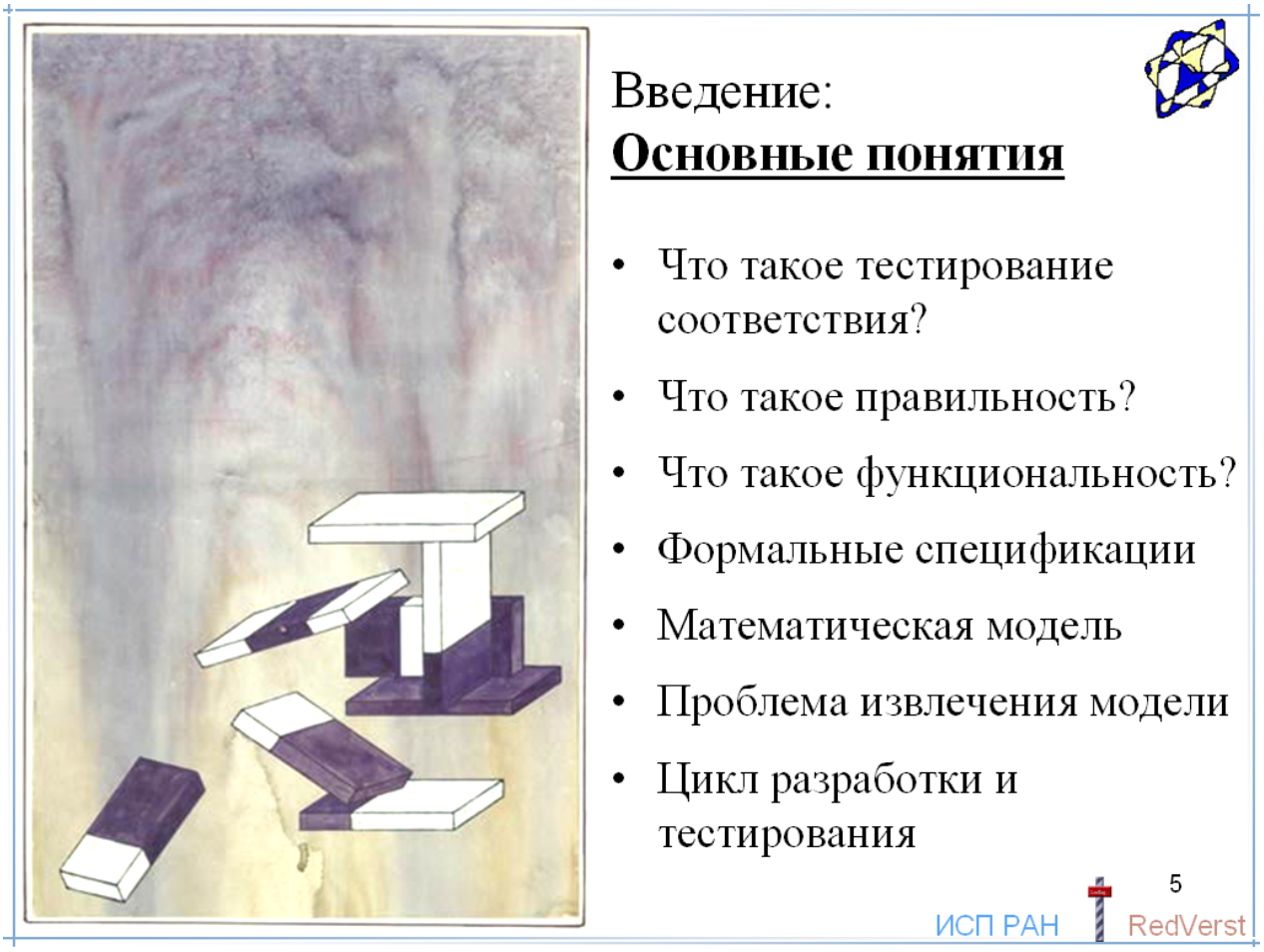
- Основные понятия
- Формализация эксперимента
- Математическая модель

ВВЕДЕНИЕ

ОСНОВНЫЕ ПОНЯТИЯ

ФОРМАЛИЗАЦИЯ ЭКСПЕРИМЕНТА

МАТЕМАТИЧЕСКАЯ МОДЕЛЬ



Что такое тестирование соответствия?

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 реализацией.

Чем отличается тестирование от других методов верификации?
Тестирование – это проверка правильности *в эксперименте.*
Этим оно отличается от аналитических методов *доказательства* правильности.

Что такое правильность?

Тестирование соответствия 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.

Важно подчеркнуть: не любые требования описываются такой спецификацией. Эти требования, прежде всего, должны быть функциональны.

Что такое функциональность?

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

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.

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.

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*?).

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.

Формальные спецификации.

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

Зачем?

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.

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.

Что касается тестирования, то здесь можно выделить две вещи, которые создаются на основе спецификации: 1) тесты, которые должны проводить интересующие нас эксперименты над реализацией, и 2) тестовые оракулы, которые проверяют правильность поведения реализации в каждом таком эксперименте.

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.

$S \in Spec$

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

$I \in Impl$

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 сосредотачиваем 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.

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

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.

$R \subseteq Impl \times Spec$

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

Но сначала мы хотим обратить внимание на очевидные проблемы, проистекающие из способа описания модели.

ПРОБЛЕМЫ: Извлечение модели.

Теперь мы 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 общая проблема *доказательства* и *тестирования* соответствия:
Как извлечь спецификационную 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 ещё и преобразуется для удобства тестирования. Это, так называемый, процесс факторизации модели, который мы рассмотрим позже.

Дополнительная проблема *доказательства* соответствия:
Как извлечь реализационную модель из «текста» реализации?
И что делать, если «текста» нет?

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. 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.

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, 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.

Цикл разработки и тестирования.

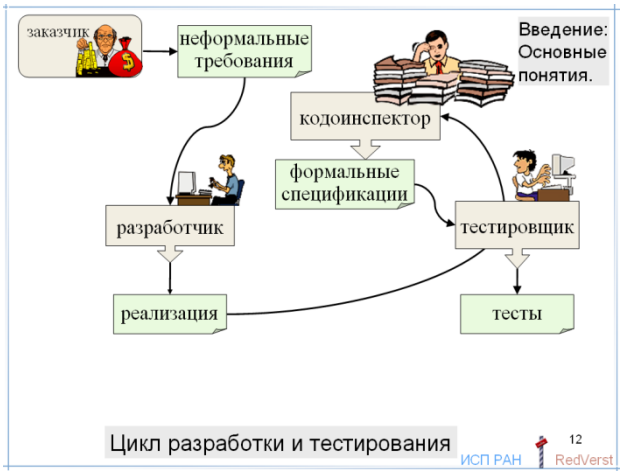
Прежде, чем двигаться дальше, имеет смысл рассмотреть место тестирования соответствия в общем процессе создания программного продукта.

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

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

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 по этому коду или одновременно с его написанием создаётся документация, которую можно использовать для спецификации. 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.

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



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.

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.

Частично и реализация и тесты могут генерироваться из спецификации автоматически. Для реализации это означает создание прототипа. Для тестовой системы автоматически могут генерироваться многие её части, которые мы рассмотрим позже. Но доля ручного труда всё равно остаётся.

Когда реализация и тесты готовы, начинаются тестовые эксперименты.

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. Это тоже оценивает группа анализа.

Цель тестирования – найти ошибки в реализации.

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.

Понятно, что наиболее неприятна ошибка в спецификации, поскольку она вызывает переделку, как реализации, так 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. Мы в группе RedVerst тоже кое-что пытались сделать для автоматизации кодоинспекции. Был проект, в котором предполагалось заспецифицировать миллионы строк кода, что, конечно, без хотя бы какой-нибудь автоматизации невозможно сделать за приемлемое время. К сожалению, это проект был прекращён не по нашей

инициативе или вине, но кое-что было придумано и сделано. 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.

Реализация и окружение.

Теперь вернёмся к соответствию между реализацией и спецификацией. Вспомним, что тестирование – это проверка правильности в процессе эксперимента над реализацией. Само понятие эксперимента подразумевает, что реализация не замкнута, то есть как-то взаимодействует с окружающей средой (окружением).

При тестировании тест подменяет собой окружение или его часть.

Тем самым, тест проверяет правильность взаимодействия реализации и окружения.

Функциональность требований.

Соответственно, функциональные требования к реализации должны быть сформулированы в терминах взаимодействия реализации с окружением. Здесь уже функциональность – это слово, производное от глагола *функционировать*, то есть действовать и взаимодействовать.

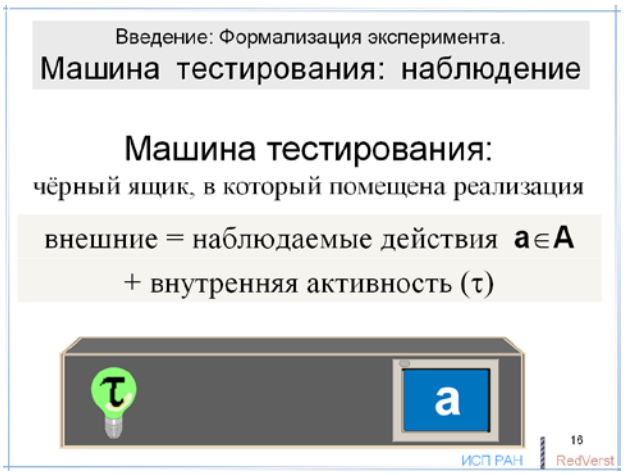
Теперь уже не любые требования функциональные в смысле *функции* остаются функциональными в смысле *взаимодействия*.

Вот лишь несколько примеров требований, которые часто предъявляют к программным продуктам, но которые не могут быть проверены при тестировании:
- Лицензионная чистота.

- Использование определённого языка программирования.
- Хорошая самодокументированность программы.
- Отсутствие неприличных идентификаторов.

Машина тестирования: наблюдение.

Теперь нам нужно формализовать само понятие эксперимента. Для этого используется термин *сценарий тестирования*, то есть формальное описание того, как происходит взаимодействие теста и реализации. (Не путать с термином сценарий тестирования как описание теста.) На самом деле такое описание ещё не вполне математическое, это как бы мостик между интуитивным представлением о том, что такое взаимодействие, и математическим формализмом. Поэтому, на первый взгляд, такое описание выглядит немного смешно, но оно очень полезно, если мы хотим понимать, что мы имеем в виду, употребляя слово «взаимодействие». Инструментом описания здесь служит, так называемая, *машина тестирования*. Такая машина формализует важную часть тестирования, которая, как мы уже говорили, во многом определяет то соответствие между реализацией и спецификацией, которое мы можем выбирать. Это *тестовые возможности*, а именно: возможности *по наблюдению* за поведением реализации и возможности *по управлению* этим поведением.



Машина тестирования представляет собой чёрный ящик, внутрь которого помещена 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.

Функционирование машины понимается как последовательность дискретных действий, которые она совершает. Речь идёт о, так называемых, внешних или наблюдаемых действиях. Иными словами, мы должны уметь разбить внешнее, наблюдаемое поведение машины на отдельные действия. Наверное, можно было бы рассматривать машины непрерывного действия, но мы пока не встречали работ на эту тему. Правда, мы не особенно их искали, поскольку вся практика тестирования, с которой приходилось иметь дело, вполне укладывается в дискретную форму.

Считается, что задан алфавит внешних действий A. Когда машина совершает действие $a \in A$, мы может это действие наблюдать. Для этого машина снабжается дисплеем, на котором высвечивается символ a. Чтобы различать несколько действий a, идущих подряд, обычно считают, что дисплей гаснет после завершения одного действия и снова загорается в начале следующего действия. Тем самым, десять a подряд вызовет десять миганий дисплея. Вместо дисплея можно представлять себе печатающее устройство, выводящее на бумагу символ a каждый раз, когда совершается действие a.

Кроме внешних действий, машина может иметь, так называемую, *внутреннюю активность*, которую принято обозначать символом τ. Это та часть поведения машины, которая нам не видима, поэтому её и нет смысла разбивать на отдельные и, тем более, различные действия. Тем не менее, иногда мы можем наблюдать сам факт того, что машина имеет внутреннюю

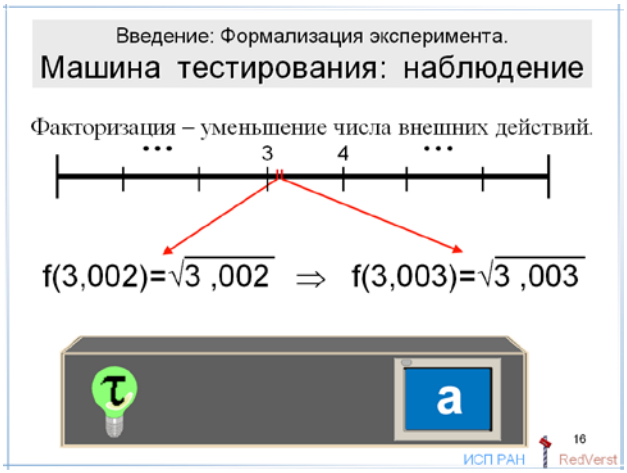
активность: мы видим, что машина что-то делает, а что именно неизвестно. В таком случае машина снабжается зелёной лампочкой, которая горит, пока машина имеет внутреннюю активность. Следует отметить, что такая тестовая возможность может, как быть, так и не быть. Например, если тест и реализация работают 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.

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.

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 подготовки к генерации тестов. Факторизация модели, на которую мы уже намекали, предназначена, в том числе, и для того, чтобы уменьшить алфавит действий.

Например, для функции вычисления квадратного корня под действием естественно понимать передачу в функцию аргумента `x` и возврат из функции результата `y`. Если мы рассматриваем функцию в действительных числах (не в комплексных), домен функции – это все неотрицательные действительные числа. А это очень много, даже если ограничиться конечным подмножеством, которое задаётся разрядной сеткой компьютера. Все их не переберёшь.

Один из методов факторизации предлагает разбить это множество на отрезки и проверять по одному числу из каждого отрезка. Тем самым, мы перестаём различать число `3,002` и число `3,003`.



