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

Игорь Борисович Бурдонов
Александр Сергеевич Косачев

ТЕСТИРОВАНИЕ СООТВЕТСТВИЯ
(conformance testing)

1. Титул.

Тема нашего сегодняшнего рассказа – тестирование соответствия, по-английски conformance testing.

2. О чём пойдёт речь?

Сегодня 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.

3. ОГЛАВЛЕНИЕ

4. ВВЕДЕНИЕ

5. Введение: Основные понятия

6. 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.

7. 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.

8. 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`**`_√(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.

9. 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.

10. 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.

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

♣

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.

♣

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

11. 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 проблема: как извлечь модель из спецификации?

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

Как решать проблему извлечения модели из спецификации? Здесь приходится балансировать, если можно так выразиться, между желаниями человека и компьютера (или другого человека). Тот, кто пишет спецификации, хочет иметь универсальный язык, хотя бы потому, что ему лень изучать много разных языков. Ещё он хочет, чтобы спецификация была ему самому понятной. А для автоматического извлечения модели

желательно, чтобы тип этой модели как можно более явно отражался в конструкциях языка, делая его, тем самым, менее универсальным. И здесь важна не понятность для человека, а формальное соответствие типу модели. На практике часто приходится делать выбор между понятностью спецификации и явностью отражения в ней модели. И всё равно самый первый этап генерации тестов делается вручную как раз потому, что на этом этапе извлекается модель. Кроме того, эта модель ещё и преобразуется для удобства тестирования. Это, так называемый, процесс факторизации модели, который мы рассмотрим позже.

♣

Можно отметить, что для методов доказательства, в отличие от тестирования, требуется явное задание не только спецификационной, но и реализационной модели. А эта проблема на порядок сложнее. Понятно, что если у нас нет исходного кода реализации, то извлекать модель просто не из чего, и проблема становится неразрешимой. 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.

12. 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.

13. Введение: Формализация эксперимента

14. 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.

15. 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.

16. 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 нашим

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 я уже говорил, 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 я пока не встречал работ на эту тему. Правда, я не особенно их искал, 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 я уже намекал, 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.

17. 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.

18. 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».

19. 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.

20. 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 временно*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.

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

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

♣

Я уже говорил, 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`y 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 я скажу подробнее позже.

22. ПРОБЛЕМЫ: 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.

23. 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.

24. 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.

25. 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 $z_1, z_2, ... z_n$ может продолжаться бесконечным числом n+1-ых наблюдений $a_{n+1}, b_{n+1}, c_{n+1}, ...$. 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.

26. 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.

27. Введение: Математическая модель

28. Labelled Transition System (LTS).

Итак, мы отметили три основные проблемы тестирования: дивергенцию, недетерминизм и полнота тестового набора. Я намеренно рассказывал 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 $\mathbf{S}=(V,C,E,s_0)$.

♣

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 a соответствует переход, помеченный символом a; это внешний или наблюдаемый переход.

♣

Внутренней активности соответствует переход, помеченный символом τ; это внутренний, ненаблюдаемый переход.

♣

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. Очевидно, стабильное состояние конвергентно.

29. Функционирование и трассы наблюдений.

На слайде 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 слайде приведён пример двух таких трасс.

♣

Ещё одной разновидностью трасс являются, так называемые, *ready-traces*. Они похожи на трассы с отказами, но только, по 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.

30. Обозначения для трасс наблюдений.

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

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

♣ *Переход по трассе из одного символа* z:

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

♣ *Переход по произвольной трассе* σ = z1,z2,…,zn :

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

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

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

♣ *Переход из* a *по трассе* σ:

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

♣ *Нет перехода из* a *по трассе* σ:

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

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

a ***after*** σ = { b | $a=\sigma\Rightarrow b$ }

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

S ***after*** σ = s_0 ***after*** σ

31. 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.

32. 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».

33. 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.

♣

А 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.

34. 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.

35. 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.

36. 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.

37. 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.

38. 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`a с помощью θ-перехода оказывается наблюдением стационарности, которую мы тоже будем обозначать символом δ.

Замечу, 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.

39. Три машины тестирования.

Теперь, с учётом запрета блокирующего deadlock`a, 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.

Здесь вопросительный и восклицательный знаки выражают точку зрения теста: реакции он ждёт, а стимулы посылает. Для получения из трассы теста трассы реализации нужно применить биекцию подчёркивания, а символ истечения тайм-аута θ поменять на символ стационарности δ.

♣

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 кнопку приёма всех реакций тайм-аут встроен или нет, в зависимости от машины.

♣

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.

40. 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 *тра-та-та* тогда и только тогда, когда это возможно в спецификации после этой же трассы. Под *тра-та-та* 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 ?4δ, 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 *тра-та-та* тогда и только тогда, когда это возможно в спецификации после этой же трассы. Под *тра-та-та* 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 *тра-та-та* тогда и только тогда, когда это возможно в спецификации после этой же трассы. Под *тра-та-та* 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`a 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.

41. 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`a.

♣

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`a, как было сказано выше, реализация должна принимать все стимулы в каждом стабильном состоянии.

♣

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.

42. 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.

43. 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***.

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

♣

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.

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.

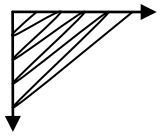
♣

Вместо перечислимости стимулов и реакций после трассы мы должны говорить о <u>перечислимости</u> 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.

♣

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.

44. Генерация тестов для ioco. Оптимизация.

Способ генерации тестов, который мы рассмотрели, далёк от оптимального. Я выбрал 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.

45. 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.

46. ПРОБЛЕМЫ асинхронного тестирования.

С асинхронным тестированием связаны две основные проблемы. Проиллюстрируем их на примере.

♣ Пусть у нас есть вот такая спецификация.

♣ Рассмотрим вот такую реализацию.

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 слайде 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 слайде 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.

47. АВТОМАТ МИЛИ

Сейчас мы рассмотрим один широко распространённый частный случай, для которого стирается различие между синхронным и асинхронным тестированием. Это автоматы Мили. Кроме того, на этом примере можно продемонстрировать общую структуру тестовой системы, способы извлечения модели из спецификации, генерации тестов и методы тестирования.

48. Автомат Мили: Введение

49. Определение и соответствие.

Хотя автомат Мили является частным случаем асинхронного автомата, он появился намного раньше и давно используется. В автомате Мили на каждый стимул выдаётся ровно одна реакция так, что любая трасса оказывается строго чередующейся последовательностью стимулов и ответных реакций. В таком автомате на переходе написан не один стимул или одна реакция, а всегда пара <стимул-реакция>, то есть это LTS без τ-переходов, алфавит которой – декартовое произведение множества стимулов на множество реакций.

♣

Автомат Мили можно преобразовать в асинхронный автомат, если такой переход заменить последовательностью из двух переходов: сначала по стимулу из пресостояния в промежуточное состояние, потом по реакции из промежуточного состояния в постсостояние.

♣

Мы будем считать, что промежуточное состояние одно для всех переходов из данного пресостояния по данному стимулу.

Если мы говорим, что наша модель – это автомат Мили, значит, автоматом Мили является как спецификационная, так и реализационная модель. Если автомат Мили рассматривать как частный случай более общей модели асинхронного автомата, то для спецификации просто проверяется, что она есть автомат Мили, а для реализации мы имеем реализационную гипотезу: реализационный автомат – это не любой асинхронный автомат, а автомат Мили.

Все состояния асинхронного автомата, представляющего автомат Мили, разделяются на стационарные, в которых только принимаются стимулы, и посылающие, в каждом из которых только выдаются реакции. Переход по стимулу всегда ведёт из стационарного состояния в посылающее, а переход по реакции – из посылающего в стационарное.

♣

Поэтому для автомата Мили отношение **ioco** оказывается эквивалентным отношению трассовой сводимости **tr**. Нам не нужен тайм-аут при ожидании реакций, поскольку заранее известно, что после стимула будет реакция, а после реакции будет стационарное состояние. Алгоритм генерации полного тестового набора для **ioco** легко модифицируется – в сторону упрощения – для трассовой сводимости.

♣

Для автомата Мили, конечно, можно использовать общую машину тестирования асинхронного автомата без встроенных тайм-аутов (машина **A**). Однако, поскольку передача стимулов и приём реакций строго чередуются, удобнее совместить последовательное нажатие кнопки стимула и кнопки приёма всех реакций.

Тогда у нас будет для каждого стимула одна кнопка, нажатие которой означает передать этот стимул и принять реакцию. В терминах автомата Мили как LTS это означает, что на кнопке написано множество символов вида (x,y), где стимул x фиксирован, а реакция y пробегает всё множество реакций.

♣При нажатии такой кнопки, если стимул принимается, то высвечивается пара (x,y), где x – посланный стимул, а y – принятая реакция.

♣

♣Если стимул не определён в данном состоянии, кнопка остаётся утопленной, но на дисплей ничего не высвечивается.

Дальше мы рассмотрим различные стратегии тестирования трассовой сводимости для конечного автомата Мили. Мы будем двигаться от простого к сложному, начиная с довольно сильных реализационных гипотез и/или сильных тестовых возможностей и простого устройства тестовой системы, и постепенно снимая ограничения на реализацию, довольствуясь меньшими тестовыми возможностями и усложняя структуру тестовой системы.

50. Три вида детерминизма.

Для автомата Мили можно дать два определения детерминизма: сильное и слабое аналогично сильному и слабому детерминизму для асинхронных автоматов.

♣

Автомат *сильно детерминирован*, если пресостояние и стимул однозначно определяют реакцию и постсостояние. В таком автомате в каждом состоянии определено не более одного перехода по каждому стимулу.

♣

В *слабо детерминированном* автомате Мили может быть несколько переходов из одного пресостояния по одному стимулу, различающихся реакциями, но тройка пресостояние, стимул и реакция однозначно определяют постсостояние.

♣

Очевидно, что сильный детерминизм влечёт слабый детерминизм

♣и наблюдаемый детерминизм. Обратное, вообще говоря, не верно. Если реализация слабо, но не сильно, детерминирована, то она наблюдаемо недетерминирована.

♣Сильно недетерминированным будем называть автомат, который не является слабо детерминированным. Такой автомат может быть наблюдаемо детерминированным, то есть пара пресостояние и стимул однозначно определяют реакцию, но неоднозначно определяют постсостояние. Здесь важно, что даже тройка пресостояние, стимул и реакция неоднозначно определяют постсостояние.

51. Спецификация в пред- и постусловиях.

Одной из наиболее адекватных и удобных для человека формой спецификации автомата Мили является спецификация в пред- и постусловиях. Предусловие – это предикат от пресостояния и стимула, который описывает, в каких состояниях какие стимулы определены в спецификационном автомате. Постусловие – это предикат от пресостояния, стимула, реакции и постсостояния, который в предположении, что предусловие выполнено, говорит, какие реакции могут быть в ответ на этот

стимул, поданный на автомат в этом пресостоянии, и в какие постсостояния может перейти автомат.

Такая спецификация описывает автомат неявно. Чтобы узнать, какие переходы определены в данном пресостоянии, нужно решить уравнение **PRE**(пресостояние,стимул)=**true** относительно стимула. А чтобы после этого узнать, какие могут быть реакции и постсостояния, нужно решить уравнение **POST**(пресостояние,стимул,реакция,постсостояние)=**true** относительно реакции и постсостояния. Понятно, что решать подобные уравнения в общем случае очень трудно. Для примера, который мы посмотрим на следующем слайд, всё очень просто, но что делать, если постусловие вот такое:

♣

POST: ∃a,b,c∈**Натуральные** $a^{\text{постсостояние}}+b^{\text{постсостояние}}=c^{\text{постсостояние}}$.

52. Пример спецификации.

В качестве примера рассмотрим сумматор над ограниченным множеством чисел 0, 1 и 2 с двумя недетерминированными операциями: ***увеличение*** и ***уменьшение***, которые на слайде обозначены как **+** и **–**. Состоянием является содержимое сумматора: 0, 1 или 2. При ***увеличении*** содержание сумматора строго увеличивается; эта операция имеет предусловие: состояние должно быть меньше максимального числа 2. Соответственно, при ***уменьшении*** содержание сумматора строго уменьшается; эта операция имеет предусловие: состояние должно быть больше минимального числа 0. Каждая операция возвращает величину, на которую изменился сумматор.

53. Спецификационный подавтомат.

Теперь мы рассмотрим проблему извлечения модели из спецификации. К счастью, для этого во многих случаях мы можем обойтись без решения или почти без решения уравнений пред- и постусловия. Идея заключается в том, чтобы извлекать модель из спецификации в процессе тестирования, и делать это автоматически. Модель, которую мы будем извлекать из спецификации, не будет полностью описывать спецификационный автомат. Это будет подавтомат, зависящий от реализации. А именно: та часть спецификационного автомата, которая необходима и достаточна для проверки трассовой сводимости данной реализации.

♣

Для нашего примера спецификационный автомат, если его извлечь из пред- и постусловий, выглядит следующим образом. В состоянии 0 есть два перехода по стимулу ***увеличение***, а в состоянии 2 – два перехода по стимулу ***уменьшение***.

♣

Пример реализации выбран такой, чтобы сумматор всегда изменялся ровно на 1. Реализационный автомат отличается от спецификационного автомата отсутствием переходов, связанных с изменением сумматора на 2. Хотя этот автомат не всюду определён по стимулам, запрет блокирующего deadlock`a сохраняется: после любой общей трассы реализация принимает все стимулы, которые определены в спецификации после этой трассы.

♣

Очевидно, при тестировании, пока не обнаружена ошибка, могут проходиться только общие трассы реализации и спецификации. Ошибка заключается в том, что после некоторой общей трассы σ и стимула x, которым трасса σ может продолжаться в спецификации, реализация возвращает запрещённую спецификацией реакцию y, то есть в спецификации нет трассы σ(x,y). Чтобы проверить реакцию, нам нужно знать все переходы спецификации по стимулу x, определённые во всех спецификационных состояниях, достижимых по трассе σ.

Поэтому спецификационный подавтомат определяется множеством состояний спецификации, достижимых по общим трассам, и всеми выходящими из них переходами. Если через пресостояние перехода проходит общая трасса, но никакая общая трасса не проходит через сам переход, то такой переход будем называть висящим.

♣Висящие переходы не могут проходиться при тестировании, но они добавляются для проверки конформности.

В нашем примере есть два висящих перехода: это увеличение и уменьшение сумматора на 2. Напомним, что выделение того или иного подавтомата зависит от реализации.

♣Например, рассмотрим реализацию, которая изменяет сумматор на максимально возможную величину.

♣Прежде всего, здесь появляется состояние, недостижимое из начального состояния, и его можно удалить.

♣Мы получаем другой подавтомат общих трасс и, соответственно, другой спецификационный подавтомат. Здесь нет переходов по изменению сумматора на 1 в состояниях 0 и 2, эти переходы становятся висящими.

Мы ставим задачу построения спецификационного подавтомата в процессе тестирования. Разумеется, для того, чтобы в конечном тестовом эксперименте можно было выделить спецификационный подавтомат, этот подавтомат должен быть *конечным*: в нём должно быть конечное число состояний и переходов. Формально нам не требуется конечность всего спецификационного или реализационного автомата.

В нашем примере мы могли бы, не меняя функциональных требований, сделать спецификацию бесконечной – она бесконечно продолжалась бы после висящих переходов, то есть после реакций, отсутствующих в реализации. Правда, число таких висящих переходов, да и вообще всех переходов, определённых в каждом состоянии спецификационного подавтомата, должно быть конечно. Соответственно, реализацию также легко сделать бесконечной, добавив в неё новые стимулы, отсутствующие в спецификации; реализация бесконечно продолжалась бы после переходов по этим стимулам. Таких стимулов может быть бесконечно много – всё равно они нас не интересуют.

54. Автомат Мили: Структура тестовой системы

55. Четыре компонента тестовой системы.

♣Для определения стимулов, которые нам нужно подавать на реализацию в известном пресостоянии, мы можем осуществлять перебор всех стимулов из алфавита стимулов и каждый проверять на *предусловие*. Это фильтрация по стимулам.

♣Программа, которая осуществляет такой перебор с фильтрацией, называется *итератором стимулов*.

В нашем примере итератор стимулов перебирает всего два стимула: **+** и **-**.

♣Реакцию нам вообще не нужно вычислять, поскольку при тестировании мы получаем её от реализации в ответ на стимул. Уж какая реакция пришла – такая пришла; тест ждёт всех реакций.

♣Другое дело постсостояние – его нужно определить. Во-первых, чтобы проверить постусловие, и, во-вторых, чтобы использовать как пресостояние при подаче следующего стимула. Компонент тестовой системы, который определяет постсостояние, называется *генератором постсостояния*. В общем виде он имеет вид:

```
постсостояние = функция_от(пресостояние,стимул,реакция).
```

Конечно, написать такую функцию – это то же самое, что решить уравнение постусловия относительно постсостояния. Для недетерминированной (ни сильно, ни слабо) спецификации эта функция будет к тому же многозначна, то есть генератор должен вернуть множество возможных постсостояний.

Однако в некоторых случаях такую функцию написать можно. В частности, если спецификационный автомат детерминирован, хотя бы слабо, то постсостояние определяется однозначно, и обычно не составляет труда определить эту функцию уже при написании постусловия. Тогда генератор постсостояния пишется легко, его даже можно генерировать из спецификации. Другой вариант опирается на возможность наблюдения состояния реализации, но об этом мы поговорим позже.

В нашем примере спецификация слабо детерминирована, а постсостояние вычисляется предельно просто: оно равно сумме пресостояния и реакции.

Заметим, что генератор постсостояния не проверяет, что при заданных пресостоянии, стимуле и реакции такое постсостояние будет удовлетворять постусловию.

♣Эту проверку делает компонент, который называется *тестовым оракулом*. Если постусловие ложно, то это квалифицируется как ошибка реализации; в противном случае тестирование продолжается, а постсостояние становится новым пресостоянием спецификации.

На нашем примере это хорошо видно.

Если возможных постсостояний несколько, постусловие отфильтровывает их. Ошибка фиксируется, если ни одно из возможных постсостояний не удовлетворяет постусловию. Если оказывается, что несколько возможных постсостояний удовлетворяют постусловию, то возникает проблема недетерминизма: мы не знает, какое постсостояние выбрать в качестве текущего для продолжения тестирования.

56. Три проблемы перечисления.

Прежде чем двигаться дальше, отметим три проблемы перечисления, которые возникают при тестировании. Это перечисление стимулов, постсостояний и реакций, которое нужно выполнять в состоянии спецификации.

Перечисление стимулов осуществляет итератор стимулов с фильтрацией по предусловию. При заданном пресостоянии он должен последовательно выдавать все стимулы, удовлетворяющие предусловию. Иными словами, перечисление стимулов – это перечисление решений уравнения предусловия для заданного пресостояния относительно стимула. Это общая задача для всех видов тестирования.

Перечисление постсостояний осуществляет генератор постсостояния с фильтрацией по постусловию. Для автомата Мили при заданном пресостоянии, стимуле и реакции он перечисляет решения уравнения постусловия относительно постсостояния. При сильном детерминизме постсостояние однозначно определяется парой пресостоянием и стимулом. При слабом детерминизме оно определяется тройкой, то есть зависит ещё и от реакции. Если автомат сильно недетерминирован, постсостояние неоднозначно определяется даже тройкой пресостояние, стимул и реакция. Именно в этом случае требуется перечисление постсостояний для автомата Мили.

Реакцию мы получаем от реализации, и поэтому нам не нужно её вычислять. Другое дело, что для полноты тестирования нам нужно получить все имеющиеся в реализации реакции на данный стимул при разных прогонах теста данной трассы. Как это делать – отдельная проблема.

57. Пятый компонент тестовой системы: обходчик конечного автомата. Условия обхода.

Итак, мы обозначили четыре компонента тестовой системы: итератор стимулов, проверка предусловия, генератор постсостояния и тестовый оракул, проверяющий постусловие. Для извлечения модели нам нужен ещё один, пятый компонент, отвечающий за общую стратегию тестирования. Он называется *обходчиком автомата*.

Обходчик автомата решает задачу «везде побывать, во все двери постучаться»: в каждом состоянии спецификации, в которое мы можем попасть по трассе, полученной в результате тестирования реализации без обнаружения ошибки, то есть общей для реализации и спецификации, мы должны попробовать каждый переход, определённый в этом состоянии в спецификации. Эта проверка либо обнаружит ошибку, либо добавит переход в строящийся подавтомат спецификации.

Автомат Мили называется сильно связным, если из каждого состояния в каждое другое состояние ведёт хотя бы один маршрут (цепочка переходов). В таком автомате, если он конечен, существует маршрут, начинающийся в начальном состоянии и проходящий через все переходы. Такой маршрут называется обходом. Сильно связность и конечность является достаточным и *почти* необходимым условием существования обхода.

♣Это условие становится необходимым для алгоритмов обхода заранее неизвестного автомата. Таким образом, если подавтомат конечен и сильно связен, задача теоретически может быть решена одним тестовым примером, без рестарта – не отрывая пера от бумаги. В противном случае может потребоваться несколько тестовых примеров, покрывающих в совокупности все переходы подавтомата спецификации. Для бесконечного автомата, очевидно, число тестов также будет бесконечно.

♣При обходе автомата мы сталкиваемся с проблемой недетерминизма. Дело в том, что для обхода нам нужно уметь выбирать в данном состоянии выходящий из него переход по своему усмотрению. Однако при фиксированном спецификационном пресостоянии переход определяется

тремя параметрами: стимулом, реакцией и постсостоянием, а тест умеет управлять только стимулом.

♣Поэтому сначала мы выдвинем два условия: 1) при тестировании реакция будет только одна, и 2) при этой реакции постсостояние тоже будет одно.

♣Второе условие означает, что спецификация слабо детерминирована. Может возникнуть вопрос: насколько сильным является это ограничение? Иными словами, имеют ли слабо детерминированные спецификации такую же спецификационную мощность, как и любые спецификации, или меньшую? Известно, что для любого недетерминированного порождающего автомата существует детерминированный автомат, порождающий то же множество трасс. Если недетерминированный автомат конечен, то соответствующий детерминированный автомат тоже конечен. Это утверждение известно как основная теорема о регулярных множествах.

Нас, однако, интересует не столько сам спецификационный автомат, сколько описывающая его спецификация, которая всегда есть конечная запись. Если автомат конечен, то, очевидно, его всегда можно описать спецификацией конечного размера. Однако конечная спецификация может описывать не только конечные автоматы. Не может ли получиться так, что конечная спецификация описывает некоторый недетерминированный бесконечный автомат, а соответствующий ему детерминированный автомат не описывается конечной спецификацией? Для решения этой задачи нам, очевидно, не хватает формального понимания того, что такое сама конечная спецификация. Здесь есть некоторые «ножницы» между моделью автомата, описываемой спецификацией, и самой спецификацией.

Но вернёмся к тестированию.

♣Если спецификация слабо детерминирована, то условие единственности реакции означает реализационную гипотезу о том, что для данной реализации спецификационный подавтомат сильно детерминирован, если не считать висящих переходов. Это свойство будем называть *спецификационным детерминизмом реализации*. При запрете блокирующего deadlock`а из этого свойства следует, что реализация наблюдаемо детерминирована, по крайней мере, на общих трассах. Однако эта гипотеза требует от реализации большего, а именно, определённого соответствия состояний. Будем говорить, что два состояния реализации и спецификации соответствуют друг другу, если они достижимы по некоторой общей трассе, соответственно, в реализационном автомате и спецификационном автомате (и, следовательно, подавтомате). Спецификационный детерминизм можно

определить так: если два реализационных состояния соответствуют одному и тому же спецификационному состоянию, то

♣для любого стимула, определённого в этом спецификационном состоянии,

♣эти реализационные состояния, принимая этот стимул,

♣♣не могут возвратить разные, но обе правильные реакции. Если возвращаемая реакция не допускается спецификацией, то это означает ошибку несоответствия.

♣Итак, мы требуем, чтобы для данной реализации спецификационный подавтомат был сильно детерминированным, конечным и сильно связным.

В нашем примере мы выбрали как раз такую реализацию.

58. Пример сильно детерминированной, но спецификационно недетерминированной реализации.

Спецификационный детерминизм, вообще говоря, более сильное требование, чем сильный детерминизм.

♣В нашем примере мы можем скопировать два состояния: **`0`** и **`1`**. Копии обозначены штрихом и выделены цветом. Эта реализация уже имеет не `3`, а `5` состояний, но она всё ещё спецификационно детерминирована.

♣Однако, если в состоянии **`0`**` для стимула **`+`** выбрать реакцию не `+1`, а `+2`, что допускает спецификация, то получится спецификационно недетерминированная реализация. Реализационные состояния **`0`** и **`0`**` соответствуют спецификационному состоянию **`0`**. Спецификация разрешает в этом состоянии выдавать в ответ на стимул **`+`** как реакцию `+1`, так и реакцию `+2`. И реализация в состоянии **`0`** выдаёт `+1`, а в состоянии **`0`**` выдаёт `+2`. Получается, что эта реализация сильно детерминирована и конформна спецификации.

♣Однако спецификационный подавтомат недетерминирован.

♣В нём из одного состояния выходят два перехода по одному стимулу, но с разными реакциями.

Здесь возникает вопрос: насколько практична наша гипотеза о спецификационном детерминизме? Прежде всего, можно заметить: если и реализация, и спецификация обе сильно детерминированы, то гипотеза верна.

В этом случае спецификационному недетерминизму просто неоткуда взяться. Если же, как в нашем примере, сильно детерминирована только реализация, а спецификация даёт выбор реакций, то есть слабо детерминирована, то лазейка для спецификационного недетерминизма остаётся. Вопрос в том, что означает этот выбор? Если это трактуется как внешний выбор, то есть выбор между разными реализациями, гипотеза верна. В одной реализации всегда выбирается реакция `+1`, а в другой `+2`. Гипотеза говорит о том, что выбор не должен быть внутренним, то есть выбором между состояниями одной реализации: в одном состоянии `+1`, а в другом состоянии `+2`. Во многих случаях мы можем быть уверены, что разные реакции, разрешаемые спецификацией, предполагают разные реализации, а не разные состояния одной реализации.

59. Обход автомата.

Теперь наша тестовая система состоит из пяти компонентов.

♣Посмотрим, как она работает на нашем примере.

♣Сначала обходчик запрашивает у итератора стимулов новый стимул, определённый в текущем состоянии.

♣Итератор, выбирает первый стимул и проверяет его через постусловие. Если стимул отвергается, итератор выбирает следующий стимул.

♣Стимул, прошедший проверку предусловия, поступает в реализацию.

♣В ответ мы получаем реакцию.

♣После этого генератор постсостояния, зная пресостояние, стимул и реакцию, вычисляет постсостояние.

♣Затем эта четвёрка проверяется тестовым оракулом. Если постусловие выполнено, обходчик автомата добавляет переход в строящийся подавтомат.

♣ ещё один цикл

♣Обходчик может подавать в реализацию стимул, который раньше уже подавался в текущем состоянии спецификации, то есть, минуя итератор стимулов.

♣Если реализация вернула неправильную реакцию,

♣то это ловится тестовым оракулом, который выдаёт вердикт ***fail***.

♣…♣

Тестирование продолжается, по крайней мере, до тех пор, пока хотя бы в одном состоянии построенного подавтомата есть определённый в нём, но ещё не опробованный стимул. Если таких состояний и стимулов нет, то обход совершён, и подавтомат полностью построен.

♣Теперь мы видим, что мы построили именно подавтомат: в нём нет спецификационных переходов, изменяющих состояние сумматора на 2.

Я не буду рассказывать о самом алгоритме обхода, это вопрос интересный, но технический. Об этом можно прочитать в нашей статье в журнале «Программирование». Важно лишь отметить, что это обход неизвестного заранее спецификационного подавтомата. О том, куда ведёт и какой реакцией помечен переход, начинающийся в данном пресостоянии и помеченный данным стимулом, мы узнаём только тогда, когда оказываемся в этом пресостоянии и подаём этот стимул. Только тогда мы получаем в ответ заранее неизвестную реакцию и определяем заранее неизвестное постсостояние. И только тогда мы можем этот переход нарисовать.

60. Обход автомата и полнота тестирования.

Итак, мы построили спецификационный подавтомат в процессе тестирования с одновременным обходом этого подавтомата. Такое тестирование, очевидно, значимое, но оно может оказаться ещё не полным.

Тем не менее, даже такое тестирование практически полезно. Когда мы делали свой первый проект с Нортелем по спецификации и тестированию интерфейса ядра операционной системы, мы применяли как раз такой метод. И было найдено около 200 ошибок, хотя предполагалось, что их почти нет.

Обход становится полным тестированием только при очень сильной реализационной гипотезе, которая фактически предполагает выполненным ровно то, что из неё должно следовать, а именно: реализация такова, что обход порождаемого ею спецификационного подавтомата оказывается полным тестированием. Такая гипотеза далеко не всегда может быть принята.

♣В качестве примера можно рассмотреть спецификационно детерминированную реализацию нашего сумматора с пятью состояниями.

♣После того как будет закончен обход спецификационного подавтомата, окажется не проверенным переход из состояния **1** в состояние **2** по стимулу **+**. Эта реализация конформна спецификации.

♣Но в ней легко сделать ошибку, которую мы не обнаружим при обходе, заменив в не пройденном переходе правильную реакцию `+1` на неправильную реакцию `-1`.

♣Хотя достаточно было бы дать ещё четыре стимула **++-+**, чтобы ошибку обнаружить. Похожий случай у нас был на практике, кажется, с ним столкнулся в своё время Лёша Демаков.

Конечно, во всём этом нет ничего страшного, поскольку наша цель заключалась не в том, чтобы построить полный тест, а в том, чтобы извлечь из спецификации нужную спецификационную подмодель, что мы и сделали. Мы можем делать двухэтапное тестирование:

♣сначала делаем обход с построением подавтомата,

♣а потом применяем любые методы тестирования соответствия, основанные на явном задании спецификационной модели. В частности, мы можем применять общий алгоритм генерации полного тестового набора для трассовой сводимости.

♣Такое двухэтапное тестирование полезно не только для полноты тестирования, но и для построения быстрого теста, предназначенного специально для обнаружения конкретной ошибки. Если при обходе мы обнаружили ошибку на некотором переходе, то по пройденному к этому моменту автомату мы можем построить тест как цепочку переходов без циклов, начинающуюся в начальном состоянии и заканчивающуюся проверкой этого перехода. При желании можно даже строить такую цепочку минимальной длины. После того как в реализации эта ошибки исправлена,

мы можем вместо того, чтобы снова делать полный обход, прогонять этот быстрый тест, чтобы проверить, что ошибка действительно исправлена. Это называется *спрямлением* теста.

61. Полное тестирование явно заданного конечного автомата.

С другой стороны, специально для сильно детерминированных автоматов Мили создана довольно хорошо разработанная теория тестирования соответствия, доведённая до практически применимых алгоритмов с неплохими оценками сложности. В основном предлагаемые алгоритмы стремятся построить один полный тест, то есть полное тестирование ведётся без рестарта – не отрывая пера от бумаги. Правда, такие алгоритмы налагают дополнительные ограничения, как на спецификацию, так и на реализацию.

♣Спецификация должна быть *минимальной* – в ней не должно быть эквивалентных состояний. Это не страшно, поскольку существуют алгоритмы и даже инструменты минимизации явно заданного автомата Мили.

Ограничения на реализацию означают соответствующую реализационную гипотезу и сводятся, в основном, к ограничению числа состояний реализации. Если **модель ошибок**, то есть наши предположения о том, какие могут быть ошибки, утверждает, что *ошибки не увеличивают числа состояний*, то число состояний в реализации не больше, чем в спецификации. Такие ошибки могут быть двух типов: *не та реакция* или *не то постсостояние*, но не дополнительное постсостояние.

♣Алгоритмы работают не только тогда, когда число состояний реализации не превосходит числа состояний спецификации, но и тогда, когда имеется ограниченное число дополнительных состояний. Нужно только учитывать, что каждое дополнительное состояние увеличивает время тестирования в число раз, равное числу стимулов. Это утверждение доказано Василевским в 1973 г. Кстати, я обнаружил в интернете *42* ссылки на эту работу Василевского в англоязычной литературе и *ни одной* в русскоязычной.

Чтобы не было недоразумений, нужно сказать, что само по себе отсутствие в реализации «лишних» состояний не гарантирует ни того, что реализация изоморфна спецификационному подавтомату, ни того, что обход спецификационного подавтомата будет полным тестированием. Вот два примера реализации нашего сумматора. При желании можно проверить, что обе реализации конформны спецификации. В них ослаблены предусловия операций: в каждом состоянии можно увеличивать и уменьшать.

♣Левая реализация имеет всего одно состояние. Она не изоморфна спецификационному подавтомату, но обход является полным тестированием.

♣Правая реализация имеет три состояния, как и спецификация, но, во-первых, она также не изоморфна спецификационному подавтомату, а, во-вторых, при обходе спецификационного подавтомата мы не проверяем один реализационный переход, в котором как раз и могла бы быть ошибка.

62. Автомат Мили: Тестирование с открытым состоянием

63. Смешанный подавтомат.

