И. Бурдонов, А. Косачев
Семантики взаимодействия с отказами, дивергенцией и разрушением. Часть 1
ИСПРАН
25 слайдов

Семантики взаимодействия с отказами, дивергенцией и разрушением.

1. Формализация тестирования

Слайд 1. Читаю название

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

Слайд 3. 1. Формализация тестирования.
Моделирование

Задача верификации – проверка правильности исследуемой системы.

►Под «правильностью» понимается соответствие системы заданным требованиям.

►Для того, чтобы это отношение формализовать, используются формальные модели. В модельном мире система отображается в реализационную модель (реализацию), а требования – в спецификационную модель (спецификацию).

►Соответствие исследуемой системы требованиям отображается в бинарное отношение конформности.

Спецификация всегда задана, а существование реализации (как модели реальной системы) предполагается (тестовая гипотеза). Если

реализация также задана явно, верификация конформности может быть выполнена аналитически.

►Для реализации, устройство которой неизвестно («черный ящик») или слишком сложно для анализа, приходится применять тестирование как проверку конформности в процессе тестовых экспериментов.

►Разумеется, в этом случае требования должны быть функциональными, то есть, выражены в терминах взаимодействия системы с окружающим миром, который при тестировании подменяется тестом.

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

Слайд 4. 1. Формализация тестирования.
Семантика взаимодействия (1)

Семантика взаимодействия формализует имеющийся набор тестовых возможностей по управлению и наблюдению за поведением тестируемой системы. При тестировании мы можем наблюдать только такое поведение реализации, которое, во-первых, «спровоцировано» тестом (управление) и, во-вторых, наблюдаемо во внешнем взаимодействии. Такое взаимодействие может моделироваться с помощью, так называемой, машины тестирования.

►Она представляет собой «чёрный ящик», внутри которого находится реализация.

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

Подчеркнём, что при управлении оператор разрешает реализации выполнять именно множество действий, а не обязательно одно действие.

▶Мы предлагаем считать, что оператор может нажимать только одну кнопку, но каждой кнопке соответствует своё множество разрешаемых действий.

▶Наблюдения (на «дисплее» машины) бывают двух типов. Первый тип – это наблюдение некоторого *внешнего (наблюдаемого) действия*, разрешённого оператором и выполняемого реализацией.

После наблюдения кнопка отжимается, и все внешние действия запрещаются.

▶Далее оператор может нажать другую (или ту же самую) кнопку.

▶Второй тип наблюдения – это наблюдение *отказа* как отсутствия каких бы то ни было наблюдаемых действий. То есть все действия, разрешенные нажатой кнопкой, в реализации выполняться не могут.

▶Тестовые возможности определяются тем, какие «кнопочные» множества есть на клавиатуре машины, а также, для каких кнопок возможно наблюдение отказа. Тем самым, семантика взаимодействия определяется алфавитом внешних действий $\boldsymbol{L}$ и двумя наборами кнопок машины тестирования: с наблюдением соответствующих отказов – семейство $\boldsymbol{R}\subseteq 2^{\boldsymbol{L}}$ и без наблюдения отказа – семейство $\boldsymbol{Q}\subseteq 2^{\boldsymbol{L}}$.

▶Предполагается, что $\boldsymbol{L}\cap\boldsymbol{2}^{\boldsymbol{L}}=\varnothing$, $\boldsymbol{R}\cap\boldsymbol{Q}=\varnothing$ и $\cup\boldsymbol{R}\cup\cup\boldsymbol{Q}=\boldsymbol{L}$. Такую семантику мы называем $\boldsymbol{R}/\boldsymbol{Q}$-семантикой.

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

▶Еще один пример – это семантика популярного отношения ***ioco***, когда действия разбиваются на стимулы (input) и реакции (output) $\boldsymbol{L}=\boldsymbol{I}\cup\boldsymbol{O}$. Есть только одна $\boldsymbol{R}$-кнопка приема всех реакций $\boldsymbol{R}=\{\delta\}$, где $\delta=\boldsymbol{O}$, а соответствующий отказ δ называется по-английски *quiescence*, то есть молчание. Мы обычно называем это *стационарностью*. Каждый стимул x посылается в реализацию с помощью $\boldsymbol{Q}$-кнопки $\{x\}$, $\boldsymbol{Q}=\{\{x\}|x\in\boldsymbol{I}\}$.