Разумеется, при этом предполагается выполненной следующая реализационная гипотеза: если реализация правильно вычисляет квадратный корень из `3,002`, то она его правильно вычислит из `3,003`. Насколько обоснованными могут быть такие гипотезы – отдельная тема.

Но вернёмся к машине тестирования.

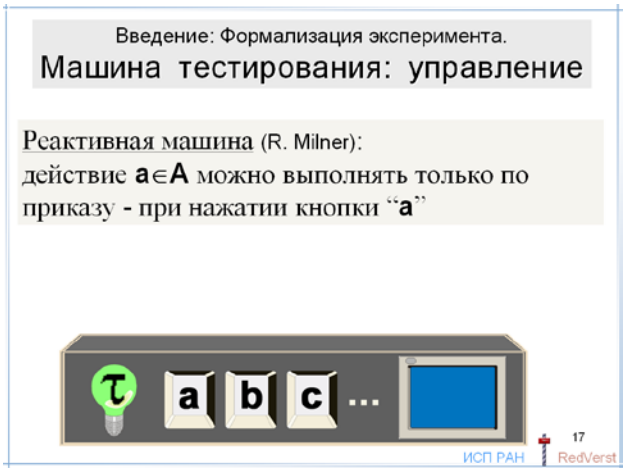
Машина тестирования: управление.

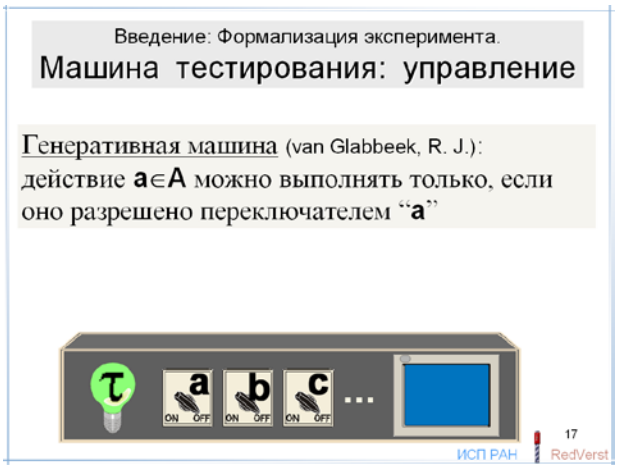
Обычно считается, что управлять можно только внешними действиями машины, а внутренняя активность неуправляема. Тем не менее, существует разновидность тестирования, когда машина находится под полным управлением теста. Это хорошо всем известная отладка, обычно автономная, когда мы можем двигаться по командам или вызовам процедур. Что понимать под действием, а что под внутренней активностью, определяется уровнем рассмотрения. Если мы двигаемся по вызовам процедур, то зацикливание внутри процедуры оказывается неуправляемым внутренним действием. Но если мы двигаемся по командам, такую

внутреннюю активность мы проходим шаг за шагом как последовательность наблюдаемых действий.

Если не считать этот особый случай, утверждение о неуправляемости внутренней активности остаётся верным для любого тестирования. Ещё об одном возможном исключении пойдёт речь позже. Мы не раз будем возвращаться к внутренней активности, а сейчас посмотрим, как можно управлять внешними действиями. Есть две модели такого управления.

Первая модель – это реактивная машина, предложенная R. Milner. Она работает *по приказу*. Для выполнения действия a нужно нажать кнопку, на которой написано a. Если при нажатии кнопка проваливается, машина выполняет действие a; если кнопка не проваливается (заблокирована), это значит, что машина не может выполнить действие a, но тогда она и никакого другого действия не выполняет.





Вторая модель – это генеративная машина, предложенная van Glabbeek. Она работает как бы *сама по себе*. Но оператор может запретить или разрешить выполнение действия a с помощью переключателя, на котором написано a и которое может быть в двух положениях **on** и **off**.

Glabbeek же показал, что различия между этими моделями маргинальны. Если разрешить в реактивной машине нажимать сразу несколько кнопок, оставляя за машиной выбор того из этих действий, которое она совершает, то сценарии тестирования с этими машинами имеют одинаковую мощность тестирования. В дальнейшем мы не будем делать различий между этими типами машин.

Здесь важно отметить, что машин тестирования может быть много: они описывают различные тестовые возможности по наблюдению и управлению взаимодействием. Мы будем рассматривать класс машин, в которых есть кнопки, и на кнопке написан не один символ, а подмножество символов, из которых машина выбирает один символ для выполнения по своему усмотрению. Тогда тестовые возможности управления определяются тем, для каких подмножеств есть кнопки, а для каких нет.

Одновременно может быть нажата только одна кнопка. Если нет нажатых кнопок, мы можем нажать любую кнопку, и она провалится. Кнопка остаётся утопленной до тех пор, пока машина не выполнит одно из действий, написанных на кнопке, и на дисплее не появится символ этого действия. После этого кнопка автоматически отжимается.

Кроме этого, у нас может быть или не быть возможность самим отжать кнопку, если нам надоело ждать, то есть по истечении какого-то времени – тайм-аута. Мы будем считать, что

механизм тайм-аута встроен в саму кнопку машины тестирования так, что по истечении тайм-аута на дисплее появляется специальный символ истечения тайм-аута. Кнопка при этом может автоматически отжиматься, а может остаться навечно утопленной. Различие определяется тем, можем мы продолжать тестирование после истечения тайм-аута или нет. Ещё раз повторим, что такой тайм-аут в некоторые кнопки может быть встроен, а в некоторые нет. В последнем случае кнопка остаётся нажатой до тех пор, пока машина не выполнит одно из разрешённых действий или до конца тестового эксперимента.

Например, функция вычисления квадратного корня.

Для ввода аргумента, например, `4` используется кнопка, на которой написано «`аргумент 4`»; для аргумента `5` будет своя кнопка «`аргумент 5`», и так далее. Нам, очевидно, бессмысленно разрешать машине выбирать по своему усмотрению аргумент квадратного корня, поэтому для ввода аргументов других кнопок нет.

Однако, поскольку мы не можем, да и не хотим, запретить функции вернуть то значение, которое она вычислила, мы должны после ввода аргумента `4` нажать кнопку, на которой написано «`результат любой`», то есть множество всех действительных чисел. И других кнопок для получения результата у нас нет. Эта ситуация довольно типична для взаимодействий, основанных на разделении действий на ввод и вывод. К этому мы ещё вернёмся.

Машина тестирования: функционирование.

Теперь сформулируем общие правила функционирования машины тестирования. Как она работает?

Прежде всего, выполняется принцип, который можно назвать принципом синхронного тестирования:

- ♣Машина выполняет только те внешние действия, которые разрешены кнопкой, нажатой оператором.
- ♣Если в машине есть внутренняя активность, она может, как выполнять внешние действия, так и не выполнять их. Это ситуация, когда машина «думает».
- ♣Если в машине нет внутренней активности, она либо выполняет внешнее действие, либо «стоит».

Когда внутренняя активность отсутствует, говорят, что машина находится в *стабильном состоянии*. Это состояние она может покинуть, только выполнив внешнее действие – одно из тех, которые в этом состоянии определены в самой машине и которые разрешены оператором извне с помощью нажатой кнопки. Если таких внешних действий нет, машина ничего не делает – «стоит».

Трассы наблюдений.

Таким образом, мы видим, что результатом эксперимента является последовательность наблюдений, которую называют *трассой наблюдений*.

Какие бывают наблюдения?

Прежде всего, конечно, внешние действия. Это основной тип наблюдения и он присутствует во всех машинах тестирования.

Если есть зелёная лампочка, мы имеем наблюдение внутренней активности.

Если выполнение внешнего действия ограничено по времени, то мы можем распознавать остановку машины в стабильном состоянии: зелёная лампочка погасла, после чего время ожидания внешнего действия истекло, кнопка с тайм-аутом автоматически отжалась или не было нажатых кнопок. В этом случае мы можем также видеть, что некоторые действия не определены в этом стабильном состоянии, а именно те, которые разрешены нажатой кнопкой. Про действия, запрещённые нажатой кнопкой, мы ничего сказать не можем: они могут быть, как определены, так и не определены в этом стабильном состоянии.

Такое множество действий, которые машины отказывается выполнить, называют *отказом* или, по-английски, *refusal set*. Трассы с такими наблюдениями называют *трассами с отказами* или *failure traces.*

В качестве примера вычисление квадратного корня уже не годится. Зато годятся многие графические интерфейсы: мы нажимаем какую-то кнопку, а ничего не происходит; подглядываем за процессорным временем и видим, что приложение не работает, хотя по приоритету имеет такую возможность. Значит, оно стоит, не реагируя на нашу кнопку. Аналогичная ситуация возникает в сетевых протоколах, когда другая сторона молчит долгое время. Правда, поскольку здесь мы лишены зелёной лампочки, у нас нет уверенности, стоит ли другая сторона, блокируя наши обращения, или она зациклилась во внутренней активности. Кстати, теория тестирования соответствия возникла, главным образом, как раз из практики тестирования протоколов.

Итак, мы рассмотрели уже четыре типа наблюдений: внешнее действие, внутренняя активность, остановка и отказ.

van Glabbeek описал примерно 30 таких типов наблюдений.

Не все из них легко реализуемы на практике, а про некоторые можно сказать, что их практическая реализация невозможна. Тем не менее, польза от них не только теоретическая. Хотя они не наблюдаемы на практике, зато позволяют ввести нужные ограничения на реализацию, чтобы тестирование было достоверным. Характерным примером является наблюдение дивергенции, который мы сейчас рассмотрим.

Но сначала заметим, что в общем случае выбор набора типов наблюдений определяется тестовыми возможностями и реализационными гипотезами. В каждом конкретном случае нужно добиться ясного понимания того, что мы можем делать при тестировании, и какими могут быть тестируемые реализации.

Набор типов наблюдений определяет алфавит – множество наблюдений этих типов. В этом алфавите рассматриваются трассы как последовательности наблюдений. Поскольку всё, что мы имеем от эксперимента – это трассы наблюдений, нас будут интересовать только такие соответствия между реализацией и спецификацией, которые могут быть сформулированы в терминах трасс наблюдений. Более конкретно: сравниваются множество трасс наблюдений, которые могут быть получены в экспериментах над реализацией, и множество трасс наблюдений, задаваемое спецификационной моделью.

van Glabbeek описал 155 типов таких соответствий. Это уже большой прогресс по сравнению с необъятным числом математических соответствий как подмножеств декартового произведения. Правда, van Glabbeek не рассматривал машины с разделением действий на ввод и вывод, а здесь есть свои особые соответствия, к которым мы ещё вернёмся.

А сейчас подробнее рассмотрим две проблемы: дивергенцию и недетерминизм.

ПРОБЛЕМЫ: Дивергенция.

Машина *дивергентна*, если в ней есть бесконечная внутренняя активность.
Машина *конвергентна*, если любая её внутренняя активность конечна, то есть исчезает через конечное время (лампочка гаснет).

Характерным примером нереализуемого на практике наблюдения является наблюдение дивергенции.
Дивергенция – это бесконечная внутренняя активность, моделирующая, в частности, зацикливание программы.

Даже если у нас есть зелёная лампочка, для дивергентной машины мы не можем сказать, погаснет лампочка через какое-то время или будет гореть бесконечно долго. Иными словами, мы не знаем, ждать нам какого-то внешнего действия или это бессмысленно. Хуже всего то, что при отсутствии зелёной лампочки мы не можем различить дивергенцию и остановку в стабильном состоянии. Тем самым, под вопросом оказываются наблюдения отказов. Поэтому, как правило, на реализацию налагают дополнительное ограничение: в ней не должно быть дивергенции. Такие машины называют *конвергентными* (точнее, *строго конвергентными*). К проблеме дивергенции мы ещё вернёмся, а сейчас посмотрим, что у нас получается с понятием соответствия.

Если машина конвергентна и есть зелёная лампочка, то при тестировании мы гарантированно наблюдаем через конечное время выполнение внешнего действия или остановку машины.

Если зелёной лампочки нет, то обычно налагают временн*о*е ограничение не только на внешнее действие, но и на внутреннюю активность. Понятно, что это ограничение на тестируемую реализацию и должно быть чётко сформулировано в виде реализационной гипотезы. Это ограничение не означает, что мы не можем *определить* дивергенцию, а только то, что мы не можем *различить* дивергенцию как бесконечную внутреннюю активность и конечную, но слишком долгую, внутреннюю активность. В этом случае при моделировании мы можем абстрагироваться от такого различения и слишком долгую внутреннюю активность изображать бесконечной внутренней активностью, то есть дивергенцией. Понятно, что на практике это различение во многих случаях и не требуется по той причине, что превышение тайм-аута на внутреннюю активность тоже рассматривается как ошибка.

Однако, это далеко не всегда так. Проблема не так проста, как может показаться.

ПРОБЛЕМЫ: Дивергенция не обязательно ошибка. Внешняя и внутренняя дивергенция.

Прежде всего, нужно подчеркнуть, что дивергенция – это вовсе не обязательно ошибка зацикливания программы.

Мы уже говорили, что при тестировании тест может подменять собой или перехватывать обращения не ко всему окружению, а только к его части. В этом случае дивергенция может быть бесконечным взаимодействием реализации с остальной частью окружения. А это может быть не только не ошибочное, но, напротив, предписываемое требованиями поведение реализации. Например, другая часть окружения имеет просто б*о*льший приоритет, чем тест, и реализация, учитывая приоритеты, обрабатывает в первую очередь неиссякающий поток обращений от более приоритетных клиентов. Эта ситуация весьма характерна для фоновых тестов, которые не должны не только нарушать работу системы в целом, но и существенно снижать её производительность.

Конечно, идеальным решением проблемы был бы полный перехват взаимодействия реализации с окружением. Чтобы не нарушать работу, тест только фиксировал бы взаимодействие с другой частью окружения и прозрачно пропускал бы это взаимодействие через себя. К сожалению, часто полный перехват взаимодействия с окружением оказывается невозможным, а если и возможным, то экономически невыгодным – на разработку перехватчиков всех видов взаимодействия большой реализации, например, операционной системы или её ядра, требуется много труда.