Другой способ достичь полноты тестирования предполагает дополнительные тестовые возможности и позволяет достичь полноты тестирования уже при обходе. Эти тестовые возможности следующие: в каждый момент времени мы можем узнать состояние реализации. Такое тестирование называется *тестированием с открытым состоянием*.

Это можно понимать как наличие специальной операции, которая называется *status message*.

♣Предполагается, во-первых, что такая операция определена в каждом состоянии реализации. В нашей модели это означает, что в каждом состоянии автомата Мили определён переход-петля, стимул которого – это запрос состояния, а реакция – значение состояния. Во-вторых, предполагается, что эта операция достоверна: это действительно петля и возвращаемая реакция действительно значение состояния. Эту операцию мы не тестируем, предполагая, что она работает правильно. Иными словами, наличие такой тестовой возможности одновременно предполагает выполненной реализационную гипотезу о достоверности операции *status message*.

♣Другое понимание – это когда мы считаем, что значение постсостояния мы получаем вместе с реакцией как её часть. Какое понимание выбрать – это лишь вопрос интерпретации и удобства отображения в модели. На практике у нас может быть, например, специальная операция в классе объекта, возвращающая состояние объекта. Или состояние реализации – это поля данных объекта, которые мы можем просто прочитать. Опять-таки, мы используем реализационную гипотезу о том, что состояние реализации полностью хранится именно в этих полях данных и ни в каком другом месте – других полях данных этого объекта, в других полях данных других объектов, в глобальных переменных и т.п.

Что нам даёт открытость реализационных состояний?

♣Мы может строить и обходить в процессе тестирования не спецификационный, а смешанный подавтомат, состояния которого – это пара состояний: вычисленное спецификационное состояние и считанное из реализации реализационное состояние. Выполняя переход, мы считываем реализационное постсостояние и вычисляем спецификационное

постсостояние. Последнее нужно не только для того, чтобы проверить постусловие, но и для того, чтобы осуществить итерацию стимулов, то есть определить, какой следующий стимул посылать в реализацию. Эту итерацию мы делаем отдельно для каждого реализационного состояния, входящего в пару с данным спецификационным состоянием.

Такой смешанный подавтомат, конечно, тоже может быть «меньше» всей реализации, поскольку в реализации могут быть дополнительные стимулы, которых нет в спецификации, и поэтому мы их не тестируем при обходе.

Хотя состояние подавтомата – это пара состояний реализации и спецификации, однако, это не даёт нам право не читать реализационное постсостояние, если мы вычислили то спецификационное постсостояние, в котором уже были. Пример спецификации сумматора с *тремя* состояниями и реализации с *пятью* состояниями показывает, что одному спецификационному состоянию может соответствовать несколько реализационных. Точно также, мы не можем отказаться от вычисления спецификационного постсостояния в случае, если попали в то реализационное состояние, в котором уже были. Пример реализации сумматора с *двумя* состояниями показывает, что одному реализационному состоянию может соответствовать несколько спецификационных, для каждого из которых нужна своя итерация стимулов.

В общем случае мы имеем некоторое соответствие между состояниями реализации и спецификации, причём такое, что для перехода из одного состояния должен найтись так же помеченный переход из соответствующего состояния, и концы этих переходов также соответствуют друг другу. Такого рода отношения между LTS называются *симуляциями* или, если они симметричны, *бисимуляциями*. Для LTS общего вида существует несколько таких симуляций: строгая, слабая, branchig, delay и другие.

В слабо детерминированных автоматах Мили нет τ-переходов и отношение, которое здесь может быть, является разновидностью строгой симуляции. Отличие в том, что в этих автоматах различаются стимулы и реакции, и соответствия типа **ioco** несимметричны относительно них. Реализационное и спецификационное состояние соответствуют друг другу, если мы можем оказаться в этих состояниях после общей трассы в реализации и спецификации.

♣Если пресостояния `s` и `i` соответствуют друг другу, а стимул `x` определён в `s`, то,

♣в силу запрета блокирующего deadlock`а, этот стимул также определён в состоянии `i`.

♣Для *сильно* детерминированной реализации каждый такой стимул `x` однозначно определяет реакцию `y` и постсостояние `i`` в реализации, тем самым, однозначно определяя пару `(x,y)`.

♣Эта пара, в свою очередь, для *слабо* детерминированной спецификации либо приводит к нарушению постусловия, либо однозначно определяет постсостояние `s`` и переход в спецификации.

Это показывает, что обход смешанного подавтомата остаётся детерминированным: выбирая стимул, мы можем получить только одну реакцию, реализация может перейти только в одно состояние, а при этих условиях, если не обнаружена ошибка, спецификационное состояние также единственно.

Все трассы автомата, который мы строим и обходим, являются общими для реализации и спецификации. Когда обход завершён, оказывается, что мы прошли все общие трассы. Любая трасса реализации, ответвляющаяся от общей трассы, делает это по «лишнему» стимулу, которого нет после этой трассы в спецификации и который мы, тем самым, не должны тестировать. Аналогично, любая трасса спецификации, ответвляющаяся от общей трассы, делает это по «лишней» реакции, которая не обязана быть в реализации, и поэтому мы не можем требовать появления такой реакции при тестировании.

♣Тем самым, обход оказывается полным тестированием.

64. Тестирование с открытым состоянием. Оптимизация

Легко заметить, что для парных состояний мы делаем избыточное тестирование. Если реализационное состояние входит в пару с `10` спецификационными состояниями и в `5` из них определён стимул `x`, то получится, что мы, по крайней мере, `5` раз подавали стимул `x` в этом реализационном состоянии, трактуя этот стимул как новый. Он новый для каждого из этих `5` спецификационных состояний, но реализационное состояние одно и то же, поэтому и результат в сильно детерминированной реализации будет один и тот же. Поэтому достаточно было бы это делать `1` раз.

♣Оптимизация заключается в том, чтобы хранить только реализационное состояние и отфильтровывать не только те стимулы, которые нарушают предусловие, но и те, которые мы уже опробовали в данном реализационном состоянии. Это можно сделать двумя способами.

♣Первый способ заключается в следующем. Как только мы попадаем в данное реализационное состояние, мы, как и ранее, вычисляем соответствующее спецификационное состояние. Но делаем мы это не для того, чтобы искать или создавать новую пару состояний, а для того, чтобы начать итерацию стимулов для вычисленного спецификационного состояния с самого начала, отфильтровывая уже опробованные стимулы.

♣Другой способ оптимизации лучше объяснить на примере объектно-ориентированной реализации тестовой системы. В этом случае итератор стимула – это объект, который мы можем создавать для каждого реализационного состояния и каждого спецификационного состояния, когда это спецификационное состояние оказывается в паре с реализационным состоянием.

♣Вместо создания пары состояний мы храним вместе с реализационным состоянием множество таких созданных итераторов. Когда мы попадаем в это реализационное состояние, мы вычисляем спецификационное состояние и ищем во множестве его итератор. Если такого итератора ещё нет, создаём его и он начинает работать с начала. Если же такой итератор уже есть, то мы вызываем его. В любом случае делаем фильтрацию по уже опробованным стимулам.

65. ПРОБЛЕМЫ: Взрыв состояний.

Такое тестирование с открытым состоянием порождает одну из проблем, имеющих общее название «взрыв состояний». Дело в том, что наш смешанный подавтомат может иметь слишком много состояний. Это происходит за счёт реализационных состояний, входящих в пару. Поэтому оптимизация, когда вместо пары используется только реализационное состояние, ничего не меняет. Спецификация – это всё-таки некоторая абстракция, то есть мы отвлекаемся от множества несущественных деталей; поэтому спецификационных состояний может быть приемлемое количество. Реализационное же состояние – это просто набор бит. Если суммарный объём полей данных, представляющих состояние, равен `100` битам, то мы

имеем 2^{100} реализационных состояний. Все они формально разные. Понятно, что для таких величин тест будет работать неприемлемо долго.

♣

Здесь мы сталкиваемся с классическим противоречием двух желаний: желание абстрагироваться от реализационных деталей и желания иметь полное тестирование, учитывающее все реализационные детали.

66. ПРОБЛЕМЫ: Взрыв состояний. Решение.

Решение проблемы можно найти, вводя эквивалентность реализационных состояний, реализационную эквивалентность. До этого места у нас был договор: мы умеем читать реализационное состояние. Но мы не договаривались, что мы умеем его понимать. Именно поэтому для нас все эти 2^{100} состояний различны, и мы формально вынуждены стремиться тестировать поведение реализации в каждом из них, если мы его можем достигнуть.

Предположим, что мы кое-что знаем об устройстве реализации, что она не совсем уж «чёрный» ящик. На этом этапе нам не нужно знание её алгоритмов, закодированных в программном коде; нам нужно знать только, как устроено её состояние. Мы можем сообразить, что из этих `100` бит `30` несущественны, поскольку не влияют на выполнение нужных нам операций, то есть обработку стимулов. Может быть, эти `30` бит нужны для других операций, но мы эти другие операции не тестируем. Далее, оказывается, что байтовое поле служит для хранения не `256` разных значений, а только пяти символов в коде ASCII. И так далее.

В целом мы получаем отношение эквивалентности на множестве реализационных состояний, и нам не нужно тестировать каждый стимул в каждом состоянии, а только каждый стимул в каждом классе эквивалентности. Разумеется, такой вывод можно сделать только на основе соответствующей реализационной гипотезы: те состояния, которые мы признали эквивалентными, действительно эквивалентны.

♣Это значит, в терминах модели ошибок: если ошибка проявляется для одного из состояний, то

♣ она проявляется и для каждого эквивалентного состояния.

Если для одного состояния нет ошибок, то их нет для всех эквивалентных состояний. Во многих случаях такая гипотеза вполне приемлема.

Конечно, здесь можно заметить, что спецификационные состояния тоже в каком-то смысле опираются на некоторую эквивалентность реализационных состояний. Почему же нам понадобилась другая эквивалентность? Это объясняется известными «ножницами» между уровнем абстракции спецификации и уровнем абстракции признаваемой модели ошибок, которую мы кладём в основу реализационной эквивалентности. Какие-то детали могут оказаться несущественными с точки зрения функциональных требований, и потому не отражены в спецификации. Однако при поверхностном взгляде на реализацию мы видим, что эти детали составляют существенную её часть, и нет уверенности, что они не влияют на функциональность. Иными словами, только глубокое изучение реализации позволило бы подтвердить или опровергнуть гипотезу о несущественности этих деталей. Вместо глубокого изучения мы оставляем эти детали различимыми в реализационной эквивалентности, и их не влияние на функциональное поведение будем проверять тестированием. Зато другие детали можно проигнорировать, поскольку в их невлиянии на функциональность мы уверены.

67. Отображение открытых состояний.

Тестирование с открытым состоянием может использоваться не только и не столько для усиления полноты тестирования при обходе, на также как способ определения спецификационного постсостояния. Идея заключается в том, что, зная кое-что об устройстве реализационного состояния, мы можем определить спецификационное постсостояние как функцию от реализационного состояния.

♣Иными словами, предлагается гипотеза о реализации: между реализационными и спецификационными состояниями существует соответствие, это соответствие является однозначным отображением реализационных состояний в спецификационные, и нам это отображение известно.

При всей внешней экстравагантности такой гипотезы, она имеет право на существование во многих случаях. Во-первых, если спецификация создаётся по коду реализации в процессе кодоинспекции, то это практически гарантирует правильное отображение. Оно будет оставаться правильным также и при модификации реализации, разумеется, в определённых пределах. Во-вторых, если, наоборот, реализация создавалась по спецификации, то есть большая доля уверенности, что разработчик достаточно ленив, чтобы не изобретать реализационные состояния совсем уж непохожими на спецификационные состояния. Он может что-то добавлять, выбирать форму представления данных и т.п., но обычно всегда остаётся связь со спецификацией, которую можно оформить как нужное нам отображение.

Если у нас есть такое отображение, то мы можем при построении спецификационного подавтомата не вычислять постсостояние как решение уравнения постусловия, а получать его как функцию от прочитанного реализационного постсостояния. Это можно делать и в тех случаях, когда решение уравнения постусловия далеко не так очевидно, как в нашем примере с сумматором. Более того, это можно делать и для недетерминированной реализации, хотя о методах тестирования такой реализации мы ещё не говорили.

♣Как обычно, эта медаль имеет оборотную сторону: возникает проблема достоверности отображения состояний. Отнюдь не всегда можно считать его стопроцентно достоверным, а отказываться от отображения не хочется. Что делать?

♣К счастью, тестовая система обладает хорошим самоконтролем: полученное (неважно каким путём) спецификационное постсостояние не просто принимается на веру, а проверяется постусловием. Если постусловие истинно, такое постсостояние могло бы быть. Если же тестовый оракул выносит вердикт ***`fail`***, то это может означать ошибку не в реализации, а в отображении. Особенно это удобно, когда спецификация слабо детерминирована, потому что тогда правильное постсостояние, если оно есть, то единственно, и истинность постусловия гарантированно подтверждает нам, что постсостояние то, какое надо.

68. Отображение открытых состояний (окончание).

Если отображение инъективно, то есть разным реализационным состояниям соответствуют разные спецификационные состояния, то реализационные и спецификационные состояния, участвующие в тестировании, взаимно-однозначно соответствуют друг другу. Иными словами, они отличаются только способом их кодирования. Поэтому спецификационный подавтомат, если из него удалить висящие переходы, можно считать тем самым подавтоматом реализации, который мы и должны были проверить на трассовую сводимость. Все остальные переходы реализации либо ведут из состояний, которые недостижимы по общим трассам, либо из достижимых состояний, но по стимулам, которые не определены в спецификации после соответствующих общих трасс. Тем самым, наличие или отсутствие этих переходов в реализации не влияет на её соответствие спецификации по трассовой сводимости. Таким образом, при тестировании с открытым состоянием и инъективным отображением состояний сильно

детерминированной реализации для проверки соответствия достаточно обхода спецификационного подавтомата.

♣При наличии реализационной эквивалентности (с соответствующей реализационной гипотезой) отображение можно понимать как отображение класса эквивалентности в спецификационное состояние. Соответственно, инъективность будет означать, что разные классы отображаются в разные состояния. Очевидно, инъективное отображение классов почти всегда является неинъективным отображением состояний (если только все классы не являются синглетонами, то есть множествами состоящими из одного элемента).

♣Если отображение неинъективно, то одному спецификационному состоянию могут соответствовать несколько реализационных состояний. Когда мы проверяем стимул, определённый в этом спецификационном состоянии, то может оказаться, что мы проверяем его только в одном из соответствующих состояний реализации. Аналогично обстоит дело с неинъективным отображением классов реализационной эквивалентности в спецификационные состояния. Обход спецификационного подавтомата уже не будет полным тестом.

Для полноты нужно использовать либо смешанный подавтомат, либо другие методы, о которых мы говорили раньше. Альтернативный вариант: считать эквивалентными те реализационные состояния, которые соответствуют одному спецификационному. Разумеется, это означает соответствующую реализационную гипотезу об отсутствии тех «ножниц», о которых мы говорили: между уровнем абстракции спецификации и уровнем абстракции признаваемой модели ошибок, которую мы кладём в основу реализационной эквивалентности.

69. Автомат Мили: Медиаторы

70. Шестой компонент тестовой системы: медиатор.

До сих пор мы неявно предполагали, что алфавиты стимулов и реакций реализации и спецификации совпадают. На практике, это почти всегда не так. Поэтому требуется ещё один компонент тестовой системы – медиатор.

Медиатор – это программа, которая осуществляет связь между двумя моделями: спецификационной и реализационной.

Тест создаётся и работает в терминах спецификационной модели, поэтому медиатор осуществляет преобразование спецификационных стимулов в реализационные стимулы и, в обратном направлении, реализационных реакций в спецификационные реакции.

Как минимум это отображение алфавитов реализации и спецификации. В программном смысле это преобразование из одного типа в другое. Спецификация и реализация работают с разными типами, которые представляют стимулы и реакции, более того, это могут быть типы разных языков. Например, тип стимула – это имя функции или метода класса плюс типы формальных параметров, а тип реакции – это тип функции или типы ответных параметров для языков с несколькими ответными параметрами.

В более сложных случаях преобразование стимула или реакции может зависеть от состояния спецификации или реализации и вообще от предыстории взаимодействия теста и реализации.

♣В терминах автоматной модели, медиатор – это автомат, который непосредственно взаимодействует с реализацией, то есть, компонуется с ней с помощью оператора параллельной композиции. Тем самым тест взаимодействует не с реализацией, а с результатом такой компоновки. Это очень похоже на асинхронное тестирование. Но здесь есть одно весьма существенное отличие: медиатор – это часть тестовой системы, он создаётся разработчиком тестов. Поэтому мы не только всё знаем об этом медиаторе, но он находится под полным нашим контролем.

71. Интерфейс теста и медиатора.

Интерфейс теста и медиатора отличается от непосредственного интерфейса теста и реализации.

♣Передача стимула. Тест посылает спецификационный стимул в медиатор, который готов принять любой спецификационный стимул от теста.

♣После этого медиатор преобразует спецификационный стимул в реализационный и посылает его в реализацию.

♣Поскольку мы запретили блокирующиё deadlock, передача стимула в реализацию всегда проходит, и медиатор сообщает об этом тесту.

♣Приём реакций. Тест посылает в медиатор запрос на приём реакций.

♣Медиатор начинает ждать от реализации всех реакций.

♣Когда приходит некоторая реализационная реакция, медиатор преобразует её в спецификационную реакцию, которую передаёт в тест.

♣Если реакций нет, медиатор обнаруживает это по тайм-ауту, то есть по θ-переходу,

♣и посылает в тест сообщение о стационарности δ.

72. Многоуровневые спецификации.

Спецификация обычно не зависит от языка реализации и операционной среды, в которой работает реализация, поскольку, как правило, это выходит за пределы функциональности требований. Но для того, чтобы тест мог взаимодействовать с реализацией, нужна привязка к этим реализационным вещам, что как раз и делает медиатор. Для реализаций разного типа могут создаваться разные медиаторы при сохранении одной и той же спецификации. Такие медиаторы называют первичными.

♣При многоуровневых спецификациях используются медиаторы между соседними уровнями спецификации, осуществляющие аналогичные преобразования. Разные уровни спецификации отличаются различной степенью абстрагирования от реализационных особенностей. Более точно: спецификация более высокого уровня описывает более слабые функциональные требования к реализации. Соответственно, тестирование также может быть многоуровневым. Но эту тему мы сегодня развивать больше не будем, за исключением одного специального случая, о котором речь пойдёт позже.

73. Преобразование стимулов и реакций.

Для медиаторного преобразования предполагается, что спецификация более абстрактна, чем реализация: реализационный стимул или реакция соответствуют только одному стимулу или реакции спецификации.

♣Преобразование реакций может быть не инъективно: разные реализационные реакции преобразуются в одну спецификационную реакцию. Но здесь никаких проблем не возникает, поскольку на уровне спецификации мы не различаем эти разные реализационные реакции.

♣Трудности возникают, когда медиаторное преобразование стимулов неоднозначно: одному спецификационному стимулу может соответствовать множество стимулов реализации, и медиатор выбирает один из них. Тем самым, возникает вопрос: а вдруг реализация делает ошибку при другом выборе?

74. Медиатор стимулов как «погода».

Если вспомнить машину тестирования, то медиатор можно понимать как её переднюю панель, осуществляющую интерфейс с реализацией. Когда мы видим на дисплее символ a, то это спецификационное a, которое медиатор преобразовал из реализационных a1, a2, и так далее. Когда мы нажимаем на кнопку a, то это спецификационное a, которое медиатор преобразует в одно из реализационных a1, a2, и так далее. В этом смысле медиатор стимулов является частью погоды, от состояния которой зависит выполнение реализации. Но это такая погода, которая находится под нашим контролем: медиаторы создаются разработчиками тестовой системы, и составляют её часть.

Таким образом, у нас есть выбор:

♣Нет плохой погоды: мы принимаем реализационную гипотезу об эквивалентности реализационных стимулов, соответствующих по медиаторному отображению одному спецификационному стимулу.

♣Управление погодой: медиатор должен перебирать реализационные стимулы, соответствующие одному спецификационному стимулу. Важно, что это должно учитываться при обходе подавтомата: в состоянии

подавтомата оказываются определёнными не спецификационные, а соответствующие им реализационные стимулы.

75. Преобразование состояний.

При тестировании с открытым состоянием отображение состояний также является частью медиатора. Такое отображение может быть в первичном медиаторе, а может и не быть, если спецификационное постсостояние вычисляется по постусловию. Однако такое отображение почти всегда присутствует в медиаторах более высокого уровня, соединяющих соседние уровни спецификации. В общем случае, даже когда спецификационное состояние вычисляется, генератор постсостояния можно считать частью медиатора. Таким образом, медиатор осуществляет три преобразования:

- стимулов,
- реакций
- и состояний.

- Можно заметить, что медиатор похож на среду передачи при асинхронном тестировании: в обоих случаях предполагается их параллельная композиция с реализацией так, что тест компонуется уже с соответствующим композиционным автоматом. Существенное отличие в том, что медиатор является частью тестовой системы и находится под полным её контролем, а среда передачи – внешняя по отношению к тестовой системе так же, как реализация.

Мы рассмотрели медиаторы для автоматов Мили, но аналогичные медиаторы создаются и для асинхронных автоматов общего вида.

- При асинхронном тестировании мы имеем композицию четырёх автоматов: реализации, среды передачи, медиатора и теста.

76. Автомат Мили: Факторизация

77. Цели и проблемы факторизации.

Медиатор связывает реализацию, как детальную модель, и спецификацию, как обобщённую модель. Тем не менее, для тестирования спецификация часто оказывается всё ещё слишком подробной. Для того чтобы тестирование стало возможно и практично, приходится эту модель огрублять. Общий метод заключается в факторизации спецификации.

Основная цель факторизация: уменьшение размера спецификационной модели и, как следствие, размера тестов и длительности их выполнения.

♣В самом общем смысле спецификацию можно рассматривать как множество трасс наблюдений. Вводится эквивалентность трасс и строится фактор-спецификация на основе этой эквивалентности.

♣Для автомата Мили естественно строить эквивалентность трасс на базе эквивалентностей трёх видов: эквивалентности состояний, эквивалентности стимулов и эквивалентности реакций.

♣С факторизацией связаны две проблемы: 1) недетерминизм и 2) запрет блокирующего deadlock`a. Эти проблемы мы сейчас рассмотрим.

Проблема недетерминизма заключается в том, что при факторизации может увеличиться степень недетерминизма.

♣Самое интересное в том, что факторизация может и уменьшить недетерминизм. Такое уменьшение недетерминизма может даже составлять главную цель факторизации в некоторых случаях.

Выбор правильной факторизации – это задача тестировщика. Это та часть создания тестовой системы, которая плохо поддаётся автоматизации, здесь требуются чисто интеллектуальные усилия.

78. Тестовая модель.

Фактор-спецификация – это тоже спецификация, но с ослабленными требованиями. Можно было бы строить тестовую систему непосредственно

по фактор-спецификации, выбросив исходную спецификацию. Но тогда мы потеряли бы возможность более точно проверять правильность реализации.

Вместо этого используется трёхуровневая структура тестовой системы.

♣Уровень спецификации используется для генерации тестовых оракулов. Первичный медиатор обеспечивает отслеживание текущего спецификационного состояния, стимула и реакции.

♣Выше вводится уровень тестовой модели. Она связана с уровнем спецификации медиатором, который и осуществляет факторизацию.

♣Эта модель непосредственно используется для генерации тестов.

Таким образом, тестовая, более грубая, модель определяет более короткий тест, но проверка правильности получаемых от реализации реакций осуществляется, по-прежнему, тестовыми оракулами уровня спецификации.

79. Эквивалентность реакций.

Эквивалентность реакций предполагает реализационную гипотезу о том, что для полноты тестирования достаточно проверить выдачу реализацией хотя бы одной реакции из класса эквивалентных реакций. Простейшая эквивалентность определяется просто на множестве всех реакций. В общем случае нужно говорить об эквивалентности переходов так, что эквивалентные переходы должны отличаться только реакциями.

Формально, эквивалентность реакций уменьшает размер автомата, поскольку несколько переходов могут склеиться в один переход. Однако такие склеиваемые переходы отличаются только реакциями, выбором которых тест в большинстве случаев не умеет управлять. Поэтому введение эквивалентности реакций предназначено, в основном, для уменьшения недетерминизма.

♣Простейший случай, когда источником недетерминизма являются только реакции: пресостояние и стимул однозначно определяют постсостояние, но неоднозначно определяют реакцию. Этот вид детерминизма отличается от сильного и слабого детерминизма.

В спецификации допускаются кратные переходы, которые различаются только реакциями.

♣ Если такие реакции объявить эквивалентными,

♣ автомат становится сильно детерминированным.

Вообще говоря, эквивалентность реакций вовсе не гарантирует уменьшения недетерминизма. Более того, в некоторых случаях она может дать обратный эффект.

♣ Если из одного состояния вели два перехода по одному стимулу и разным реакциям в разные состояния, то это слабый детерминизм.

♣ Склеивая эти реакции,

♣ мы получаем недетерминизм.

80. Эквивалентность стимулов.

Эквивалентность стимулов в основном предназначена для уменьшения размера автомата. Поскольку стимулами управляет тест, то чем меньше будет стимулов, тем быстрее будет проходить тестирование.

При факторизации по стимулам нужно следить за детерминизмом.

♣ ♣ Если склеиваются стимулы кратных переходов, степень детерминизма не изменяется.

♣ ♣ ♣ Однако если склеиваются стимулы переходов с разными постсостояниями, то это может привести к недетерминизму.

81. Эквивалентность состояний.

Эквивалентность состояний также предполагает соответствующую реализационную гипотезу, и может использоваться как для уменьшения размера автомата, так и для уменьшения недетерминизма.

Поскольку такая факторизация уменьшает число состояний, она уменьшает число переходов, что сокращает время тестирования.

С детерминизмом дело обстоит сложнее.

Переходы под одним и тем же стимулам и реакциям, которые вели из одного пресостояния в разные склеиваемые постсостояния, склеиваются в один фактор-переход.

♣♣Недетерминированный автомат может стать слабо детерминированным.

♣Если бы в этом примере не было состояний s3,s4, то исходный автомат оставался бы недетерминированным, а фактор-автомат стал бы сильно детерминированным.

Нужно отметить, что сама по себе эквивалентность не обязательно уменьшает недетерминизм; она может даже увеличить его. Вот пример.

♣Исходный автомат сильно детерминирован.

♣После склеивания состояний s1,s2, автомат стал недетерминирован.

♣Но если бы мы склеили также состояния s5,s6, автомат остался бы сильно детерминированным.

Поэтому искусство построения тестовой модели заключается в том, чтобы найти хороший баланс между практически обоснованной реализационной гипотезой об эквивалентности состояний и требованиями детерминизма.

82. Пример пула из трёх атомов.

Обычно при факторизации используются все три эквивалентности: реакций, стимулов и состояний. В качестве примера рассмотрим пул из фиксированного числа атомов с двумя операциями: `alloc` и `dealloc`. Чтобы автомат пула уместился на экране, число атомов возьмём маленьким – всего `3` атома. Атомы имеют номера `0,1,2`. Состояние пула – это три двоичных числа: `i`-ое число равно `0`, если `i`-ый атом свободен и `1`, если он занят.

♣Операция `alloc` – это один стимул, в ответ мы получаем номер выделенного нам атома: `0`, `1` или `2`. Это реакция. Если есть несколько свободных атомов, пул может вернуть номер любого из них и атом с этим номером становится занятым. Тем самым, по операции `alloc` пул слабо детерминирован. Будем считать, что эта операция имеет предусловие:

запрашивать атом можно только в том случае, когда в пуле есть свободные атомы.

По этой операции пул слабо детерминирован.

♣Операция `dealloc` освобождает атом, номер которого передаётся как параметр: `0`, `1` или `2`. Тем самым, здесь мы имеем три стимула. Реакция на каждый из этих стимулов одна и та же – подтверждение освобождения атома, и на слайде она не указана. По этой операции пул, очевидно, сильно детерминирован. Каждый из стимулов `dealloc` также имеет предусловие: освобождать можно только занятый атом.

♣Все вместе выглядит так.

♣Эквивалентными объявим состояния с одинаковым числом занятых атомов.

♣Тем самым, фактор-состояние – это, фактически, число занятых атомов: `0`,`1`,`2` или `3`.

♣Далее факторизуем реакции: все три реакции на `alloc` объявим эквивалентными.

♣Наконец, факторизуем стимулы: все три стимула, соответствующие операции `dealloc`, объявим эквивалентными.

Очень важно отметить, что факторизующий медиатор преобразует фактор-стимул `dealloc` в операцию `dealloc` с параметром – номером освобождаемого атома. Этот номер медиатор берёт из соответствующего спецификационного состояния: выбирается любой занятый атом. Какой именно атом выбрать – дело медиатора. Этот выбор может быть полностью недетерминированным, или, поскольку медиатор создаётся разработчиком теста и находится под полным контролем тестовой системы, мы можем заставить медиатор выбирать тот или иной занятый атом по каким-то удобным для нас правилам.

В результате мы видим, что получился сильно детерминированный фактор-автомат. То есть нам удалось увеличить степень детерминизма: был слабый, стал сильный.

♣На слайде видно, как сильно уменьшился автомат: даже для пула из 3-х атомов число состояний уменьшилось вдвое, а число переходов – в 4 раза.

83. ПРОБЛЕМЫ: Запрет блокирующего deadlock`а.

А теперь рассмотрим вторую проблему факторизации. Мы договаривались запретить блокирующий deadlock.

♣Однако при факторизации состояний этот запрет может быть нарушен, если в один класс эквивалентности попадают два состояния, и в одном из них, `s1`, стимул `x` определён, а в другом, `s2`, – не определён.

Мы должны определить переход из такого фактор-состояния по стимулу `x` для того, чтобы проверить этот стимул в состоянии `s1`. Однако реализация может оказаться в состоянии, аналогичном состоянию `s2`, в котором стимул `x` не определён. Примером такой реализации может служить сама исходная спецификация, которая, очевидно, конформна сама себе и не приводит к блокирующему deadlock`у при тестировании по исходной спецификации. Однако при тестировании по фактор-спецификации такой deadlock как раз и возникнет.

♣Одно из решений этой проблемы основано на введении канонических состояний и канонических путей в канонические состояния. В нашем примере для стимула `x` каноническим будет состояние `s1`, в котором этот стимул определён.

♣Из состояния `s2` определяется канонический путь в состояние `s1`. Когда мы хотим в нашем фактор-состоянии дать стимул `x`, мы смотрим, в каком состоянии мы находимся. Если это состояние `s1`, то стимул можно давать. Если же это состояние `s2`, то сначала мы даём цепочку стимулов, приводящую нас по каноническому пути в состояние `s1`, а потом уже даём стимул.

Такой метод может применяться, если спецификация обладает достаточным детерминизмом и связностью, так как в противном случае гарантированного пути из `s2` в `s1` может не существовать.

♣Но самое главное: выделение канонических состояний и путей требует явного задания спецификации. Если же спецификация имплицитна, то мы получаем такое явное представление только постепенно в процессе тестирования.

Более того, тестирование мы хотим вести не по исходной спецификации, а по фактор-спецификации. Здесь получается своеобразный логический круг. Поэтому такой метод для имплицитной спецификации не столько решает проблему, сколько создаёт новые проблемы.

♣Другой метод состоит в том, чтобы ограничиться только такими эквивалентностями состояний, которые называются вполне определёнными. Это означает, что у нас просто не должно возникать ситуации, когда стимул определён в одном состоянии, но не определён в другом, эквивалентном ему, состоянии. Во многих практических случаях удаётся найти такую эквивалентность. В частности, нам это удалось в примере с пулом.

84. Автомат Мили: Слабый детерминизм

85. Сильно детерминированная реализация.

До сих пор мы стремились получить автомат, который был бы конечным, сильно связным и сильно детерминированным, поскольку только для такого автомата у нас был алгоритм обхода. Для этого мы либо предполагали, что спецификационный подавтомат сильно детерминирован, либо, если состояния реализации открыты, использовали сильно детерминированный смешанный или, после оптимизации, реализационный подавтомат. Во всех случаях сама реализация должна быть сильно детерминирована, по крайней мере, на общих трассах.

У нас остался ещё один случай сильно детерминированной реализации и слабо детерминированной спецификации. Это случай тестирования с закрытым состоянием, когда спецификационный подавтомат остаётся слабо детерминированным.

Можно предложить идею алгоритма, который, используя сильный детерминизм реализации, позволял бы по-прежнему строить и обходить сильно детерминированный автомат. Этот автомат получается из слабо детерминированного спецификационного подавтомата с помощью расщепления состояний и размыкания циклов.

♣Проиллюстрируем эту идею на примере. Пусть в процессе тестирования мы в какой-то момент времени построили вот такой автомат. До последнего перехода мы имели сильный детерминизм, а последний переход делает автомат слабо детерминированным, поскольку в состоянии `s2` в ответ на стимул `x` мы получаем реакцию не `y`, как раньше, а `y1`.

