И. Бурдонов, А. Косачев
Тестирование и верификация систем на основе формальных моделей. Лекция 2.
http://old.compsciclub.ru/courses/testingverification/
Уральский федельный университет, Computer Science клуб, Екатеринбург, 6,7,8 апреля 2012
39 слайдов

Тестирование и верификация систем на основе формальных моделей.

3. Гипотеза о безопасности и безопасная конформность

Слайд 3. ВОСПОМИНАНИЯ О ПРОШЛОЙ ЛЕКЦИИ

Рассмотрим ещё раз пример калькулятора. Мы знаем, что можно нажимать кнопки: 2, +, 3, =. На экране дисплея машины тестирования будут наблюдения: 2, +, 3, 5.

Интуитивно мы знаем, как устроен калькулятор. На самом деле, нам это только кажется. Он ведь может быть устроен по-разному. Мы примерно знаем, как он должен работать, если мы работаем правильно. Вот мы правильно нажимаем кнопки и знаем: трассовая история ⟨“2”,2,“+”,+,“3”,3,“=”,5⟩ безопасна: мы её можем пройти при безопасном тестировании. Более того, и ответ 5 правильный. Но что будет происходить, если мы ошиблись в наборе?

Например, вопрос: можно ли нажимать второй раз «+», то есть нажимать кнопки: 2, +, +?

1▶Здесь могут быть два ответа: нельзя, можно.

Рассмотрим сначала первый вариант: второй раз подряд кнопку «+» нажимать **нельзя**.

Почему?

2▶Причина может быть только одна: нажатие этой кнопки опасно. Мы знаем три вида опасности: дивергенция, разрушение и ненаблюдаемый отказ.

3▶Дивергенция. Мы уже говорили, что дивергенция сама по себе не опасна, опасна попытка выхода из неё. Для того, чтобы нажатие второй кнопки “+” был опасным, нужно, чтобы дивергенция возникла до того, как мы эту кнопку нажимаем. Значит, она возникла после нажатия первой кнопки “+”. Но тогда мы не могли бы делать

правильный набор, когда вместо второго “+” мы нажимаем “3”. Кнопка “3” тоже была бы опасной.

4▶Разрушение. Это возможно, но что это значит? Разрушение – это абстракция, моделирующая любое нежелательное поведение. Но причины нежелательности могут быть разными.

5▶Конечно, можно предположить, что от второго нажатия кнопки “+” калькулятор просто сломается. Это такой плохо спроектированный калькулятор, который работает правильно только, если мы правильно нажимаем кнопки. А как только делаем ошибку, он ломается: в нём нет «защиты от дурака».

Другой вид нежелательности: поведение калькулятора неопределено. Ничего не ломается, но результат вычисления непредсказуем, он может быть любым. Чтобы восстановить правильную работу калькулятора, нужно сделать сброс или выключить, а потом включить его.

6▶Чем отличаются друг от друга эти два вида разрушения?

В первом случае (калькулятор ломается) нельзя делать такое тестирования, то есть нельзя нажимать вторую кнопку “+”. А во втором случае (калькулятор не ломается, но поведение произвольное) такое тестирование делать бессмысленно.

Ненаблюдаемый отказ. Это значит, что кнопка “+” – это **Q**-кнопка. Калькулятор может игнорировать второе нажатие кнопки “+”, его состояние при этом не меняется. Может, но не обязан: вместо отказа он мог бы выдать какое-то сообщение, что-то вроде «ошибка в наборе». Но, поскольку отказ ненаблюдаем, сколько бы времени мы не ждали ответа, мы не узнаем, что происходит: то ли возник ненаблюдаемый отказ, то ли будет получен ответ «ошибка в наборе». Мы ведь не знаем точно, как именно устроен калькулятор.

7▶Теперь рассмотрим второй вариант ответа на наш первый вопрос: второй раз подряд кнопку «+» нажимать **можно**.

Но что при этом должно происходить? Раз нажимать кнопку можно, значит через конечное время мы получим наблюдение на экране калькулятора.

8▶Какое это наблюдение?

Если кнопка “+” – это **Q**-кнопка, ответ может быть только один: +. Если кнопка “+” – это **R**-кнопка, может быть два ответа: + и θ.

Если мы наблюдаем θ, то это значит, что возник наблюдаемый отказ. Калькулятор не изменил своего состояния и мы можем спокойно продолжать набор так, как если бы мы не нажимали второй раз кнопку “+”. То есть мы продолжаем набор и нажимаем кнопки “3” и “=”. В результате должны получить правильный ответ 5.

9▶Если же мы видим на экране символ +, значит отказа нет. Но что калькулятор сделал? Интуитивно ясно, что его состояние тоже не должно бы измениться.

10▶Это означает пустое выполнение: в LTS-модели калькулятора переход по второму символу «+» - это переход-петля в состоянии, где он оказывается после первого перехода по символу «+».

11▶А мы уже знаем, что пустое выполнение нужно тестировать также, как всякое другое. Иными словами, мы должны проверять не только «a+b=» с любыми числами «a» и «b», но и «a++b=».

Хочу отметить, что мы рассмотрели ещё не все возможные варианты поведения калькулятора при нажатии второй раз подряд кнопки “+”.

12▶Например, калькулятор может складывать как положительные, так и отрицательные числа, у него есть ещё кнопка “-”. Но мы прибавляем к числу 2 не -3, а +3. По умолчанию отсутствие знака обычно трактуется как знак +. Но почему бы нам явно не указать этот знак. Тогда запись 2++3 то же самое, что запись 2+3. Результат должен быть 5 или +5.

13▶Другой пример: в калькуляторе для операндов и результата есть регистры, то есть переменные. Набирая 2, мы присваиваем значение 2 регистру первого операнда. А дальше калькулятор может понимать набор кнопок, который мы выполняем, как выражения языка Си. Тогда запись 2++3 означает: присвоить число 2 регистру первого операнда, выполнить операцию увеличения регистра первого операнда на 1, а дальше – непонятно. То есть поведение калькулятора может быть самым разным не на втором плюсе, а на числе 3 после двух плюсов. Здесь, наверное, можно рассмотреть те же варианты, которые у нас были раньше для второго плюса.

Общий вывод такой: интуитивное понимание поведения системы – это обычно понимание того, как система должна себя вести, если мы правильно с ней взаимодействуем. Это эквивалентно

использованию конструкций use-case в языке моделирования UML. Вообще-то на языке UML можно описать и полное поведение, то есть при неправильном поведении окружения (в случае калькулятора – при неправильном наборе кнопок). Но конструкции use-case используются для описания только правильного поведения окружения.

Но, как мы видим, полная спецификация системы должна описывать поведение системы не только при правильном, но также и при неправильном поведении окружения. Замечу только, что такое описание может быть и очень коротким: поведение системы неопределено или опасно. То есть с помощью разрушения. В то же время системы общего пользования, к которым, между прочим, относится и калькулятор, не должны иметь предусловий. В них должна быть предусмотрена «защита от дурака». Иными словами, поведение окружения может быть любым, а в спецификации нужно написать, как именно будет реагировать на это система.

Слайд 4. ВОСПОМИНАНИЯ О ПРОШЛОЙ ЛЕКЦИИ

Напомню о моделях, которые мы рассматривали на прошлой лекции.

Основная модель – это LTS. Переходы в ней помечены внешними действиями из алфавита **L**, а также символом тау – внутренне действие, и гамма – разрушение.

Для построения R-трасс LTS, то есть трасс, содержащих R-отказы мы проводим:

1►- виртуальные петли R-отказов в стабильных состояниях LTS (там, где нет тау и гамма);

2►- переходы по разрушению перенаправляем в терминальные состояния;

3►- в дивергентных состояниях (где начинаются бесконечные тау-маршруты) проводим переходы по дельте тоже в терминальные состояния.

4►Для спецификации мы выбираем как раз трассовую R-модель, то есть множество R-трасс некоторой LTS. Вот здесь показаны все R-трассы этой LTS. Эти трассы записаны в виде регулярного выражения, также берутся все их префиксы.

5▶Для реализации важны также Q-отказы. Вот здесь в LTS в двух состояниях есть отказ {a}.

6▶В качестве реализации мы берём R∪Q-модель. Вот эти трассы.

7▶Рассмотрим, какие из этих трасс безопасны.

Трассы в первых двух строках опасны.

В первой строке трассы, заканчивающиеся или продолжающиеся гаммой, опасны по определению. Префикс до действия d – это трассы, которые являются также префиксами трасс в третьей строке.

Во второй строке трассы, заканчивающиеся дельтой, опасны по определению. Их префиксы непосредственно перед дивергенцией, заканчивающиеся вторым действием a, тоже опасны. Почему? Потому второе a разрешается только кнопкой {a}, а в реализации эта кнопка опасна после трасс, заканчивающихся первым a: есть ненаблюдаемый отказ {a}. Префиксы, заканчивающиеся первым a являются также префиксами трасс в третьей строке.

В третьей строке множество трасс бесконечно за счёт произвольного числа повторений как отказов, так и цикла из двух действий c и b. Опасны только те трассы, которые содержат ненаблюдаемый отказ {a}.

Слайд 5. 3. Гипотеза о безопасности и безопасная конформность.

Предварительные рассуждения

Сначала порассуждаем о том, как проходит безопасное тестирование.

1▶Вот мы при тестировании проходим некоторую трассу в реализации.

2▶Поскольку тестирование безопасно, эта трасса должна быть безопасной в реализации, то есть её можно пройти, нажимая такие кнопки, что не возникают три опасности, о которых мы говорили. Такие кнопки – безопасные в реализации после соответствующих трасс.

3▶Если мы ещё не обнаружили ошибку, то такая же трасса должна быть и в спецификации.