Но даже если мы будем рассматривать только «внутреннюю» дивергенцию, без взаимодействия с окружением, то и такое зацикливание не обязательно ошибка. На самом деле проблема ведь не в том, что машина может долго или даже бесконечно долго думать о чём-то своём, машинном, а в том, что она предаётся пустым размышлениям тогда, когда извне от неё требуется выполнение конкретных действий. Если каждый раз, когда такое требование возникает, машина прерывает свои размышления и делает то, что от неё требуется, то это то поведение, какое надо, и наличие дивергенции не является ошибкой. Характерный пример – сборка мусора, которой реализация может заниматься, сколько её душе угодно, но только не тогда, когда поступает запрос на выполнение работы.

К сожалению, подобная ситуация плохо отражается в современной теории тестирования. Тот же van Glabbeek предполагает, что разрешение внешнего действия a определяется только переключателем a и не зависит от положения переключателя b. Иными словами, нет никакого встроенного механизма приоритетов внешних действий по отношению друг к другу, на что van Glabbeek`у указал Jan Bergstra.

На самом деле, такой механизм часто используется на практике: например, приказ «`выполнить операцию A`» может запустить целую серию действий, а приказ «`отменить операцию A`» должен прерывать эту серию в любой точке, то есть он должен быть более приоритетным, чем любое из действий в серии.

В обычно используемых моделях таких приоритетов нет, и отмена приказа выполняется равновероятно с продолжением выполнением приказа до самого конца независимо от того, в какой момент времени пришёл приказ об отмене. В ещё большей степени это относится к взаимным приоритетам внешних действий и внутренней активности: предполагается, что внутренняя активность может быть независимо от положения переключателей, то есть независимо от требований выполнить внешние действия.

В нашей группе RedVerst разработан тип моделей, в которых ввод (но не вывод) имеет приоритет над внутренней активностью. Но об этом мы скажем подробнее позже.

ПРОБЛЕМЫ: (Наблюдаемый) недетерминизм.

Работа машины может зависеть не только от действий оператора (нажатие кнопок в определённой последовательности), но и от неучитываемых внешних факторов – «*погодных условий*».

Тем самым, один и тот же тест при разных прогонах может давать разные трассы наблюдений. Это и называется *недетерминизмом*. Более точно это нужно называть *наблюдаемым недетерминизмом*, поскольку речь идёт только о таком недетерминизме реализации, который наблюдаем в тестовом эксперименте.

В чём здесь проблема? В том, что, сравнивая трассы реализации, полученные в результате некоторого числа прогонов тестов, со спецификацией, мы не можем достоверно определить, правильная реализация или нет. Обычно, если мы нашли ошибку, то соответствия нет. Однако если мы ошибку не нашли, то это не означает, что её нет. Может быть, если бы мы ещё раз прогнали те же самые тесты, то из-за недетерминизма они дали бы новые трассы реализации, которые спецификация признала бы ошибочными.

Понятно, что в такой постановке проблема недетерминизма не имеет решения. Нужно искать обходные пути.

Одним из таких путей является, конечно, учёт этих самых неучитываемых погодных условий. Нам, однако, нужно не просто учитывать, но и управлять погодой, а это часто невозможно. Дело даже не в том, что часто мы не знаем, как это делать. Дело в том, что для того, чтобы это делать, мы должны выйти за рамки модели, которая как раз и абстрагировалась от второстепенных деталей внешних факторов. Тем самым, тестирование становится зависящим не только от спецификации, но и от реализационных деталей, от того, что можно назвать операционной обстановкой, в которой работает реализация. Для каждого варианта такой операционной обстановки мы будем вынуждены создавать свой набор тестов. Тем не менее, в некоторых частных случаях на этом пути можно получить практические выгоды.

В качестве примера можно рассмотреть недетерминизм, являющийся следствием параллелизма. Если реализация раскладывается на несколько параллельных взаимодействующих процессов, или если её функциональность предполагает одновременное обслуживание нескольких параллельных процессов-клиентов, недетерминизм возникает как следствие того или иного порядка выполнения процессов. Этот порядок определяется системным планировщиком процессов. Если у теста есть возможность влиять на работу планировщика, хотя бы меняя различным образом приоритеты процессов, то, тем самым, у нас есть возможность перебора погодных условий.

Другой обходной путь – это простой запрет на недетерминизм. То есть реализационная гипотеза ограничивает класс реализаций только детерминированными реализациями. При всей своей наивности, это достаточно распространённый приём. Обоснованием может служить то, что во многих случаях заранее известно, что интересующие нас реализации детерминированы.

Например, функция вычисления квадратного корня. Как известно, корень из числа `4` имеет два значения: `+2` и `−2`. Однако трудно представить себе такую реализацию, которая недетерминированным образом возвращала бы то `+2`, то `−2`. Теоретически такая реализация может быть, и её даже нетрудно написать, но только если специально ставить себе цель ввести недетерминизм; детерминированное вычисление квадратного корня написать проще, что все и делают.

ПРОБЛЕМЫ: Недетерминизм спецификации.

Здесь возникает важная смежная проблема недетерминизма спецификации. Для того же квадратного корня спецификация в имплицитной форме $y^2=x$ ничего не говорит о знаке возвращаемого значения, допуская тем самым как $+2$, так и -2. Эксплицитная спецификация, если она не хочет вводить дополнительного ограничения, делает это ещё более явно с помощью дизъюнкции: $y=+\textbf{алгоритм}\_\sqrt{(x)} \vee y=-\textbf{алгоритм}\_\sqrt{(x)}$. Таким образом, спецификация недетерминирована, однако, этот недетерминизм не обязательно понимать так, что реализация недетерминирована. Мы можем потребовать дополнительно, чтобы реализация была детерминированной, однако спецификация всё равно останется недетерминированной.

Это происходит потому, что спецификация описывает, фактически, не одну реализацию, а класс всех реализаций, в данном случае класс всех детерминированных реализаций, удовлетворяющих сформулированным требованиям. Дизъюнкция не означает, что реализация должна уметь возвращать при аргументе 4 как $+2$, так и -2. Она всегда может возвращать что-то одно, но только не 3 и не 5.

Детерминизм или недетерминизм спецификации определяется формой требований, которые она предъявляет реализации. В общем случае реализация и спецификация связаны отношениями *may* & *must*, *может* и *должна*. Реализация *должна* выполнять одни действия, определяемые спецификацией, но *может* выполнять лишь некоторые действия из списка возможных вариантов, предлагаемых спецификацией. Грубо говоря, спецификация недетерминирована, если этот список состоит более чем из одного элемента. В нашем примере функция квадратного корня должна уметь вычислять корень из аргумента 4, но может выдать в ответ любой из двух результатов $+2$ или -2. Этот пример является частным случаем разделения всех действий на две группы: стимулы и реакции. Обычно считается, что реализация *должна* принимать стимулы и *может* выдавать некоторые из разрешённых реакций.

В нашей практике был случай далеко не такой тривиальный, как квадратный корень. Это была одна из программ распределения памяти – буфер для строк переменной длины. Написать спецификацию этой программы детерминированным образом было невозможно, не вводя лишних, то есть не функциональных, ограничений. Однако реализация, естественно, предполагалась детерминированной, как оно на самом деле и было. Более того, алгоритм тестирования был способен тестировать только детерминированные реализации. При этом он мог обнаруживать проявление недетерминизма, и, если это случалось, фиксировалась ошибка. И всё прекрасно работало. Кажется, даже была найдена какая-то ошибка.

Итак, первый вариант: это детерминированная реализация при недетерминированной спецификации.

Проблема возникает, когда по самому смыслу спецификации она предполагает недетерминированную реализацию. В этом случае применяются специальные методы факторизации спецификации.

Один из методов состоит в том, что множество спецификационных трасс разбивается на классы эквивалентности так, что фактор-спецификация оказывается детерминированной. Тем самым, фактор-реализация по той же эквивалентности должна быть детерминированной, если она соответствует спецификации. Естественно, как при любой эквивалентности, здесь предполагается реализационная гипотеза: если всё хорошо для одной трассы из класса, то всё

будет хорошо и для других трасс этого класса. При тестировании не только проверяется по фактор-спецификации правильность полученной трассы, но и контролируется детерминизм. Обнаружение недетерминизма при таком фактор-тестировании также означает ошибку.

Этот метод не всегда может работать, то есть даже после разумной факторизации спецификация остаётся недетерминированной. Тестирование по такой спецификации при некоторых ограничениях также возможно. Это, так называемые, слабо детерминированные спецификации для систем с разделением на стимулы и реакции. Но об этом мы поговорим позже.

Тесты: значимые, исчерпывающие и полные.

А теперь более подробно рассмотрим вопрос: что такое тест? В английской терминологии используется термин *test case*. По-русски это можно перевести как *тестовый пример*. Для краткости мы будем говорить просто *тест*.

Для практического использования тест предполагается конечным: он должен заканчиваться за конечное время. В терминах машины тестирования это означает, во-первых, конечное число тестовых воздействий (нажатий кнопок) и наблюдений (считываний символов с дисплея и миганий зелёной лампочки). Во-вторых, это означает, что тест не должен попадать в deadlock: не должно возникать ситуации, когда тест ждёт чего-то бесконечно долго. Последнее решается выходом из deadlock`а с помощью тайм-аутов: тест ждёт только ограниченное, фиксированное время, а по его истечению продолжает свою работу, естественно «зная», что сработал тайм-аут. В терминах машины тестирования это означает встроенность тайм-аута в те кнопки, при нажатии которых возможно слишком долгое ожидание.

В конце работы тест должен вынести вердикт: ***pass*** или ***fail***. Вердикт ***fail*** означает, что обнаружена ошибка. Вердикт ***pass*** означает, что ошибка не обнаружена, а вовсе не то, что её нет.

В силу недетерминизма разные прогоны одного и того же теста могут давать, вообще говоря, разные вердикты. Как мы говорили выше, это зависит от погодных условий.

♣

Говорят, что реализация *проходит* тест, если при любых погодных условиях тест выносит вердикт ***pass***.

Набор тестов по-английски называется *test suite*. Когда говорят, что нужно сгенерировать по спецификации тесты, имеется в виду набор тестов.

Набор тестов *значимый* – по-английски *sound*, если любая реализация, соответствующая спецификации, проходит каждый тест из набора. Именно для значимых наборов верно правило: найденная ошибка таковой является, а если ошибка не найдена, то неизвестно, есть она или нет. Пример: проверяется, что квадратный корень из 4 равен ± 2; понятно, что отсюда не следует, что правильно вычисляется квадратный корень из 5.

Набор тестов *исчерпывающий* – по-английски *exhaustive*, если, наоборот, любая реализация, которая проходит каждый тест из набора, соответствует спецификации. Исчерпывающие тесты более экзотичны: они могут найти ошибку в правильной реализации, но зато, если ошибка не найдена, то её точно нет. Пример: проверяется, что квадратный корень вычисляется правильно

и всегда положителен; в правильной реализации, возвращающей отрицательное значение, будет обнаружена ошибка.

Наконец, набор тестов полный – по-английски *complete*, если он значимый и исчерпывающий, то есть та, и только та реализация соответствует спецификации, которая проходит каждый тест из набора.

Главная задача тестирования соответствия – генерация по спецификации полного тестового набора.

К сожалению, в такой постановке почти всегда любое решение этой задачи практически непригодно, хотя в теории может быть простым и безукоризненным.

ПРОБЛЕМЫ: Бесконечность полного тестового набора.

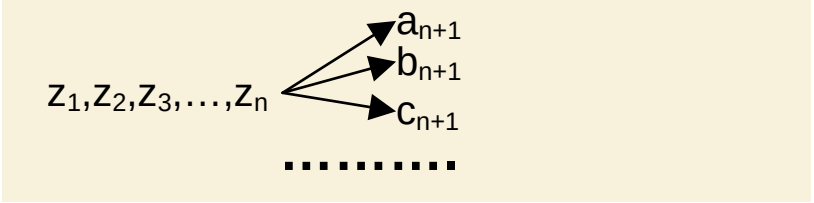
Дело в том, что почти всегда полный тестовый набор оказывается бесконечным. Причина – в бесконечном числе трасс реализации, которые должны быть протестированы для проверки соответствия спецификации. Реализации, в которых число трасс конечно, называются реализациями с конечным поведением. Но таких реализаций на практике очень мало.

В общем случае трассы реализации делятся на две группы: безразличные трассы и значимые трассы.

Наличие или отсутствие безразличных трасс не влияет на соответствие. Примером может служить функция вычисления квадратного корня, реализованная как метод класса. Тестируемая реализация – это объект этого класса. Спецификация требует, чтобы объект содержал метод вычисления квадратного корня, и чтобы этот метод работал правильно. Но она безразлична к наличию или отсутствию других методов в объекте, которые, естественно, порождают другие – безразличные для данной спецификации трассы.

Поэтому, говоря о бесконечности трасс реализации, имеются в виду только значимые трассы.

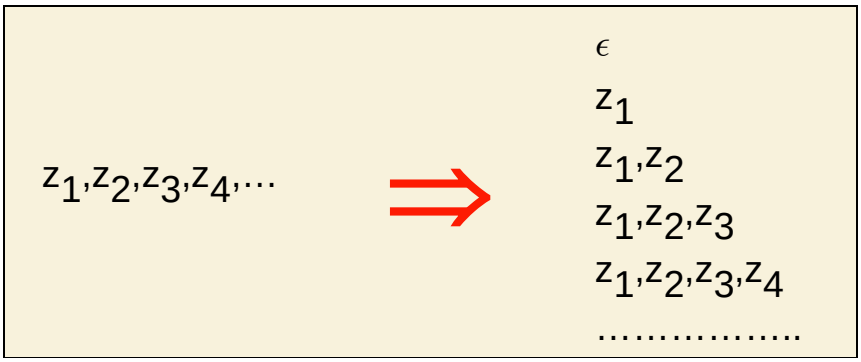
Можно выделить две причины бесконечности множества значимых трасс: бесконечное ветвление множества трасс и бесконечные трассы.



