ФОРМАЛИЗАЦИЯ ТЕСТОВОГО ЭКСПЕРИМЕНТА –II

Игорь Бурдонов <igor@ispras.ru>, Александр Косачев kos@ispras.ru

Аннотация. Статья развивает подход к тестированию, рассмотренный в [1]. Предлагается формальная модель тестового взаимодействия самого общего вида и конформность типа редукции, для которых зависимость между ошибками практически отсутствует. Показывается, что многие известные конформности в различных семантиках взаимодействия являются частными случаями этой общей модели. Статья посвящена проблеме зависимости между ошибками, определяемыми спецификацией, и связанной с ней проблеме оптимизации тестов. Между ошибками имеется зависимость, если существует такое строгое подмножество ошибок, что любая неконформная реализация (то есть реализация, в которой есть какая-нибудь ошибка) содержит ошибку из этого подмножества. Соответственно, достаточно, чтобы тесты обнаруживали ошибки только из этого подмножества. В предлагаемой общей модели зависимость между ошибками может возникать, когда в качестве класса тестируемых реализаций выбирается то или иное строгое подмножество класса всех реализаций. Частные семантики взаимодействия и/или различные гипотезы о реализации (в частности, гипотезы о безопасности) как раз и предполагают, что тестируемая реализация не любая, а относится к некоторому подклассу (безопасных) реализаций.

1. Введение

Правильность исследуемой системы в самом широком смысле понимается как её соответствие заданным требованиям. Для верификации (проверки) этого соответствия с помощью формальных методов, объекты и отношения реального мира отображаются в модельные, математические объекты и отношения. Модель исследуемой системы называется реализацией, модель требований – спецификацией, а соответствие требованиям отображается в модельную конформность. Последняя понимается как обычное математическое соответствие, то есть подмножество декартового произведения множеств реализаций и спецификаций.

Спецификация и конформность считаются заданными. Что касается реализации как модели исследуемой системы, то *тестовая гипотеза* предполагает, что такая модель существует для каждой исследуемой системы [8].

Если реализация, как модель исследуемой системы, известна, то возможна статическая (аналитическая) верификация, которая сводится к проверке того, что пара формальных моделей <реализация, спецификация> принадлежит допустимому множеству таких пар, определяемому отношением конформности.

Что делать, если реализация неизвестна (или её слишком сложно построить по исследуемой системе)? В этом случае возникает необходимость в тестировании, которое понимается как динамическая верификация конформности, то есть её проверка в процессе экспериментов. Конечно, для того, чтобы тестирование было возможным, сама конформность должна быть выражена в терминах взаимодействия реализации с окружающей средой. Тест взаимодействует с системой, подменяя собой окружение.

В данной работе мы будем рассматривать не любое тестирование, а только такое, которое основано на трёх предположениях.

Первое предположение. Мы будем рассматривать только *дискретное* тестовое взаимодействие, которое сводится к последовательности дискретных событий двух видов: тестовых воздействий на реализацию и наблюдений над поведением реализации. Эту последовательность мы будем называть *трассой*. Отметим, что в общем случае не всякое поведение реализации может наблюдаться в тестовом эксперименте, то есть в реализации могут происходить события, которые не наблюдаемы и, тем самым, не различимы между собой и не входят в трассу. Такое ненаблюдаемое поведение реализации традиционно обозначается символом τ и называется τ-активностью.

Дискретное взаимодействие моделируется с помощью, так называемой, машины тестирования, внутри которой находится реализация. Тестовому воздействию соответствует нажатие той или иной кнопки на клавиатуре машины, а наблюдению – появление его символа на экране дисплея. Таким образом, трасса – это последовательность кнопок и наблюдений. Тест понимается как инструкция оператору машины, в которой указывается, что оператор должен делать после той или иной трассы: нажимать кнопки (и какие кнопки) и/или ждать наблюдений.

Всё, что мы можем узнать о реализации с помощью тестирования, – это множество её трасс. Наблюдение в эксперименте некоторой трассы предполагает, что все её префиксы наблюдались в этом же эксперименте в более ранние моменты времени. Поэтому множество трасс реализации префикс-замкнуто. Тестовый эксперимент может быть пустым: ни одна кнопка не нажимается, и наблюдения не ожидаются. В этом случае естественно считать, что наблюдается пустая трасса. Тем самым, в каждой реализации есть пустая трасса, то есть множество трасс реализации не пусто. Поскольку тестовое воздействие (нажатие кнопки) не зависит от самой реализации (оператор в любой момент времени может нажать любую кнопку), продолжение трассы реализации кнопкой также даёт трассу реализации. Это означает, что множество трасс реализации вместе с каждой трассой σ содержит и все трассы вида $\sigma\rho$, где ρ – последовательность кнопок.

