УДК 681.3.06

Семантики взаимодействия с отказами, дивергенцией и разрушением.

И.Б. Бурдонов, А.С. Косачев

Институт системного программирования РАН (ИСП РАН)

igor@ispras.ru, kos@ispras.ru

Исследуются формальные методы тестирования конформности исследуемой системы спецификации. Семантика взаимодействия определяет тестовые возможности, сводимые к наблюдению действий и отказов (отсутствие действий). Семантика параметризуется семействами наблюдаемых и ненаблюдаемых отказов. Вводится разрушение – запрещённое действие, которого следует избегать при взаимодействии. Определяются понятие безопасного тестирования, реализационная гипотеза о безопасности и безопасная конформность, а также генерация полного набора тестов по спецификации. Исследуются эквивалентность трасс, спецификаций, отношений безопасности и семантик взаимодействия. Предлагается пополнение спецификации, при помощи которого можно удалить из спецификации неактуальные (отсутствующие в безопасно-тестируемых реализациях) и неконформные трассы спецификации. Вводится понятие тотального тестирования, обнаруживающего все ошибки в реализации (а не хотя бы одну как для полного тестирования). На основе анализа зависимостей между ошибками предлагается метод минимизации тестового набора. Исследуется проблема сохранения конформности при композиции (монотонность конформности) и предлагается монотонное преобразование спецификации, решающее эту проблему.

1. Семантика взаимодействия и безопасное тестирование

Статья посвящена проблемам и методам проверки «правильности» программных систем. Под «правильностью» понимается соответствие исследуемой системы заданным требованиям. В модельном мире система отображается в реализационную модель (реализацию), требования – в спецификационную модель (спецификацию), а их соответствие – 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.

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 [1,3,15,16,21]. Она представляет собой «чёрный ящик», внутри которого находится реализация (Рис.1). 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. Наблюдения (на «дисплее» машины)

бывают двух типов: наблюдение некоторого *внешнего (наблюдаемого) действия*, разрешённого оператором и выполняемого реализацией, и наблюдение *отказа* как отсутствия каких бы то ни было наблюдаемых действий из числа тех, что разрешены нажатыми кнопками.

Рис.1. Машина тестирования

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. После наблюдения (действия или отказа) кнопка отжимается, и все внешние действия запрещаются. Далее оператор может нажать другую (или ту же самую) кнопку.

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 **L** и двумя наборами кнопок машины тестирования: с наблюдением соответствующих отказов – семейство $\mathbf{R}\subseteq 2^{\mathbf{L}}$ и без наблюдения отказа – семейство $\mathbf{Q}\subseteq 2^{\mathbf{L}}$. Предполагается, что $\mathbf{R}\cap\mathbf{Q}=\varnothing$ и $\cup\mathbf{R}\cup\cup\mathbf{Q}=\mathbf{L}$. Такую семантику мы называем **R**/**Q**-семантикой. Если $\mathbf{Q}=\varnothing$ и $\mathbf{R}=2^{\mathbf{L}}$, то это хорошо известна *failure trace semantics* [12,15,16,20,23,26]. Еще один пример – это семантика популярного отношения ***ioco*** [25,26], когда действия разбиваются на стимулы (input) и реакции (output) $\mathbf{L}=\mathbf{I}\cup\mathbf{U}$. Есть только одна **R**-кнопка приема всех реакций $\mathbf{R}=\{\delta\}$, где $\delta=U$, а соответствующий отказ δ называется *стационарностью* (*quiescence*). Каждый стимул x посылается в реализацию с помощью **Q**-кнопки $\{x\}$, $\mathbf{Q}=\{\{x\}|x\in\mathbf{I}\}$.

Кроме внешних действий реализация может совершать внутренние (ненаблюдаемые) действия, обозначаемые 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. В этой статье мы ограничимся только системами без приоритетов.

Предполагается, что любая конечная последовательность любых действий (как внешних, так и внутренних) совершается за конечное время, а бесконечная – за бесконечное время. Также предполагается, что «передача» тестового воздействия (нажатие кнопки) от машины тестирования в реализацию и наблюдения от реализации на дисплей машины выполняются за конечное время. Это гарантирует наблюдение 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.

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.

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.

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 (например, кнопка немедленного саморазрушения для систем оборонного назначения). Разрушение является одним из способов интерпретации неспецифицированного поведения. Исключение из рассмотрения взаимодействий, разрушающих реализацию, часто мотивируют тем, что реализация должна проверять корректность параметров обращения к ней [17,26]. Если параметры некорректны, реализация либо игнорирует обращение, либо сообщает об ошибке. Такое требование к реализации естественно, если это «система общего пользования»: в ней должна быть предусмотрена «защита от дурака». Однако при взаимодействии между внутренними компонентами или подсистемами, доступ к которым строго ограничен, взаимные проверки корректности обращений излишни. Такое часто встречается при обращении с параметрами сложной структуры и нетривиальными условиями корректности, когда накладные расходы на проверку неоправданно увеличивают трудозатраты на создание системы, её объём и время выполнения. Альтернативой в этом случае является строгая спецификация предусловий вызова операций [18]. Например, вызов операции освобождения участка памяти, который не был ранее получен через операцию запроса памяти, является нарушением предусловия. Проверке подлежит не поведение компонентов в ответ на некорректные обращения от других компонентов, а правильность обращения компонентов друг к другу. Иными словами, поскольку мы хотим убедиться в правильности реализации, а не окружения, нас не интересует поведение реализации при неправильном взаимодействии с ней. Разрушение в спецификации отмечает те ситуации, когда в реализации допускается любое поведение, в том числе и реальное разрушение. Семантика разрушения предполагает, что оно не должно возникать при правильном поведении окружения.

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

2. Модели реализации и спецификации

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 $P \in \mathbf{R}$. 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. Поэтому *трассовая модель* как множество трасс, которые можно наблюдать при работе с системой, является наиболее естественной моделью этой системы.

Если в трассах в качестве отказов могут встречаться любые подмножества алфавита внешних действий **L**, такие трассы и такая трассовая модель мы называем *полными*. Соответствующая $2^{\mathbf{L}}/\varnothing$-семантика хорошо известна в литературе под названием *failure trace semantics*, а полные трассы, но только без дивергенции и разрушения, называются *failure traces*. Трасса, в которой все отказы наблюдаемы для заданной **R**/**Q**-семантики, то есть принадлежат **R**, называется ***R**-трассой*, а подмножество **R**-трасс полной трассовой модели – ***R**-моделью*. При безопасном тестировании в **R**/**Q**-семантике наблюдаться могут только **R**-трассы, не содержащие символов Δ и γ. Для ***ioco***-семантики **R**-трассы (без Δ и γ) называются *suspension traces*. Другие трассы полной трассовой модели нужны для указания на то, какие **R**-трассы являются безопасными (могут наблюдаться при безопасном тестировании), а какие нет. Для этого достаточно учитывать только трассы, в которых отказы принадлежат множеству $\mathbf{R}\cup\mathbf{Q}$.

Семантика взаимодействия запрещает некоторые трассы в трассовой модели. Например, трасса должна быть *согласованной*: после отказа P не может следовать действие $z\in P$, а также символы Δ и γ. Поэтому моделью может считаться не любое множество трасс, а только такое, которое удовлетворяет некоторому необходимому и достаточному набору условий. Формальное определение этих условий и доказательство их взаимной независимости, необходимости и достаточности даны в [3].

Другой, эквивалентной моделью реализации и спецификации является 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 состояния s в состояние s` по символу z обозначается $s—z\rightarrow s`$. 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, разрешаемым текущей нажатой кнопкой P машины тестирования, то есть $z\in P\cup\{\tau,\gamma\}$.

Отказ P в LTS порождается в *стабильном* состоянии, из которого не выходят τ- и γ-переходы, при условии, что из этого состояния не выходят также переходы по действиям $z\in P$. Состояние *дивергентно*, если в нем начинается бесконечный τ-маршрут, то есть маршрут, все переходы которого помечены символом τ.

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. LTS является наиболее наглядной моделью, поскольку для любой LTS множество ее трасс является трассовой моделью.

В то же время 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 состояния s`. Из-за этого трасса в LTS заканчивается, вообще говоря, не в одном состоянии, а во множестве состояний. Авторы статьи предложили еще одну эквивалентную модель, в которой этого неудобства нет. Она называется RTS (Refusal Transition System) и представляет собой детерминированную LTS не в алфавите $\mathbf{L}\cup\{\tau,\gamma\}$, а в алфавите $\mathbf{L}\cup\mathbf{R}\cup\{\tau,\Delta,\gamma\}$, то есть в ней могут быть явные переходы по **R**-отказам, а бесконечная цепочка τ-переходов $s—\tau\rightarrow\ldots$ заменена одним переходом $s—\Delta\rightarrow$. Правда, за это приходится расплачиваться наглядностью: не всякая детерминированная LTS в таком алфавите является моделью, а только такая, которая удовлетворяет специальному набору условий, определяемых семантикой взаимодействия. При преобразовании LTS в RTS состояниями RTS становятся множества состояний LTS в конце трасс, что аналогично известному алгоритму «детерминизации» порождающего автомата (или графа).

Класс моделей в алфавите **L** обозначим ***MODEL***(**L**), понимая под моделью трассовую модель, LTS- или RTS-модель в зависимости от контекста.