Бесконечное ветвление – это когда трасса $z_1, z_2, \ldots z_n$ может продолжаться бесконечным числом $n+1$-ых наблюдений $a_{n+1}, b_{n+1}, c_{n+1}, \ldots$. Понятно, что бесконечное ветвление предполагает бесконечный алфавит наблюдений. Примером служит алфавит наблюдений для функции вычисления квадратного корня: прежде всего, бесконечное число значений аргумента. Можно, конечно, ограничиться конечным подмножеством, определяемым разрядкой сеткой компьютера. Но, во-первых, это всё равно слишком много, то есть практически бесконечно, а, во-вторых, это ограничение, зависящее от архитектуры машины и даже от языковой поддержки, которая может включать библиотечные программы работы с числами большей разрядности.

Тем самым, это ограничение не функционально. Между прочим, эти соображения – главная причина, по которой в языках спецификации не должно быть подобных ограничений.

Можно отметить, что при бесконечном алфавите наблюдений ветвление может оказаться и конечным. Например, реализация – это очередь, в которой хранятся булевские значения, и которая имеет три операции: «`поместить в начало очереди указанное значение`», «`удалить значение из конца очереди и вернуть его`» и «`сообщить длину очереди`». Ограничимся наблюдениями только внешних действий, то есть операций над очередью с их прямыми и обратными параметрами. Если очередь не ограничена, то алфавит наблюдений бесконечен: операция «`сообщить длину очереди`» может вернуть любое натуральное число или `0`. Однако, ветвление конечно: в каждый момент времени, то есть после каждой трассы, имеется не более четырёх наблюдений: два наблюдения помещения в очередь **`true`** или **`false`**, одно наблюдение удаления из очереди (если она не пуста) и одно наблюдение длины очереди.



Бесконечная трасса также порождают бесконечное множество конечных трасс (своих начальных отрезков) даже для конечного алфавита наблюдений и, следовательно, конечного ветвления. Если в том же примере с очередью удалить операцию опроса длины очереди или считать очередь ограниченной длины, алфавит наблюдений становится конечным. Однако здесь есть бесконечные трассы: мы можем бесконечно то помещать в очередь, то удалять из неё.

Забегая вперёд, скажу, что бесконечные трассы существуют даже в конечных системах, то есть системах с конечным алфавитом и конечным числом состояний. Достаточно иметь хотя бы один цикл. Тем самым, конечное поведение имеют только реализации с конечным алфавитом, конечным числом состояний и без циклов. Большинство практических реализаций, конечно, не такое.

ПРОБЛЕМЫ: Выбор значимого тестового набора.

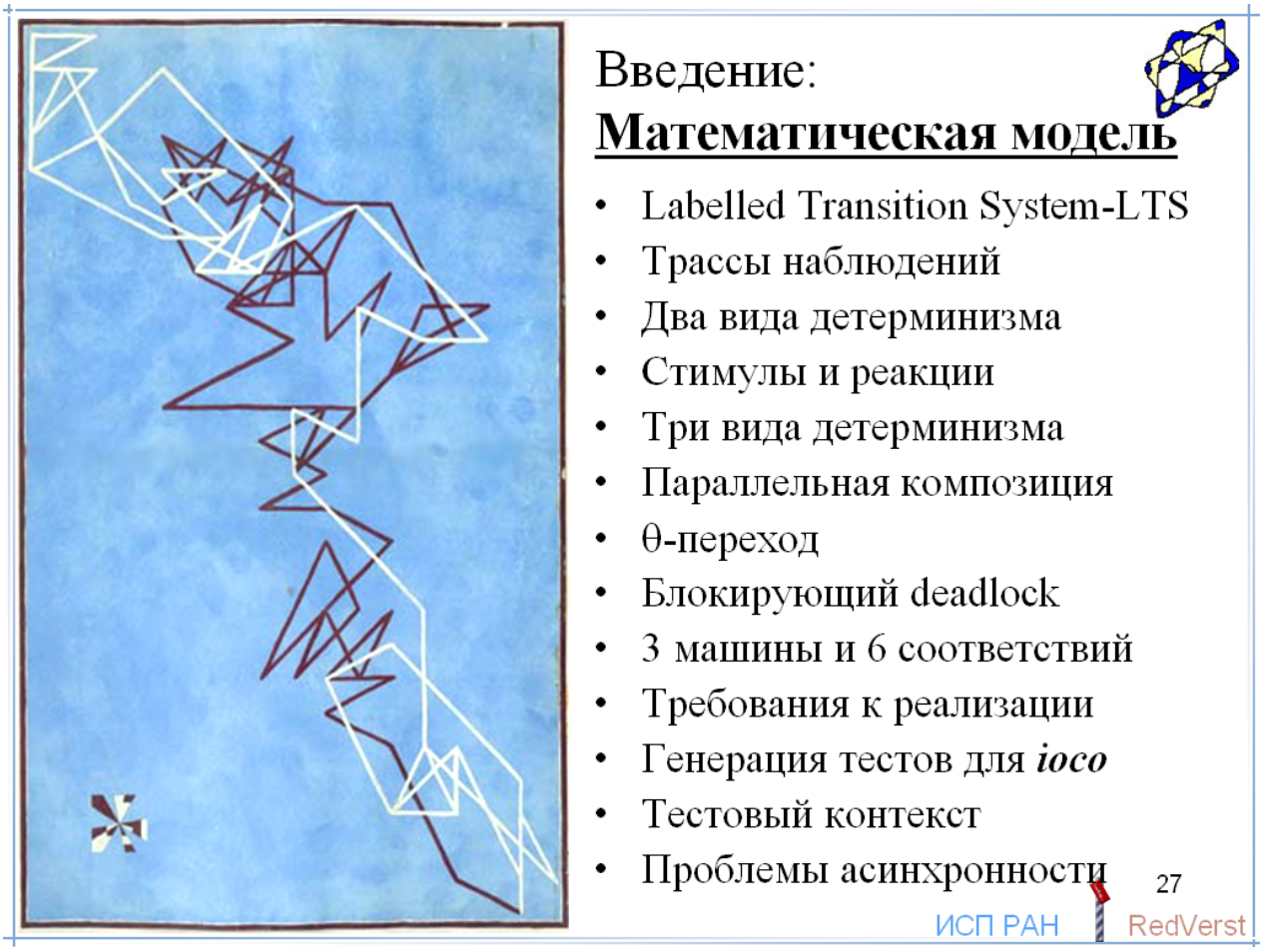
Таким образом, на практике почти всегда приходится ограничиваться тестовыми наборами, которые только значимы, но не полны.

Тогда возникают вопросы: какие тестовые наборы «достаточно хорошие»? каковы критерии «достаточно хорошего» набора?

Универсальный подход к решению этой проблемы – это введение эквивалентности значимых трасс и соответствующая факторизация спецификации. Для каждого класса эквивалентности мы должны проверить хотя бы одну трассу. Такая эквивалентность, естественно, предполагает соответствующую реализационную гипотезу: если одна трасса из класса правильная, то и все

остальные правильные. Если число классов эквивалентности оказывается конечным, то полный тестовый набор для фактор-спецификации тоже конечен. Разумеется, реализационная гипотеза остаётся гипотезой, быть может, и не верной. Поэтому, фактически, полученный конечный тестовый набор для исходной спецификации остаётся только значимым. Но, по крайней мере, мы понимаем, что делаем: эквивалентность подбирается из некоторых разумных предположений о том, как может быть устроена реализация. Тем самым, мы надеемся, что вероятность необнаружения ошибки достаточно мала.

Базовый способ построения конечного значимого тестового набора заключается в следующем. Пусть полный тестовый набор перечислим и у нас есть алгоритм его перечисления. Пусть также у нас есть критерий, по которому мы определяем, годится ли данный конечный тестовый набор или нет. Будем предполагать, что в полном тестовом наборе существует конечное подмножество, удовлетворяющее нашему критерию. Тогда, используя алгоритм перечисления, выбираем тесты один за другим и проверяем наш критерий на всех подмножествах уже выбранных тестов до тех пор, пока не получим искомый тестовый набор. Конечно, критерий может быть устроен лучше: так, что нам не нужно перебирать все такие подмножества, а только проверять, добавлять ли следующий тест в набор или нет, то есть проводить фильтрацию перечисляемых тестов.



Labelled Transition System (LTS).

Итак, мы отметили три основные проблемы тестирования: дивергенцию, недетерминизм и полнота тестового набора. Мы намеренно рассказывали об этих проблемах до введения математической модели, опираясь только на формализацию эксперимента с помощью машины тестирования. Дело в том, что существует различные модели, согласующиеся с таким пониманием тестового эксперимента. И эти три проблемы – общие для всех моделей.

А теперь рассмотрим одну модель, которая, во-первых, адекватна машине тестирования и, во-вторых, является наиболее распространённой в тестировании соответствия. Это даст нам возможность более конкретно говорить о соответствии реализации и спецификации. Сразу оговорюсь, что эта модель тоже имеет множество разновидностей, разного рода усложнений, улучшений, оптимизаций и т.п. Однако все эти формы не меняют существа дела и, фактически, представляют собой оптимизированные для разных случаев формы сокращённой записи.

Модель, о которой идёт речь, – это, так называемая, система помеченных переходов или, по-английски, Labelled Transition System, сокращённо LTS.

Формально LTS определяется как четвёрка **S**=(V,C,E,s_0).

Она представляет собой совокупность непустого множества V состояний, одно из которых выделено как начальное s_0, и множества E помеченных переходов.

Каждый переход ведёт из одного состояния, которое можно называть пресостоянием, в другое состояние, которое можно называть постсостоянием. Если постсостояние совпадает с пресостоянием, такой переход называется петлёй. Вообще, здесь широко используется представление LTS в виде графа переходов и соответствующая терминология. Переход помечен либо одним из символов внешних действий, либо символом τ. Алфавит внешних действий C также считается заданным как часть LTS.

Внешнему действию z соответствует переход, помеченный символом z; это внешний или наблюдаемый переход: $s \text{—} z \rightarrow s` = (s,z,s`) \in E$ из состояния s в состояние s` по внешнему действию $z \in C$.

Внутренней активности соответствует переход, помеченный символом τ; это внутренний, ненаблюдаемый переход: $s \text{—} \tau \rightarrow s` = (s,\tau,s`) \in E$ из состояния s в состояние s` по внутреннему действию τ.

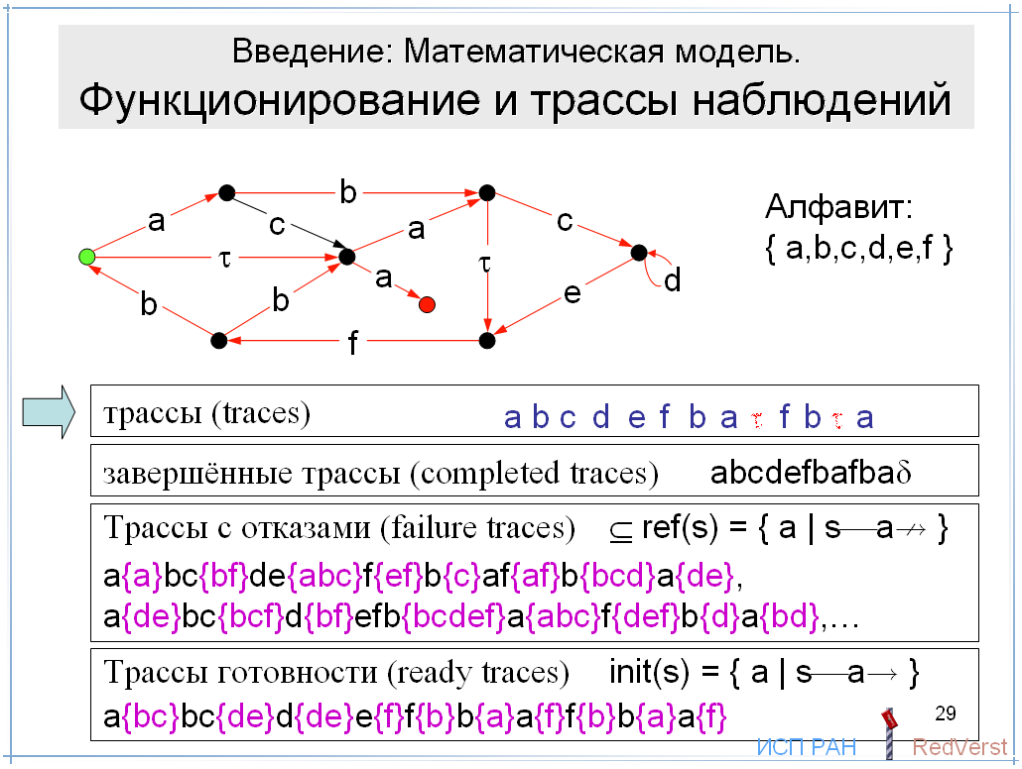
Внутренняя активность, тем самым, это цепочка τ-переходов. Дивергенция – бесконечная цепочка, в частности, порождаемая циклом, τ-переходов.

Состояние дивергентно, если в нём начинается дивергенция, то есть бесконечная цепочка τ-переходов. В противном случае, состояние конвергентно.

Стабильное состояние – это состояние, из которого не ведут τ-переходы: $s \text{—} \tau \not\rightarrow$. Очевидно, стабильное состояние конвергентно.

Функционирование и трассы наблюдений.

На рисунке показан пример LTS, где есть внутренние переходы, нестабильные и стабильные состояния, начальное состояние.



Функционирование LTS заключается в том, что она совершает последовательность смежных переходов: пресостояние следующего перехода совпадает с постсостоянием предыдущего перехода. Пресостояние первого перехода – это начальное состояние.

Простейшая трасса – трасса внешних действий – строится как раскраска маршрута, то есть цепочки переходов, символами внешних действий. Эта раскраска получается из полной раскраски маршрута, то есть последовательности символов, которыми помечены его переходы, удалением вхождений символа τ.

Завершённая трасса получается, если при попадании в терминальное состояние добавлять символ δ.