4▶После этой трассы мы можем нажать очередную кнопку, которая должна быть безопасна в реализации после этой трассы.

5▶Поскольку кнопка безопасна, через конечное время мы получаем наблюдение.

6▶Тогда мы смотрим, а какие наблюдения есть в спецификации после той же самой трассы и разрешаемые той же самой кнопкой.

7▶Если наблюдение, полученное от реализации, одно из наблюдений в спецификации, то всё нормально. Спецификация как раз и описывает допустимые наблюдения.

8▶Если это не так, тест фиксирует ошибку.

9▶А теперь вопрос: откуда мы знаем, что кнопка безопасна в реализации после этой трассы? Ведь реализация нам неизвестна.

10▶ А мы этого и не знаем, но предполагаем, опираясь на спецификацию. Кроме спецификации, мы ничего не имеем, а она предназначена не только для того, чтобы говорить, какое поведение правльное, а какое нет, при безопасном тестировании, но и описывает требования к реализации по безопасности. Конкретно: именно спецификация говорит нам, должна кнопка быть в реализации безопасной после трассы или не обязательно. Если спецификация говорит, что кнопка должна быть безопасной, то такую кнопку будем называть безопасной в спецификации после этой трассы.

11▶Только такие безопасные в спецификации кнопки мы и будем нажимать при безопасном тестировании.

12▶Безопасная трасса спецификации – это трасса, в которой каждое наблюдение разрешается некоторой кнопкой, безопасной после префикса трассы, непосредственно предшествующего этому наблюдению.

13▶Гипотеза о безопасности как раз и говорит: если кнопка безопасна в спецификации после трассы, которая имеется и безопасна в спецификации и в реализации, то эта кнопка должна быть безопасна в реализации после этой трассы.

Тем самым, гипотеза о безопасности выделяет класс реализаций, которые можно безопасно тестировать на конформность данной спецификации.

Заметим, что если кнопка опасная, то есть не безопасная, в спецификации после безопасной трассы, то это не означает, что она

должна быть опасной после этой трассы в реализации. Там она может быть как опасной, так и безопасной. Но такие кнопки мы не нажимаем, потому что мы не знаем, какая именно реализация из класса безопасно-тестируемых реализаций находится в машине тестирования. Та, в которой эта кнопка безопасна, или та, в которой эта кнопка опасна.

Слайд 6. 3. Гипотеза о безопасности и безопасная конформность.

Предварительные рассуждения

Если бы мы нажимали все кнопки подряд, не обращая внимания на их безопасность, то мы бы тем самым проверяли, что все трассы реализации являются трассами спецификации. Такая конформность, означающая простую вложенность трасс, называется редукцией.

Но мы обращаем внимание на безопасность и нажимаем после безопасной трассы только те кнопки, которые безопасны в спецификации после этой трассы. Если же кнопка опасна в спецификации после трассы, то в реализации она может быть как опасна, так и безопасна. В любом случае мы не проверяем, какие продолжения имеет эта трасса по наблюдениям, которые разрешаются только такими кнопками, которые опасны после трассы в спецификации. Естественно, что какие-то из таких продолжений могут в спецификации отсутствовать. Так получается конформность типа квази-редукции. Она означает, что после общей безопасной трассы реализации и спецификации множество наблюдений в реализации, разрешаемое кнопкой, которая безопасна в спецификации после этой трассы, вложено во множество наблюдений в спецификации, разрешаемое этой же кнопкой.

Слайд 7. 3. Гипотеза о безопасности и безопасная конформность.

Пример отношения ***ioco***

Теперь мы подошли к тому, чтобы определить гипотезу о безопасности и безопасную конформность.

Сначала рассмотрим в качестве примера отношение ***ioco***.

Я напомню, что в семантике этого отношения алфавит внешних действий делится на стимулы – множество X, и реакции – множество Y. Для каждого стимула имеется единственная разрешающая его

кнопка, которая разрешает только этот стимул. Это Q-кнопка, соответствующий отказ – он называется блокировкой стимула – ненаблюдаем. Все реакции разрешаются одной кнопкой, это R-кнопка, соответствующий отказ – он обозначает символом дельта маленькая – наблюдаемый.

В первоначальной семантике отношения ***ioco***, которую предложил автор этого отношения Ян Тритманс, предполагается, прежде всего, отсутствие дивергенции и разрушения как в спецификации, так и в реализации. Поэтому опасность могут представлять собой только ненаблюдаемые отказы – блокировки стимулов.

Гипотеза о безопасности описывает класс реализаций, которые можно безопасно тестировать для проверки их конформности заданной спецификации. Такая гипотеза для отношения ***ioco*** предполагает, что реализация всюду определена по стимулам – input-enabled, то есть в ней нет блокировок стимулов.

Конформность означает, что после общей трассы спецификации и реализации любая реакция, которая есть в реализации, должна быть и в спецификации. А также: если в реализации есть отказ «отсутствие реакций», то он есть и в спецификации после той же трассы.

Отношение ***ioco*** – это частный случай конформности типа *редукции*. Что это значит? Это значит, что реализация «сводится» к спецификации в том смысле, что в реализации не должно быть ничего, чего нет в спецификации. Однако заметим, что это относится только к реакциям и отказу «отсутствие реакций», но не относится к стимулам. Более того, если реализация всюду определена по стимулам, то спецификация не обязана быть такой.

Из определения конформности ***ioco*** следует, что при тестировании после трассы выполняется передача стимула только в том случае, когда такой стимул есть после трассы в спецификации.

Теперь я сформулирую для ***ioco*** отношение безопасности кнопок: «кнопка безопасна после трассы».

Поскольку дивергенции и разрушения нет, а кнопка приёма всех реакций – это R-кнопка, она всегда безопасна.

Рассмотрим передачу стимула. Это Q-кнопка.

1▶В реализации передача стимула безопасна после трассы, если после этой трассы в реализации нет блокировки стимула.

2▶В спецификации нам нужно, чтобы передача стимула была безопасна после трассы, если мы собираемся передавать стимул после этой трассы. Тем самым, можно считать, что передача стимула безопасна в спецификации после трассы, если трасса продолжается этим стимулом. И здесь уже не важно, продолжается эта трасса в спецификации также и блокировкой стимула или нет.

На основе безопасности кнопок определяется безопасность наблюдений. Стимул безопасен, если кнопка передачи этого стимула безопасна. Реакция и отказ «отсутствие реакций» всегда безопасны, поскольку всегда безопасна разрешающая эти наблюдения кнопка «принять реакции».

3▶На основе безопасности наблюдений определяется безопасность трасс. Трасса безопасна, если после каждого её строгого префикса, то есть префикса, не совпадающего со всей трассой, следующее в трассе наблюдение безопасно. Так мы получаем множество безопасных по ***safe in*** трасс реализации и множество безопасных по ***safe by*** трасс спецификации. Для спецификации – это все её трассы. Это следствие того, что мы предположили отсутствие в спецификации дивергенции и разрушения. В реализации могут быть небезопасные трассы, если в ней встречаются ненаблюдаемые отказы – блокировки стимулов. Но если реализация всюду определёна по стимулам, все ей трассы безопасны.

На основе этих понятий о безопасности кнопок, наблюдений и трасс мы можем сказать, что требование всюду определённости по стимулам для ***ioco*** излишне. Почему? Потому что, если в спецификации трасса не продолжается стимулом, то при тестировании стимул в реализацию не посылается. А тогда не важно, есть в реализации после этой трассы блокировка стимула или нет. В одной реализации такая блокировка может быть, а другая всюду определена по стимулам. Эти две реализации неразличимы при тестировании, но первая заведомо неконформна, поскольку даже не входит в домен отношения ***ioco***, а вторая может быть конформной.

4▶Поэтому естественно ослабить требование всюду определённости по стимулам для реализации. Нам ведь важно, чтобы не возникала блокировка стимула при тестировании этой реализации

для проверки её конформности данной спецификации. Иными словами, в реализации не должно быть блокировок стимула после трассы только в том случае, когда трасса в спецификации продолжается этим стимулом.

Это означает следующую ослабленную гипотезу о безопасности: если после трассы спецификации стимул безопасен, то он должен быть безопасен после этой трассы и в реализации. Разумеется, если в реализации вообще есть эта трасса. В качестве таких трасс достаточно рассматривать безопасные по ***safe by*** трассы спецификации. Для отношения ***ioco*** они совпадают с множеством всех трасс спецификации, но в общем случае это будет не так.

5▶Соответственно меняется и определение отношения ***ioco*.**

Слайд 8. 3. Гипотеза о безопасности и безопасная конформность.

Отношение безопасности кнопок

Вот теперь перейдём к общему случаю произвольной R/Q-семантики.

Как возможно безопасное тестирование, если реализация неизвестна? Например, если в LTS-реализации переход по разрушению определен в начальном состоянии, то такую реализацию не только нельзя тестировать, но даже запускать на выполнение, поскольку она может разрушиться до первого тестового воздействия, то есть до первого нажатия кнопки.

Выход в том, чтобы ограничиться теми реализациями, которые можно безопасно тестировать для проверки конформности заданной спецификации.

Это ограничение формулируется как гипотеза о безопасности. По сути, она предполагает, что реализация относится к классу тех реализаций, которые можно безопасно тестировать для проверки конформности заданной спецификации.

На примере отношения ***ioco*** мы уже видели, как может формулироваться такая гипотеза. Теперь сделаем это в общем случае.

В силу эквивалентности трассовой и LTS-моделей нам достаточно определить гипотезу о безопасности и конформность только для трассовых моделей реализации и спецификации.

Безопасное тестирование, прежде всего, предполагает формальное определение на уровне модели отношения безопасности «кнопка безопасна в модели после трассы». При безопасном тестировании будут нажиматься только безопасные кнопки. Это отношение различно для реализационной и спецификационной моделей. Мы уже видели это на примере отношения ***ioco***, а теперь рассмотрим в общем случае.