Слайд 5. 1. Формализация тестирования.

Семантика взаимодействия (2)

Кроме внешних действий реализация может совершать внутренние (ненаблюдаемые) действия, которые, поскольку ненаблюдаемы, неразличимы между собой и обозначаются одним символом τ. Эти действия считаются всегда разрешенными (при нажатии любой кнопки или при отсутствии нажатой кнопки).

► Для выполнимости любого действия (как внешнего, так и внутреннего) необходимо, чтобы оно было определено в реализации и разрешено оператором. Если этого условия также и достаточно, то есть на выполнение может быть выбрано любое действие, удовлетворяющее этому условию, то говорят, что в системе нет приоритетов. Пока мы ограничимся только системами без приоритетов.

►«Практические предположения»: 1) Любая конечная последовательность любых действий (как внешних, так и внутренних) совершается за конечное время, а бесконечная – за бесконечное время. 2) Передача тестового воздействия (нажатие кнопки) от машины тестирования в реализацию и передача наблюдения обратно от реализации на дисплей машины выполняются за конечное время. Эти предположения гарантируют возможность наблюдения внешнего действия, выполняемого реализацией, через конечное время после нажатия кнопки, разрешающей это действие.

Это предположение часто используется для реализации наблюдения ***R***-отказа, но в усиленном варианте: время выполнения каждого действия, разрешаемого кнопкой, вместе с возможными предшествующими ему внутренними действиями не только конечно, но и ограничено. В этом случае вводится тайм-аут, истечение которого без наблюдения действия трактуется как отказ. Следует отметить, что это не единственный возможный способ реализации наблюдения отказа.

►Бесконечная последовательность τ-действий («зацикливание») называется *дивергенцией* и обозначается символом Δ. Дивергенция сама по себе не опасна, но при попытке выхода из неё, когда оператор нажимает любую (***R***- или ***Q***-) кнопку, он не знает, нужно ли ждать

наблюдения (внешнего действия или ***R***-отказа) или бесконечно долго будут выполняться только внутренние действия. Поэтому оператор не может ни продолжать тестирование, ни закончить его.

►При отсутствии дивергенции после нажатия ***R***-кнопки через конечное время оператор наблюдает или разрешенное этой кнопкой внешнее действие или соответствующий отказ. Однако, при нажатии ***Q***-кнопки, если в реализации возможен отказ, то, поскольку этот отказ не наблюдаем, оператор не знает, нужно ли ему ждать наблюдения внешнего действия или такого действия не будет, поскольку возник отказ. Поэтому оператор не может ни продолжать тестирование, ни закончить его.

►Кроме этого мы вводим специальное, также не регулируемое кнопками действие, которое называем *разрушением* и обозначаем символом γ. Оно моделирует любое нежелательное поведение системы, в том числе и ее реальное разрушение, которого нельзя допускать при взаимодействии.

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

►Тестирование, при котором не возникает попыток выхода из дивергенции, ненаблюдаемых отказов и разрушения, называется *безопасным*.

2. Модели реализации и спецификации

Слайд 6. 2. Модели.
LTS-модель

Наиболее распространенной моделью реализации и спецификации является система помеченных переходов – LTS (Labelled Transition System), которая определяется как ориентированный граф, вершины которого называются состояниями, а дуги помечены внешними действиями или символами τ или γ и называются переходами.

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

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

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

►В примере мы видим 4 отказа в трех состояниях.

►Состояние *дивергентно*, если в нем начинается бесконечный τ-маршрут, то есть маршрут, все переходы которого помечены символом τ. Состояние конвергентно, если оно не дивергентно.

►В примере имеется одно дивергентное состояние.