3. Гипотеза о безопасности и безопасная конформность

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 нажатия какой бы то ни было кнопки. Выход в том, чтобы ограничиться теми реализациями, которые можно безопасно тестировать для проверки конформности заданной спецификации. Это ограничение класса тестируемых реализаций формулируется как гипотеза о безопасности, и конформность определяется только для тех реализаций, которые этой гипотезе удовлетворяют. В силу эквивалентности трассовой, LTS- и RTS-моделей нам достаточно определить гипотезу о безопасности и безопасную конформность только для трассовых моделей реализации и спецификации. Такие гипотеза о безопасности и конформность можно назвать *трассовыми*, поскольку они зависят только от трасс реализации и спецификации, но не от их состояний и соответствия состояний.

Безопасное тестирование, прежде всего, предполагает формальное определение на уровне модели отношения безопасности «кнопка P безопасна в модели **M** 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): $\forall P \in \mathbf{R} \cup \mathbf{Q}\ \forall \sigma \in \mathbf{I}$

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

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

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: $\forall R \in \mathbf{R}\ \forall z \in \mathbf{L}\ \forall Q \in \mathbf{Q}\ \forall \sigma \in \mathbf{S}$

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

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

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

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

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

$<\gamma>\notin$**I** & $\forall\mu\ \forall u\ (\mu\cdot<u>\le\sigma \Rightarrow u$ ***safe in*** **I** ***after*** $\mu)$,

$<\gamma>\notin$**S** & $\forall\mu\ \forall u\ (\mu\cdot<u>\le\sigma \Rightarrow u$ ***safe by*** **S** ***after*** $\mu)$.

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

Заметим, что пустая **Q**-кнопка опасна как после любой безопасной по отношению ***safe in*** трассы любой реализации, так и после любой безопасной по любому отношению ***safe by*** трассы любой спецификации. Такую кнопку никогда нельзя нажимать при безопасном тестировании. Поэтому в дальнейшем будем считать, что такой **Q**-кнопки в семантике нет: $\varnothing\notin$**Q**. В то же время пустая **R**-кнопка имеет смысл: она безопасна после любой безопасной трассы, не продолжающейся дивергенцией, а наблюдение пустого **R**-отказа означает, что реализация оказалась в стабильном состоянии.

Из определения отношения безопасности ***safe by*** видно, что множество ***SafeBy***(**S**) безопасных трасс спецификации **S** однозначно определяется множеством ее **R**-трасс. Из определения отношения безопасности ***safe in*** видно, что множество ***SafeIn***(**I**) безопасных трасс реализации **I**, кроме множества ее **R**-трасс, дополнительно зависит от продолжения **R**-трасс **Q**-отказами во множестве **I**.

Требование безопасности тестирования выделяет класс *безопасно-тестируемых* реализаций ***SafeImp***(**R**/**Q**,**S**,***safe by***), то есть таких, которые могут быть безопасно протестированы для проверки их конформности или неконформности 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 реализации и спецификации любая кнопка, безопасная в спецификации, безопасна после этой трассы в реализации:

I ***safe for*** **S** $=_{\mathbf{def}}$ $(<\gamma>\notin$**S** $\Rightarrow$ $<\gamma>\notin$**I**) & $\forall\sigma\in$***SafeBy***(**S**)$\cap$**I** $\forall$P$\in$**R**$\cup$**Q** (P ***safe by*** **S** ***after*** σ $\Rightarrow$ 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 [25]. 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 безопасна 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***(σ,P,**I**)$\subseteq$***obs***(σ,P,**S**), где ***obs***(σ,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***(**R**/**Q**,**S**,***safe by***).

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

4. Генерация тестов

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 $u\in P \vee u=P\in$**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 после подтрассы этой истории.

Тест безопасен тогда и только тогда, когда в каждой его истории каждая кнопка безопасна в спецификации после подтрассы непосредственно предшествующего этой кнопки префикса истории. Иными словами, тест безопасен тогда и только тогда, когда подтрассы всех его историй являются тестовыми, где *тестовая трасса* – это пустая безопасная трасса или трасса $\sigma\cdot\langle u\rangle$, где трасса σ безопасна в спецификации, а наблюдение u безопасно в спецификации **S** после σ (но не обязательно продолжает σ в **S**): $\sigma \in$ ***SafeBy***(**S**) & u ***safe by*** **S** ***after*** σ. Очевидно, достаточно ограничиться только такими тестовыми трассами, которые могут встречаться в реализационных моделях. А для этого трасса должна удовлетворять тем условиям, которые налагаются на трассы в трассовой модели. В частности, если безопасная трасса σ заканчивается отказом P, то любое действие $u \in P$ безопасно после σ, но трасса $\sigma\cdot\langle u\rangle$ несогласована и не может встречаться в модели.

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*, если каждая конформная реализация его проходит; *исчерпывающий*, если каждая неконформная реализация его не проходит; *полный*, если он значимый и исчерпывающий. Для определения конформности или неконформности 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 [3]. Примитивный тест строится по одной выделенной немаксимальной (по отношению «≤») 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`. Безопасность трассы гарантирует безопасность кнопки R и для каждого действия z в трассе наличие разрешающей его безопасной кнопки P, а немаксимальность безопасной трассы гарантирует наличие последней кнопки P`. Выбор кнопок P и P` может быть неоднозначным: по одной безопасной трассе спецификации можно сгенерировать, вообще говоря, несколько разных примитивных тестов. Однако множества тестов, сгенерированных по разным трассам, не пересекаются.

Если наблюдение, полученное после нажатия кнопки, продолжает трассу, тест продолжается (немаксимальная в тесте история). Наблюдение, продолжающее трассу, то есть 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.

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. Здесь предполагается, что недетерминизм поведения реализации – это явление того уровня абстракции, которое определяется нашими тестовыми

возможностями по наблюдению и управлению, то есть семантикой взаимодействия. Иными словами, поведение реализации недетерминировано, поскольку оно зависит от неких не учитываемых нами факторов – «погодных условий», которые определяют выбор выполняемого действия детерминировано.

Для того чтобы тестирование могло быть полным, мы должны предположить, что любые погодные условия могут быть воспроизведены в тестовом эксперименте, причём для каждого теста. Если такая возможность есть, то есть при бесконечной последовательности прогона теста будут воспроизведены все возможные погодные условия, тестирование называется *глобальным* [21]. Мы абстрагируемся от количества вариантов погодных условий. Здесь нам важна только потенциальная возможность проверить поведение системы при любых погодных условиях и любом поведении оператора. Без этого мы не можем быть уверены, что провели тестовые испытания каждого теста для всех возможных погодных условий, то есть при любом возможном недетерминированном поведении реализации.

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. 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.

Можно отметить, что последовательность прогонов тестов, чередующихся с рестартом системы, можно рассматривать как один тест, в котором, кроме тестовых воздействий, может встречаться рестарт системы. Вместо конечности каждого теста в наборе тестов мы должны потребовать конечность каждого отрезка единого теста между рестартами, то есть отсутствие в нем бесконечного постфикса без рестарта. 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. Эквивалентность трасс, спецификаций, отношений безопасности и семантик взаимодействия

Гипотеза о безопасности и конформность задаются тройкой: семантика взаимодействия **R**/**Q**, спецификационная модель **S**, отношение безопасности ***safe by***. Две такие тройки (**R**$_1$/**Q**$_1$,**S**$_1$,***safe by***$_1$) и (**R**$_2$/**Q**$_2$,**S**$_2$,***safe by***$_2$) в одном алфавите **L**=∪**R**$_1$∪∪**Q**$_1$=∪**R**$_2$∪∪**Q**$_2$ можно считать *эквивалентными*, если они определяют одинаковые классы безопасно-тестируемых реализаций ***SafeImp***(**R**$_1$/**Q**$_1$,**S**$_1$,***safe by***$_1$)=***SafeImp***(**R**$_2$/**Q**$_2$,**S**$_2$,***safe by***$_2$) и одинаковые классы конформных реализаций ***ConfImp***(**R**$_1$/**Q**$_1$,**S**$_1$,***safe by***$_1$)=***ConfImp***(**R**$_2$/**Q**$_2$,**S**$_2$,***safe by***$_2$). Соответственно, назовем *эквивалентным преобразованием* такое преобразование семантики, спецификационной модели и/или отношения безопасности, которое дает тройку, эквивалентную исходной.

Требования к отношению ***safe by*** однозначно определяют безопасность **R**-кнопок, но оставляют достаточно много свободы в объявлении безопасных и опасных **Q**-кнопок. При фиксированной семантике и спецификационной модели можно рассматривать эквивалентные отношения безопасности ***safe by*** как отношения, определяющие одни и те же классы безопасно-тестируемых и конформных реализаций. Предметом отдельных исследований может служить вопрос о том, когда меняется, а когда сохраняется класс

конформных реализаций при преобразовании отношения ***safe by***, сохраняющем только класс безопасно-тестируемых реализаций.

В некоторых случаях можно говорить о «несогласованности» отношения ***safe by*** в следующем смысле: хотя некоторая **Q**-кнопка объявлена безопасной после одной трассы спецификации и опасной – после другой трассы, в любой реализации после этих двух трасс эта кнопка одинаково безопасна по отношению ***safe in***. Это происходит тогда, когда в любой LTS-реализации эти две трассы заканчиваются в одном и том же множестве состояний. А это, в свою очередь, происходит тогда и только тогда, когда трассы *эквивалентны* в следующем смысле: в этих трассах на соответствующих местах стоят одинаковые внешние действия или последовательности отказов с одинаковым множеством отвергаемых внешних действий. Этим последовательностям отказов соответствуют стабильные состояния, в которых нет переходов по всем действиям, принадлежащим каким-нибудь отказам в последовательности. Это дает возможность говорить о *нормальном* отношении ***safe by***, которое определяет одинаковые безопасные кнопки после эквивалентных трасс. Любое отношение ***safe by*** можно *нормализовать*, если после каждой заданной трассы объявить безопасными те и только те кнопки, которые исходным отношением объявлены безопасными после *какой-нибудь* трассы, эквивалентной заданной трассе. Нормализация является эквивалентным преобразованием.