Спецификацию теперь можно понимать как описание того, какие множества трасс реализаций правильные (конформные), а какие нет. В общем случае, если реализация имеет множество трасс I, то конформна или неконформна не каждая отдельная трасса $\sigma \in I$, а всё множество трасс I целиком.

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

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

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

При тестировании выполняется прогон некоторых тестов из некоторого набора тестов при некоторых погодных условиях. Результатом тестирования является множество X трасс, наблюдаемых во всех этих тестовых экспериментах. Выносится вердикт ***pass*** (проходит) или ***fail*** (ошибка). Набор тестов *значимый*, если каждая конформная реализация его проходит, *исчерпывающий*, если каждая неконформная реализация его не проходит (обнаруживается ошибка), и *полный*, если он значимый и исчерпывающий. Заметим, что X – это не обязательно множество всех трасс реализации. Если спецификация утверждает, что любая реализация с большим множеством трасс $I \supseteq X$ неконформна, то значимый набор тестов может (хотя и не обязан), а полный набор тестов должен выносить вердикт ***fail*** при наблюдении множества трасс X (или любого его надмножества). Если спецификация утверждает, что любая реализация с большим множеством трасс $I \supseteq X$, наоборот, конформна, то исчерпывающий (и полный) набор тестов может (хотя и не обязан), а полный набор тестов должен выносить вердикт ***pass*** при наблюдении множества трасс X (или любого его надмножества).

Второе предположение. В настоящей работе мы ограничимся только теми конформностями, которые отвечают *принципу независимости трасс*: любая трасса реализации конформна или неконформна независимо от других её трасс. Конформности такого типа называются *редукциями*. Не является редукцией, например, конформность, которая разрешает реализации иметь как трассу σ_1, так и трассу σ_2, но не обе одновременно. Принцип независимости исключает из рассмотрения конформности типа симуляций, которые основаны на соответствии состояний реализации и спецификации, а также конформности, которые требуют обязательного наличия в реализации тех или иных наблюдений после тех или иных трасс.

Для редукции можно считать, что спецификация S (прямо или косвенно) определяет множество *разрешаемых* трасс $\boldsymbol{tr}(S)$. Если реализация имеет множество трасс I, то конформность означает вложенность $I \subseteq \boldsymbol{tr}(S)$ и является частичным (нестрогим) порядком (рефлексивное, симметричное и транзитивное отношение). Трасса $\sigma \notin \boldsymbol{tr}(S)$ называется *ошибкой*.

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

Третье предположение. На практике, естественно, используются только конечные тесты, точнее, тесты, которые заканчиваются за конечное время. При дискретном взаимодействии это означает, что при задании спецификации и генерации тестов используются только конечные трассы. Заметим, что для спецификации, основанной на конечных трассах, бесконечные тестовые эксперименты ничего не добавляют. Однако в общем случае это не так. Например, рассмотрим две реализации, в которых возможны только два наблюдения: *x* и *y*. Кнопки не используются. В одной реализации есть бесконечная цепочка *x*. В другой реализации такой бесконечной цепочки нет, но есть бесконечный «веер» конечных цепочек *x*. В обеих реализациях нет перехода по *y*. При конечных тестовых экспериментах эти две реализации неразличимы. Для спецификации, в которой наблюдение *y* считается ошибкой после любого числа *x*, эти реализации обе конформны. В то же время бесконечный тестовый эксперимент позволяет эти реализации различить: в первой реализации есть бесконечная трасса *x*, а во второй нет. Если допускаются бесконечные трассы, то спецификация может трактовать бесконечную цепочку *x* как ошибку. Разумеется, такая ошибка не может быть найдена за конечное время.

По определению любая реализация, содержащая ошибку, неконформна. В то же время, кроме ошибок, определяемых спецификацией, то есть трасс, не принадлежащих множеству разрешаемых трасс $\boldsymbol{tr}(S)$, могут быть другие трассы (принадлежащие $\boldsymbol{tr}(S)$), которые, тем не менее, не встречаются в конформных реализациях. Такие трассы будем называть неконформными. После этого под ошибкой мы будем понимать любую неконформную трассу, а ошибки, определяемые спецификацией (трассы не из $\boldsymbol{tr}(S)$), будем называть ошибками 1-го рода. Ошибка 2-го рода – это неконформная трасса, не являющаяся ошибкой 1-го рода, то есть принадлежащая $\boldsymbol{tr}(S)$.