Отказ выполнения внешнего действия `a` происходит в стабильном состоянии, из которого не выходит переход, помеченный символом `a`. Для стабильного состояния `s` через **`ref`**`(s)` обозначим множество всех таких символов `a`, по которым нет переходов из `s`. Наблюдаемое множество отказов – это любое подмножество **`ref`**`(s)`.

Трасса с отказами, *failure trace*, получается, если при прохождении каждого состояния `s` добавлять любую конечную последовательность множеств отказов, вложенных в **`ref`**`(s)`: $A_1, A_2, \ldots \subseteq$ **`ref`**`(s)`. На рисунке приведён пример двух таких трасс.

Ещё одной разновидностью трасс являются, так называемые, *ready-traces*. Они похожи на трассы с отказами, но только, во-первых, указывается не множество отвергаемых действий **`ref`**`(s)`, а множество действий, которые могут быть выполнены в данном состоянии **`init`**`(s)` , то есть дополнение **`ref`**`(s)` до всего алфавита действий, и, во-вторых, указывается всё множество целиком, а не его подмножество. Такие трассы полезны, например, для графических интерфейсов, где множество действий, которые могут быть выполнены, – это набор кнопок графического интерфейса, высвеченных на экране в данный момент времени.

Обозначения для трасс наблюдений.

Для трасс наблюдений обычно используются следующие обозначения:
Переход по пустой трассе:

$a = \epsilon \Rightarrow b \quad =_{def} \quad a=b$ или $a \longrightarrow \tau \rightarrow \ldots \longrightarrow \tau \rightarrow b$

♣*Переход по трассе из одного символа* z:

$a = z \Rightarrow b \quad =_{def} \quad a = \epsilon \Rightarrow \longrightarrow z = \rightarrow \epsilon \Rightarrow b$

♣*Переход по произвольной трассе* $\sigma = z1,z2,\ldots,zn$:

$a = \sigma \Rightarrow b \quad =_{def} \quad a = z1 = \Rightarrow z2 \Rightarrow \ldots = zn \Rightarrow b$

♣*Нет перехода из* a *в* b *по трассе* σ:

$a = \sigma \not\Rightarrow b \quad =_{def} \quad \neg(a = \sigma \Rightarrow b)$

♣*Переход из* a *по трассе* σ:

$a = \sigma \Rightarrow \quad =_{def} \quad \exists b \; a = \sigma \Rightarrow b$

♣*Нет перехода из* a *по трассе* σ:

$a = \sigma \not\Rightarrow \quad =_{def} \quad \neg(a = \sigma \Rightarrow)$

♣*Множество состояний после трассы* σ*, начинающейся в состоянии* a*:*

a ***after*** σ $=_{def}$ { b | a=σ⇒b }

Для LTS **S** с начальным состоянием s_0:

S ***after*** σ $=_{def}$ s_0 ***after*** σ

Машина тестирования и два вида детерминизма.

LTS *детерминирована*, если пресостояние и символ действия однозначно определяют постсостояние, и, кроме того, нет τ-переходов. Это определение эквивалентно тому, что каждая трасса LTS заканчивается ровно в одном состоянии.

Напомним, что реализацию мы называли наблюдаемо детерминированной, если наблюдаемая трасса не зависела от погодных условий, то есть однозначно определялась последовательностью нажатия кнопок оператором. Здесь имеется в виду, что на каждой кнопке написано ровно одно действие и каждое действие из алфавита действий написано на некоторой кнопке. В терминах LTS наблюдаемый детерминизм означает, что в каждом состоянии, в котором заканчивается трасса, определены одни и те же действия.

Очевидно, что детерминизм LTS влечёт наблюдаемый детерминизм. Обратное, вообще говоря, не верно: реализация может быть наблюдаемо детерминирована, но LTS-недетерминирована.

Недетерминированной реализацией будем называть реализацию, которая не является LTS-детерминированной.

Стимулы и реакции.

Теперь мы будем рассматривать частный, но широко распространённый и самый важный тип LTS. Этот тип LTS предназначен для моделирования такого взаимодействия между реализацией и окружением, которое сводится к обмену информацией между ними.

Внешние действия разбиваются на две группы: стимулы или ввод, и реакции или вывод. Стимул – это передача информации или сообщения из окружения в реализацию, а реакция – это передача в обратном направлении, из реализации в окружение. Можно сказать, что реализация и окружения зеркально симметричны друг другу: реализация принимает стимул, который окружение выдаёт; реализация выдаёт реакцию, которую окружение принимает. При тестировании, когда тест подменяет собой окружение, стимул – это тестовое воздействие на реализацию, а реакция – это, условно говоря, ответное действие реализации. Условно, потому что мы не предполагаем, что на один стимул непременно должна быть реакция, что эта реакция единственная; более того, реакция может поступить независимо от подачи стимулов. Системы, работающие по принципу «один стимул – одна реакция», являются особым частным случаем LTS. Это хорошо известные и появившиеся гораздо раньше автоматы Мили, о которых мы поговорим позже.

Мы будем использовать нотацию алгебры процессов CCS – Calculus of Communicating Systems [R.Milner]: в качестве префикса для стимулов используется вопросительный знак "?", а для реакций – восклицательный знак "!". Для каждого действия определяется противоположное ему с помощью биективного отображения «подчёркивание»: `?x = !x, !y = ?y`.

Такие LTS имеют много разных названий:

IOA - Input-Output Automaton,
IOLTS - Input-Output Labelled Transition System,
IOTS - Input-Output Transition System,
IOSM - Input-Output State Machine,
IOFSM - Input-Output Finite State Machine,
AA - Асинхронный автомат.

Мы будем использовать название «асинхронный автомат».

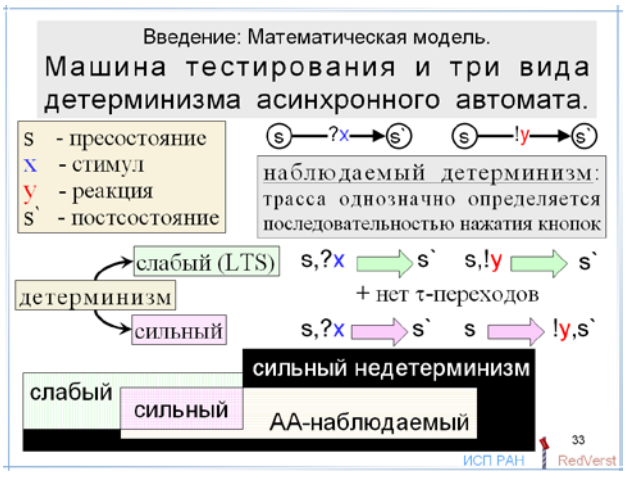
Машина тестирования и три вида детерминизма асинхронного автомата.

Машина тестирования асинхронного автомата имеет одну кнопку для каждого стимула и общую кнопку для всех реакций.

При нажатой кнопке стимула на дисплее может высветиться только этот стимул.

При нажатии на кнопку всех реакций на дисплее может высветиться реакция, но, вообще говоря, эта реакция определяется неоднозначно: при одном нажатии кнопки высвечивается одна реакция, а при другом – другая, даже если эти нажатия выполняются в одной и той же ситуации.

Асинхронный автомат, как и общая LTS, наблюдаемо детерминирован, если трасса однозначно определяется последовательностью нажатия кнопок. Для стимулов это означает, что предшествующая трасса однозначно определяет, будет или не будет высвечен на дисплее данный стимул при нажатии кнопки этого стимула. Для реакций предшествующая трасса однозначно определяет не только то, будет или нет высвечена на дисплее какая-нибудь реакция при нажатии кнопки приёма всех реакций, но и саму эту реакцию.



Кроме наблюдаемого детерминизма, для асинхронных автоматов можно рассматривать ещё два вида детерминизма: слабый и сильный.

В обоих случаях пресостояние и стимул, определённый в этом состоянии, однозначно определяют постсостояние. Иными словами, в каждом состоянии определено не более одного перехода по каждому стимулу.

При слабом детерминизме пресостояние и реакция, определённая в этом состоянии, однозначно определяют постсостояние. Иными словами, в каждом состоянии определено не более одного перехода по каждой реакции.

При сильном детерминизме в каждом состоянии определено не более одного перехода по всем реакциям, то есть либо ни одного перехода по реакциям, либо только один переход по одной реакции.

В обоих случаях не должно быть внутренних переходов.

Очевидно, что слабый детерминизм – это то, что мы называли детерминизмом LTS общего вида.

Сильный детерминизм влечёт слабый детерминизм.

Кроме этого, сильный детерминизм влечёт наблюдаемый детерминизм. Обратное, вообще говоря, не верно: реализация может быть наблюдаемо, но не сильно детерминирована. Если реализация слабо, но не сильно, детерминирована, то она наблюдаемо недетерминирована: нажатие кнопки приёма всех реакций неоднозначно определяет получаемую реакцию.

Важно отметить, что наблюдаемый детерминизм для асинхронных автоматов, – это не то же самое, что наблюдаемый детерминизм для LTS общего вида. Различие определяется разными машинами тестирования: в LTS общего вида для каждой реакции своя кнопка, а для асинхронных автоматов – одна кнопка на все реакции. Поэтому, в частности, хотя слабый детерминизм асинхронного автомата совпадает с LTS-детерминизмом, этот детерминизм влечёт наблюдаемый детерминизм LTS, но, вообще говоря, не влечёт наблюдаемый детерминизм асинхронных автоматов. Более того, если реализация слабо детерминирована, но не сильно детерминирована, то она наблюдаемо детерминирована как LTS, но не является наблюдаемо детерминированной как асинхронный автомат.

Недетерминированной реализацией будем называть реализацию, которая не является слабо детерминированной. Такой автомат может быть наблюдаемо детерминированным, то есть пара пресостояние и стимул однозначно определяют реакцию, но не однозначно определяют постсостояние. Здесь важно, что даже тройка пресостояние, стимул и реакция неоднозначно определяют постсостояние.

Асинхронный автомат: Параллельная композиция.

Теперь мы будем считать, что реализация и тест моделируются асинхронными автоматами. Для краткости мы будем говорить просто реализация и тест, имея в виду их модели в виде асинхронных автоматов. Алфавиты реализации и теста взаимно обратны: реализация принимает стимулы, а тест их выдаёт; реализация выдаёт реакции, а тест их принимает. Можно сказать, что стимулы реализации являются реакциями теста и, наоборот, реакции реализации являются стимулами теста.

Теперь нам нужно математически описать взаимодействие реализации и теста. Это взаимодействие рассматривается как параллельное частично синхронизованное выполнение двух асинхронных автоматов.

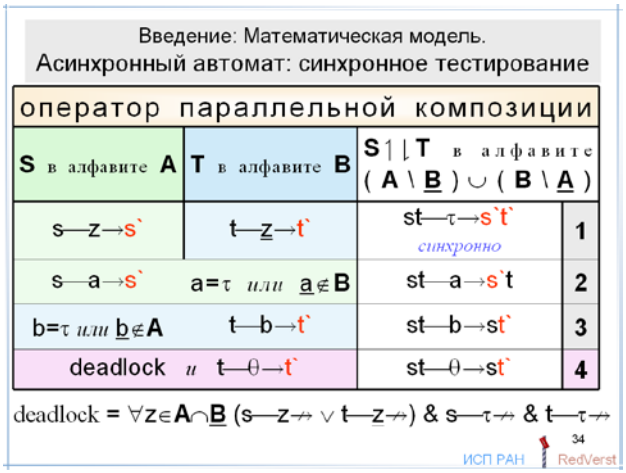
Синхронизация возникает на передаче стимулов и реакций. Переход в одном автомате по приёму символа происходит синхронно с переходом в другом автомате по выдаче этого же символа. Оба автомата одновременно меняют свои состояния.

Внутренние переходы выполняются асинхронно, изменяется состояние только того автомата, в котором совершается внутренний переход.

Если автоматы имеют противоположные, то есть взаимно-обратные по подчёркиванию, алфавиты, то этого достаточно.

В общем же случае, если в одном автомате есть переход по действию, для которого в алфавите другого автомата нет противоположного действия, то такой переход совершается асинхронно аналогично внутреннему переходу.

Если для текущих состояний двух автоматов возможно несколько вариантов дальнейшего поведения, то выбор делается недетерминированным образом некоторым, как говорят, «мистическим» синхронизатором.



Такое взаимодействие описывается с помощью оператора параллельной композиции. Формально, оператор строит новый асинхронный автомат, состояние которого – это пара состояний автоматов-операндов, а все переходы – это внутренние переходы, соответствующие синхронной паре переходов в обоих автоматах, или одному асинхронному переходу в одном автомате.

Если один автомат – это реализация, а другой автомат – это тест, то они имеют взаимно-обратные алфавиты только в том случае, когда тест подменяет собой всё окружение. Тогда все переходы композиции будут внутренними. В противном случае в композиции остаются переходы по стимулам и реакциям реализации, через которые реализация может взаимодействовать с остальной частью окружения. Если в реальном эксперименте эта остальная часть окружения отсутствует или не взаимодействует с реализацией, то нам безразличны эти композиционные переходы по стимулам и реакциям.

Однако если это не так, то для получения полной картины мы должны были бы сначала скомпоновать реализацию с остальной частью окружения, а потом результат такой композиции скомпоновать с тестом. Некоторые или все переходы по стимулам и реакциям реализации становятся внутренними уже после первой компоновки. Из-за этого результат тестирования как раз и становится зависящим от такого взаимодействия реализации с остальной частью окружения. Иными словами, мы начинаем тестировать не реализацию, а её композицию с остальной частью окружения. Проблема состоит в том, что мы на самом деле часто ничего не знаем об этой остальной части, поскольку спецификация – это требования только к реализации, а не к этой остальной части окружения. В результате тестирование на основе такой спецификации может стать недостоверным. Это нужно учитывать.

Пока будем считать, что тест подменяет собой всё окружение. Для выдачи вердикта терминальные состояния теста помечены как ***pass*** или как ***fail***.