Другим полезным ограничением на отношение ***safe by*** является требование совпадения множеств безопасных кнопок после трасс, имеющих одинаковые трассовые продолжения. Если в LTS-модели трассы заканчиваются в одном множестве состояний, то они имеют одинаковые продолжения, но обратное, вообще говоря, не верно. Тем не менее, для LTS-спецификации можно ограничиться отношением ***safe by***, при котором требуется совпадения множеств безопасных кнопок после трасс, заканчивающихся в одном множестве состояний (для RTS-спецификации – в одном состоянии). Такое отношение ***safe by*** назовем *2-нормальным*. В общем случае для фиксированных семантики и спецификационной модели может не существовать 2-нормального отношения ***safe by***, эквивалентного исходному. Однако это не уменьшает спецификационную мощность: если фиксировать только семантику, то любую спецификационную модель с любым отношением ***safe by*** можно эквивалентно преобразовать в другую модель с другим, уже 2-нормальным отношением ***safe by***. Правда, остается неисследованным вопрос о том, когда такое преобразование сохраняет конечность спецификационной модели, что важно для практического применения. 2-нормальное отношение ***safe by*** удобно для практического применения тем, что для спецификации с конечным числом состояний конечно и семейство множеств состояний после трасс, а 2-нормальное отношение ***safe by*** определяется как раз для таких множеств.

Тестирование существенно зависит от имеющегося набора тестовых возможностей по управлению и наблюдению за поведением тестируемой системы, формализуемых в семантике взаимодействия. Поэтому представляет интерес сравнение различных семантик по мощности тестирования.

Будем говорить, что $\mathbf{R_1/Q_1}$-семантика *не сильнее* $\mathbf{R_2/Q_2}$-семантики в том же алфавите **L** и обозначать $\mathbf{R_1/Q_1 \le R_2/Q_2}$, если для каждой спецификационной модели **S** с отношением $\boldsymbol{safe\ by_1}$ в $\mathbf{R_1/Q_1}$-семантике существует отношение $\boldsymbol{safe\ by_2}$ в $\mathbf{R_2/Q_2}$-семантике с теми же классами безопасных и конформных реализаций, то есть тройки ($\mathbf{R_1/Q_1}$,**S**,$\boldsymbol{safe\ by_1}$) и ($\mathbf{R_2/Q_2}$,**S**,$\boldsymbol{safe\ by_2}$) эквивалентны. Соответственно семантики эквивалентны, если каждая из них не сильнее другой. В [4] показано, что отношение «не сильнее» является предпорядком (рефлексивно и транзитивно). Соответственно, их эквивалентность рефлексивна, транзитивна и симметрична. Найдено необходимое и достаточное условие отношения $\mathbf{R_1/Q_1 \le R_2/Q_2}$: 1) каждая $\mathbf{Q_1}$-кнопка является также $\mathbf{Q_2}$-кнопкой, 2) каждая $\mathbf{Q_2}$-кнопка представима в виде объединения $\mathbf{R_1}$- и $\mathbf{Q_1}$-кнопок, 3) каждая $\mathbf{R_1}$-кнопка представима в виде объединения конечного числа $\mathbf{R_2}$-кнопок, и, наоборот, каждая $\mathbf{R_2}$-кнопка представима в виде объединения конечного числа $\mathbf{R_1}$-кнопок. Соответственно, две семантики эквивалентны,

если совпадают семейства их **Q**-кнопок, а каждая **R**-кнопка одной семантики представима в виде объединения конечного числа **R**-кнопок другой семантики.

В связи с этим представляют интерес эквивалентные семантики минимальные и наименьшие по вложенности семейств кнопок. Кнопку из **R** будем называть *конечно-разложимой*, если ее можно представить в виде объединения конечного числа кнопок из **R**, отличных от них самих (в противном случае кнопка *конечно-неразложима*). Доказано, что если для исходной **R**/**Q**-семантики существует эквивалентная ей минимальная $\mathbf{R_0}$/**Q**-семантика, то семейство $\mathbf{R_0}$ совпадает с множеством конечно-неразложимых кнопок из **R**. Для существования такой эквивалентной минимальной $\mathbf{R_0}$/**Q**-семантики необходимо и достаточно, чтобы любая конечно-разложимая **R**-кнопка разлагалась в объединение конечного числа конечно-неразложимых **R**-кнопок. Если конечно число бесконечных **R**-кнопок (в частности, конечно семейство **R**), то минимальная эквивалентная семантика существует. Если минимальная эквивалентная семантика существует, то она же является наименьшей.

Отношение «не сильнее» для семантик можно обобщить, если не фиксировать спецификационную модель, а только потребовать сохранения как мощности тестирования, так и мощности специфицирования: $\mathbf{R_1}$/$\mathbf{Q_1}$-семантика *не сильнее* $\mathbf{R_2}$/$\mathbf{Q_2}$-семантики в том же алфавите **L**, если для каждой спецификационной модели $\mathbf{S_1}$ с отношением ***safe by***$_1$ в $\mathbf{R_1}$/$\mathbf{Q_1}$-семантике существует спецификационная модель $\mathbf{S_2}$ с отношением ***safe by***$_2$ в $\mathbf{R_2}$/$\mathbf{Q_2}$-семантике с теми же классами безопасных и конформных реализаций. Соответствующим образом обобщается и эквивалентность семантик.

Дальнейшее обобщение позволяет связывать отношением «не сильнее» и эквивалентностью семантики в разных алфавитах $\mathbf{L_1}$ и $\mathbf{L_2}$. Естественно в качестве реализаций рассматриваются только те, которые определены в пересечении алфавитов $\mathbf{L}=\mathbf{L_1}\cap\mathbf{L_2}$. Здесь используется то обстоятельство, что любую LTS-реализацию в алфавите **L** можно рассматривать как LTS-реализацию в большем алфавите $\mathbf{L_i}$ (i=1,2): в ней просто не будет переходов по действиям из $\mathbf{L_i}\backslash\mathbf{L}$. Для $\mathbf{R_i}$/$\mathbf{Q_i}$-семантики реализация рассматривается в алфавите $\mathbf{L_i}$, и используются ее $\mathbf{R_i}$-трассы.

Такие обобщенные отношения семантик составляют предмет дальнейших исследований, в частности, для тех случаев, когда преобразование спецификационной модели сохраняет ее конечность. Один частный случай эквивалентности семантик рассматривается ниже при пополнении спецификации (раздел 7).

6. Неактуальные и неконформные трассы спецификации

Трассу спецификации назовем актуальной, если она встречается в безопасно-тестируемых реализациях. В [3] показано, что, если в семантике нет **Q**-кнопок, то спецификация удовлетворяет собственной гипотезе о безопасности. Поэтому для таких семантик все безопасные трассы спецификации актуальны. При наличии **Q**-кнопок, вообще говоря, не всякая безопасная трасса спецификации актуальна.

Пример приведен на Рис.2. Здесь в LTS-спецификации $\mathbf{S_1}$, изображенной слева, кнопка {a} безопасна после пустой трассы, и трасса <{a,b}> безопасна. По гипотезе о безопасности в безопасно-тестируемой реализации после пустой трассы не должно быть **Q**-отказа {a}. Однако, если в реализации есть трасса <{a,b}>, то хотя бы в одном состоянии после пустой трассы имеется **R**-отказ {a,b}, а тогда в этом состоянии имеется **Q**-отказ {a}. Поэтому в безопасно-тестируемых реализациях не может быть трассы <{a,b}>, хотя она является безопасной трассой спецификации.

Понятно, что достаточно генерировать тесты только по актуальным безопасным трассам спецификации, что дает возможность оптимизации генерации тестов.

Точно также тестовая, но отсутствующая в спецификации, трасса может оказаться неактуальной. Прежде всего, все несогласованные тестовые трассы неактуальны. Но могут быть и согласованные неактуальные тестовые трассы. На Рис.2 в LTS-спецификации $\mathbf{S_2}$, изображенной справа, тестовая трасса <{a,b}> отсутствует в спецификации, согласована, но

неактуальна. Тестовые истории с такими неактуальными тестовыми трассами удаляются из каждого теста, сгенерированного по актуальной безопасной трассе.

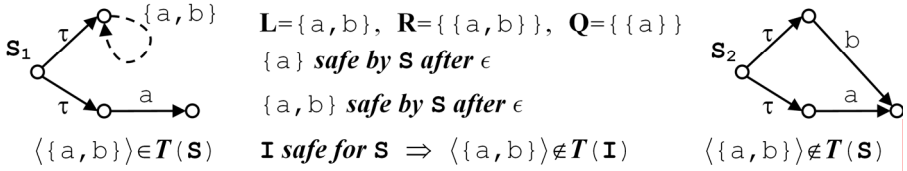


Рис.2. Неактуальные безопасная и тестируемая трассы спецификации

Трассу, которая встречается в конформных реализациях, будем называть *конформной*. В [3] показано, что, если в семантике нет **Q**-кнопок, то спецификация безопасно-тестируема и конформна сама себе. Поэтому для таких семантик все безопасные трассы спецификации конформны. При наличии **Q**-кнопок, вообще говоря, не всякая безопасная трасса спецификации конформна.