Поскольку мы знаем, что реализация сильно детерминирована, мы можем заключить, что спецификационное состояние `s2` соответствует двум разным реализационным состояниям: в одном состоянии на стимул `x` приходит реакция `y`, а в другом – `y1`.

♣Тогда мы расщепляем состояние `s2` на два состояния, то есть, добавляем состояние `s2`. Соответственно, предпоследний переход теперь будет вести не в состояние `s2`, а в состояние `s2`. Из-за этого, в свою очередь, состояние `s1` становится слабо детерминированным.

♣Поэтому мы расщепляем и это состояние, добавляя состояние `s1`.

Эту процедуру расщепления состояний мы делаем до тех пор, пока не получится сильно детерминированный автомат. Это произойдёт тогда, когда будет разомкнут цикл переходов.

Для того чтобы можно было так работать, нам нужно помнить не только пройденный автомат, но и полный путь в нём от начального состояния до текущего состояния.

♣Следует отметить, что это только идея алгоритма. Самого алгоритма ещё нет, здесьмогут быть кое-какие проблемы.

86. Обход по стимулам.

Если реализация не является сильно детерминированной, то никакими ухищрениями нам не удастся обходить сильно детерминированный автомат. Если реализация в некотором состоянии `i` в ответ на стимул `x` может вернуть две разные реакции `y1` и `y2`, то, какому бы состоянию `s` автомата, который мы обходим, ни соответствовало реализационное состояние `i`, мы можем получать в состоянии `s` обе реакции в ответ на стимул `x`.

Выходом, конечно, является подходящая факторизация. Но факторизация – это всегда огрубление тестирования. Может оказаться, что такого огрубления, которое делает автомат сильно детерминированным, мы себе позволить не можем.

Задача гарантированного обхода автомата, который не является сильно детерминированным, очевидно, не имеет решения. Реализация, в зависимости от погоды, может любое число раз подряд выдавать реакцию `y1` и любой конечный тест не сможет определить, возможна ли реакция `y2`, или нет.

Раз такая цель недостижима, приходится довольствоваться другой, достижимой целью.

Мы можем поставить задачу: в каждом состоянии, которого мы смогли достичь при тестировании, попробовать каждый стимул. Такой обход называется обходом по стимулам.

♣Для обхода по стимулам неизвестного слабо детерминированного автомата необходимо и достаточно двух условий: 1) конечность, 2) сильно-Δ-связность.

Последнее условие означает, что мы можем гарантированно попасть из каждого состояния в каждое другое состояние. Для сильно детерминированного автомата такая цепочка переходов для заданных состояний однозначно определяется последовательностью подаваемых стимулов. Однако, для слабо детерминированного автомата это не так. Здесь требуется подбирать следующий стимул в зависимости от полученной реакции. Такую цепочку иногда называют *адаптивной*. Фактически, это алгоритм, вырабатывающий очередной стимул в зависимости от пройденной трассы.

Я не буду здесь останавливаться на точном определении сильно-Δ-связности и алгоритме обхода по стимулам. Это вопросы интересные, но технические. Об этом можно прочитать в нашей статье в журнале «Программирование».

87. Полнота тестирования. Сильно-Δ-связный подавтомат.

Вместо этого посмотрим лучше, как можно вести тестирование на основе такого обхода по стимулам. Мы будем считать, что спецификационный подавтомат конечен, слабо детерминирован и сильно-Δ-связен.

♣В процессе обхода по стимулам мы можем обнаружить ошибку, и тестирование на этом заканчивается.

♣Если же обход по стимулам удалось довести до конца, мы можем применять любые методы тестирования по явно заданной модели.

♣Однако в отличие от сильно детерминированного автомата, здесь мы можем получить новую реакцию в ответ на стимул, который уже давали раньше в данном текущем состоянии. Поскольку обход по стимулам закончен, сильно-Δ-связность гарантирует, что постсостояние этого нового перехода будет одним из уже пройденных состояний. Иными словами, у нас могут появляться только такие новые переходы, которые соединяют старые состояния и помечены старым стимулом, но новой реакцией.

♣При появлении такого нового перехода мы добавляем его в автомат

♣и продолжаем тестирование, быть может, начиная его заново, поскольку автомат изменился.

Так мы и двигаемся по циклу, пока не найдём ошибку, или пока тестирование не завершится с вердиктом pass.

88. Полнота тестирования. Сильно связный, но *не* сильно-Δ-связный подавтомат.

Если спецификационный слабо детерминированный подавтомат оказывается сильно связным, но не сильно-Δ-связным, обход не всегда возможен.

♣Тест либо завершает обход, либо обнаруживает, что не может его завершить.

После этого мы также можем применять дополнительное тестирование по явно заданной модели.

Однако, теперь, получив новую реакцию, мы можем обнаружить, что попали в новое постсостояние.

♣Тогда продолжаем обход,

♣пока не закончим его или пока это возможно.

Соответственно, автомат достраивается новыми состояниями и переходами. При этом также могут проходиться новые переходы или даже обнаруживаться ошибки. Если ошибок не обнаружено, продолжаем тестирование.

Следует отметить, что для тестирования по явно заданной модели нам нужно гарантированно попадать в интересующие нас состояния. Если прогоняется несколько тестов с рестартами между ними, то есть, отрывая перо от бумаги, то нам требуется гарантированно попасть в каждое состояние из начального состояния, но не обязательно обратно. Соответственно, требование сильно-Δ-связности ослабляется.

♣Понятно, что, если мы не можем гарантированно попасть в некоторое нужное нам состояние, то мы не сможем опробовать в этом состоянии все стимулы. В таком случае нужно искать обходные пути: другие реализационные гипотезы или тестовые возможности, например, пытаться управлять погодой и т.п.

89. Автомат Мили: Сильный недетерминизм

90. ПРОБЛЕМЫ: Сильный недетерминизм.

Автомат называется сильно недетерминированным, если он не является слабо детерминированным автоматом. Иными словами, пресостояние, стимул и реакция неоднозначно определяют постсостояние.

Для такого автомата, кроме перечисления стимулов, мы должны уметь делать перечисление постсостояний.

♣Есть кое-какие идеи, но нет никакого алгоритма обхода сильно недетерминированного автомата.

91. ПРОБЛЕМЫ: Все виды недетерминизма.

В общем, для недетерминированной реализации у нас остаётся проблема полноты тестирования, поскольку реализация может не выполнять все те переходы, которые у неё есть.

Как заставить реализацию вести себя по-разному?

♣Можно предложить два варианта.

Первый вариант – это те или иные тестовые возможности по управлению погодой в той или иной степени.

Второй вариант – введение вероятностей переходов и использование их при тестировании. Тогда тест будет проверять нужные переходы с некоторой вероятностью.

92. АСИНХРОННЫЙ АВТОМАТ

93. Асинхронный автомат: Допущение полноты

94. Тестовый контекст.

Напомним, что асинхронное тестирование – это тестирование через посредника – среду передачи.

Мы рассматриваем случай асинхронного тестирования, когда среда передачи – это две неограниченные очереди: входная очередь стимулов и выходная очередь реакций.

95. ПРОБЛЕМЫ асинхронного тестирования.

Как мы уже говорили, с асинхронным тестированием связаны две основные проблемы.

Проблема *вседозволенности* заключается в том, что некоторые ошибки, которые обнаруживаются при синхронном тестировании, не могут быть обнаружены при асинхронном тестировании. Иными словами, соответствие реализации и спецификации ослабляется. Это нормальная ситуация, она означает, что асинхронное тестирование менее точное, чем синхронное. Возмущение вносит среда передачи, она рассинхронизовывает тест и реализацию.

Вторая проблема – это проблема *несохранения соответствия*: асинхронное тестирование может обнаружить ошибку, которая не обнаруживается при синхронном тестировании. В этом случае происходит незаконное усиление соответствия. В чём причина несохранения соответствия?

96. Почему соответствие не сохраняется?

Причина, по которой соответствие не сохраняется, заключается в следующем: неспецифицированный стимул не подаётся в реализацию при *синхронном* тестировании, но может быть подан в неё при *асинхронном* тестировании. Именно это мы и наблюдали в нашем примере.

Почему так происходит?

♣Дело в том, что при асинхронном тестировании используется не исходная спецификация, а её композиция со средой. Для неограниченной входной очереди такая композиция оказывается всюду определённой по стимулам, то есть в ней нет неспецифицированных стимулов. Поэтому-то все стимулы и подаются в реализацию.

♣Тестирование по всюду определённой по стимулам спецификации называется *строгим*. Если же спецификация не всюду определённая, то при тестировании неспецифицированные стимулы не подаются и, тем самым, не проверяется поведение реализации на этих стимулах. Такое тестирование называется *слабым*.

♣Таким образом, причина несохранения соответствия заключается в том, что синхронное тестирование оказывается слабым, а асинхронное – сильным.

Если бы исходная спецификация была всюду определённой по стимулам, то проблемы несохранения соответствия не было бы.

Это наводит на мысль доопределить в спецификации все неспецифицированные стимулы.

♣Самый главный вопрос, который здесь возникает: что означает неспецифицированный стимул?

♣Можно считать, что если стимул не специфицирован, то это означает некоторое поведение по этому стимулу, которое подразумевается по умолчанию. Модель асинхронного автомата должна быть уточнена однозначной интерпретацией неспецифицированного стимула. Такого рода интерпретации называются *допущениями полноты*, а соответствующая процедура изменения спецификации – её *пополнением*.

97. Проблема самоприменимости спецификации.

Проблема неспецифицированных стимулов связана не только с проблемой несохранения соответствия, но и с проблемой *самоприменимости спецификации*. Эту проблему можно ещё назвать проблемой понимания спецификации разработчиком реализации.

♣Казалось бы, спецификацию можно рассматривать как одну из возможных реализаций. Это самая полная реализация, если иметь в виду, что в ней реализовано не только то, что должно быть реализовано, но и всё то, что может быть реализовано, то есть все те возможности, которые спецификация предлагает на выбор, по принципу *may&must*.

Разработчик, казалось бы, имеет право создать реализацию совпадающей со спецификацией. Если в стабильном состоянии стимул не специфицирован, то разработчик свободен в выборе, принимать стимул и что-то делать после его приёма или не принимать, блокировать стимул.

♣Однако для отношения ***ioco*** мы запретили блокирующий deadlock: реализация не может блокировать стимул в стабильном состоянии. Поэтому формально спецификация, имеющая неспецифицированные стимулы в стабильных состояниях, не может быть своей собственной реализацией.

♣Если же снять запрет блокирующего deadlock`a, то спецификация будет не конформна сама себе по отношению ***ioco***. Действительно, в недетерминированной спецификации некоторая трасса σ может закончиться в двух разных стабильных состояниях. При этом может оказаться, что в одном состоянии некоторый стимул x определён, а в другом – нет. В этом случае ***ioco*** требует, чтобы после трассы σ принимался стимул x, и при тестировании мы можем дать этот стимул, но принимается он только в одном состоянии, а в другом – блокируется, и требования отношения ***ioco*** не выполнены.

Поэтому и здесь возникает необходимость однозначной интерпретации неспецифицированного стимула, то есть того или иного допущения полноты.

98. Продолжение, блокировка и разрушение.

Существует три типа допущений полноты.

♣Стимул принимается. Предполагается, что неспецифицированный стимул можно подавать в реализацию, и реализация его принимает. Поведение после приёма стимула может быть различным. Дальше мы рассмотрим несколько вариантов такого поведения. В каждом из этих вариантов в спецификации определяется переход по приёму стимула с нужным дальнейшим поведением.

♣*Блокировка*. До сих пор мы запрещали блокирующий deadlock. Однако запрет на блокировку стимулов реализацией делает невозможным тестирование систем, в функциональность которых входит такая блокировка. Примером может служить ограниченная FIFO-очередь: передача стимула – это помещение символа в конец очереди, передача реакции – выборка символа из головы очереди. Спецификация очереди требует, чтобы стимул блокировался тогда и только тогда, когда очередь полностью заполнена. Такая очередь часто реализуется как очередь сообщений, когда операция *послать* (send) завершается с соответствующим кодом ответа, если сообщение не может быть передано немедленно. Иногда операция *послать* снабжается тайм-аутом (встроенным или передаваемым как параметр). Если сообщение не передано до истечения тайм-аута, операция *послать* заканчивается с соответствующим кодом ответа, что соответствует абстракции θ-действия.

Блокировка. Стимул можно подавать в реализацию, но реализация не принимает стимул. Возникает блокирующий deadlock. В этом случае никакого перехода по стимулу добавить в спецификацию нельзя. Поэтому этот вид умолчания можно считать в некотором смысле основным, а остальные умолчания интерпретировать как переопределение блокировки.

♣При тестировании конвергентной реализации блокировку можно обнаружить с помощью тайм-аута на передачу стимула.

♣*Разрушение*. Следующий тип умолчания предложен в группе RedVerst, хотя о такой семантике неспецифицированного стимула, как об одной из возможных, говорится в обзоре Lee и Yannakakis, посвящённом тестированию конечных автоматов Мили. Этот тип умолчания основан на понимании предусловия как абсолютного запрета на передачу в реализацию стимулов, нарушающих предусловие. Предполагается, что поведение реализации по поводу этого стимула полностью не определено: она может его блокировать, принимать с тем или иным продолжением, но самое главное, реализация может быть *разрушена*, что считается недопустимым. В некоторых языках спецификации для такого поведения имеется соответствующее ключевое слово, например, в RSL (Raise Specification Language) – это слово **`swap`**.

Исключение из рассмотрения блокируемых и разрушающих стимулов часто мотивируют тем, что реализация после приёма стимула должна проверять его корректность. Если стимул некорректен, реализация либо просто игнорирует его, либо сообщает об ошибке ответной реакцией. Такое требование к

реализации естественно, если это система «общего пользования», в ней должна быть предусмотрена «защита от дурака». Однако часто требуется тестировать внутренние компоненты или подсистемы, доступ к которым строго ограничен и взаимные проверки корректности обращений излишни. Такое часто встречается для стимулов со сложной внутренней структурой и нетривиальными условиями корректности, когда накладные расходы на проверку неоправданно увеличивают трудозатраты на создание системы, её объём и время выполнения. Альтернативой в этом случае является строгая спецификация предусловий взаимодействующих компонентов. Тестированию подлежит не поведение компонентов в ответ на некорректные стимулы, а правильность обращения компонентов друг к другу. Последнее означает, фактически, проверку поведения каждого компонента (по его постусловию) только для таких стимулов, которые удовлетворяют предусловию компонента.

Есть и ещё класс ситуаций, когда полезно специфицировать разрушение. Таким способом мы можем маркировать то поведение реализации, которое по каким-то причинам не хотим тестировать. Причиной может быть незаконченность реализации – какие-то части пока не полностью реализованы. Другая причина – желание проверить только часть поведения, но зато максимально полно; здесь мы пытаемся сократить время тестирования за счёт исключения из него неинтересующей нас части поведения. При тестировании, совмещённом с реальной работой системы, мы можем исключать поведение, перегружающее систему и ставящее под угрозу реальную работу. Например, можно тестировать распределение памяти, но нельзя приводить систему в состояние критически малого объёма свободной памяти, хотя поведение системы в этой ситуации может быть особенно интересным в другом отношении.

Разрушение. Стимул принимается реализацией, но это может привести к её разрушению. Разрушение не обязательно происходит сразу после приёма стимула. Возможно, перед этим выполняется выдача некоторых реакций. Здесь предполагается, что после приёма стимула реакции выдаются по инициативе самой реализации, а окружение может вести себя пассивно, то есть, окружение только принимает эти реакции. Таким образом, можно считать, что разрушение инициировано передачей в реализацию этого стимула. Разрушение мы предлагаем моделировать переходом, помеченным специальным символом γ.

♣Стимул, который может привести к разрушению, мы не должны подавать в реализацию при тестировании.

Именно интерпретация неспецифицированного стимула как разрушающего позволяет оставить тестирование слабым.

♣Мы будем считать, что тестируемая система не разрушается «сама по себе», то есть без приёма каких бы то ни было стимулов. Такой асинхронный автомат будем называть *безопасным*.

99. Асинхронный автомат: βγδ-трассы.

100. βγδ-машина тестирования.

Теперь мы можем рассматривать наблюдения, в которые, кроме внешних действий, входят не только стационарность, но также блокировки стимулов и разрушение. Стационарность изображается символом δ, разрушение – символом γ, а блокировка стимула `?x` – стимулом `?x` в фигурных скобках `{?x}`.

♣ Трассы таких обогащённых наблюдений мы будем называть βγδ-трассами, а соответствие ***ioco*** заменяется более общим соответствием $\boldsymbol{ioco}_{\beta\gamma\delta}$.

Сначала посмотрим, как устроена соответствующая машина тестирования, формализующая тестовые возможности, необходимые для такого рода наблюдений. Мы будем называть её βγδ-машиной.

Эта машина похожа на машину тестирования для трасс с задержками.

♣ Результатом эксперимента является βγδ-трасса, которую проходит тест.

♣ Передача стимулов и приём реакций происходит так же, за исключением двух случаев.

♣ ♣ ♣ ♣ ♣ Во-первых, в ответ на посылку стимула мы можем получить истечение тайм-аута, что означает блокировку стимула.

♣ ♣ ♣ Во-вторых, мы можем получить разрушение машины.

В трассе теста мы различаем переход по θ в двух типах состояний. В принимающем состоянии – это по-прежнему стационарность, а в посылающем состоянии – блокировка посылаемого стимула.

♣ Чтобы получить трассу реализации, нужно заменить

♣ вопросительный знак восклицательным и наоборот,

♣ символ θ в принимающем состоянии – символом δ,

♣а в посылающем состоянии – блокировкой посылаемого стимула.

101. Оператор параллельной композиции с разрушением.

Посмотрим, что меняется в операторе параллельной композиции при добавлении нового типа перехода – перехода по разрушению?

♣Наиболее отвечает семантике разрушения асинхронное выполнение перехода по разрушению. В этом разрушение аналогично внутреннему переходу или переходу по внешнему стимулу или реакции, которые есть в алфавите одного автомата, а противоположные (по подчёркиванию) им символы отсутствуют в алфавите другого автомата.

102. Безопасные трассы.

Рассмотрим наш пример βδγ-трассы: `?a !y δ δ !b {?a} ?c !y γ`.

♣Тестирование должно быть безопасным, то есть оно не должно приводить к разрушению реализации. Это означает, что при тестировании мы должны проходить только *безопасные* трассы – трассы, начальные отрезки которых которых не могут быть продолжены реакциями и далее разрушением.

В нашем примере βδγ-трасса не безопасна: приём стимула `c` приводит к разрушению.

♣Её максимальный безопасный начальный отрезок: `?a !y δ δ !b {?a}`.

103. Гипотеза о безопасности и конвергентности.

Итак, любая трасса, которая может быть получена при тестировании, должна быть безопасна в реализации.

Кроме этого, мы должны быть уверены, что истечение тайм-аута действительно означает блокировку в посылающем состоянии теста или стационарность в принимающем состоянии. Для этого каждая тестируемая трасса не должна проходить через дивергентные состояния, а время перехода

по стимулу или реакции так же, как цепочки τ-переходов, должно быть ограничено сверху. Иными словами, если в реализации есть дивергенция, то она должна быть недостижима по трассам, используемым в тестировании.

Трассу, которая безопасна и проходит только через конвергентные состояния, назовём *безопасно-тестируемой*. О безопасно-тестируемости трасс реализации мы можем судить только на основании спецификации.

♣Для этого принимается реализационная гипотеза, которая называется *гипотезой о безопасности и конвергентности*:

> Если βγδ-трасса безопасна в спецификации и имеется в реализации, то
> безопасность: трасса безопасна в реализации,
> конвергентность: реализация после этой трасса конвергентна, если спецификация после этой трассы конвергентна.

♣Поскольку продолжение безопасной трассы реакцией или стационарностью оставляет трассу безопасной, условие безопасности эквивалентно следующему: после общей трассы, безопасной в спецификации, любой стимул, безопасный в спецификации, безопасен в реализации.

104. Отношение *ioco*$_{\beta\gamma\delta}$.

Соответственно меняется определение соответствия для **ioco**$_{\beta\gamma\delta}$:

> Если трасса безопасна в спецификации и есть в реализации, то после этой трассы реализация может ... тогда и только тогда, когда это возможно в спецификации после этой же трассы. Под многоточием ... понимается теперь одно из четырёх:
> 1) принимать стимул ?x, безопасный после трассы в спецификации;
> 2) блокировать стимул {?x}, безопасный после трассы в спецификации;
> 3) выдавать реакцию !y;
> 4) не выдавать никаких реакций, то есть находиться в стационарном состоянии δ.

105. Спецификация.

Для этого нового соответствия мы можем использовать при пополнении спецификации любое допущение полноты: не только продолжение, но также

блокировку и разрушение. При любом таком допущении проблема самоприменимости спецификации решается автоматически.

♣Для того чтобы тестирование по безопасным трассам спецификации было возможным, требуется 1) чтобы такие трассы были и 2) чтобы после таких трасс не было дивергенции.

Безопасность: Спецификация должна быть безопасной (не саморазрушающейся). В противном случае множество тестов на соответствие этой спецификации пусто, а сама спецификация вырожденная – не предъявляет к реализации никаких функциональных требований. В частности, реализация может быть саморазрушающейся, и при любой попытке тестирования, ещё до начала собственно тестирования, реализация может разрушиться.

Безопасно-конвергентность: Все состояния спецификации, достижимые по безопасным трассам, должны быть конвергентны. Иными словами, если в спецификации есть дивергенция, то она недостижима по безопасным трассам, используемым в тестировании.

106. Генерация тестов.

Генерация тестов для $\boldsymbol{ioco}_{\beta\gamma\delta}$ отличается от генерации тестов для ***ioco*** возможностью блокировки стимулов.

♣Рассмотрим генерацию теста для βγδ-трассы, которая отличается от той, что у нас была для соответствия ***ioco*** заменой одного стимула на его блокировку.

Когда тест в посылающем состоянии выдаёт стимул в реализацию, он устанавливает тайм-аут, а по истечении тайм-аута считает, что реализация блокировала стимул.

♣Пусть некоторый начальный отрезок трассы продолжается в трассе приёмом стимула. В спецификации этот начальный отрезок может иметь продолжение блокировкой стимула (в другом состоянии спецификации, естественно) или продолжаться только приёмом стимула.

В случае разрешения блокировки стимула в посылающем состоянии теста делается переход по блокировке стимула в состояние ***pass,***

♣а в случае, когда стимул может только приниматься, – в состояние ***fail***.

♣Пусть теперь некоторый начальный отрезок трассы продолжается в трассе блокировкой стимула. В спецификации этот начальный отрезок может иметь продолжение приёмом стимула (в другом состоянии спецификации, естественно) или продолжаться только блокировкой стимула.

В случае разрешения приёма стимула в посылающем состоянии теста делается переход по приёму стимула в состояние ***pass***,

♣а в случае, когда стимул может только блокироваться, – в состояние ***fail***.

♣В конце «инвертируем» все символы: приём стимула `?x` заменяется выдачей стимула `!x`, а выдача реакции `!y` заменяется приёмом реакции `?y`. Кроме того, символ δ и блокировки стимулов заменяются символом θ. Интерпретация символа θ зависит от состояния теста: в принимающем состоянии – это стационарность δ, а в состоянии, посылающем стимул, это блокировка этого стимула.

107. Конечность теста и перечислимость полного тестового набора.

Повторим, что для безопасного тестирования спецификация должна быть безопасной и безопасно-конвергентной.

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

♣**Трассовая спецификация**.

Поскольку тест строится по каждой $\beta\gamma\delta$-трассе, которая есть и безопасна в спецификации, число таких трасс должно быть перечислимо. Для конечных трасс это эквивалентно перечислимости множества реакций и безопасных стимулов после каждой безопасной трассы σ. Заметим, что стимулов, которые опасны после трассы, может быть произвольное количество; эти стимулы (так же, как их блокировки) всё равно не используются при построении безопасных трасс.

1) Будем считать, что задан итератор $SI(\sigma)$ – алгоритм перечисления реакций и безопасных стимулов после трассы σ.

Итератор, возвращая безопасный стимул, не сообщает, принимается этот стимул, блокируется или и то и другое (естественно, в разных состояниях).

Заметим, что такой итератор не перечисляет стимулы, которые являются разрушающими после трассы. Такой разрушающий стимул всегда принимается после трассы (с возможным последующим разрушением), но может также блокироваться после трассы.

Для построения теста мы должны уметь после каждой $\beta\gamma\delta$-трассы за конечное время определить, продолжается ли эта трасса в спецификации данной реакцией, стационарностью, безопасным стимулом или его блокировкой. Поэтому множество этих символов должно быть разрешимо.

2) Будем считать, что задан алгоритм $P(\sigma,z)$, который за конечное время проверяет, имеет ли трасса σ продолжение реакций z=!y, безопасным стимулом z=?x или блокировкой безопасного стимула z={?x}.

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

Итератор безопасных $\beta\gamma\delta$-трасс строится так. Для каждой трассы итератор реакций и безопасных стимулов после трассы задаёт нумерацию реакций, безопасных стимулов и их блокировок, которыми трасса может продолжаться. Заметим, что номер символа определяется не только самим символом, но и предыдущей трассой. Если трасса продолжается символом δ, то этому символу присвоим номер 1, а символы, возвращаемые итератором, будем нумеровать, начиная с 2. Если итератор возвращает реакцию, то присвоим ей очередной номер n. Если итератор возвращает безопасный стимул, то трасса продолжается этим стимулом и/или его блокировкой. Если трасса продолжается только стимулом, то присвоим ему очередной номер n; если трасса продолжается только блокировкой стимула, то этой блокировке присвоим номер n; если трасса продолжается и тем и другим, то присвоим стимулу номер n, а его блокировке номер n+1.

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 $P(\sigma,z)$ для проверки:
1) каждой реакции z=!y, получаемой от реализации в принимающем состоянии теста,
2) каждого безопасного стимула z=?x, принимаемого реализацией в посылающем состоянии теста, когда следующий в трассе символ – это блокировка {?x},

3) блокировки каждого безопасного стимула z={?x} в посылающем состоянии теста, когда следующий в трассе символ – это стимул ?x.

Также используем алгоритм $P_\delta(\sigma)$ – для проверки стационарности, то есть истечение тайм-аута в принимающем состоянии теста, когда трасса продолжается реакцией.

♣Автоматная спецификация. Безопасные стимулы (принимаемые и блокируемые).

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

Вместо перечислимости реакций и безопасных стимулов *после безопасной трассы*, мы должны говорить о перечислимости реакций и безопасных стимулов *в состоянии, достижимом по безопасной трассе*.

1) Будем считать, что задан итератор `Sl(s)`, перечисляющий реакции и стимулы, безопасные в состоянии s.

Итератор, возвращая безопасный стимул, не сообщает, принимается этот стимул или блокируется.

Заметим, что такой итератор не перечисляет стимулы, которые являются разрушающими в состоянии. Стимул, разрушающий в состоянии, естественно принимается в этом состоянии, поэтому блокировки этого стимула в этом состоянии быть не может.

Для перечислимости трасс нам было бы достаточно перечислимости переходов по каждой реакции и каждому безопасному стимулу в каждом состоянии.

ОДнако для проверки реакций, безопасных стимулов и их блокировок мы должны уметь алгоритмически проверять, имеется ли в состоянии s переход по данному безопасному символу. Теперь нам нужно за конечное время проверять каждый безопасный символ после трассы, опираясь на проверку безопасных символов 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 стимулу или реакции в каждом состоянии. 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, достижимого по безопасной трассе, и, во-вторых, число состояний достижимых из такого состояния по τ-переходам. Второе условие перекрывается требованием безопасно-конвергентности.

2) Будем считать, что задан алгоритм перебора переходов по реакциям и безопасным стимулам в состоянии – итератор `Tl(s,z)`.

Чтобы проверить реакцию или безопасный стимул z в состоянии s теперь достаточно вызвать итератор переходов по z в состоянии s и проверить, что он возвращает хотя бы один переход. Чтобы проверить блокировку безопасного стимула x в состоянии s теперь достаточно вызвать итератор переходов по x в состоянии s и проверить, что он не возвращает ни одного перехода.

Однако стационарность состояния означает, что в нём нет переходов ни по одной реакции из, быть может, бесконечного множества реакций. Поэтому нам требуется отдельный алгоритм проверки стационарности.

3) Будем считать, что задан алгоритм $P_\delta(s)$, проверяющий, что в состоянии s, достижимом по безопасной трассе, нет переходов по реакциям, то есть стационарность состояния.

4) Стимул, безопасный в одном состоянии после безопасной трассы, может оказаться опасным в другом состоянии после этой же трассы. Поэтому, кроме перечисления стимулов, безопасных в состоянии, нам нужна проверка безопасности стимула в другом состоянии. Должен быть задан алгоритм $P_\gamma(s,x)$, проверяющий безопасность стимула x в состоянии s, достижимом по безопасной трассе.

Итерация безопасных символов после безопасной трассы сводится к их итерации реакций и безопасных стимулов 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, и безопасен ли он во всём множестве состояний. Стимул безопасен во множестве состояний, если он безопасен в каждом состоянии этого множества. Если символ, возвращаемый итератором, уже был, или оказался опасным, 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. Заметим, что, выбирая безопасный стимул, мы должны дополнительно проверить, принимается он в данном состоянии или блокируется. Это делается с помощью итератора переходов. Если стимул блокируется,

трасса может продолжаться блокировкой этого стимула. После такого продолжения мы переходим к подмножеству состояний, в которых этот стимул блокируется. 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.

Рассмотренный способ задания автомата весьма специфический: явно указываются (специфицируются) не только принимаемые безопасные стимулы, но и блокировки безопасных стимулов. В то же время для разрушающих стимулов указывается только тот факт, что они разрушающие. Куда ведут переходы по разрушающему стимулу, несущественно, важно лишь, что из постсостояния хотя бы одного перехода по данному стимулу, двигаясь по τ-переходам и переходам по реакциям, можно достигнуть разрушения. Тем не менее, такой способ задания спецификационного автомата полезен на практике.

♣Автоматная спецификация. Принимаемые стимулы (безопасные и разрушающие).

Теперь мы рассмотрим альтернативный (и для нашей модели теоретически более естественный) способ, когда явно задаются (специфицируются) только переходы по стимулам и реакциям, определённые в состоянии, в том числе по разрушающим стимулам, а блокировки – это блокировки всех тех стимулов, переходы по которым не заданы, то есть стимулов, неспецифицированных в состоянии.

В этом случае все неспецифицированные стимулы блокируются в данном состоянии и, тем самым, безопасны в нём. Для специфицированного стимула мы должны проверить его безопасность. Таким образом, общее число стимулов, как безопасных в состоянии, так и разрушающих в состоянии, должно быть перечислимо.

♣*Стимул опасен в состоянии*, если из постсостояния хотя бы одного перехода по данному стимулу, двигаясь по τ-переходам и переходам по реакциям, можно достигнуть разрушения, то есть γ-перехода.

♣*Гамма-состоянием* будем называть состояние, в котором определён такой переход по разрушению.

♣Для того чтобы можно было проверить безопасность стимула в состоянии, требуется конечность числа состояний, которые можно достигнуть из каждого состояния, достижимого по безопасной трассе + стимул, двигаясь далее по маршруту из τ-переходов и переходов по реакциям. Для каждого такого маршрута мы проверяем, не заканчивается ли он в гамма-состоянии.

Поскольку нас не интересуют маршруты, которые проходят через гамма-состояния и двигаются дальше, это требование можно ослабить: нужно учитывать только те маршруты, которые не проходят через гамма-состояния, но могут заканчиваться в гамма-состояниях.

Это свойство можно назвать *регулярностью по реакциям* по аналогии с регулярными множествами последовательностей, порождаемыми конечными автоматами. В частности, число реакций, по которым определены переходы из данного состояния, должно быть конечным.

♣Таким образом, мы должны иметь два итератора: итератор `Xl(s)` перечисляет не более чем счётное множество стимулов, определённых в состоянии, а итератор `Yl(s)` перечисляет конечное множество реакций, определённых в состоянии.

Итератор переходов `Tl(s,z)` по-прежнему требуется.

Поскольку число реакций в состоянии конечно, нам не требуется отдельный алгоритм проверки стационарности состояния. Для такой проверки достаточно итератором переходов проверить отсутствие τ-переходов, а итератором реакций – отсутствие реакций.

Для проверки безопасности стимула в конечном множестве состояний U нам теперь не нужен специальный алгоритм $P_\gamma(s,x)$. Вместо этого мы вычисляем с помощью итератора реакций и итератора переходов конечное множество состояний, достижимых из состояний множества U по τ-переходам и переходам по реакциям. Нам нужно лишь проверить наличие или отсутствие γ-перехода в каждом из этих состояний. Для этого достаточно, чтобы алгоритм `Tl(s,`γ`)` мог перечислять также все переходы из состояния s по разрушению γ, хотя нам достаточно знать только, пусто или не пусто это множество переходов.