►Для взаимодействия, основанного на наблюдениях, единственным результатом тестового эксперимента является чередующаяся последовательность кнопок (тестовых воздействий) и наблюдений, которую будем называть (*тестовой*) *историей*. В силу семантики дивергенции и разрушения достаточно рассматривать только такие истории, в которых символы Δ и γ либо не встречаются, либо являются последними символами. Поскольку дивергенция и разрушение всегда разрешены, им не предшествует в истории никакая кнопка. Любое другое наблюдение *u* (внешнее действие или ***R***-отказ) разрешается непосредственно предшествующей ему кнопкой *P*, то есть $u \in P$ или $u = P$ для $P \in \boldsymbol{R}$. Подпоследовательность истории, состоящая только из наблюдений (включая Δ и γ), называется трассой.

Для систем без приоритетов важны только трассы, поскольку возможность или невозможность появления данного наблюдения после некоторой трассы определяется только тем, что нажимаемая кнопка разрешает данное наблюдение, и не зависит от того, какие еще наблюдения она разрешает. В этом случае для данной тестируемой системы множество ее историй однозначно восстанавливается по множеству ее трасс.

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

Слайд 7. 2. Модели.
Трассовая модель

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

►Множество трасс LTS можно рассматривать как самостоятельную *трассовую модель*. Можно дать и независимое от LTS определение трассовой модели как множество трасс, обладающее определенным набором из 5 свойств.

►
Допустимость: Δ и γ только в конце трасс.
Согласованность: после отказов нет действий из этих отказов, нет Δ и нет γ.
Конвергентность: если в конце трассы и после нее нет Δ и γ, то трасса продолжается ∀ **R**-отказом или действием из него.
Замкнутость: замкнутость по удалению отказа из трассы.
Полнота: замкнутость по вставке **R**-отказа в трассу после префикса, не продолжающегося действиями из этого отказа и заканчивающегося каким-нибудь отказом.

По сути этот набор свойств эквивалентен тому, что существует LTS с этим множеством трасс. И наоборот: множество трасс любой LTS обладает этим набором свойств. С этой точки зрения LTS-модель и трассовая модель эквивалентны.

▶Для произвольной ***R***/***Q***-семантики трасса, в которой все отказы принадлежат семейству ***R***, мы называем ***R***-*трассой*, а трассовую модель, содержащую только ***R***-трассы, – ***R***-*моделью*. При безопасном тестировании в ***R***/***Q***-семантике наблюдаться могут только ***R***-трассы, не содержащие символов Δ и γ.

▶ *Простой* трассой называется трасса, в которой нет отказов, то есть она содержит только действия.

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

▶Для ***ioco***-семантики ***R***-трассы (без Δ и γ) называются *suspension traces*. Это трассы, в которых встречается только отказ δ.

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

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

▶С другой стороны, LTS обладает существенным неудобством, связанным с недетерминизмом. Недетерминизм в LTS проявляется как наличие τ-переходов и/или «веера» переходов $s—z\rightarrow s`$ из одного состояния s по одному и тому же внешнему действию z, ведущих в

разные постсостояния *s`*. Из-за этого трасса в LTS заканчивается, вообще говоря, не в одном состоянии, а во множестве состояний. Этот недетерминизм в LTS, вообще говоря, неустраним. Это объясняется тем, что трасса может продолжаться как отказом, так и действием из этого отказа. Такая трасса не может заканчиваться только в одном состоянии, поскольку в одном состоянии не может быть и отказа и действия из этого отказа.

3. Гипотеза о безопасности и безопасная конформность

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

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. 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 *P* безопасна в модели после трассы σ». При безопасном тестировании будут нажиматься только безопасные кнопки. Это отношение различно для реализационной и спецификационной моделей.

►В полной трассовой реализации ***I*** отношение безопасности называется ***safe in***. Что оно означает? Прежде всего, кнопка должна быть *неразрушающей* после трассы, то есть 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 ***safe***$_{\gamma\Delta}$ ***I after*** σ $=_{\mathbf{def}}\ \sigma\cdot\langle\Delta\rangle\notin\boldsymbol{I}\ \&\ \forall u\in P\ \sigma\cdot\langle u,\gamma\rangle\notin\boldsymbol{I}$.