На Рис.3 приведен пример, взятый из [1]. Здесь используется семантика отношения ***ioco***. Это отношение предполагает всюду-определенность реализации по стимулам, то есть отсутствие в ней **Q**-отказов. Отношение ***saco*** более либерально, но даже для него в LTS-спецификации **S** кнопка {x} по 2-ому правилу отношения ***safe by*** должна быть безопасной после трассы <x>. Если в реализации есть трасса <δ,x,δ>, то в ней есть и трасса <x>, после которой кнопка {x} по гипотезе о безопасности должна быть безопасной по ***safe in***. А тогда кнопка {x} безопасна по ***safe in*** после трассы <δ,x,δ>. Поскольку это **Q**-кнопка, ее безопасность после трассы <δ,x,δ> означает наличие в реализации трассы <δ,x,δ,x>. Поскольку δ **R**-кнопка и в спецификации нет разрушения, эта кнопка безопасна после любой безопасной трассы спецификации, в частности после трассы <x,x>, а тогда в реализации она безопасна по ***safe in*** после трассы <x,x> и, следовательно, безопасна по ***safe in*** после трассы <δ,x,δ,x>. Поэтому в реализации должна быть хотя бы одна из трасс <δ,x,δ,x,a>, <δ,x,δ,x,b> или <δ,x,δ,x,δ>. Но тогда в реализации есть хотя бы одна из трасс <x,δ,x,a>, <δ,x,x,b> или <x,x,δ>. Каждая из этих трасс является продолжением безопасной трассы спецификации наблюдением, разрешаемым безопасной кнопкой δ, но отсутствующим в спецификации, что противоречит конформности. Следовательно, трасса <δ,x,δ>, хотя и безопасна в спецификации, но неконформна.

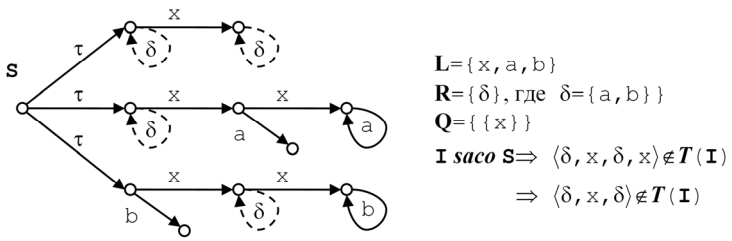


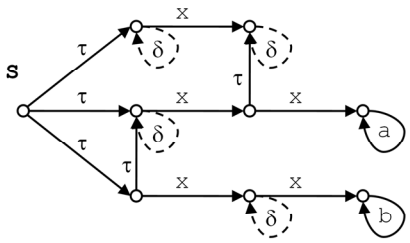
Рис.3. Неконформная безопасная трасса спецификации

Наличие в спецификации неконформных безопасных трасс само по себе не создает проблем, но дает возможность оптимизации тестирования. Обычный ***ioco***-тест после трассы <δ,x,δ> закончит свою работу (повторно нажимать кнопку δ заведомо излишне) с вердиктом ***pass***. Модифицированный тест, «распознающий» неконформные трассы, вынесет в этом случае вердикт ***fail***, что позволяет быстрее обнаруживать неконформность. Такое распознавание неконформных трасс может существенно уменьшить объем генерируемых тестов.

Рис.4 является модификацией Рис.3: для того, чтобы в нужных состояниях не было отказа δ, вместо первых переходов по действиям a и b используются τ-переходы. По-прежнему в конформной реализации не может быть трассы <δ,x,δ>. Но после трассы <δ,x> кнопка δ

безопасна, но не может быть действий a или b. Следовательно, в конформной реализации не может быть трассы <δ,x>. Если бы в конформной реализации была трасса <δ>, то, поскольку кнопка {x} – это **Q**-кнопка, безопасная после трассы <δ>, в реализации должна была бы быть и неконформная трасса <δ,x>. Следовательно, в конформной реализации нет трассы <δ>. В то же время в ней после пустой трассы не должно быть действий a или b. Следовательно, в конформной реализации не может быть пустой трассы. Следовательно, пустая трасса также неконформна, а тогда все трассы спецификации неконформны.

Этот несколько курьезный пример показывает, насколько полезным может оказаться анализ неконформных трасс. Для этого примера имеется бесконечное множество безопасно-тестируемых реализаций, но нет ни одной конформной реализации. Полное тестирование без дополнительных предположений или тестовых возможностей будет бесконечным, поскольку имеется цикл по действию a после безопасной трассы <δ,x,x> и, тем самым, имеется бесконечное число безопасных трасс вида <δ,x,x,a,…,a> и актуальных тестовых, но ошибочных трасс вида <δ,x,x,a,…,a,b> или <δ,x,x,a,…,a,δ>. В то же время тестирование, «распознающее» неконформные трассы, вообще не будет проводиться, так как будет обнаружено, что все трассы спецификации неконформны.



$\mathbf{L}=\{x,a,b\}$
$\mathbf{R}=\{\delta\}$, где $\delta=\{a,b\}\}$
$\mathbf{Q}=\{\{x\}\}$
$\mathbf{I}$ *saco* $\mathbf{S} \Rightarrow \langle\delta,x,\delta,x\rangle \notin \boldsymbol{T}(\mathbf{I})$
$\Rightarrow \langle\delta,x,\delta\rangle \notin \boldsymbol{T}(\mathbf{I})$
$\Rightarrow \langle\delta,x\rangle \notin \boldsymbol{T}(\mathbf{I})$
$\Rightarrow \langle\delta\rangle \notin \boldsymbol{T}(\mathbf{I})$
$\Rightarrow \epsilon \notin \boldsymbol{T}(\mathbf{I})$

Рис.4. Спецификация, в которой нет конформных трасс

7. Пополнение спецификации

Причиной, по которой в спецификации могут быть безопасные неактуальные или неконформные трассы, является различие в определении безопасности **Q**-кнопок в отношениях ***safe in*** и ***safe by***. Кроме того, из-за этого различия спецификация, в которой после безопасных трасс встречаются **Q**-отказы. Например, в ***ioco***-семантике любая спецификация, в которой некоторые трассы продолжаются как стимулом x, так и блокировкой стимула {x}, не удовлетворяет своей гипотезе о безопасности ***safe for*** (тем более, требованию всюду определенности по стимулам) и, следовательно, неконформна. Это означает, что отношение ***saco*** (в частности, отношение ***ioco***), вообще говоря, нерефлексивно. В общем случае это отношение также нетранзитивно [3]. Нерефлексивность спецификации и наличие в ней неактуальных или неконформных безопасных трасс прямо противоречит интуиции: если реализацию «списать» со спецификации, то мы получим заведомо ошибочную реализацию, которую к тому же иногда нельзя безопасно тестировать.

В то же время, если все отказы наблюдаемы (нет **Q**-кнопок), отношение ***saco*** рефлексивно и транзитивно, то есть является предпорядком [3]. Поскольку в этом случае спецификация конформна сама себе, в ней не может быть неактуальных или неконформных безопасных трасс. Это наводит на мысль перейти от **R**/**Q**-семантики к **R**∪**Q**/∅-семантике, когда все отказы наблюдаемы. Переход делается с помощью преобразования спецификации, которое называется *пополнением*, и на пополненной спецификации определяется отношение безопасности ***safe by*** = ***safe in***. В пополненной спецификации нет **Q**-отказов.

Пополненная спецификация эквивалентна исходной спецификации в **R**/**Q**-семантике: сохраняются классы безопасно-тестируемых и класса конформных реализаций. В то же время, если пополненную спецификацию рассматривать в **R**∪**Q**/∅-семантике, то тоже сохраняется класс конформных реализаций, а класс безопасно-тестируемых реализаций не сужается, и тем самым не теряется возможность тестировать те реализации, которые мы

могли тестировать по исходной спецификации, но может расширяться, что позволяет тестировать дополнительные реализации.

Конечно, для того, чтобы тестировать реализацию по пополненной спецификации не в исходной **R**/**Q**-семантике, а в **R**∪**Q**/∅-семантике, требуется наличие соответствующих тестовых возможностей: все отказы должны быть наблюдаемы. Если такой возможности нет, при тестировании по-прежнему используется **R**/**Q**-семантика, а смысл пополнения – в том, чтобы удалить из спецификации (или, по крайней мере, как-то отметить) неактуальные и неконформные трассы.

Пополнение спецификации является также первым шагом к решению проблемы монотонности конформности, которой посвящен раздел 9.

