И. Бурдонов, А. Косачев
Формализация тестового эксперимента - II
Томск
27 слайдов

Слайд 1. Титул.

ФОРМАЛИЗАЦИЯ ТЕСТОВОГО ЭКСПЕРИМЕНТА – II

Слайд 2. Основные понятия.

Прежде всего, определим основные понятия, которые мы будем использовать.

Отношение «конформности» – это отношение «правильности», которое говорит, что реализация удовлетворяет требованиям спецификации.

Тестирование – это проверка конформности в процессе эксперимента. Тест, подменяя собой окружение реализации, взаимодействует с реализацией.

В общем случае в конце, то есть после всех прогонов всех тестов из набора тестов, выносится вердикт ***pass*** – «проходит», или вердикт ***fail*** – ошибка.

Набор тестов *значимый*, если каждая конформная реализация его проходит, *исчерпывающий*, если каждая неконформная реализация его не проходит (обнаруживается ошибка), и *полный*, если он значимый и исчерпывающий, то есть реализация проходит полный набор тестов тогда и только тогда, когда она конформна.

Слайд 3. Основные понятия.

Я хочу рассказать о новой модели тестового взаимодействия и соответствующей модели реализации, модели спецификации и отношении конформности между реализациями и спецификациями.

Слайд 4. Три предположения: Первое предположение.

Мы будем рассматривать не любое тестирование, а только такое, которое основано на трёх предположениях.

Первое предположение. Взаимодействие теста и реализации будет пониматься как дискретный процесс, элементами которого могут быть тестовые воздействия на реализацию, наблюдения поведения реализации и наблюдения её состояний.

Поведение – это то, что реализация делает или, наоборот, не делает. В общем случае не всякое поведение реализации может наблюдаться в тестовом эксперименте, то есть в реализации могут происходить события, которые не наблюдаемы. Поскольку они не наблюдаемы, они не различимы между собой извне реализации и не входят в трассу. Такое ненаблюдаемое поведение реализации традиционно обозначается символом τ и называется τ-активностью.

Поведение реализации зависит от её состояния: в разных состояниях поведение реализации может быть различно при одних и тех же тестовых воздействиях или отсутствии таковых.

Если наблюдать можно только поведение реализации, а её состояния нам не видны, то взаимодействие сводится к последовательности тестовых воздействий и наблюдений поведения реализации. Эту последовательность мы будем называть *трассой*, а конформности, основанные только на трассах, – трассовыми конформностями.

Если кроме поведения реализации можно наблюдать её текущие состояния, то возможны конформности, основанные на соответствии между состояниями реализации и спецификации. Это, так называемые, отношения симуляции.

Пока что мы будем рассматривать только трассовые конформности.

Дискретное взаимодействие моделируется с помощью, так называемой, машины тестирования, внутри которой находится реализация. Тестовому воздействию соответствует нажатие той или иной кнопки на клавиатуре машины, а наблюдению – появление его символа на экране дисплея. Таким образом, трасса – это последовательность кнопок и наблюдений.

Тест понимается как инструкция оператору машины, в которой указывается, что оператор должен делать после той или иной трассы: нажимать кнопку (и какую кнопку) и/или ждать наблюдений.

Всё, что мы можем узнать о реализации с помощью такого тестирования, – это множество её трасс. Наблюдение в эксперименте некоторой трассы предполагает, что все её префиксы наблюдались в этом же эксперименте в более ранние моменты времени. Поэтому множество трасс реализации префикс-замкнуто.

Тест (как инструкция оператору) также задаётся непустым префикс-замкнутым множеством трасс. Тестовый эксперимент – это прогон теста, который заканчивается, когда получается максимальная в тесте трасса или «ответвление в сторону», то есть после некоторой трассы теста получается наблюдение, которым эта трасса не продолжается в тесте. Результатом прогона теста является трасса реализации. Различные прогоны одного и того же теста могут давать разные результаты, если реализация и/или тест недетерминированы.

Тест недетерминирован, если после какой-то трассы теста недетерминировано поведение оператора машины тестирования: он может, как ждать наблюдения, так и нажимать кнопки, или он может только нажимать кнопки, но таких кнопок несколько. Иными словами, тест недетерминирован, если некоторая его немаксимальная трасса продолжается в тесте и наблюдениями и кнопками, или несколькими кнопками. Недетерминированный тест эквивалентен набору детерминированных тестов в том смысле, что они дают возможность наблюдения одного и того же множества трасс реализации. Поэтому, как правило, рассматривают только детерминированные тесты.

Что касается реализации, то предполагается, что её недетерминизм – это результат абстрагирования от некоторых неучитываемых внешних факторов – погодных условий, которые определяют выбор того или иного поведения детерминировано. *Гипотеза о глобальном тестировании* предполагает, что любые погодные условия могут быть воспроизведены в тестовом эксперименте. Для этого даже детерминированный тест должен

прогоняться несколько раз, чтобы наблюдать все возможные для этого теста трассы реализации.

Тестирование состоит из отдельных тестовых экспериментов. В каждом таком эксперименте выполняется прогон какого-то теста из заданного набора тестов при некоторых погодных условиях. Результатом тестирования является множество трасс, наблюдаемых во всех этих тестовых экспериментах. Выносится вердикт ***pass*** (проходит) или ***fail*** (ошибка).

Спецификацию можно понимать как описание того, какие множества трасс реализаций правильные (конформные), а какие нет. В общем случае мы должны считать, что конформна или неконформна не каждая отдельная трасса, а всё множество трасс реализации целиком. Дальше мы рассмотрим второе предположение, которое упростит эту ситуацию.

Заметим, что результат тестирования – множество трасс X – это не обязательно множество всех трасс реализации. Если спецификация утверждает, что любая реализация, множество трасс которой включает X неконформна, то значимый набор тестов может (хотя и не обязан), а полный набор тестов должен выносить вердикт ***fail*** при наблюдении множества трасс X или любого его надмножества. Если спецификация утверждает, что любая реализация, множество трасс которой включает X, наоборот, конформна, то исчерпывающий (и полный) набор тестов может (хотя и не обязан), а полный набор тестов должен выносить вердикт ***pass*** при наблюдении множества трасс X или любого его надмножества.

Слайд 5. Три предположения: Второе предположение.

Пока что мы ограничимся только теми конформностями, которые отвечают *принципу независимости трасс*: любая трасса реализации, вообще говоря, конформна или неконформна независимо от других её трасс. Конформности такого типа называются *редукциями*.

Не является редукцией, например, конформность, которая разрешает реализации иметь любую из двух указанных трасс, но не

обе одновременно. Принцип независимости исключает из рассмотрения конформности типа симуляций, которые основаны на соответствии состояний реализации и спецификации, а также конформности, которые требуют обязательного наличия в реализации тех или иных наблюдений после тех или иных трасс.

Для редукции можно считать, что спецификация (прямо или косвенно) определяет множество *разрешаемых* трасс. Конформность означает вложенность множества трасс реализации во множество разрешаемых трасс. Такая конформность является частичным (нестрогим) порядком (рефлексивное, симметричное и транзитивное отношение). Неразрешаемая спецификацией трасса называется *ошибкой*. То есть множество ошибок – это дополнение множества разрешаемых трасс до множества всех трасс в алфавите кнопок и наблюдений.

При тестировании редукции обнаружение любой ошибки означает, что реализация неконформна. Поэтому значимый набор тестов (тест) может (хотя и не обязан) выносить вердикт ***fail*** сразу, как только наблюдается ошибка. Набор тестов исчерпывающий, если для каждой неконформной реализации хотя бы одна имеющаяся в ней ошибка может быть обнаружена некоторым тестом из набора, то есть является трассой этого теста. Это означает, что неконформность реализации обнаруживается всегда за конечное время, тогда как вывод о конформности реализации может быть сделан, вообще говоря, только после всех прогонов при всех возможных погодных условиях всех тестов полного набора (число таких прогонов может быть бесконечно).

Слайд 6. Три предположения: Третье предположение.

Третье предположение. На практике, естественно, используются только конечные тесты, точнее, тесты, которые заканчиваются за конечное время. При дискретном взаимодействии это означает, что при задании спецификации и генерации тестов используются только конечные трассы.

Заметим, что для спецификации, основанной на конечных трассах, бесконечные тестовые эксперименты ничего не добавляют.

Однако в общем случае это не так. Например, рассмотрим две реализации, в которых возможны только два наблюдения: *x* и *y*. В одной реализации есть бесконечная цепочка *x*. В другой реализации такой бесконечной цепочки нет, но есть бесконечный «веер» конечных цепочек *x*. В обеих реализациях нет перехода по *y*. При конечных тестовых экспериментах эти две реализации неразличимы. Для спецификации, в которой наблюдение *y* считается ошибкой после любого числа *x*, эти реализации обе конформны. В то же время бесконечный тестовый эксперимент позволяет эти реализации различить: в первой реализации есть бесконечная трасса *x*, а во второй нет. Если допускаются бесконечные трассы, то спецификация может трактовать бесконечную цепочку *x* как ошибку. Разумеется, такая ошибка не может быть найдена за конечное время.

Слайд 7. Три проблемы.

Для чего понадобилась эта новая модель? Занимаясь долгое время теорией тестирования, мы поняли, что существуют три важные проблемы: проблема оптимизации тестов, проблема композиции и проблема приоритетов.

Слайд 8. Проблема оптимизации тестов.