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

►В полной трассовой спецификации ***S*** отношение безопасности (***safe by***) отличается только для ***Q***-кнопок: мы не требуем, чтобы после трассы σ не было ***Q***-отказа Q, но требуем, чтобы было хотя бы одно действие $z\in Q$. Кроме того, если действие разрешается хотя бы одной неразрушающей кнопкой, то оно должно разрешаться какой-нибудь безопасной кнопкой. Если это неразрушающая ***R***-кнопка, 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. Однако в целом указанные требования неоднозначно определяют отношение ***safe by***, и при задании спецификации ***S*** дополнительно указывается конкретное отношение ***safe by***. Требования к отношению ***safe by*** записываются так: $\forall R\in\boldsymbol{R}\ \forall z\in\boldsymbol{L}\ \forall Q\in\boldsymbol{Q}\ \forall\sigma\in\boldsymbol{S}$

R ***safe by S after*** $\sigma\Leftrightarrow R$ ***safe***$_{\gamma\Delta}$ ***S after*** σ,

$\exists P\in\boldsymbol{R}\cup\boldsymbol{Q}\ P$ ***safe***$_{\gamma\Delta}$ ***S after*** $\sigma\ \&\ z\in P\ \&\ \sigma\cdot\langle z\rangle\in\boldsymbol{S}\Rightarrow\exists P`\in\boldsymbol{R}\cup\boldsymbol{Q}\ z\in P`\ \&$ $P`$ ***safe by S after*** σ,

Q ***safe by S after*** $\sigma\Rightarrow Q$ ***safe***$_{\gamma\Delta}$ ***S after*** $\sigma\ \&\ \exists v\in Q\ \sigma\cdot\langle v\rangle\in\boldsymbol{S}$.

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.

►Теперь мы можем определить *безопасные* ***R****-трассы*. ***R***-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).

►Наблюдение опасно после трассы, если оно не является безопасным после трассы.

►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.

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

Заметим, что для ***ioco***-семантики мы разрешаем в безопасно-тестируемых реализациях «безопасные» блокировки стимулов – блокировки стимулов после трасс, которые в спецификации этими стимулами не продолжаются. Это более либерально, чем требование всюду определенности реализации по стимулам, предлагаемое автором отношения ***ioco*** Яном Тритмансом [16,17].

Таким образом, мы устраняем «несогласованность» отношения ***ioco***, когда всюду определенная по стимулам реализация и реализация, отличающаяся от нее только тем, что в ней есть «безопасные» блокировки, неразличимы при ***ioco***-тестировании, однако первая может быть конформна, а вторая заведомо неконформна, поскольку не является всюду определенной по стимулам и тем самым не входит в домен отношения ***ioco***.

▶После этого можно определить отношение (безопасной) *конформности*: реализация ***I*** *безопасно конформна* (или просто

конформна) спецификации ***S***, если она безопасно-тестируема и выполнено

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

Это отношение определяет класс конформных реализаций ***ConfImp***.

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

Эта пара классов реализаций 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 – это уже будут не все возможные пары.

Слайд 11.2. Вернемся к Модели.
RTS-модель

Как уже отмечалось выше, LTS обладает существенным неудобством, связанным с недетерминизмом. Причем этот недетерминизм неустраним, пока мы пользуемся LTS-моделью. Из-за этого трасса в LTS заканчивается, вообще говоря, не в одном состоянии, а во множестве состояний.

Мы предложили еще одну эквивалентную модель, в которой этого неудобства нет. Она называется RTS (Refusal Transition System) и представляет собой детерминированную LTS, но не в алфавите ***L***, а в

алфавите $\boldsymbol{L}\cup\boldsymbol{R}\cup\{\Delta\}$, то есть в ней могут быть явные переходы по ***R***-отказам даже в другие состояния, а бесконечная цепочка τ-переходов $s\text{—}\tau\rightarrow\ldots$ заменена Δ-переходом $s\text{—}\Delta\rightarrow$. Иными словами, ***R***-отказы, поскольку они входят в алфавит, считаются действиями. Такая RTS всегда существует, поскольку любое префикс-замкнутое множество трасс порождается графом, в котором есть одна начальная вершина – начальное состояние, и все вершины конечные. Также известно, что такой порождающий граф всегда может быть выбран детерминированным.

▶По исходной LTS такую RTS-модель можно получить следующим преобразованием.