Более строго, пополненную спецификацию в общем случае не удается построить в той же **R**/**Q**-семантике, она строится в эквивалентной $\mathbf{R}^{\#}/\mathbf{Q}^{\#}$-семантике в расширенном алфавите $\mathbf{L}^{\#}$. Идея в том, чтобы для каждого отказа P∈**R**∪**Q** добавить в алфавит внешних действий и в сам отказ P специальное фиктивное действие *не-отказ* P, обозначаемое ₽. Новый отказ обозначается $P^{\#}$=P∪{₽}. Предполагается, что $P \neq P` \Rightarrow ₽ \neq ₽`$. Спецификационная модель **S** преобразуется в спецификационную модель $\mathbf{S}^{\#}$, в которой определяется новое отношение безопасности ***safe by*** = ***safe in***. В результате тройки (**R**/**Q**,**S**,***safe by***) и ($\mathbf{R}^{\#}/\mathbf{Q}^{\#}$,$\mathbf{S}^{\#}$,***safe in***) эквивалентны на классе реализаций в исходном алфавите $\mathbf{L}=\mathbf{L}\cap\mathbf{L}^{\#}$: они определяют одинаковые классы безопасно-тестируемых реализаций ***SafeImp***(**R**/**Q**,**S**,***safe by***)=***SafeImp***($\mathbf{R}^{\#}/\mathbf{Q}^{\#}$,$\mathbf{S}^{\#}$,***safe in***)∩***MODEL***(**L**) и одинаковые классы конформных реализаций ***ConfImp***(**R**/**Q**,**S**,***safe by***)=***ConfImp***($\mathbf{R}^{\#}/\mathbf{Q}^{\#}$,$\mathbf{S}^{\#}$,***safe in***)∩***MODEL***(**L**). Таким образом, **R**/**Q**- и $\mathbf{R}^{\#}/\mathbf{Q}^{\#}$-семантики эквивалентны (на пересечении алфавитов **L**).

Общий алгоритм пополнения спецификации описан в [3]. Его идея восходит к способу «детерминизации» порождающего автомата (или графа), однако имеет существенные отличия. В трассовой модели $\mathbf{S}^{\#}$ определяется как замыкание $\mathbf{S}_{\mathbf{final}}$ множества *финальных* трасс по операции ***d*** удаления того или иного отказа из трассы. Трасса σ финальна, если с помощью операций удаления ***d***, перестановки ***t*** соседних и повторения ***r*** отказов из нее можно получить трассу σ`, безопасную в исходной спецификации **S**. Такая трасса σ` называется ***drt***-подтрассой трассы σ. Кроме того, каждая такая финальная трасса σ продолжаются не-отказом ₽, если отказ P не входит в постфикс отказов трассы σ, и далее дивергенцией Δ, если кнопка P безопасна после некоторой ***drt***-подтрассы трассы σ, или разрушением γ в противном случае.

Для конечного алфавита преобразование пополнения алгоритмизуемо. Можно построить LTS-пополнение $\mathbf{S}_{\mathbf{final}}$, состояниями которого будут финальные трассы без повторных отказов (в любой подпоследовательности отказов). Однако такое LTS-пополнение и соответствующее ему RTS-пополнение в общем случае бесконечны, даже если исходная спецификационная LTS-модель **S** конечна, что неприемлемо на практике. Выход – в использовании 2-нормального отношения ***safe by***. В этом случае можно построить RTS-пополнение $\mathbf{S}_{\mathbf{final}}$, беря в качестве состояния вместо финальной трассы σ, не содержащей не-отказов, семейство множеств состояний, в которых заканчиваются безопасные в **S** ***drt***-подтрассы трассы σ, плюс множество действий, запрещаемых постфиксом отказов трассы σ (объединение отказов из этого постфикса). Для LTS **S** с конечным числом состояний число таких состояний RTS $\mathbf{S}_{\mathbf{final}}$ также конечно.

В полученном таким образом RTS-пополнении $\mathbf{S}_{\mathbf{final}}$ уже можно распознать все неактуальные и неконформные трассы.

Состояние s RTS $\mathbf{S}_{\mathbf{final}}$ *неактуально*, если для некоторой **Q**-кнопки Q каждое действие z∈Q принадлежит некоторому **R**-отказу z∈R, по которым в s определен переход-петля s—R→s, а переход по не-отказу ₽, если он есть, продолжается дивергенцией s—₽→—Δ→. Это означает, что кнопка $Q^{\#}$ безопасна после некоторой трассы σ, заканчивающейся в состоянии s и имеющей в постфиксе отказов все такие **R**-отказы R. Однако единственно возможное наблюдение – это не-отказ ₽, которого не бывает в реализациях в алфавите **L**.

Следовательно, если в такой реализации есть трасса σ, то после нее при нажатии кнопки Q возникнет ненаблюдаемый отказ, то есть такая реализация не является безопасно-тестируемой.

Удаляя из RTS $\mathbf{S_{final}}$ все неактуальные состояния и ведущие в них переходы, мы получаем эквивалентную спецификацию $\mathbf{S_{act}}$, все трассы которой актуальны.

Например, для спецификации $\mathbf{S_1}$ на Рис.2 после пополнения исчезает неактуальная безопасная трасса <{a,b}>, оставаясь неактуальной тестовой трассой. По этой трассе не будут генерироваться тесты, а в любом тесте, начинающемся с нажатия кнопки {a,b}, наблюдение {a,b} удаляется из возможных наблюдений: остаются только конформное наблюдение a (вердикт ***pass*** или продолжение тестирования) и неконформное наблюдение b (вердикт ***fail***).

Состояние s *неконформно*, если для некоторой **R**- или **Q**-кнопки P единственно возможным наблюдением является неразрушающий не-отказ $\not\Vdash$, то есть $s\text{—}\not\Vdash\!\!\rightarrow\text{—}\Delta\!\rightarrow$, нет переходов $s\text{—}z\!\rightarrow$ для $z\in P$ и нет перехода $s\text{—}P\!\rightarrow$ для $P\in\mathbf{R}$. Трасса, заканчивающаяся в неконформном состоянии, неконформна, так как после нее при нажатии безопасной кнопки P не может быть ни одного конформного наблюдения в **R**/**Q**-семантике.

После удаления из $\mathbf{S_{act}}$ всех неконформных состояний и ведущих в них переходов, мы можем получить новые неконформные состояния. И так далее. Если RTS $\mathbf{S_{act}}$ конечна, то, очевидно, этот процесс закончится через конечное число шагов, и мы получим эквивалентную спецификацию $\mathbf{S_{conf}}$, все трассы которой конформны.

Например, для спецификации на Рис.3 после пополнения исчезает безопасная неконформная трасса <δ,x,δ>, оставаясь тестовой трассой, после получения которой выносится вердикт ***fail***, а не ***pass***, как по исходной спецификации.

Заметим, что в полученной спецификации могут остаться трассы, которых нет в исходной спецификации. Например, если в спецификации на Рис.3 заменить переходы по действию b на переходы по действию a, то трасса <δ,x,δ> становится конформной, но при пополнении будут построены также новые трассы <δ,x,δ,x> и <δ,x,δ,x,a>. После этих новых трасс возможны и ошибки: <δ,x,δ,x,b>, <δ,x,δ,x,δ>, <δ,x,δ,x,a,b> и <δ,x,δ,x,a,δ>. Однако наличие или отсутствие тестов, сгенерированных по этим трассам и обнаруживающих эти ошибки, не влияет на полноту тестирования. Поэтому все эти вновь построенные трассы можно пометить для того, чтобы не генерировать по ним тесты. В RTS $\mathbf{S_{conf}}$ эту оптимизацию можно выполнить, помечая некоторые переходы (в примере на Рис.3 помечается переход по x после трассы <δ,x,δ>).

8. Полное и тотальное тестирование. Зависимости между ошибками

Целью полного тестирования является только проверка конформности или неконформности реализации. Если реализация неконформна, то достаточно найти хотя бы одну ошибку. В то же время тестирование является лишь этапом в жизненном цикле разработки целевой системы [10]. 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.

Возможность наличия в спецификации неконформных безопасных трасс позволяет разделить возможные ошибки на три рода. *Ошибкой 1-го рода* будем называть продолжение безопасной и конформной трассы спецификации безопасным наблюдением, отсутствующим в спецификации. *Ошибка 2-го рода* – это безопасная, но неконформная трасса спецификации.

Ошибка 3-го рода – продолжение безопасной, но неконформной трассы спецификации безопасным наблюдением, отсутствующим в спецификации.

Следует заметить, что род ошибки не является инвариантом при эквивалентных преобразованиях спецификации. Более того, при эквивалентных преобразованиях может не сохраняться множество актуальных тестовых трасс, тем самым некоторые актуальные ошибки, определяемые одной спецификацией, могут не быть тестовыми трассами (и, следовательно, ошибками) для другой эквивалентной спецификации.

Например, для примера на Рис.3 трасса <δ,x,δ> в исходной спецификации **S** формально является ошибкой 2-го рода, но, поскольку в исходной спецификации неконформные трассы никак не отмечены, тесты эту ошибку не обнаруживают; в спецификации $\mathbf{S_{final}}$ или $\mathbf{S_{act}}$ с уже обнаруженными и отмеченными неконформными трассами, это ошибка 2-го рода, а в спецификации $\mathbf{S_{conf}}$ – ошибка 1-го рода. Трасса <δ,x,δ,x,a> в исходной спецификации **S** и в спецификации $\mathbf{S_{conf}}$ не является ошибкой, поскольку в них нет ее префикса <δ,x,δ,x>, а в спецификации $\mathbf{S_{final}}$ или $\mathbf{S_{act}}$ с уже обнаруженными и отмеченными неконформными трассами, это ошибка 3-го рода.