Что такое оптимизация тестов? Вот пусть у нас есть полный набор тестов. Возникает вопрос: а нельзя ли его уменьшить, удалив некоторые тесты так, чтобы набор тестов оставался полным? Тест предназначен для ловли ошибок. Тест ловит ошибку, если она может проявиться при взаимодействии с данным тестом и после этого тест выносит вердикт ***fail***. Каждый тест ловит некоторое множество ошибок. Понятно, что если каждую ошибку, которую ловит данный тест, ловит также какой-нибудь другой тест из набора, то данный тест можно удалить из набора. Но этой очевидной оптимизацией дело не заканчивается.

Во-первых, кроме ошибок, определяемых спецификацией, могут быть другие трассы, которые формально разрешены спецификацией, но, тем не менее, не встречаются в конформных

реализациях. Такие трассы будем называть неконформными. После этого под ошибкой мы будем понимать любую неконформную трассу. Ошибки, определяемые спецификацией, будем называть ошибками 1-го рода. Ошибка 2-го рода – это неконформная трасса, не являющаяся ошибкой 1-го рода, то есть формально разрешённая спецификацией.

Почему могут существовать ошибки 2-го рода? Всё дело в зависимости между ошибками. По определению, любая реализация из класса тестируемых реализаций, содержащая ошибку 2-го рода, неконформна, а это означает, что она содержит какую-то ошибку 1-го рода.

В общем случае можно говорить о множественной зависимости между ошибками. Будем говорить, что из множества ошибок A следует множество ошибок B, если любая реализация из класса тестируемых реализаций, содержащая какую-то ошибку из A, содержит также какую-то ошибку из B. Тогда, если A – это множество всех ошибок, а B его строгое подмножество, то в наборе тестов достаточно оставить только те тесты, которые ловят ошибки из B. Если, в свою очередь, из B следует его строгое подмножество C, то оптимизацию можно продолжить. В общем случае возникает задача поиска минимальных по этому отношению множеств ошибок. И ещё возникает вопрос, а есть ли среди таких минимальных множеств наименьшее? Особенно это важно в том случае, когда множество всех ошибок бесконечно, но существует следующее из него конечное множество ошибок. Тогда вместо бесконечного набора тестов мы можем использовать конечный набор. Эта проблема зависимости между ошибками очень сложна, и в общем виде не решена.

1►Здесь важно отметить, что неконформность трасс и вообще зависимость между ошибками существенно зависят от класса тестируемых реализаций. Может оказаться, что в классе всех возможных реализаций существует реализация, которая содержит ошибку из класса A, но не содержит ошибок из класса B. Это значит, что на классе всех реализаций из множества ошибок A не следует множество ошибок B. Однако, если любая такая реализация не принадлежит рассматриваемому подклассу тестируемых реализаций, то на этом подклассе зависимость между

ошибками имеется: из множества ошибок A следует множество ошибок B.

Слайд 9. Проблема композиции.

Теперь рассмотрим проблему композиции. Для описания сложных систем, состоящих из нескольких компонентов, используется оператор композиции. Составная система получается из своих компонентов с помощью этого оператора.

Пусть у нас есть спецификация системы и спецификации всех её компонентов. Допустим, реализация каждого компонента конформна спецификации этого компонента. Возникает вопрос: будет ли композиция реализаций компонентов конформна спецификации системы? Если на этот вопрос можно ответить положительно для любых конформных реализаций компонентов, то такую спецификацию системы будем называть *корректной*.

Понятно, что если спецификация системы вообще никак не связана со спецификациями компонентов, то трудно ожидать, что она будет корректной. По сути, это вопрос о согласованности спецификации системы со спецификациями компонентов. Эта проблема носит название «верификация декомпозиции системных требований». При проектировании сверху вниз сначала создаётся спецификация системы, а потом она раскладывается в спецификации компонентов. Верификация того, правильно или нет делается такая «раскладка», как раз и есть верификация декомпозиции системных требований.

Решение этой проблемы заключается в том, чтобы по спецификациям компонентов построить такую корректную спецификацию системы, которая предъявляла бы к системе наибольшие требования среди всех корректных спецификаций системы. Что значит наибольшие требования? Это значит, что она определяет наибольшее множество ошибок. Мы называем такую спецификацию системы *косой* композицией спецификаций компонентов. Почему «косой»? Это чтобы отличить такую композицию от композиции реализационных моделей, которую можно назвать *прямой* композицией.

Заданная спецификация системы корректна, если она определяет только такие ошибки, которые являются ошибками с точки зрения косой композиции спецификаций компонентов. Иными словами, множество ошибок, определяемое спецификацией системы, должно быть вложено в множество ошибок, определяемое косой композицией.

Построение косой композиции спецификаций является довольно сложной проблемой. В том случае, когда реализационная и спецификационная модель одного и того же типа, можно рассматривать и прямую композицию спецификаций, то есть компоновать спецификации компонентов точно так же, как компонуются реализации этих компонентов. Тогда проблема композиции звучит так: будет ли композиция реализаций компонентов, конформных спецификациям этих компонентов, конформна прямой композиции спецификаций компонентов? Если будет, то это называется сохранением конформности при композиции или монотонностью конформности относительно композиции.

К сожалению, в общем случае конформность не сохраняется при композиции. Для решения этой проблемы предлагаются эквивалентные преобразования спецификаций. Эквивалентность означает, что преобразованная спецификация, во-первых, применима для того класса тестируемых реализаций, который рассматривался для исходной спецификации, и, во-вторых, на этом классе она определяет тот же самый подкласс конформных реализаций. Эти преобразования подбираются так, чтобы для преобразованных спецификаций всё уже было хорошо: композиция конформных реализаций компонентов конформна прямой композиции преобразованных спецификаций компонентов. Такая прямая композиция преобразованных спецификаций оказывается их косой композицией.

1►И опять, как и в случае оптимизации тестов, построение косой композиции и монотонность конформности зависит от класса тестируемых реализаций. Пусть на реализации компонентов не налагаются никакие ограничения, кроме их конформности спецификациям компонентов, и при этих условиях спецификация системы *C* корректна, в частности, является косой композицией. Тогда она, очевидно, остаётся корректной и в том случае, когда

конформные реализации компонентов не любые, а выбираются из каких-то подклассов, заранее заданных для этих компонентов. В то же время спецификация системы *D* может быть корректна только для заданных подклассов реализаций компонентов: спецификации *D* конформна композиция только таких конформных реализаций компонентов, которые принадлежат соответствующим подклассам. В частности, спецификация *D* может быть косой композицией для заданных подклассов реализаций компонентов. Такая спецификация *D* может оказаться даже некорректной на классах всех реализаций, но зато на заданных подклассах будет предъявлять к системе большие требования по сравнению со спецификацией *C*. Если *C* и *D* – это косые композиции, то *C*, хотя и остаётся корректной на подклассах реализаций, но уже не будет косой композицией на этих подклассах, поскольку *D* предъявляет к системе больше требований.

Слайд 10. Проблема приоритетов.

Теперь рассмотрим проблему приоритетов. Что это такое? Суть проблемы в недетерминизме реализации. Если при взаимодействии с тестом у реализации есть несколько вариантов поведения, то возникает вопрос: как выбирается тот или другой вариант? В существующих моделях реализации и взаимодействия предполагается, что этот выбор полностью недетерминирован: реализация может выбрать любой из этих вариантов. В то же время в некоторых реальных системах существуют приоритеты, требующие выбор одних вариантов и запрещающих другие варианты.

Я поясню смысл приоритетов на примере выхода из дивергенции. Внутренняя активность, называемая также τ-активностью, – это выполнение системой каких-то действий, которые не наблюдаются и никак не проявляются во внешнем взаимодействии. В существующих моделях τ-активность, если она имеется в реализации, считается равноприоритетной с внешним взаимодействием.

1►Поэтому, если в системе есть τ-активность, и мы оказываем на неё внешнее воздействие *x*, то может получиться, что реализация

будет выполнять τ-активность и обратит внимание на внешнее воздействие *x* только тогда, когда τ-активность закончится.

2►Дивергенция – это бесконечная внутренняя активность системы. Если в системе возникла дивергенция, то система может бесконечно долго выполнять свои внутренние действия, не обращая внимания на внешнее воздействие.

В то же время в большинстве реальных систем внешнее воздействие гарантированно приводит к выходу из дивергенции. Современный компьютер почти никогда не стоит, он всё время что-то делает: что-то опрашивает по интернету, получает обновления, ведёт статистику, собирает мусор и тому подобное. И это может продолжаться бесконечно долго. Но только до тех пор, пока мы не тронули мышку или не нажали клавишу клавиатуры. Вот тогда компьютер через короткое время реагирует на наше внешнее воздействие, прерывая свои внутренние действия. Это не значит, что он не будет больше выполнять эти внутренние действия. Это означает лишь, что компьютер воспринял наше внешнее воздействие и готов взаимодействовать с нами, быть может, продолжая выполнять внутренние действия. Это означает, что внешнее воздействие имеет больший приоритет, чем дивергенция. Заметим, что внешнее воздействие вовсе не обязано немедленно прерывать τ-активность: какое-то небольшое её количество может быть выполнено до обработки внешнего воздействия. Компьютер реагирует быстро, но не обязательно мгновенно. Иными словами, внешнее воздействие приоритетнее дивергенции, но не приоритетнее конечной τ-активности.

3►Другим примером может служить операция *cancel*, которая должна отменить выполнение цепочки действий, инициированной предыдущим запросом, и запустить цепочку завершающих действий. Если приоритетов нет, у реализации есть выбор: продолжать выполнять цепочку действий или принять операцию cancel.

4►И понятно, что допустимо и такое поведение, когда реализация не реагирует на cancel до тех пор, пока полностью не выполнит всю цепочку действий, даже если операция cancel выдана сразу после инициации этой цепочки. Иными словами, операция cancel гарантированно ничего не отменяет.