1►В реализации отношение безопасности называется ***safe in***. Что оно означает? Прежде всего, кнопка должна быть *неразрушающей* после трассы. Это означает, что ее нажатие не может означать попытку выхода из дивергенции (трасса не продолжается дивергенцией) и не может вызывать разрушение (после действия, разрешаемого кнопкой). Такое отношение «неразрушаемости кнопки после трассы» называется $\boldsymbol{safe}_{\gamma\Delta}$. Кнопка, безопасная по ***safe in***, должна быть, во-первых, неразрушающей и, во-вторых, нажатие кнопки не должно приводить к ненаблюдаемому отказу, если это ***Q***-кнопка: $\forall P \in \boldsymbol{R} \cup \boldsymbol{Q}\ \forall \sigma \in \boldsymbol{I}$

$$P\ \boldsymbol{safe}_{\gamma\Delta}\ \boldsymbol{I}\ \boldsymbol{after}\ \sigma \quad =_{\mathbf{def}} \sigma\cdot\langle\Delta\rangle \notin \boldsymbol{I}\ \&\ \forall u \in P\ \sigma\cdot\langle u,\gamma\rangle \notin \boldsymbol{I}.$$

$$P\ \boldsymbol{safe\ in}\ \boldsymbol{I}\ \boldsymbol{after}\ \sigma \quad =_{\mathbf{def}} P\ \boldsymbol{safe}_{\gamma\Delta}\ \boldsymbol{I}\ \boldsymbol{after}\ \sigma\ \&\ (P \in \boldsymbol{Q} \Rightarrow \sigma\cdot\langle P\rangle \notin \boldsymbol{I}).$$

2►В спецификации отношение безопасности называется ***safe by*** и должно удовлетворять трём правилам, которые написаны на слайде. Для **R**-кнопок оно точно такое же, как для ***safe in*** – правило 1. Отличие только для ***Q***-кнопок: мы не требуем, чтобы после трассы σ не было ***Q***-отказа Q, но требуем, чтобы было хотя бы одно действие $z \in Q$. Это правило 3. Кроме того, если действие разрешается хотя бы одной неразрушающей кнопкой, то оно должно разрешаться какой-нибудь безопасной кнопкой. Это правило 2. Оно позволяет использовать спецификацию «по-максимуму». Если действие разрешается неразрушающей ***R***-кнопкой, то она же и безопасна. Но если все неразрушающие кнопки, разрешающие действие, являются ***Q***-кнопками, то хотя бы одна из них должна быть объявлена безопасной.

Такое отношение безопасности всегда существует: достаточно объявить безопасной каждую неразрушающую кнопку, разрешающую действие, продолжающее трассу. Так это и делается для отношения ***ioco***. Более того, для ***ioco*** и выбора-то нет, какие кнопки объявлять безопасными, поскольку каждое действие

разрешается только одной кнопкой. Однако в целом указанные требования неоднозначно определяют отношение ***safe by***, и при задании спецификации, кроме её модели, дополнительно указывается конкретное отношение ***safe by***.

Слайд 9. 3. Гипотеза о безопасности и безопасная конформность.

Безопасные наблюдения и трассы

Безопасность кнопок определяет безопасность наблюдений. ***R***-отказ *R* безопасен, если после трассы безопасна кнопка *R*. Действие *z* безопасно, если оно разрешается некоторой кнопкой, безопасной после трассы:

z ***safe in I after*** $\sigma =_{\mathbf{def}} \exists P \in \boldsymbol{R} \cup \boldsymbol{Q}\ z \in P$ & P ***safe in I after*** σ.

z ***safe by S after*** $\sigma =_{\mathbf{def}} \exists P \in \boldsymbol{R} \cup \boldsymbol{Q}\ z \in P$ & P ***safe by S after*** σ.

1▶Теперь мы можем определить *безопасные трассы*. Трасса σ безопасна, если эта трасса есть в модели и 1) модель не разрушается с самого начала (сразу после включения машины ещё до нажатия первой кнопки), то есть, в ней нет трассы $<\gamma>$, 2) каждый символ трассы безопасен после непосредственно предшествующего ему префикса трассы:

$<\gamma> \notin \boldsymbol{I}$ & $\forall \mu\ \forall u$ ($\mu \cdot <u>$ префикс $\sigma \Rightarrow u$ ***safe in I after*** μ),

$<\gamma> \notin \boldsymbol{S}$ & $\forall \mu\ \forall u$ ($\mu \cdot <u>$ префикс $\sigma \Rightarrow u$ ***safe by S after*** μ).

2▶Наблюдение опасно после трассы, если оно не является безопасным после трассы.

3▶Трасса (не обязательно принадлежащая спецификационной модели) опасна, если в спецификации нет безопасных трасс (есть трасса $<\gamma>$), или после максимального безопасного префикса трассы следующее наблюдение опасно.

4▶Множества безопасных трасс реализации ***I*** и спецификации ***S*** обозначим ***SafeIn***(***I***) и ***SafeBy***(***S***), соответственно.

Аналогично можно определить множество неразрушающих трасс модели, опираясь на отношение $\boldsymbol{safe}_{\gamma\Delta}$.

В реализации безопасные трассы неразрушающие, но могут быть неразрушающие трассы, которые опасны по ***safe in*** из-за **Q**-отказов.

В то же время в спецификации неразрушающие трассы и трассы, безопасные по ***safe by***, – это одно и то же. Иными словами, мы определили такие требования к ***safe by***, чтобы использовать спецификацию по-максимуму. Коли уж трасса продолжается в спецификационной модели неразрушающим наблюдением, то это наблюдение должно быть безопасным.

Из определения отношения безопасности ***safe by*** видно, что множество безопасных трасс спецификации однозначно определяется множеством ее ***R***-трасс. Из определения отношения безопасности ***safe in*** видно, что множество безопасных трасс реализации, кроме множества ее ***R***-трасс, дополнительно зависит от продолжения ***R***-трасс ***Q***-отказами.

Слайд 10. 3. Гипотеза о безопасности и безопасная конформность.

Требование безопасности тестирования выделяет класс *безопасно-тестируемых* реализаций ***SafeImp***, то есть таких, которые могут быть безопасно протестированы для проверки их конформности заданной спецификации ***S*** с заданным отношением ***safe by*** в заданной ***R***/***Q***-семантике.

Этот класс определяется следующей *гипотезой о безопасности*: реализация ***I*** *безопасно-тестируема* для спецификации ***S***, если 1) в реализации нет разрушения с самого начала, если этого нет в спецификации, 2) после общей безопасной трассы спецификации и реализации любая кнопка, безопасная в спецификации, безопасна после этой трассы в реализации:

$$\boldsymbol{I}\ \textbf{safe for}\ \boldsymbol{S} =_{\textbf{def}} (\langle\gamma\rangle \notin \boldsymbol{S} \Rightarrow \langle\gamma\rangle \notin \boldsymbol{I})\ \&\ \forall \sigma \in \boldsymbol{SafeBy}(\boldsymbol{S}) \cap \boldsymbol{I}\ \forall P \in \boldsymbol{R} \cup \boldsymbol{Q}\ (P\ \textbf{safe by}\ \boldsymbol{S}\ \textbf{after}\ \sigma \Rightarrow P\ \textbf{safe in}\ \boldsymbol{I}\ \textbf{after}\ \sigma).$$

1►После этого можно определить отношение (безопасной) *конформности*: реализация ***I*** *безопасно конформна* (или просто *конформна*) спецификации ***S***, если она безопасно-тестируема и выполнено

2►*тестируемое условие*: любое наблюдение, возможное в реализации в ответ на нажатие безопасной (в спецификации) кнопки, разрешается спецификацией:

I saco S $=_{\mathbf{def}}$ ***I safe for S*** & $\forall\sigma\in$***SafeBy***$(\boldsymbol{S})\cap$ ***I*** $\forall P$ ***safe by S after*** σ ***obs***$(\sigma,P,\boldsymbol{I})\subseteq$***obs***$(\sigma,P,\boldsymbol{S})$,
где ***obs***$(\sigma,P,\boldsymbol{M}) =_{\mathbf{def}} \{u|\sigma\cdot\langle u\rangle\in\boldsymbol{M}$ & $(u\in P \vee u=P$ & $P\in\boldsymbol{R})\}$ – множество наблюдений, которые можно получить над полной трассовой моделью ***M*** при нажатии кнопки *P* после трассы σ.

Гипотеза о безопасности и проверяемое условие определяют конформность ***saco*** как отношение типа редукции: 1) в реализации не может быть опасности, если её нет в спецификации в той же самой ситуации, то есть после той же самой трассы; 2) в реализации не может быть безопасного наблюдения, которого не было в спецификации в той же самой ситуации.

Следует отметить, что гипотеза о безопасности не проверяема при тестировании и является его предусловием; тестирование проверяет тестируемое условие конформности.

Отношение ***saco*** определяет класс конформных реализаций ***ConfImp***. Эта пара классов реализаций ***SafeImp*** и ***ConfImp*** как раз и описывает все спецификационные требования: какие реализации тестируются и какие из них считаются конформными. Какой бы способ представления спецификационных требований мы ни выбрали, в конечном счёте они сводятся к определению этой пары классов реализаций. Выбрав представление в виде трассовой или LTS-модели и отношения ***safe by***, мы, тем самым, определили возможные пары классов реализаций для ***saco*** – это уже будут не все возможные пары.

4. Генерация тестов

Слайд 11. 4. Генерация тестов.