Поэтому, говоря о том, что тотальное тестирование должно обнаруживать все ошибки, мы должны под ошибками понимать трассы, которые являются ошибками для какой-нибудь спецификации из класса эквивалентных спецификаций. Лучший способ определить все такие ошибки – это построить «самую большую» эквивалентную спецификацию, которая определяла бы все эти ошибки, то есть эквивалентную спецификацию с наибольшим множеством актуальных тестовых трасс. Для этого, в отличие от полного тестирования, мы должны не «уменьшать» пополнение $\mathbf{S_{act}}$ до $\mathbf{S_{conf}}$, а, наоборот, «увеличивать», добавляя недостающие актуальные тестовые трассы, но оставляя «разметку» неконформных трасс. Получится спецификация $\mathbf{S_{total}}$, которая определяет все ошибки.

Для оптимизации набора тестов при полном или тотальном тестировании мы должны учитывать возможные зависимости между ошибками (ошибочными тестовыми трассами). Будем говорить, что из ошибка μ_1 *следует* ошибка μ_2, если в любой безопасно-тестируемой реализации, в которой есть трасса μ_1, есть и трасса μ_2. Отношение следования ошибок, очевидно, является предпорядком (рефлексивно и транзитивно). Поскольку отношение следования, вообще говоря, не антисимметрично, оно определяет нетривиальную эквивалентность (рефлексивное, транзитивное и симметричное отношение) ошибок. Эта эквивалентность определяет фактор-отношение следования для ошибок, которое уже будет частичным порядком (рефлексивно, транзитивно и антисимметрично).

Ясно, что для полноты тестирования достаточно обнаруживать только такие ошибки, которые минимальны по предпорядку следования с точностью до их эквивалентности, то есть хотя бы одну ошибку из каждого *минимального* класса эквивалентности. Для тотальности тестирования достаточно обнаруживать ошибки с точностью до их эквивалентности, то есть хотя бы одну ошибку из *каждого* класса эквивалентности.

Заметим, что если из класса M_1 следует класс $M_2 \neq M_1$, то в тотальном наборе тестов должны быть как тесты, обнаруживающие ошибки из M_1, так и тесты, обнаруживающие ошибки из M_2. Действительно, если в реализации есть ошибки из M_1, но нет ошибок из M_2, то для обнаружения ошибок из M_1 нужны тесты для M_1 (недостаточно тестов для M_2). Если же в реализации нет ошибок из M_1, но есть ошибки из M_2, то для обнаружения ошибок из M_2 нужны тесты для M_2 (недостаточно тестов для M_1). Другое дело, что во время реального прогона тестов, если обнаруживается ошибка из M_1, то после этого можно не прогонять тесты для M_2, поскольку такие ошибки заведомо есть в реализации. И, наоборот, если мы убедились, что в реализации нет ошибок из M_2, то после этого можно не прогонять тесты для M_1, поскольку таких ошибок заведомо нет в реализации.

9. Проблема монотонности конформности и монотонное преобразование спецификации

Проблема монотонности конформности возникает в связи с композицией системы. Композиционная система – это составная система, собранная из компонентов с помощью применения определенных правил композиции. В данной работе компоненты моделируются LTS, а правила композиции – оператором ↑↓ параллельной композиции LTS в духе CCS (Calculus of Communicating Systems [22], также [19]). Будем считать, что на универсуме внешних действий задана инволюция (биекция, обратная сама себе) «подчёркивание», которая каждому внешнему действию z ставит в соответствие *противоположное* действие $\underline{z}$ так, что $\underline{\underline{z}} = z$. Для двух LTS-операндов в алфавитах **A** и **B** композиция – это LTS в алфавите $\mathbf{C}=(\mathbf{A}\setminus\underline{\mathbf{B}})\cup(\mathbf{B}\setminus\underline{\mathbf{A}})$. Состояниями композиции являются пары состояний LTS-операндов, начальное состояние – пара начальных состояний. Переходы композиции определяются как наименьшее множество, порождаемое следующими правилами вывода: для любых состояний a,a` первого LTS-операнда и любых состояний b,b` второго LTS-операнда:

(1) $z\in(\mathbf{A}\cup\{\gamma\tau\})\setminus\underline{\mathbf{B}}$ & a—z→a` ⊢ ab—z→a`b,

(2) $z\in(\mathbf{B}\cup\{\gamma\tau\})\setminus\underline{\mathbf{A}}$ & b—z→b` ⊢ ab—z→ab`,

(3) $z\in\mathbf{A}\cap\underline{\mathbf{B}}$ & a—z→a` & b—$\underline{z}$→b` ⊢ ab—τ→a`b`,

Основная проблема композиционных систем звучит так: если компоненты работают правильно, то почему система в целом работает неправильно? В теории конформности правильность реализации отдельного компонента определяется как конформность реализации компонента спецификации этого компонента, а правильность реализации системы – как конформность реализации системы спецификации системы.

Правильность реализации компонента проверяется тестированием этого компонента, когда тест непосредственно взаимодействует с компонентом, подменяя собой его окружение. Такое тестирование называют *автономным* (тестируется отдельный компонент) или *синхронным*. Очевидно, одной из причин неправильной работы системы может оказаться неправильная работа её компонентов, которые при автономном тестировании не были полностью проверены. С учетом бесконечности полного набора тестов и недетерминизма реализации это вполне возможно и действительно часто встречается на практике.

Однако проблема композиционных систем этим не исчерпывается. Реализации компонентов могут быть конформны своим спецификациям, а собранная из этих компонентов система неконформна спецификации системы в целом. В чём причина и что делать в такой ситуации?

Проблема здесь в том, что само соотношение спецификаций компонентов и спецификации системы неправильно. Это уже ошибка декомпозиции спецификации системы на спецификации её компонентов [2,3,11,13,14]. Такие ошибки значительно хуже ошибок в отдельных компонентах, поскольку их труднее обнаруживать, и они имеют более печальные последствия. Поэтому системное или комплексное тестирование не просто продолжает незаконченное тестирование компонентов, но предназначено также для обнаружения ошибок архитектурного уровня, которые не обнаруживаются автономными тестами. Такие ошибки могут привести к достаточно радикальным изменениям в спецификациях, что потребует модификации или даже повторной реализации всех или части компонентов.

Правильное соотношение спецификаций компонентов и спецификации системы должно удовлетворять *условию монотонности*: композиция компонентов, конформных своим спецификациям, конформна спецификации системы. Спецификация системы *корректна*, если она удовлетворяет условию монотонности при заданных спецификациях компонентов. Самая сильная (то есть, предъявляющая максимальные требования) корректная спецификация системы определяется самим условием монотонности. Однако это определение неявно, и не ясно, существует ли такая самая сильная корректная спецификация и можно ли её построить алгоритмически.

Проблема в том, что самая сильная корректная спецификация системы, если она существует, оказывается слабее композиции спецификаций, проводимой по тем же правилам, что и композиция реализаций компонентов. Если композицию реализаций $\uparrow\downarrow$ называть *прямой*, то нужна другая – *косая* – композиция спецификаций $\nearrow\!\!\downarrow$, совпадающая с самой сильной корректной спецификацией системы, если она существует. 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-действия).

Особым случаем композиции является асинхронное тестирование или тестирование в контексте [24]. Такое тестирование можно рассматривать как тестирование системы из двух компонентов, один из которых – реализация, а другой – фиксированная среда взаимодействия. При асинхронном тестировании, в отличие от синхронного тестирования, возникают две проблемы [19]: 1) «вседозволенность» (*permissiveness*), когда асинхронные тесты не ловят ошибку, обнаруживаемую синхронными тестами, и 2) «несохранение соответствия» (*non preservation of conformance*), когда асинхронные тесты ловят «ложную» ошибку. С первой проблемой, по-видимому, приходится мириться: асинхронное (и вообще композиционное) тестирование – это более «косвенное» (через контекст) тестирование реализации. Оно может не позволить создать все те режимы взаимодействия и наблюдать всё то поведение реализации, которые возможны в синхронном тестировании, когда тест и реализация взаимодействуют непосредственно друг с другом. Вторая проблема более серьёзная – это частный случай общей проблемы монотонности.

Таким образом, ставится задача построения косой композиции для выбранной конформности. Пополнение спецификаций – это первый шаг к решению этой задачи. У нас появляется возможность решать эту задачу только для **R**/∅-семантики, когда **Q**-кнопок нет. Тогда под спецификацией можно понимать пополненную спецификацию, для которой уже совершён переход от **R**/**Q**-семантики к **R**∪**Q**/∅-семантике. Напомним, что в семантике без **Q**-кнопок отношения ***safe by*** и ***safe in*** совпадают, и кнопка опасна тогда и только тогда, когда она разрешает разрушающее действие.

Прежде всего, отметим, что пополнением проблема монотонности не исчерпывается. На Рис.5 приведён пример несохранения конформности при асинхронном тестировании для семантики со стимулами и реакциями, в которой наблюдаемы все отказы: стационарность (отсутствие всех реакций !a,!b,…) и для каждого стимула ?x его блокировка {?x}. Среда **Q** – это одна ограниченная выходная очередь (на рисунке длины 1) с дополнительной командой «обнуления» (b). Спецификация $\mathbf{S}_0$ описывает следующие требования к системе: сначала стимул блокируется, но можно выдавать цепочку реакций, а потом можно (но необязательно) продолжить: обнулить очередь (b), принять стимул x и закончить в терминальном состоянии. Реализация $\mathbf{S}_1$ конформна: она не обнуляет очередь и, соответственно, не принимает стимул. Когда с очередью компонуется спецификация, первый посылаемый стимул не блокируется. Однако при композиции реализации $\mathbf{S}_1$ такая блокировка появляется, что при асинхронном тестировании будет квалифицировано как ошибка (показано стрелками).

