И. Бурдонов, А. Косачев
Семантики взаимодействия с отказами, дивергенцией и разрушением
8-ая Российская конференция с международным участием «Новые информационные технологии в исследовании сложных структур», 5-8 октября 2010 г. Томск
47 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, следующие три – практическому тестированию. В конце я скажу 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 $\boldsymbol{R}\cap\boldsymbol{Q}=\varnothing$ 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 *стационарностью* (*quiescence*). 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)

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.

►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).

►«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 разрушения, попыток выхода из дивергенции и ненаблюдаемых отказов, 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. 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 свойств. Например, трасса должна быть *согласованной*: после отказа P не может следовать действие $z \in P$, а также символы Δ и γ. 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.

Хотя трассовая модель наименее избыточна с точки зрения верификации конформности, она не очень удобна для практического использования. Фактически, LTS 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 $s—z→s`$ 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.

Слайд 8. 2. Модели.
RTS-модель

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, а бесконечная цепочка τ-переходов $s—\tau→\ldots$ заменена Δ-переходом $s—\Delta→$.

►Множество простых трасс RTS не является трассовой моделью, но его замыкание по операции удаления отказов является трассовой ***R***-моделью.

►Дополнительно будем предполагать, что все безопасные трассы этой ***R***-модели являются простыми трассами RTS.

Для любой ***R***-модели существует RTS с такими свойствами. Ее можно получиться следующим преобразованием.

►Состояниями RTS становятся множества состояний LTS в конце ***R***-трасс, что аналогично известному алгоритму «детерминизации» порождающего автомата (или графа).

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 эквивалентен тому, что замыкание множества простых трасс по операции удаления отказов является трассовой ***R***-моделью. И наоборот: для любой трассовой ***R***-модели существует такая RTS, что замыкание множества ее простых трасс по операции удаления отказов является этой трассовой ***R***-моделью.

►С этой точки зрения RTS-модель и трассовая модель эквивалентны. Тем самым, все три модели: LTS, трассовая модель и RTS – эквивалентны.

3. Гипотеза о безопасности и безопасная конформность

Слайд 9. 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, LTS- и RTS-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 (***safe in***) означает, во-первых, что кнопка *P* после трассы σ *неразрушающая*: ее нажатие не может означать попытку выхода из дивергенции (трасса не продолжается дивергенцией) и не может вызывать разрушение (после действия, разрешаемого кнопкой), и, во-вторых, нажатие кнопки не может привести 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*** σ & $(P \in \boldsymbol{Q} \Rightarrow \sigma\cdot\langle P\rangle \notin \boldsymbol{I})$.

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

Слайд 10. 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 $\langle\gamma\rangle$, 2) каждый символ трассы безопасен после непосредственно предшествующего ему префикса трассы:
$\langle\gamma\rangle \notin \boldsymbol{I}$ & $\forall \mu\ \forall u$ ($\mu \cdot \langle u \rangle$ префикс $\sigma \Rightarrow u$ ***safe in I after*** μ),
$\langle\gamma\rangle \notin \boldsymbol{S}$ & $\forall \mu\ \forall u$ ($\mu \cdot \langle u \rangle$ префикс $\sigma \Rightarrow u$ ***safe by S after*** μ).

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

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 11. 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 $=_{\mathbf{def}}$ ($<\gamma>\notin \boldsymbol{S} \Rightarrow <\gamma>\notin \boldsymbol{I}$) & $\forall\sigma\in \boldsymbol{SafeBy}(\boldsymbol{S})\cap \boldsymbol{I}$ $\forall P\in \boldsymbol{R}\cup\boldsymbol{Q}$ (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*** & $\forall\sigma\in \boldsymbol{SafeBy}(\boldsymbol{S})\cap \boldsymbol{I}$ $\forall P$ ***safe by S after*** σ $\boldsymbol{obs}(\sigma,P,\boldsymbol{I})\subseteq \boldsymbol{obs}(\sigma,P,\boldsymbol{S})$,
где $\boldsymbol{obs}(\sigma,P,\boldsymbol{M}) =_{\mathbf{def}} \{u \mid \sigma\cdot<u>\in \boldsymbol{M}$ & $(u\in P \vee u=P$ & $P\in \boldsymbol{R})\}$ – множество

наблюдений, которые можно получить над полной трассовой моделью ***M*** при нажатии кнопки *P* после трассы σ.

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.

4. Генерация тестов

Слайд 12. 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 трасса $\sigma \cdot \langle u \rangle$, где трасса σ безопасна в спецификации, а наблюдение *u* безопасно в спецификации после σ (но не обязательно продолжает σ в спецификации).

Слайд 13. 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 (по отношению «является префиксом») 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 можно ограничиться рассмотрением только примитивных тестов.