В данной статье рассматривается проблема зависимости между ошибками и тесно связанная с ней проблема оптимизации полного набора тестов. Будем говорить, что из множества ошибок *A* следует множество ошибок *B* и обозначать $A \rightarrow B$, если любая реализация, в которой есть ошибка из *A*, содержит ошибку из *B*. Если $A \rightarrow B$, то вместо тестов, которые ловят ошибки из *A* можно использовать тесты, которые ловят ошибки из *B*. Если *A* – это множество всех ошибок, то $B \subset A$ и, очевидно, $B \rightarrow A$. Если также $A \rightarrow B$, то множества *A* и *B* эквивалентны (обозначается $A \sim B$). Одним из таких подмножеств ошибок, эквивалентных множеству всех ошибок, является, конечно, множество ошибок 1-го рода. Однако могут существовать и другие множества ошибок, эквивалентные множеству ошибок 1-го рода, в том числе его строгие подмножества. Бывает и так, что множество ошибок 1-го рода бесконечно, но существует эквивалентное ему конечное множество ошибок. Это даёт возможность существенной оптимизации тестов.

Один вид такой зависимости между ошибками присущ любой конформности типа редукции для любого дискретного взаимодействия. Во-первых, любая реализация, содержащая трассу σ, содержит и трассу $\sigma\rho$, где ρ – последовательность кнопок. Поэтому, если $\sigma\rho$ ошибка, то σ тоже ошибка. Поэтому, если $\sigma\rho \in A$, то $A \rightarrow A \cup \{\sigma\}$. Во-вторых, множество трасс реализации префикс-замкнуто. Поэтому если ошибка μ является префиксом трассы σ (будем обозначать это $\mu \leq \sigma$), то σ тоже ошибка. Поэтому, если $\mu \in A$, то $A \rightarrow A \backslash \{\sigma\}$. Это даёт возможность следующей оптимизации тестов: для полноты тестирования достаточно обнаруживать только такие ошибки, которые минимальны по префиксности во множестве всех ошибок (а не только ошибок 1-го рода), такие ошибки не заканчиваются кнопками. Множество таких ошибок эквивалентно множеству ошибок 1-го рода и, тем самым, множеству всех ошибок.

В то же время существует много различных конформностей типа редукции, для которых между ошибками имеются и другие зависимости. Нахождение таких зависимостей и связанная с этим оптимизация тестов иногда представляют собой трудную задачу [например, 6,7].

Цель данной статьи – определить общую природу зависимостей между ошибками. Для этого мы формально определим общую модель дискретного взаимодействия и общую конформность типа редукции. Мы покажем следующее. Во-первых, для такой общей редукции не существует зависимости между ошибками, кроме указанной выше. Во-вторых, другие конформности типа редукции являются частным случаем общей редукции, то есть сводятся к ней. При этом сужается класс рассматриваемых спецификаций и тестируемых реализаций. В-третьих, сужение класса спецификаций не влияет на зависимость между ошибками, тогда как сужение класса тестируемых реализаций влечет появление дополнительных зависимостей между ошибками [например, 6,7]. Важно отметить, что каждая такая частная редукция определяет некий естественный для неё класс тестируемых реализаций. Однако на практике часто используются дополнительные ограничения на тестируемые реализации, что, в свою очередь, также приводит к появлению дополнительных зависимостей между ошибками [например, 16,17]. Иными словами, мы сведём проблему зависимостей между ошибками на классе тестируемых реализаций, естественном для той или иной частной редукции, к общей проблеме зависимостей между ошибками, возникающей как результат сужения класса тестируемых реализаций.

2. Общая модель

В этом разделе мы формально определим общую модель дискретного взаимодействия и общую конформность типа редукции.

2.1. Семантика взаимодействия

Для многих семантик взаимодействия частного вида между кнопками и наблюдениями существует та или иная предустановленная связь. Некоторые из таких семантик мы рассмотрим ниже. В общем же случае никакой предустановленной связи между кнопками и наблюдениями мы предполагать не будем. Другое дело, что в конкретной реализации такая связь может быть.

Будем считать, что заданы два непересекающихся универсума символов: $\boldsymbol{B}$ – тестовых воздействий (кнопок – ***b****uttons*) и $\boldsymbol{O}$ – наблюдений (***o****bservations*). Такую семантику будем называть ***B/O****-семантикой*. Трасса – это последовательность в алфавите $\boldsymbol{B}\cup\boldsymbol{O}$. Семантику будем называть *конечной*, если суммарное число кнопок и наблюдений конечны.

2.2. Машина тестирования

B/***O***-семантика моделируется машиной тестирования, представляющей собой «чёрный ящик», внутри которого находится реализация. Машина снабжена клавиатурой для управления и дисплеем для наблюдения.

Клавиатура представляет собой множество кнопок $\boldsymbol{B}$. Тестовое воздействие осуществляется нажатием той или иной кнопки на клавиатуре. Когда нажимается кнопка, машина тестирования передаёт в реализацию однократный сигнал о соответствующем тестовом воздействии и дожидается ответного сигнала о том, что реализация «приняла к сведению» это тестовое воздействие, после чего машина может передавать в реализацию

сигнал о следующем тестовом воздействии. Кнопка не фиксируется (автоматически отжимается), что даёт возможность оператору машины нажимать следующую (другую или ту же самую) кнопку. Одновременно можно нажимать только одну кнопку, соответствующую ровно одному тестовому воздействию.