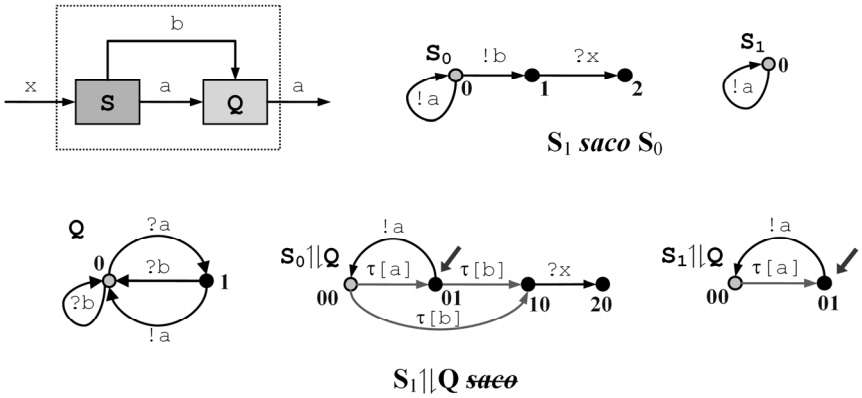


Рис.5. Несохранение конформности при асинхронном тестировании

В [3] даётся формальное определение косой композиции для отношения ***saco*** в семантике **R**/∅. Показано, что косая композиция совпадает с прямой композицией *преобразованных* спецификаций. Такое преобразование называется *монотонным* для этой конформности, а конформность – *монотонной* относительно этого преобразования.

Следует отметить, что монотонное преобразование определяется неоднозначно. В [3] описываются два таких преобразования. Одно основано на объединении конформных реализаций и применимо всегда. Другое применимо лишь при определённых условиях, но зато определяется конструктивно (индуктивно), что даёт возможность его алгоритмизации.

Заметим также, что разные компоненты могут иметь разные алфавиты, но даже в одном алфавите их конформность может быть определена в разных **R**/∅-семантиках, то есть при разных наборах **R**-кнопок машины тестирования. Возникает вопрос: в какой семантике следует рассматривать композицию компонентов? По счастью, этот вопрос снимается тем, что прямая композиция монотонно преобразованных спецификаций оказывается косой композицией *в любой* **R**/∅-семантике (естественно, для одного и того же композиционного алфавита, однозначно определяемого алфавитами операндов и не зависящего от **R**-семантик операндов при тех же алфавитах).

При асинхронном тестировании предполагается, что в среде нет ошибок и она известна. Поэтому монотонное преобразование нужно применять только к спецификации реализации, а среда остаётся неизменной. В этом случае мы говорим о левомонотонном преобразовании, имея в виду сохранение среды, задаваемой как правый операнд композиции (название условно в связи с коммутативностью композиции). При решении проблемы монотонности учитывается и этот случай.

Косая композиция спецификаций компонентов составной системы позволяет решить две задачи: 1) верификация имеющейся спецификации системы (её согласованности со спецификациями компонентов), и 2) при отсутствии спецификации системы её генерация по спецификациям компонентов.

Первая задача иногда называется задачей верификации декомпозиции системных требований, то есть проверкой правильности декомпозиции требований к системе в требования к компонентам системы. Для её решения необходимо построить косую композицию системы и, рассматривая её как реализацию, проверить её конформность имеющейся спецификации системы. Такая проверка может выполняться аналитически, в том числе, моделируя тестирование косой композиции, рассматриваемой как реализация системы, по полному набору тестов, сгенерированному из заданной спецификации системы.

В качестве примера можно рассмотреть композицию (Рис.6) приёмника с двумя входными портами и передатчика с двумя выходными портами. Спецификация составной системы очень проста: нужно принять пару аргументов (x,y), и выдать значение $\boldsymbol{f}(\boldsymbol{g}(x),\boldsymbol{h}(y))$. Интуитивно именно это и должны делать в совокупности реализации компонентов: передатчик **T** со спецификаций $\mathbf{T}_0$ вычисляет функции $\boldsymbol{g}$ и $\boldsymbol{h}$, а приёмник **F** со спецификаций $\mathbf{F}_0$ вычисляет функцию $\boldsymbol{f}$. Каждый компонент рассматривается в двух семантиках. В одной

семантике конформна только сама спецификация, в другой – все показанные реализации. Для передатчика **T** первая семантика разрешает одновременную выдачу только в один выходной порт, а вторая – в оба выходных порта. Для приемника **F** первая семантика разрешает одновременный прием только из одного входного порта, а вторая – из обоих входных портов. Клетки с темным фоном соответствуют случаям, когда композиция реализаций компонентов конформна спецификации системы, а клетки со светлым фоном – когда это не так: система принимает аргументы, но не выдает результата. Мы видим, что не всякое сочетание семантик компонентов **T** и **F** гарантирует согласованность спецификаций компонентов со спецификацией системы. Из четырёх вариантов, одно ошибочное, а три правильных.

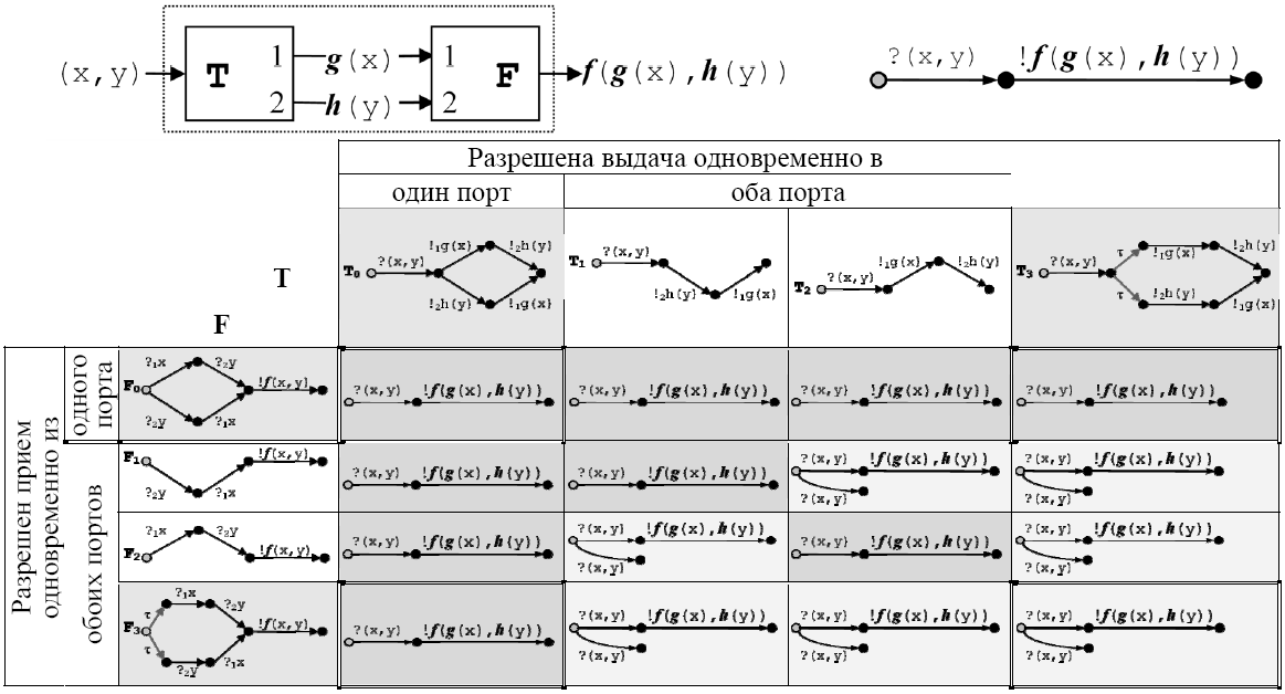


Рис.6. Верификация декомпозиции системных требований

Как же верифицировать декомпозицию? На рисунке для каждого компонента **T** и **F** более темным фоном клетки показаны монотонно преобразованные спецификации в зависимости от семантики, в которой рассматривалась исходная спецификация. Двойной рамкой обведены те клетки таблицы, в которых нарисована как раз композиция монотонно преобразованных спецификаций, но только в сокращённом виде, без τ-переходов. В клетках с темным фоном и двойной рамкой получается композиция спецификаций, которая конформна заданной спецификации, а в одной клетке со светлым фоном и двойной рамкой – неконформная композиция спецификаций. В этом последнем случае спецификации компонентов и системы не согласованы.

Решение второй задачи (генерация спецификации системы) преследует три цели. 1) Полученная спецификация системы используется как документ, описывающий систему с точки зрения её пользователей. 2) Такая спецификация может рассматриваться как техническое задание разработчику системы, в том числе, при возможных дальнейших модификациях системы (её компонентов или даже самой схемы композиции). 3) По спецификации системы могут генерироваться системные (комплексные) тесты.

Остановимся немного подробнее на последней цели. Теоретически, если компоненты полностью протестированы на конформность своим спецификациям, нам нет нужды тестировать саму систему. Вместо этого достаточно либо убедиться в правильном соотношении имеющихся спецификаций системы и её компонентов (первая задача), либо просто сгенерировать отсутствующую спецификацию системы по имеющимся

спецификациям компонентов (первые две цели второй задачи). Однако, в связи с тем, что полный тестовый набор, как правило, бесконечен, а реализация может быть недетерминирована, всякое практическое тестирование оказывается не полным и, следовательно, мы не можем быть стопроцентно уверены, что автономное тестирование выловило все ошибки в компонентах. Именно поэтому на практике всегда остаётся нужда в системных тестах. В частности, системное тестирование создаёт такие режимы взаимодействия компонентов, которые, с одной стороны, ближе к режимам реальной эксплуатации системы, а, с другой стороны, часто не воспроизводятся в практическом автономном тестировании (или это очень трудно сделать).