Состояниями RTS становятся множества состояний LTS в конце ***R***-трасс, что аналогично известному алгоритму «детерминизации» порождающего графа. Простые трассы такой RTS – это все трассы исходной LTS.

▶Для наших целей достаточно, чтобы множество простых трасс RTS не обязательно было трассовой моделью. Мы потребуем только, чтобы трассовой моделью было его замыкание по операции удаления отказов.

За детерминизм RTS приходится расплачиваться наглядностью: не всякая детерминированная LTS в алфавите $\boldsymbol{L}\cup\boldsymbol{R}\cup\{\Delta\}$ является RTS, а только такая, которая удовлетворяет специальному набору условий, определяемых семантикой взаимодействия. По сути этот набор свойств достаточен для того, чтобы замыкание множества простых трасс по операции удаления отказов являлось трассовой моделью. И наоборот: для любой трассовой модели существует такая RTS, что замыкание множества ее простых трасс по операции удаления отказов является этой трассовой моделью.

▶В качестве такого набора свойств достаточно выбрать следующий набор: далее по экрану.

▶Проверка этих свойств требует времени линейного по отношению к числу переходов и состояний при заданном числе **R**-кнопок.

►Тем самым, все три модели: LTS, трассовая модель и RTS – эквивалентны в том смысле, что преобразуются одна в другую по определенным правилам.

Слайд 12.2. Вернемся к Модели.
Финальная RTS-модель

Для чего нам понадобилось, чтобы RTS представляла в качестве множества своих простых трасс не всю трассовую модель, а только ее часть?

Это объясняется тем, что для определения гипотезы о безопасности и безопасной конформности не нужно знать все **R**-трассы модели и все кнопки, безопасные после них по ***safe by***. Достаточно знать только безопасные по ***safe by*** трассы и кнопки, безопасные только после таких трасс. Безопасные трассы – это все неразрушающие трассы, то есть они не зависят от ***safe by***. А безопасные кнопки – это, прежде всего, неразрушающие кнопки, то есть безопасные по $\boldsymbol{safe}_{\gamma\Delta}$**.** То есть $\boldsymbol{safe}_{\gamma\Delta}$ определяет все ограничения модели, налагаемые на возможное отношение ***safe by***.

►В сухом остатке нам нужны трассы, которые мы будем называть *финальными*. Это, во-первых, неразрушающие трассы модели, и, во-вторых, их продолжения дивергенцией или действием, за которым следует разрушение.

►И вот здесь выясняется, что множество финальных трасс модели не является трассовой моделью. Такое множество мы будем называть *финальной моделью*. Её характеристические свойства такие: 1) все свойства трассовой модели, кроме замкнутости, 2) дополнительные два свойства: финальность и финальная замкнутость.

►*Финальность* означает, что все трассы финальной модели финальны.
Финальная замкнутость означает, что удаление отказа из безопасной трассы либо оставляет трассу безопасной и сохраняет все опасные после неё **R**-кнопки, как и положено при замкнутости, либо делает трассу опасной.

►Поскольку финальная модель не является **R**-моделью, она, в частности, не может быть представлено как множество **R**-трасс LTS. Зато мы можем ее представить как множество простых трасс RTS. Такую RTS будем называть *финальной*. Она уже содержит некий минимум трасс, необходимый для определения безопасных трасс, отношения ***safe by***, гипотезы о безопасности и безопасной конформности.

►Оценка проверки финальности RTS, то есть того, что, если взять ***d***-замыкание множества простых трасс RTS, а потом взять его финальные трассы, то получится исходное множество простых трасс RTS. Эта оценка линейна по отношению к числу переходов при заданном числе **R**-кнопок.

4. Генерация тестов

Слайд 13.4. Генерация тестов.
Определение теста

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 $u \in P \vee u = P \in \boldsymbol{R}$.

►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.

►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.

Слайд 14.4. Генерация тестов.
Полный набор тестов

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

►Полный набор тестов всегда существует, в частности, им является набор всех *примитивных* тестов [2]. Примитивный тест строится по одной выделенной немаксимальной безопасной ***R***-трассе спецификации. Для этого в трассу вставляются кнопки, которые оператор должен нажимать: перед каждым отказом R вставляется кнопка R, перед каждым действием z – какая-нибудь безопасная (после соответствующего префикса трассы) кнопка P, разрешающая действие z, а после всей трассы вставляется любая безопасная после нее кнопка P`. По одной безопасной трассе спецификации можно сгенерировать, вообще говоря, несколько разных примитивных тестов, выбирая разные кнопки. Однако множества тестов, сгенерированных по разным трассам, не пересекаются.