Слайд 14. 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 15. 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.

►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 16. 5. Оптимизация тестов.
Вычисляемые отказы

Не все примитивные тесты нужны для полноты тестирования.

►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 отказами.

Слайд 17. 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 18. 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 $\langle \delta, x, \delta \rangle$ – на рисунке сверху – 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⟩ – на рисунке внизу, а реакция *b* не конформна после подтрассы ⟨δ, x, x⟩ – на рисунке в середине, 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 19. 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 эквивалентным преобразованием спецификации.

►Спецификации ***S*** и ***S`*** *эквивалентны*, если они определяют одинаковые классы безопасно-тестируемых и конформных реализаций в алфавите ***L***. То есть они определяют одинаковые пересечения классов безопасно-тестируемых реализаций и, соответственно, классов конформных реализаций с классом реализаций в заданном алфавите ***L***. Здесь мы предполагаем, что эквивалентные спецификации могут быть заданы в разных семантиках с разными алфавитами и с разными отношениями безопасности кнопок ***safe by***. Иными словами, формально следовало бы говорить об эквивалентности не спецификационных моделей, а спецификационных троек: семантика взаимодействия, спецификационная модель и отношение ***safe by***.

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

►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 20. 5. Оптимизация тестов.
Отношение следования ошибок

Наличие неконформных трасс спецификации позволяет говорить об ошибках двух родов. Определим ошибку как продолжение конформной безопасной трассы σ спецификации безопасным после неё наблюдением *u*, которое
(*ошибка 1-го рода*) отсутствует в спецификации или
(*ошибка 2-го рода*) имеется в спецификации, но делает трассу $\cdot\sigma\langle u\rangle$ неконформной.

►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.

►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.

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.

Слайд 21. 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.

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

Слайд 22 - План доклада. 2 часть.

6. Проблемы практического тестирования

Слайд 23. 6. Проблемы практического тестирования.
Проблема конечности

Для практического применения конечными по времени должны быть как генерация тестов, так и тестирование по этим тестам.

►Отсюда вытекает конечность полного набора тестов (или единого теста с рестартом). Для этого «почти» необходимы, хотя и недостаточны, конечность алфавита внешних действий ***L*** и конечность LTS-спецификации (числа ее переходов), что на практике вполне приемлемо.

►Конечность тестирования опирается на конечность полного набора тестов, конечность времени прогона каждого теста и конечность требуемого числа прогонов каждого теста.

►Конечность времени прогона теста гарантируется для конечного теста «практическими предположениями» о семантике: 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) в реализацию и наблюдения от реализации выполняются за конечное время. Эти предположения гарантируют наблюдение внешнего действия, выполняемого реализацией, через конечное время после нажатия кнопки, разрешающей это действие.

►Таким образом, остаются две основные проблемы: 1) конечность полного набора тестов и 2) конечность требуемого числа прогонов теста.

Эти проблемы не имеют решения в общем виде, поэтому такие решения приходится искать в частных случаях: либо ограничивая классы рассматриваемых спецификаций и/или реализаций, либо предполагая наличие дополнительных тестовых возможностей, либо сочетая одно с другим.

Слайд 24. 6. Проблемы практического тестирования.
Тестирование с открытым состоянием

Одной из таких возможностей является опрос текущего состояния реализации. Если можно «подсматривать» состояния реализации, говорят о тестировании с открытым состоянием, в противном случае – о тестировании с закрытым состоянием. Дальше мы рассмотрим эти два вида тестирования и соответствующие ограничения на классы реализаций и/или спецификаций.

Слайд 25. 6. Проблемы практического тестирования.
Ограничения на недетерминизм реализации

Проблема конечности требуемого числа прогонов теста общая для этих двух видов тестирования. Гипотеза о глобальном тестировании дает только теоретическую возможность обнаружить любую ошибку в любой неконформной реализации. На практике нам, прежде всего, требуется конечность различимых «погодных условий», а также либо какие-то способы «управления погодой», либо гипотезы, ограничивающие возможные проявления недетерминизма реализации.

▶Первое возможно в каких-то частных случаях, например, когда недетерминизм является следствием псевдопараллелизма, то есть псевдопараллельного выполнения нескольких детерминированных процессов на одном процессоре. Если мы можем вмешиваться в работу планировщика, мы тем самым можем «управлять погодой».

▶Второй способ используется чаще всего в его экстремальном виде, когда ограничиваются только детерминированными реализациями (при недетерминированной спецификации).