Мы уже пытались ввести приоритеты в модели реализации и спецификации, учитывать их при определении конформности и при композиции компонентов. Теперь эту же задачу мы попытались решить в новой модели.

Слайд 11. История.

Теперь я попробую рассказать о том, откуда взялась наша новая модель: на что она похожа и чем отличается. Как обычно, что-то новое оказывается очень похожим на что-то старое и давно известное. Отличие, во-первых, в осмыслении, в интерпретации, и, во-вторых, в нюансах. Поэтому я буду вести рассказ в некотором историческом ключе, но нас будет интересовать не столько реальная история тестирования, сколько логическая история.

Исторически самой первой моделью была автоматная модель. В этой модели воздействие называется стимулом или input'ом, а наблюдается реакция или output. Стимулы и реакции обычно интерпретируются как некие сообщения, посылаемые из теста в реализацию – это стимул, или обратно, из реализации в тест – это реакции. При этом считается, что на каждый стимул в ответ выдаётся ровно одна реакция. Автомат изображается в виде ориентированного графа, вершины которого называются состояниями, а каждая дуга помечена парой (стимул, реакция) и называется переходом.

Здесь нужно подчеркнуть, что между стимулами и реакциями не предполагается никакой предустановленной связи. Какая реакция может быть выдана в ответ на тот или иной стимул, определяется только самой реализацией и, в общем, это может быть любая реакция. У нас просто есть два алфавита: алфавит стимулов и алфавит реакций. Поэтому любой ориентированный граф, дуги которого помечены символами из декартового произведения этих алфавитов, является автоматной моделью. Соответственно, любое префикс-замкнутое множество трасс является множеством трасс некоторого автомата.

Поначалу рассматривались только детерминированные и всюду-определённые по стимулам автоматы. Детерминизм означает, что в каждом состоянии по каждому стимулу определено

не более одного перехода. Всюду-определённость означает, что в каждом состоянии по каждому стимулу определён хотя бы один переход. Всё вместе означает, что в каждом состоянии по каждому стимулу определён ровно один переход. Для таких автоматов была создана достаточно хорошо разработанная теория, во всяком случае, по сравнению с теориями для других автоматов и других моделей. Работы по исследованию недетерминированных и частично-определённых автоматов ведутся и сегодня.

Однако оказалось, что такая автоматная модель не очень хорошо подходит для моделирования поведения некоторых систем. Оказалось, что некоторые системы могут выдавать в ответ на стимул не одну, а несколько реакций последовательно. Также они готовы принимать последовательность стимулов без обязательной выдачи промежуточных реакций. Таким образом, концепция «один стимул – одна реакция» была пересмотрена в сторону большей свободы. Теперь на переходах стали писать не пару «стимул-реакция», а только стимул или только реакцию. Естественно, алфавиты стимулов и реакций не должны пересекаться. Такая модель называется системой помеченных переходов – Labelled Transition System, сокращённо LTS. Точнее, это IOLTS – Input-Output LTS, поскольку символ на переходе обязательно должен быть либо стимулом, либо реакцией. Если символы на переходах берутся из одного алфавита, не разбитого на стимулы и реакции, то это просто LTS.

Разделив стимулы и реакции, мы получили большую свободу в моделировании систем, но возникли и вопросы. Например, что произойдёт, если после некоторой цепочки стимулов тест ожидает реакции, а реализация оказалась в состоянии, где нет переходов по реакциям? Возникает тупиковая ситуация, deadlock: тест ждёт реакций, а реализация ждёт стимулов. Проблема, конечно, не в самом deadlock’е, а в его обнаружении. Если тест каким-то образом может узнать, что реакций нет, то вместо бесконечного их ожидания он может просто выдать какой-нибудь стимул. Но тогда у нас возникает новое наблюдение, которое не является реакцией. Это наблюдение отсутствия реакций. По-английски оно называется quiescence – покой, неподвижность, молчание, и обозначается малой греческой буквой δ. Отношение трассовой конформности, учитывающее это новое δ-наблюдение, придумано Яном

Тритмансом и называется ioco – Input Output Conformance. Но нас сейчас интересует не само это отношение, а лежащая в его основе семантика взаимодействия.

Как может быть на практике реализовано δ-наблюдение? Может быть много разных способов. Самый простой – это предположение о том, что любая реакция должна выдаваться реализацией за ограниченное время. Если время ожидания реакций превышает это ограничение, этот тайм-аут, то фиксируется δ-наблюдение.

В исходной LTS-модели δ-наблюдению не соответствует никакой переход. Конечно, мы можем, как это часто делают, добавить такой переход, нарисовав в состоянии, где нет реакций, петлю и пометив её символом δ. У нас получится IOLTS, алфавит реакций которой пополнен символом δ. Однако теперь уже не любая IOLTS в таком пополненном алфавите может рассматриваться как модель реализации, поскольку δ-переход обязательно должен быть петлёй. Соответственно, теперь уже не любое префикс-замкнутое множество трасс, в которые входит символ δ, является множеством трасс некоторой реализации: трассы, в которых после δ стоит реакция, запрещены. Это и понятно: если реакций нет, то и после наблюдение этого факта реакций тоже нет.

Но дело не только в запрете некоторых трасс, дело в том, что появляются дополнительные зависимости между трассами реализации. Если в реализации есть трасса aδb, то в ней неизбежно есть и трасса ab, поскольку δ-переход – это петля. А это означает, что возникает необходимость в оптимизации тестов. Если трасса ab трактуется спецификацией как ошибка, то трасса aδb тоже ошибка. Поэтому в любом полном наборе тестов достаточно оставить тест, который ловит ошибку ab, и удалить тест, который ловит ошибку aδb.

Это, конечно, тривиальный случай зависимости между ошибками. Есть и куда менее очевидные зависимости.

В частности, даже некоторые трассы самой спецификации могут быть признаны ошибочными, точнее неконформными, поскольку они не могут встречаться в конформных реализациях. А

это означает, что при появлении такой трассы тестирование не нужно продолжать, хотя спецификация описывает такие продолжения, можно сразу выдавать вердикт fail. Анализ и удаление из спецификации неконформных трасс представляет собой непростую проблему. Нам удалось решить эту проблему для ioco и общей трассовой конформности, учитывающей наблюдения тех или иных deadlock'ов. Решение это непростое.

Но я бы хотел вернуться к семантике взаимодействия. Сначала на примере ioco-семантики. Итак, при тестировании мы либо посылаем стимул, либо ждём реакций. Посылая стимул, мы не ожидаем каких-то наблюдений, поскольку уверены в том, что реализация сделает единственное: примет наш стимул. Почему? Во-первых, потому, что для ioco предполагается, что реализация всюду определена по стимулам: в каждом её состоянии принимается каждый стимул. Во-вторых, реализация не может выдать реакцию, если мы её не ожидаем. Наоборот, ожидая реакции, мы можем получить либо реакцию, либо δ-наблюдение, но реализация не может принять стимул, поскольку мы никаких стимулов не посылаем.

Итак, при посылке стимула мы разрешаем реализации принимать этот стимул и не разрешаем её делать что-то другое. При ожидании реакций мы разрешаем реализации выдать реакцию или ничего не делать, если реакций нет, но не разрешаем совершать переходы по стимулам. В обоих случаях мы какие-то действия разрешаем выполнять, а какие-то запрещаем. А тогда само это разрешение и запрет можно понимать как тестовое воздействие, которое сводится к множеству разрешаемых действий.

В общей модели LTS символы на переходах не подразделяются на стимулы и реакции, и называются просто действиями. Кроме того, допускаются переходы по ненаблюдаемым действиям, τ-действиям. Взаимодействие моделируется с помощью, так называемой, машины тестирования. Такую машину сначала придумал Милнер, а потом её модифицировал Ван Глаббек. В машине Милнера каждому действию соответствует кнопка на клавиатуре машины. Нажимая кнопку, мы разрешаем реализации выполнять это действие, после чего кнопка автоматически отжимается.

Однако, для того, чтобы принять любую реакцию, мы должны были бы суметь одновременно нажать все кнопки реакций. Именно это и предложил Ван Глаббек, но только в его машине действию соответствует не кнопка, а переключатель, который может находиться в двух положениях: “on” и “off”. Это генеративная машина: реализация непрерывно выполняет действия, соответствующие тем переключателям, которые находятся в положении “on”, а оператор машины может в любой момент любым способом изменить положение переключателей. В общем-то, как пишет сам Ван Глаббек, реактивная машина Милнера и генеративная машина, в конечном счёте, эквивалентны по мощности тестирования, если в машине Милнера разрешить нажимать сразу несколько кнопок.

Что мы можем наблюдать на такой машине тестирования? Прежде всего, выполняемые реализацией наблюдаемые действия, то есть все, кроме τ, – они высвечиваются на дисплее машины. С помощью специальной зелёной лампочки можно наблюдать сам факт того, что реализация что-то делает, то есть выполняет какие-то переходы. Это важно, если на дисплей ничего не высвечивается, то есть выполняются только τ-переходы. Когда дисплей погашен и зелёная лампочка не горит, это означает наблюдаемый deadlock: реализация не может выполнить ни одно из разрешённых действий. Это наблюдение называется отказом и определяется множеством разрешённых, но не выполнимых, действий – refusal set. Трассы с отказами называются failure traces, а семантика взаимодействия, учитывающая отказы, называется failure trace семантикой.

Здесь важно отметить, что в машине Ван Глаббека нет никаких ограничений по установке тех или иных переключателей, то есть нет ограничений на множество разрешаемых действий. Именно поэтому в систематизации конформностей по Глаббеку отсутствует такое отношение конформности как ioco. Для ioco разрешить можно либо один стимул, либо все реакции.