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

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

В то же время рестарт позволяет выполнить не более чем счётное число тестовых экспериментов, тем самым, для не более чем счётного числа погодных условий. Для того чтобы обойти это ограничение, машина тестирования может быть снабжена не кнопкой рестарта, а кнопкой *репликации*. Однократное нажатие такой кнопки создаёт множество копий машины тестирования произвольной мощности. Тестирование происходит с каждой копией машины независимым образом, то есть на каждой копии выполняется свой тестовый эксперимент. Для каждой копии фиксируется свой вариант погодных условий. Гипотеза о глобальном тестировании предполагает, что для каждого варианта погодных условий при репликации создается, по крайней мере, одна копия машины.

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

2.3. Реализация

Для конформности типа редукции реализация, фактически, сводится к множеству её трасс. Такая *трассовая модель реализации* формально определяется как множество $I\subseteq(\boldsymbol{B}\cup\boldsymbol{O})^*$, которое: 1) не пусто, 2) префикс-замкнуто, 3) вместе с каждой трассой σ содержит и все трассы вида $\sigma\rho$, где ρ – последовательность кнопок.

Для компактного задания множества трасс, в частности, для задания бесконечного множества трасс конечным образом, используется модель LTS (*Labelled Transition System*). Она представляет собой ориентированный граф, вершины которого называются состояниями, одно состояние выделено в качестве начального, дуги помечены символами из $\boldsymbol{B}\cup\boldsymbol{O}$ и называются переходами. Ненаблюдаемая τ-активность понимается как цепочка элементарных τ-событий, каждое из которых изображается переходом, помеченным

символом τ. LTS-реализацию будем называть *конечной*, если конечно число её состояний, достижимых из начального состояния.

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

Маршрутом называется последовательность смежных переходов, когда начало любого перехода, кроме первого, совпадает с концом предыдущего перехода. Трасса реализации – это последовательность пометок переходов маршрута, начинающегося в начальном состоянии, с пропуском символа τ.

Будем считать, что переходы выполняются мгновенно[1]. В каждом состоянии реализация ожидает некоторое конечное время, после чего она должна выполнить один из выходящих переходов. Эти предположения обычно называют *progress assumption* [11]. Также предполагается, что за конечное время реализация может пройти только конечный маршрут. Это означает, что реализация рассматривается как дискретная система [11]. Для этого достаточно, например, считать, что время ожидания в каждом состоянии ограничено снизу некоторой константой, большей нуля.

Множество трасс LTS-реализации является трассовой моделью реализации и, наоборот, для любой трассовой модели реализации существует LTS с таким же множеством трасс.

Очевидно, что для каждой трассы σ существует наименьшая по множеству трасс реализация, содержащая эту трассу σ, – это множество трасс $\{\mu\rho \mid \mu \le \sigma \,\&\, \rho \in \boldsymbol{B}^*\}$. Соответствующая LTS-реализация изображена на Рис. 1.

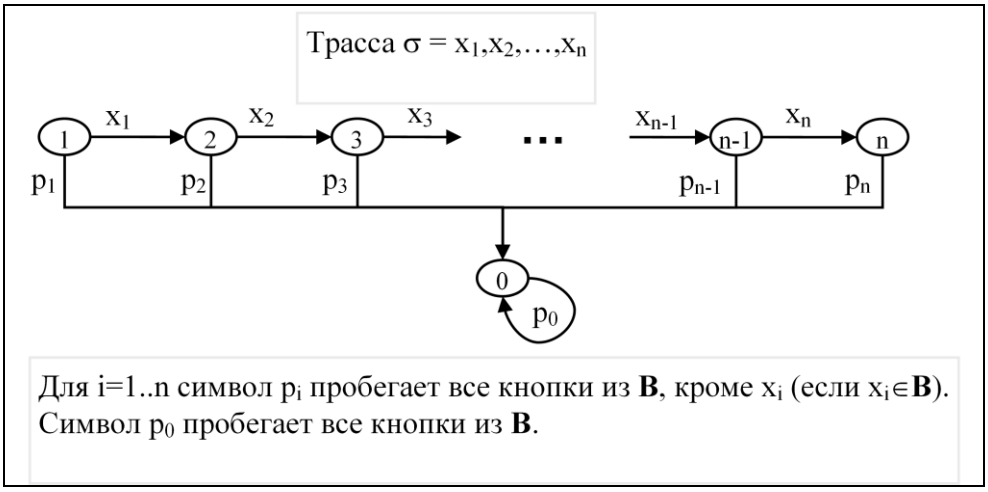

Рис. 1.