▶Здесь мы должны уточнить понятие детерминизма. Обычно LTS-реализация определяется как детерминированная, если каждая ее трасса заканчивается ровно в одном состоянии. Однако в общем случае для тестирования в ***R/Q***-семантике дополнительно требуется, чтобы в каждом достижимом состоянии для каждой кнопки P было определено не более одного перехода по действию $z \in P$. В терминах

тестов это можно сформулировать так: безопасно-тестируемая LTS-реализация детерминирована в ***R***/***Q***-семантике для заданной спецификации, если каждый тест при любом его прогоне либо заканчивается только с вердиктом ***fail***, быть может, в разных состояниях реализации, либо только с вердиктом ***pass*** и только в одном состоянии реализации.

►В дальнейшем мы исходим из следующей гипотезы о *t-недетерминизме*: если в любом состоянии *i* реализации любую кнопку *P* нажимать *t* раз, то реализация продемонстрирует все возможные варианты поведения, то есть будут получены все возможные пары (наблюдение, постсостояние). При *t*=1 мы имеем детерминированную реализацию.

Слайд 26. 6. Проблемы практического тестирования.
Ограниченное отношение безопасности кнопок ***safe by***

Остановимся на одном независимом аспекте проблемы конечности полного набора тестов: проблеме определения безопасных трасс в спецификации. Дело в том, что правила для отношения безопасности кнопок ***safe by*** позволяют после различных трасс, заканчивающихся в LTS-спецификации в одном множестве состояний (в детерминированной RTS-спецификации – в одном состоянии), по-разному определять безопасные кнопки.

►При наличии циклов мы получаем бесконечную цепочку конечных трасс с произвольным распределением безопасных кнопок после этих трасс.

►Отношение ***safe by***, при котором трассы, заканчивающиеся в одном множестве состояний LTS-спецификации (в одном состоянии RTS-спецификации), имеют одинаковые множества безопасных после них ***Q***-кнопок (и, тем самым, всех кнопок, поскольку безопасность ***R***-кнопок одинакова для всех таких трасс), будем называть *ограниченным*.

►Это дает нам возможность говорить о безопасности кнопки (и соответствующих наблюдений) во множестве состояний LTS-спецификации (или в одном состоянии RTS-спецификации).

Множество безопасных трасс спецификации ***S*** будем обозначать ***Safe***(***S***).

►Важно, что для LTS-спецификации с конечным числом состояний конечно число множеств состояний (и состояний соответствующей RTS-спецификации).

7. Тестирование с закрытым состоянием

Слайд 27. 7. Тестирование с закрытым состоянием.
Ограничение на семантику и спецификацию

Решение проблемы конечности полного набора тестов будем искать двумя способами: сужая классы рассматриваемых спецификаций или реализаций. Для конечной семантики по каждой трассе генерируется конечное число примитивных тестов (тестов, сгенерированных по одной трассе спецификации). Поэтому достаточно конечности полного набора трасс, что для конечной семантики эквивалентно ограниченности длины трасс полного набора.

►**Ограничение на спецификацию.** Если число ***R***-кнопок конечно, а в конечной спецификации нет циклов, то число трасс спецификации конечно, тем более конечен полный набор трасс. В то же время не любой цикл приводит к бесконечности полного набора трасс.

►Для RTS-спецификации будем называть *демоническим* состояние *s*, после которого не бывает ошибок, то есть любая актуальная (встречающаяся в безопасно-тестируемых реализациях) тестовая трасса $\mu\cdot\lambda$, где трасса μ заканчивается в *s*, не является ошибкой.

►Для того, чтобы для конечной спецификации в конечном алфавите существовал полный набор трасс ограниченной длины необходимо и достаточно, чтобы в RTS-спецификации любой маршрут с безопасной трассой не содержал цикла, проходящего через недемонические состояния. Длина трасс такого полного набора имеет точную верхнюю оценку *K*, где *K* число состояний RTS-спецификации.

Слайд 28. 7. Тестирование с закрытым состоянием.
Ограничение на размер реализации

Ограничение на размер реализации. Путь задана RTS-спецификация ***S*** и рассмотрим некоторую произвольную безопасно-тестируемую LTS-реализацию ***I***.

Пусть σ безопасная трасса спецификации, а *P* - кнопка, безопасная в спецификации после трассы σ.

Обозначим через s_σ (единственное) состояние спецификации, в котором заканчивается трасса σ, а через i_σ – одно из состояний реализации ***I***, в которых заканчивается трасса σ.

►Если для каждой тройки (s_σ, i_σ, P) в наборе тестов будет примитивный тест, сгенерированный по какой-нибудь трассе σ` такой, что $s_\sigma = s_{\sigma`}$, в котором после трассы σ` нажимается кнопка *P*, то такой набор тестов, очевидно, будет полным для ограниченного отношения ***safe by***.

►При однократном прогоне теста на реализации ***I*** мы получаем последовательность пар состояний (s_μ, i_μ), где μ - префикс трассы σ.