♣**Автоматная спецификация. Конечно-безопасно-ветвящаяся.**

Как частный широко распространённый случай спецификаций, для которых выполнены последние условия, можно рассматривать спецификации, которые:

1. Конечно-безопасно-ветвящиеся, то есть такие, в каждом состоянии которых, достижимом по безопасной трассе, определено конечное число переходов. А в каждом состоянии, достижимом по безопасной трассе + стимул, определено конечное число τ-переходов и переходов по реакциям. И задан итератор TI(s) всех переходов в таких состояниях.
2. Регулярны по реакциям.
3. Имеют перечислимое множество стимулов и задан итератор стимулов X.

108. βγδ-трассы: интерфейс теста и медиатора.

Теперь мы сняли запрет блокирующего deadlock`a, и поэтому меняется интерфейс теста и медиатора.

♣Здесь добавляется θ-переход в медиаторе при попытке передачи стимула в реализацию. Если происходит блокировка, медиатор сообщает об этом тесту.

109. Асинхронный автомат: Пополнение спецификаций.

110. Переопределение блокировок стимулов.

В самом общем виде пополнение спецификаций определяется как отображение множества спецификаций в себя. Соответственно отношение **ioco** связывает реализацию не с исходной, а с пополненной спецификацией.

♣Мы говорили, что пополнение спецификаций следует понимать как переопределение всех или некоторых блокировок стимулов. Поэтому пополнение спецификаций как отображение должно удовлетворять следующим двум требованиям:

1. Старые βγδ-трассы без блокировок сохраняются.
2. Добавляются только такие βγδ-трассы, которые имеют вид μ?xλ, где βγδ-трасса μ была раньше и заканчивалась хотя бы в одном состоянии, в котором не был специфицирован стимул x (была трасса μ{?x}).

♣Различают два вида пополнений: 1) пополнение состояний и 2) пополнение трасс.

При пополнении состояний переопределяются блокировки стимулов в каждом отдельно взятом состоянии.

При пополнении трасс переопределяются блокировки стимулов, которыми может непосредственно продолжаться каждая отдельно взятая трасса. Можно показать, что если спецификационный автомат нужным образом преобразовать, то пополнение состояний преобразованного автомата эквивалентно пополнению трасс исходного автомата. Поэтому дальше мы будем рассматривать пополнения состояний.

111. Основные виды пополнений.

Рассмотрим несколько видов пополнения спецификации.

♣Для первого типа допущения полноты, когда стимул принимается без разрушения, пополнение определяется дальнейшим, после приёма стимула, поведением автомата.

Автомат может после приёма стимула выдать специальное *сообщение об ошибке*, то есть реакцию, и перейти в специальное ошибочное состояние, игнорируя в дальнейшем все стимулы.

♣Если автомат просто игнорирует принимаемый стимул, сохраняя своё состояние, то такое пополнение называют *ангельским*.

♣*Демоническое* пополнение определяет после приёма стимула произвольное поведение, то есть любую последовательность принимаемых стимулов, выдаваемых реакций и стационарности, но без блокировок стимулов и без разрушения.

♣Второй тип допущения полноты сохраняет блокировку стимула. Фактически, соответствующее пополнение спецификации – тождественное преобразование, ничего не меняющее.

♣Третий тип допущения полноты – разрушение после приёма стимула. Оно моделируется γ-переходом.

♣Кроме этих основных видов пополнений, рассматриваются также различные комбинированные варианты. Блокировка стимула переопределяется в зависимости от вида состояния.

112. Классическая семантика отношения *ioco*. Проблема самоприменимости спецификации.

Теперь мы посмотрим, какое пополнение спецификации сохраняет классическую семантику отношения **`ioco`** (не ***ioco***$_{\beta\gamma\delta}$, а просто ***ioco***). Прежде всего, заметим, что такое пополнение должно удовлетворять двум правилам.

♣Первое правило – это правило приоритета приёма стимула над его блокировкой. Пусть в спецификации трасса μ непосредственно продолжается как приёмом стимула x, так и его блокировкой (естественно, в разных состояниях). Тогда, по определению отношения ***ioco***, реализация, в которой такая трасса μ есть, должна после этой трассы μ принимать стимул `x`. Таким

образом, все продолжения трассы μ блокировкой стимула x, которые есть в спецификации, должны быть удалены.

♣Второе правило – это правило сохранения поведения после стационарности. Пусть в спецификации трасса μ непосредственно продолжается как приёмом стимула x, так и стационарностью, но после стационарности не продолжается стимулом x. Очевидно, любое продолжение после трассы $\mu\cdot\delta$ должно быть и после трассы μ: переход по δ – это петля. Назовём это *законом петли*.

После пополнения трасса $\mu\cdot\delta$ не может продолжаться блокировкой $\{x\}$, так как в противном случае, по закону петли, трасса μ также продолжалась бы блокировкой $\{x\}$, что противоречит правилу приоритета. Значит, после пополнения трасса $\mu\cdot\delta$ продолжается стимулом x, то есть после трассы $\mu\cdot\delta$ добавляются продолжения вида $x\cdot\lambda$.

Любое добавленное продолжение $x\cdot\lambda$ после трассы $\mu\cdot\delta$, по закону петли, должно быть также после трассы μ. Но тогда это продолжение $x\cdot\lambda$ было после трассы μ в исходной спецификации, так как иначе пополнение ослабит соответствие: разрешит продолжение $x\cdot\lambda$ после трассы μ, которое раньше не разрешалось.

Наконец, если в исходной спецификации после трассы μ было продолжение $x\cdot\lambda$, то после пополнения оно должно быть после $\mu\cdot\delta$. В противном случае произошло бы усиление соответствия. Ведь до пополнения после трассы $\mu\cdot\delta$ в конформной реализации могло быть продолжение $x\cdot\lambda$, поскольку вообще не проверялось, может ли реализация после трассы $\mu\cdot\delta$ принимать стимул x, а, если может, то какое дальнейшее поведение допустимо, а какое нет.

Итак, правило сохранения поведения после стационарности звучит так: если трасса μ продолжается как стимулом x, так и стационарностью δ, а трасса $\mu\cdot\delta$ не продолжается стимулом x, то после пополнения трасса $\mu\cdot\delta$ не продолжается блокировкой $\{x\}$ и продолжается теми и только теми трассами вида $x\cdot\lambda$, которыми до пополнения продолжалась трасса μ.

Теперь мы можем пополнить спецификацию в соответствии с этими двумя правилами.

♣Это пополнение решает проблему самоприменимости спецификации. Теперь, если стимул принимается в одном состоянии после трассы, то он

принимается в каждом состоянии после трассы, и, если стимул блокируется в одном состоянии после трассы, то он блокируется в каждом состоянии после трассы. Все состояния, в которых заканчивается трасса, имеют одно и то же множество блокируемых стимулов и одно и то же множество принимаемых стимулов.

113. Классическая семантика отношения *ioco*. Проблема несохранения соответствия.

Для решения проблемы сохранения соответствия мы должны продолжить пополнение спецификации. Дело в том, что если в исходной спецификации трасса во всех своих конечных состояниях не продолжалась стимулом, а только его блокировкой, то это так и останется. Поскольку классическая семантика **`ioco`** не проверяет блокировку, мы должны эту блокировку заменить приёмом стимула с соответствующим продолжением. Таким продолжением может быть либо произвольное продолжение без блокировок и разрушения, то есть демоническое пополнение, либо продолжение разрушением, то есть пополнение разрушения.

Для всюду определённых по стимулам реализаций эти два пополнения эквивалентны с точки зрения соответствия. Реализация конформна демонически пополненной спецификации тогда и только тогда, когда она конформна спецификации, пополненной разрушением. При тестировании это проявляется в том, что в первом случае давать стимул бессмысленно, а во втором случае стимул давать нельзя.

♣Однако пополнение разрушения имеет ряд преимуществ по сравнению с демоническим пополнением.

1) Пополнение разрушения не усиливает соответствие для не всюду определённых реализаций, поскольку стимул, который нельзя давать в реализацию, реализация может блокировать. Демоническое пополнение запрещает такую блокировку.

   Конечно, здесь мы уже используем модифицированное соответствие: не ***ioco***, а $\boldsymbol{ioco}_{\beta\gamma\delta}$. Поскольку блокировок у нас нет, это соответствие можно назвать $\boldsymbol{ioco}_{\gamma\delta}$.

2) Демоническое пополнение решает проблему несохранения соответствия при переходе к асинхронному тестированию, но при этом теряется информация о том, что стимул бессмысленно давать в реализацию. При пополнении разрушения стимул остаётся разрушающим, мы не будем его давать в реализацию и при асинхронном тестировании тоже.

Для решения проблемы несохранения соответствия нужно, чтобы синхронное и асинхронное тестирование были одинаковыми: одинаково строгими, то есть проверяющими все стимулы, или одинаково слабыми, то есть не проверяющими одни и те же неспецифицированные стимулы. Демоническое пополнение предлагает в обоих случаях строгое тестирование, а пополнение разрушения – слабое тестирование. Что, конечно, лучше.

3) Семантика разрушения вообще шире семантики произвольного поведения, поскольку допускает в реализации не только приём стимула с любой последующей последовательностью приёма стимулов, выдачи реакций и стационарности, но также блокировку стимула и разрушение реализации после приёма стимула. Более того, после приёма разрушающего стимула допускается даже дивергенция.

114. Комбинированный вариант группы RedVerst.

В группе RedVerst предлагается комбинированный вариант пополнения: в нестационарных состояниях, в которых реализация может иметь внутреннюю активность или готова выдать реакции, неспецифицированный стимул не принимается, а в стационарных состояниях – разрушает реализацию. При асинхронном тестировании с входной очередью стимулов и выходной очередью реакций блокировка стимула в нестационарном состоянии оказывается временной: приём стимула из входной очереди откладывается до перехода системы в стационарное состояние.

1♣Сейчас мы рассмотрим пример системы и её асинхронного тестирования, имея в виду, что среда передачи – это входная и выходная очереди. Система выполняет в цикле различные виды работ. Каждая работа выполняется в два этапа. Первый этап инициируется в начальном состоянии 1 командой, то есть стимулом, *начало*. Эта команда может иметь различные параметры, определяющие вид работы. Это означает, что, фактически, может быть несколько стимулов, которые мы называем командой *начало*. Для нашего примера мы эти реализационные стимулы не различаем. Если всё хорошо, система проходит ряд нестационарных состояний 2,3,4, они отмечены красным цветом, выдаёт реакцию *ok* и попадает в стационарное состояние 5,

2♣после чего можно инициировать второй этап работы командой *конец*.

3♣В процессе выполнения первого этапа у системы могут возникнуть проблемы, требующие дальнейших указаний. В этом случае она не выдаёт реакцию *ok* и переходит в стационарное состояние 6. Мы намеренно не сделали в системе выдачу реакции, чтобы показать необходимость при тестировании проверять не только наличие реакции, но и отсутствие реакций, то есть стационарность.

4♣После этой стационарности можно командой *дальше* продолжить выполнение работы. Эта команда также может иметь различные параметры, которыми мы управляем дальнейшим выполнением работы. Это означает, что, фактически, может быть несколько стимулов, которые мы называем командой *дальше*. После этой команды уже можно не ждать реакцию *ok* и сразу давать команду *конец*.

5♣Кроме этого, предоставляется возможность в любой момент времени после того, как первый этап инициирован, но до перехода ко второму этапу, приостановить выполнение работы командой *стоп*. Этого нельзя делать до начала работы или после команды *конец*, то есть в состоянии 1 команда *стоп* разрушающая.

Из трёх нестационарных состояний 2, 3 и 4 команда *стоп* может приниматься только в состояниях 2 и 4. Отметим, что в рассматриваемой модели асинхронного автомата система не обязана принимать стимул в нестабильном состоянии 2, поскольку ей разрешается асинхронно выполнять внутренний переход. При асинхронном тестировании система также не обязана принимать стимул в стабильном, но нестационарном, состоянии 4, поскольку выходная очередь всегда готова принимать реакции.

Если команда *стоп* принимается в состоянии 2, система может отменить выполнение работы, очищая все её следы, и становится готовой к выполнению следующей работы.

6♣Об этом система извещает реакцией *откат*. Теперь у нас появляется нестационарное состояние 7, в котором, однако, команду *стоп* давать нельзя.

7♣Если команда *стоп* приходит в систему позже, в состоянии 4, то система может перейти в стационарное состояние 6, ожидая дальнейших указаний,

8♣то есть команду *дальше*.

Команда *стоп* может поступить и тогда, когда система находится в состоянии 3. Однако приём команды откладывается до перехода в стационарное состояние, в котором команда всегда принимается, или нестационарное состояние, в котором она может приниматься. В данном случае это нестационарное состояние 4, которое мы уже рассматривали.

9♣ Если команда *стоп* откладывается до стационарного состояния 6, то в нём она принимается, но состояние не меняется. После этого можно продолжить командой *дальше*.

10♣ Если команда *стоп* откладывается до стационарного состояния 5, с предварительной выдачей реакции *ok*, то в нём она также принимается без изменения состояния. После этого можно продолжить командой *дальше* или перейти ко второму этапу командой *конец*.

11♣ Теперь посмотрим, какие трассы мы можем получить, учитывая безопасность тестирования.

12♣ Множество безопасных трасс описывается следующим регулярным выражением: (A|B|C)*.

С самого начала мы можем давать только команду *начало*, поскольку начальное состояние стационарное и, следовательно, все остальные команды, которые не определены в этом состоянии, могут привести к разрушению реализации.

Вариант A: Сразу после команды *начало*, мы можем безопасно давать только команду *стоп*, поскольку она определена во всех стационарных состояниях, достижимых по τ-переходом и переходам по реакциям после команды *начало*. Любая другая команда этим свойством не обладает.

После команды *стоп* мы можем получить реакцию *откат*, и на этом цикл завершается.

Вариант B: Реакцию *ok* мы можем получить независимо от того, давали мы команду *стоп* или нет. После реакции ok мы можем сколько угодно раз давать команду *стоп* и, принимая реакции,

обнаруживать стационарность. Затем, давая команду конец, завершаем цикл.

Вариант C: Отсутствие реакций, то есть стационарность, мы также можем получить после начала работы независимо от того, давали мы команду стоп или нет. После этого мы также можем сколько угодно раз давать команду стоп и, принимая реакции, обнаруживать стационарность. Но, в отличие от реакции ok, после этого мы можем давать не команду конец, которая запрещена в стационарном состоянии 6, а команду дальше. После этого мы опять можем сколько угодно раз давать команду стоп и, принимая реакции, обнаруживать стационарность, а затем, давая команду конец, завершаем цикл.

Этот пример демонстрирует следующие важные случаи:

1) Некоторые стимулы можно безопасно давать в стационарном состоянии, а некоторые нет. В нашем примере в начальном состоянии можно давать только команду начало.
2) Некоторые стимулы, которые нельзя давать вначале, можно давать после других стимулов. Команду стоп можно давать после команды начало.
3) Некоторые стимулы можно давать только после некоторых реакций или их отсутствия. Команду дальше вслед за командой начало и, быть может, последующей командой стоп можно давать после стационарности. После команды дальше можно давать команду конец, но её можно давать и после реакции ok.

13♣ В целом безопасность стимула x определяется предшествующей трассой σ.

14♣ Пусть такая трасса σ, начинающаяся с начального состояния асинхронного автомата, может быть продолжена такой последовательностью реакций (в частности, пустой), после которой мы можем оказаться в стационарном состоянии, где стимул x не определён.

15♣ Тогда, в соответствии с нашим допущением полноты, в таком стационарном состоянии стимул x понимается как разрушающий . Такой стимул x опасен, и его нельзя давать при тестировании после трассы σ.

16♣ В противном случае, если при любом, возможном в автомате, продолжении трассы σ последовательностью реакций мы можем оказаться

только в тех стационарных состояниях, в которых стимул x определён, то он безопасен и его можно при тестировании давать после трассы σ.

115. Асинхронный автомат: Спецификация в пред- и постусловиях.

116. Пред- и постусловия.

Мы рассматривали спецификацию автоматов Мили в пред- и постусловиях. Теперь посмотрим такую спецификацию для асинхронных автоматов общего вида.

В отличие от автоматов Мили реакция асинхронного автомата, вообще говоря, не связана жёстко со стимулом: после стимула может не быть реакций, или быть несколько реакций. В общем случае мы уже не можем рассматривать реакцию как ответную на тот или иной стимул. В целом допустимость стимула или реакции определяется состоянием автомата. Поэтому стимулы и реакции специфицируются раздельно.

♣Предусловие стимула – это предикат от пресостояния и стимула: `PRE(s,x)`. Постусловие стимула – это предикат от пресостояния, стимула и постсостояния: `POST(s,x,s`)`. Соответственно, предусловие реакции – это предикат от пресостояния и реакции: `PRE(s,y)`. Постусловие реакции – это предикат от пресостояния, реакции и постсостояния: `POST(s,y,s`)`.

Мы видим, что пред- и постусловия стимула и реакции определяются полностью аналогично. Однако, их интерпретация различна.

Посмотрим, что означает предусловие стимула и реакции для разработчика реализации и для пользователя, создающего программу, которая будет обращаться к реализации. Поскольку при тестировании тест подменяет собою окружение (или его часть), точка зрения тестировщика, фактически, совпадает с точкой зрения пользователя.

♣Для разработчика предусловие стимула означает: только для тех стимулов, которые удовлетворяют предусловию стимулов, нужно реализовать поведение, требуемое постусловием стимула. Если стимул не удовлетворяет предусловию, можно считать, что такой стимул в реализацию не поступает; во всяком случае, разработчик может не беспокоиться о том, что случиться, если такой стимул всё-таки поступит. Для пользователя предусловие стимула определяет, когда он может обращаться к реализации, а когда нет. Иными словами, предусловие стимула – это требование к пользователю.

♣Предусловие реакции, наоборот, для реализатора определяет, когда он может выдавать такую реакцию, а когда нет. Пользователь может рассчитывать, что он получит только такую реакцию, которая удовлетворяет предусловию реакции. Во всяком случае, он не должен беспокоиться, что произойдёт, если такая реакция всё-таки поступит от реализации. Иными словами, предусловие реакции – это требование к реализатору.

♣Кроме переходов по стимулам и реакциям, в асинхронном автомате могут быть τ-переходы. Для них предусловие – это предикат от пресостояния `PRE(s)`, а постусловие – предикат от пресостояния и постсостояния `PRE(s,s`)`.

117. Спецификация без τ-переходов.

Может показаться странным, что в спецификациях UniTesK нет τ-переходов.

На самом деле, во многих случаях спецификация с τ-переходами может быть заменена спецификацией без τ-переходов, эквивалентной ей, по крайней мере, в смысле отношения $\boldsymbol{ioco}_{\beta\gamma\delta}$. Действительно, τ-переход из s в s` не наблюдаем при тестировании. После стимула или реакции, наблюдаемых при переходе в состояние s, может следовать любой стимул или реакция, наблюдаемые при переходе из состояния s`, или любого другого состояния, достижимого из состояния s` по τ-переходам.

♣Поэтому вместо τ-перехода из состояния s в состояние s` мы можем все такие наблюдаемые переходы сдублировать из состояния `s`.

118. Проблема ложной стационарности.

Проблема может возникнуть только со стационарностью.

♣Если из состояния s и состояний, достижимых из s по τ-переходам, вели только переходы по стимулам, то после удаления τ-переходов состояние s становится стационарным.

♣Пусть при этом некоторый стимул ?z был определён в s, но не был определён ни в одном из стационарных состояний, достижимых из s.

♣Тогда до преобразования в состоянии s не было трассы δ?z, а после преобразования такая трасса появляется.

Таким образом, удаление из спецификации τ-переходов изменяет соответствие ***ioco***. Точнее, строит автомат, который не эквивалентен по отношению ***ioco*** исходной спецификации.

♣Другое дело, что при нашем допущении полноты этой проблемы не возникает. Напомню, что мы интерпретируем неспецифицированный стимул в зависимости от состояния: в стационарном состоянии он разрушающий, а в нестационарном – блокируемый. В этом случае при безопасном тестировании мы никогда не будем давать такой стимул z, который не принимается в некотором стационарном состоянии, достижимом из текущего состояния по τ-переходам. Поэтому, при удалении τ-переходов перед безопасным стимулом не может возникнуть стационарность, если её не было до преобразования.

119. Спецификация разрушения.

При нашем допущении полноты разрушающий стимул – это стимул, нарушающий предусловие в стационарном состоянии. Тем самым разрушение специфицировано неявно.

Однако в некоторых случаях полезно уметь явно специфицировать разрушение.

Спецификация перехода по разрушению аналогична спецификации τ-перехода. Предусловие – это предикат от пресостояния, а постусловие – это предикат от пресостояния и постсостояния.

120. Блокировка в стационарных состояниях.

Проблема ложной стационарности остаётся, если использовать иные допущения полноты. Например, наше допущение полноты можно обобщить. А именно: разрешить некоторые стимулы, неспецифицированные в стационарном состоянии, понимать не как разрушающие, а как блокируемые. Тогда может оказаться, что стимул принимается без последующего разрушения в нестационарном состоянии, но блокируется во всех стационарных состояниях, достижимых из данного состояния по τ-

переходам. При удалении τ-переходов возникнет ложная стационарность перед этим стимулом.

Это обобщение нашего допущения полноты практически полезно. Оно не обязательно вызывает глухой deadlock при асинхронном тестировании. Например, если среда передачи содержит не одну, а две параллельные входные очереди стимулов, автомат может в стационарном состоянии блокировать все или некоторые стимулы из одной очереди и принимать все стимулы из другой очереди. Такое асинхронное тестирование мы подробнее рассмотрим позже (в разделе о гипертестировании). А сейчас посмотрим, как может выглядеть спецификация при таком обобщённом допущении полноты.

Проблема специфицирования заключается в том, что в данном состоянии (при нашем обобщённом допущении полноты – в данном стационарном состоянии) мы должны некоторые стимулы интерпретировать как блокируемые, а некоторые – как разрушающие. Как это описать?

Существуют два варианта такого описания. Они отличаются тем, как понимаются стимулы, удовлетворяющие предусловию для стационарного состояния: как принимаемые или как безопасные.

В первом варианте (предусловие описывает стимулы принимаемые в стационарном состоянии) для указания разрушающего стимула мы могли бы в постусловии использовать ключевое слово **swap**. Однако более общим является явная спецификация не разрушающего стимула, а перехода по разрушению, которую мы рассмотрели на предыдущем слайде.

♣Во втором варианте (предусловие описывает стимулы безопасные в стационарном состоянии) предлагается предусловие стимула понимать как предикат, выделяющий все безопасные в данном состоянии стимулы: как принимаемые в этом состоянии, так и блокируемые в нём.

Тогда в постусловии для блокируемого стимула должно быть явно указано, что он блокируется или принимается.

Для нестационарного состояния стимул, нарушающий предусловие, – всегда блокируемый.

♣ Если стимул принимается, то после ключевого слова ***then*** (или аналогичной конструкции в языке спецификации) описывается предикат от пресостояния, постсостояния и стимула. Этот предикат описывает приём стимула, фактически, неявно указывая возможные постсостояния, поскольку

к этому моменту возможные стимул и пресостояние известны (они выделены предусловием и условие branch`a).

♣Для указания не приёма, а блокировки стимула предлагается использовать вместо такого предиката специальное ключевое слово ***block***.

♣В целом выполнение требований постусловия записывается как исключающая дизъюнкция предиката от пресостояния, стимула и постсостояния для принимаемых безопасных стимулов и предиката от пресостояния и стимула для блокируемых безопасных стимулов.

♣Предусловие реакции задаёт множество реакций, которые должны перечисляться для данного состояния итератором реакций и безопасных стимулов. Предусловие стимула теперь задаёт множество безопасных стимулов, которые должны перечисляться этим итератором, причём какие-то из этих стимулов принимаются в данном состоянии, а какие-то блокируются. Именно это множество стимулов должно быть перечислимым.

Таким образом, итератор символов SI(s) перечисляет реакции и стимулы, удовлетворяющие предусловиям реакций и стимулов, соответственно, для данного состояния s.

Итератор переходов TI(s,z) перечисляет постсостояния, которые возможны для пресостояния s и реакции !y, если они удовлетворяют предусловию реакций, или для пресостояния s и стимула ?x, если они удовлетворяют предусловию стимулов, а в постусловии для этой пары не написано ***block***.

Стабильность состояния s проверяется по спецификации τ-переходов: состояние s не удовлетворяет предусловию этой спецификации.

Алгоритм $P_\delta(s)$ проверяет, что ни одна реакция не удовлетворяет предусловию реакций для данного стабильного состояния s.

Дополнение множества безопасных стимулов – это множество стимулов, разрушающих в данном состоянии. Поскольку нам требуется разрешимость множества безопасных стимулов, множество разрушающих стимулов также должно быть перечислимо.

Алгоритм $P_\gamma(s,x)$ проверяет, что стимул x разрушающий для состояния s, то есть не удовлетворяет предусловию стимула для состояния s.

121. Привязанные реакции.

Частный, но широко распространённый, случай реакций в асинхронном автомате – это, так называемые, *привязанные* реакции. Такая реакция всегда является ответом на некоторый стимул, но она может выдаваться реализацией не обязательно сразу после приёма стимула.

♣Реализация может принять несколько стимулов, а потом уже выдавать реакции на них в том же порядке

♣или в порядке, отличном от порядка стимулов.

Концептуально привязанные реакции – это способ сокращённой записи. Привязку реакции к стимулу всегда можно отобразить через состояния автомата.

Привязанную реакцию естественно специфицировать вместе со спецификацией стимула. Такое локальное описание имеет ряд преимуществ. Во-первых, реализуется понятие привязанности: мы понимаем, что после стимула мы должны получить одну из привязанных реакций, хотя, в отличие от автомата Мили, не обязательно немедленно. Во-вторых, имена привязанных реакций могут быть локальными, они также привязываются к имени стимула. При взаимодействии реализации с окружением это реализуется параметром реакции, указывающим тот стимул, на который выдаётся эта реакция.

♣Можно также считать, что стимул, кроме привязанных реакций, может иметь немедленную реакцию, которая должна поступить сразу после стимула.

♣Это своеобразная переходная форма от автомата Мили к общему случаю асинхронного автомата.

♣Кроме того, можно разрешить выдачу нескольких привязанных реакций в ответ на один стимул. Правда, если мы хотим уметь определять окончание передачи реакций в ответ на стимул, требуется указание, какая из реакций является последней. Это похоже на вход отстройки в кластерной технологии программирования.

122. АСИНХРОННОЕ ТЕСТИРОВАНИЕ

В отличие от автоматов Мили для асинхронных автоматов общего вида имеется существенное различие между синхронным и асинхронным тестированием.

123. Асинхронное тестирование: Стационарное

124. Ограничение на среду передачи.

Мы будем рассматривать стационарное тестирование только для среды передачи, которая представляет собой одну неограниченную очередь стимулов и одну неограниченную очередь реакций.

125. Осцилляция.

Идея стационарного тестирования заключается в том, чтобы давать стимулы только в стационарных состояниях. В таких состояниях ничего не меняется: автомат стоит и ждёт стимулов.

Получив стимул, автомат может выдать последовательность реакций. Он двигается по некоторой цепочке переходов, начинающейся в стационарном состоянии с перехода по стимулу и далее состоящей из внутренних переходов и переходов по реакциям. Заканчивается такая цепочка в стационарном состоянии, где можно давать следующий стимул.

♣Проблема возникает, если автомат может выдать бесконечную последовательность реакций. В этом случае говорят, что автомат *осциллирует*.

♣Бесконечная последовательность реакций может выдаваться в цикле из τ-переходов и переходов по реакциям.

♣Осцилляция создаёт две проблемы при асинхронном тестировании: проблему дивергенции и проблему шага тестирования.

♣Проблема дивергенции. Если среда передачи всегда готова принимать реакции от реализации, то в цикле осцилляции реализации мистический синхронизатор может всегда выбирать синхронный переход по передаче реакции из реализации в среду. В результате в композиции среды и реализации возникает бесконечная цепочка τ-переходов, то есть дивергенция. Иными словами, среда всё время принимает реакции, но не выдаёт их в тест, который этих реакций ждёт. Истечение тайм-аута в тесте в этом случае не обязательно означает стационарность. Примером такой среды может служить среда с неограниченной выходной очередью.

Если в среде нет бесконечной цепочки τ-переходов и переходов по приему реакций из реализации, то композиция реализации и среды будет конвергентной. Примером может служить среда с ограниченной выходной очередью.

♣Проблема шага тестирования. Шаг тестирования, который мы будем рассматривать, представляет собой последовательность посылаемых стимулов и принимаемых реакций, которая имеет конечную длину и завершается стационарностью, когда всё «успокаивается».

Однако осцилляция реализации может привести к тому, что тест будет всё время принимать реакции, но так и не дойдёт до стационарного состояния реализации. Проблема в том, что мы не имеем возможности столкнуть реализацию с цикла осцилляции даже в том случае, когда в этом цикле определены также переходы по приёму стимулов. Мы не можем заставить реализацию принять стимул, поскольку приём стимулов не имеет большего приоритета, чем выдача реакций. К проблеме осцилляции и приоритетов переходов мы ещё вернёмся.

♣А пока первое требование, которое предъявляется к реализации при стационарном тестировании, – реализация должна быть неосциллирующей.

126. Стационарный автомат.

Стационарное тестирование неосциллирующего асинхронного автомата очень похоже на тестирование автомата Мили. В обоих случаях мы даём один стимул, а потом ждём реакций. Отличие в том, что в асинхронном автомате мы можем получить не одну реакцию, а конечную, в частности, пустую последовательность реакций, заканчивающуюся символом стационарности δ.

♣Эта аналогия может быть выражена вполне формально: преобразованием асинхронного автомата в автомат Мили, который можно назвать стационарным автоматом. Состояния стационарного автомата – это стационарные состояния исходного автомата. Стимулы стационарного автомата – это стимулы исходного асинхронного автомата, а реакции – это конечные последовательности реакций исходного автомата, заканчивающиеся символом δ. В стационарном автомате определяется переход $s \text{—} x,(y_1,y_2,\ldots y_n,\delta) \rightarrow s`$, если в исходном автомате состояния $s$ и $s`$ стационарны и имеется цепочка переходов из s в $s`$ с трассой $x,y_1,y_2,\ldots y_n$.

♣Небольшая проблема возникает, если начальное состояние автомата нестационарно. Тогда первый шаг тестирования отличается от остальных шагов тем, что начинается в нестационарном состоянии. Поэтому на этом шаге мы не посылаем стимул в реализацию, а только принимаем реакции от неё. Для построения стационарного автомата нужно ввести фиктивный пустой стимул и на переходе из начального состояния написать пару (пустой стимул, последовательность реакций). Пустой стимул – это абстракция, смысл которой в том, что мы не посылаем никакого стимула. В дальнейшем, для простоты, мы не будем специально оговаривать этот особый случай.

Все требования конечности и детерминированности, которые мы предъявляли к автомату Мили для того, чтобы тот или иной метод тестирования был возможен, теперь при стационарном тестировании предъявляются к стационарному автомату. Тестирование с открытым состоянием предполагает теперь открытость стационарных состояний асинхронного автомата. Иными словами, задача сводится к предыдущей.

127. Работа тестовой системы.

Несколько слов нужно сказать о том, каким образом при стационарном тестировании работает тестовый оракул и генератор постсостояний.

♣Пусть мы в стационарном состоянии s даём стимул x.

♣Сначала генератор постсостояния определяет постсостояние s_1 для перехода из стационарного состояния s по стимулу x.

♣Тестовый оракул проверяет, что тройка (s, x, s_1) удовлетворяет постусловию стимула.

♣Затем генератор постсостояния определяет постсостояние s_2 для перехода из состояния s_1 по первой реакции y_1, а тестовый оракул проверяет, что тройка (s_1, y_1, s_2) удовлетворяет постусловию реакции.

♣Далее такая процедура повторяется для реакций $y_2, y_3, \ldots y_n$.

♣В конце проверяем, что полученное постсостояние s_{n+1} стационарно, то есть для любой реакции y предусловие реакции $PRE(s_{n+1}, y)$ ложно.

128. Недетерминизм.

Конечно, может оказаться, что постсостояние s_i определяется неоднозначно.

♣Если генератор постсостояния способен перечислять возможные постсостояния, то мы сможем строить несколько цепочек переходов. Некоторые из них мы отфильтруем, если на каком-то шаге их отбраковывает тестовый оракул.

♣♣

♣Ошибка фиксируется только в том случае, когда ни одна из цепочек не годится.

♣Если годится хотя бы одна цепочка, ошибки нет.

♣Если годится несколько цепочек, то возможен недетерминизм, если эти цепочки заканчиваются в разных постсостояниях.

♣Тем не менее, само по себе наличие нескольких цепочек не обязательно означает недетерминизм. В асинхронном автомате может быть несколько цепочек переходов для одного и того же стимула и одной и той же последовательности реакций, заканчивающихся, тем не менее, в одном

стационарном состоянии. Если для разных реакций постсостояние окажется разным, стационарный автомат будет слабо детерминирован.

♣Если все цепочки для данного стимула заканчиваются в одном состоянии, независимо от последовательности реакций, то стационарный автомат будет сильно детерминирован.

♣Заметим, что исходный автомат может быть недетерминированным.

129. Перечисление в состоянии спецификации.