Как обобщение и машины Ван Глаббек и ioco-семантики мы предложили, так называемую, R/Q-машину. В ней вместо переключателей опять кнопки, которые можно нажимать только по одной, как в машине Милнера. Но у нас кнопка разрешает не одно

действие, а множество действий. Кроме того, кнопки делятся на R-кнопки и Q-кнопки, отказ наблюдаем только для R-кнопок.

Например, для ioco будет отдельная Q-кнопка для каждого стимула и ещё одна R-кнопка для приёма реакций и δ-наблюдения. Часто рассматриваются семантики с наблюдением блокировки стимулов: это когда в IOLTS тест посылает стимул, а реализация не всюду определена по стимулам и находится в состоянии, где нет перехода по этому стимулу. Тут тоже возникает deadlock, который называется блокировкой стимула. Если блокировка стимула наблюдаема, его кнопка будет R-кнопкой, а если нет, как для ioco, то Q-кнопкой. Для failure trace семантики будет кнопка для каждого непустого подмножества алфавита действий.

Для R/Q-семантики общего вида также могут быть неконформные трассы спецификации. Нам удалось решить проблему удаления таких неконформных трасс и в общем случае. Однако, в отличие от ioco, для R/Q-семантики общего вида неконформные трассы спецификации – это лишь частный случай зависимости между ошибками. В общем случае приходится говорить о множественной зависимости.

Теперь попробуем разобраться: что же мы приобрели и что потеряли, перейдя от автомата и IOLS к общей модели LTS, машине тестирования и R/Q-семантике, в частности, ioco-семантике?

Понятно, что приобрели мы новые наблюдения, что, конечно, усиливает наши возможности моделирования систем. Но одновременно мы получили предустановленную связь между тестовыми воздействиями и наблюдениями. Из-за этого уже не всякие LTS в алфавите тестовых воздействий и наблюдений годятся в качестве модели: отказы должны изображаться петлями. Не всякие трассы в алфавите тестовых воздействий и наблюдений допустимы. Но плохо не это, а то, что не всякие префикс-замкнутые множества даже допустимых трасс годятся в качестве трассовых моделей, то есть не всегда они порождаются соответствующими LTS-моделями. Как результат появляются зависимости между ошибками, анализ которых иногда представляет собой сложную проблему.

Глядя на этот путь развития, может возникнуть закономерный вопрос: а почему какие-то новые наблюдения нельзя было считать просто реакциями? Мне кажется, отчасти это вызвано интуитивным пониманием стимулов и реакций как сообщений посылаемых туда и обратно. Отсутствие реакций, конечно, таким сообщением не является, так же как блокировка стимула, да и вообще любой отказ. Точно так же ожидание реакций не является сообщением. Но тогда это вопрос интерпретации стимула и реакции. Если под стимулом понимать произвольное тестовое воздействие, а под реакцией – произвольное наблюдение, ограничения снимаются.

Правда, остаётся ещё одна проблема: добавленные тестовые воздействия (например, ожидание реакций) и наблюдения (отказы) несут в себе определённую смысловую нагрузку. Именно из-за этого возникают ограничения на допустимые LTS и множества трасс, что и вызывает зависимость между ошибками. Эти ограничения означают, что мы рассматриваем определённые подклассы LTS-моделей и трассовых моделей.

Однако в формальных, то есть математических, моделях иногда полезно абстрагироваться от таких смысловых нагрузок для того, чтобы проблема стала более ясной и прозрачной. Такое абстрагирование всегда означает упрощение. Проблема при этом остаётся, но её формулировка существенно меняется, также как и способы решения.

Вот это я и хочу продемонстрировать в новой модели. Мы придумали такую модель, в которой, самой по себе, нет ограничений на рассматриваемые классы LTS-реализаций. Как следствие, нет зависимостей между ошибками, кроме неизбежной зависимости по префиксу: если трасса ошибочна, то любое её продолжение тоже ошибочно. Старые модели оказываются частными случаями этой новой модели, но применяются для определённых подклассов реализаций, что и вызывает зависимость между ошибками на этих подклассах.

Это становится частным случаем общей ситуации, когда вместо класса всех реализаций рассматривается какой-то подкласс. Таким образом, мы сводим проблему зависимости между ошибками для тех или иных семантик взаимодействия к проблеме сужения нашей новой модели на те или иные подклассы

реализаций. И здесь можно отметить, что ограничение рассматриваемых реализаций тем или иным подклассом может быть вызвано и совсем другими причинами. Такие ограничения часто практически обоснованы и используются на практике.

Появление новых наблюдений, а именно, отказов сказалось и на проблеме композиции.

Для LTS имеется более-менее стандартный оператор их композиции. Он встречается в различных модификациях, но суть везде одна и та же. Композиционное состояние – это пара состояний LTS-операндов. Действия разделяются на синхронные и асинхронные. Переходы по асинхронным действиям наследуются из LTS-операндов, также наследуются τ-переходы в операндах. А пара синхронных действий в обоих LTS-операндах порождает τ-переход в композиции.

И всё получается хорошо, пока единственным наблюдениями являются действия, написанные на переходах LTS. Трассы действий обладают полезным свойством *аддитивности* относительно композиции: множество трасс композиции LTS совпадает с множеством попарных композиций трасс LTS-операндов.

Но что происходит с отказами? К сожалению, отказ в композиционном состоянии невозможно представить как результат композиции каких-то двух наблюдений в состояниях-операндах. Из-за этого невозможно так определить композицию трасс с отказами, чтобы выполнялось свойство аддитивности. Именно из-за этого возникает несохранение конформности при композиции со всеми вытекающими проблемами. Когда мы решали эту проблему для R/Q-семантики, нам пришлось в качестве инструмента придумать специальные фи-трассы, которые обладают свойством аддитивности и по которым можно вычислить трассы с отказами.

Эти фи-трассы похожи на, так называемые, трассы готовности, ready traces. В трассах готовности вместо отказов находятся другие наблюдения, так называемые, множества готовности. Это множество внешних действий по которым в текущем состоянии реализации определены переходы. Он могут наблюдаться при тех же условиях, при которых наблюдаются отказы, но, конечно, семантика взаимодействия должна позволять

делать такие наблюдения. На практике что-то похожее на множества готовности возникают в графических интерфейсах, когда на экране монитора появляется меню тех действий, которые программа может выполнить.

Трассы готовности тоже обладают свойством аддитивности, поскольку множество готовности композиционного состояния вычисляется по множествам готовности состояний-операндов.

Что касается проблемы приоритетов, то я уже говорил, что в современных моделях приоритетов, к сожалению, нет. Для машины тестирования Ван Глаббека отсутствие приоритетов означает, что реализация может выполнить определённое в ней действие при единственном условии: переключатель этого действия должен находиться в положении “on”, и независимо от положения других переключателей. Вот эта «независимость» и означает отсутствие приоритетов. Мы сделали одну попытку ввести приоритеты для R/Q-семантики. В общем-то она получилась удачной, хотя и не без недостатков.

Слайд 12. Новая модель.

Теперь я готов рассказать о новой модели тестового взаимодействия, модели реализации, модели спецификации и отношении конформности между реализациями и спецификациями.

Основная идея заключается в том, чтобы явно указывать в модели реализации все тестовые воздействия, которые я буду называть кнопками, и все наблюдения, которые могут быть сделаны при тестировании. К таким наблюдениям могут относиться не только реакции в IOLTS или действия в LTS, но также отказы, множества готовности и тому подобное. При этом мы будем считать, что между тестовыми воздействиями и наблюдениями нет никакой предустановленной связи, как для стимулов и реакций.

Семантика взаимодействия.

Будем считать, что заданы два непересекающихся универсума символов: ***B*** – тестовых воздействий (кнопок – ***b****uttons*) и ***O*** – наблюдений (***o****bservations*). Такую семантику будем называть ***B/O-***

семантикой. Трасса – это последовательность в объединённом алфавите кнопок и наблюдений.

Машина тестирования.

В машине тестирования клавиатура представляет собой множество кнопок $\boldsymbol{B}$. Тестовое воздействие осуществляется нажатием той или иной кнопки на клавиатуре. Когда нажимается кнопка, машина тестирования передаёт в реализацию однократный сигнал о соответствующем тестовом воздействии и дожидается ответного сигнала о том, что реализация «приняла к сведению» это тестовое воздействие, после чего машина может передавать в реализацию сигнал о следующем тестовом воздействии. Кнопка не фиксируется (автоматически отжимается), что даёт возможность оператору машины нажимать следующую (другую или ту же самую) кнопку. Одновременно можно нажимать только одну кнопку, соответствующую ровно одному тестовому воздействию.

На дисплее машины последовательно высвечиваются символы кнопок и наблюдений, то есть символы из $\boldsymbol{B \cup O}$. Последовательность символов, появляющихся на экране, как раз и является трассой, наблюдаемой в процессе тестового эксперимента. Символ кнопки появляется на экране в тот момент, когда оператор нажимает соответствующую кнопку. Наблюдение появляется на экране дисплея тогда, когда в реализации происходит соответствующее наблюдаемое событие. Ненаблюдаемая τ-активность реализации никак не отражается на экране дисплея.

Для того чтобы можно было выполнять несколько тестовых экспериментов, машина может быть снабжена кнопкой *рестарта*. Эта кнопка сбрасывает реализацию в начальное состояние и гасит экран. Каждый новый рестарт машины может вызывать изменение погодных условий, от которых зависит поведение реализации. Гипотеза о глобальном тестировании предполагает, что в последовательности рестартов воспроизводятся все возможные погодные условия.