Трассовый тест

В терминах машины тестирования тест – это инструкция оператору машины. В каждом пункте инструкции указывается кнопка, которую оператор должен нажимать, и для каждого наблюдения – пункт инструкции, который должен выполняться

следующим, или вердикт (***pass*** или ***fail***), если тестирование нужно закончить. В тесте после кнопки P допускается только такое наблюдение u, которое разрешается кнопкой P, то есть $u \in P \vee u=P \in \boldsymbol{R}$.

1►В трассовой теории тест можно понимать как префикс-замкнутое множество конечных историй, в котором 1) каждая максимальная история заканчивается наблюдением, и ей приписан вердикт; 2) каждая немаксимальная история, заканчивающаяся кнопкой, может продолжаться во множестве только теми наблюдениями, которые разрешаются этой кнопкой, и обязательно продолжается теми наблюдениями, которые могут встречаться в безопасно-тестируемых реализациях.

2►Тест безопасен тогда и только тогда, когда в каждой его истории каждая кнопка безопасна в спецификации после подтрассы непосредственно предшествующего этой кнопке префикса истории. Иными словами, тест безопасен тогда и только тогда, когда подтрассы всех его историй являются тестовыми, где *тестовая трасса* – это безопасная трасса или безопасная трасса, продолженная безопасным после неё наблюдением, но не обязательно имеющимся в спецификации после этой трассы.

3►Такой тест мы будем называть трассовым, поскольку он представляет собой множество историй, то есть трасс в объединённом алфавите внешних действий и кнопок.

Слайд 12. 4. Генерация тестов.

Полный набор тестов

Реализация *проходит* тест, если её тестирование с помощью этого теста всегда заканчивается с вердиктом ***pass***.

Для *трассового* теста это означает следующее: если какая-то трасса реализации является подтрассой некоторой максимальной истории теста, то этой истории приписан вердикт ***pass***.

Реализация проходит набор тестов, если она проходит каждый тест из набора.

Набор тестов *значимый (sound)*, если каждая конформная реализация его проходит;

исчерпывающий (exhaustive), если каждая неконформная реализация его не проходит;

полный (complete), если он значимый и исчерпывающий.

Для проверки конформности любой безопасно-тестируемой реализации ставится задача генерации полного набора тестов по спецификации.

Слайд 13. 4. Генерация тестов.

Примитивный тест

Полный набор тестов всегда существует, в частности, им является набор всех *примитивных* тестов. Доказательство этого утверждения тоже можно найти в моей диссертации по адресу внизу слайд:

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

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

Примитивный тест строится по одной выделенной немаксимальной безопасной трассе спецификации, то есть безопасной трассе, у которой есть безопасное продолжение.

Вот пример такой трассы: отказ **A,** действие **b**. Я буду большими буквами обозначать отказы и кнопки, а маленькими – действия.

1►Для получения теста сначала в трассу вставляются кнопки, которые оператор должен нажимать. Для наглядности они здесь заключены в кавычки. Перед каждым отказом **A** вставляется кнопка **A**, перед каждым действием **b** – какая-нибудь безопасная (после соответствующего префикса трассы) кнопка **B** большое, разрешающая действие **b**, а после всей трассы вставляется любая безопасная после нее кнопка **C**.

Безопасность трассы гарантирует, что мы можем всегда подобрать такие безопасные кнопки **A** большое и **B** большое. А немаксимальность трассы среди всех безопасных трасс гарантирует наличие последней кнопки **C** большое.

Получается безопасная тестовая история.

2▶Вот здесь она нарисована. Кнопки заключены в кружки, а наблюдения нарисованы на стрелках.

Потом добавляются все ответвления по всем возможным наблюдениям и расставляются вердикты. Вот как это делается на примере.

3▶Если кнопка **A** большое разрешает действие **a** хорошее, которое имеется в спецификации, то добавляется ответвление по этому действию с вердиктом ***pass***.

4▶Если кнопка **A** большое разрешает действие **a** плохое, которое отсутствует в спецификации, то добавляется ответвление по этому действию с вердиктом ***fail***.

Далее смотрим, какие наблюдения после трассы **A** большое разрешает следующая кнопка **B** большое.

5▶Аналогично, если кнопка **B** большое разрешает действие **b** хорошее, которое имеется в спецификации, то после трассы **A** большое добавляется ответвление по этому действию с вердиктом ***pass***.

6▶Если кнопка **B** большое разрешает действие **b** плохое, которое отсутствует в спецификации, то после трассы **A** большое добавляется ответвление по этому действию с вердиктом ***fail***.

Если кнопка **B** большое – это **Q**-кнопка, то это все наблюдения по этой кнопке. Если же кнопка **B** большое – это **R**-кнопка, то есть ещё одно наблюдение – отказ **B** большое.

7▶Если отказ **B** большое имеется в спецификации, то после трассы **A** большое добавляется ответвление по нему с вердиктом ***pass***.

8▶Если отказ **B** большое отсутствует в спецификации, то после трассы **A** большое добавляется ответвление по нему с вердиктом ***fail***.

Далее аналогично смотрим, какие наблюдения после трассы **A** большое **b** маленькое разрешает следующая кнопка **C** большое.

9▶Если кнопка **C** большое разрешает действие **c** хорошее, которое имеется в спецификации, то после трассы **A** большое **b** маленькое добавляется ответвление по этому действию с вердиктом ***pass***.

10▶Если кнопка **C** большое разрешает действие **c** плохое, которое отсутствует в спецификации, то после трассы **A** большое **b** маленькое добавляется ответвление по этому действию с вердиктом ***fail***.

Если кнопка **C** большое – это **Q**-кнопка, то это все наблюдения по этой кнопке. Если же кнопка **C** большое – это **R**-кнопка, то есть ещё одно наблюдение – отказ **C** большое.

11▶Если отказ **C** большое имеется в спецификации, то после трассы **A** большое **b** маленькое добавляется ответвление по нему с вердиктом ***pass***.

12▶Если отказ **B** большое отсутствует в спецификации, то после трассы **A** большое **b** маленькое добавляется ответвление по нему с вердиктом ***fail***.

13▶В результате у нас получается вот такое множество историй с вердиктами на максимальных историях. Здесь бледным цветом показаны альтернативы: отказ имеется или отсутствует в спецификации после соответствующего префикса трассы.

Слайд 14. 4. Генерация тестов.

Примитивный тест

По одной немаксимальной безопасной трассе спецификации можно сгенерировать, вообще говоря, несколько разных примитивных тестов, выбирая разные кнопки. Однако множества тестов, сгенерированных по разным трассам, не пересекаются.

Ещё раз скажу, какой принцип назначения вердиктов.

1▶Вердикт назначается после наблюдения, которое получено после нажатия последней кнопки, и после наблюдения, «ответвляющегося» от трассы. При этом вердикт ***pass*** выносится, если полученная трасса, заканчивающаяся этим наблюдением, имеется в спецификации, а вердикт ***fail*** – если отсутствует.

2▶Такие вердикты соответствуют *строгим* тестам, которые, во-первых, значимые (не фиксируют ложных ошибок) и, во-вторых, не пропускают обнаруженных ошибок.

Любой тест – это множество историй. Строгий тест равен объединению некоторого множества примитивных тестов, то есть

этот строгий тест и множество соответствующих ему примитивных тестов обнаруживают те же самые ошибки. Поэтому в теории можно ограничиться рассмотрением только примитивных тестов.

Слайд 15. 4. Генерация тестов.

Глобальное тестирование

Теперь зададимся таким вопросом: при каких условиях мы можем алгоритмически генерировать тесты полного набора тестов и пропускать их на машине тестирования, гарантируя полное тестирование?

1▶Чего мы хотим? Прежде всего, мы хотим, чтобы любая ошибка обнаруживалась за конечное время. Если реализация неконформна, то через конечное время на одном из прогонов одного из тестов полного набора будет вынесен вердикт ***fail***.

Проблема останется только с конформными реализациями. Если на классе тестируемых реализаций имеется бесконечное число ошибок, то полная проверка конформности может потребовать бесконечного времени. Но это уже проблема практического тестирования, о которой пойдёт речь позднее.

2▶Для алгоритмизации тестирования нам нужно, прежде всего, генерировать пусть бесконечный, но перечислимый полный набор тестов.

3▶Поскольку примитивный тест генерируется по одной конечной немаксимальной безопасной трассе спецификации, нам нужна перечислимость множества безопасных трасс спецификации.

Для этого нам нужно уметь перечислять кнопки, безопасные после трассы, и наблюдения, разрешаемые этими кнопками. Для этого требуется перечислимость множества всех кнопок и конечное время на проверку безопасности кнопки. Мы будем перечислять все кнопки и проверять безопасность каждой кнопки.

4▶Безопасная кнопка – это, прежде всего, неразрушающая кнопка. Как определить, что кнопка неразрушающая? Если безопасная трасса продолжается дивергенцией, то все кнопки разрушающие. Как определить, продолжается трасса дивергенцией или нет?

Слайд 16. 4. Генерация тестов.

Глобальное тестирование

Рассмотрим LTS-спецификацию. Она должна быть задана алгоритмическим способом. Для этого должен существовать алгоритм, который по достижимому состоянию перечисляет переходы, ведущие из него.

1▶Как определить в LTS, продолжается трасса дивергенцией или нет? Для этого нужно перебрать все состояния LTS, в которых заканчивается эта трасса и проверить, что каждое из них конвергентно.

2▶Поскольку нам это нужно сделать за конечное время, число состояний после трассы должно быть конечным. А для этого необходимо и достаточно, чтобы были выполнены два условия.

1) LTS должна быть конечно-ветвящейся: в каждом достижимом состоянии должно быть определено конечное число переходов.
2) LTS должна быть тау-ограниченной: из каждого достижимого состояния по тау-переходам должно быть достижимо конечное число состояний.

3▶В тау-ограниченной LTS дивергенция может быть реализована только в виде цикла тау-переходов на конечном множестве состояний. Иными словами, не может быть бесконечного тау-маршрута, проходящего через бесконечное число состояний.