Для верификации наблюдаемой трассы нам нужно уметь перечислять стимулы и постсостояния для переходов, определённых в состоянии спецификации, достижимом по наблюдаемой трассе.

Поскольку тестирование должно быть безопасным, нам нужно проверять безопасность стимулов, которые мы посылаем в реализацию в стационарных состояниях.

♣Если предусловие спецификации описывает безопасные стимулы (как принимаемые, так и блокируемые), то стимул безопасен в стационарном состоянии, если он в нём специфицирован.

♣Однако, если предусловие описывает принимаемые стимулы (как безопасные, так и разрушающие), то для проверки безопасности стимула нам нужно дополнительно уметь перечислять реакции во всех тех нестационарных состояниях, в которые мы можем попасть в спецификации после приёма этого стимула в стационарном состоянии и далее через τ-переходы и переходы по реакциям.

130. Полнота тестирования.

Понятно, что стационарное тестирование не является полным, поскольку мы не пытаемся проверить приём реализацией стимулов в нестационарных состояниях.

♣Для нашего примера мы не проверим команду *стоп* в состояниях 2 и 4.

♣Кроме того, нетрудно изменить наш пример так, чтобы при стационарном тестировании достигались не все стационарные состояния. Например, по реакции *откат* автомат не сразу возвращается в начальное состояние 1, а сначала переходит в стационарное состояние 8, где ждёт команду *дальше*, чтобы только после этого вернуться в состояние 1.

♣Как естественное обобщение автомата Мили можно рассматривать асинхронный автомат, который принимает стимулы только в стационарных состояниях. Такой автомат, фактически, эквивалентен своему стационарному автомату.

♣Стационарное тестирование может быть полным, разумеется, со всеми теми оговорками, которые мы делали для автоматов Мили.

131. Асинхронное тестирование: Нестационарное.

132. Ограничение на среду передачи.

Мы будем рассматривать нестационарное тестирование только для среды передачи, которая представляет собой одну неограниченную очередь стимулов и одну неограниченную очередь реакций.

133. Цикл стационарного и нестационарного тестирования.

Нестационарное тестирование является следующим после стационарного этапом тестирования.

♣Его задача – проверить приём реализацией стимулов в нестационарных состояниях.

♣Если при этом достигаются новые стационарные состояния, мы можем переключиться обратно в режим стационарного тестирования. Таким образом, возникает цикл из стационарного и нестационарного тестирования.

134. Наблюдаемые и гипотетические трассы.

Каким образом может быть реализовано нестационарное тестирование?

Прежде всего, заметим, что при нестационарном асинхронном тестировании наблюдаемые трассы, вообще говоря, не совпадают с реальными трассами, которые проходит реализация. *Гипотетической* трассой будем называть трассу, которую могла бы пройти реализация при условии, что мы имеем данную наблюдаемую трассу.

♣Посмотрим на наш пример при нашем допущении полноты.

♣Пусть наблюдается трасса *?начало?стоп*δ.

♣В реализации эту трассу имеют два разных маршрута. Один маршрут из состояния 3 переходит по τ-переходу в состояние 6, где принимается команда *стоп*, а потом обнаруживается, что состояние стационарное. Другой маршрут из состояния 3 переходит по τ-переходу в состояние 4, где принимается команда *стоп* с переходом в то же самое стационарное состояние 6. На

самом деле, здесь нам повезло, что состояние то же самое; оно могло бы быть и другим.

♣Теперь пусть наблюдается трасса *?начало?стоп!ok*. Хотя тест выдаёт команду *стоп* сразу после команды *начало*, однако, реализация выдаёт реакцию *ok* перед тем, как она примет команду *стоп*. Здесь происходит рассинхронизация: можно считать, что одновременно тест выдаёт команду *стоп*, а реализация – реакцию *ok*. В тот момент, когда тест выдаёт команду *стоп*, реакция *ok* уже могла оказаться в выходной очереди. Реализация пройдёт трассу *?начало!ok?стоп*.

♣Однако реализация могла бы быть устроена по-другому. Она могла бы в состоянии 4 принимать команду *стоп* недетерминированно: с переходом не только в состояние 6, но и в какое-нибудь состояние 9, промежуточное между 4 и 5, а при переходе из состояния 9 в состояние 5 выдавать реакцию *ok*.

♣Тогда в ней были бы две разные трассы, соответствующие одной наблюдаемой трассе.

135. Завершённые трассы.

Шаг тестового эксперимента при нестационарном тестировании так же, как при стационарном тестировании, начинается и заканчивается в стационарном состоянии. Отличие в том, что, кроме первого стимула, с выдачи которого начинается шаг тестирования, мы можем выдавать дополнительные стимулы, не дожидаясь перехода в стационарное постсостояние. В целом шаг тестирования – это последовательность выдачи стимулов в реализацию и приёма реакций от неё, начинающаяся с выдачи первого стимула и заканчивающаяся тогда, когда при ожидании реакций мы обнаруживаем их отсутствие, то есть стационарность. Заметим, что отсюда вовсе не следует, что все промежуточные состояния нестационарны. Например, выдача подряд двух стимулов может привести к тому, что второй стимул реализация примет в стационарном состоянии.

Итак, наблюдаемая трасса, соответствующая одному шагу нестационарного тестирования, представляет собой последовательность стимулов, реакций и символа δ, начинающуюся со стимула и заканчивающаяся символом δ, причём это единственное его вхождение в трассу. При асинхронном тестировании наблюдаемая трасса, вообще говоря, не совпадает с трассой, которую прошла реализация, но она определяет множество гипотетических трасс. Это определение зависит от среды передачи.

♣Мы рассматриваем среду, состоящую из неограниченных входной и выходной очередей. Для этой среды сформулируем каузальные правила, то есть правила, определяющие порядок стимулов и реакций в гипотетической трассе.

♣Правило очерёдности стимулов: Стимулы выбираются реализацией из входной очереди, очевидно, в том же порядке, в каком они поступали в очередь из теста. Поэтому порядок следования стимулов в гипотетической трассе такой же, как в наблюдаемой трассе.

♣Правило очерёдности реакций: Аналогично, тест выбирает реакции из выходной очереди в том же порядке, в каком реализация выдавала их в очередь. Поэтому порядок следования реакций в гипотетической трассе такой же, как в наблюдаемой трассе.

♣Правило «стимул после реакции»: Далее, если мы выдавали некоторый стимул после того, как получили некоторую реакцию, то, очевидно, в гипотетической трассе эта реакция предшествует этому стимулу. Это правило имеет общий характер – для любой среды.

♣Правило первого стимула: Поскольку шаг тестирования начинается в стационарном состоянии, гипотетическая трасса начинается со стимула, но не с реакции. Для одной входной очереди этим стимулом, очевидно, будет первый стимул наблюдаемой трассы.

♣Правило стационарности: Наконец, трасса заканчивается стационарностью.

Мы перечислили все каузальные зависимости, которым должны подчиняться символы гипотетической трассы для данной среды передачи. В целом эти правила определяют частичный порядок своих символов.

♣Гипотетическая трасса – это любая трасса, то есть любой линейный порядок, не противоречащий такому частичному порядку.

Как мы уже говорили, гипотетическая трасса может проходить через стационарные состояния. Это означает, что гипотетическая трасса может не

только заканчиваться стационарностью, но и содержать её символ внутри. Разумеется, за внутренней стационарностью может следовать только стимул или стационарность (это петля и её можно повторять сколько угодно раз), но не реакция. Для нашей среды это не имеет значения, поскольку тест такую внутреннюю стационарность всё равно не видит. Поэтому для простоты мы будем рассматривать гипотетические трассы без внутренней стационарности.

♣Для трассы, нарисованной на слайде, мы имеем 19 гипотетических трасс. Они отличаются тем, что стимулы могут сдвигаться к концу трассы до тех пор, пока это не нарушает каузальные зависимости. Красным жирным шрифтом выделены такие сдвинутые стимулы.

Наблюдаемая трасса, соответствующая одному шагу нестационарного тестирования, заканчивается стационарностью, но не содержит стационарность внутри. Тем самым, это завершённая трасса. Нужно только помнить, что мы рассматриваем трассы, начинающиеся не в начальном, а в текущем состоянии автомата.

♣Заметим, что при стационарном тестировании гипотетическая завершённая трасса всегда совпадает с наблюдаемой завершённой трассой. Действительно, стимул, с которого начинается наблюдаемая трасса, всегда является первым символом гипотетической трассы. При стационарном тестировании далее идут одни реакции, которые сохраняют свой порядок, а заканчивается всё стационарностью.

136. Мультистимульное стационарное тестирование.

Частный, но важный и полезный случай тестирования, когда мы сначала выдаём последовательность стимулов, а потом получаем реакции до стационарности. В наблюдаемой трассе сначала идут все стимулы, потом все реакции и в конце – стационарность. Такое тестирование отличается от стационарного тестирования, которое мы рассматривали раньше, только тем, что мы выдаём не один стимул, а последовательность стимулов. В этом случае достаточно ограничиться рассмотрением только завершённых трасс – так же, как при обычном стационарном тестировании.

♣В целом у нас есть три случая, когда тестирование асинхронного автомата сводится к тестированию автомата Мили, получаемого из исходного автомата специальным преобразованием.

Тестирование автомата Мили: в ответ на один стимул мы получаем одну реакцию. В этом случае преобразование тождественное.

♣Одностимульное стационарное тестирование асинхронного автомата: в ответ на один стимул, подаваемый в стационарном состоянии, мы получаем последовательность реакций. Соответствующий автомат Мили – это стационарный автомат. Его состояния – это стационарные состояния исходного автомата. Переход из одного стационарного состояния в другое стационарное состояние помечен одним стимулом и последовательностью реакций.

♣Мультистимульное стационарное тестирование асинхронного автомата: в ответ на последовательность стимулов, которая начинает подаваться в стационарном состоянии, мы получаем последовательность реакций. Соответствующий автомат Мили – это мультистимульный стационарный автомат. Его состояния – это стационарные состояния исходного автомата. Переход из одного стационарного состояния в другое стационарное состояние помечен последовательностью стимулов и последовательностью реакций.

Можно сказать, что мультистимульное тестирование занимает промежуточное положение между стационарным и нестационарным тестированиями. По стратегии тестирования, основанной на обходе автомата Мили, это тестирование стационарное. Однако оно нуждается в умении перечислять реакции, допускаемые спецификацией после некоторой трассы, для того, чтобы определить безопасность последовательности подаваемых стимулов. Кроме того, одной наблюдаемой трассе здесь может соответствовать несколько гипотетических трасс, получающихся сдвигом стимулов к концу трассы. В этом оно схоже с нестационарным тестированием.

137. Незавершённые трассы.

В общем случае мы можем перемежать выдачу стимулов и приём реакций. Безопасность выдаваемого стимула зависит от реакций, принятых к этому моменту времени. Поэтому мы должны по ходу дела, то есть ещё до завершения трассы, во-первых, проверять правильность получаемых реакций и, во-вторых, если реакции правильные, определять безопасные стимулы, которые можно было бы выдать в этот момент времени.

♣Незавершённая наблюдаемая трасса – это трасса, которая не заканчивается стационарностью.

♣Если трасса не завершена, то мы не знаем, получили мы все реакции, которые реализация уже выдала, или ещё не все. Поэтому мы считаем, что гипотетическая трасса может содержать дополнительные реакции. Эти дополнительные реакции, по правилу очерёдности реакций, идут после уже принятых реакций. Однако порядок следования ещё не принятых реакций и уже посланных стимулов может быть любым с учётом правила первого стимула.

Кроме того, мы не знаем, приняла ли реализация все посланные стимулы или ещё нет. Гарантированно можно говорить лишь о том, что, если тест получил хотя бы одну реакцию, то реализация приняла первый стимул. (Поскольку в стационарном состоянии, с которого мы начинаем, реакции не выдаются.)

♣Гипотетические трассы нам будут нужны для двух целей:

- Верификация: проверка, что реализация выдала правильные реакции.
- Безопасность: вычисление стимулов, которые можно безопасно посылать в реализацию после наблюдаемой трассы.

138. Безопасность тестирования.

Как мы уже говорили, тестовый эксперимент должен быть безопасным, то есть не приводить к разрушению реализации. О безопасности данного тестового эксперимента при синхронном тестировании мы судили на основании гипотезы о безопасности, которая утверждает, что после трассы, которая есть как в реализации, так и в спецификации, можно безопасно давать только такой стимул, которым эта трасса безопасно продолжается в спецификации. Поскольку при стационарном асинхронном тестировании наблюдаемая трасса совпадает с гипотетической, нам достаточно этой же гипотезы о безопасности. Но для нестационарного асинхронного тестирования мы должны гипотезу о безопасности усилить.

♣Если в тестовом эксперименте мы получили некоторую наблюдаемую трассу, то мы должны посмотреть, какие трассы могла бы пройти спецификация, если бы в этом тестовом эксперименте она занимала место реализации. Иными словами, мы должны взять пересечение гипотетических трасс, порождаемых данной наблюдаемой трассой, и трасс спецификации, начинающихся с текущего спецификационного состояния.

Стимул, который мы можем послать после наблюдаемой трассы, будет принят реализацией после того, как она примет все стимулы и выдаст все реакции, которые есть в наблюдаемой трассе. При этом реализация пройдёт такую гипотетическую трассу, которая содержат все посланные стимулы, все полученные реакции и, быть может, ещё какие-то дополнительные реакции. Поскольку по нашему допущению полноты в нестационарном состоянии неспецифицированный стимул не приводит к разрушению, нас интересуют только стационарные состояния, в которых могут заканчиваться такие гипотетические трассы.

♣Иными словами, нам достаточно ограничиться завершёнными гипотетическими трассами, которые содержат все посланные стимулы и все полученные реакции.

♣Вот эти трассы для нашего примера.

♣Гипотеза о безопасности будет звучать так: если стимул безопасен после каждой спецификационной трассы, порождаемой данной наблюдаемой трассой, то этот стимул безопасен после любой реализационной трассы, порождаемой данной наблюдаемой трассой. Эта гипотеза сильнее, поскольку реализационная трасса может не совпадать ни с одной из спецификационных трасс, порождаемых данной наблюдаемой трассой.

139. Верификация наблюдаемых трасс.

Верификация наблюдаемой незавершённой трассы – это проверка того, что реализация выдала правильные реакции.

Для этого годятся завершённые гипотетические трассы, которые мы использовали для проверки безопасности стимулов. Однако они несут лишнюю информацию: для верификации нам не важны реакции и стационарность, которые мы ещё не получили.

♣Достаточно ограничиться трассами, которые содержат все полученные реакции, даже если они содержат не все посланные стимулы.

140. Сериализация (шаг тестирования).

Теперь посмотрим, как работает тестовая система на одном шаге нестационарного тестирования.

Эта работа похожа на случай стационарного тестирования, Отличие в том, что мы можем давать стимулы, не дожидаясь стационарного состояния, и у нас могут получаться разные спецификационные трассы, порождаемые одной и той же наблюдаемой трассой.

Пусть в какой-то момент времени мы получили некоторую наблюдаемую трассу. Под трассой мы будем понимать трассу, которая наблюдалась не с начала тестирования, а с начала текущего шага тестирования.

♣В начале шага тестирования это пустая трасса.

♣Мы должны определить все спецификационные трассы, порождаемые этой наблюдаемой трассой. Это называется сериализацией наблюдаемой трассы.

Для этого сначала вычисляем все гипотетические трассы, порождаемые наблюдаемой трассой, которые нужны для верификации: трассы, которые содержат все полученные реакции, даже если они содержат не все посланные стимулы. Каждую такую гипотетическую трассу верифицируем по спецификации аналогично тому, как мы это делали в стационарном случае.

Если гипотетическая трасса отвергается, то, в отличие от стационарного случая, это не означает ошибку. Просто мы эту трассу отбрасываем и верифицируем следующую гипотетическую трассу.

♣Ошибка фиксируется, когда сериализация закончена, и все гипотетические трассы отброшены.

♣Если ошибок нет, то мы смотрим, завершена ли наблюдаемая трасса, то есть, заканчивается она стационарностью или нет.

♣Если трасса завершена, шаг тестирования завершён.

♣В противном случае, мы должны принять решение: дать следующий стимул или ждать реакций. Выбор определяется общей стратегией нестационарного тестирования.

♣Если мы решаем ждать реакций, то ждём реакций, а, получив реакцию или стационарность, повторяем описанный процесс для новой наблюдаемой трассы.

♣Если мы решаем выдать стимул, то стратегия тестирования определяет, какие стимулы нам нужно давать в текущей ситуации, то есть после пройденной наблюдаемой трассы. С другой стороны, мы должны определить, какие стимулы можно безопасно давать в реализацию. Здесь важно, что у нас может получиться не одна, а несколько спецификационных трасс. Стимул, который можно безопасно давать после данной наблюдаемой трассы, – это стимул, который в спецификации безопасен после каждой из полученных спецификационных трасс.

В целом мы ищем те стимулы, которые *можно* и *нужно* давать.

♣Если таких стимулов нет, нам ничего другого не остаётся, кроме как ждать реакций.

♣Если такой стимул есть, мы можем его дать

♣ и повторить процесс для новой наблюдаемой трассы.

141. Перечисление.

При нестационарном тестировании нам требуется перечислять в состоянии спецификации стимулы, постсостояния и реакция. Это нужно как для верификации, так и для определения безопасных стимулов.

142. ПРОБЛЕМЫ: Выбор стратегии тестирования (аналог обхода автомата).

В целом у теста должна быть некоторая стратегия тестирования, которая определяет, какие стимулы нужно давать в данной ситуации, а какие не нужно, например, потому, что тест уже давал этот стимул в этой ситуации. В то же время тест адаптивно подстраивается под текущую ситуацию, определяемую получаемыми от реализации реакциями. Стимул даётся тогда, когда его не только можно, но и нужно давать в данной ситуации. В противном случае тест ожидает реакций, изменяющих ситуацию.

♣Прежде всего, отметим, что без каких-либо дополнительных условий нестационарный тест не может гарантированно проверить ничего, кроме того, что проверяет стационарный тест. Например, если тест работает достаточно медленно, то, пока он «думает», реализация успевает принять стимул, выдать все реакции и перейти в стационарное состояние. Таким образом, в любом тестировании стимулы всё равно будут приниматься только в стационарных состояниях, и, следовательно, тестирование будет стационарным.

Для того чтобы говорить о каких-то гарантиях при нестационарном тестировании, нужны дополнительные тестовые возможности и соответствующие реализационные гипотезы.

♣Что можно предложить?

Временные ограничения. Если считать, что, наоборот, реализация работает достаточно медленно, а среда и тест – достаточно быстро, то мы можем выдать во входную очередь серию стимулов. При этом мы будем уверены, что реализация может принять второй стимул в первом же состоянии, в которое она попадёт после первого стимула, и в котором этот стимул принимается. Аналогично для остальных стимулов. Однако само по себе это недостаточно для того, чтобы реализация действительно принимала стимулы тогда, когда они уже есть во входной очереди.

Поэтому дополнительно требуется реализационная гипотеза о приоритете приёма стимулов над выдачей реакций и τ-переходами.

Управление погодой и мистическим синхронизатором. Это тестовые возможности общего вида, которые заставляют реализацию вести себя так или иначе и обеспечивают передачу стимулов из входной очереди в реализацию или передачу реакций из реализации в выходную очередь тогда, когда нам это нужно. Фактически, все такие тестовые возможности вносят в тестирование элементы синхронизации, присущие синхронному тестированию.

Вероятности переходов. Введение вероятностей переходов позволяет строить нестационарные тесты, которые эти переходы проверяют с той или иной вероятностью.

♣Дополнительно нужно сделать следующее важное замечание:

Формально, если тестирование прошло при всех «погодных условиях», после этого в спецификации могут остаться непроверенные стимулы, определённые в нестационарных состояниях. Однако это такие стимулы, которые ни при каких условиях нельзя безопасно давать в реализацию. Можно сказать, что такие стимулы избыточны в спецификации. Причин, по которым эти стимулы всё же определены, несколько.

Первая причина: это могут быть стимулы, которые опасны при асинхронном тестировании, но безопасны при синхронном тестировании.

Вторую причину мы уже обсуждали, когда говорили о маркировке разрушением поведения системы, которое нежелательно тестировать по тем или иным причинам. Для одной и той же спецификации можно было бы применять различные маркировки и, тем самым, выполнять то или иное тестирование. В зависимости от маркировки одни и те же стимулы могут быть как разрушающими, так и безопасными.

143. Асинхронное тестирование: Гипертестирование.

144. Среда передачи.

Стационарное и нестационарное тестирование мы рассматривали для фиксированной среды передачи – две неограниченные очереди: очередь стимулов и очередь реакций. Если среда передачи другая, то в тестировании многое меняется. Тестирование, которое можно вести для разных сред из некоторого класса сред, мы будем называть гипертестированием.

♣Мы будем исходить из общей схемы тестирования, которая состоит из последовательности шагов тестирования. На каждом шаге мы имеем наблюдаемую трассу, а среда задаёт отображение наблюдаемой трассы во множество гипотетических трасс.

♣Для спецификации мы пока по-прежнему будем предполагать наше комбинированное допущение полноты: неспецифицированный стимул блокируется в стабильном нестационарном состоянии и ведёт к разрушению в стационарном состоянии.

♣Итак, мы рассматриваем общий случай среды передачи, но такой, который удовлетворяет следующим ограничениям:

- Среда передаёт стимулы и реакции без потерь.
- Среда не генерирует лишних стимулов и реакций.

♣Теперь нам нужны ограничения, которые, *во-первых*, позволяли бы определять стационарность реализации по тайм-ауту при ожидании реакций. *Во-вторых*, шаг тестирования предполагает, что при срабатывании тайм-аута среда пуста: реализация уже приняла все стимулы, посланные тестом, а тест принял все реакции, выданные реализацией. Наконец, *в-третьих*, нам по-прежнему нужно, чтобы не возникало d`adlock`a при передаче стимула из теста в среду.

Асинхронное тестирование реализации мы определили как синхронное тестирование композиции реализации со средой передачи. Поэтому тайм-аут в тесте при ожидании реакций для любой среды означает дивергенцию или стационарное состояние этой композиции. В общем случае такое состояние

вовсе не обязательно означает стационарность реализации или пустоту среды.

♣Мы налагаем на среду и реализацию следующие дополнительные ограничения:

- Композиция среды и реализации конвергентна.
- В стационарном состоянии композиции среда пуста, а реализация находится в стационарном состоянии.
- В стабильном состоянии композиция принимает от теста все стимулы.

♣Из наших ограничений следует, что гипотетическая трасса, порождаемая данной наблюдаемой трассой, состоит из тех же самых стимулов и реакций, но, быть может, расположенных в другом порядке. Здесь мы по-прежнему рассматриваем наблюдаемые трассы, начинающиеся в текущем стационарном состоянии. У нас по-прежнему будут работать каузальные правила: правило «стимул после реакции», правило первого стимула и правило стационарности.

Отличие от одной входной очереди в том, что для правила первого стимула таким первым стимулом гипотетической трассы будет не обязательно первый стимул наблюдаемой трассы.

Эти три каузальных правила являются общими для всех таких сред.

Однако правило очерёдности стимулов и правило очерёдности реакций верны только для среды с одной неограниченной очередью стимулов и одной неограниченной очередью реакций. Для других сред могут быть другие каузальные правила, играющие аналогичную роль.

Сначала мы рассмотрим несколько характерных примеров различных сред передачи.

145. Среда: несколько очередей.

Первым примером является среда, моделируемая не одной входной и одной выходной очередями, а большим числом очередей – входных и выходных. Будем считать, что алфавит стимулов разбит на столько классов, сколько есть входных очередей, так что стимул из i-го класса помещается в i-ую очередь. Аналогично, алфавит реакций разбит на столько классов, сколько

есть выходных очередей, так что реакция из j-го класса помещается в j-ую очередь.

Пусть, например, стимулы и реакции – это натуральные числа. Среда имеет две входные очереди: для чётных и нечётных стимулов, и две выходные очереди: для чётных и нечётных реакций.

♣Множество гипотетических трасс определяется, во-первых, линейным порядком стимулов в каждой входной очереди.

♣Во-вторых, линейным порядком реакций в каждой выходной очереди.

Это модифицированные правила очерёдности стимулов и реакций, которые у нас были для одной входной и одной выходной очереди.

Кроме того, работают общие каузальные правила:

♣правило «стимул после реакции» (но не обязательно первый посылаемый стимул),

♣правило первого стимула,

♣и правило стационарности.

Эти трассы нельзя задать частичным порядком на множестве стимулов и реакций наблюдаемой трассы. Причина – в модифицированном правиле первого стимула. Первый символ трассы должен быть стимулом, но здесь у нас на эту роль претендуют два стимула – первые стимулы двух входных очередей. Один из них будет первым символом трассы, а другой может быть принят реализацией позже некоторых реакций.

146. Передача стимулов и реакций, минуя среду.

При асинхронном тестировании у нас может быть возможность передавать часть стимулов и реакций непосредственно между реализацией и тестом, минуя среду. Такие стимулы и реакции будем называть *прямыми*. Это допускается оператором параллельной композиции, если алфавиты реализации, среды и теста удовлетворяют соответствующему условию.

147. Передача стимулов и реакций, минуя среду (пример).

Рассмотрим пример, аналогичный предыдущему примеру: 2 входные очереди и 2 выходные очереди. Но только у нас будет стимул и реакция, которые передаются, минуя среду. Они имеют номер 3.

У нас появляются новые каузальные зависимости:

- ♣Порядок прямых стимулов и реакций сохраняется при переходе от наблюдаемой трассы к гипотетической трассе.
- ♣Прямой стимул принимается реализацией раньше, чем любой стимул, посланный тестом после этого прямого стимула.
- ♣Прямая реакция выдаётся реализацией позже, чем любая реакция, полученная тестом до этой прямой реакции.

148. Среда: «куча» (множество).

Другая среда – это «куча», то есть неупорядоченное множество. Например, если вместо входной очереди используется куча стимулов, то для наблюдаемой трассы из трёх стимулов 1,2,3 любая их линейная последовательность является гипотетической трассой.

149. Среда: стек.

Если вместо входной очереди используется стек, то для наблюдаемой трассы из трёх стимулов 1,2,3 пять из шести линейных порядков являются гипотетическими трассами. Один линейный порядок не может быть гипотетической трассой – это порядок 3,1,2.

♣Реализация может первым принять стимул 3 только в том случае, когда в стеке находятся стимулы 1 и 2. Поскольку в вершине стека расположен как раз стимул 2, реализация не может принять стимул 1 раньше стимула 2.

150. Среда: приоритеты очередей, куч и стеков.

Среда передачи стимулов так же, как передачи реакций, может быть комбинированной, состоящей из набора очередей, куч и стеков. Между этими компонентами, кроме того, могут быть установлены приоритеты.

Например, стимулы – это натуральные числа и они поступают в две входные очереди с разными приоритетами. Среда предлагает реализации стимул из

низкоприоритетной очереди только в том случае, когда высокоприоритетная очередь пуста. Пусть в низкоприоритетную очередь поступают нечётные числа, а в высокоприоритетную очередь – чётные числа. Для наблюдаемой трассы из трёх стимулов 1,2,3 гипотетическими трассами являются только два линейных порядка из шести. Три линейных порядка не годятся, потому что они нарушают порядок следования стимулов в одной очереди. Четвёртый порядок нарушает приоритетность очередей.

♣Если в этом примере заменить очереди кучами, то три линейных порядка из шести отбраковываются, потому что они нарушают приоритетность куч.

151. Зависимость между стимулами и реакциями.

До сих пор мы рассматривали примеры сред, в которых стимулы и реакции передаются независимо друг от друга. Между стимулами и реакциями (или стационарностью) не было никакого порядка следования, если не считать общих каузальных правил: правило первого стимула, предшествующего всем реакциям, правило «стимул после реакции», и правило стационарности, завершающей трассу.

Однако в некоторых случаях между стимулами и реакциями (или стационарностью) есть дополнительные каузальные зависимости, которые мы можем учитывать при построении гипотетических трасс.

♣Первый пример – привязанные реакции. Про такую реакцию мы всегда знаем, что она выдана реализацией после приёма стимула, к которому она привязана.

♣Про немедленную реакцию мы знаем больше: она идёт в гипотетической трассе непосредственно после стимула, ответом на который является.

♣Среда может сама устанавливать дополнительные каузальные зависимости между стимулами и реакциями. Например, среда может предлагать реализации некоторые стимулы только тогда, когда реализация не выдаёт реакций, то есть, находится в стационарном состоянии. Такие стимулы можно назвать стационарными. Для обнаружения стационарности в реализации среда может использовать θ-переход при ожидании реакций от реализации аналогично тому, как это делает тест. В гипотетической трассе таким стационарным стимулам непосредственно предшествуют внутренние

символы стационарности δ. Фактически, для такой среды нестационарное тестирование сводится к стационарному.

152. Автомат среды: Передача стимулов и реакций.

Итак, мы рассмотрели примеры различных сред. В общем, виде среда задаётся асинхронным автоматом. Посмотрим, как наложенные нами ограничения на среду, могут быть сформулированы в терминах автомата среды.

Мы уже говорили, что формально взаимодействие теста и среды, с одной стороны, и среды и реализации, с другой стороны, происходят в разных, непересекающихся алфавитах стимулов и реакциях. Поскольку мы хотим, чтобы среда только передавала стимулы и реакции без потерь и без генерации лишних стимулов и реакций, можно считать, что между стимулами теста и стимулами реализации существует взаимно-однозначное соответствие так же, как между реакциями теста и реакциями реализации. На слайде стимулы и реакции реализации обозначены красным цветом, а стимулы и реакции теста – синим цветом.

♣Среда содержит стимулы и реакции, которые выданы одной стороной (тестом или реализацией), но ещё не приняты другой стороной. Это можно описать в виде мультимножества, который определён в каждом состоянии среды. В мультимножестве, в отличие от обычного множества, стимул или реакция может входить не один, а несколько раз. Формально, мультимножество – это отображение, которое каждому элементу ставит в соответствие число вхождений. Если число вхождений стимула или реакции равно нулю, это означает, что в среде нет этого стимула или этой реакции.

♣Правило, согласно которому среда передаёт стимулы без потерь и не генерирует лишних стимулов, означает соответствующие правила по изменению мультимножества при передаче стимула. Если стимул принимается из теста, число вхождений этого стимула увеличивается на 1. Стимул может выдаваться в реализацию только в том случае, если число его вхождений больше нуля, в этом случае при переходе число вхождений стимула уменьшается на 1. Аналогично изменяется число вхождений реакции при её передаче в среду или из среды.

♣Тестирование основано на заданном отображении наблюдаемой трассы в гипотетические трассы. В общем случае это отображение зависит от

состояния среды в начале шага тестирования. Мы уже потребовали, чтобы в начале шага тестирования среда была пуста, но отсюда не следует, что такое состояние среды единственное. Поэтому нам нужно дополнительно потребовать, чтобы в каждом состоянии автомата среды, соответствующем отсутствию в среде стимулов и реакций, отображение трасс было одно и то же. В начальном состоянии среда пуста.

153. Автомат среды: Конвергентность композиции среды и реализации.

Теперь посмотрим, что нужно, чтобы композиция среды и реализации была конвергентна. Напомним, что конвергентность нужна для того, чтобы в тесте правильно интерпретировать истечение тайм-аута при ожидании реакций. Такая интерпретация должна означать стационарность, а не дивергенцию.

Если тест ожидает реакций, взаимодействие среды и реализации состоит из переходов четырёх типов:

1) асинхронные τ-переходы в реализации,
2) асинхронные τ-переходы в среде,
3) синхронные переходы по передаче стимула из среды в реализацию,
4) синхронные переходы по передаче реакции из реализации в среду.

Посмотрим, сколько может быть переходов разного типа в цепочке переходов композиции. Для конвергентной реализации число переходов первого типа конечно. Поскольку тест от начала работы мог послать в среду только конечное число стимулов, а среда не генерирует лишних стимулов, число переходов третьего типа также конечно. Для конечности числа переходов второго типа требуется конвергентность среды.

♣ Для конечности переходов четвёртого типа требуется либо A) отсутствие осцилляции в реализации, либо B) отсутствие в среде бесконечной цепочки τ-переходов и переходов по реакциям. Последний случай также перекрывает требование конвергентности среды.

Заметим, что неограниченная очередь реакций конвергентна, но не удовлетворяет условию B): в ней есть бесконечная цепочка приёма реакций от реализации. Именно поэтому для такой среды мы требовали, чтобы реализация была неосциллирующей. В общем же случае возможен как вариант A), так и вариант B).

Когда мы рассматривали шаг асинхронного тестирования, мы потребовали не только конвергентности композиции, но и завершённости шага тестирования: не бывает бесконечной последовательности получаемых реакций, каждая

последовательность реакций рано или поздно заканчивается стационарностью.

♣Поэтому пока будем требовать, чтобы реализация была неосциллирующей. Как можно тестировать осциллирующие реализации, мы посмотрим позже.

♣

154. Автомат среды: Таблица условий.