При взаимодействии реализации и теста может возникнуть deadlock: стимулы, которые реализация готова принимать, тест не выдаёт, а реакции, которые реализация готова выдавать,

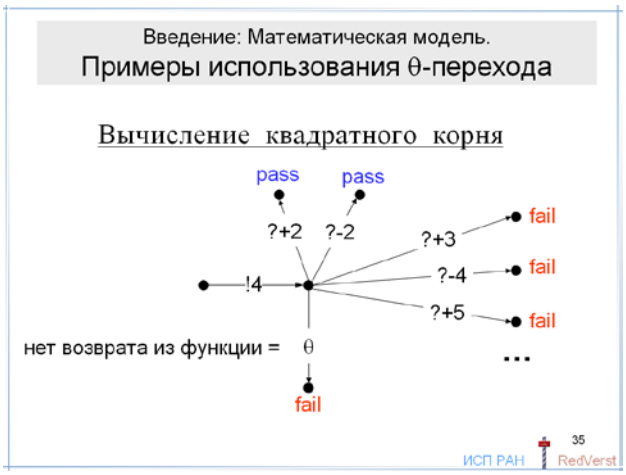
тест не принимает. Для разрешения deadlock`а в тесте вводится, так называемый, θ-переход. Его можно интерпретировать как переход по тайм-ауту в том случае, когда внешние переходы и любые цепочки τ-переходов в реализации ограничены сверху по времени. В частности, реализация должна быть конвергентна. Если ограничены по времени только действия (внешние и внутренние), то истечение тайм-аута означает либо дивергенцию, либо слишком длинную цепочку τ-переходов, либо deadlock. В дальнейшем мы будем рассматривать только конвергентные реализации с ограниченными по времени внешними переходами и цепочками τ-переходов.

Такое тестирование называют *синхронным*. Приведём несколько примеров, показывающих смысл θ-перехода в реальном тестировании.

Примеры использования θ-перехода. Вычисление квадратного корня.

Для функции вычисления квадратного корня после передачи стимула-аргумента `4` в тесте задаётся приём двух результатов `+2` и `-2`, а также θ-переход, играющий роль перехода по умолчанию **`default`** в операторе выбора типа **`switch`**.

В этом случае мы предполагаем, что не только любой результат, отличный от `+2` и `-2`, но и отсутствие результата (нет возврата из функции) является ошибкой: θ-переход ведёт в ***`fail`***-состояние.



Заметим, что объединение в одном θ-переходе различных реакций и отсутствия реакций является оптимизацией. Точно так же мы могли бы определить переход по каждому возвращаемому значению и отдельно θ-переход по отсутствию реакций; все эти переходы вели бы в ***`fail`***-состояние.

Примеры использования θ-перехода. Передача пакета с ошибочным заголовком.

Обратный пример, когда отсутствие реакции не является ошибкой, встречается в сетевых протоколах.

Если тест посылает неправильное сообщение-стимул, например, пакет с неправильным заголовком (с возможной ошибкой в адресе отправителя), то реализация не может послать никакого ответного сообщения-реакции, поскольку не знает, кому посылать.

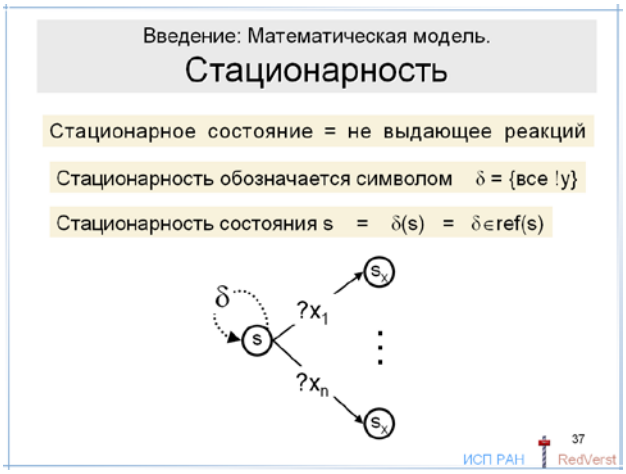
В этом случае θ-переход по отсутствию реакции либо продолжает тест, либо заканчивает с

вердиктом ***pass***, а получение любой реакции на этот стимул является ошибкой.

Стационарность.

В этих примерах θ-переход делается в таком состоянии теста, в котором он только принимает реакции. Если эти примеры не оптимизировать, можно считать, что в том состоянии теста, в котором он принимает реакции, он принимает все реакции, а θ-переход в этом состоянии предназначен для обнаружения отсутствия реакций.

Состояние реализации, в котором она не выдаёт ни одной реакции, называется *стационарным*. Таким образом, по крайней мере в этих примерах, можно считать, что θ-переход предназначен для наблюдения стационарности.

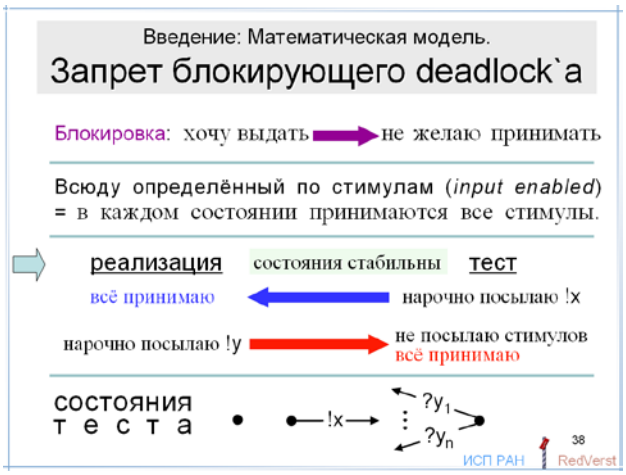


Стационарность обозначается символом δ. Его можно интерпретировать как множество всех реакций, принадлежащее refusal set`у состояния. Теперь мы можем рассматривать трассы, в которых встречается символ стационарности. Для того чтобы получить такие трассы в автомате достаточно в стационарных состояниях добавить петлю, помеченную символом δ.

Запрет блокирующего deadlock`а.

Сейчас мы будем рассматривать взаимодействие автоматов со следующим важным ограничением: deadlock может возникнуть только в том случае, когда обе стороны, и реализация и тест, ждут приёма сообщений. Если хотя бы на одной стороне инициирована выдача сообщения, то deadlock`а быть не должно. Это означает, что при передаче сообщений активной стороной, инициирующей акт передачи, является передающая сторона, а принимающая сторона – пассивна и она должна соглашаться принять сообщение. Иными словами, мы вводим запрет на блокировку передачи принимающей стороной. Посмотрим, какие ограничения нужно наложить на класс всех реализаций и всех тестов, чтобы при параллельной композиции любой реализации и любого теста не возникало блокирующего deadlock`а.

Очевидно, запрет блокирующего deadlock`а будет выполнен, если в обоих автоматах в каждом состоянии определён приём каждого стимула (в тесте, соответственно, реакции). Такие автоматы называются всюду определёнными по стимулам (по-английски, input enabled).



Всюду определённость по стимулам – сильное требование; для наших целей его можно ослабить.

Действительно, deadlock, очевидно, возникает только в стабильных состояниях реализации и теста. Deadlock с блокировкой приёма стимула реализацией может возникнуть тогда, когда реализация в стабильном состоянии принимает не все стимулы. Мы всегда можем подобрать такой тест, соответствующее состояние которого стабильно и в нём определён единственный переход по выдаче не принимаемого реализацией стимула. Поэтому потребуем, чтобы реализация принимала все стимулы во всех своих стабильных состояниях.

Тогда может быть только deadlock с блокировкой приёма реакции тестом. Соответствующее состояние теста стабильно и в нём не могут выдаваться стимулы, так как в противном случае реализация приняла бы стимул и deadlock`а бы не было. Если тест в этом состоянии принимает не все реакции, то всегда можно подобрать такую реализацию, которая в соответствующем состоянии принимает все стимулы и выдаёт единственную не принимаемую тестом реакцию, возникает deadlock с блокировкой приёма реакции тестом. Следовательно, если тест в стабильном состоянии не выдаёт стимулов, то он должен принимать все реакции. Исключение, конечно, составляют терминальные состояния, в которых тест заканчивает свою работу с вынесением вердикта.

Итак, вместо требования всюду определённости мы можем сформулировать более слабое требование: реализация принимает все стимулы в каждом стабильном состоянии, а тест в каждом стабильном состоянии либо не принимает реакций, либо принимает все реакции.

Для того чтобы избежать ненужного недетерминизма при тестировании, мы будем считать, что в тесте нет τ-переходов и нет выдачи нескольких стимулов из одного состояния.
Таким образом, все состояния теста стабильны и делятся на три типа:
1) терминальное состояние, в котором тестирование заканчивается с вынесением вердикта;
2) посылающее состояние, которое выдаёт один стимул и не принимает никаких реакций;
3) принимающее состояние, которое принимает все реакции и не выдаёт никаких стимулов.

Теперь, если не считать терминальных состояний теста, deadlock может возникнуть только в стационарном состоянии реализации, когда тест находится в принимающем (то есть, тоже стационарном) состоянии. Наблюдение deadlock`а с помощью θ-перехода оказывается наблюдением стационарности, которую мы тоже будем обозначать символом δ.

Заметим, что эти требования к реализации и тесту никак не влияют на спецификацию: в каждом её состоянии стимул может быть, как определён, так и не определён без каких-либо ограничений.

Три машины тестирования.

Теперь, с учётом запрета блокирующего deadlock`а, рассмотрим работу машины тестирования. В зависимости от наших тестовых возможностей у нас могут быть три такие машины. В каждой из этих машин нет зелёной лампочки. Для передачи в реализацию каждого стимула имеется отдельная кнопка, а для получения реакций от реализации имеется одна кнопка – принять все реакции.

Поскольку блокирующий deadlock запрещён, каждая из этих машин при нажатии кнопки стимула обязана высветить на дисплее этот стимул. Поэтому в кнопку стимула не встроен тайм-аут.

Если нажимается кнопка приёма реакций, и реализация выдаёт реакцию, то во всех машинах эта реакция высвечивается на дисплее.

Различия начинаются тогда, когда реакций нет в течение какого-то промежутка времени.

В машине **A** при отсутствии реакций ничего не происходит. Кнопка остаётся нажатой. Никакого тайм-аута у нас нет, и мы в любой момент после ожидания реакции можем закончить тестирование. Однако такое окончание не означает, что дальше нет реакций: может быть, их нет, а может быть, мы их просто не дождались.

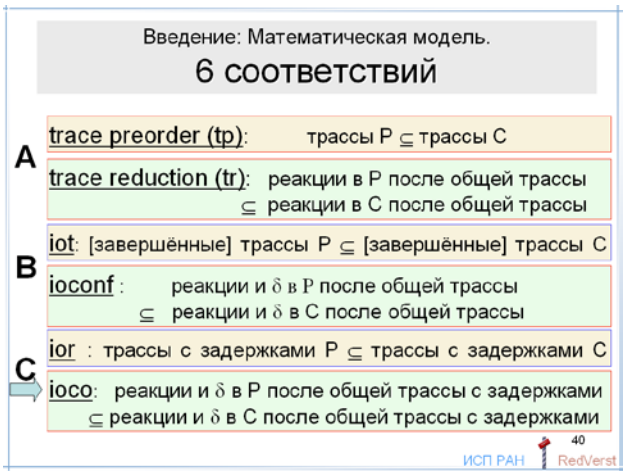
В машине **B** имеется тайм-аут для ожидания реакций, и по истечении тайм-аута на экран выводится символ θ, а кнопка приёма остаётся навечно нажатой. Тестирование заканчивается, а полученная трасса завершается символом стационарности.
Символ θ трактуется как наблюдаемая стационарность.

В машине **C** также по истечении тайм-аута на экран выводится символ θ, но кнопка приёма отжимается, и можно нажимать ту же самую кнопку или любую другую кнопку.
Символ θ трактуется как наблюдаемая стационарность.

Шесть соответствий.

Эти три машины тестирования описывают тестовые возможности для трёх пар соответствий. Два соответствия одной пары отличаются тем, проверяются ли в процессе экспериментов все возможные трассы или только те, что есть в спецификации.

Машине **A** соответствуют трассы наблюдений, состоящие только из наблюдаемых действий – последовательности символов внешних действий. Такие трассы наблюдений традиционно называют просто трассами. На трассах основано простейшее соответствие, которое называется трассовым предпорядком, *trace preorder*, сокращённо **`tp`**, и которое определяется просто как вложенность множества трасс реализации во множество трасс спецификации. Соответствующая эквивалентность, то есть равенство трасс, называется трассовой эквивалентностью.



Трассовый предпорядок фиксирует простую, но важную идею: трасса, которую можно наблюдать в эксперименте над реализацией, можно наблюдать в эксперименте над спецификацией. Обратное не требуется, потому что спецификация описывает реализацию «с избытком», предлагая альтернативы и оставляя выбор за разработчиком по принципу

may&must. Например, реализация квадратного корня имеет право при аргументе `4` возвращать только `+2`, хотя в спецификации написана дизъюнкция: `+2` или `-2`. Было бы странно требовать от реализации обязательного возвращения и `+2` и `-2`, в зависимости от погоды.

Это соответствие имеет очевидный недостаток: оно требует, чтобы поведение реализации после приёма *любого* стимула было таким же, как в спецификации. Однако спецификация не обязана быть всюду определённой по стимулам. Если в некотором состоянии спецификации не определён приём стимула, то обычно это трактуется как отсутствие требований к реализации: она не обязана принимать этот стимул, а если принимает, то нам не важно, какое будет дальнейшее поведение. Поскольку наши реализации всюду определённые, получается, что, какое бы ни было поведение реализации, начиная с приёма этого стимула, такого поведения нет в спецификации. Тем самым, мы зафиксируем ложную ошибку.

Решение этой проблемы достаточно просто: нам не нужно при тестировании подавать такой стимул в реализацию и, тем самым, наблюдать и проверять трассы, начинающиеся с этого стимула.

Соответствие, которое можно назвать *трассовой сводимостью, trace reduction*, сокращённо **`tr`**, по аналогии с соответствием *reduction* для автоматов Мили, определяется следующим правилом:

> Если трасса может наблюдаться как в реализации, так и в спецификации, то после этой трассы реализация может ... тогда и только тогда, когда это возможно в спецификации после этой же трассы. Под ... здесь понимается выдача некоторой реакции.