►Будем говорить, что тест *простой* для данной реализации ***I***, если хотя бы при одном его прогоне в полученной последовательности пар состояний нет одинаковых пар.

Очевидно, что для полноты тестирования данной реализации ***I*** достаточно тестов, которые простые для этой реализации. Для полноты тестирования всех реализаций достаточно, чтобы в набор тестов входили все тесты, которые просты для той или иной реализации.

Слайд 29. 7. Тестирование с закрытым состоянием.
Оценка длины простого теста

Оценим сверху длину *N* простого теста.

Если число состояний реализации не превосходит n, то, очевидно, $N \le nK$.

Если набор тестов состоит из всех тестов длины не больше nK, то такой набор тестов содержит все простые тесты для всех реализаций с числом состояний не больше n.

►Эта верхняя оценка является точной по порядку: для любых k и n существует семантика с $O(k)$ действиями, LTS-спецификация с $O(k)$ состояниями и неконформная реализация с $O(n)$ состояниями, ошибка в которой не может быть обнаружена тестом длины меньше $O(n2^k)$.

►Можно показать, что даже для семантики с ограниченным число действий оценка остается суперполиномиальной.

В частности, существует семантика с двумя действиями, для которой в точной верхней оценке показатель степени k заменяется на $c(\ln k)^2$.

►LTS-спецификации, на которых достигаются эти оценки, существенно недетерминированы.

Понятно, что для детерминированной LTS-спецификации $K=k$ (а не 2^k по порядку).

Однако даже минимальный недетерминизм LTS-спецификации, когда нет τ-переходов и только в одном состоянии только по одному действию имеются два перехода в разные состояния, оставляет оценки суперполиномиальными.

Слайд 30. 7. Тестирование с закрытым состоянием.
Ограничение на недетерминизм реализации

►**Ограничение на недетерминизм реализации.** При тестировании с закрытым состоянием нужна усиленная гипотеза о t-недетерминизме: нас интересует число $T(\sigma,P)$ возможных поведений реализации при нажатии кнопки P не в состоянии реализации (которого мы не видим), а после трассы σ.

Если число состояний реализации ограничено числом n, то $T(\sigma,P) \le t^n$.

Но можно использовать и независимую от n гипотезу о том, что $T(\sigma,P)$ ограничено некоторым числом T.

Такую гипотезу мы вынуждены использовать, если ограничения налагаются только на спецификацию, но не на размер реализации.

►Число прогонов примитивного теста с N кнопками равно $O(T^N)$.

Эта оценка достижима для некоторых реализаций, но, конечно, для каких-то реализаций она может оказаться гораздо меньше.

В частности, для детерминированных реализаций оценка $O(N)$.

8. Тестирование с открытым состоянием

Слайд 31. 8. Тестирование с открытым состоянием.
Один адаптивный тест с рестартом

Для тестирования с открытым состоянием мы будем строить не набор тестов, а один адаптивный тест с возможными рестартами в середине теста.

►Для полноты тестирования нужно проверить все переходы реализации, лежащие на маршрутах с трассами, безопасными в спецификации.

►Для этого, поскольку у нас только один тест, LTS-реализация должна быть сильно-связной: из каждого состояния достижимо по переходам каждое другое.

Рестарт понимается как одно из внешних действий, отличающихся только тем, что гарантированно переводит реализацию в начальное состояние.

Переход по рестарту делает трассу пустой.

Переходы по рестарту дополнительно учитываются при определении сильно-связности.

►Также требуется, чтобы начальное состояние реализации было стабильным или хотя бы в одном состоянии, достижимом по безопасной трассе спецификации, был определен рестарт.

►Действительно, в примере реализация в начале тестирования может перейти в состояние 1. После этого без рестарта она попадет в состояние 0 только после трассы <*x*>, после которой в спецификации кнопка {*y*} опасна: ее нельзя нажимать при тестировании. Тем самым ошибка в реализации не будет обнаружена.

Слайд 32. 8. Тестирование с открытым состоянием.
Структуры данных для RTS-спецификации ***S***

Предварительно строятся структуры для спецификации ***S***, которые не зависят от реализации и используются без модификации для верификации любой реализации в той же ***R***/***Q***-семантике.

Будем предполагать, что спецификация ***S*** задана как RTS.

►Рассматриваются состояния *s* в конце безопасных трасс, для каждого из которых определяем:

A(*s*) — множество кнопок, безопасных в состоянии *s*;

B(*s*) — множество безопасных наблюдений плюс символ τ;

C(*s*,*u*) – постсостояние после перехода по наблюдению *u* из состояния *s*.

Если таких переходов нет, ***C***(*s*,*u*)=∗, где ∗ не совпадает ни с одним состоянием спецификации.