Если наблюдение, полученное после нажатия кнопки, продолжает трассу, тест продолжается (немаксимальная в тесте история). Наблюдение, полученное после нажатия последней кнопки, и любое наблюдение, «ответвляющееся» от трассы, всегда заканчивают тестирование (максимальная в тесте история). Вердикт ***pass*** выносится, если полученная ***R***-трасса (подтрасса максимальной истории) есть в спецификации, а вердикт ***fail*** – если нет.

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

Любой строгий тест (как множество историй) равен объединению некоторого множества примитивных тестов, то есть они обнаруживают те же самые ошибки. Поэтому в теории можно ограничиться рассмотрением только примитивных тестов.

Слайд 15.4. Генерация тестов.
Глобальное тестирование

Как уже было сказано, в каждый момент времени реализация может выполнять любое определённое в ней и разрешённое оператором внешнее действие, а также определённые и всегда разрешённые внутренние действия. Если таких действий несколько, выбирается одно из них недетерминированным образом. Для полноты тестирования мы должны проверить все возможные варианты недетерминированного поведения реализации. Прежде всего, для этого требуется, чтобы число таких вариантов было не более, чем счётно. Для этого нужно, чтобы алфавит действий и множество состояний реализации были не более, чем счётными.

►Основное предположение о недетерминизме реализации сводится к тому, что недетерминизм – это явление того уровня абстракции, которое определяется семантикой взаимодействия. Иными словами, поведение реализации, на самом деле, детерминировано, но однозначно определяется некими не учитываемыми нами факторами – «погодными условиями».

►Для того чтобы тестирование могло быть полным, мы должны предположить, что любые погодные условия могут быть воспроизведены в тестовом эксперименте, причём для каждого теста. Заметим, что нас интересуют лишь неэквивалентные погодные условия, то есть такие, которые определяют различное поведение реализации. Если такая возможность есть, то при бесконечной последовательности прогона теста будут воспроизведены все возможные погодные условия с точностью до их эквивалентности.

Такое тестирование называется *глобальным*. Без этого мы не можем быть уверены в полноте тестирования.

►Если гипотеза о глобальном тестировании верна, то для того, чтобы полный набор тестов можно было прогонять при всех возможных погодных условий, набор тестов должен быть перечислим. Поскольку по одной трассе можно сгенерировать много тестов, что определяется имеющимися кнопками, для перечислимости набора тестов дополнительно требуется перечислимость множеств кнопок ***R*** и ***Q***.

►Тогда тесты прогоняются по мере их перечисления таким образом, чтобы каждый перечисленный тест прогонялся неограниченное число раз в «диагональном» процессе перечисления тестов и последовательности прогонов каждого теста. Поскольку все тесты конечны, каждый из них заканчивается через конечное время, после чего выполняется рестарт системы, и прогоняется следующий тест или повторно один из уже перечисленных тестов.

►Если реализация неконформна, то полнота набора тестов гарантирует обнаружение ошибки через конечное время. Однако конформность реализации не может быть обнаружена за конечное время, если набор тестов бесконечен или глобальное тестирование требует не конечного, а бесконечного числа прогонов тестов. Но это уже проблема практического тестирования.

Слайд 16.4. Генерация тестов.
Рестарт

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

►Можно отметить, что последовательность прогонов тестов в «диагональном процессе», чередующихся с рестартом системы, можно рассматривать как один тест, в котором, кроме тестовых воздействий, может встречаться рестарт системы.

►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.

Пример существует даже, когда спецификация и реализации сильно-связны.

►Кроме того, во всех случаях предполагается, что рестарт системы всегда выполняется правильно (система действительно «сбрасывается» в начальное состояние), и его не нужно тестировать.

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

5. Оптимизация тестов

Слайд 17.5. Оптимизация тестов.
Вычисляемые отказы

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: найти конечный полный набор примитивных тестов, если такой существует.