Теперь посмотрим, что нужно, чтобы выполнялись правила стационарности и отсутствия блокирующего deadlock`a. Напомним, что мы требуем, чтобы стационарность композиции среды и реализации означала стационарность реализации и пустоту среды. В этом случае обнаружение стационарности в конце шага тестирования означает достижение стационарного состояния реализации, все посланные тестом стимулы приняты реализацией, а все выданные реализацией реакции получены тестом. Гипотетическая трасса состоит из тех же стимулов и реакций, что и наблюдаемая трасса и выполняются общие каузальные правила.

Мы рассмотрим четыре вида стабильных состояний среды в зависимости от того, имеются или нет в ней не переданные стимулы и реакции. В терминах мультимножества это означает наличие или отсутствие стимулов или реакций с ненулевым числом вхождений.

Соответственно, мы рассмотрим два вида состояний теста: принимающее состояние, когда тест ждёт всех реакций и тайм-аута, и посылающее, когда он посылает один стимул. Эти стабильные состояния теста стационарное и нестационарное, соответственно. Точно так же мы рассмотрим два вида стабильных состояний реализации: стационарное, когда реализация принимает все стимулы, быть может, с разрушением, и не выдаёт реакции, и нестационарное, когда реализация выдаёт хотя бы одну реакцию.

Всего у нас получается 16 вариантов, которые на слайде находятся в центральном квадрате 4x4. Левые 8 вариантов, когда тест ждёт реакций, соответствуют правилу стационарности, а правые 8 вариантов, когда тест посылает стимул, соответствуют запрету блокирующего deadlock`a.

Для стационарности только один из 8 вариантов удовлетворяет правилу стационарности: реализация находится в стационарном состоянии, а среда пуста. В этом случае возникает deadlock и тест совершает θ-переход. В остальных 7 вариантах записаны необходимые и достаточные условия

отсутствия deadlock`а для любой реализации. Большая буква X или Y под знаком вопроса означают приём средой всех стимулов от теста или всех реакций от реализации, соответственно. Маленькая буква x или y под восклицательным знаком означает, что среда предлагает хотя бы один стимул для передачи в реализацию или хотя бы одну реакцию для передачи в тест, соответственно.

Для блокировки имеется 8 вариантов, для каждого из которых записано необходимое и достаточное условие отсутствия deadlock`а для любой реализации.

Строка таблицы, содержащая 4 варианта, соответствует одному виду состояния среды. Общее условие для данного вида состояния среды, очевидно, является конъюнкцией условий этих 4-х вариантов. Оно записано в самом правом итоговом столбце, выделенном зелёным цветом.

155. Автомат среды: Правила работы.

Этот итог можно описать отдельно как правила работы среды в зависимости от вида её состояния, то есть от наличия или отсутствия в среде не переданных стимулов и реакций.

Среда должна принимать все стимулы от теста, если в ней нет стимулов, и все реакции от реализации, если в ней нет реакций.

Среда должна выдавать в реализацию хотя бы один стимул, если у неё есть не переданные стимулы, и нет реакций.

Соответственно, среда должна выдавать хотя бы одну реакцию в тест, если у неё есть не переданные реакции, и нет стимулов.

Если же в среде есть и стимулы и реакции, то для неё допустимы два типа поведения: 1) среда может выдавать реакцию в тест и принимать все стимулы от теста; 2) среда может выдавать стимул в реализацию и принимать все реакции от реализации.

Разумеется, среда может делать что-то ещё, кроме того, что от неё требуется. Например, при наличии в ней и стимулов и реакций, она может принимать все стимулы от теста и все реакции от реализации и выдавать некоторые не переданные стимулы в реализацию и некоторые не переданные реакции в тест.

156. Автомат среды: Дополнительные примеры сред.

Основываясь на этих правилах, мы можем рассмотреть дополнительные примеры сред.

Ограниченная очередь стимулов и неограниченная очередь реакций. Что произойдёт, когда входная очередь заполнена, а тест выдаёт ещё один стимул? В этом случае среда не принимает стимул от теста, но она предлагает реализации стимул, находящийся в голове входной очереди, и принимает от реализации любую реакцию в свою неограниченную выходную очередь. Если реализация находится в стационарном состоянии, она примет предлагаемый стимул и во входной очереди появится место для нового стимула от теста. Если реализация в нестационарном стабильном состоянии не принимает головной стимул входной очереди, она выдаёт реакцию, которую среда примет. Разумеется, поскольку выходная очередь не ограничена, среда имеет бесконечную цепочку приёма реакций от реализации и для того, чтобы композиция среды и реализации была конвергентной, реализация должна быть неосциллирующей.

Неограниченная очередь стимулов и ограниченная очередь реакций. Блокирующий deadlock возникнуть не может, поскольку среда всегда готова принять стимул в свою неограниченную входную очередь. Что произойдёт, когда будет заполнена выходная очередь? Среда принимает все стимулы от теста и предлагает тесту реакцию, находящуюся в голове выходной очереди. Если тест ждёт реакций, он эту реакцию получит. Если тест посылает стимул, он будет принят. В этой среде нет бесконечной цепочки приёма реакций от реализации, и поэтому для конвергентности композиции нам не требуется, чтобы в реализации не было осцилляции. Однако для завершённости шага тестирования требование отсутствия осцилляции в реализации пока остаётся.

Заметим, что обе очереди, входная и выходная, не могут быть ограниченными, так как это может привести к блокирующему deadlock`у. Когда обе очереди заполнены, среда не принимает стимулы и реакции. Если реализация в стабильном нестационарном состоянии не принимает головной стимул входной очереди и посылает реакции, эти реакции блокируются средой. Если тест посылает стимул в среду, этот стимул также блокируется средой.

157. ПРОБЛЕМЫ: Автомат среды: как по автомату среды построить отображение наблюдаемых трасс в гипотетические трассы?

Для конкретных примеров среды, которые мы рассмотрели, легко описать отображение наблюдаемых трасс в гипотетические трассы. Однако в общем случае остаётся проблема: как по автомату среды вычислить такое отображение?

158. Внешняя блокировка: Адаптивное тестирование.

До сих пор мы считали, что среда не блокирует стимул, посылаемый тестом. Если это ограничение снять, то такая блокировка, тем не менее, вовсе не означает блокировку этого стимула реализацией. Более того, такая блокировка может происходить и тогда, когда реализация по-прежнему удовлетворяет нашему допущению полноты, то есть, не блокирует стимулы в стационарном состоянии. Например, среда имеет ограниченную ёмкость для стимулов и не принимает стимул от теста, когда эта ёмкость полностью заполнена, а реализация не принимает стимулы потому, что находится в нестационарном состоянии.

♣Блокировку стимулов, передаваемых из теста в среду, будем называть *внешней блокировкой*.

♣Внешнюю блокировку тест может обнаружить с помощью θ-перехода (тайм-аута) в посылающем состоянии.

♣Что может делать тест после обнаружения внешней блокировки? В общем случае он может попробовать послать другой стимул или начать ждать реакций.

159. Внешняя блокировка: Таблица условий.

Посмотрим, что меняется в таблице условий для автомата среды.

Разрешая внешнюю блокировку, мы уже не можем требовать, чтобы среда в любой ситуации принимала любой стимул, посылаемый тестом.

♣Единственный случай, когда это необходимо, – это пустая среда. Если в этой ситуации среда не примет стимул, тест начнёт ждать реакций. В среде реакций нет и, если реализация находится в стационарном состоянии, реакций и не будет. Таким образом, тест обнаружит стационарность, что правильно, но дальнейшая работа невозможна.

♣Итак, мы оставляем требование, чтобы среда принимала все стимулы от теста в том случае, когда среда пуста. В остальных случаях мы не накладываем на среду никаких ограничений.

♣В итоговом столбце условий получается, что среда не обязана принимать стимулы от теста, если в ней есть реакции. Адаптивный тест, обнаружив внешнюю блокировку, может выбрать эти реакции и тогда гарантированно среда примет от него любой стимул.

160. Внешняя блокировка: Правила работы среды.

Правила работы среды меняются следующим образом.

♣Приём всех стимулов от теста обязателен только для пустой среды.

Если в среде нет реакций, она обязана принимать все реакции из реализации.

Если в среде есть реакции, она обязана хотя бы одну из них предлагать тесту.

Среда обязана предлагать хотя бы один стимул реализации, если в среде нет реакций. Но, если в среде есть и стимулы и реакции, то она может выбирать: выдавать реакцию в тест или выдавать стимул в реализацию и принимать все реакции от реализации.

♣В качестве примера годится любой входной и любой независимый выходной набор очередей, куч и стеков.

♣Более интересен пример среды, в которой ограничено сверху суммарное число стимулов и реакций, находящихся в среде. Без дополнительных ограничений такая среда может не удовлетворять нашим условиям. Например, среда всегда принимает стимулы из теста, пока она не заполнена. Тогда может получиться, что реализация принимает первый стимул, посланный тестом, и переходит в нестационарное состояние, намереваясь выдавать реакции. В этот момент среда пуста и тест успевает заполнить её своими стимулами. После этого тест не может послать стимул, реализация не может послать реакцию, и тест не может выбрать реакции из среды, потому что их в ней нет. В тесте возникает тайм-аут при ожидании реакций, но это

ложная стационарность. Ограничение, которое нужно наложить на такую среду, следующее: она должна блокировать стимулы из теста, если в ней нет реакций и есть только одно свободное место. Это место среда резервирует для реакции от реализации.

161. Внутренняя блокировка. Порты.

До сих пор мы предполагали, что в стационарных состояниях нет блокировки стимулов. Каждый стимул в таком состоянии принимается, а если он не специфицирован, то это означает приём с возможным разрушением.

Для одной входной очереди такой подход естественен, поскольку, если реализация в стационарном состоянии не принимает головной стимул очереди, то она останавливается и никакими действиями теста мы не сдвинем реализацию с мёртвой точки.

Теперь мы ослабим требование к реализации, разрешив в стационарном состоянии не только приём стимула и разрушение, но и блокировку. В нестационарном состоянии по-прежнему допускается блокировка стимула, но принимаемый стимул не разрушающий.

♣Ситуация, когда реализация находится в стационарном состоянии, но реализация не принимает ни один из стимулов, предлагаемых средой, хотя такие стимулы есть, называется *внутренней блокировкой*.

Для реализации, которая принимает все стимулы в каждом стационарном состоянии, не бывает внутренней блокировки. В общем же случае это не так.

♣Если никакими действиями теста мы не можем снять внутреннюю блокировку, она называется *постоянной*. Такая постоянная блокировка может возникнуть для одной входной очереди, если реализация в стационарном состоянии принимает не все стимулы.

Также, если стек или куча ограниченной ёмкости, то их заполнение может привести к постоянной блокировке, если реализация принимает не все стимулы из этого стека или кучи.

♣Однако для *не*ограниченных кучи или стека внутренняя блокировка всегда является *временной*, то есть может быть снята некоторыми действиями теста. Если реализация принимает хотя бы один стимул, то тест

может послать этот стимул, и он окажется в куче или в голове стека, и будет принят реализацией.

При наличии нескольких входных очередей внутренняя блокировка будет постоянной только в том случае, когда реализация принимает стимулы только из тех очередей, которые не пусты, но принимаемые стимулы не совпадают с головными стимулами этих очередей. Если реализация принимает стимул хотя бы из одной пустой очереди, то тест может послать в эту очередь этот стимул, и реализация примет его.

Например, можно считать естественным поведение реализации, которая в некотором стационарном состоянии принимает все стимулы из первой очереди и не принимает, или принимает не все стимулы из второй очереди. Внутренняя блокировка снимается посылкой тестом любого стимула в первую очередь.

♣ В общем случае можно рассматривать системы с *портами*. Принимаемый реализацией или передаваемый тестом стимул принимается или передаётся через некоторый порт. Иными словами, мы расширяем алфавит стимулов: теперь стимул – это пара <номер порта, стимул>. Это эквивалентно разбиению алфавита на непересекающиеся классы, каждому классу соответствует один порт. Важно, что это разбиение и этот набор портов статические и известны не только реализации, но и тесту.

С портом среда может связывать очередь, кучу или стек. Проблему может представлять случай, когда с каждым портом связана очередь, ограниченная куча или ограниченный стек, то есть то, что может вызвать постоянную блокировку. Для того чтобы постоянная блокировка никогда не возникала, реализация должна в каждом стационарном состоянии принимать все стимулы хотя бы из одного порта.

162. Внутренняя блокировка: требования к реализации.

Как правило, в реальной работе реализации с окружением через среду стремятся избежать постоянной блокировки. Для этого нужно сформулировать соответствующие требования к реализации при данном наборе портов и данной среде. Поскольку спецификация как раз и является описанием требований к реализации, то возникает вопрос, в чём заключаются эти требования и как их описывать в спецификации? Фактически, речь идёт о том, что мы должны по спецификации проверить, может ли реализация, конформная этой спецификации, привести к постоянной блокировке. Эта задача, однако, уже не относится к

тестированию соответствия и представляет собой предмет аналитической верификации самой спецификации. Поэтому об этом мы говорить не будем.

С точки зрения тестирования, если внутренняя блокировка, постоянная или временная, допускается спецификацией, то это нормальная ситуация. Иными словами, при тестировании таких реализаций мы уже не можем считать, что в конце шага тестирования, когда обнаружена стационарность, реализация приняла все посланные ей стимулы.

♣Итак, мы ослабляем требование к реализации, разрешая в стационарном состоянии не только приём стимула и разрушение, но и блокировку.

♣Мы уже говорили, что удобнее всего понимать предусловие стимулов в стационарном состоянии как описание безопасных стимулов. Тогда нужна конструкция языка, различающая безопасный принимаемый стимул и безопасный блокируемый стимул. Мы предлагали использовать ключевое слово **block** в постусловии стимула.

♣

163. Внутренняя блокировка: отображение трасс.

При завершении шага тестирования могут остаться стимулы, которые посланы тестом, но не приняты реализацией, то есть «застрявшие» в среде. При завершении шага тестирования гипотетическая трасса уже не обязана содержать все посланные стимулы, хотя она по-прежнему должна содержать все полученные тестом реакции и заканчиваться стационарностью. Здесь возникает проблема отображения наблюдаемой трассы во множество гипотетических трасс.

Как мы можем учесть застрявшие стимулы при построении отображения трасс? Сначала рассмотрим два примера.

Пример 1. Если есть одна входная очередь, то все застрявшие стимулы располагаются в ней в том порядке, в каком мы их посылали на предыдущем шаге и, кроме того, – это последние посланные стимулы. Нам нужно только знать их количество, которое зависит от гипотетических трасс предыдущих шагов. Добавляя последовательность застрявших стимулов в начало наблюдаемой трассы текущего шага, мы получаем модифицированную наблюдаемую трассу, для которой и строим отображение. Иными словами, мы строим гипотетические трассы для модифицированной наблюдаемой трассы, подаваемой в пустую среду. Застрявшие стимулы находятся в начале

модифицированной трассы, но они не будут приняты, то есть снова застрянут в среде, по крайней мере, до тех пор, пока реализация не примет какой-нибудь последующий стимул.

Пример 2. Однако уже для стека такой метод не работает. Например, в стеке застряли два стимула a и b, причём b находится в вершине стека и блокируется реализацией. Если реализация не блокирует стимул a, то любая последовательность ab или ba, подаваемая в пустую среду, вызовет приём стимула a. Таким образом, застрявшие стимулы, как бы они не располагались в начале модифицированной наблюдаемой трассы, снова не застрянут.

Эти примеры показывают, что проблема состоит в том, что состояние среды в начале шага тестирования может быть различным. Но тогда отображение начинает зависеть от этого состояния. Мы уже не можем задавать такое отображение для трассы, наблюдаемой между двумя стационарностями.

♣Общим решением является построение отображения не для наблюдаемой трассы текущего шага, а для полной трассы от начала тестирования. Верификация гипотетической трассы проводится по спецификации, начиная также от её начального состояния.

♣Естественно также считать, что стимулы застревают в среде по вине реализации, а не среды. Среда предлагает какие-то из застрявших стимулов реализации, но та их блокирует; остальные застрявшие стимулы просто ждут своей очереди и не предлагаются средой. Это означает, что в гипотетические трассы должны входить блокировки застрявших стимулов, предлагаемых средой, но блокируемых реализацией.

164. Внутренняя блокировка: Таблица условий.

А теперь посмотрим, какая получается таблица условий для среды, когда реализация может блокировать стимулы в стационарных состояниях. Внешнюю блокировку мы пока запретим.

Теперь в стационарном состоянии реализация может принимать не все стимулы, а лишь некоторые или даже ни одного.

♣♣Поэтому в столбцах, соответствующих стационарному состоянию реализации, и строках, соответствующих наличию стимулов в среде, мы должны сделать изменения.

Теперь мы не можем требовать от среды, чтобы она предлагала реализации тот стимул, которого реализация ждёт в стационарном состоянии. Реализация может не ждать ни одного из тех стимулов, которые есть в среде.

♣Пусть в среде есть стимулы, но нет реакций.

Если тест ждёт реакций, то возможен deadlock, который означает стационарность реализации и отсутствие реакций в среде. При этом стимулы в среде могут оставаться. Среда должна предлагать хотя бы один стимул реализации, так как в противном случае deadlock возникнет по вине среды, а не реализации.

♣Если тест посылает стимул, то мы должны избежать блокировки теста. Но теперь мы уже не можем рассчитывать на то, что реализация обязательно примет предлагаемый средой стимул. Поэтому остаётся требование приёма всех стимулов от теста.

♣Соответственно меняется итоговый столбец условий на среду.

♣Пусть в среде есть и стимулы и реакции.

Если тест ждёт реакций, deadlock не должен возникать, потому что мы договорились, что в конце шага тестирования в среде не должно оставаться реакций. Поскольку мы не можем рассчитывать на то, что реализация обязательно примет предлагаемый средой стимул, остаётся требование выдачи реакции из среды в тест.

♣Если тест посылает стимул, то остаётся требование приёма всех стимулов от теста.

♣Соответственно меняется итоговый столбец условий на среду.

165. Внутренняя блокировка: Правила работы среды.

Итоговые правила работы среды меняются.

♣Теперь среда должна всегда принимать все стимулы от теста.

Если в среде нет реакций, она должна ждать всех реакций от реализации.

♣Если в среде есть реакции, она должна хотя бы одну из них предлагать тесту.

Кроме того, чтобы не было deadlock`а по вине среды она должна предлагать хотя бы один стимул реализации, если в среде есть стимулы и нет реакций.

♣Примером может служить любой набор неограниченных входных очередей, куч и стеков. Реакции могут выдаваться реализацией через любой набор очередей, куч и стеков как неограниченной, так и ограниченной ёмкости.

166. Внешняя и внутренняя блокировка.

Наконец, рассмотрим общий случай, когда возможны и внешняя и внутренняя блокировки.

Наличие внешней блокировки делает тест адаптивным: после тайм-аута в посылающем состоянии тест продолжает работать. Он может попробовать послать другой стимул или ждать реакций.

♣Мы ослабляем требование к реализации, разрешая в стационарном состоянии не только приём стимула и разрушение, но и блокировку.

♣Отображение трасс строится не для наблюдаемой трассы текущего шага, а для полной наблюдаемой трассы от начала тестирования.

♣В гипотетические трассы должны входить блокировки застрявших стимулов, предлагаемых средой, но блокируемых реализацией.

167. Внешняя и внутренняя блокировка: Таблица условий.

Таблица условий совмещает особенности таблиц для внешней и внутренней блокировки.

♣♣

Возможность внутренней блокировки влечёт возможность обнаружения стационарности при отсутствии реакций в среде. Но стимулы могут «застрять» в среде. Если тест ожидает реакций, а в среде есть и стимулы и реакции, то среда не обязана предлагать стимул реализации, поскольку реализация всё равно может такие стимулы блокировать. Важно лишь, чтобы она предлагала реакцию тесту.

Если тест посылает стимул, то поведение среды любое, кроме случая отсутствия в ней стимулов и реакций. В этом случае среда должна принимать стимул от теста.

♣Соответственно меняется итоговый столбец условий.

168. Внешняя и внутренняя блокировка: Правила работы среды.

♣Если реакции в среде нет, среда должна принимать все реакции от реализации. Если в среде есть реакции, среда обязана хотя бы одну из них предлагать тесту.

Что касается стимулов, то поведение среды ограничивается только для случая отсутствия в ней реакций. Если в среде нет стимулов, она обязана принимать все стимулы от теста. Если в среде есть стимулы, она обязана предлагать хотя бы один из них реализации.

Чем отличаются эти правила от случая, когда есть только внешняя блокировка, а внутренней блокировки нет? Только поведением среды при наличии в ней как стимулов, так и реакций.

Если внутренней блокировки нет, то мы разрешали среде на выбор предлагать реакцию тесту или принимать все реакции от реализации и предлагать стимул реализации. Если реализация находится в нестационарном состоянии, она пошлёт реакцию и среда её примет. Если же реализация находится в стационарном состоянии, она обязана принять стимул, предлагаемый средой. В любом случае взаимодействие возможно.

При наличии внутренней блокировки реализация уже не обязана принимать стимул, предлагаемый ей средой. Поэтому у среды такого выбора уже нет. Она обязана выдавать реакцию в тест. Разумеется, это не означает, что

вместе с этим (а не на выбор) она не может предлагать стимулы реализации и/или принимать от неё реакции.

169. ПРОБЛЕМЫ: Осциллирующая реализация. Проблема дивергенции.

Когда мы начинали говорить об асинхронном тестировании, мы наложили запрет на осцилляцию реализации по двум причинам: из-за дивергенции и из-за незавершённости шага тестирования.

♣Сначала рассмотрим проблему дивергенции. Для того чтобы дивергенция не возникала, среда должна периодически выдавать накопленные в ней реакции в тест, не принимая новых реакций от реализации.

Такой режим работы всегда обеспечивается средой, в которой ограничено число хранимых реакций. Например, ограниченные выходные очереди, стеки и кучи.

При неограниченной ёмкости такое поведение среды также возможно. Например, среда принимает от реализации не более n реакций подряд, то есть без выдачи реакций в тест. Если принято подряд n реакций, среда оказывается в состоянии, где только выдаёт реакцию в тест, но не принимает реакции от реализации. В остальных состояниях она может принимать реакции от реализации. Здесь число реакций, которые среда может принять из реализации без выдачи реакций в тест, всегда конечно и ограничено сверху числом n, но число реакций, накапливаемых в среде, может стать сколь угодно большим.

Другим решением было бы установление приоритета выдачи реакций из среды в тест над приёмом реакции из реализации в среду. В этом случае даже при обычной неограниченной выходной очереди тест получал бы реакцию сразу, если он задаёт приём реакций и реакция уже есть в очереди, или, если в очереди нет реакций, как только такая реакция поступит из реализации.

Фактически, речь идёт об изменении работы мистического синхронизатора, реализующего оператор параллельной композиции автоматов. Сейчас при наличии вариантов он выбирает любой из них полностью недетерминированно. Проблема в том, что такое решение выходит за рамки обычной модели асинхронного автомата и параллельной композиции автоматов. Фактически, нам нужно описывать такие приоритеты в спецификации или, по крайней мере, иметь соответствующие умолчания.

Оператор параллельной композиции должен эти приоритеты учитывать. Об этом мы поговорим позже.

170. ПРОБЛЕМЫ: Осциллирующая реализация. Проблема незавершённости шага тестирования.

Предположим теперь, что проблема дивергенции тем или иным способом решена, то есть при тестировании композиция реализации и среды не впадает в дивергенцию. Тогда остаётся проблема незавершённости шага тестирования. Тест может непрерывно принимать реакции из среды и никогда не обнаружит стационарность просто потому, что в реализации есть цикл по выдаче реакций.

В настоящее время есть только идея решения этой проблемы, которую можно назвать идеей *прогнозирующего теста*. Эту идею высказал Витя Кулямин. Получив некоторую последовательность реакций, мы делаем прогноз, что ничего нового мы уже не узнаем: реализация прошла некоторый цикл выдачи реакций и дальше будет только повторять его снова и снова. В этом случае тест переключается на выдачу стимула, пытаясь таким образом столкнуть реализацию с этого цикла. Такой прогноз мы можем сделать только на основании спецификации, в которой есть такой цикл, и реализационной гипотезе, ограничивающей недетерминизм реализации и число её состояний.

♣Объясним эту идею на следующем простом примере. Пусть в спецификации только два состояния и три перехода. В начальном состоянии спецификация принимает один стимул и переходит во второе состояние, в котором выдаёт в цикле-петле одну реакцию и принимает стимул, возвращаясь в начальное состояние: $s_0 — x \to s_1 — y \to s_1 — x \to s_0$.

♣Предположим, что реализация детерминирована в смысле детерминизма LTS: каждое её состояние стабильно и в нём есть не более одного перехода по каждому стимулу и каждой реакции. Предположим также, что в реализации не более трёх состояний.

♣Тогда, обнаружив стационарность в начальном состоянии, послав один стимул и получив три одинаковые реакции, то есть пройдя трассу $\delta xyyy$, мы можем заключить, что никаких других реакций мы больше не получим и реализация не перейдёт в стационарное состояние.

Здесь возможны три реализации, конформные спецификации, если не учитывать возможность приёма стимула не в начальном состоянии:

1) $s_0 —x→ s_1 —y→ s_1$,

2) ♣ $a_0 —x→ a_1 —y→ a_2 —y→ a_2$,

3) ♣ $b_0 —x→ b_1 —y→ b_2 —y→ b_1$.

После такого прогноза мы перестаём принимать реакции, и посылаем в реализацию стимул, пытаясь столкнуть реализацию с цикла выдачи реакций.

♣ Если это удалось, мы должны обнаружить стационарность в принимающем состоянии теста. Тем самым, мы проверим трассу δxyyxδ.

♣ Разумеется, реализация не обязана сходить с цикла выдачи реакций и принимать стимул. Вместо трассы δxyyxδ мы можем получить трассу δxyyxyyy....

♣ Для того чтобы гарантированно стимулом столкнуть реализацию с цикла выдачи реакций, нам опять нужны приоритеты. На этот раз в реализации следует установить приоритет приёма стимулов над выдачей реакций.

В общем, для тестирования осциллирующих реализаций нам нужно, во-первых, уметь сталкивать реализацию с цикла выдачи реакций с помощью приоритетов переходов в автомате или каким-то ещё способом ограничения недетерминированного поведения и, во-вторых, разработать алгоритм генерации прогнозирующих тестов по спецификации, основанный на соответствующей реализационной гипотезе.

171. КОМПОЗИЦИОННОЕ ТЕСТИРОВАНИЕ

172. Композиционное тестирование: Постановка проблемы

173. Основная проблема композиционного тестирования.

Композиционное тестирование – это тестирование системы, собранной из компонентов по известной схеме компоновки с помощью применения определённых правил композиции.

Асинхронное тестирование является частным случаем композиционного тестирования. Здесь система состоит из двух компонентов: реализация и среда. При этом предполагается, что модель среды известна.

♣Основная проблема композиционных систем звучит так: если компоненты работают правильно, то почему система в целом работает неправильно?

В тестировании соответствия правильность реализации компонента определяется как её соответствие спецификации компонента, а правильность системы – как ее соответствие спецификации системы. Разработчик компонента руководствуется единственным документом – техническим заданием, формальной моделью которого является спецификация компонента. Правильность реализации компонента проверяется автономным тестированием компонента. Очевидно, одной из причин неправильной работы системы является то, что, на самом деле, ее компоненты работают неправильно: при автономном тестировании они не были полностью проверены. С учетом бесконечности полного набора тестов такая ситуация вполне возможна и действительно часто встречается на практике.

Однако проблема композиционных систем этим не исчерпывается. Может оказаться, что реализации всех компонентов соответствуют своим спецификациям, в то время как собранная из этих компонентов система не соответствует спецификации системы в целом. В силу вышесказанного претензий к разработчикам компонентов быть не может.

♣Тогда кто же виноват, и что делать?

Очевидно, проблема лежит в иной плоскости: само соотношение спецификаций компонентов и спецификации системы неправильно.

♣А это уже ошибка архитектора, который неправильно декомпозировал спецификацию системы на спецификации ее компонентов. По вполне понятным причинам такие ошибки гораздо хуже ошибок разработчиков, поскольку их труднее обнаруживать, и они имеют более печальные последствия.

С этой точки зрения, более сложное системное или комплексное тестирование не просто продолжает более простое, но незаконченное, автономное тестирование, но предназначено также для обнаружения ошибок архитектурного уровня, которые не обнаруживаются автономными тестами. Последствиями таких ошибок является изменение спецификаций, иногда достаточно радикальное, что требует модификации или даже повторной реализации всех или части компонентов.

174. Монотонность.

Какое соотношение спецификаций компонентов и спецификации системы следует признать правильным? Очевидно, такое, которое удовлетворяет следующему условию монотонности: любые реализации компонентов, соответствующие своим спецификациям, образуют систему, соответствующую спецификации системы.

Замечу, что отсюда вовсе не следует, что, если реализации компонентов образуют систему, соответствующую спецификации системы, то сами эти реализации компонентов соответствуют спецификациям компонентов. Мы этого не требуем, да нас это и не должно волновать: важно лишь, чтобы система отвечала функциональным требованиям, зафиксированным в системной спецификации, а как она это делает – не имеет значения. Это общая ситуация, частным случаем которой является то, что мы называли вседозволенностью асинхронного тестирования.

♣Корректной спецификацией системы можно назвать такую спецификацию системы, которая удовлетворяет условию монотонности при заданных спецификациях компонентов.

♣Самая сильная (то есть, предъявляющая максимальные функциональные требования) корректная спецификация системы определяется самим условием монотонности при заданных спецификациях компонентов и схеме компоновки.

Если задана спецификация системы, то верификация её согласованности со спецификациями компонентов заключается в проверке того, что она корректна, то есть, эквивалентна или слабее самой сильной корректной спецификации. Отношение определяется соответствием $\boldsymbol{ioco}_{\beta\gamma\delta}$: самая сильная корректная спецификация системы должна быть конформна заданной спецификации системы.

Если у нас вообще нет спецификации системы, то мы могли бы её построить как самую сильную корректную спецификацию.

♣Здесь, однако, возникают три проблемы:

1) Определение самой сильной корректной спецификации неявно.
2) Существует ли такая самая сильная корректная спецификация?
3) Если она существует, то как её построить по спецификациям компонентов и схеме компоновки?

175. *Косая* композиция.

Спецификационные модели компонентов – это безопасные и безопасно-конвергентные асинхронные автоматы без θ-переходов. Реализации компонентов – это асинхронные автоматы, конформные соответствующим спецификационным моделям. Правила композиции реализаций компонентов моделируются оператором $\|$ параллельной композиции.

♣Первой неожиданностью стало то, что самая сильная корректная спецификация системы, если она существует, может оказаться слабее композиции спецификаций компонентов, если эту композицию проводить по тем же правилам, что и композицию реализаций компонентов при сборке системы.

♣Несохранение соответствия при асинхронном тестировании можно рассматривать как частный случай этой проблемы. Здесь система состоит из двух компонентов: реализация и среда, которые компонуются друг с другом.

♣Если композицию с помощью оператора $\|$ называть реализационной, то оказалось, что требуется иная – *спецификационная* – композиция спецификаций, вычисляющая самую сильную корректную спецификацию системы, если она существует, как функцию спецификаций двух компонентов. Спецификационную композицию будем обозначать косым

знаком $/\!/$. Для удобства терминологии такую композицию будем называть *косой композицией* асинхронных автоматов.

Причиной такого положения, то есть несовпадения прямой и косой композиции, является разный уровень абстракции, используемый в определении соответствия и в определении прямой композиции. Соответствия основаны на наблюдаемых трассах – последовательностях наблюдаемых действий реализационной модели, а композиция – на полной модели реализации. Отношение $\boldsymbol{ioco}_{\beta\gamma\delta}$ основано на следующих наблюдаемых действиях: прием или блокировка стимула, выдача реакции или отсутствие выдачи какой-либо реакции (стационарность). Прямая композиция дополнительно учитывает состояния, ненаблюдаемые действия (τ-переходы) и соотношение состояний и действий.

176. Компонуемость спецификаций.

Частный случай общей проблемы композиции – это то, что прямая композиция конформных реализаций может оказаться не безопасно-тестируемой относительно косой композиции спецификаций компонентов.

♣Это может быть в том случае, когда косая композиция спецификаций не безопасна или не безопасно-конвергентна. Поэтому после построения косой композиции её нужно верифицировать на безопасность и безопасно-конвергентность.

♣Спецификации будем называть *компонуемыми*, если их косая композиция безопасна и безопасно-конвергентна.

177. Формальные определения.

На этом слайде даны формальные определения корректной спецификации композиции и косой композиции.

Соответствие $\boldsymbol{ioco}_{\beta\gamma\delta}$ является отношением предпорядка: рефлексивно и транзитивно. Поэтому для него можно определить нижний и верхний конусы.

Нижний конус автомата S – это множество автоматов, конформных автомату S. Верхний конус автомата S – это, наоборот, множество автоматов, которым конформен автомат S.

Корректная спецификация композиции – это автомат из верхнего конуса прямой композиции нижних конусов спецификаций: такой автомат, которому конформна композиция любых реализаций, конформных данным спецификациям.

Косая композиция – как самая сильная корректная спецификация – это супренум прямой композиции нижних конусов спецификаций: такая корректная спецификация композиции, которой конформна любая другая корректная спецификация композиции.