[1] Достаточно считать, что переход выполняется за конечное время, но если это переход по наблюдению, то символ наблюдения появляется на экране дисплея машины тестирования в начале (не в середине и не в конце) этого перехода.

2.4. Взаимодействие с реализацией

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

Что касается ненаблюдаемого поведения реализации, то мы исходим из *основного допущения о τ-активности*: между любыми двумя наблюдениями (и до первого в трассе наблюдения) в реализации может быть любая конечная τ-активность [11]. В терминах LTS это означает, что между двумя переходами по наблюдению (и перед первым таким переходом) реализация может выполнить любое конечное число τ-переходов. Мы распространяем это допущение также на тестовые воздействия: реализация может выполнить любое конечное число τ-переходов между любыми двумя переходами по наблюдениям или кнопкам (и перед первым таким переходом).

Особенностью нашей модели взаимодействия является *приоритет тестового воздействия над поведением реализации*, как наблюдаемым, так и ненаблюдаемым. Если после получения в тестовом эксперименте трассы σ оператор не нажимает кнопку, а ждёт наблюдений, то может быть получено любое наблюдение, которое в реализации есть после трассы σ, то есть может быть получена любая имеющаяся в реализации трасса σ*u*, где *u* – наблюдение. Кроме того, если после трассы σ в реализации есть бесконечная τ-активность (*дивергенция* как бесконечная цепочка τ-переходов), то никаких наблюдений может и не быть. Если же сразу после наблюдения трассы σ оператор нажимает кнопку *p*, то реализация обязана выполнить переход по кнопке *p*, будет получена трасса σ*p*. Однако по основному допущению о τ-активности между переходом по последнему символу (наблюдению или кнопке) трассы σ и переходом по следующему наблюдению *u* в трассе σ*u* или следующей кнопке *p* в трассе σ*p* реализация может выполнить любое конечное число τ-переходов. Итак, реализация обязана «принять к сведению» тестовое воздействие через конечное время после нажатия соответствующей кнопки. В то же время мы никак не оговариваем, что означает это «принятие к сведению», допускается и простое игнорирование тестового воздействия, что в LTS моделируется переходом-петлёй по этой кнопке (отсутствие перехода по кнопке интерпретируется как наличие такой петли).

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

Заметим, что при таком протоколе взаимодействия появление на экране дисплея символа кнопки в момент нажатия кнопки, фактически, эквивалентно его появлению в момент совершения реализацией перехода по этой кнопке. Именно поэтому наблюдаемая на

экране дисплея трасса совпадает с трассой маршрута, который реализация за это время проходит.

При наличии в состоянии нескольких переходов, которые реализация может выполнять, выбирается один из них недетерминированным образом. Также при нажатой кнопке выбор числа τ-переходов, которые выполняются до перехода по кнопке, происходит недетерминировано. Оба этих выбора понимаются как выборы в зависимости от погодных условий. Гипотеза о глобальном тестировании гарантирует возможность перебора всех возможных погодных условий.

В любом сеансе взаимодействия с реализацией через машину тестирования на экране дисплея наблюдается некоторая трасса реализации, а также (в более ранние моменты времени) все ее префиксы.

2.5. Оператор машины тестирования

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

Если тест разрешает оператору нажимать кнопку p после трассы μ, то предполагается, что оператор может нажать эту кнопку через любое время после получения трассы μ. Тем самым, оператору *не запрещено* выдерживать любые паузы после получения тех или иных трасс, в том числе перед нажатием следующей кнопки: в любой момент времени он может устроить себе «перерыв на чай». Это означает, что когда оператор нажимает кнопку p спустя какое-то время после трассы μ, а после нажатия кнопки ждёт ещё какое-то время, то реально будет получена не трасса μp, а трасса $\mu\pi_1 p\pi_2$, где π_1 и π_2 – последовательности наблюдений.

В то же время для полноты тестирования необходимо, чтобы любая интересующая нас трасса реализации могла наблюдаться при взаимодействии с реализацией через машину тестирования. А для этого оператор должен иметь *возможность* достаточно быстро нажимать кнопки после полученных трасс. Это значит, что задержка между получением трассы и нажатием следующей кнопки хотя бы в одном сеансе тестирования может оказаться меньше, чем время ожидания реализацией в состоянии после трассы. Тогда, нажимая кнопку p сразу после трассы μ, оператор будет наблюдать именно трассу μp. Разумеется, если после этой трассы оператор какое-то время не выключает машину и не нажимает кнопок, то может быть получено продолжение этой трассы последовательностью наблюдений, то есть трасса $\mu p\pi_2$.

2.6. Спецификация и общая редукция

Как было сказано во введении, спецификация явно или неявно определяет множество разрешённых трасс и одновременно его дополнение – множество ошибок 1-го рода. В данной работе под спецификацией будет пониматься произвольное множество конечных трасс, понимаемое просто как множество ошибок 1-го рода. Реализация конформна, если в

ней нет ошибок 1-го рода, то есть трасс спецификации. Такую конформность будем называть далее *общей редукцией*.