►Например, вычисляемые отказы:
После наблюдения цепочки отказов не надо нажимать кнопку *P*, вложенную в объединение этих отказов, так как наблюдаться будет только отказ *P*. Его можно просто вычислить.

►Без потери полноты тестирования мы можем удалить из набора тестов все тесты, сгенерированные по трассам, в которых есть вычисляемые отказы.

Слайд 18.5. Оптимизация тестов.
Неактуальные трассы

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.

►Прежде всего, неактуальны несогласованные тестовые трассы, то есть когда действие, разрешаемое кнопкой, запрещается предшествующими отказами.

►Однако, это не единственный случай неактуальности. В спецификации могут быть согласованные, но неактуальные тестовые и даже безопасные трассы. Это показывается двумя примерами.

Трасса, состоящая из одного отказа $\{a,b\}$, согласована, но в обоих примерах 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 самом начале в спецификации.

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 19.5. Оптимизация тестов.
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.

►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 ⟨δ, x, δ, x⟩ 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.

▶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⟩.

▶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⟩.

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, вообще говоря, 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.

Слайд 20.5. Оптимизация тестов.
Пополнение спецификации

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

►Спецификация ***S`*** «заменяет» спецификацию ***S***, если она 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***. Здесь мы предполагаем, что заменяющая спецификация может быть задана в другой семантике с другим алфавитом и с другим отношением безопасности кнопок ***safe by***. Иными словами, формально следовало бы говорить о заменяющих не спецификационных моделях, а о заменяющих спецификационных тройках: семантика взаимодействия, спецификационная модель и отношение ***safe by***.

►Для того, чтобы сохранить возможности тестирования реализаций в алфавите **L**, **R`/Q`**-семантика заменяющей спецификации 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**.

►Пополнением будем называть преобразование спецификации в эквивалентную спецификацию, которая конформна сама себе и, следовательно, в ней нет неактуальных и неконформных трасс.

►Наличие неконформных трасс спецификации позволяет говорить об ошибках двух родов. Определим ошибку как продолжение конформной безопасной трассы спецификации безопасным после неё наблюдением, которое
(*ошибка 1-го рода*) отсутствует в спецификации или
(*ошибка 2-го рода*) имеется в спецификации, но делает трассу неконформной.
После пополнения ошибки 2-го рода становятся ошибками 1-го рода, а некоторые ошибки 1-го рода исчезают (вторичные – после ошибки 2-го рода).

►В общем случае не существует пополнения *в той же самой* семантике. Существуют примеры таких семантик и спецификаций. Но можно построить пополнение в другой, расширенной семантике, в которой в каждую кнопку добавлено новое действие, не принадлежащее другим кнопкам. Тестирование тех реализаций, которые определены в алфавите ***L***, в новой семантике эквивалентно их тестированию в старой семантике.

►В то же время для некоторых семантик, в частности, для ***ioco***-семантики удается сделать пополнение в той же семантике.

Слайд 21.5. 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.

►При генерации тестов происходит перечисление безопасных трасс и фильтрация получающихся 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***.

►***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.

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 генерировать тест по максимальному префиксу трассы, после которого нет однозначности вердикта по этой кнопке. 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.

►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.

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

Слайд 22.5. Оптимизация тестов.
Отношение следования ошибок

Тест *обнаруживает* ошибку, если она является трассой некоторой истории теста.

►Определим отношение *следования* ошибок: из ошибки a *следует* ошибка b, если в любой безопасно-тестируемой реализации, где есть трасса a, есть и трасса b. Это отношение, очевидно, является предпорядком, то есть рефлексивно и транзитивно.

► Пример следования ошибок – вычисляемые отказы: если трасса ошибки содержит такой отказ, то трасса без отказа тоже ошибка, и эти ошибки эквивалентны по следованию.

►Набор тестов полный, если он обнаруживает хотя бы одну ошибку из каждого минимального (по предпорядку) класса эквивалентности ошибок.

Из набора примитивных тестов можно удалить любой поднабор тестов, если все оставшиеся тесты обнаруживают ошибки, эквивалентные всем ошибкам из минимальных классов эквивалентности, которые обнаруживают удаляемые тесты. Полнота тестирования при таком удалении сохраняется.