Это соответствие также всё ещё неудовлетворительно, но по другой причине: реализация, которая только принимает все стимулы, но не выдаёт ни одной реакции, оказывается конформной любой спецификации – по принципу «кто ничего не делает, тот не ошибается». Это не совсем то, что мы хотели бы от реализации. Поэтому в следующем соответствии предпринимается попытка запретить такое «ничегонеделание».

Машина **B.** Добавляется ещё один тип наблюдения – наблюдение стационарности. Кроме обычных трасс, рассматриваются также завершённые трассы. При наших ограничениях, это трассы, заканчивающиеся символом стационарности δ. Теперь мы уже можем запретить ничегонеделание. Например, если в спецификации после стимула `?4` определены реакции `!+2` и `!-2`, то реализация уже не может ничего не делать, то есть не возвращать реакцию, поскольку такое поведение порождало бы трассу `?4`δ, которой нет в спецификации.

Соответствующее отношение реализации и спецификации называется *input-output testing relation*, сокращённо **`iot`**. Другое название: трассовый предпорядок со стационарностью, *quiescent trace preorder*. Оно определяется как вложенность не только множества трасс, но и множества завершённых трасс реализации в соответствующие множества трасс спецификации.

Для борьбы с ложными ошибками на завершённых трассах, аналогично трассовому предпорядку и трассовой сводимости, вводится соответствие, которое называется *input-output conformance*, сокращённо **`ioconf`**. Оно определяется следующим правилом:

> Если трасса может наблюдаться как в реализации, так и в спецификации, то после этой трассы реализация может ... тогда и только тогда, когда это возможно в спецификации после этой же трассы. Под ... понимается одно из двух:
> 1) выдать некоторую реакцию `!y`;
> 2) не выдавать никаких реакций, то есть находиться в стационарном состоянии.

Заметим, что после стационарности мы получаем уже завершённую трассу в тесте, которая для этого соответствия, также как для **iot**, считается терминальной, то есть, не имеет никаких продолжений. Иными словами, тест по θ-переходу, то есть по тайм-ауту, заканчивает свою работу с вынесением того или иного вердикта. Так и нужно делать, если мы подозреваем, что тайм-аут может истечь не только по причине стационарности, но и вследствие дивергенции, после которой нет продолжения.

Машина **C**. Если предположить, что реализация конвергентна, то есть принять такую реализационную гипотезу, то можно было бы продолжить тестирование после наблюдения стационарности. Естественно, продолжение начинается с передачи стимула в реализацию. Тут нужно обратить внимание, что наблюдение стационарности мы делаем в тесте, когда после некоторой трассы определяем приём все реакций, а потом уже, после θ-перехода, передаём стимул. В другом тесте после этой же трассы мы могли бы сразу передавать этот стимул. Поведение после этого стимула может быть различным в этих двух случаях, точнее поведение после стационарности является, очевидно, частным случаем поведения без наблюдения стационарности. Поэтому-то тестирование с продолжением после стационарности будет более эффективным, поскольку различает эти случаи. Например, в спецификации некоторое поведение возможно только после приёма стимула в *нестационарном* состоянии. Тогда мы поймаем ошибку в реализации, которая имеет это поведение после приёма стимула в *стационарном* состоянии.

Эта идея воплощается в соответствии, которое называется *input-output refusal relation,* сокращённо **ior**. Другое название: *отношение с повторяющейся стационарностью, repetitive quiescence relation*. Оно основано на вложенности трасс, также как трассовый предпорядок и соответствие **iot**, но разрешает продолжение после стационарности. Иными словами, теперь рассматриваемые трассы наблюдений – это произвольные последовательности в алфавите стимулов, реакций и символа стационарности δ, а не только те, которые содержат δ лишь как последний символ. Такие трассы называются трассами с задержками, *suspension traces*. Их можно считать частным случаем трасс с отказами, failure traces, когда единственное множество отказов, refusal set, встречающееся в трассе – это множество всех реакций, то есть стационарность.

После этого естественно соединить преимущества соответствий **ioconf** и **ior**. Для этого, как в **ior** нужно рассматривать трассы с задержками, но требовать не вложенности таких трасс, а, как в **ioconf**, рассматривать продолжения общих трасс реализации и спецификации реакциями и стационарностью. Так мы получаем соответствие, которое называется **ioco** (Jan Tretmans). Несколько неудачно исторически сложилось, что расшифровка аббревиатуры такая же, как для ioconf – *input-output conformance*. Соответствие **ioco** определяется аналогичным правилом:

> Если трасса с задержками может наблюдаться как в реализации, так и в спецификации, то после этой трассы реализация может ... тогда и только тогда, когда это возможно в спецификации после этой же трассы. Под ... понимается одно из двух:
> 1) выдать реакцию `!y`;
> 2) не выдавать никаких реакций, то есть находиться в стационарном состоянии.

Соответствие ***ioco*** – это основное соответствие, с которым мы будем работать дальше.

Оно обладает следующим важным свойством.

Если взять за основу общие трассы реализации и спецификации, то можно сказать, что все остальные трассы спецификации – это ответвления от общих трасс через реакции или

стационарность, а все остальные трассы реализации – это ответвления от общих трасс через стимулы. Образно говоря, в спецификации больше реакций, а в реализации больше стимулов. Реализация и спецификация связаны отношениями *may* & *must, может* и *должна*. Реализация *должна* принимать стимулы, определяемые спецификацией, но *может* выдавать лишь некоторые из реакций или не выдавать никаких реакций из списка возможных вариантов, предлагаемых спецификацией.

Теперь мы можем ослабить требование запрета блокирующего deadlock`а для реализации. Новое требование звучит так: после *общей* трассы (то есть трассы общей для реализации и спецификации) реализация не должна блокировать стимулы, которые определены, то есть принимаются спецификацией после этой трассы.

Требования к реализации.

Теперь сформулируем требования к реализации для этих соответствий, которые обеспечивают выполнение двух условий тестирования:

- Условие стационарности: истечение тайм-аута в принимающем состоянии теста означает стационарное состояние реализации.
- Условие запрета блокирующего deadlock`а.

1) Соответствия **`tp`**, **`iot`** и **`ior`**, эквивалентные вложенности множеств соответствующих трасс.

   Для выполнения условия стационарности, прежде всего, требуется конвергентность реализации: все её состояния, достижимые из начального состояния, должны быть конвергентны. Кроме этого, должно быть ограничено сверху время выдачи реакции и время прохода любой цепочки τ-переходов.

   Для запрета блокирующего deadlock`а, как было сказано выше, реализация должна принимать все стимулы в каждом стабильном состоянии.

2) Для остальных соответствий **`tr`**, **`ioconf`** и **`ioco`** можно ослабить эти требования к реализации. Фактически, нам нужно, чтобы требования выполнялись только после *общей* трассы наблюдений реализации и спецификации.

   Вместо конвергентности всех состояний реализации, достижимых из начального, потребуем, чтобы были конвергентны только те состояния реализации, которые достижимы по общим трассам реализации и спецификации. Соответственно, переход по реакции и цепочка τ-переходов должны быть ограничены по времени только, если они начинаются в таком состоянии.

   Требование о приёме всех стимулов в каждом стабильном состоянии, достижимом из начального, ослабляется в двух отношениях: 1) достаточно учитывать только те состояния, которые достижимы по общим трассам реализации и спецификации; 2) достаточно учитывать только те стимулы, которые определены в спецификации после таких трасс.

Генерация тестов для ioco. Вывод тестового примера.

Для каждой трассы с задержками строится тестовый пример следующим образом:

- Сначала строим детерминированный автомат (в частности, без τ-переходов), в котором есть только одна заданная трасса. Естественно, вместе со всеми своими начальными отрезками.
- ♣Терминальному состоянию припишем вердикт ***pass***.
- ♣Пусть в трассе после её начального отрезка есть δ-переход. Тогда из начала перехода проводим «направо» переходы по всем реакциям, которыми этот отрезок трассы может продолжаться в спецификации. Справа у нас будут ***pass***-состояния.
- ♣«Налево» проводим переходы по остальным реакциям, то есть, по реакциям, которых нет в спецификации. Слева у нас будут ***fail***-состояния.
- ♣Пусть в трассе после её начального отрезка есть переход по реакции. Тогда для остальных реакций делаем всё аналогично: направо – правильные реакции, а налево – ошибочные.
- ♣Если спецификация после начального отрезка трассы может оказаться в стационарном состоянии, проводим δ-переход направо,
  ♣в противном случае – налево.
- ♣Теперь «инвертируем» все символы: приём стимула ?x меняется на выдачу стимула !x, а выдача реакции !y меняется на приём реакции ?y. Кроме того, символ δ меняем на символ θ.
- Тест для трассы готов.

Нетрудно показать, что каждый такой тест является значимым, а вся их совокупность – полной.

Конечность теста и перечислимость полного тестового набора.

В духе соответствия ***ioco*** требование конвергентности реализации следовало бы сформулировать более осторожно: в реализации может быть дивергенция только в том случае, когда она может быть в спецификации в такой же ситуации, то есть после той же самой трассы с задержками. Поскольку мы хотим, чтобы дивергенция не возникала при тестировании, мы не должны проверять те трассы спецификации, которые в спецификации могут заканчиваться дивергенцией. Сейчас мы ограничимся спецификациями, в которых нет трасс, заканчивающихся дивергенцией, то есть конвергентными спецификациями. Все трассы такой спецификации годятся для тестирования. В дальнейшем мы ослабим требование конвергентности, когда будем рассматривать общий случай трасс с блокировками и разрушением и соответствующее обобщение соответствия ***ioco***.

Математическая модель

Как мы уже говорили, с практической точки зрения нам требуются две вещи: 1) перечислимость полного тестового набора и 2) конечность теста.

Спецификация трасс.

Поскольку тест строится по каждой трассе с задержками, которая есть в спецификации, число таких трасс должно быть перечислимо. Для конечных трасс это эквивалентно перечислимости множества стимулов и реакций, которыми может продолжаться каждая трасса.

1) Будем считать, что задан алгоритм перечисления множества стимулов и реакций, которыми может продолжаться каждая трасса – итератор $SI(\sigma)$.

Для построения *конечного* теста мы должны уметь после каждой трассы за конечное время определить, продолжается ли эта трасса в спецификации данной реакцией, полученной от реализации. Аналогично, мы должны определить, продолжается ли эта трасса в спецификации стационарностью δ. Таким образом, множество, составленное из реакций и символа δ, которыми продолжается каждая трасса σ, должно быть разрешимо.

2) Будем считать, что задан алгоритм $P(\sigma,y)$, который за конечное время проверяет, имеет ли трасса продолжение реакцией y.

3) Будем считать, что задан алгоритм $P_\delta(\sigma)$, который за конечное время проверяет, имеет ли трасса σ продолжение стационарностью δ.

Итератор трасс строится так. Для каждой трассы итератор базовых символов после трассы, фактически, задаёт нумерацию этих символов. Заметим, что номер символа определяется не только самим символом, но и предыдущей трассой. Если трасса продолжается символом δ (проверяется алгоритмом $P_\delta(\sigma)$), то этому символу присвоим номер 1, а базовые символы, возвращаемые итератором, будем нумеровать, начиная с 2. Индексом трассы назовём сумму номеров её символов. Очевидно, что число трасс с данным индексом конечно, и его можно перебрать алгоритмически. Тогда итерация трасс реализуется двумя вложенными циклами: во внешнем цикле перечисляем индекс, а во внутреннем – трассы с этим индексом.

При построении теста по трассе мы используем алгоритм $P(\sigma,y)$ для проверки каждой реакции y, получаемой от реализации в принимающем состоянии теста, когда трасса продолжается другой реакцией или стационарностью, а также алгоритм $P_\delta(\sigma)$ – для проверки стационарности, то есть истечение тайм-аута в принимающем состоянии теста, когда трасса продолжается реакцией.

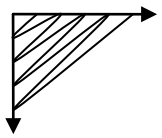
Спецификация асинхронного автомата.

Этих трёх требований нам было бы достаточно, если бы спецификация задавалась как множество трасс с задержками. Однако наша модель – это асинхронный автомат. Поэтому мы должны переформулировать наши требования в терминах состояний и переходов.

Вместо перечислимости стимулов и реакций после трассы мы должны говорить о перечислимости стимулов и реакций, по которым определены переходы в состоянии.

1) Будем считать, что задан алгоритм перечисления стимулов и реакций, по которым определены переходы в состоянии, – итератор $SI(s)$.

Для перечислимости трасс нам было бы достаточно перечислимости *переходов* по каждому символу в каждом состоянии.



Если в каждом состоянии определено перечислимое множество переходов, множество состояний после конечного маршрута также перечислимо. Имеем перечислимое множество символов, для каждого из которых имеем перечислимое множество символов. Устраиваем «диагональ».

Так мы перечисляем все маршруты и, тем самым, все трассы: нужно только отфильтровывать те трассы, которые уже были получены раньше.

Для проверки реакций мы должны уметь алгоритмически проверять, имеется ли в состоянии s переход по реакции y. Теперь нам нужно проверять каждую реакцию после трассы, опираясь на проверку реакции во всех состояниях, достижимых по этой трассе. Это возможно только в том случае, когда трасса заканчивается в конечном числе состояний. Такое требование эквивалентно конечности числа переходов по каждому базовому символу z в каждом состоянии s. Поэтому нам нужен соответствующий итератор переходов в состоянии.

Кроме того, нужно учитывать τ-переходы, поскольку они не меняют трассу. У нас должны быть конечными, во-первых, число τ-переходов из каждого состояния и, во-вторых, число состояний достижимых из данного состояния по τ-переходам. Второе условие перекрывается требованием конвергентности.

2) Будем считать, что задан алгоритм перебора переходов из состояния s по символу z, где z – это стимул, реакция или символ τ, – итератор $TI(s,z)$.

Чтобы проверить реакцию в состоянии теперь достаточно вызвать итератор переходов по этой реакции в состоянии и проверить, что он возвращает хотя бы один переход.