Слайд 17. 4. Генерация тестов.

Глобальное тестирование

Если оба условия выполнены, то для проверки дивергенции после трассы нужно узнать, имеется ли на конечном множестве состояний после трассы тау-цикл или нет. Это можно сделать за конечное время.

1▶Дальше, для проверки неразрушаемости кнопки нам нужно проверить, что она не разрешает действия, после которого в спецификации есть разрушение. При выполнении наших двух условий мы можем за конечное время составить конечный список действий, по которым есть переходы из состояний в конце трассы.

Также за конечное время мы можем узнать для каждого из таких переходов, достижимо после него разрушение или нет.

2►В этот момент мы, фактически, получаем алгоритм, задающий конечно-ветвящуюся трассовую модель спецификации. Этот алгоритм для каждой трассы перечисляет конечное множество символов, продолжающих её: действия, отказы, дивергенцию и разрушение.

Слайд 18. 4. Генерация тестов.

Глобальное тестирование

Для проверки неразрушаемости кнопки нам нужно уметь за конечное время определять, разрешает данная кнопка данное действие или нет. Это эквивалентно разрешимости кнопочного множества относительно алфавита внешних действий. Будем считать, что это так и задан алгоритм, который для каждого внешнего действия за конечное время определяет, принадлежит оно заданному кнопочному множеству или нет. Это будет 3-е условие.

1►Тогда проверка неразрушаемости кнопки после трассы выполняется за конечное время. Пусть кнопка неразрушающая.

Слайд 19. 4. Генерация тестов.

Глобальное тестирование

Пусть кнопка неразрушающая. Будет ли она безопасной по ***safe by***?

1►Если это **R**-кнопка, то она же будет безопасной. Это первое правило ***safe by***.

2►Если это **Q**-кнопка, то нужно проверить, продолжается ли трасса каким-нибудь действием из кнопки. Это третье правило ***safe by***. Применяем алгоритм разрешения кнопки к каждому действию, которым трасса продолжается. За конечное время мы это определим.

3►Наконец, опасность или безопасность оставшихся кнопок определяется отношением ***safe by*** с учётом выполнения 2-го правила ***safe by***. Будем считать, что отношение ***safe by*** задано алгоритмически: имеется алгоритм, который для каждой такой **Q**-кнопки и каждой

безопасной трассы определяет, безопасна эта кнопка после этой трассы или нет.

4►Например, отношение ***safe by*** для отношения ***ioco*** задано алгоритмически. Для определения безопасности посылки стимула нужно проверить, продолжается трасса этим стимулом или нет.

Слайд 20. 4. Генерация тестов.

Глобальное тестирование

Перечисление кнопок с фильтрацией по их безопасности – это, фактчиески, перечисление кнопок, безопасные после трассы.

Для каждой безопасной кнопки проверяем для каждого действия, которым продолжается трасса, принадлежит ли оно кнопке.

Если принадлежит, то действие безопасно после трассы. Его можно вычеркнуть из списка проверяемых действий.

Если не принадлежит, оно остаётся в списке проверяемых действий.

Для проверки безопасности отказа нужно просто проверить соответствующую кнопку.

1►Получив безопасное наблюдение u, начинаем параллельно исследовать трассу $\sigma\cdot\langle u\rangle$. И, поскольку σ не максимальная, генерируем по ней тест.

Для назначения вердикта достаточно алгоритма, который определяет, имеется ли полученное от реализации наблюдение в спецификации или нет. Поскольку в спецификации таких наблюдений после трассы конечное число, такой алгоритм очевиден.

Слайд 21. 4. Генерация тестов.

Глобальное тестирование

Как только тест сгенерирован и вердикты назначены, мы инициируем последовательность его прогонов на машине тестирования.

Таким образом, генерация тестов и их прогон – это единый процесс. Полнота тестирования для полного набора тестов

гарантируется гипотезой о глобальном тестировании. Но для этого каждый тест мы должны прогонять, вообще говоря, бесконечное число раз. Это делается с помощью «диагонального» процесса генерации тестов и их прогона.

1►Вот здесь показано, как идёт этот диагональный процесс.

Поскольку все тесты сгенерированы по конечным трассам, каждый из них заканчивается через конечное время, после чего выполняется рестарт системы, и прогоняется следующий тест или повторно один из уже перечисленных тестов.

Итак, если реализация неконформна, ошибка будет найдена через конечное время. Если реализация конформна, процесс генерации тестов и их прогона будет бесконечным, всё время будет выдаваться вердикт ***pass*.**

Слайд 22. 4. Генерация тестов.

Глобальное тестирование

Повторим четыре условия, при которых может алгоритмически выполняться генерация тестов и их прогон на машине тестирования.

1►LTS конечно-ветвящаяся: в каждом достижимом состоянии определено конечное число переходов. Задан алгоритм, перечисляющий переходы из заданного состояния.

2►LTS τ-ограниченная: из каждого достижимого состояния по τ- переходам достижимо конечное число состояний.

3►Каждое кнопочное множество разрешимо относительно алфавита внешних действий. Задан алгоритм, который для заданного внешнего действия за конечное время определяет, принадлежит оно заданному кнопочному множеству или нет.

4►Отношение ***safe by*** задано алгоритмически: задан алгоритм, который для заданной безопасной трассы и заданной неразрушающей **Q**-кнопки, разрешающей хотя бы одно действие, продолжающее трассу в спецификации, за конечное время определяет, безопасна эта кнопка после этой трассы или нет.

Слайд 23. 4. Генерация тестов.

Рестарт

Итак, по окончанию одного прогона теста делается *рестарт* системы, после чего прогоняется тот же или другой тест.

1▶Последовательность прогонов тестов в «диагональном процессе», чередующихся с рестартом системы, можно рассматривать как один тест, в котором, кроме тестовых воздействий, может встречаться рестарт системы.

2▶В общем случае вместо набора тестов можно рассматривать один тест с рестартом. Вместо конечности каждого теста в наборе тестов мы должны потребовать отсутствие в его историях бесконечного постфикса без рестарта. Как правило, нарушение этого требования влечет неполноту теста, что эквивалентно неполноте набора не обязательно конечных тестов.

3▶Кроме того, предполагается, что:

1) Рестарт системы всегда выполняется правильно (система действительно «сбрасывается» в начальное состояние), и его не нужно тестировать.
2) Рестарт имеет приоритет над внутренней активностью. Если это не так, то при дивергенции нельзя не только нажимать управляющие кнопки, но и выполнять рестарт. Тогда дивергенцию приходится понимать как нежелательное поведение и приравнивать к разрушению. В начале наших исследований мы так и делали, и только потом ослабили требование к безопасности тестирования, запретив не появление дивергенции, а попытку выхода из неё через управляющие кнопки.

4▶Тем самым, различие между (перечислимым) набором тестов и одним тестом с рестартами условное и определяется удобством организации тестирующей системы.

Слайд 24. 4. Генерация тестов.

LTS-тест. Композиция LTS

До сих пор мы рассматривали трассовый тест как множество конечных историй.

Сейчас мы посмотрим, как делается тест в LTS-модели.

Дело в том, что для LTS можно определить оператор параллельной композиции. Это один из операторов алгебры процессов. Результатом композиции двух LTS является новая LTS как результат взаимодействия LTS-параметров.

Таким образом в LTS-теории моделируется взаимодействие. Я Для трассовой модели, в которой трассы – это трассы наблюдений, аналогичную композицию моделей определить нельзя. С другой стороны, композицию можно определить для трасс готовности – ready traces. Мы придумали новый вид трасс, мы назвали их фи-трассами. Они похожи на трассы готовности и тоже позволяют определить композицию моделей. Но сейчас речь пойдёт о композиции LTS.

Есть много разных операторов параллельной композиции, но они сводятся к двум типам, соответствующих двум алгебрам процессов: CCS (Calculus of Communicating Systems), эту алгебру предложил Милнер, и CSP – Communicating Sequential Processes, эту алгебру позже придумал Hoar.

Мы используем параллельную композицию в духе CCS.

1▶Прежде всего предполагается, что для каждого внешнего действия определено противоположное действие: мы его будем подчёркивать. Это преобразование является инволюцией: если действие подчеркнуть два раза получится исходное действие.

Состояниями композиционной LTS являются пары состояний LTS-параметров. Начальное состояние – пара начальных состояний. Переходы определяются следующим образом.

2▶Если в одной LTS имеется переход по внешнему действию z, а в другой LTS – по противоположному действию $\underline{z}$, то в композиции будет тау-переход. Такой переход называется синхронным: обе LTS-операнда меняют своё состояние и конечное состояние тау-перехода композиции – это пара конечных состояний переходов LTS-операндов.

Например, передача сообщения и его приём – это противоположные действия. Если одна LTS передаёт сообщение, а другая LTS принимает то же самое сообщение, то композиция меняет своё состояние, но внешне никакие действия не наблюдаются.

Если в одной LTS имеется переход по внешнему действию z, а противоположное действие $\underline{z}$ есть в алфавите другой LTS, но перехода по этому противоположному действию нет, то в композиции по этому поводу никаких переходов не возникает.

Например, если одна LTS хочет послать сообщение другой LTS, а другая LTS не хочет его принимать, то ничего не происходит.

3►Если в первой LTS имеется тау-переход, или переход по разрушению, или переход по такому внешнему действию a, что противоположное действие $\underline{a}$ вообще не входит в алфавит второй LTS, то это называется асинхронным переходом. Только одна, первая, LTS меняет своё состояние, и в композиции получается переход тоже по тау, разрушению или действию a.

4►Симметрично тот же самый асинхронный переход возникает, если условия меняются местами для первой и второй LTS.