Доопределим ***C***(*s*,τ)=*s*.

Слайд 33. 8. Тестирование с открытым состоянием.
Построение модели реализации ***I*** *в процессе тестирования*

Получая наблюдения и опрашивая состояния реализации, мы будем поэтапно строить LTS-реализацию с одновременной проверкой тестируемого условия.

Более точно: строится LTS-модель, имеющая такое же как в реализации множество трасс, безопасных в спецификации, и такое же множество состояний, достижимых по этим трассам.

►В начале тестирования и после каждого перехода опрашиваем состояние реализации.

Переход $i—z\rightarrow i`$ по внешнему действию $z$ добавляется, когда после опроса состояния $i$ нажимается кнопка $P$, после чего наблюдается действие $z\in P$, а затем опрашивается постсостояние $i`$.

Если наблюдается отказ с *тем же самым* постсостоянием $i`=i$, то добавляется виртуальный переход-петля по отказу $i—P\rightarrow i$.

Если отказ P наблюдается с *другим* постсостоянием $i`\neq i$, то добавляются переходы $i—\tau\rightarrow i`—P\rightarrow i`$.

Вместе с каждым переходом по внешнему действию $i—z\rightarrow i`$ будем хранить кнопку, нажатие которой вызвало этот переход.

Слайд 34. 8. Тестирование с открытым состоянием.
Структуры данных для LTS-реализации ***I*** *(1)*

При построении реализации с каждым построенным состоянием i будем связывать множество $\boldsymbol{S}(i)$. Это состояния спецификации в конце трасс, безопасных в спецификации, имеющихся в реализации и заканчивающихся там в состоянии i.

В процессе тестирования мы будем строить эти множества $\boldsymbol{S}(i)$, постепенно добавляя в них новые состояния.

►Кнопка P *допустима в* i, если она безопасна хотя бы в одном состоянии $s\in\boldsymbol{S}(i)$.

Только допустимые кнопки будут нажиматься в состоянии i.

Для каждой допустимой кнопки P определим счётчик $c(P,i)$ числа ее нажатий в состоянии i.

Кнопка P *полна в состоянии* i, если
1) счетчик=1 и в состоянии i есть виртуальный переход-петля по отказу P, или
2) счетчик=t.

В обоих случаях уже получены все возможные переходы из состояния i при нажатии кнопки P.

Состояние *полно*, если каждая допустимая в нем кнопка полна.

Слайд 35. 8. Тестирование с открытым состоянием.
Структуры данных для LTS-реализации ***I*** *(2)*

Кроме $\boldsymbol{S}(i)$ формируются следующие структуры данных для каждого состояния i:
$\boldsymbol{A}(i)=\cup\{\boldsymbol{A}(s)|s\in\boldsymbol{S}(i)\}$ — множество кнопок, допустимых в состоянии i;
$\boldsymbol{D}(i)=\{i—z\rightarrow i`\}$ — множество переходов из состояния i.

▶В начале тестирования после опроса состояния в ***I*** есть только одно состояние $i\in(\boldsymbol{I}\ \textbf{\textit{after}}\ \varepsilon)$, где ε пустая трасса, и для этого состояния $\boldsymbol{S}(i)=\{s_0\}$, $\boldsymbol{A}(i)=\boldsymbol{A}(s_0)$, $\boldsymbol{D}(i)=\varnothing$ и $c(P,i)=0$ для каждой допустимой кнопки P.

▶Тестируемое условие эквивалентно следующему условию в терминах введенных обозначений. Мы будем проверять это условие на каждом шаге тестирования.

Слайд 36. 8. Тестирование с открытым состоянием.
Общая схема работы алгоритма тестирования (1)

На рисунке изображена общая схема работы алгоритма.
Сначала проверяем полноту текущего состояния.

▶Если текущее состояние полное, то для продолжения тестирования нужно перейти в неполное состояние.

►Если таких состояний нет, алгоритм заканчивается с вердиктом ***pass***.

►Рассмотрим переход в неполное состояние.

Слайд 37. 8. Тестирование с открытым состоянием.
Общая схема работы алгоритма тестирования (2)

В графе LTS реализации всегда можно выбрать множество деревьев, покрывающих все состояния так, что из каждого состояния выходит не более одного перехода, принадлежащего деревьям, и все эти деревья ориентированы к своим корням, которыми являются все неполные состояния. Будем двигаться по переходам этих деревьев, нажимая управляющие кнопки этих переходов. Из-за недетерминизма мы можем оказаться не в постсостоянии перехода, а в другом состоянии. Если это не полное состояние, мы снова будем нажимать управляющую кнопку. За конечное число шагов мы гарантированно попадем в неполное состояние.