С системным тестированием связаны две проблемы. Во-первых, косая композиция спецификаций должна удовлетворять тем требованиям, которые делают возможным алгоритмическую генерацию тестов. Для этого приходится налагать на исходные спецификации дополнительные ограничения. Во-вторых, системное тестирование должно быть безопасным. Как показано в [3], без дополнительных гипотез о возможных реализациях компонентов, кроме их безопасности, или специальных средств проверки безопасности композиция реализаций компонентов может оказаться опасной для практически любой спецификации системы, если такие компоненты вообще взаимодействуют друг с другом. В результате системное тестирование либо опасно, либо сводится к автономному тестированию отдельных компонентов, которые не взаимодействуют друг с другом.

В теории монотонности мы имеем дело с конформностью и композицией. Конформность определяется в рамках теории трасс наблюдений, а композиция определяется для LTS-моделей. При этом трасс наблюдений (**R**-трасс) недостаточно для определения композиции: одной модели трасс наблюдений соответствует множество различных LTS, в том числе LTS, дающих при композиции существенно разные результаты с точки зрения конформности.

Тем не менее, в трассовой теории также можно определить композицию моделей, имеющую тот же смысл, если использовать другие трассы. В [3] вводятся трассы, которые названы ϕ-трассами, и определяется композиция ϕ-трасс таким образом, что ϕ-трассы композиции LTS-моделей совпадают с композицией ϕ-трасс этих моделей. Это свойство названо *аддитивностью* ϕ-трасс относительно композиции. Заметим, что ϕ-трассы не являются, вообще говоря, трассами наблюдений, по крайней мере, в **R**/**Q**-семантике. Вместе с тем, по ϕ-трассам LTS-модели можно получить все её **R**-трассы, хотя обратное неверно. Это свойство названо *генеративностью* ϕ-трасс.

ϕ-трассы похожи на трассы готовности (*ready traces*), но имеют два отличия. 1) Вместо множества готовности (*ready set*) используется его дополнение – ***ref***-множество, то есть множество действий, которые не могут выполняться в данном стабильном состоянии. Это сделано по аналогии с отвергаемым множеством (*refusal set*) в трассах с отказами (*failure traces*) и не носит принципиального характера. 2) Также вводится ***gamma***-множество: множество (непосредственно) разрушающих действий, определённых в данном стабильном состоянии. Это сделано для удобства построения теории. Пару (***ref***-множество, ***gamma***-множество) называется ϕ-символом, и ϕ-трасса определяется как последовательность внешних действий и ϕ-символов. Понятно, что по ϕ-трассам LTS-модели легко вычисляются её трассы готовности.

Теория ϕ-трасс позволяет определить ϕ-модель как множество ϕ-трасс с определенным набором свойств, эквивалентную модели трасс наблюдений, LTS- и RTS-моделям, определить композицию ϕ-моделей и построить монотонное преобразование ϕ-моделей.

10. Заключение

Предлагаемые в статье параметризованная **R**/**Q**-семантика взаимодействия, гипотеза о безопасности, безопасная конформность и метод генерации тестов составляют основу теории конформности, которая обобщает многие встречающиеся в литературе и используемые на практике конформности и методы их тестирования. В частности, *failure trace semantics*

[12,15,16,20,23,26] и семантика популярного отношения ***ioco*** [25,26] являются частными случаями **R**/**Q**-семантики.

Теория развивается по нескольким направлениям. В частности: 1) введение в модель приоритетов и соответствующая модификация конформности и генерации тестов [6], 2) распространение предлагаемого подхода на симуляции – конформности, основанные не только на трассах наблюдений, но и на соответствии состояний реализации и спецификации [9], 3) обобщенные семантики взаимодействия, допускающие наблюдение отказов, не обязательно совпадающих с множеством разрешаемых действий (и даже не обязательно вложенных в него) [5].

Другое направление исследований связано с применением этой теории на практике. Основная задача, решаемая здесь, – это поиск практически приемлемых ограничений на семантику, реализацию и/или спецификацию, а также дополнительных тестовых возможностей, которые позволяли бы проводить полное тестирование за конечное время. Сюда относятся ограничения на размер реализации, ограничения на недетерминизм реализации, возможность наблюдения текущего состояния реализации в процессе тестирования и использование программ-медиаторов, осуществляющих преобразование тестовых воздействий и наблюдений [7,8,9].

Литература

1. Бурдонов И.Б., Косачев А.С., Кулямин В.В. Формализация тестового эксперимента. «Программирование», 2007, No. 5.
2. Бурдонов И.Б., Косачев А.С., Кулямин В.В. Теория соответствия для систем с блокировками и разрушением. «Наука», 2008.
3. Бурдонов И.Б. Теория конформности для функционального тестирования программных систем на основе формальных моделей. Диссертация на соискание учёной степени д.ф.-м.н., Москва, 2008. http://www.ispras.ru/~RedVerst/RedVerst/Publications/TR-01-2007.pdf
4. Бурдонов И.Б., Косачев А.С. Эквивалентные семантики взаимодействия // Труды ИСП РАН. – 2008. – №14.1. – С. 55-72.
5. Бурдонов И.Б., Косачев А.С. Обобщенные семантики тестового взаимодействия // Труды ИСП РАН. – 2008. – №15. – С. 69-106.
6. Бурдонов И.Б., Косачев А.С. Системы с приоритетами: конформность, тестирование, композиция // Программирование. – 2009. – №4. – С. 24-40.
7. Бурдонов И.Б., Косачев А.С. Полное тестирование с открытым состоянием ограниченно недетерминированных систем // Программирование. – 2009. – №6. – С. 3-18.
8. Бурдонов И.Б., Косачев А.С. Тестирование с преобразованием семантик // Труды ИСП РАН. – 2009. – №17. – С. 193-208.
9. Бурдонов И.Б., Косачев А.С. Тестирование конформности на основе соответствия состояний // Труды ИСП РАН. – 2010. – №18.
10. Кулямин В.В. Технологии программирования. Компонентный подход. – М.: Интернет-Университет Информационных технологий; БИНОМ. Лаборатория знаний, 2007. – 463 с.
11. Липаев В.В. Тестирование крупных комплексов программ на соответствие требования. М., ИПЦ «Глобус», 2008.
12. Baeten J.C.M. Procesalgebra. Programmatuurkunde. Kluwer. Deventer. In Dutch. 1986.
13. van der Bijl M., Rensink A., Tretmans J. Compositional testing with ioco. Formal Approaches to Software Testing: Third International Workshop, FATES 2003, Montreal, Quebec, Canada, October 6th, 2003. Editors: Alexandre Petrenko, Andreas Ulrich ISBN: 3-540-20894-1. LNCS volume 2931, Springer, pp. 86-100.
14. van der Bijl M., Rensink A., Tretmans J. Component Based Testing with ioco. CTIT Technical Report TR-CTIT-03-34, University of Twente, 2003.

15. van Glabbeek R.J. The linear time – branching time spectrum. In J.C.M. Baeten and J.W. Klop, editors, CONCUR'90, Lecture Notes in Computer Science 458, Springer-Verlag, 1990, pp 278–297.
16. van Glabbeek R.J. The linear time - branching time spectrum II; the semantics of sequential processes with silent moves. Proceedings CONCUR '93, Hildesheim, Germany, August 1993 (E. Best, ed.), LNCS 715, Springer-Verlag, 1993, pp. 66-81.
17. Heerink L. Ins and Outs in Refusal Testing. PhD thesis, University of Twente, Enschede, The Netherlands, 1998.
18. Hoare C.A.R. An axiomatic basis for computer programming. Communications of the ACM, 12(10):576–585, October 1969.
19. Jard C., Jéron T., Tanguy L., Viho C. Remote testing can be as powerful as local testing. In Formal methods for protocol engineering and distributed systems, FORTE XII/ PSTV XIX' 99, Beijing, China, J. Wu, S. Chanson, Q. Gao (eds.), pp. 25-40, October 1999.
20. Langerak R. A testing theory for LOTOS using deadlock detection. In E.Brinksma, G.Scollo, and C.A.Vissers, editors, Protocol Specification, Testing, and Verification IX, pages 87–98. North-Holland, 1990.
21. Milner R. Modal characterization of observable machine behaviour. In G. Astesiano & C. Bohm, editors: Proceedings CAAP 81, LNCS 112, Springer, pp. 25-34.
22. Milner R. Communication and Concurrency. Prentice-Hall, 1989.
23. Phillips I. Refusal testing. Theoretical Computer Science, 50(2):241-284, 1987.
24. Revised Working Draft on "Framework: Formal Methods in Conformance Testing". JTC1/SC21/WG1/Project 54/1 // ISO Interim Meeting / ITU-T on, Paris, 1995.
25. Tretmans J. Conformance testing with labelled transition systems: implementation relations and test generation. Computer Networks and ISDN Systems, v.29 n.1, p.49-79, Dec. 1996.
26. Tretmans J. Test Generation with Inputs, Outputs and Repetitive Quiescence. In: Software-Concepts and Tools, Vol. 17, Issue 3, 1996.