Алфавит композиции, тем самым, состоит из внешних действий первой LTS, для которых алфавит второй LTS не содержит противоположных действий, а также из внешних действий второй LTS, для которых алфавит первой LTS не содержит противоположных действий.

При тестировании первая LTS – это LTS-модель реализации, а вторая LTS – это LTS-модель теста.

Для тестирования нам ещё нужно уметь распознавать наблюдаемые отказы.

5►Отказ возникает в ситуации deadlock’a: для данной пары состояний реализации и теста не возникают синхронные переходы, а также асинхронные тау-переходы и переходы по разрушению.

Какой это отказ? Нажатой кнопке соответствует множество разрешённых внешних действий из алфавита реализации. Действие разрешено, если в соответствующем состоянии теста определён переход по противоположному действию. Таким образом, если возникает deadlock, то соответствующий отказ вычисляется как множество внешних действий из алфавита реализации, для каждого из которых в соответствующем состоянии теста имеет переход по противоположному действию.

Если это **R**-отказ, то он наблюдаем. В этом случае в состоянии теста может быть определён переход по символу θ. Итак, если

возникает deadlock, а в состоянии теста есть θ-переход, то в композиции тоже будет θ-переход, причём меняется только состояние теста.

В LTS-тесте вердиктам соответствуют два специальных состояния ***pass*** и ***fail***.

Реализация *проходит* LTS-тест, если в их композиции недостижимо состояние ***fail***.

Слайд 25. 4. Генерация тестов.

Примитивный LTS-тест

По примитивному трассовому тесту примитивный LTS-тест строится очень легко.

1►Состояния соответствуют нажимаемым кнопкам.

2►А кроме того, вводятся два состояния, соответствующие вердиктам ***pass*** и ***fail***.

3►Внешние действия обозначают переходы. Но внешние действия меняются на противоположные.

4►Отказы заменяются на символ θ, обозначающий также соответствующие переходы.

Слайд 26. 4. Генерация тестов.

Трассовые и LTS-тесты

Рассмотрим связь трассовых и LTS-тестов.

Если задана LTS-спецификация, то по ней строится соответствующая трассовая модель как множество трасс этой LTS. Далее по трассовой спецификации генерируется набор трассовых тестов. Трассовый тест – это префикс-замкнутое множество конечных историй. Вердикты приписаны максимальным историям теста.

Также по LTS-реализации строится соответствующая трассовая модель как множество трасс этой LTS.

Трассовая реализация проходит трассовый тест, если любой максимальной истории теста, подтрасса которой есть в реализации, приписан вердикт ***pass***, то есть не вердикт ***fail***.

Трассовый тест можно преобразовать в соответствующий LTS-тест аналогично тому, как было показано для примитивного теста.

LTS-реализация проходит LTS-тест, если в их композиции недостижимо состояние ***fail***.

Согласованность моделей трассового и LTS-тестирования опирается на согласованность отношения *passes* – «реализация проходит тест». Как мы видели на примере примитивного теста трассовая реализация проходит трассовый тест тогда и только тогда, когда соответствующая LTS-реализация проходит соответствующий LTS-тест.

1►Разумеется, можно напрямую из LTS-спецификации генерировать LTS-тесты и прогонять их на LTS-реализациях с помощью оператора параллельной композиции.

5. Оптимизация тестов

Слайд 27. 5. Оптимизация тестов.

Вычисляемые отказы

Теперь мы немножко займёмся оптимизацией тестов в общем случае, то есть без каких-либо дополнительных предположений о реализациях, кроме того, что они безопасно-тестируемы, и не опираясь на какие-либо дополнительные тестовые возможности.

Оптимизация основана на том, что, хотя набор всех примитивных тестов полон, могут быть и меньшие полные наборы тестов. Иными словами, не все примитивные тесты нужны для полноты тестирования.

Оптимизация – это удаление из набора лишних тестов.

Но если не существует конечного полного набора тестов, то такая оптимизация мало что даёт на практике: одна бесконечность практически ничем не лучше другой бесконечности.

Практически наиболее значимая задача такая: выяснить, существует ли для заданной спецификации конечный полный набор тестов и, если существует, найти его.

1►Примером лишних тестов могут служить тесты с вычисляемыми отказами.

Что это такое?

Рассмотрим цепочку отказов в некоторой трассе наблюдений. Если **R**-кнопка P вложена в объединение этих отказов, то нет никакого смысла после этой цепочки отказов нажимать кнопку P. Почему? Потому что единственным наблюдением, которое возможно и которое обязательно будет, – это отказ P. Такой отказ можно просто вычислить.

2▶Поэтому без потери полноты тестирования мы можем удалить из набора тестов все тесты, сгенерированные по трассам, в которых есть вычисляемые отказы.

Слайд 28. 5. Оптимизация тестов.

Неактуальные трассы

Другой пример – неактуальные тестовые трассы.

Напомню, что *тестовая трасса* – это безопасная трасса или безопасная трасса, продолженная безопасным после неё наблюдением, но не обязательно имеющимся в спецификации после этой трассы. У всех историй безопасного трассового теста подтрассы являются тестовыми.

Тестовую трассу будем называть *актуальной*, если она встречается хотя бы в одной безопасно-тестируемой реализации.

Понятно, что из теста можно удалить все истории, трассы которых не актуальны, без потери полноты тестирования.

1▶Прежде всего, неактуальны несогласованные тестовые трассы.

Напомню, что трасса несогласована, если она продолжается действием, которое принадлежит отказам в конце трассы. Поскольку отказ означает отсутствие действий в стабильном состоянии, после отказа нет ни внутренней активности, ни действий из отказа. Поэтому в трассовой модели несогласованные трассы не встречаются, в том числе в безопасно-тестируемых реализациях. Поэтому несогласованные тестовые трассы заведомо неактуальны. Однако в тесте такие трассы могут быть подтрассами некоторых максимальных историй. Это происходит в том случае, когда в тесте после кнопки мы определяем все возможные наблюдения, разрешаемые этой кнопкой, не обращая внимания на то, допускаются они или нет отказами перед

этой кнопкой. Разумеется, если тест строгий, то таким историям может быть приписан только вердикт ***fail***.

2►Однако, это не единственный случай неактуальности. Могут быть согласованные, но неактуальные тестовые и даже безопасные, то есть имеющиеся в спецификации, трассы. Это показывается двумя примерами.

Трасса, состоящая из одного отказа $\{a,b\}$, согласована, но в обоих примерах она не актуальна. Действительно, если бы в реализации была эта трасса, то хотя бы в одном состоянии после пустой трассы был бы ***R***-отказ {a,b}, а тогда в этом состоянии был бы ***Q***-отказ {a}. Однако, по гипотезе о безопасности в безопасно-тестируемой реализации после пустой трассы не должно быть ***Q***-отказа $\{a\}$, поскольку ***Q***-кнопка $\{a\}$ безопасна в спецификации после пустой трассы.

В примере слева трасса из одного отказа $\{a,b\}$ является тестовой трассой спецификации, но отсутствует в самой спецификации. В примере справа эта трасса имеется в спецификации, то есть является не только тестовой, но и безопасной трассой спецификации.

Слайд 29. 5. Оптимизация тестов.

Неконформные трассы

Третий пример – неконформные трассы.

Безопасная трасса спецификации *конформная*, если она встречается хотя бы в одной конформной реализации. Без потери полноты тестирования мы можем удалить из набора тестов все тесты, сгенерированные по неконформным безопасным трассам спецификации.

1►В этом примере используется обычная ***ioco***-семантика. Имеется один стимул x и две реакции a и b. Трасса $\langle \delta, x, \delta \rangle$ – конформна в левой спецификации и неконформна в правой спецификации. Это объясняется тем, что после такой трассы в любой реализации должно быть наблюдение x. После этого, согласно левой спецификации, конформно наблюдение реакции a. Однако, согласно правой спецификации, после трассы $\langle \delta, x, \delta, x \rangle$ не может быть никакого конформного наблюдения, так как реакция a не конформна

после подтрассы ⟨x, δ, x⟩, а реакция *b* не конформна после подтрассы ⟨δ, x, x⟩, и отказ δ не конформен после любой подтрассы.

2►Второй пример отличается тем, что вместо перехода-петли по реакции *a* в 5-ом состоянии имеется τ-переход. Из-за этого уже более короткая трасса ⟨δ⟩ и, следовательно, все её имеющиеся продолжения конформны в левой спецификации и неконформны в правой спецификации. Это объясняется тем, что после трассы ⟨δ⟩ в любой реализации должно быть наблюдение *x*, после которого конформен только отказ δ, а все реакции неконформны. Тем самым, наличие трассы ⟨δ⟩ в конформной реализации влекло бы наличие в ней неконформной трассы ⟨δ, x, δ⟩.

3►Ещё более удивителен третий пример, в котором и переход-петля по реакции *b* в 3-ем состоянии заменяется на τ-переход. Из-за этого даже пустая трасса, а тем самым и все имеющиеся трассы, конформны в левой спецификации и неконформны в правой спецификации. Это объясняется тем, что после пустой трассы конформен только отказ δ, а все реакции неконформны.

Тем самым, наличие пустой трассы в конформной реализации влекло бы наличие в ней неконформной трассы ⟨δ⟩. Но пустая трасса есть в любой реализации, следовательно, все реализации неконформны.

Эти примеры показывают, насколько полезным может оказаться анализ неконформных трасс. Полный набор примитивных тестов для любой из спецификаций в этих примерах бесконечен, поскольку бесконечно число безопасных трасс (за счет цикла в состоянии 7). Тем самым полное тестирование бесконечно.

Для правой спецификации из первого примера после получения трассы ⟨δ, x, δ⟩ можно сразу закончить тестирование с вердиктом ***fail***.