Заметим, что косая композиция спецификаций неоднозначно определяет асинхронный автомат. По сути, мы только наложили ограничения на такой автомат. Тем самым, косую композицию можно понимать как множество автоматов, удовлетворяющих этому ограничению. Наша задача – суметь построить по заданным спецификациям компонентов хотя бы один такой автомат.

178. Преобразование спецификаций.

Основная идея построения косой композиции заключается в следующем. Мы хотим найти такое преобразование $\mathcal{F}$ спецификаций компонентов, чтобы косая композиция исходных спецификаций была равна прямой композиции преобразованных спецификаций: $\mathbf{A} /\!\!/ \mathbf{B} = \mathcal{F}(\mathbf{A}) \upharpoonright\!\downarrow \mathcal{F}(\mathbf{B})$. В этом случае будем говорить, что соответствие *монотонно относительно преобразования* $\mathcal{F}$.

Вообще говоря, таких преобразований $\mathcal{F}$ может быть много.

♣ Существование хотя бы одного преобразования показывается легко. Для этого достаточно взять объединение всех реализаций, конформных данной спецификации:
$\mathcal{M}(\mathbf{S}) = \cup(\mathbf{S}^{\triangledown})$.

♣ Объединение автоматов строится так.

♣ Добавляется новое начальное состояние,

♣ и из него проводятся τ-переходы в начальные состояния объединяемых автоматов.

♣Понятно, что такое преобразование неконструктивно: число конформных реализаций бесконечно. Поэтому ставится задача найти такое другое преобразование, которое можно было бы осуществлять алгоритмическим способом.

Построение может быть итеративным, что важно для бесконечных спецификаций. Итеративность будем понимать так, что для любого, наперёд заданного, числа n мы можем построить часть преобразованной спецификации, которая содержала бы все безопасные трассы длиной n.

179. Цели косой композиции.

Итак, построение косой композиции преследует три цели.

1. Проверка компонуемости, то есть проверка того, что косая композиция спецификаций компонентов безопасна и безопасно-конвергентна. Это гарантирует, что композиция любых конформных реализаций будет безопасно-тестируемой для любой корректной спецификации системы.

2. Проверка корректности системной спецификации, то есть проверка того, что косая композиция спецификаций компонентов конформна системной спецификации.

3. Генерация системных тестов по косой композиции спецификаций компонентов. Это полезно, когда у нас нет корректной системной спецификации, или эта спецификация недостаточно сильная для проверки интересующих нас свойств системы.

180. Композиционное тестирование: Блокировка и разрушение

Сейчас мы более детально исследуем причины немонотонности соответствия. Таких причин две: блокировки стимулов и разрушение. Мы также рассмотрим спецификации, на которые наложены ограничения по блокировке и разрушению. Для таких спецификаций можно определить очень простые преобразования, относительно которых соответствие монотонно.

181. Пример немонотонности из-за блокировки.

Сначала покажем, что, если в спецификации есть блокировки стимулов, то соответствие $\boldsymbol{ioco}_{\beta\gamma\delta}$ немонотонно относительно тождественного преобразования спецификаций. Иными словами, при наличии блокировок стимулов прямая композиция не является косой композицией.

Для этого достаточно рассмотреть вот такой простой пример.

♣ В первой строке показаны спецификация и конформная ей реализация.

♣ Во второй строке спецификация и реализация совпадают.

♣ В третьей строке показаны композиции спецификаций и реализаций.

♣ Мы видим, что трасса {?x} является общей для прямой композиции спецификаций и прямой композиции реализаций. Однако в композиции реализаций эта трасса продолжается реакцией !y, а в композиции спецификаций – не продолжается.

182. Спецификации без блокировок.

Если в спецификации нет блокировок, то соответствие $\boldsymbol{ioco}_{\beta\gamma\delta}$ можно называть соответствием $\boldsymbol{ioco}_{\gamma\delta}$. Для таких спецификаций оно монотонно относительно тождественного преобразования: $\mathcal{F}(\mathbf{S})=\mathbf{S}$. Иными словами, прямая композиция спецификаций без блокировок является также и косой композицией.

♣ Именно поэтому при пополнении спецификаций стремятся избавиться от блокировок. В частности, пополнение спецификации, основанное на правиле приоритета стимула над его блокировкой и правиле сохранения поведения после стационарности и, после этого, демоническом пополнении, сохраняет

классическую семантику отношения ***ioco*** и обеспечивает монотонность соответствия для пополненных спецификаций без дальнейших преобразований.

183. Пример немонотонности из-за разрушения.

При наличии разрушения довольно странно требовать отсутствия блокировок после приёма разрушающего стимула. В этом случае поведение автомата считается неопределённым с точки зрения соответствия $\boldsymbol{ioco}_{\beta\gamma\delta}$.

Поэтому мы ослабим требование отсутствия блокировок: потребуем, чтобы блокировок не было в безопасных βγδ-трассах.

Это, однако, уже не гарантирует монотонность соответствия при тождественном преобразовании спецификаций.

Рассмотрим пример. Здесь тоже

♣ В первой строке показаны спецификация и конформная ей реализация.

♣ Во второй строке спецификация и реализация совпадают.

♣ В третьей строке показаны композиции спецификаций и реализаций.

♣ Мы видим здесь, что трасса ?x является общей для прямой композиции спецификаций и прямой композиции реализаций. Однако в композиции реализаций эта трасса продолжается тем же стимулом ?x, а композиция спецификаций – только блокировкой {?x}.

Если внимательно изучить этот пример, то можно увидеть, что немонотонность возникает из-за того, что в спецификации разрушение идёт не сразу после разрушающего стимула, а после некоторых реакций.

184. Гамма-нормализованные спецификации без блокировок в безопасных трассах.

Будем называть спецификацию *гамма-нормальной*, если в ней после перехода по разрушающему стимулу сразу идёт переход по разрушению.

♣ Гамма-нормализацией назовём соответствующее преобразование

спецификаций. Это преобразование добавляет гамма-петлю в конце перехода по разрушающему стимулу. Такое состояние становится гамма-состоянием.

Прежде всего, отметим, что гамма-нормализованная спецификация $\boldsymbol{ioco}_{\beta\gamma\delta}$-эквивалентна исходной спецификации.

♣Если спецификации гамма-нормализованы и в их безопасных трассах нет блокировок, то соответствие $\boldsymbol{ioco}_{\beta\gamma\delta}$ также монотонно относительно тождественного преобразования спецификаций. Тем самым, для спецификаций, не содержащих блокировок в безопасных трассах, соответствие монотонно относительно гамма-нормализации: $\mathcal{F}(\mathbf{S})=\mathcal{N}(\mathbf{S})$.

♣Теперь при пополнении спецификаций после преобразований по правилам приоритета и сохранения мы применяем не демоническое пополнение, а пополнение разрушения. Такое преобразование также сохраняет классическую семантику отношения ***ioco***, но для более широкого класса спецификаций, поскольку в небезопасных трассах допускаются блокировка и дивергенция. Поскольку полученная спецификация гамма-нормальна, обеспечивается монотонность соответствия для пополненных спецификаций без дальнейших преобразований.

185. Композиционное тестирование: Общий случай

Теперь мы перейдём к общему случаю, когда на спецификации не накладывается никаких ограничений, кроме безопасности и безопасно-конвергентности.

186. Достаточные условия монотонности.

Введём обозначения:

$\mathbf{\Psi}$ – *множество всех βγδ-трасс;*
ψ: $\mathbf{AA} \to \wp(\mathbf{\Psi})$ – *отображение, которое каждому асинхронному автомату ставит в соответствие множество его βγδ-трасс;*

♣ $\mathbf{\Phi}$ – *множество трасс, которые мы будем называть ϕ-трассами; определение ϕ-трасс мы дадим позже, а здесь сформулируем только требования к ним;*
ϕ: $\mathbf{AA} \to \wp(\mathbf{\Phi})$ – *отображение, которое каждому асинхронному автомату ставит в соответствие множество его ϕ-трасс;*
π: $\mathbf{\Phi} \to \wp(\mathbf{\Psi})$ – *отображение, котороpое преобразует ϕ-трассу во множество βγδ-трасс;*

♣ Γ: $\wp(\mathbf{\Psi}) \to \wp(\mathbf{\Psi})$ – *отображение, преобразующее множество βγδ-трасс в другое, расширенное, множество βγδ-трасс; мы будем называть его гамма-расширением δγβ-трасс; определение гамма-расширения мы дадим позже, а здесь сформулируем только требования к нему;*

♣ $\lfloor\phi\rfloor$: $\mathbf{\Phi} \times \mathbf{\Phi} \to \wp(\mathbf{\Phi})$ – *отображение, которое по паре ϕ-трасс строит множество ϕ-трасс; мы будем называть его композицией ϕ-трасс; определение композиции ϕ-трасс мы дадим позже, а здесь сформулируем только требования к нему;*

♣ $\mathcal{M}_\phi(\mathbf{S})$ – *максимальная ϕ-реализация для спецификации* $\boldsymbol{S}$*: такой автомат, множество ϕ-трасс которого равно объединению множеств ϕ-трасс всех конформных реализаций:*
$\phi \circ \mathcal{M}_\phi(\mathbf{S}) = \cup \circ \phi(\mathbf{S}^\nabla)$.

♣ Условия, которые нам нужны, следующие:

1. Эквивалентность соответствия вложенности гамма-расширений βγδ-трасс:
   $\mathbf{I}$ $\boldsymbol{ioco}_{\beta\gamma\delta}$ $\mathbf{S} \Leftrightarrow \Gamma \circ \psi(\mathbf{I}) \subseteq \Gamma \circ \psi(\mathbf{S})$.

2. ♣Свойства гамма-расширения:

   $A \subseteq \Gamma(A)$ & $\Gamma \circ \Gamma = \Gamma$ & ($A \subseteq B \Rightarrow \Gamma(A) \subseteq \Gamma(B)$).

   Левое свойство: гамма-расширение действительно является расширением, то есть, может долько добавлять βγδ-трассы, но не удалять их.
   Среднее свойство: гамма-расширение добавляет все нужные βγδ-трассы, то есть, его повторное применение ничего не меняет.
   Правое свойство: гамма-расширение сохраняет вложнность множеств βγδ-трасс.

3. ♣Производность βγδ-трасс от ϕ-трасс: $\psi = \cup \circ \pi \circ \varphi$.

   βγδ-трассы автомата могут быть получены из его ϕ-трасс с помощью оператора π.

4. ♣Аддитивность ϕ-трасс относительно композиции:

   $\phi(A \wr B) = \cup(\phi(A) \wr_{\varphi} \phi(B))$.

   ϕ-трассы композиции автоматов могут быть получены как объединение всех попарных композиций ϕ-трасс этих автоматов.

5. ♣Существует максимальная ϕ-реализация: $\exists \mathcal{M}_{\phi}(S)$.

   То есть для каждой спецификации существует автомат, множество ϕ-трасс которого равно объединению множеств ϕ-трасс всех конформных реализаций.

♣Если эти условия выполнены, то в качестве преобразования спецификации можно взять преобразование $\mathcal{M}_{\phi}$, то есть соответствие $\boldsymbol{ioco}_{\beta\gamma\delta}$ $\mathcal{M}_{\phi}$-монотонно. Отметим, что преобразование $\mathcal{M}_{\phi}$ так же, как преобразование $\mathcal{M}$, неоднозначно, то есть максимальных ϕ-реализаций может быть много. Нам нужно суметь построить хотя бы одну из них.

187. Условия монотонности 1 и 2: Гамма-расширение

$\Gamma: \wp(\Psi) \to \wp(\Psi)$.

Определим гамма-расширение спецификации – это такое преобразование βγδ-трасс, при котором соответствие $\boldsymbol{ioco}_{\beta\gamma\delta}$ было бы эквивалентно вложенности гамма-расширенного множества βγδ-трасс реализации в гамма-расширенное множество βγδ-трасс спецификации.

♣Неявно гамма-расширение можно определить как объединение всех βγδ-трасс всех конформных реализаций.

♣Явное определение описывает способ построения βγδ-трасс гамма-расширения из βγδ-трасс исходного автомата. Интуиция подсказывает, что гамма-расширение – это добавление трасс, порождаемых трассами с разрушением. Если в спецификации есть стимул, разрушающий после трассы, то отношение $\boldsymbol{ioco}_{\beta\gamma\delta}$ не накладывает никаких ограничений на реализацию по поводу этого стимула: реализация может его принимать или блокировать с любым дальнейшим поведением, если такое дальнейшее поведение не индуцирует запрещённое поведение в безопасных трассах.

Мы рассмотрим два случая: приём и блокировка разрушающего стимула.

Приём разрушающего стимула может иметь в качестве продолжения любую возможную βγδ-трассу.

Если реализация блокирует стимул x после трассы μ, то в ней после трассы $\mu\cdot\langle\{x\}\rangle$ может быть, вообще говоря, не любая βγδ-трасса. Действительно, наличие трассы $\mu\cdot\langle\{x\}\rangle\cdot\lambda$ влечёт наличие трассы $\mu\cdot\lambda$, а на такую трассу конформность реализации может накладывать ограничения, если эта трасса безопасна в спецификации. Сформулируем условия, при которых гамма-расширение спецификации может для безопасной трассы $\mu\cdot\lambda$ вставить между μ и λ блокировку стимула $\{x\}$, разрушающего после μ, не нарушая конформности (сохраняя требования к реализации).

Первое условие: в трассе λ первый символ, отличный от блокировки или стационарности, не совпадает с x. Это следует из того, что после блокировки стимула не может следовать этот стимул.

Второе условие: вставку можно делать, если между трассами μ и λ до гамма-расширения существовали отказы, или трасса μ до гамма-расширения продолжалась всеми стимулами и хотя бы одной реакцией. Дело в том, что блокировка стимула возможна только в стабильном состоянии, и это условие означает, что между трассами μ и λ в конформной реализации может быть стабильное состояние.

Действительно, если второе условие нарушено, то до гамма-расширения трасса μ либо a) не продолжалась некоторым стимулом $x`$, либо b) не продолжалась никакой реакцией. Поскольку стимул, разрушающий после трассы, продолжает эту трассу, стимул $x`$ неразрушающий после трассы μ. Поэтому для сохранения конформности свойство a) или

b) должно остаться и после гамма-расширения. С другой стороны, после гамма-расширения этим свойством должна обладать также трасса $\mu\cdot\langle\{x\}\rangle$, так как любое продолжение после этой трассы есть и после трассы μ. Следовательно, поскольку после гамма-расширения существует трасса $\mu\cdot\langle\{x\}\rangle\cdot\lambda$, должна существовать либо a) трасса $\mu\cdot\langle\{x\}\rangle\cdot\langle\{x`\}\rangle\cdot\lambda$, либо b) трасса $\mu\cdot\langle\{x\}\rangle\cdot\langle\delta\rangle\cdot\lambda$. Но тогда после гамма-расширения существует либо a) трасса $\mu\cdot\langle\{x`\}\rangle\cdot\lambda$, либо b) трасса $\mu\cdot\langle\delta\rangle\cdot\lambda$. Поскольку второе условие не выполнено, между трассами μ и λ до гамма-расширения не существовали отказы. Следовательно, трасс $\mu\cdot\langle\{x`\}\rangle\cdot\lambda$ и $\mu\cdot\langle\delta\rangle\cdot\lambda$ до гамма-расширения не было, поэтому они должны появиться в результате гамма-расширения. Эти новые трассы не нарушают конформность только в том случае, когда они опасны. Однако, поскольку трасса $\mu\cdot\lambda$ безопасна, трасса $\mu\cdot\langle\delta\rangle\cdot\lambda$ также безопасна. Аналогично, трасса $\mu\cdot\langle\{x`\}\rangle\cdot\lambda$ опасна только, если стимул $x`$ разрушающий после трассы μ, что неверно.

188. Разрушение.

Посмотрим, как можно реализовать произвольное поведение после приёма разрушающего стимула.

♣Состояние в конце перехода по разрушающему стимулу назовём состоянием «разрушение». Определим в нём τ-переход в каждое стабильное состояние, определяемое подмножеством Z стимулов и реакций, по которым нет перехода из этого стабильного состояния.

♣Прежде всего, в этом стабильном состоянии определим гамма-переход.

♣Во-вторых, мы должны в этом стабильном состоянии определить переход по каждому символу, не входящему в Z. Тем самым, мы определим все возможные βγδ-трассы.

♣Наконец, поскольку произвольное поведение допускает дивергенцию после любой βγδ-трассы, определим в состоянии «разрушение» τ-петлю.

189. ϕ-трассы $\phi:\mathbf{AA}\to\wp(\Phi)$.

Множества βγδ-трасс компонентов неоднозначно определяют множество βγδ-трасс композиции. Теперь мы рассмотрим разновидность трасс, для которых такая однозначность есть. Мы будем называть их ϕ-трассами.

♣При построении βγδ-трасс проход через стабильное состояние добавляет в трассу блокировки стимулов, означающие отсутствие переходов по этим стимулам в этом состоянии, и стационарность, означающую отсутствие переходов по реакциям в этом состоянии. При построении ϕ-трасс проход через стабильное состояние добавляет в трассу сразу всё множество отвергаемых стимулов и реакций, то есть стимулов и реакций, по которым не определены переходы из данного состояния. Это множество мы будем называть ϕ-множеством, мы уже обозначали его как ***ref***(s), где s – стабильное состояние. Более строго, проход стабильного состояния добавляет в трассу любое (в том числе нулевое) число ϕ-множеств этого состояния. Это можно понимать так, что в каждом стабильном состоянии определён переход-петля, помеченный ϕ-множеством состояния.

ϕ-трассы похожи на трассы готовности (ready traces), но только в последних используется не множество ***ref***(s) отвергаемых символов, а, наоборот, множество ***init***(s) символов, по которым есть переходы из состояния.

190. Условие монотонности 3: ϕ-трассы и βγδ-трассы: $\psi = \cup \circ \pi \circ \phi$**.**

ϕ-трасса порождает множество βγδ-трасс. Соответствующее отображение – это оператор π.

♣Например, пусть у нас есть ϕ-трасса.

♣Оператор π каждое ϕ-множество заменяет последовательностью в алфавите, состоящем из блокировок отвергаемых стимулов, то есть стимулов, принадлежащих ϕ-множеству, и стационарности, если отвергаются все реакции, то есть ϕ-множество содержит все реакции. Очевидно, что для любой цепочки переходов каждая βγδ-трасса, соответствующая этой цепочке, получается с помощью оператора π из некоторой ϕ-трассы, соответствующей этой цепочке.

♣ϕ-трассу будем называть безопасной, если она порождает хотя бы одну безопасную βγδ-трассу.

191. Условие монотонности 4: Аддитивность ϕ-трасс: $\phi(A \between B) = \cup(\phi(A) \between_{\varphi} \phi(B))$**.**

Теперь мы определим композицию ϕ-трасс.

0) Композиция двух пустых ϕ-трасс состоит из одной пустой ϕ-трассы.
1) 2) Если одна из ϕ-трасс продолжается внешним стимулом или реакций, или расширением γ, то есть символом, переход по которому выполняется асинхронно, то композиционная ϕ-трасса также продолжается этим символом.
3) Если обе ϕ-трассы продолжаются противоположными символами, то есть выполняется синхронный переход, который в композиционном автомате изображается как τ-переход, то композиционная ϕ-трасса остаётся той же.
4) Если обе ϕ-трассы продолжаются ϕ-множествами a и b, то композиционная ϕ-трасса продолжается значением функции Δ от a и b.

♣Рассмотрим подробнее последний, 4-ый случай. Сформулируем условие стабильности композиционного состояния. Очевидно, оно будет стабильным только в том случае, когда оба состояния компонентов стабильны. Однако это необходимое, но не достаточное условие. Если состояния обоих компонентов стабильны, то для стабильности композиционного состояния дополнительно требуется, чтобы не было ни одного синхронного перехода. A\a – это множество символов переходов из состояния левого операнда, B\b – это множество символов переходов из состояния правого операнда. Синхронного перехода не будет, если для каждого символа из одного множества в другом множестве нет противоположного символа: $(A\backslash a) \cap (\underline{B\backslash b}) = \varnothing$.

Если условие стабильности нарушено, композиционная ϕ-трасса остаётся той же, то есть не продолжается никаким ϕ-множеством. В этом случае функция Δ возвращает пустую трассу.

Если же условие стабильности выполнено, то композиционная ϕ-трасса продолжается ϕ-множеством, которое состоит из тех внешних символов, по которым нет переходов ни в одном из компонентов. $A\backslash\underline{B}$ – это внешние символы левого операнда. $(A\backslash\underline{B})\backslash a$ – это внешние символы левого операнда, по которым в нём есть переходы. Соответственно, $(B\backslash\underline{A})\backslash b$ – это внешние символы правого операнда, по которым в нём есть переходы. Объединение этих множеств – это все внешние символы, по которым есть переходы из композиционного состояния. Алфавит композиции – это $C=A\backslash\underline{B} \cup B\backslash\underline{A}$. Вычитая из него символы, по которым есть переходы их композиционного состояния, получаем ϕ-множество композиционного состояния: $C\backslash((A\backslash\underline{B})\backslash a)\cup(B\backslash\underline{A})\backslash b)$.

Две ϕ-трассы компонуемы, если их композиция не пуста.

Какие ϕ-трассы компонуемы? Возьмём в каждой ϕ-трассе подпоследовательность, получающуюся удалением внешних стимулов и реакций, и символа разрушения, то есть тех символов, переходы по которым выполняются асинхронно. Тогда эти подтрассы компонентов должны иметь одинаковую длину и на каждой i-ой позиции должны находиться либо 1) противоположные внутренние символы, что соответствует синхронному переходу, либо 2) ϕ-символы. Это второе условие (соответствие ϕ-символов) всегда можно выполнить, если выполнено первое условие (соответствие противоположных символов), подбором нужного количества идущих подряд ϕ-символов. Поэтому-то для удобства определения композиции ϕ-трасс мы разрешили ϕ-символ повторять в ϕ-трасе любое число раз.

Определённая таким образом композиция ϕ-трасс оказывается аддитивной: множество ϕ-трасс композиции равно объединению всех попарных композиций ϕ-трасс компонентов.

192. Условие 5: Существование максимальной реализации.

Максимальная реализация, как объединение всех конформных реализаций, очевидно, содержит те, и только те ϕ-трассы, которые есть в конформных реализациях. Тем самым существование максимальной ϕ-реализации доказано:

$\phi\circ\mathcal{M}(\mathbf{S}) = \phi\circ\cup(\mathbf{S}^{\triangledown}) = \cup\circ\phi(\mathbf{S}^{\triangledown})=\phi\circ\mathcal{M}_{\phi}(\mathbf{S})$.

Определение максимальной ϕ-реализации через ϕ-трассы всех конформных реализаций всё ещё неконструктивно. Вопрос стоит так: как можно получить ϕ-трассы конформных реализаций, глядя на ϕ-трассы спецификации?

♣ Максимальных ϕ-реализаций может быть много. Мы ставим задачу найти способ *построения* одной такой максимальной ϕ-реализации по заданной спецификации.

193. Композиционное тестирование: Базовые и нормальные ϕ-трассы

194. Базовые ϕ-трассы.

Среди ϕ-трасс максимальной ϕ-реализации, на самом деле, есть лишние с точки зрения косой композиции.

♣Вполне достаточно ограничиться только некоторыми из них, которые мы будем называть базовыми ϕ-трассами.

Формально базовые ϕ-трассы – это ϕ-трассы, производные от ϕ-трасс автомата. Они задаются отображением множеств ϕ-трасс ***Base***.

Максимальная реализация, однако, содержит все свои базовые ϕ-трассы.

Главное свойство базовых ϕ-трасс состоит в том, что композиция множеств ϕ-трасс максимальных реализаций $\boldsymbol{ioco}_{\beta\gamma\delta}$-эквивалентна композиции подмножеств базовых ϕ-трасс.

Поэтому вместо того, чтобы строить автомат, ϕ-трассы которого – это все ϕ-трассы максимальной реализации, нам достаточно построить автомат, все базовые ϕ-трассы которого – это все базовые ϕ-трассы максимальной реализации.

♣На самом деле мы построим автомат, все ϕ-трассы которого – это ровно все базовые ϕ-трассы максимальной реализации $\mathcal{M}_{\mathtt{base}\phi}(\mathbf{S})$.

♣Мы рассмотрим три типа ϕ-трасс: финальные, релевантные и сингулярные ϕ-трассы. Базовые ϕ-трассы – это ϕ-трассы, которые одновременно финальные, релевантные и сингулярные.

195. Финальные ϕ-трассы.

Финальные ϕ-трассы определяются на основе правила: «после разрушающего стимула произвольное поведение».

♣ *Финальные* ϕ-трассы – это ϕ-трассы вида σ, σ·x и σ·x·γ, где σ - безопасная ϕ-трасса, а стимул x разрушающий после σ.

♣ Пусть ϕ-трасса максимальной реализации не финальна и имеет вид **a)** σ·x·λ, где ϕ-трасса σ безопасна, стимул x разрушающий после σ. Поскольку после разрушающего стимула поведением конформной реализации может быть произвольным, есть такая конформная реализация, которая сразу разрушается.

♣ Значит, максимальная реализация имеет ϕ-трассу **b)** σ·x·γ.

♣ При композиции ϕ-трасса **a)** σ·x·λ компонуется с некоторой ϕ-трассой σ`·λ`, где σ` компонуется с σ·x, а λ` с λ.

Тогда ϕ-трасса **b)** σ·x·γ компонуется с ϕ-трассой σ`.

♣ Таким образом, в композиции любая из ϕ-трасс σ·x1⌊σ` продолжается двумя способами: для **a)** ϕ-трассой из λ1⌊λ` и для **b)** разрушением γ.

♣ При гамма-расширении композиции всё, что порождают композиционные ϕ-трассы для случая **a)**, порождается композиционными ϕ-трассами для случая **b)**. А это и означает, что достаточно ограничиться только финальными ϕ-трассами.

196. Релевантные ϕ-трассы.

Релевантные ϕ-трассы определяются на основе правила: «разрушающий стимул можно как принимать, так и блокировать».

♣ *Релевантные* ϕ-трассы – это ϕ-трассы, ϕ-множества которых содержат только такие стимулы, блокировки которые входят в порождаемые безопасные βγδ-трассы.

♣ Пусть ϕ-трасса максимальной реализации не релевантна и имеет вид **a)** σ·R·λ, где ϕ-трасса σ безопасна, а R – ϕ-множество, в которое входит стимул x, не используемый при порождении безопасных βγδ-трасс.

♣Тогда есть такая конформная реализация, в которой есть ϕ-трасса, отличающаяся только тем, что в это ϕ-множество не входит стимул x, то есть ϕ-трасса **b)** σ·R\{x}·λ. Тогда после ϕ-трассы σ этот стимул x принимается и является разрушающим. В максимальной реализации после приёма такого стимула сразу должно быть разрушение.

♣При композиции ϕ-трасса **a)** σ·R·λ компонуется с некоторой ϕ-трассой σ`·R`·λ`, где σ` компонуется с σ и даёт σ``, R` компонуется с R, а λ` с λ и даёт λ``.

♣Тогда с этой же ϕ-трассой σ`·R`·λ` компонуется ϕ-трасса **b)** σ·R\{x}·λ, причём σ` компонуется с σ и даёт ту же самую трассу σ``, R` компонуется с R\{x}, а λ` компонуется с λ и даёт ту же самую трассу λ``.

Рассмотрим два варианта в зависимости от того, является ли композиционное состояние в случае **a)** стабильным или нет.

♣<u>Композиционное состояние после ϕ-трассы σ`` нестабильно в случае **a)**</u>.

♣Тогда композиционное состояние после ϕ-трассы σ`` нестабильно и в случае **b)**. Поэтому все βγδ-трассы, порождаемые в композиции в случае **a)**, порождаются также в случае **b)**.

♣<u>Композиционное состояние после ϕ-трассы σ`` стабильно в случае **a)**</u>. Здесь возможны три подварианта в зависимости от стимула x: он внешний или внутренний и, если он внутренний, происходит по нему синхронизация или нет в случае **b)**.

♣<u>Стимул x внешний</u>. После композиционной ϕ-трассы σ`` композиционное состояние стабильно и в случае **b)**, но стимул x будет приниматься и далее идёт разрушение. Поэтому при гамма-расширении композиции всё, что порождают композиционные ϕ-трассы для случая **a)**, порождается композиционными ϕ-трассами для случая **b)**.

♣<u>Стимул x внутренний, но не синхронизируемый</u>. Случаи **a)** и **b)** в композиции вообще не различаются. Поэтому и в этом варианте при гамма-расширении композиции всё, что порождают композиционные ϕ-трассы для случая **a)**, порождается композиционными ϕ-трассами для случая **b)**.

♣Стимул x внутренний и происходит синхронизация по передаче этого стимула. В случае **b)** композиционная ϕ-трасса σ`` становится разрушающей. Поэтому и в этом варианте при гамма-расширении композиции всё, что порождают композиционные ϕ-трассы для случая **a)**, порождается композиционными ϕ-трассами для случая **b).**

♣Таким образом, во всех вариантах можно оставить только такие ϕ-трассы, которые «более» релевантны. В конечном счёте, остаются только релевантные ϕ-трассы.

197. Сингулярные ϕ-трассы.

Сингулярные ϕ-трассы определяются на основе правила *may&must*: «можно выдавать не все реакции, допускаемые спецификацией, но стационарность должна сохраняться, то есть вообще не выдавать реакции можно только тогда, когда в спецификации нет реакций».

Для того чтобы два ϕ-множества порождали одинаковые последовательности отказов (блокировок и стационарностей), они должны содержать одни и те же стимулы. Однако реакции учитываются «все скопом» как стационарность. Поэтому эти ϕ-множества должны либо оба содержать все реакции (стационарность), либо оба не содержать всех реакций (нестационарность). В последнем случае они могут содержать разные подмножества реакций.

♣Сингулярные ϕ-трассы – это ϕ-трассы, в которых ϕ-множества содержат либо все реакции (стационарность), либо все реакции, кроме одной. В сингулярном состоянии, то есть состоянии с сингулярным ϕ-множеством, либо не определены переходы по реакциям, либо определены переходы ровно по одной реакции.

♣Пусть в конформной реализации есть стабильное состояние s, которое достижимо по безопасной βγδ-трассе, и в котором определены переходы по двум реакциям a и b.

♣Преобразуем реализацию, сдублировав это состояние так, что в одном из двух состояний s_a удаляются переходы по реакции b, а в другом s_b – по реакции a. Очевидно, преобразованная реализация также конформна.

Поэтому максимальная реализация вместе с каждой ϕ-трассой содержит все сингулярные ϕ-трассы, которые из неё могут быть получены аналогичным расщеплением ϕ-множеств. Это означает, что, если такую процедуру расщепления стабильных состояний мы будем делать в максимальной ϕ-реализации, то мы не изменим её ϕ-трассы. Иными словами такое расщепление оставляет реализацию ϕ-максимальной.

Покажем, что для композиции можно оставить в максимальной ϕ-реализации только сингулярные ϕ-трассы. Для этого достаточно показать, что после расщепления стабильного состояния максимальной ϕ-реализации исходное состояние можно удалить. Если такую процедуру проделать по всем стабильным состояниям и всем реакциям, то у нас останутся только сингулярные состояния.

Мы покажем, что при композиции для любого композиционного маршрута, начинающегося в паре состояний ss`, есть маршрут с той же βγδ-трассой, начинающийся в паре состояний $s_a$s` или s_bs`.

♣Пусть композиционное состояние ss` нестабильно, но не из-за переходов по реакциям a и b, то есть либо парное состояние s` нестабильно, либо есть внутренний символ, отличный от a и b, по которому есть переход в s, а в парном состоянии s` есть переход по противоположному символу. Тогда, очевидно, композиционные состояния $s_a$s` и s_bs` также будут нестабильны, и для каждого композиционного маршрута, начинающегося в состоянии ss`, есть маршрут с той же βγδ-трассой, начинающийся в состоянии s_as` или в состоянии $s_b$s`.

♣Теперь пусть композиционное состояние ss` либо стабильно, либо нестабильно, но только из-за переходов по реакциям a и b, то есть один из этих переходов при композиции выполняется синхронно, что даёт τ-переход. Здесь возможны варианты в зависимости от реакций a и b: реакция внешняя или внутренняя и, если она внутренняя, происходит по ней синхронизация или нет. Всего получается 2^3=9 вариантов, но из-за симметрии достаточно рассмотреть 6 из них.

♣1. Обе реакции внешние. Все композиционные состояния ss`, $s_a$s` и s_bs` стабильные. Для любого композиционного маршрута, начинающегося в состоянии ss`, есть маршрут с той же βγδ-трассой, начинающийся: 1) в состояние s_as`, если в этой βγδ-трассе первый базовый символ – это реакция

a, или 2) в состоянии $s_b s`$, если первый базовый символ – это реакция b, или 3) в каждом из этих состояний, если первый базовый символ отличен от a и b.