Спецификацию как множество ошибок 1-го рода можно задавать с помощью порождающего графа, то есть LTS с выделенными конечными вершинами. LTS-спецификацию будем называть *конечной*, если конечно число её состояний, достижимых из начального состояния. Для некоторых бесконечных множеств ошибок 1-го рода LTS-спецификация может быть конечной. Как известно, для любого порождающего графа существует процедура детерминизации, строящая детерминированный граф, порождающий то же множество последовательностей. Поэтому для любой спецификации существует детерминированная LTS-спецификация, определяющая то же множество трасс. Детерминированность здесь означает, что в каждом достижимом состоянии нет τ-переходов и для каждого символа $x \in \boldsymbol{B} \cup \boldsymbol{O}$ определено не более одного перехода по x из этого состояния. Если LTS-спецификация конечная, то после детерминизации она тоже остаётся конечной.

Спецификация S задаёт класс конформных реализаций, который будем обозначать как $\boldsymbol{C}_S$. Если спецификация содержит пустую трассу, то есть пустая трасса считается ошибкой 1-го рода, то все реализации неконформны, поскольку любая реализация содержит пустую трассу. Если спецификация – это пустое множество, то все реализации конформны.

2.7. Тест

Тестом будем называть множество T конечных трасс. Получение при тестировании любой из этих трасс приводит к вынесению вердикта ***fail***, получение любой другой трассы – к получению вердикта ***pass***. Тест понимается как инструкция оператору машины тестирования. Задача теста – проверить, имеется ли в реализации хотя бы одна из трасс теста: если это так, то тестирование заканчивается и выносится общий вердикт ***fail***.

Обозначим префикс-замыкание множества T последовательностей: $\boldsymbol{pre}(T) = \{\mu \mid \exists \sigma \in T \; \mu \le \sigma\}$.

Нажатие кнопки. В процессе тестирования после получения трассы μ оператор может нажать кнопку p, если трасса μp является префиксом некоторой трассы теста $\mu p \in \boldsymbol{pre}(T)$. Предполагается, что в этом случае хотя бы в одном сеансе тестирования после получения трассы μ оператор нажимает кнопку p, причем достаточно быстро после получения трассы μ. Также предполагается, что если трасса μu, где u наблюдение, является префиксом некоторой трассы теста $\mu u \in \boldsymbol{pre}(T)$, то хотя бы в одном сеансе тестирования после получения трассы μ оператор ожидает наблюдения. Если эти предположения выполнены, то гипотеза о глобальном тестировании гарантирует, что если некоторая трасса $\sigma \in T$ встречается в реализации, то она будет получена хотя бы в одном сеансе тестирования.

Выключение машины. В процессе тестирования после получения трассы μ возможны три случая:

1) $\mu \in \boldsymbol{pre}(T) \setminus T$, то есть μ является строгим префиксом какой-либо трассы из T;
2) $\mu \in T$;
3) $\mu \notin \boldsymbol{pre}(T)$, то есть μ не является префиксом какой-либо трассы из T.

В случае 1 оператор должен продолжать сеанс тестирования, то есть не должен выключать машину тестирования. В случае 2 и 3 оператор должен закончить сеанс

тестирования, то есть должен выключить машину тестирования. При этом выносится вердикт: в случае 2 – ***fail***, в случае 3 – ***pass***.

Трасса, которая может быть получена в некотором сеансе тестирования с данным тестом (не только в конце сеанса), имеет вид либо $\mu\pi$, где μ является префиксом некоторой трассы $\sigma\in T$, а π последовательность наблюдений, либо $\mu\pi_1 p\pi_2$, где μp является префиксом некоторой трассы $\sigma\in T$, p – кнопка, а π_1 и π_2 последовательности наблюдений. Множество таких трасс будем называть расширением теста и обозначать:
$\boldsymbol{exp}(T) = \{\mu\pi \mid \mu\in\boldsymbol{pre}(T) \,\&\, \pi\in\boldsymbol{O}^*\} \cup \{\mu\pi_1 p\pi_2 \mid \mu p\in\boldsymbol{pre}(T) \,\&\, p\in\boldsymbol{B} \,\&\, \pi_1\in\boldsymbol{O}^* \,\&\, \pi_1\in\boldsymbol{O}^*\}$.
Множество трасс, которые могут быть получены в конце сеанса тестирования, равно $(\boldsymbol{exp}(T) \setminus \boldsymbol{pre}(T)) \cup T = \boldsymbol{exp}(T) \setminus (\boldsymbol{pre}(T) \setminus T)$.