Для правой спецификации из второго примера то же самое относится уже к более короткой трассе ⟨δ⟩.

Для правой спецификации из третьего примера тестирование вообще излишне. У этой спецификации нет конформных реализаций,

что определяется без всякого тестирования простым анализом спецификации.

4►Запомним эту спецификацию S_6. Она нам ещё понадобится.

Слайд 30. 5. Оптимизация тестов.

Пополнение спецификации – удаление неконформных трасс

Наличие неконформных трасс спецификации позволяет говорить об ошибках двух родов.

Определим ошибку как продолжение $\sigma\cdot\langle u\rangle$ конформной безопасной трассы σ спецификации **S** безопасным после неё наблюдением u, которое

- (*ошибка 1-го рода*) отсутствует в спецификации $\sigma\cdot\langle u\rangle \notin \mathbf{S}$, или
- (*ошибка 2-го рода*) имеется в спецификации $\sigma\cdot\langle u\rangle \in \mathbf{S}$, но делает трассу $\sigma\cdot\langle u\rangle$ неконформной.

1►Ошибка $\sigma\cdot\langle u\rangle$ *первичная*, если нет ошибки $\mu\cdot\langle v\rangle < \sigma\cdot\langle u\rangle$.

Ошибка $\sigma\cdot\langle u\rangle$ *вторичная*, если есть ошибка $\mu\cdot\langle v\rangle < \sigma\cdot\langle u\rangle$.

Понятно, что для проверки конформности достаточно искать только первичные ошибки.

2►Поэтому наша задача – удалить из спецификации неконформные трассы.

Тогда все первичные ошибки 2-го рода становятся первичными ошибками 1-го рода, а все вторичные ошибки удаляются.

Для этого делается преобразование *пополнения* спецификации.

Слайд 31. 5. Оптимизация тестов.

Пополнение спецификации – удаление неконформных трасс

До сих пор мы рассматривали спецификацию в семантике, которая подразумевалась, и говорили, что вместе со спецификационной моделью нужно ещё задавать отношение ***safe by***. Тем самым, полностью (без умолчаний) спецификация задаётся

спецификационной тройкой : **R/Q**-семантика, **R**-модель **S**, отношение ***safe by***.

1►Пусть заданы две спецификационные тройки **T** и **T`**.

Определим отношение **L**-вложенности спецификационных троек.

Будем говорить, что тройка **T** **L**-вложена в тройку **T`**, если последняя определяет не меньший класс безопасно-тестируемых и тот же самый класс конформных реализаций в алфавите ***L***.

То есть она определяет не меньшее пересечение класса безопасно-тестируемых реализаций и, соответственно, то же самое пересечение класса конформных реализаций с классом реализаций в заданном алфавите ***L***.

Заметим, что это определение годится не только в том случае, когда две спецификации определены для одной и той же семантики взаимодействия.

2►Для того, чтобы сохранить возможности тестирования реализаций в алфавите **L**, **R`/Q`**-семантика должна быть **L**-эквивалентна исходной **R/Q**-семантике: если взять пересечения всех её **R`**-кнопок с алфавитом **L** должно получиться семейство **R**, а если взять пересечения всех её **Q`**-кнопок с алфавитом **L** должно получиться семейство **Q**. Иными словами, её кнопки отличаются от кнопок исходной семантики только наличием в них каких-то дополнительных действий не из алфавита **L**.

Понятно, что реализация в алфавите **L** не содержит этих дополнительных действий. Поэтому нажатие кнопки на **R`/Q`**-машине, внутри которой находится эта реализация, вызывает тот же эффект, что нажатие соответствующей кнопки на **R/Q**-машине, внутри которой находится эта реализация.

3►**L**-конусом назовём множество спецификационных троек, в которые **L**-вложена данная тройка **T** и которые имеют **L**-эквивалентную семантику.

Смысл этого определения в том, что любая тройка из **L**-конуса может заменять собой исходную тройку **T**, то есть использоваться вместо неё.

Действительно, тестовые возможности для тестирования реализаций в алфавите **L** те же самые.

Но нас интересует не просто тестирование, а безопасное тестирование. Поскольку класс безопасно-тестируемых реализаций в алфавите **L** не сужается, мы по-прежнему, можем безопасно-тестировать те же самые реализации, что для исходной тройки.

Наконец, класс конформных реализаций в алфавите **L** не меняется. А это означает, что мы предъявляем к реализациям те же самые спецификационные требования, но только сформулированные несколько иным способом.

Слайд 32. 5. Оптимизация тестов.

Пополнение спецификации – удаление неконформных трасс

Наличие неактуальных и неконформных трасс в спецификации является следствием нерефлексивности конформности ***saco***, в том числе и отношения ***ioco*.**

Если бы спецификация была конформна сама себя, она была бы, во-первых, безопасно-тестируемой реализацией и, следовательно, в ней не могло бы быть неактуальных трасс, а, во-вторых, она была бы конформной реализацией и, следовательно, в ней не могло бы быть неконформных трасс.

Заметим, что если спецификация удовлетворяет собственной гипотезе о безопасности, то она конформна сама себе. Иными словами, если спецификация не конформна сама себе, то она не безопасно-тестируемая.

Нерефлексивность конформности неприятна ещё и тем, что реализация буквально «списанная» со спецификации в общем случае оказывается заведомо ошибочной, поскольку не удовлетворяет гипотезе о безопасности. Эта проблема может быть также решена пополнением спецификации.

1►*Пополнением* будем называть преобразование спецификационной тройки в такую заменяющую её тройку, то есть тройку из **L**-конуса, которая конформна сама себе и, следовательно, в ней нет неактуальных и неконформных трасс.

2►Для некоторых семантик, в частности, для ***ioco***-семантики удается сделать пополнение в той же семантике.

Доказательство этого утверждения можно найти в нашей прошлогодней статье в журнале «Программирование»:

Пополнение спецификации для ioco. «Программирование», 2011, No. 1.

3▶В качестве примера рассмотрим спецификацию $\mathbf{S}_2$, которые мы уже видели раньше. В ней неконформна трасса дельта-x-дельта.

4▶Для пополнения достаточно удалить один переход.

Слайд 33. 5. Оптимизация тестов.

Пополнение спецификации – удаление неконформных трасс

В общем случае не существует пополнения *в той же самой* семантике.

1▶Вот пример слева спецификации $\mathbf{S}_7$, которая как свою часть использует LTS S_6, в которой все трассы неконформны.

2▶В этой спецификации трасса <y> неконформна. Но если мы удалим переход по y, ведущий в начальное состояние LTS $\mathbf{S}_6$, то у нас возникнет отказ дельта, после которого x станет опасным. Из-за этого класс безопасно-тестируемых реализаций не уменьшится. Однако, рассмотрим безопасно-тестируемую реализацию, в которой есть трасса дельта-x-дельта. Она неконформна для $\mathbf{S}_7$, поскольку после дельта-x допускается только y. А после преобразования она станет конформной. Поэтому такое преобразование не является пополнением.

Более строгое доказательство этого утверждения я опускаю для экономии времени.

3▶Его можно найти в нашей работе:

Удаление из спецификации неконформных трасс. Препринт №23, ИСП РАН, 2011.

http://www.ispras.ru/ru/preprints/archives/prep_23_2011.php

4▶Более того, пополнения нет в любой семантике, где **1)** есть хотя бы две разные **R**-кнопки и **2)** хотя бы одна **Q**-кнопка Q разрешает действие, не разрешаемое этими двумя кнопками.

Более точно: в такой семантике существует такая спецификационная модель и такое отношение ***safe by***, что конформные реализации существуют, но не существует пополнения в той же семантике.

Заметим, что ***ioco***-семантика не относится к такому классу семантик: в ней есть только одна **R**-кнопка приёма всех реакций. Семантика спецификации $\mathbf{S}_7$ относится к такому классу семантик: она получается из ***ioco***-семантики добавлением пустой **R**-кнопки, которая позволяет наблюдать переход LTS-реализации в стабильное состояние, то есть исчезновение внутренней активности при запрете всех внешних действий.

5►Однако можно построить пополнение в другой, расширенной семантике. Для этого во все или некоторые кнопки добавляются новые действия не из алфавита **L**. В каждую кнопку P добавляется одно действие, которое мы называем не-отказом – не-P и обозначаем двойным зачёркиванием. Очевидно, такая семантика **L**-эквивалентна исходной семантике.

6►Для примера спецификации $\mathbf{S}_7$ на слайде справа изображено её пополнение в такой расширенной семантике. Здесь не-отказ добавляется только в кнопку дельта.

Слайд 34. 5. Оптимизация тестов.

Пополнение спецификации – удаление неконформных трасс

С помощью не-отказов удаётся построить пополнение для любой исходной спецификации в любой исходной семантике с любым отношением ***safe by***.

1►Доказательство этого утверждения и алгоритм построения пополнения можно найти в нашей работе:

Удаление из спецификации неконформных трасс. Препринт №23, ИСП РАН, 2011.

2►Когда можно построить конечное пополнение – в виде конечной LTS?

Это удаётся сделать, если выполнены следующие достаточные условия:

- *Конечен* алфавит внешних действий.
- *Конечна* LTS-модель спецификации: конечно число достижимых состояний и переходов.
- Отношение ***safe by*** *ограниченное*.

3►Что значит ограниченность отношения ***safe by***?

Это значит, что в LTS-модели спецификации безопасность кнопок одинакова после трасс, заканчивающихся в одном и том же множестве состояний.

Есть гипотеза, что для конечности пополнения достаточно не ограниченности ***safe by***, а его регулярности.

Отношение ***safe by*** *регулярно*, если регулярно множество трасс вида $\sigma\cdot\langle Ꝑ\rangle$, где кнопка P безопасна по ***safe by*** после трассы σ.