Слайд 38. 8. Тестирование с открытым состоянием.
Общая схема работы алгоритма тестирования (3)

►Если текущее состояние неполное, то выбираем неполную кнопку, нажимаем ее и получаем один переход или два перехода. Постсостояние перехода становится новым текущим состоянием. Корректируем счетчик нажимавшейся кнопки.

►Переход, который мы получили, может быть новым или построенным ранее.

►Если получен старый переход, возвращаемся к началу работы алгоритма.

►Если получен новый переход, то корректируем множество переходов $\boldsymbol{D}(i)$.

После этого выполняется блок «Распространие с верификацией».

Слайд 39. 8. Тестирование с открытым состоянием.
Общая схема работы алгоритма тестирования (4)

Для работы этого блока создается вспомогательный список $\boldsymbol{W}$ всех пар (состояние спецификации и переход реализации), где состояние спецификации сопоставлено пресостоянию перехода. В самом начале множество $\boldsymbol{W}$ содержит все такие пары для каждого нового перехода.

Опишем шаг работы блока.

Если список $\boldsymbol{W}$ не пуст, выбираем первый элемент из списка, удаляя его из списка.

►Проверяем для него тестируемое условие.

►Если условие не выполнено, фиксируется ошибка и алгоритм заканчивается с вердиктом ***fail***.

►Если ошибки нет, рассматриваем состояние спецификации в конце перехода из состояния s по наблюдению u – $\boldsymbol{C}(s,u)$.

Если $\boldsymbol{C}(s,u)$ уже сопоставлено постсостоянию $j`$ реализационного перехода по u, то выбираем следующий элемент из списка $\boldsymbol{W}$.
В противном случае сопоставляем состояния спецификации и реализации, добавляя $\boldsymbol{C}(s,u)$ в $\boldsymbol{S}(j`)$,
и помещаем в список $\boldsymbol{W}$ все требуемые пары.
Выбираем следующий элемент из списка $\boldsymbol{W}$.

►Когда список $\boldsymbol{W}$ становится пустым, переходим к началу работы.

Слайд 40. 8. Тестирование с открытым состоянием.
Общая схема работы алгоритма тестирования (4)

Отметим, что при таком тестировании верифицируются не только наблюдения, полученные после *реальных* трасс, пройденных при тестировании, но и возможные наблюдения после *потенциальных* трасс, то есть наблюдения и трассы, про которые установлено, что они есть в реализации. Это даёт существенную экономию числа тестовых воздействий, необходимых для проверки конформности: мы

выполняем множество проверок без реального тестирования, основываясь на полученном знании о поведении реализации.

Например, если при тестировании получены две трассы $\mu_1 \cdot \lambda$ и μ_2, где трассы μ_1 и μ_2 заканчиваются в реализации в одном состоянии i, то мы можем проверить обе трассы: как $\mu_1 \cdot \lambda$, которую реально прошли при тестировании, так и потенциальную трассу $\mu_2 \cdot \lambda$.

Это преимущество дает дополнительная тестовая возможность опроса состояния реализации.

Слайд 41. 8. Тестирование с открытым состоянием.
Оценки сложности алгоритма. Число тестовых воздействий

При тестировании обычно наиболее важным считается число тестовых воздействий.
Оценка экспоненциальная в общем случае и полиномиальная для детерминированных реализаций.

►Такая полиномиальная оценка достигается не только для детерминированной реализации, но и во всех случаях, когда переход в неполное состояние можно гарантированно выполнить, проходя *путь* (маршрут без самопересечений), длина которого ограничена n. Приведем три таких случая:

►1. Упомянутый выше случай недетерминизма как следствия псевдопараллелизма, когда мы можем вмешиваться в работу планировщика процессов, тем самым «управляя погодой».

►2. Сильно-Δ-связные LTS-реализации. Это такие LTS, в которых для любой пары состояний a и b можно в каждом состоянии i найти такую кнопку $P(i)$, что, нажимая только такие кнопки, мы гарантированно окажемся в состоянии b, хотя путь из a в b, который мы проходим, зависит от недетерминированного поведения LTS. Детерминированные LTS — это частный случай сильно-Δ-связных LTS.

►3. Недетерминизм может быть следствием повышения уровня абстракции при моделировании детерминированной исследуемой

системы такой реализацией, которая оказывается недетерминированной в той семантике взаимодействия, которая используется в спецификации. При тестировании связь уровней абстракции осуществляется промежуточной программой (медиатором). В этом случае при нажатии в состоянии *i* кнопки *P* медиатор может сообщить тесту некий дополнительный параметр, от которого абстрагируется модель. Если есть возможность при повторном нажатии в состоянии *i* кнопки *P* сообщить медиатору этот параметр, то гарантированно выполнится тот же реализационный переход.