В то же время рестарт позволяет выполнить не более чем счётное число тестовых экспериментов, тем самым, для не более чем счётного числа погодных условий. Для того чтобы обойти это ограничение, машина тестирования может быть снабжена не кнопкой рестарта, а кнопкой *репликации*. Однократное нажатие

такой кнопки создаёт множество копий машины тестирования произвольной мощности. Тестирование происходит с каждой копией машины независимым образом, то есть на каждой копии выполняется свой тестовый эксперимент. Для каждой копии фиксируется свой вариант погодных условий. Гипотеза о глобальном тестировании предполагает, что для каждого варианта погодных условий при репликации создается, по крайней мере, одна копия машины.

Важно отметить, что для конформностей типа редукции репликацию достаточно делать один раз перед началом тестирования, а не много раз после получения тех или иных трасс. Многократная репликация (после каждого шага тестирования, то есть после каждого наблюдения и после нажатия каждой кнопки) требуется для конформностей типа симуляции.

Реализация.

Для конформности типа редукции реализация, фактически, сводится к множеству её трасс. Такая *трассовая модель реализации* формально определяется как множество конечных трасс в алфавите кнопок и наблюдений, которое: 1) не пусто, 2) префикс-замкнуто, 3) вместе с каждой трассой содержит её продолжение каждой кнопкой. Последнее объясняется тем, что оператор машины может в любой момент времени нажать любую кнопку и она тут же отобразится на экране дисплея, продолжив трассу.

Для компактного задания множества трасс, в частности, для задания бесконечного множества трасс конечным образом, используется модель LTS как порождающий граф этого множества трасс. Поскольку множество трасс префикс-замкнуто, все состояния графа считаются конечными. LTS будет конечной, если множество трасс регулярно.

Поскольку тестовое воздействие (нажатие кнопки машины тестирования) на реализацию выполняется извне её и не зависит от неё, переход по кнопке означает лишь тот факт, что реализация «узнала» о выполненном тестовом воздействии. В результате такого перехода реализация меняет своё состояние, что впоследствии может привести к изменению её поведения, то есть к появлению других наблюдений. Если в некотором состоянии реализация игнорирует тестовое воздействие, то это эквивалентно

тому, что в этом состоянии есть переход-петля по этой кнопке. Поэтому отсутствие перехода по кнопке в состоянии реализации трактуется как наличие перехода-петли по этой кнопке в этом состоянии.

Будем считать, что переходы выполняются мгновенно. В каждом состоянии реализация ожидает некоторое конечное время, после чего она должна выполнить один из выходящих переходов. Эти предположения обычно называют *progress assumption*. Также предполагается, что за конечное время реализация может пройти только конечный маршрут. Это означает, что реализация рассматривается как дискретная система. Для этого достаточно, например, считать, что время ожидания в каждом состоянии ограничено снизу некоторой константой, большей нуля.

Взаимодействие с реализацией.

При тестировании в любой наблюдаемой трассе подпоследовательность кнопок содержит ровно те кнопки, которые оператор машины нажимал и ровно в том порядке, в котором он их нажимал. С внешней точки зрения тестовое воздействие влияет лишь на те наблюдения, которые появляются в трассе. Иными словами, нажатие кнопки лишь регулирует поток наблюдений над поведением реализации. Это достигается с помощью переходов по кнопкам, которые меняют состояние реализации при нажатии кнопки и, тем самым, дальнейший поток наблюдений.

Что касается ненаблюдаемого поведения реализации, то мы исходим из *основного допущения о τ-активности*, которое распространяем не только на наблюдения, но и на кнопки: между любыми двумя переходами по наблюдениям или кнопкам (и перед первым таким переходом) в реализации может быть любая конечная τ-активность.

Особенностью нашей модели взаимодействия является *приоритет тестового воздействия над поведением реализации*, как наблюдаемым, так и ненаблюдаемым. Если после получения в тестовом эксперименте трассы σ оператор не нажимает кнопку, а ждёт наблюдений, то может быть получено любое наблюдение, которое в реализации есть после трассы σ, то есть может быть получена любая имеющаяся в реализации трасса σu, где u –

наблюдение. Однако по основному допущению о τ-активности перед переходом по наблюдению *u* реализация может выполнить любое конечное число τ-переходов.

Кроме того, если после трассы σ в реализации есть бесконечная τ-активность, то есть дивергенция, то никаких наблюдений может и не быть.

Если же сразу после наблюдения трассы σ оператор нажимает кнопку *p*, то реализация обязана выполнить переход по кнопке *p*, будет получена трасса σ*p*. Однако по основному допущению о τ-активности перед переходом по кнопке *p* реализация может выполнить любое конечное число τ-переходов.

Итак, реализация обязана «принять к сведению» тестовое воздействие через конечное время после нажатия соответствующей кнопки. В то же время мы никак не оговариваем, что означает это «принятие к сведению», допускается и простое игнорирование тестового воздействия, что в LTS моделируется переходом-петлёй по этой кнопке (отсутствие перехода по кнопке интерпретируется как наличие такой петли).

Слайд 13. Новая модель.

Протокол взаимодействия. В результате мы получаем следующий протокол взаимодействия. Если нет тестового воздействия, то есть никакая кнопка не нажата, реализация может выполнить, в зависимости от погодных условий, любую цепочку переходов по наблюдениям и τ-переходов, начинающуюся в её текущем состоянии. Если осуществляется тестовое воздействие, то есть оператор нажал кнопку *p*, то реализация может выполнить, в зависимости от погодных условий, любую конечную цепочку τ-переходов, после чего должна выполнить любой переход по кнопке *p*. Выполняя *p*-переход, реализация как бы «сообщает» машине тестирования о том, что тестовое воздействие ею воспринято. После этого реализация готова к восприятию следующего тестового воздействия (нажатию той или иной кнопки).

При наличии в состоянии нескольких переходов, которые реализация может выполнять, выбирается один из них

недетерминированным образом. Также при нажатой кнопке выбор конечного τ-маршрута, который проходится до перехода по кнопке, происходит недетерминировано. Оба этих выбора понимаются как выборы в зависимости от погодных условий. Гипотеза о глобальном тестировании гарантирует возможность перебора всех возможных погодных условий.

Оператор машины тестирования.

Оператор машины тестирования моделирует работу тестовой системы. Мы предполагаем, что при тестировании оператор выполняет тест, понимаемый как инструкция оператору. В этой инструкции указывается, что может делать оператор после получения той или иной трассы: ждать наблюдений и/или нажимать кнопки и какие именно кнопки. Естественно, что если тест определяет поведение оператора после трассы, то для того, чтобы можно было получить эту трассу, тест должен определять поведение оператора и после любого префикса трассы.

Если тест разрешает оператору нажимать кнопку p после трассы μ, то предполагается, что оператор может нажать эту кнопку через любое время после получения трассы μ. Тем самым, оператору *не запрещено* выдерживать любые паузы после получения тех или иных трасс, в том числе перед нажатием следующей кнопки: в любой момент времени он может устроить себе «перерыв на чай».

В то же время для полноты тестирования необходимо, чтобы любая интересующая нас трасса реализации могла наблюдаться при взаимодействии с реализацией через машину тестирования. А для этого оператор должен иметь *возможность* достаточно быстро нажимать кнопки после полученных трасс. Это значит, что задержка между получением трассы и нажатием следующей кнопки хотя бы в одном сеансе тестирования может оказаться меньше, чем время ожидания реализацией в состоянии после трассы.

Слайд 14. Новая модель: дополнительное обоснование.

Сейчас, после того как модель реализации и взаимодействие с ней через машину тестирования определены, я хочу немного

остановиться на дополнительном обосновании этой модели. Рассмотрим шесть вопросов, которые возникают в связи с этой моделью.

1. Когда кнопка попадает в трассу?

Нажатие кнопки выполняется оператором машины тестирования, и в этот момент времени он знает, какая трасса уже получена. Поэтому ничто не мешает оператору отметить тот факт, что он нажал данную кнопку после наблюдения данной трассы.

С другой стороны, при выбранном протоколе взаимодействия после нажатия кнопки выполниться может только переход по кнопке, если не считать ненаблюдаемых τ-переходов. Это объясняется тем, что нажатие кнопки блокирует наблюдения. А поскольку нажатие кнопки также блокирует дивергенцию и переход по кнопке определён в каждом состоянии реализации, такой переход обязательно будет выполнен после нажатия кнопки.

Тем самым появление символа кнопки на экране дисплея в момент нажатия кнопки, фактически, эквивалентно его появлению в момент совершения реализацией перехода по этой кнопке. Поэтому отдельное наблюдение перехода по кнопке излишне. Наблюдаемая на экране дисплея трасса совпадает с трассой маршрута, который реализация за это время проходит. Это самое главное.

2. Почему нажатие кнопки блокирует наблюдения?

Если нажатие кнопки не блокирует наблюдения, то появляется дополнительная зависимость между трассами реализации (и, следовательно, между ошибками). Поясним это на примере. Пусть при взаимодействии с реализацией может наблюдаться трасса *up*, где *u* наблюдение, а *p* кнопка. Тогда, поскольку наблюдается трасса *up*, то наблюдается и её префикс – трасса *u*. Если оператор нажимает кнопку *p* до наблюдения *u*, но наблюдения не блокируются этим нажатием, то реализация всё равно может выполнить переход по *u*. Тем самым, будет наблюдаться трасса *pu*. Следовательно, если при взаимодействии с реализацией может наблюдаться трасса *up*, то может наблюдаться и трасса *pu*.

В выбранной нами модели взаимодействия такой дополнительной зависимости между трассами нет. В то же время

режим работы с отсутствием блокировки наблюдений при нажатии кнопки легко моделируется в нашей модели.