Реализация *проходит* тест, если при любом сеансе тестирования (при любых погодных условиях) выносится вердикт ***pass***. Реализация *проходит* набор тестов, если она проходит каждый тест из набора. Для заданного класса реализаций (в частности, для класса всех реализаций) набор тестов (тест) значимый, если каждая конформная реализация из этого класса его проходит, исчерпывающий, если каждая неконформная реализация из этого класса его не проходит, и полный, если он значимый и исчерпывающий.

Тест *детерминированный*, если он однозначно определяет поведение оператора. Это значит, что любая трасса из префикс-замыкания теста в этом префикс-замыкании либо продолжается одной кнопкой и не продолжается наблюдениями, либо не продолжается кнопками:
$\forall\mu\in\boldsymbol{pre}(T)\ (\ |\{p\in\boldsymbol{B} \mid \mu p\in\boldsymbol{pre}(T)\}| = 1 \,\&\, \{u\in\boldsymbol{O} \mid \mu u\in\boldsymbol{pre}(T)\} = \varnothing\ ) \vee \{p\in\boldsymbol{B} \mid \mu p\in\boldsymbol{pre}(T)\} = \varnothing$.

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

2.8. Нормализация спецификации и оптимизация тестов

Как уже было отмечено во введении, кроме ошибок 1-го рода, то есть трасс спецификации, могут быть и другие ошибки – неконформные трассы, то есть трассы, не встречающиеся в конформных реализациях. Ошибки, не являющиеся ошибками 1-го рода, называются ошибками 2-го рода. Для полноты тестирования достаточно обнаруживать только такие ошибки, которые минимальны по префиксности во множестве всех ошибок (а не только ошибок 1-го рода). Такие ошибки будем называть *первичными* ошибками, *вторичная* ошибка – это ошибка, у которой есть строгий префикс, являющийся ошибкой. Первичные ошибки не заканчиваются кнопками. Множество первичных ошибок эквивалентно множеству ошибок 1-го рода и, тем самым, множеству всех ошибок. Оно, очевидно, является наименьшим по вложенности подмножеством ошибок, эквивалентным множеству всех ошибок. Его можно рассматривать как спецификацию, которую будем называть *нормализованной* спецификацией.

Для каждой спецификации S множество всех ошибок строится с помощью систематического применения следующих операций:

1) Если p – кнопка и $\sigma p\in S$, то добавляем в S трассу σ.

2) Если $\mu \in S$ и $\mu < \sigma$, то добавляем в S трассу σ.

Если S^* – множество всех ошибок для спецификации S, то процедура нормализации сводится к удалению неминимальных по префиксности ошибок: если $\mu \in S$, $\sigma \in S$ и $\mu < \sigma$, то удаляем из S трассу σ.

Нормализацию можно выполнить и непосредственно по исходной спецификации S как систематическое применение следующих операций:

1) Если p – кнопка и $\sigma p \in S$, то добавляем в S трассу σ.
2) Если $\mu \in S$, $\sigma \in S$ и $\mu < \sigma$, то удаляем из S трассу σ.

Нормализованные спецификации взаимно-однозначно соответствуют своим классам конформных реализаций: $A = B \Leftrightarrow \boldsymbol{C}_A = \boldsymbol{C}_B$.

Пусть спецификация S нормализованная. Как было отмечено выше, для каждой трассы σ существует наименьшая по множеству трасс реализация, содержащая эту трассу σ, – это множество трасс $\{\mu\rho \mid \mu \le \sigma \ \& \ \rho \in \boldsymbol{B}^*\}$. Отсюда следует, что набор $\boldsymbol{T}$ тестов значимый тогда и только тогда, когда каждая трасса каждого теста из набора имеет в качестве префикса ошибку из S, то есть S коинициально $\cup\boldsymbol{T}$. Набор $\boldsymbol{T}$ тестов исчерпывающий тогда и только тогда, когда каждая трасса из S имеет префикс, являющийся трассой некоторого теста из набора, то есть $\cup\boldsymbol{T}$ коинициально S.

Набор $\boldsymbol{T}$ тестов полный тогда и только тогда, когда S и $\cup\boldsymbol{T}$ взаимно коинициальны. Поскольку S нормализована, условие $\cup\boldsymbol{T}$ коинициально S можно заменить условием вложенности $S \subseteq \cup\boldsymbol{T}$. Действительно, в противном случае найдётся трасса $\mu \in S \setminus \cup\boldsymbol{T}$, а тогда, поскольку $\cup\boldsymbol{T}$ коинициально S, найдётся трасса $\mu_1 < \mu$ такая, что $\mu_1 \in \cup\boldsymbol{T}$, а тогда, поскольку S коинициально $\cup\boldsymbol{T}$, найдётся трасса $\mu_2 \le \mu_1$ такая, что $\mu_2 \in S$, следовательно, в S имеются две ошибки $\mu_2 < \mu$, что противоречит нормализованности S.