Слайд 42. 8. Тестирование с открытым состоянием.
Оценки сложности алгоритма. Объем вычислений

Оценка объема вычислений, кроме тестовых воздействий, содержит три слагаемых:
1) вычисления, необходимые для поиска опрошенного состояния среди пройденных при каждом тестовом воздействии, имеют экспоненциальную оценку в общем случае и полиномиальную для детерминированных реализаций;
2) построение множества деревьев имеет полиномиальную оценку;
3) вычисления в блоке «Распространение с верификацией» имеют O от m K большое.

9. Развитие теории конформности

Слайд 43. 9. Развитие теории конформности.
Проблема монотонности

Теперь я хочу сказать несколько слов о дальнейшем развитии теории безопасной конформности. Какие-то проблемы здесь уже решены, а какие-то ещё предстоит решить.

Первой такой проблемой является проблема монотонности конформности. Эта проблема возникает в связи с композицией. Композиционная система – это составная система, собранная из компонентов с помощью применения правил параллельной композиции, определенных для LTS-моделей.

Неформально проблема звучит так: если компоненты работают правильно, то почему система в целом работает неправильно?

Формально это означает, что композиция реализаций компонентов, которые конформны спецификациям этих компонентов, не конформна композиции спецификаций компонентов. Это вызвано разными уровнями абстракции, используемыми в определениях конформности и прямой композиции. Конформность основана на трассах наблюдений над поведением реализационной модели, а композиция, кроме того, на ненаблюдаемых напрямую состояниях и ненаблюдаемых действиях (τ-действиях).

Особым случаем композиции является тестирование в контексте, которое можно рассматривать как тестирование системы из двух компонентов, один из которых – реализация, а другой – фиксированная среда взаимодействия. Здесь возможно несохранение конформности, когда ловится «ложная» ошибка, что является частным случаем общей проблемы монотонности.

Решением является такое эквивалентное преобразование спецификаций компонентов, что композиция конформных реализаций компонентов конформна композиции преобразованных спецификаций компонентов. Такое преобразование называется монотонным. Преобразование пополнения, о котором мы говорили, является первым шагом к монотонному преобразованию, но оно недостаточно для монотонности.

При тестировании в контексте предполагается, что в среде нет ошибок и она известна. Поэтому монотонное преобразование нужно применять только к спецификации реализации, а среда остаётся неизменной. При решении проблемы монотонности учитывается и этот случай.

Если спецификация системы задана независимым образом, то она должна быть *согласованной* со спецификациями компонентов так, чтобы композиция конформных реализаций компонентов была конформна этой спецификации системы. Композиция монотонно преобразованных спецификаций оказывается самой сильной согласованной спецификацией системы, то есть предъявляющей к

системе наибольшие требования среди всех согласованных спецификаций.

Здесь возникает вопрос: в какой семантике следует рассматривать композицию монотонно преобразованных спецификаций? Дело в том, что разные компоненты могут быть определены в разных алфавитах и разных семантиках. По счастью, этот вопрос снимается тем, что композиция монотонно преобразованных спецификаций оказывается самой сильной согласованной спецификацией *в любой* семантике. Естественно, для одного и того же композиционного алфавита, однозначно определяемого алфавитами компонентов и не зависящего от семантик компонентов при тех же алфавитах компонентов.

Монотонное преобразование позволяет решить две основные задачи: 1) верификации декомпозиции системных требований, то есть верификация согласованности имеющейся спецификации системы со спецификациями компонентов, и 2) при отсутствии спецификации системы её генерация по спецификациям компонентов.

Я здесь не буду останавливаться на технических деталях: как именно определяется монотонное преобразование, как его сделать алгоритмическим и тому подобное. Это довольно сложное преобразование и рассказ о нем потребовал бы очень много времени.

Слайд 44. 9. Развитие теории конформности.
Приоритеты

Теория, которую я здесь рассказывал, да и вообще все существующие теории конформности работают с системами без приоритетов, когда любое действие, разрешённое оператором и определённое в реализации, может быть недетерминированным образом выбрано на выполнение. В то же время на практике часто встречаются системы с приоритетами. Вот только несколько примеров, показывающих важность и полезность приоритетов.

Выход из дивергенции. Если нет приоритетов, то при возникновении дивергенции дальнейшее взаимодействие с системой проблематично, поскольку она может бесконечно долго выполнять только внутренние

действия. В то же время дивергенция, как бесконечная внутренняя работа, встречается во многих реальных системах, но при возникновении внешнего воздействия система сразу же на него реагирует. Это происходит из-за того, что внешние действия имеют больший приоритет, чем внутренние τ-действия.