Для этого достаточно систематически выполнить следующее преобразование реализации, пока оно возможно: если в реализации имеются переходы $s—p→s_p$, $s—u→t$ и $t—p→t_p$, то добавим переход $s_p—u→t_p$. Тем самым, если в состоянии s начиналась трасса up, заканчивающаяся в состоянии t_p, то теперь будет и трасса pu, заканчивающаяся в том же состоянии.

В результате такого моделирования класс всех реализаций для модели без блокировки отображается в строгий подкласс всех реализаций для модели с блокировкой. Как и в общем случае, такое сужение класса реализаций приводит к появлению зависимостей между трассами реализации (в частности, между ошибками).

Кроме этого, блокировка наблюдений является следствием приоритета тестового воздействия над наблюдениями. Такой приоритет необходим для того, чтобы можно было моделировать поведение систем с приоритетами. В частности, для прерывания цепочки внешних действий командой «отменить» (*cancel*).

3. Зачем оператору нужно быстро нажимать кнопки?

Прежде всего, отметим, что мы исходим из основного допущения о τ-активности: в реализации τ-активность может быть перед и после любого наблюдения, а также перед и после любого перехода по кнопке. Понятно, что любые ограничения на τ-активность только сузили бы класс рассматриваемых реализаций, что могло бы привести к появлению дополнительных зависимостей между ошибками. Наличие τ-активности ещё не означает, что она обязательно будет выполняться, но, конечно, предполагается, что хотя бы при некотором взаимодействии она выполняется. Естественно, что τ-активность может выполняться, когда никакая кнопка не нажата. Блокирует ли нажатие кнопки τ-активность или нет, мы рассмотрим в следующем пункте.

Для полноты тестирования нам нужно, чтобы любой достижимый переход в LTS-реализации мог быть выполнен при том или ином взаимодействии с ней (в зависимости от поведения оператора и погодных условий, моделирующих

недетерминированное поведение реализации). Если это не так, и какой-то переход не выполняется ни при каком взаимодействии, то это эквивалентно отсутствию этого перехода в реализации. А это, в свою очередь, приводит к сужению класса рассматриваемых реализаций, что также чревато появлением дополнительных зависимостей между ошибками.

Почему мы требуем, чтобы оператор мог нажимать кнопку достаточно быстро после получения трассы, хотя и не обязан это делать всегда? Если оператор не может нажать кнопку достаточно быстро после трассы, то реализация может успеть выполнить после этой трассы один или несколько τ-переходов. Тем самым, переход по кнопке, начинающийся в состоянии до этих τ-переходов, никогда не будет выполнен. В то же время такое быстрое нажатие кнопки не означает, что будет выполнен самый первый переход по кнопке. Допущение о τ-активности и гипотеза о глобальном тестировании гарантируют, что любой переход по кнопке, достижимый из текущего состояния по τ-переходам, может быть выполнен после быстрого нажатия кнопки в одном из прогонов теста.

Таким образом, оператор должен уметь быстро нажимать кнопки для обеспечения полноты тестирования.

4. Почему нажатие кнопки не блокирует τ-активность?

Здесь мы снова опираемся на требование выполнимости каждого достижимого перехода. Если нажатие кнопки блокирует τ-активность, то для того, чтобы реализация могла выполнить некоторую цепочку τ-переходов (а после неё переход по кнопке), оператор не должен нажимать кнопку до тех пор, пока эта цепочка не будет выполнена, и должен нажать кнопку сразу же после выполнения этой цепочки. Поскольку τ-активность ненаблюдаема, оператор должен просто выждать некоторый интервал времени, прежде чем нажать кнопку. Тем самым, к оператору предъявляются весьма нетривиальные требования по скорости его работы: после получения трассы он должен выдерживать паузу, прежде чем нажимать кнопку, причём длительность этой паузы должна быть, вообще говоря, произвольной в разных сеансах тестирования.

Вместо этого мы выбрали вариант, когда нажатие кнопки не блокирует τ-активность. Тогда к оператору предъявляется только одно требование, рассмотренное в предыдущем пункте: он должен уметь нажимать кнопку достаточно быстро после получения трассы, хотя и не обязан это делать всегда. У реализации появляется выбор: выполнять τ-переход или переход по нажатой кнопке. Как обычно, этот выбор недетерминирован и определяется погодными условиями.

5. Почему нажатие кнопки блокирует дивергенцию?

Нажатие кнопки не блокирует τ-активность, но разрешает только конечную τ-активность, то есть разрешает выполнять только конечное число τ-переходов. Тем самым, нажатие кнопки блокирует дивергенцию.

Это необходимо для того, чтобы реализовать «выход из дивергенции», то есть приоритет тестового воздействия над дивергенцией.

6. Почему переход по каждой кнопке определён в каждом состоянии реализации?

До сих пор мы предполагали, что переход по кнопке определён в каждом состоянии LTS-реализации (по умолчанию отсутствие такого перехода в состоянии трактуется как наличие перехода-петли). На самом деле, это требование не столь принципиально: если его опустить, то это влияет лишь на условие выполнения τ-активности при нажатой кнопке.

Новое условие такое: реализация может не выполнить никакого перехода по нажатой кнопке p только в том случае, если она после конечного числа τ-переходов будет бесконечно двигаться по бесконечному τ-маршруту, который проходит только через такие состояния, где нет переходов по кнопке p. В противном случае реализация выполняет конечное число τ-переходов и затем переход по кнопке p.

LTS-реализацию, в которой переходы по кнопкам определены не во всех состояниях, можно промоделировать с помощью LTS, в которой такие переходы есть во всех состояниях. При внешнем взаимодействии они будут неразличимы.

Для этого в исходную LTS-реализацию вносятся следующие изменения.

1. Если переход по кнопке *p* отсутствует в стабильном состоянии *s*, то возникает *deadlock*: реализация не может выполнить переход по *p* или τ-переход, поскольку их нет, и не может выполнить переход по наблюдению, поскольку такие переходы блокируются нажатой кнопкой *p*, а разблокированы могут быть только после перехода по *p*. Внешне (для оператора машины тестирования) такой *deadlock* выглядит как отсутствие наблюдений. Из такого *deadlock*`а можно выйти, нажав другую кнопку, по которой в стабильном состоянии *s* есть переход. В нашей модели это реализуется добавлением перехода $s \text{—} p \rightarrow s`$, ведущего в новое состояние *s*`, в котором определяются переходы-петли $s` \text{—} q \rightarrow s`$ по всем кнопкам *q*, по которым из состояния *s* нет переходов, а также переходы по кнопкам, по которым есть переходы из состояния *s*, ведущие туда же, куда они ведут из состояния *s*: переход $s_p \text{—} r \rightarrow t$ проводится, когда есть переход $s \text{—} r \rightarrow t$.

2. Если переход по кнопке *p* отсутствует в нестабильном состоянии *s*, в котором не начинается бесконечный τ-маршрут, проходящий только конечное число раз через состояния, где есть переход по кнопке *p*, то через конечное число τ-переходов будет выполнен какой-нибудь переход по *p*. Нам достаточно добавить любой переход $s \text{—} p \rightarrow t`$, если из состояния *s* по τ-переходам достижимо состояние *t* и имеется (или добавлен в п.1) переход $t \text{—} p \rightarrow t`$.

3. Если переход по кнопке *p* отсутствует в дивергентном, состоянии *s*, в котором начинается бесконечный τ-маршрут, проходящий только конечное число раз через состояния, где есть переход по кнопке *p*, то возможно, что реализация будет проходить именно этот бесконечный τ-маршрут. В этом случае может не выполниться никакой переход по *p*, и переходы по наблюдениям останутся заблокированы. Внешне (для оператора

машины тестирования) такая дивергенция выглядит как отсутствие наблюдений. Из неё можно выйти, нажав другую кнопку, для которой условие этого пункта не будет выполняться. Это полностью аналогично п.1, при моделировании в нашей модели выполняются те же изменения в реализации, которые описаны в п.1.

Слайд 15. Новая модель.

Спецификация и общая редукция.

Как я уже говорил, спецификация определяет множество разрешённых трасс и одновременно его дополнение – множество ошибок 1-го рода. Поскольку спецификация нам нужна только для генерации тестов, которые предназначены для ловли ошибок, то удобнее считать, что спецификация непосредственно определяет именно дополнение – множество ошибок 1-го рода. Тем самым, трассовая спецификация – это произвольное множество конечных трасс, понимаемых как ошибки 1-го рода. Реализация конформна, если в ней нет ошибок 1-го рода, то есть трасс спецификации. Такую конформность будем называть далее *общей редукцией*.

Как и реализацию, спецификацию как множество трасс можно задавать с помощью порождающего графа, то есть LTS. Но только теперь в ней уже нужно выделить конечные вершины, поскольку множество ошибок 1-го рода не является префикс-замкнутым.

Спецификация задаёт класс конформных реализаций. Если спецификация содержит пустую трассу, то есть пустая трасса считается ошибкой 1-го рода, то все реализации неконформны, поскольку любая реализация содержит пустую трассу. Если спецификация – это пустое множество, то все реализации конформны.

Тест.

Тестом будем называть множество конечных трасс. Получение при тестировании любой из этих трасс приводит к вынесению вердикта ***fail***. Если тест значимый, то все его трассы – это ошибки. Если получена трасса, которая трассой теста не является, но является строгим префиксом некоторой трассы теста,

вердикт не выносится и тест продолжается. Если получена трасса, которая не является префиксом какой-либо трассы теста, выносится вердикт ***pass***.