Однако стационарность стабильного состояния означает, что в нём нет переходов ни по одной реакции из, быть может, бесконечного множества реакций. Стабильность состояния (отсутствие τ-переходов) проверяется итератором $TI(s,\tau)$. Нам требуется отдельный алгоритм проверки стационарности стабильного состояния.

3) Должен быть задан алгоритм $P_\delta(s)$, проверяющий, что в стабильном состоянии s нет переходов по реакциям, то есть стационарность состояния.

Требование конечности числа переходов по каждому символу доказывается так. Пусть есть неразрешимое перечислимое множество M. Пусть в конце трассы перечислимое, но не конечное, множество состояний U, в каждом из которых определён один символ из M, причём каждый символ из M определён ровно в одном состоянии из U. Тогда существование алгоритма, определяющего, что символ продолжает трассу, эквивалентно существованию алгоритма, разрешающего множество M.

Итерация символов после трассы сводится к итерации символов в конечном множестве состояний, в котором эта трасса заканчивается, и проверке стационарности в этих состояниях. Для этого с каждым состоянием этого множества связываем итератор символов в этом состоянии. Все эти итераторы связываем в цикл и вызываем по циклу. Когда итератор символов в состоянии возвращает очередной символ, мы проверяем, а не было ли такого символа раньше. Если такой символ уже был или итератор сообщил об окончании итерации, мы переходим к

следующему по циклу итератору. Итерация символов во множестве состояний заканчивается, когда итераторы символов во всех состояниях множества закончили итерацию. Для того чтобы продолжить трассу стационарностью, нужно проверить стационарность во всех состояниях множества. Если хотя бы одно из состояний стационарно, трасса может продолжаться символом δ. После такого продолжения мы переходим к подмножеству стационарных состояний.

Конечно-ветвящийся автомат.

Как частный широко распространённый случай спецификаций, для которых выполнены условия 1,2,3, можно рассматривать конечно-ветвящиеся спецификации, то есть такие, в каждом достижимом состоянии которых определено конечное число переходов. В этом случае требуется задать только итератор переходов в состоянии.

Генерация тестов для ioco. Оптимизация.

Способ генерации тестов, который мы рассмотрели, далёк от оптимального. Мы выбрали этот вариант, потому что он самый простой и наглядный. Что можно оптимизировать?

Выбрасываем вложенные матрёшки. Если у нас есть тест для трассы σ, а трасса μ – её начальный отрезок, то нам не нужен тест для трассы μ, поскольку он перекрывается тестом для трассы σ. Правда, здесь есть нюанс: если в спецификации есть бесконечная трасса, то для построения полного тестового набора нам не обойтись без таких повторов. Чтобы протестировать эту бесконечную трассу, мы вынуждены иметь бесконечное число тестов для её начальных отрезков конечной длины, поскольку среди них нет максимального. Но на практике мы перечисляем полный тестовый набор только для того, чтобы получить его конечное значимое подмножество. Вот в таком конечном тестовом наборе уже можно обойтись без повторов.

Дважды не ждём. Нам не нужны тесты, в которых есть несколько θ-переходов подряд. Ведь θ-переход выполняется в результате того, что мы ждали реакций в принимающем состоянии теста и не дождались их, тайм-аут истёк. Поскольку реализация предполагается конвергентной, если мы будем ждать ещё раз, то получим то же самое.

Нет ошибок – нечего и тестировать. Тест можно закончить тогда, когда исчезла неопределённость в возможном вердикте. Если любое продолжение трассы приводит к появлению только вердиктов pass или fail, то нам не нужно строить тесты для продолжений этой трассы.

Всё-таки лучше один раз сорок раз. Запуск каждого теста происходит после рестарта реализации, чтобы она начинала работать с начального состояния. Если мы хотим уменьшить число рестартов, мы должны уменьшить число тестов за счёт усложнения каждого теста. Понятно, что такая задача имеет смысл только для конечного набора тестов. Простейший способ соединить в один несколько тестов – это строить их не для каждой трассы отдельно, а для дерева трасс. Тогда тест будет выглядеть не как одна «бородатая» линия – линия с торчащими налево и направо ответвлениями единичной длина, а как «бородатое» дерево. Естественно сделать так, чтобы все pass-состояния были равноудалены от начального состояния, за исключением случаев бессмысленного продолжения, о котором мы только что говорили.

Везде был, во все двери стучался. Как продвинутое решение этой же задачи можно рассматривать тестирование, основанное на обходе автомата. В этом случае мы как бы пробуем каждый переход в каждом состоянии.

Не отрывая пера от бумаги. Как частный, но важный случай ставится задача провести тестирование с помощью одного теста, то есть вообще без рестарта. На этот счёт развивается своя интересная теория, и предлагаются свои алгоритмы и методы решения.

Тестирование, основанное на обходе автомата, является базовым для группы RedVerst, но о нём и тех ограничениях, при которых оно работает, нужно говорить отдельно.

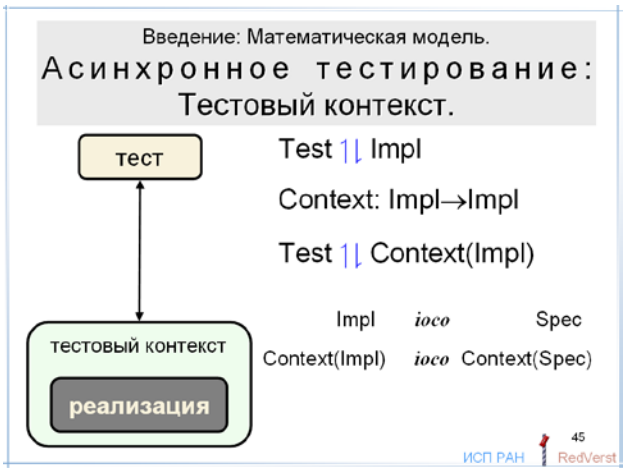
Асинхронное тестирование: Тестовый контекст.

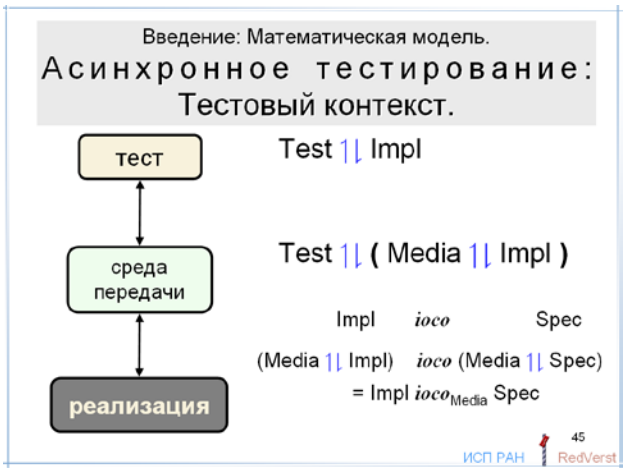
При синхронном тестировании тест непосредственно взаимодействует с реализацией. Это моделируется оператором параллельной композиции теста и реализации. Соответствие ioco непосредственно связывает реализацию и спецификацию.

При асинхронном тестировании реализация оказывается погруженной в, так называемый, тестовый контекст. Стандарт ISO определяет тестовый контекст в самом общем виде как отображение, преобразующее одну реализацию в другую. Тест компонуется с результатом такого отображения. Соответствие связывает преобразованные реализацию и спецификацию.

Понятно, что не любое такое преобразование имеет практический смысл.

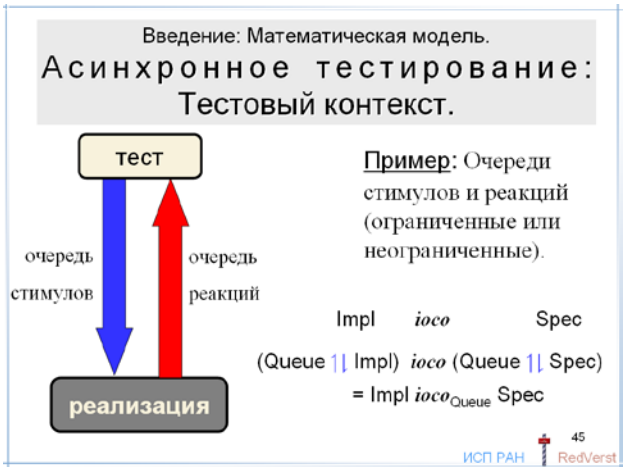
В автоматной модели естественно рассматривать тестовый контекст как среду передачи стимулов и реакций между тестом и реализацией. Такая среда также моделируется асинхронным автоматом, который компонуется с реализацией. Тест, в свою очередь, компонуется с этой парой. Асинхронное соответствие между реализацией и спецификацией определяется как соответствие между композицией реализации и среды, с одной стороны, и композицией спецификации и среды, с другой стороны.

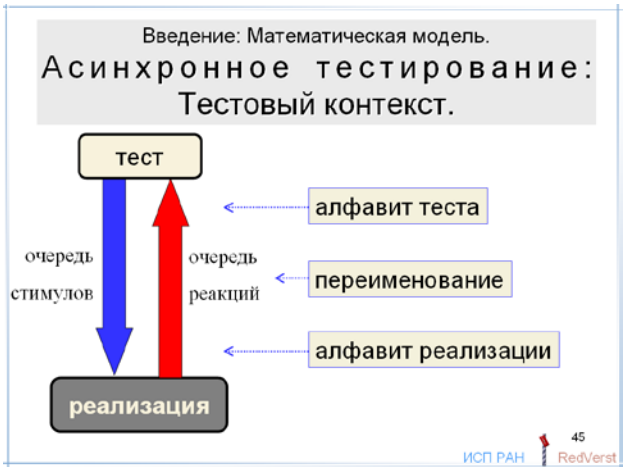




В качестве примера и наиболее широко распространённого частного случая можно рассмотреть среду, состоящую из двух очередей: очередь стимулов и очередь реакций. Каждая из этих очередей моделируется асинхронным автоматом. Очереди могут быть ограниченного размера или неограниченные.

Эту разновидность среды передачи мы будем использовать по умолчанию для иллюстрации проблем асинхронного тестирования и методов их решения. Если очередь стимулов неограниченная, то она всегда готова принять стимул от теста и блокирующего deadlock`а не возникает. Правда, реализация может не всегда быть готова принять стимул из очереди, но тест этого не видит.





Формально синхронный тест взаимодействует с реализацией по тому же алфавиту стимулов и реакций, по которому среда передачи взаимодействует с реализацией. Поэтому асинхронный тест должен взаимодействовать со средой по другому алфавиту, и эти два алфавита не пересекаются.

Для среды типа очереди эти два алфавита взаимно-однозначно соответствуют друг другу: стимулу, посылаемому тестом, соответствует стимул, принимаемый реализацией, и наоборот; реакции, выдаваемой реализацией, соответствует реакция, принимаемая тестом, и наоборот.

Для таких сред любой синхронный тест, очевидно, можно превратить в асинхронный, и наоборот, систематическим переименованием стимулов и реакций. Поэтому для простоты мы будем считать, что синхронный и асинхронный тесты работают в одном алфавите – обратном алфавиту реализации.

ПРОБЛЕМЫ асинхронного тестирования.

С асинхронным тестированием связаны две основные проблемы (см. пример на рисунке).

Первая проблема называется *вседозволенность*, по-английски, *permissiveness*. Она заключается в том, что некоторые ошибки, которые обнаруживаются при синхронном тестировании, не могут быть обнаружены при асинхронном тестировании.

На рисунке приведён пример, когда в спецификации реакция a может быть выдана с самого начала, а реакция b – после приёма стимула x.

В реализации имеется ошибка, легко обнаруживаемая при синхронном тестировании: после приёма стимула `x` выдаётся не реакция `b`, а реакция `a`.

Посмотрим, что будет происходить при асинхронном тестировании. Если тест начинает с ожидания реакций, то он получит реакцию `!a`, никакой ошибки мы не обнаружим. Если тест начинается с передачи стимула `?x`, то этот стимул попадает в очередь стимулов. И реализация, и спецификация имеют право сначала выдать реакцию `!a` в очередь реакций. Поэтому, если тест после передачи стимула начнёт ждать реакций, то он в обоих случаях может получить реакцию `!a`. Спецификация предлагает на выбор два варианта: `?x!b` или `?x!a`. Для реализации есть только один вариант: трасса `?x!a`. Поэтому, по принципу may&must никакой ошибки обнаружено не будет.

В общем-то, вседозволенность – это естественное следствие асинхронности тестирования. Тест не имеет такого непосредственного контакта с реализацией, как при синхронном тестировании, поэтому у него меньше возможностей обнаружить ошибку.

Другая проблема асинхронного тестирования не столь очевидна и не столь терпима. Это проблема *несохранения соответствия*, по-английски, *non preservation of conformance*. Асинхронное тестирование может обнаружить ошибку, которая не обнаруживается при синхронном тестировании.

На рисунке приведён пример второй реализации, в которой определён дополнительный приём стимула `z` и далее выдача реакции `c`. Рассмотрим тест, который выдаёт два стимула `x` и `z`, а потом ждёт реакций. С учётом буферизации стимулов и реакций в очередях среды передачи, спецификация может выдать реакцию `a` или `b`, а реализация – `b` или `c`. Если реализация выдаст реакцию `c`, будет обнаружена ошибка. При синхронном тестировании эта ошибка обнаружена быть не может. Действительно, в этом случае тест выдаёт стимул `z` только после приёма реакции `b`, но не выдаёт его сразу после выдачи стимула `x`, поскольку в соответствующем состоянии `s2` стимул `z` не определён.

Такого рода ошибки следует считать ложными, поскольку именно синхронное тестирование, как наиболее приближенное к реализации, является основным. Наличие ложных ошибок свидетельствует о том, что что-то не так с нашим пониманием соответствия или его моделированием с помощью асинхронного автомата. Более конкретно это связано с пониманием неспецифицированного стимула, в нашем примере – стимула `z` в состоянии `s2`.

Однако об этой проблеме и методах её решения мы поговорим подробнее позже.