Выход из осцилляции. Под осцилляцией понимается бесконечная цепочка выдачи сообщений системой. Для того, чтобы такую цепочку можно было прервать, заставив систему обрабатывать поступающий извне запрос, последний должен иметь больший приоритет, чем выдача сообщений.

Прерывание цепочки действий. Команда «отменить» (cancel) должна останавливать выполнение последовательности действий, инициированной предыдущим запросом, и вызывать цепочку завершающих действий. Если команда «отменить» не имеет наивысшего приоритета, она может быть выполнена уже после того, как вся обработка закончится, то есть, фактически, ничего «не отменяет».

Приоритетная обработка входных воздействий. Если в систему поступает одновременно несколько запросов, то часто требуется их обработка в соответствии с некоторыми приоритетами между ними. Это реализуется в виде очереди запросов с приоритетами или в виде нескольких очередей запросов с приоритетами между очередями. К этому типу приоритетов относится и обработка аппаратных прерываний в операционной системе.

Нам удалось ввести в теорию приоритеты. Конкретно мы ввели приоритеты в LTS-модель и соответствующим образом модифицировали конформность, генерацию тестов. Также удалось определить параллельную композицию систем с приоритетами. Нерешенными остаются проблемы пополнения и монотонного преобразования для систем с приоритетами.

Слайд 45. 9. Развитие теории конформности.
Симуляции

Кроме конформности, основанной только на трассах наблюдений, в литературе рассматриваются разные виды конформностей, основанных на соответствии состояний реализации и спецификации. Такие конформности называются симуляциями. Симуляция требует, чтобы правильным было не только наблюдаемое внешнее поведение реализации, но и изменение ее состояний. Все рассматриваемые в литературе симуляции либо не учитывают безопасности тестирования, предполагая отсутствие дивергенции и ненаблюдаемых отказов, либо предполагают возможность прямого наблюдения дивергенции и всех отказов. Также они не учитывают возможность разрушения.

Разумеется тестирования симуляции возможно только в том случае, когда мы можем опрашивать текущее состояние реализации, то есть это всегда тестирование с открытым состоянием.

Нам удалось определить слабую симуляцию с учетом отказов (как наблюдаемых, так и ненаблюдаемых), дивергенции и разрушения. Мы назвали ее безопасной симуляцией. Установлена связь безопасной симуляции с трассовой конформностью ***saco*.**

Теоретически исследовано полное тестирование безопасной симуляции. В отличие от трассовой конформности полное тестирование не всегда возможно. Найдено достаточное условие полноты тестирования и предложен общий алгоритм тестирования безопасной симуляции.

Для практического использования предложена модификация общего алгоритма, которая аналогична алгоритму тестирования с открытым состоянием, о которой я рассказывал, но только верификация выполняется не по ходу исследования реализация, а после завершения такого исследования.

Нерешенными остаются проблемы пополнения и монотонного преобразования для безопасной симуляции.

Слайд 465. 9. Развитие теории конформности.
Тестирование с преобразованием семантик

В теории конформности обычно предполагается, что реализация и спецификация заданы в одной семантике. Однако на практике часто требуется некоторое преобразование спецификационных тестовых воздействий в реализационные тестовые воздействия и обратное преобразование реализационных наблюдений в спецификационные наблюдения. Программа, осуществляющая эти преобразования, называется медиатором. По сути, это означает, что реализация может быть задана в другой семантике взаимодействия, и медиатор осуществляет преобразование семантик.

В простейшем случае различие семантик только в разных способах представления одних и тех же действий в реализации и спецификации. Достаточно взаимно-однозначного преобразования алфавитов реализации и спецификации. В общем случае такого преобразования недостаточно, поскольку спецификация, как правило, определяется на более высоком уровне абстракции. Медиатор может оказаться довольно сложной программой, осуществляющей медиативные функции между реализацией в одной семантике и тестом, генерируемым по спецификации в другой семантике.

Кроме тестовых воздействий и наблюдений, медиатор может преобразовывать также и состояния: из реализационного в спецификационное. Это означает, что на множестве состояний реализации предполагается заданным отношение эквивалентности, и состояние спецификации соответствует классу эквивалентных состояний реализации. Это соответствие является не предусловием тестирования, а частью проверяемого условия.

Предложена формализация устройства теста и медиатора при тестировании с преобразованием семантик. Соответствующим образом модифицированы гипотеза о безопасности и конформность ***saco***. Дополнительная модификация предложена для медиатора, осуществляющего преобразование состояний.

Для практического использования модифицирован алгоритм конечного полного тестирования с открытым состоянием, в том числе и для случая преобразования состояний в медиаторе.

Не до конца исследованы проблемы пополнения и монотонного преобразования для этого случая.