Тест *примитивный*, если он содержит только одну трассу. Очевидно, что примитивный тест детерминированный. Любой тест эквивалентен объединению множества примитивных тестов в том смысле, что они выносят вердикт ***fail*** для одних и тех же реализаций. Также очевидно, что спецификация (как множество ошибок 1-го рода) является полным тестом на классе всех реализаций. Отсюда следует, что набор примитивных тестов, построенных по всем ошибкам 1-го рода, то есть по всем трассам спецификации, является полным на классе всех реализаций.

Слайд 16. Нормализация.

Нормализация спецификации и оптимизация тестов.

До сих пор мы не налагали никаких ограничений на спецификацию, она могла быть произвольным множеством конечных трасс, трактуемых как ошибки 1-го рода. Однако понятно, что если строгий префикс трассы – это ошибка, то такую трассу можно удалить из спецификации. Если спецификация содержит трассу, заканчивающуюся кнопкой, то из этой трассы можно удалить эту кнопку. Эти две процедуры в конечном счёте строят нормализованную спецификацию, в которой, во-первых, никакая трасса не является префиксом другой трассы, и, во-вторых, каждая трасса не заканчивается кнопкой, то есть либо пуста, либо заканчивается наблюдением. Понятно, что нормализованная спецификация эквивалентна исходной, то есть определяет тот же самый класс конформных реализаций на классе всех реализаций.

Нормализованные спецификации взаимно-однозначно соответствуют своим классам конформных реализаций, то есть разные нормализованные спецификации определяют разные классы конформных реализаций. Иными словами, эквивалентность спецификаций, как совпадение их классов конформных реализаций, для нормализованных спецификаций совпадает с их равенством (как множеств ошибок).

Набор тестов полный тогда и только тогда, когда все трассы всех его тестов – это все трассы нормализованной спецификации и, возможно, некоторые их продолжения. Очевидная оптимизация – это удаление таких продолжений. Так оптимизированный набор тестов будем называть нормализованным. Тем самым, нормализованный полный набор тестов – это покрытие нормализованной спецификации. Нормализованный набор примитивных тестов содержит по одному тесту на каждую трассу нормализованной спецификации.

Слайд 17. Классы реализаций.

Класс реализаций.

По разным причинам в качестве тестируемых реализаций часто рассматриваются не любые реализации, а принадлежащие тому или иному подклассу реализаций. Это приводит к появлению дополнительных зависимостей между ошибками и, соответственно, даёт возможность дополнительной оптимизации тестов.

Фактически, речь идёт о том, что на подклассе реализаций могут быть эквивалентные, но не совпадающие нормализованные спецификации. Вот рисунок поясняет, почему так может быть.

Слайд 18. Моделирование старых семантик.

К новой модели могут быть сведены конформности типа редукции в самых разнообразных семантиках, в которых в качестве наблюдений используются стимулы и реакции, действия, отказы, в том числе δ – «отсутствие реакций», множества готовности и т.д.

LTS-модель, используемую для этих семантик, можно назвать LTS *действий* или ATS от слова *action* – действие. Соотвественно нашу LTS-модель, в которой переходы помечаются, кроме кнопок и τ, наблюдениями, можно назвать LTS *наблюдений* или OTS от слова *observation* – наблюдение.

Вот на слайде слева показано, как происходит сведение ATS к OTS. Это сведение для разных семантик, конечно, различно, поскольку они основаны на разных типах наблюдений. Для каждой

старой семантики определяются два преобразования: преобразование α ATS-реализации в OTS-реализацию и преобразование β ATS-спецификации в OTS-спецификацию. Последнее преобразование, кроме прочего, учитывает ещё и то, чито в нашей модели спецификация описывает не столько правильное поведение, сколько ошибочное.

Новая OTS-реализация – результат преобразования старой ATS-реализации, а новая OTS-спецификация – результат преобразования старой ATS-спецификации. В результате такого моделирования новая OTS-реализация конформна в новой семантике, то есть по новой конформности, новой OTS-спецификации тогда и только тогда, когда старая ATS-реализация была конформна старой ATS-спецификации.

На слайде справа показано соотношение различных классов реализаций в новой модели. Подкласс ***Impl*** – это подкласс старых ATS-реализаций, которые можно было тестировать для проверки старой конформности старой ATS-спецификации. Он преобразуется в подкласс $\alpha(\boldsymbol{Impl})$ класса всех OTS-реализаций. Старая ATS-спецификация S определяла подкласс конформных ATS-реализаций $\boldsymbol{Conf}(S)$, который преобразуется в подкласс $\alpha(\boldsymbol{Conf}(S))$. С другой стороны старая ATS-спецификация S преобразуется в новую OTS-спецификацию $\beta(S)$, которая определяет подкласс конформных реализаций $\boldsymbol{Conf}(\beta(S))$. Суть в том, что пересечение этого подкласса с подклассом α ($\boldsymbol{Impl}$) совпадает с подклассом $\alpha(\boldsymbol{Conf}(S))$.

Сейчас я, конечно, не буду рассказывать, как делаются преобразования α и β для разных семантик. Они вполне алгоритмические и, в общем, довольно простые.

Как уже говорилось, на классе всех реализаций нет зависимости между ошибками, кроме тривиальной префиксной зависимости. Для нормализованной спецификации такой зависимости между ошибками 1-го рода нет. В частности, эквивалентные нормализованные спецификации равны. Поэтому никакой оптимизации тестов, кроме тривиальной их нормализации, не требуется. Тем самым эта картинка показывает то, что мы и хотели получить: сведение зависимости между ошибками в той или

иной семантике к общей проблеме появления таких зависимостей в новой модели при сужении класса тестируемых реализаций.

Слайд 19. Композиция в новой модели: LTS событий.

LTS-реализация, которую мы рассматривали для новой модели, – это LTS *наблюдений*. Однако для композиции наблюдения, вообще говоря, не годятся. Я уже говорил, что отказ в композиционном состоянии не может быть получен как композиция каких-то наблюдений в состояниях-операндах. Из-за этого трассы наблюдений не аддитивны относительно композиции.

Поэтому сейчас мы должны модифицировать нашу модель, преследуя цель добиться аддитивности трасс. Идея заключается в том, чтобы помечать переходы LTS такими символами (мы их назвали *событиями*), которые, с одной стороны, можно компоновать для обеспечения свойства аддитивности, а, с другой стороны, можно использовать для генерации наблюдений при тестировании. Переход по событию приводит к возникновению того или иного связанного с этим событием наблюдения. Последнее свойство можно назвать *генеративностью* трасс событий: по ним вычисляются все трассы наблюдений реализации. Такую модель можно назвать LTS *событий*. Универсум событий обозначим через $\boldsymbol{E}$ – от слова *event* – событие.

Для генерации наблюдений по событиям будем считать, что задана универсальная частично-определённая однозначная функция $\boldsymbol{f}:\boldsymbol{E}\rightarrow\boldsymbol{O}$. Событие из домена функции будем называть *наблюдаемым* событием.

Прежде всего, мы должны уточнить функционирование ETS-реализации. Теперь оно определяется не только её устройством, но и функцией $\boldsymbol{f}$. Разрешается переход только по наблюдаемым событиям, переходы по ненаблюдаемым событиям никогда не выполняются.

Здесь сразу возникает вопрос: зачем нужны ненаблюдаемые события, которые не отображаются ни в какие наблюдения и переходы по которым вообще не выполняются? Забегая вперёд можно ответить: такие события нужны для композиции. События в

состояниях LTS-операндов могут быть ненаблюдаемыми, а вот их композиция – наблюдаемой. Но композицией событий мы займёмся позже, а пока вернёмся в генерации наблюдений по событиям.

Функция ***f*** определяет естественное отображение ETS в OTS, когда переход по событию заменяется переходом по соответствующему наблюдению или удаляется, если такого наблюдения нет. Тестирование ETS-реализации *S* эквивалентно тестированию OTS-реализации ***f***(*S*). Также отображение ***f*** естественно распространяется на трассы событий и множества таких трасс.

Слайд 20. Параллельная композиция событий.

Мы определим параллельную композицию ETS в духе CSP (*Communicating Sequential Processes*). В CSP результатом композиции двух ATS является ATS, состояния которой – это пары состояний операндов, начальное состояние – это пара начальных состояний, а переход либо асинхронный и соответствует переходу в одном из операндов (меняется состояние только этого операнда), либо синхронный и соответствует паре переходов по одному и тому же – синхронному – действию в обоих операндах (измениться могут состояния обоих операндов). При этом синхронность или асинхронность переходов однозначно определяется действиями, которыми эти переходы помечены, то есть все внешние действия разбиваются на синхронные и асинхронные, а внутреннее действие τ всегда считается асинхронным. В обоих случаях действие на результирующем переходе совпадает с действием на одном переходе-операнде (асинхронный случай) или на обоих переходах-операндах (синхронный случай). После этого с помощью оператора ***hide*** некоторые синхронные действия на переходах «скрываются», то есть заменяются символом τ.

Для композиции ETS мы будем всегда считать все кнопки синхронными, а события могут быть как синхронными, так и асинхронными, что задаётся алфавитом синхронных событий *U*. Также будем считать, что задана частично определённая коммутативная композиция синхронных событий, результатом которой также является синхронное событие.