►Множественное следование: $A \Rightarrow B$, если в любой безопасно-тестируемой реализации, где есть $a \in A$, есть некоторое $b \in B$. Это тоже предпорядок.

Значимый набор тестов: мн-во обнаруживаемых им ошибок ⇒ мн-во всех ошибок.
Исчерпывающий набор тестов: мн-во обнаруживаемых им ошибок ⇐ мн-во всех ошибок.
Полный набор тестов: мн-во обнаруживаемых им ошибок ⇔ мн-во всех ошибок.

►Оптимизация заключается в поиске множества ошибок, которое эквивалентно множеству всех ошибок, и которое минимально по вложенности среди всех таких множеств ошибок.

С практической точки зрения нас интересует вопрос: есть ли среди таких минимальных множеств ошибок конечные? И, если есть, как их найти? Есть гипотеза о том, что, если существует конечное минимальное множество ошибок, то все минимальные множества ошибок конечные.

В настоящее время этот раздел теории находится в стадии разработки. Уточняются необходимые и достаточные условия следования ошибок, то есть такие условия, которые можно было бы легко проверять по спецификации в процессе генерации тестов.

Что здесь самое важно для практического тестирования?
То, что полный набор тестов, построенный с учётом следования ошибок, может быть конечным, а без такого учёта – бесконечным.

Слайд 23.5. Оптимизация тестов.
Гипотеза о трёх оптимизациях

ГИПОТЕЗА: Если существует конечный полный набор тестов для класса всех безопасно-тестируемых реализаций, то его можно построить с помощью трех оптимизаций, основанных на
1) удалении неконформных трасс (пополнение),
2) учете демонических трасс,
3) множественном следовании ошибок: выделении конечного минимального по вложенности множества ошибок, эквивалентного множеству всех ошибок.

►ДОПОЛНИТЕЛЬНАЯ ПРАКТИЧЕСКАЯ ГИПОТЕЗА: Для конечной спецификации в конечной семантике это можно сделать алгоритмически за конечное время.

Слайд 24.5. Оптимизация тестов.
Тотальное тестирование

До сих пор мы предполагали, что тестирование нацелено только на обнаружение конформности или неконформности реализации. Такое тестирование мы называли полным. Для полноты тестирования достаточно найти хотя бы одну ошибку в неконформной реализации. В то же время тестирование является лишь этапом в жизненном цикле разработки целевой системы. За тестированием обычно следует фаза исправления ошибок (в реализации, а иногда и в спецификации) и повторное тестирование. Поэтому для уменьшения числа итераций жизненного цикла тестирование должно обнаруживать как можно больше ошибок, а также ситуаций, где ошибок нет, чтобы предоставить разработчику как можно больше информации. Тестирование, которое обнаруживает все имеющиеся в реализации ошибки, и соответствующий набор тестов будем называть *тотальными*. Полное тестирование выполняется «до первой ошибки», а тотальное – пока не будут обнаружены все имеющиеся ошибки. Очевидно, что тотальное тестирование является полным, но обратное, вообще говоря, не верно.

►Набор всех примитивных тестов не только полный, но и тотальный, если продолжать тестирование после обнаружения ошибки. Правда, без анализа неконформных трасс обнаруживаются все ошибки только 1-го рода.

►В общем случае набор тестов тотальный, если он обнаруживает хотя бы одну ошибку из каждого (по предпорядку) класса эквивалентности ошибок, а не только из минимальных классов как при полном тестировании.

Примером тотального набора тестов может служить набор всех примитивных тестов, если мы хотим обнаруживать все ошибки 1-го рода. Если же мы хотим обнаруживать все ошибка как 1-го, так и 2-го родов, то в наборе всех примитивных тестов мы должны установить

вердикт ***fail*** вместо вердикта ***pass*** при обнаружении ошибки 2-го рода.

Из тотального набора примитивных тестов можно удалить любой поднабор тестов, если все оставшиеся тесты обнаруживают ошибки, эквивалентные всем ошибкам, которые обнаруживают удаляемые тесты. Тотальность тестирования при таком удалении сохраняется.