Достаточно очевидно, что ограниченный ***safe by*** регулярный, но обратное, вообще говоря, не верно.

Слайд 35. 5. Оптимизация тестов.

Пополнение спецификации – удаление неконформных трасс

При построении конечного LTS-пополнения важным промежуточным результатом является модель, которую мы назвали **RTS** - Refusal Transition System.

Это детерминированная LTS, в которой отказы изображаются не виртуальными петлями в стабильных состояниях, а явными переходами, помеченными этими отказами. Переходы по отказам – не обязательно петли. Также могут быть переходы по дивергенции.

По сути RTS – это граф, порождающий префикс-замкнутое множество трасс в алфавите внешних действий, отказов и символа дивергенции.

1►Любую LTS **S** можно преобразовать в RTS ***P*(S)** с помощью известной процедуры детерминизации LTS.

Состояниями RTS становятся множества состояний исходной LTS. Начальное состояние RTS – это множество состояний исходной LTS после пустой трассы. Иными словами, множество состояний, достижимых из начального состояния LTS по тау-переходам.

Переходы RTS порождаются вот таким правилом вывода. Переход по z, где z – это действие, **R**-отказ, дивергенция или разрушение, ведет из множества A состояний LTS во множество B состояний LTS, если хотя бы в одном состоянии из A начинается трасса длины 1, состоящая из этого z. При этом множество состояний LTS, в которых заканчиваются все такие трассы, начинающиеся в состояниях из A, – это множество B.

Иными словами, переход по действию z ведет из A в B, если хотя бы в одном состоянии из A есть переход по действию z, и множество состояний, достижимых из концов таких переходов по тау-переходам, – это множество B.

Переход по отказу z ведет из A в B, если хотя бы в одном состоянии из A есть отказ z, и множество таких состояний – это множество B.

Переход по дивергенции ведет из A в B, если хотя бы одно состояние из A дивергентно, B – терминальное состояние.

Переход по разрушению ведет из A в B, если хотя бы в одном состоянии из A есть разрушение, B – терминальное состояние.

Слайд 36. 5. Оптимизация тестов.

Пополнение спецификации – удаление неконформных трасс

RTS пополнения строится в семантике, расширенной не-отказами.

В такой семантике переходы помечаются, во-первых, действиями из расширенного алфавита, то есть действиями из исходного алфавита **L** и не-отказами для всех кнопок, во-вторых, **R**-отказами, и, в третьих, Δ и γ.

В RTS пополнения выделены четыре состояния: γ, Δ, $\Delta`$, ϖ.

Состояние ϖ терминально. В нём заканчиваются только переходы по дивергенции и разрушению.

Имеется единственный переход по разрушению, ведущий из состояния γ в состояние ϖ, и два перехода по дивергенции, ведущий из состояний Δ и $\Delta`$ в состояние ϖ.

Переходы по не-отказам заканчиваются только в состояниях γ, Δ и $\Delta`$.

RTS обладает целым рядом свойств, очень удобных для генерации тестов. Детерминированность гарантирует, что каждая её трасса заканчивается только в одном состоянии. Кроме того, можно выделить ещё три свойства.

1**Безопасные трассы.**

Все трассы, не заканчивающиеся в выделенных состояниях, безопасны. Множество трасс этой RTS, вообще говоря, не является трассовой моделью. Может не выполняться свойство замкнутости по ***d***-операции – удалению отказов из трасс. Но замыкание по этой операции уже является трассовой моделью. Также LTS-модель пополнения также может быть построена из этой RTS алгоритмически.

Важно то, что в трассовой модели, соответствующей RTS, нет других безопасных трасс.

Кроме того, поскольку это результат преобразования пополнения, все безопасные трассы RTS конформны.

2**Безопасные кнопки.**

Безопасность кнопок после трасс, заканчивающихся в одном состоянии, одинаковая. Она определяется очень просто. Если из состояния переход по не-отказу ведёт в состояние γ, то соответствующая кнопка опасна. В противном случае она безопасна.

3**Актуальные наблюдения.**

Два состояния Δ и $\Delta`$ введены для того, чтобы различать актуальные и неактуальные наблюдения, которые безопасны, но отсутствуют в спецификации. Это позволяет легко оптимизировать каждый тест, оставляя в нём только актуальные наблюдения.

Безопасное действие актуально в состоянии, если оно не принадлежит отказам, которыми помечены переходы-петли в этом состоянии.

Безопасный **R**-отказ актуален в состоянии, если в этом состоянии нет перехода по соответствующему не-отказу в состояние $\Delta`$. Это эквивалентно тому, что в этом состоянии либо есть переход по отказу, либо есть переход по не-отказу, ведущий в состояние Δ.

Слайд 37. 5. Оптимизация тестов.

Демонические трассы

Бесконечный набор тестов требует, естественно, бесконечного времени генерации. Конечный набор тестов хотелось бы генерировать за конечное время. Как это сделать?

Проблема в том, что конечность полного набора тестов не означает конечности множества безопасных трасс, а тесты генерируются как раз по безопасным трассам спецификации.

1►При генерации тестов происходит перечисление безопасных трасс и фильтрация получающихся тестов для того, чтобы осталось конечное множество тестов. Отбрасываются те тесты, которые можно было бы завершить раньше: как только исчезает неопределенность в возможном вердикте. Это такие «лишние» тесты, в которых после нажатия последней кнопки возможен только вердикт ***fail*** или только вердикт ***pass***.

2►***fail***-тесты. Если после получения трассы и нажатия некоторой кнопки возможен только вердикт ***fail***, то это означает, что трасса неконформна. По такой трассе не нужно генерировать тесты. Эта задача решается с помощью пополнения.

3►***pass***-тесты. Если в спецификации трасса, по которой генерировался тест, продолжается каждым актуальным наблюдением, разрешаемым безопасной кнопкой, то после получения трассы и нажатия этой кнопки возможен только вердикт ***pass***. Такой тест тоже оказывается «лишним»: нужно либо выбирать другую кнопку, либо другую трассу. Заметим, что, вообще говоря, если тест «лишний» по вердикту ***pass***, то это не значит, что «лишние» все его продолжения. Поэтому при генерации тестов можно просто отфильтровывать такие ***pass***-тесты.

4►Особый случай возникает тогда, когда после любого безопасного продолжения безопасной трассы любое безопасное актуальное наблюдение имеется в спецификации. Иными словами, после трассы спецификация допускает произвольное, с учетом безопасности, поведение реализации. Такие трассы будем называть *демоническими*. Поскольку безопасное продолжение демонической трассы по определению демоническое, фильтрация будет работать «вхолостую» на всех демонических трассах, префиксом которых является данная демоническая трасса. Вместо этого нужно было бы просто прекратить генерировать тесты по таким трассам.

Особенно это важно в том случае, когда существует конечный полный набор тестов, но демонических трасс бесконечно много: без учета демонических трасс генерация этого конечного набора тестов

потребует бесконечного времени, впустую потраченного на фильтрацию.

Слайд 38. 5. Оптимизация тестов.

Демонические трассы

Возникает задача предварительного выделения среди всех безопасных трасс спецификации недемонических трасс и перечисления только таких трасс для генерации тестов.

Эта задача не такая сложная, как пополнение.

Но демонические трассы удобно отслеживать не на LTS, а на RTS. Поскольку RTS детерминирована, каждая её трасса заканчивается ровно в одном состоянии.

1►Нам нужно найти максимальное множество демонических состояний RTS. Состояние демоническое, если оно удовлетворяет двум условия:

1) Для каждой *безопасной* кнопки в состоянии имеется каждое *актуальное* наблюдение, разрешаемое этой кнопкой.

   Действие актуально в состоянии, если в этом состоянии нет перехода-петли по отказу, которому это действие принадлежит.

   R-отказ актуален в состоянии, если после добавления этого отказа к тем отказам, по которым в состоянии имеются переходы-петли, получается множество отказов, в совокупности не покрывающее никакую безопасную **Q**-кнопку.

   В RTS-пополнении **R**-отказ актуален в состоянии, если нет перехода по не-отказу из этого состояния в состояние дельта-штрих и, конечно, поскольку отказ безопасен, не должно быть перехода по не-отказу в состояние гамма.

2) Все переходы из этого состояния ведут тоже в демонические состояния.

Что здесь важно?

Во-первых, любая трасса, заканчивающаяся в демоническом состоянии RTS, является демонической.

Во-вторых, любая демоническая трасса заканчивается в демоническом состоянии RTS.

Это свойство верно только для RTS в силу её детерминированности. В LTS тоже можно было бы выделять демонические состояния, но там это не имеет смысла, поскольку трасса может заканчиваться в нескольких состояниях. И может оказаться, что все эти состояния недемонические, а трасса, тем не менее, демоническая, поскольку все эти состояния в совокупности дают все наблюдения по всем безопасным после трассы кнопкам, и после каждого такого наблюдения мы опять попадает во множество состояний, обладающее этим свойством. Иными словами, в LTS нужно опираться на демоническое множество состояний. А при переходе от LTS к RTS мы как раз и выбираем в качестве состояний RTS множество состояний исходной LTS.

2▶Итак, нам нужно найти максимальное множество демонических состояний RTS.

Такое множество может быть и пустым, тогда демонических трасс нет. Если же это множество не пусто, то демонические трассы – это все трассы, заканчивающиеся в демонических состояниях.

Для генерации тестов нам достаточно пометить переходы, ведущие в демонические состояния, и генерировать тесты по безопасным трассам, не проходящим по этим переходам.

3▶Как построить максимальное множество демонических состояний?

Сначала строим множество состояний, удовлетворяющих первому условию. Потом просматриваем переходы из состояний этого множества. Если переход из данного состояния ведёт за пределы множества, удаляем это состояние из множества. И так до тех пор, пока это возможно.