Тогда композиция ETS определяется следующими правилами вывода: для любых состояний левого операнда s и $s`$, любых состояний правого операнда $t$ и $t`$, любых событий a и b, и любой кнопки p :

$$a\in(\boldsymbol{E}\backslash U)\cup\{\tau\} \quad \& \; s\text{—}a\rightarrow s` \qquad\qquad \vdash st\text{—}a\rightarrow s`t,$$

$$b\in(\boldsymbol{E}\backslash U)\cup\{\tau\} \qquad\qquad \& \; t\text{—}b\rightarrow t` \qquad \vdash st\text{—}b\rightarrow st`,$$

$$p\in\boldsymbol{B} \qquad \& \; s\text{—}p\rightarrow s` \qquad \& \; t\text{—}p\rightarrow t` \qquad \vdash st\text{—}p\rightarrow s`t`.$$

$$(a,b)\in\boldsymbol{Dom}(|_U|) \quad \& \; s\text{—}a\rightarrow s` \qquad \& \; t\text{—}b\rightarrow t` \qquad \vdash st\text{—}a|_U|b\rightarrow s`t`.$$

На основе композиции событий также определяется композиция трасс событий с помощью следующих правил вывода: для любых трасс σ,σ_1,σ_2 , любых событий a и b, и любой кнопки p

$$\vdash \epsilon\in\epsilon|_U|\epsilon,$$

$$\sigma=\sigma_1|_U|\sigma_2 \quad \& \; a\in\boldsymbol{E}\backslash U \qquad \vdash \sigma\cdot a\in(\sigma_1\cdot a)|_U|\sigma_2,$$

$$\sigma=\sigma_1|_U|\sigma_2 \quad \& \; a\in\boldsymbol{E}\backslash U \qquad \vdash \sigma\cdot b\in\sigma_1|_U|(\sigma_2\cdot b),$$

$$\sigma=\sigma_1|_U|\sigma_2 \quad \& \; p\in\boldsymbol{B} \qquad \vdash \sigma\cdot p\in(\sigma_1\cdot p)|_U|(\sigma_2\cdot p),$$

$$\sigma=\sigma_1|_U|\sigma_2 \quad \& \; (a,b)\in\boldsymbol{Dom}(|_U|) \qquad \vdash \sigma\cdot(a|_U|b)\in(\sigma_1\cdot a)|_U|(\sigma_2\cdot b).$$

Если композиция трасс оказывается пустой, то будем говорить, что эти трассы не компонуются. Композиция трасс событий естественно распространяется на композицию множеств трасс событий, определяемую как объединение множеств всех попарных композиций.

Такая композиция обладает свойством аддитивности: для любых двух ETS S и T имеет место $\boldsymbol{tr}(S|_U|T) = \cup(\boldsymbol{tr}(S)|_U|\boldsymbol{tr}(T))$.

Заметим, что обычная композиция LTS действий и трасс действий является частным случаем композиции ETS и трасс событий, если считать, что действие – это событие, компонуются одинаковые действия и результат – то же действие.

После композиции с помощью оператора ***hide*** скрываются переходы по некоторым синхронным событиям, переходы по кнопкам не скрываются. Подмножество скрываемых синхронных событий задаётся явно. Очевидно, и после сокрытия свойство аддитивности сохраняется.

Слайд 21. Композиция спецификаций.

Для спецификаций нужна другая композиция, которую я вначале доклада называл косой композицией. При этом мы должны помнить, что в нашей модели спецификация описывает не правильное поведение, а множество ошибок. Достаточно очевидно, что на трассовом уровне такой косой композицией является множество трасс наблюдений, которые не генерируются трассами событий, встречающимися в композициях конформных реализаций. Используя свойство аддитивности, нетрудно показать, как следует вычислять косую композицию по заданным трассовым моделям спецификаций-операндов. Далее этот алгоритм модифицируется для случая, когда спецификации-операнды заданы в виде LTS наблюдений, результатом тоже становится LTS наблюдений. Если универсумы ***B***, ***O*** и ***E***, а также LTS-операнды конечны, то и результат композиции будет конечной LTS и её можно построить алгоритмически за конечное время.

Слайд 22. Моделирование старых семантик.

К новой ETS-модели также могут быть сведены конформности типа редукции в самых разнообразных семантиках. Вот на слайде показано, как это происходит.

Мы определяем отображение $\alpha`$ ATS-модели в ETS-модель таким образом, что дальнейшее преобразование функцией ***f*** даёт ту же OTS-модель, что и преобразование α, которое мы рассматривали для преобразования ATS-модели непосредственно в OTS-модель.

Все внешние действия из алфавита ATS являются событиями, но добавляются и другие события. Скрываются все синхронные действия и только они.

Самое главное, чего мы добиваемся, – это то, что композиция ETS-моделей, полученных из ATS-моделей с помощью преобразования $\alpha`$, совпадает с результатом преобразования по $\alpha`$ композиции исходных ATS-моделей.

Сейчас я, конечно, не буду рассказывать, как делается преобразование α` для разных семантик. Оно вполне алгоритмическое и, в общем, довольно простое.

Как уже говорилось, на классе всех реализаций для новой модели выполняется свойство аддитивности. Вместе с определенной выше косой композицией спецификаций это решает проблему композиции. Но только на классе всех ETS-реализаций. Если мы ограничиваемся какими-то подклассами ETS-операндов, композиция спецификаций, определённая выше, конечно, останется корректной спецификацией композиционной системы. Однако, вообще говоря, она уже не будет косой композицией, то есть не будет предъявлять к композиционной системе наибольшие требования среди всех корректных спецификаций системы. Иными словами, в новой модели проблема композиции возникает только при сужении классов реализаций операндов.

При моделировании, которое отображено на слайде, нужно учитывать, что классы ATS-реализаций, из которых берутся ATS-операнды композиции, после преобразования α` становятся строгими подклассами класса всех ETS-реализаций. Проблема композиции остаётся, но как частный случай общей проблемы композиции при сужении классов реализаций операндов. Это именно то, что мы хотели продемонстрировать для новой модели касательно композиции.

Слайд 23. Приоритеты.

В новой модели имеется приоритет тестового воздействия над наблюдениями и над дивергенцией. Именно на этом построно моделирования систем с приоритетами в новой модели.

Рассмотрим те же два примера, которые я показывал раньше.

Выход из дивергенции заложен с в новой модели по определению. Теперь реализация может, конечно, двигаться по циклу τ-переходов, но только конечное число раз. Через конечный, хотя и не ограниченный по длине, τ-маршрут обязательно должен выполниться переход по кнопке, определённый в конце этого маршрута так же, как в любом состоянии реализации.

Операцию *cancel* естественно рассматривать как тестовое воздействие – кнопку. Она должна прерывать выполнение цепочки действий a,b,c,d, инициированной предыдущим тестовым воздействием, и запускать терминирующую цепочку действий. Если действия понимать как наблюдения, то кнопка *cancel*, как любая кнопка, блокирует выполнение этих действий.

Если мы нажимаем кнопку *cancel* сразу после наблюдения действия b, то выполнится завершающая цепочка действий –b,–a.

Если мы нажимаем кнопку *cancel* сразу после наблюдения действия a, то выполнится одно завершающее действие –a.

Также и другие случаи приоритетов мы можем теперь моделировать:

- выход из осцилляции (приоритет приёма над выдачей),
- наоборот, приоритет выдачи над приёмом в неограниченных очередях,
- приоритетная обработка входных воздействий,
- и так далее.

Слайд 24. ИТОГИ.

Итак, новая модель полезна нам по следующим причинам.

1. Новая модель удобна для моделирования систем с приоритетами без введения каких-то дополнительных механизмов.
2. В новой модели нет зависимости между ошибками (кроме тривиальной префиксной зависимости) на классе всех реализаций. Поэтому проблема зависимости между ошибками сводится к их появлению при сужении класса тестируемых реализаций.
3. В новой модели трассы событий аддитивны относительно композиции LTS на классе всех реализаций. Поэтому проблема композиции появляется только при сужении классов тестируемых реализаций операндов. Эта проблема композиции формулируется как проблема декомпозиции системных требований, проблема несохранения конформности при композиции и т.п.

4. В новой модели моделируются старые модели LTS действий для самых разных семантик, в которых используются, кроме действий, другие наблюдения: отказы разных видов, множества готовности и т.п. Это делается как для LTS наблюдений, так и для более общей LTS событий. При этом такое моделирования оказывается согласованным как с отношением конформности, так и с композицией.

Это, конечно, не означает, что проблема зависимости между ошибками и проблема композиции решены. Это означает лишь, что теперь мы можем по-другому на них взглянуть. И увидеть причину этих проблем. Причина – в сужении класса тестируемых реализаций.

Слайд 25. ИТОГИ.

И здесь важно отметить, что такое сужение может быть вызвано не только ограничениями той или иной частной семантики взаимодействия. Вот лишь несколько примеров такого сужения, используемого на практике.

1. Класс LTS-реализаций с ограниченным числом состояний.
2. Конечный (с точностью до изоморфизма) класс тестируемых реализаций.
3. Конечный подкласс неконформных реализаций класса тестируемых реализаций. Такой подкласс называют также *классом неисправностей*.
4. Конечное подмножество ошибок такое, что каждая неконформная реализация из класса тестируемых реализаций содержит хотя бы одну ошибку из этого подмножества.

Слайд 26. ИТОГИ.

Что осталось за рамками доклада?

Во-первых, понятие о безопасности тестирования. Безопасность тестирования предназначена для того, чтобы избегать при тестировании некоторых опасных ситуаций. К таким

ситуациям относится, например, бесконечное ожидание наблюдений тогда, когда их нет в реализации. Также мы вводили раньше специальное действие разрушения, которое может быть в реализации и которого следует избегать при тестировании. В общем, проблемы безопасности также сводятся к сужению класса всех тестируемых реализаций до подкласса безопасных реализаций, то есть реализаций, которые можно безопасно тестировать для проверки их конформности заданной спецификации.

Во-вторых, мы не рассматривали конформности типа симуляции. Сейчас мы как раз пишем статью про слабую симуляцию в новой модели. Пока вроде бы всё получается хорошо.