♣2. Реакция a внешняя, реакция b внутренняя синхронизуемая. Композиционные состояния ss` и $s_b s`$ нестабильные, а состояние $s_a s`$ стабильное. Для любого композиционного маршрута, начинающегося в состоянии ss`, есть маршрут с такой же βγδ-трассой, начинающийся в состояние $s_a s`$ или $s_b s`$.

♣3. Реакция a внешняя, реакция b внутренняя несинхронизуемая. Все три композиционных состояния стабильны. Для любого композиционного маршрута, начинающегося в состоянии ss`, есть маршрут с такой же βγδ-трассой, начинающийся в состояние $s_a s`$.

♣4. Реакции a и b внутренние синхронизуемые. Все три композиционных состояния нестабильны. Для любого композиционного маршрута, начинающегося в состоянии ss`, есть маршрут с такой же βγδ-трассой, начинающийся в состояние $s_a s`$ или $s_b s`$.

♣5. Реакции a и b внутренние, но реакция a синхронизуемая, а реакция b несинхронизуемая. Композиционные состояния ss` и $s_a s`$ нестабильные, а состояние $s_b s`$ стабильное. Для любого композиционного маршрута, начинающегося в состоянии ss`, есть маршрут с такой же βγδ-трассой, начинающийся в состояние $s_a s`$.

♣6. Реакции a и b внутренние и несинхронизуемые. Все три композиционных состояния стабильны. Для любого композиционного маршрута, начинающегося в состоянии ss`, есть маршрут с такой же βγδ-трассой, начинающийся в состояние $s_a s`$ и в состоянии $s_b s`$.

Таким образом, во всех вариантах можно не только добавить состояния-дубли $s_a s`$ и $s_b s`$, но также можно удалить исходное состояние s. Если такую процедуру выполнять, пока можно, мы получим автомат, в котором все стабильные состояния, достижимые по безопасным трассам, сингулярны.

198. Нормальные ϕ-трассы.

Будем называть *нормальной* такую ϕ-трассу автомата, которая не может быть получена из другой ϕ-трассы автомата удалением единичных ϕ-символов, то есть ϕ-символов, до и после которых в ϕ-трассе нет ϕ-символов, или является начальным отрезком такой ϕ-трассы.

♣По подмножеству нормальных ϕ-трасс автоматически восстанавливается всё множество ϕ-трасс автомата. Поэтому из равенства подмножеств нормальных ϕ-трасс двух автоматов следует равенство всех ϕ-трасс этих автоматов.

♣Таким образом, мы ставим задачу: найти алгоритм построения по спецификации такого автомата $\mathcal{A}(\mathbf{S})$, множество нормальных ϕ-трасс которого совпадало бы с множеством нормальных базовых (финальных+релевантных+сингулярных) ϕ-трасс максимальной реализации.

199. Композиционное тестирование: Алгоритм $\mathcal{A}$(S).

Теперь мы рассмотрим идею алгоритма, который по спецификации S строит автомат $\mathcal{A}$(S).

200. Состояния $\mathcal{A}$(S) – это нормальные базовые ϕ-трассы максимальной реализации.

Сначала будем считать, что состояния автомата $\mathcal{A}$(S) – это все нормальные базовые ϕ-трассы максимальной реализации. Множество таких ϕ-трасс обозначим Σ.

♣Пусть ϕ-трасса $\sigma \in \Sigma$ безопасна и не заканчивается на ϕ-множество, то есть либо пуста, либо последний её символ – это стимул или реакция. В $\mathcal{A}$(S) этой трассе поставим в соответствие *нестабильное* состояние.

♣Сингулярным ϕ-множеством будем называть такое ϕ-множество, которое содержит либо все реакции, либо все, кроме одной. Множество сингулярных ϕ-множеств обозначим как $\mathcal{R}$.

Пусть ϕ-множество R сингулярно и может продолжать ϕ-трассу σ.

Тогда из состояния автомата $\mathcal{A}$(S), соответствующего ϕ-трассе σ, проведём τ-переход в состояние σ·R. Каждое состояние вида σ·R сделаем *стабильным*.

♣Далее, пусть стимул $?x_\gamma$ разрушающий после ϕ-трассы σ или после ϕ-трассы σ·R.

По каждому разрушающему стимулу проведём переход в состояние «разрушение».

♣Наконец, пусть символ (стимул или реакция) z безопасен после ϕ-трассы σ·R или после ϕ-трассы σ.

По символам z проведём переходы в соответствующие ϕ-трассы σ·z или σ·R·z.

Из состояния σ·R мы проводим переходы по тем символам z, которые не принадлежат R.

♣А из состояния σ – по тем z, по которым мы не можем получить то же самое через переходы из стабильных состояний вида σ·R. Иными словами, ϕ-трасса σ·z должна быть нормальной: есть такое её продолжение λ, то есть имеется такая ϕ-трасса σ·z·λ, что для любого имеющегося ϕ-множества R нет ϕ-трассы σ·R·z·λ.

201. Нормальные ϕ-трассы $\mathcal{A}$(S) – это нормальные базовые ϕ-трассы максимальной реализации.

В построенном автомате нормальные ϕ-трассы – это нормальные базовые ϕ-трассы максимальной реализации.

♣Действительно, пусть ϕ-трасса σ безопасная нормальная базовая ϕ-трасса максимальной реализации. И пусть она также безопасна и нормальна в $\mathcal{A}$(**S**), а состояние σ – одно из тех состояний $\mathcal{A}$(**S**), в которых мы оказываемся после этой ϕ-трассы.

♣Тогда рассмотрим все ϕ-трассы, продолжающие ϕ-трассу σ в нашем построении: σ·R, σ·R·z, σ·z, σ·R·?x_γ и σ·?x_γ. По построению, это все базовые нормальные ϕ-трассы максимальной реализации, которые минимально продолжают ϕ-трассу σ и не заканчиваются на ϕ-множество. Все эти ϕ-трассы будут и автомате $\mathcal{A}$(**S**). Можно показать, что каждая из них нормальна в $\mathcal{A}$(**S**) и заканчивается в нужном состоянии. При этом те ϕ-трассы, которые не заканчиваются в состоянии «разрушении», являются безопасными.

202. Ограничения на спецификацию.

Определение состояний автомата $\mathcal{A}$(**S**) как нормальных базовых ϕ-трасс максимальной реализации неконструктивно. Далее наша задача – попытаться определить эти состояния конструктивно. Такими состояниями у нас будут подмножества состояний спецификации **S**. Более точно: каждому состоянию

$\mathcal{A}$(**S**) мы поставим в соответствие подмножество состояний **S**, хотя разным состояниям $\mathcal{A}$(**S**), вообще говоря, могут соответствовать одинаковые подмножества состояний **S**.

Но, прежде всего, сформулируем ограничения на спецификацию, которые позволят нам это сделать.

♣Мы будем рассматривать спецификации, в которых специфицируются стимулы, принимаемые в состоянии: как безопасные, так и разрушающие. Неспецифицированные в состоянии стимулы – это стимулы, блокируемые в этом состоянии.

Для генерации перечислимого полного тестового набора мы предъявляли к спецификации следующие требования:

1. Безопасна и безопасно-конвергентна.
2. ♣Спецификация *принимаемых* стимулов (безопасных и разрушающих).
3. ♣Регулярность по реакциям.
4. ♣Конечно-безопасно-ветвящаяся.
   - ♣Итератор TI(s) конечного числа переходов из s.
5. ♣Множество стимулов перечислимо.
   - ♣Итератор X перечислимого множества стимулов.

♣Все эти требования, кроме последнего, нужны также для построения $\mathcal{A}$(**S**).

203. Максимальная βγδ-трасса.

Пусть σ - безопасная базовая ϕ-трасса максимальной реализации

1. ♣Назовём *амбивалентным стимулом* такой стимул, который принадлежит некоторому ϕ-множеству ϕ-трассы ?x∈σ(i) и существует

βγδ-трасса, порождаемая соответствующим начальным отрезком ϕ-трассы $\pi \in \mu(\sigma[1..i])$, которая в спецификации продолжается не только блокировкой этого стимула, но и самим стимулом.

Если стимул входит в ϕ-множество ϕ-трассы, но не амбивалентен, то он блокируется в любой βγδ-трассе, порождаемой предшествующей ϕ-трассой. Такой стимул, очевидно, безопасный, но «неинтересный».

2. ♣ Для конечно-безопасно-ветвящейся спецификации число амбивалентных стимулов ϕ-трассы конечно.

3. ♣ Кроме того, существует *максимальная* βγδ-трасса, порождаемая этой ϕ-трассой, $\sigma_0 \in \pi(\sigma)$, которая содержит блокировки всех амбивалентных стимулов (и только такие блокировки) и все возможные δ (для ϕ-множеств, содержащих все реакции).

4. ♣ Максимальная βγδ-трасса σ_0 заканчивается в спецификации в конечном числе состояний, и в каждом из них заканчиваются все порождаемые βγδ-трассы.

204. Бесконечное ветвление.

Если бы спецификация не была конечно-безопасно-ветвящейся, то такой максимальной βγδ-трассы могло и не быть.

Действительно, достаточно рассмотреть случай, когда некоторая ϕ-трасса есть как в максимальной реализации, так и в спецификации, и безопасна. Допустим, эта ϕ-трасса заканчивается в спецификации в бесконечном множестве состояний.

♣ Рассмотрим пример. Пусть стимулы – это все натуральные числа,

♣ пусть задана безопасная ϕ-трасса в спецификации.

♣ и пусть после ϕ-трассы все стимулы безопасны.

♣ Рассмотрим состояния после ϕ-трассы и блокируемые в них стимулы.
В 0-ом состоянии все стимулы принимаются, в 1-ом – блокируется стимул 1, во 2-ом – стимулы 1 и 2, в 3-ем – стимулы с 1-го по 3-ий и так далее.

♣Очевидно, все стимулы амбивалентны, поскольку каждый стимул в некотором состоянии спецификации принимается, а в некотором состоянии блокируется.

♣Тогда в некоторой конформной реализации эта ϕ-трасса может заканчиваться в состоянии, где блокируются все стимулы. Это объясняется тем, что любая конечная последовательность блокировок, порождаемая таким состоянием конформной реализации, порождается некоторым состоянием спецификации.

Число амбивалентных стимулов бесконечно – это все стимулы. Поскольку любая конечная βγδ-трасса содержит только конечное число блокировок, максимальной βγδ-трассы не существует.

205. τ-замкнутое множество состояний спецификации.

τ-замыканием множества состояний будем называть надмножество состояний, в которые можно попасть из состояний исходного множества по τ-переходам.

♣Через $\mathcal{T}(\mathbf{S})$ обозначим семейство всех τ-замкнутых непустых подмножеств множества состояний спецификации **S**.

♣Множество состояний в конце любой βγδ-трассы τ-замкнуто.

♣Каждой безопасной базовой нормальной ϕ-трассе σ максимальной реализации, которая пуста или последний символ которой – это стимул или реакция (не ϕ-множество), поставим в соответствие конечное τ-замкнутое множество $U(\sigma)$ = **S** ***after*** σ_0 состояний спецификации **S**, в которых заканчивается максимальная βγδ-трасса σ_0.

206. Состояния автомата $\mathcal{A}$(S).

Итак, вернёмся к построению автомата $\mathcal{A}(\mathbf{S})$, состояния которого – это базовые нормальные ϕ-трассы максимальной реализации.

♣Прежде всего, заменим ϕ-трассу σ множеством состояний спецификации U(σ).

♣

Соответственно, ϕ-трасса σ·z заменяется множеством состояний U(σ·z), а ϕ-трасса σ·R·z заменяется множеством состояний U(σ·R·z).

♣Однако, ϕ-трасса σ·R заменяется не просто множеством состояний U(σ·R), а парой: множество состояний U(σ·R) и ϕ-множество R. Почему?

Во-первых, если для некоторой базовой нормальной ϕ-трассы σ` максимальной реализации, которая также безопасна и не заканчивается на ϕ-множество, оказалось то же самое множество состояний спецификации U(σ`)=U(σ·R), то в $\mathcal{A}(\mathbf{S})$ ϕ-трассе σ` соответствует нестабильное состояние, а ϕ-трассе σ·R соответствует стабильное состояние. Поэтому мы имеем переход U(σ·R) —τ→U(σ·R),R.

Во-вторых, могут быть два разных ϕ-множества R1≠R2, которыми может продолжаться ϕ-трасса σ, но максимальные βγδ-трассы продолженных ϕ-трасс заканчиваются в одном и том же множестве состояний спецификации U(σ·R1)=U(σ·R2).

♣Условие проведения перехода из состояния U(σ) становится следующим: сушествует состояние спецификации, в которое мы можем попасть по максимальной βγδ-трассе ϕ-трассы σ·z, но не можем попасть по максимальной βγδ-трассе ни одной из существующих ϕ-трасс вида σ·R·z.

♣Теперь мы можем определить автомат $\mathcal{A}(\mathbf{S})$, вообще не используя ϕ-трасс, но вычисляя соответствующие помножества состояний спецификации.

♣Пусть U – некоторое τ-замкнутое множество состояний спецификации, в которое мы попадаем по максимальной βγδ-трассе некоторой безопасной базовой нормальной ϕ-трассы максимальной реализации, не кончающейся на ϕ-множество.

♣ Посмотрим, какими сингулярными ϕ-множествами R может продолжаться такая ϕ-трасса.

1) Пусть в R нет стимулов и есть все реакции, кроме одной реакции. Пусть по каждому стимулу есть переход из хотя бы одного состояния множества U, и есть переход по той реакции, которой нет в R, из хотя бы одного состояния множества U. Тогда через R(U) обозначим само множество U. В противном случае будем считать, что R(U) пусто.

2) Пусть R содержит хотя бы один стимул (блокировка) или содержит все реакции (стационарность). Пусть в U есть подмножество W, которое удовлетворяет следующим требованиям по стимулам и реакциям. Каждый стимул, который принадлежит R, блокируется в каждом состоянии W, а каждый стимул, который не принадлежит R, принимается хотя бы в одном состоянии W. Если R содержит все реакции, то все состояния W стационарны. Если же R не содержит одну реакцию, то эту реакцию выдаётся хотя бы в одном состоянии W. Тогда через R(U) мы обозначим наибольшее такое подмножество W. Если существует хотя бы одно W, то существует и наибольшее, потому что объединение все подмножеств W, удовлетворяющим нашим требованиям по стимулам и реакциям, также удовлетворяет этим требованиям. Если не существует ни одного такого W, то будем считать, что R(U) пусто.

♣ Через z(U) обозначим τ-замыкание непустого множества состояний, в которые можно попасть из состояний U с помощью переходов по символу z.

Теперь построение автомата $\mathcal{A}(\mathbf{S})$ выглядит так.

207. Формальные правила вывода.

Теперь мы можем записать формальные правила вывода автомата $\mathcal{A}(\mathbf{S})$ из спецификации **S**.

Пусть задана спецификация $\mathbf{S} = AA(V,C_\gamma,E,s_0)$.

♣ Пусть $U \in \mathcal{T}(\mathbf{S})$ – τ-замкнутое множество состояний спецификации.

Обозначим через $\mathbf{X}_\gamma(U)$ – стимулы, разрушающие в состояниях U,

♣ а через $U_{stab}(U)$ – подмножество всех стабильных состояний множества U.

♣ Определим $\mathcal{A}(\mathbf{S}) = AA(V`,C_\gamma,E`,s_0`)$.

$V` = \mathcal{T}(\mathbf{S}) \cup \mathcal{T}(\mathbf{S})\times\mathcal{R} \cup \{\Gamma\}$.
Состояние – это либо τ-замкнутое множество состояний спецификации,
либо пара из такого множества и сингулярного ϕ-множества,
либо состояние «разрушение», которое мы обозначаем символом Γ.

$s_0` = \mathbf{S}$ ***after*** ϵ – начальное состояние

♣ Множество переходов – E` – это наименьшее множество, порождаемое правилами вывода:
$\forall U\in\mathcal{T}(\mathbf{S})\ \ \forall z\in C\ \ \forall R\in\mathcal{R}$

♣ $z\notin \mathbf{X}_\gamma(U)\ \&\ z(U)\backslash z(U_{stab}(U))\neq\varnothing \ \vdash\ U\text{—}z\rightarrow z(U)\backslash z(U_{stab}(U))$

♣ $?x\in \mathbf{X}_\gamma(U) \ \vdash\ U\text{—}?x\rightarrow\Gamma$

♣ $R(U)\neq\varnothing \ \vdash\ U\text{—}\tau\rightarrow(R(U),R)$

♣ $z\notin \mathbf{X}_\gamma(U)\ \&\ z\notin R\ \&\ z(U)\neq\varnothing \ \vdash\ (U,R)\text{—}z\rightarrow z(U)$

♣ $x\in \mathbf{X}_\gamma(U) \ \vdash\ (U,R)\text{—}x\rightarrow\Gamma$

♣ $\vdash\ \Gamma\text{—}\gamma\rightarrow\Gamma\text{—}\tau\rightarrow\Gamma$

□

208. Спецификация безопасных стимулов.

Мы рассмотрели алгоритм для автоматов, в которых специфицированы принимаемые стимулы (безопасные и разрушающие), и неспецифицированы блокируемые стимулы.

♣Нужна модификация алгоритма для автоматов, в которых специфицированы безопасные стимулы (принимаемые и блокируемые), и неспецифицированы разрушающие стимулы.

209. Композиционное тестирование: *Косая* композиция.

210. Цели построения *косой* композиции.

Напомним, что построение косой композиции преследует три цели.

1. Проверка компонуемости, то есть проверка того, что косая композиция спецификаций компонентов безопасна и безопасно-конвергентна. Это гарантирует, что композиция любых конформных реализаций будет безопасно-тестируемой для любой корректной спецификации системы.

2. Проверка корректности системной спецификации, то есть проверка того, что косая композиция спецификаций компонентов конформна системной спецификации.

3. Генерация системных тестов по косой композиции спецификаций компонентов. Это полезно, когда у нас нет корректной системной спецификации, или эта спецификация недостаточно сильная для проверки интересующих нас свойств системы.

211. Ограничение на спецификации компонентов.

Выше мы уже сформулировали требования к спецификации **S** для того, чтобы можно было построить преобразованную специкацию $\mathcal{A}(\mathbf{S})$.

♣Обратим внимание, что сама преобразованная спецификация $\mathcal{A}(\mathbf{S})$, по построению, гамма-нормальна.

Как строится композиция двух таких преобразованных спецификаций? Для каждой пары состояний, которая у нас может получиться в процессе построения, то есть достижимая из пары начальных состояний, мы должны построить все композиционные переходы. Что для этого нужно?

Во-первых, мы должны уметь перечислять переходы в каждом из компонентов. Это обеспечивается алгоритмом TI.

Во-вторых, для каждого символа – стимула или реакции, которым помечен переход в одном компоненте, мы должны определить: внешний этот символ или внутренний? То есть, принадлежит ли он подчёркнутому алфавиту другого компонента или нет.

♣Это эквивалентно разрешимости алфавита. Соответственно, нам нужен алгоритм, вычисляющий предикат принадлежности символа алфавиту.

212. Итеративное построение.

Прежде всего, заметим, что нас никогда не интересуют переходы в композиционных гамма-состояний, кроме, естественно, самих гамма-переходов. Поэтому мы при построении косой композиции мы не будет строить переходы из композиционных гамма-состояний, кроме, естественно, гамма-перехода.

♣При этих ограничениях на спецификации компонентов мы можем для любого, наперёд заданного, числа n построить подавтомат композиции, содержащий все композиционные *маршруты* длиной не более n.

♣Для каждой из объявленных трёх целей нам нужно большее: за конечное время построить подавтомат косой композиции, содержащий все *трассы* длиной не более n?

Для этого мы должны проверять все цепочки τ-переходов, которые у нас возникают.

При композиции τ-переход появляется либо асинхронно, наследуя τ-переход в одном из компонентов, либо синхронно, как результат соединения перехода по выдаче внутреннего символа в одном компоненте с приёмом этого символа в другом компоненте. Понятно, что, если у нас возникает бесконечная цепочка τ-переходов, проходящая через бесконечное число композиционных состояний, то мы эту задачу решить не сможем.

♣Поэтому выдвигается следующее *требование внутренней регулярности компонентов*: для состояния s, достижимого минуя гамма-состояния, должно быть конечно множество состояний, достижимых из s по машрутам без переходов по внешним стимулам и реакциям, которые не проходят, но могут заканчиваться в гамма-состояниях.

213. Безопасность и безопасно-конвергентность.

Для проверки безопасности композиции мы должны убедиться, что из пары *начальных* состояний разрушение не достижимо по τ-переходам и переходам по реакциям.

Для проверки безопасно-конвергентности нам нужно просматривать такие цепочки τ-переходов и переходов по реакциям из *каждого* композиционного состояния, достижимого по безопасной композиционной βγδ-трассе + стимул.

Ясно, что такие цепочки переходов должны проходить через конечное множество композиционных состояний. Для этого мы должны выдвинуть следующее требование внешней регулярности компонентов: для состояния s, достижимого по безопасной трассе [+ внешний стимул], должно быть конечно множество состояний, достижимых из s по машрутам без переходов по внешним стимулам (но, возможно, по внешним реакциям), которые не проходят, но могут заканчиваться в гамма-состояниях.

♣Этого достаточно для проверки безопасности косой композиции за конечное время.

♣Безопасно-конвергентность косой композиции проверяется *итеративно*: для всех безопасных трасс длиной не более заранее заданного числа n.

♣Этого достаточно для полной проверки безопасно-конвергентности косой композиции конечных спецификаций.

♣Иначе безопасно-конвергентность косой композиции обеспечивается условием внешней ограниченности компонентов: для состояния s, достижимого по безопасной трассе [+ внешний стимул], должны быть конечны все начинающиеся в s трассы без внешних стимулов.

214. Корректность системной спецификации.

Аналитическая проверка корректности системной спецификации выполняется *итеративно*: для всех безопасных трасс длиной не более заранее заданного числа n.

♣Этого достаточно в двух случаях. 1) Если системная спецификация имеет конечное поведение, то есть конечное число трасс. 2) Если и системная спецификация и косая композиция обе конечны, то есть содержат конечное число достижимых состояний и переходов.

♣В противном случае полная аналитическая проверка требует бесконечного числа конечных проверок (для трасс ограниченной длины) аналогично бесконечному полному тестовому набору.

215. Генерация системных тестов по косой композиции.

Условие внешней регулярности компонентов гарантирует регулярность по реакциям косой композиции.

♣Если в каждом компоненте задан итератор перечислимого множества стимулов и алгоритм разрешения алфавита, то можно построить итератор перечислимого множества внешних стимулов.

♣Если косая композиция безопасна и безопасно-конвергентна, то генерация полного набора системных тестов делается итеративно в процессе построения косой композиции.

216. ПРИОРИТЕТЫ

217. Для чего нужны приоритеты?

В предыдущем рассказе мы несколько раз сталкивались с необходимостью введения приоритетов между переходами, определёнными в одном состоянии.

1. В реализации приоритет стимула над внутренней активностью для выхода из цикла внутренней дивергенции.
2. В реализации приоритет стимула над реакцией и внутренней активностью для выхода из цикла осцилляции.
3. В реализации приоритет стимула над реакцией и внутренней активностью для приёма стимула в нестационарном состоянии.
4. В среде приоритет выдачи реакции над приёмом реакции для выхода из цикла дивергенции композиции среды и реализации.

218. Факультативное и императивное поведение.

До сих пор мы считали, что все переходы, определённые в состоянии, равноприоритетны. Если в данный момент времени могут выполняться несколько переходов, то выбирается для выполнения один из них недетерминированным образом.

Такое поведение будем называть *факультативным*.

Императивным поведением назовём такое поведение, когда приём стимулов имеет приоритет перед выдачей реакций и внутренней активностью (τ-переходами). Если в данный момент времени могут выполняться один или несколько несколько переходов по приёму стимулов, то выбирается для выполнения один из них, независимо от наличия готовых к выполнению переходов по выдаче реакций или (всегда готовых к выполнению) τ-переходов. Если могут приниматься несколько стимулов, то выбор стимула для приёма выполняется недетерминированным образом.

♣Теперь можно говорить о факультативных автоматах, которые во всех состояниях ведут себя факультативно. Именно такие автоматы мы рассматривали до сих пор.

Императивные автоматы – это автоматы, которые во всех своих состояниях ведут себя императивно.

♣Наконец, можно говорить о факультативных или императивных состояниях в автоматах смешанного типа. В этом случае при спецификации автомата нужно указывать, какие состояния факультативные, а какие – императивные.

219. Императивность и θ-переход.

Рассмотрим асинхронное тестирование со средой, которая не тормозит реализацию по реакциям. То есть среда всегда готова принять от реализации любую реакцию.

♣В этом случае императивное поведение можно моделировать с помощью θ-перехода.

♣Для этого делаем копию состояния и проводим θ-переход из оригинала в копию. Из состояния-оригинала переходы по реакциям и τ-переходы переносятся в состояние-копию.

Сначала в состоянии-оригинале автомат пытается принять стимулы. Если подходящих стимулов нет, срабатывает θ-переход и автомат выдаёт реакции или совершает τ-переход.

♣Для того, чтобы эта схема работала, нужно, чтобы тайм-аут θ-перехода в реализации был заведомо меньше тайм-аута θ-переход в тесте. Тогда тест «не замечает» θ-переходов в реализации. Эти θ-переходы только реализуют приоритеты, но не «обманывают» тест, который по-прежнему понимает истечение тайм-аута в принимающем состоянии как стационарность, а в посылающем – как блокировку посылаемого стимула.

220. Параллельная композиция с θ-переходом в реализации.

В определении оператора параллельной композиции добавляется ещё одно правило для θ-перехода в реализации.

221. Приоритеты стимулов.

Аналогичный приём можно использовать для моделирования приоритетов между стимулами, которые определены в состоянии. Мы предполагаем, что имеется линейная последовательность приоритетов, и каждому стимулу назначен один из приоритетов.

♣Тогда автомат сначала пытается принять стимулы с наивысшим приоритетом. Если ни один из них не может быть принят, выполняется θ-переход в состояние, где автомат пытается принять стимулы следующего приоритета. И так далее.

222. ПРОБЛЕМЫ: θ-переход в общем случае.

В общем случае θ-переход в реализации создаёт проблемы.

♣Во-первых, при асинхронном тестировании, когда среда может тормозить реализацию по реакциям.

♣Во-вторых при композиционном тестировании. Здесь возникает вопрос о соотношении тайм-аутов θ-переходов взаимодействующих компонентов.

223. ε-переход.

ε-переходом назовём переход по отсутствию стимулов. Его можно понимать как переход по пустому стимулу, где пустой стимул обозначается символом ε и понимается именно как отсутствие стимулов, передаваемых в автомат в данном состоянии в текущий момент времени.

Мы можем рассматривать в автомате трассы, обогащённые символом ε.

♣В общем, для всюду определённых по стимулам автоматов ε-переход эквивалентен θ-переходу. Однако в общем случае они имеют разную семантику и не сводятся друг к другу.

224. Переход по предикату.

Для автомата в алфавите A рассматриваем предикаты на переходах, имеющие общий вид:

$P : \wp(A \cup \{\tau\}) \to \mathbf{Bool}$.

Это можно рассматривать как булевскую функцию от булевских переменных, индекс которых пробегает множество A.

Расширение предиката надмножеством.
Пусть задано надмножество $E \supseteq A$.
Определим расширение предиката надмножеством E:
$P[E]: \wp((E \cup \{\tau\})) \to \textbf{\textit{Bool}}$.
$U \subseteq E \cup \{\tau\}$: $P[E](U) =_{def} P(U \cap (A \cup \{\tau\}))$.

Сужение предиката подмножествами.
Пусть заданы два непересекающихся подмножества $T, F \subseteq A$, $T \cap F = \varnothing$.
Будем считать, что подмножество T задаёт индексы переменных, значение которых фиксируется как ***true***, а подмножество F задаёт индексы переменных, значение которых фиксируется как ***false***.
Определим сужение предиката подмножествами T и F:
$P\langle T,F\rangle: \wp((A \cup \{\tau\}) \setminus (T \cup F)) \to \textbf{\textit{Bool}}$.
$U \subseteq (A \cup \{\tau\}) \setminus (T \cup F)$: $P\langle T,F\rangle(U) =_{def} P(U \cup T)$.

225. Переход по предикату (продолжение).

Задание предикатов на переходах.
Будем считать, что предикат P, заданный на переходе из состояния s, сужен подмножествами $P\langle T,F\rangle = P$, где

$T = \{\tau \mid s \text{—}\tau\!\!\rightarrow\}$,

$F = \{z \in A \mid s \text{—}z\!\!\nrightarrow\}$.

Сужение-расширение предиката при композиции.
Рассмотрим композицию автомата **A** в алфавите A и автомата **B** в алфавите B. Пусть ab – достижимое композиционное состояние. Для каждого перехода из состояния a его предикат P сузим подмножествами T и F и расширим надмножеством E:

$T(a,b) = \{ z \in A \cap \underline{B} \mid a \text{—}z\!\!\rightarrow \;\&\; b \text{—}\underline{z}\!\!\rightarrow \}$,

$F(a,b) = \{ z \in A \cap \underline{B} \mid a \text{—}z\!\!\rightarrow \;\&\; b \text{—}\underline{z}\!\!\nrightarrow \}$,

$E(a,b) = A \setminus \underline{B} \cup B \setminus \underline{A}$,
$P(a,b) = P\langle T(a,b), F(a,b)\rangle[E(a,b)]$.

226. Переход по предикату (продолжение).

Вычисление предикатов композиционных переходов.
Предикат R асинхронного перехода композиционного автомата, порождённого переходом автомата **A** с предикатом P, вычисляется так:
R = P(a,b).
Предикат R асинхронного перехода композиционного автомата, порождённого переходом автомата **B** с предикатом Q, вычисляется так:
R = Q(b,a).
Предикат R синхронного перехода композиционного автомата, порождённого переходом автомата **A** с предикатом P и переходом автомата **B** с предикатом Q, вычисляется так:
R = P(a,b) & Q(b,a).

Оператор композиции.

$z \in A_{\gamma\tau} \setminus \underline{B}$ & a—z,P→a` ⊢ ab—z,P(a,b)→a`b;

$z \in B_{\gamma\tau} \setminus \underline{A}$ & b—z,Q→b` ⊢ ab—z,Q(b,a)→ab`;

$z \in A \cap \underline{B}$ & a—z,P→a` & b—$\underline{z}$,Q→b` ⊢
ab—τ,P(a,b)&Q(b,a)→a`b`;

227. Переход по предикату (продолжение).

Выполнение автомата.
Для выполнения автомата **A** нужно скомпоновать его с автоматом **B**, в котором нет переходов, а алфавит $B=\underline{A}$.
В такой композиции будут только τ-переходы, а все предикаты – константы (***true*** или ***false***). Удаляем переходы с ложными предикатами.

На самом деле, константные предикаты могут возникать и раньше. Мы можем сразу удалять переходы с константными ложными предикатами.

228. Примеры предикатов на переходах.

Приоритет стимулов над реакциями и τ.
Предикат на переходе по стимулу: P(U) = ***true***.
Предикат на переходе по реакции или τ: P(U) = (?U=∅),
где ?U = {?x∈U}.

Приоритеты символов 1...n.
$A = A_1 \cup ... \cup A_n$ – разбиение на непересекающиеся подмножества.
Предикат на переходе по символу из A_i: $P(U)=(U \cap (A_1 \cup ... \cup A_{i-1})=\varnothing)$.

229. Примеры предикатов на переходах (окончание).

Правила в состоянии s следующие:

$s \text{—} ?s, P_1 \rightarrow s_1$: $P_1(U) = ?b \in U$

$s \text{—} ?s, P_2 \rightarrow s_2$: $P_2(U) = ?b \notin U$

$s \text{—} ?b, P_3 \rightarrow s_3$: $P_3(U) = ?a \notin U$

$s \text{—} !y, P_4 \rightarrow s_4$: $P_4(U) = ?a \notin U \ \& \ ?b \notin U$

Символы Север, Восток, Юг, Запад. Нет τ-переходов. Выбираем любое направление из возможных, но только не среднее, если возможны три направления. Предикат перехода определяется его символом:

$P_{Север}(U) = (U \neq \{ Запад, Север, Восток \})$

$P_{Восток}(U) = (U \neq \{ Север, Восток, Юг \})$

$P_{Юг}(U) = (U \neq \{ Восток, Юг, Запад \})$

$P_{Запад}(U) = (U \neq \{ Юг, Запад, Север \})$

230. Тестовые возможности. Машина тестирования.

231. Пример: приоритет стимулов над реакциями.

232. Трассы наблюдений.

233. Гипотеза о безопасности и конвергентности.

234. Обобщённое соответствие *ioco*.

235. Спецификация.

236. Генерация тестов.

237. Асинхронное и композиционное тестирование.

238. ЧТО ОСТАЛОСЬ ЗА СКОБКАМИ?.

1. Метрики, покрытия и связанная с ними структура спецификаций.
2. Другие спецификации
   (алгебраические).
3. Другие модели
   (геометрические автоматы).
4. Время (временны*е* автоматы).
5. Вероятности. Вероятностные алгоритмы и вероятностные автоматы.

239. КОНЕЦ