Иными словами, набор $\boldsymbol{T}$ тестов полный тогда и только тогда, когда все трассы всех его тестов – это все первичные ошибки (трассы из нормализованной спецификации S) и некоторые их продолжения. Очевидная оптимизация – это удаление таких продолжений, что даёт в итоге набор $\boldsymbol{T}`$ тестов, множество всех трасс всех тестов которого – это нормализованная спецификация: $S = \cup\{\{\sigma\} \mid \sigma \in S\} = \cup\boldsymbol{T}`$. Тем самым, оптимизированный полный набор $\boldsymbol{T}`$ тестов – это покрытие S, а набор примитивных тестов $\{\{\sigma\} \mid \sigma \in S\}$ является одним из возможных разбиений S.

3. Класс реализаций

По разным причинам в качестве тестируемых реализаций рассматриваются не любые реализации, а принадлежащие тому или иному классу реализаций $\boldsymbol{I}$. Это приводит к появлению дополнительных зависимостей между ошибками (в том числе, первичными) и, соответственно, даёт возможность дополнительной оптимизации тестов.

Первое определение эквивалентности спецификаций: Будем говорить, что две спецификации A и B эквивалентны на классе реализаций $\boldsymbol{I}$, если на этом классе спецификации определяют одну и ту же конформность реализаций: $\boldsymbol{I} \cap \boldsymbol{C}_A = \boldsymbol{I} \cap \boldsymbol{C}_B$. Если $\boldsymbol{I}$ – класс всех реализаций, то $\boldsymbol{C}_A \subseteq \boldsymbol{I}$ и $\boldsymbol{C}_B \subseteq \boldsymbol{I}$, поэтому эквивалентность нормализованных спецификаций ($\boldsymbol{C}_A = \boldsymbol{C}_B$) совпадает с равенством ($A = B$). На других подклассах реализаций это, вообще говоря, не верно.

Трассу, встречающуюся в реализациях класса $\boldsymbol{I}$, будем называть *актуальной* на классе $\boldsymbol{I}$. Если $\boldsymbol{I}$ – множество трассовых реализаций, то множество актуальных трасс равно $\cup\boldsymbol{I}$. На классе всех реализаций все трассы актуальны. На других классах реализаций могут быть как актуальные, так и неактуальные трассы. Трасса, которая встречается в конформных реализациях из класса $\boldsymbol{I}$, называется конформной на классе $\boldsymbol{I}$. Это конформная трасса, которая актуальна на классе $\boldsymbol{I}$. Ошибки (в том числе ошибки 1-го рода и 2-го рода) делятся на актуальные и неактуальные. При тестировании реализаций из класса $\boldsymbol{I}$, очевидно, достаточно обнаруживать только актуальные на этом классе ошибки. Множество трасс, конформных на классе $\boldsymbol{I}$, для спецификации S равно $\cup(\boldsymbol{I}\cap\boldsymbol{C}_S)$. Соответственно, множество ошибок, актуальных на классе $\boldsymbol{I}$, равно $\cup\boldsymbol{I} \setminus \cup(\boldsymbol{I}\cap\boldsymbol{C}_S)$.

Это дает нам второе определение эквивалентности спецификаций: Будем говорить, что две спецификации A и B эквивалентны на классе реализаций $\boldsymbol{I}$, если они определяют одно и то же множество актуальных ошибок: $\cup\boldsymbol{I} \setminus \cup(\boldsymbol{I}\cap\boldsymbol{C}_A) = \cup\boldsymbol{I} \setminus \cup(\boldsymbol{I}\cap\boldsymbol{C}_B)$.

На самом деле, два определения эквивалентности спецификаций равносильны:
$\boldsymbol{I}\cap\boldsymbol{C}_A=\boldsymbol{I}\cap\boldsymbol{C}_B \Leftrightarrow \cup\boldsymbol{I} \setminus \cup(\boldsymbol{I}\cap\boldsymbol{C}_A) = \cup\boldsymbol{I} \setminus \cup(\boldsymbol{I}\cap\boldsymbol{C}_B)$.
Докажем это. Сначала покажем, что $\cup(\boldsymbol{I}\cap\boldsymbol{C}_A) = \cup(\boldsymbol{I}\cap\boldsymbol{C}_B) \Leftrightarrow \cup\boldsymbol{I} \setminus \cup(\boldsymbol{I}\cap\boldsymbol{C}_A) = \cup\boldsymbol{I} \setminus \cup(\boldsymbol{I}\cap\boldsymbol{C}_B)$.
Действительно, если $\cup(\boldsymbol{I}\cap\boldsymbol{C}_A) = \cup(\boldsymbol{I}\cap\boldsymbol{C}_B)$, то, очевидно, $\cup\boldsymbol{I} \setminus \cup(\boldsymbol{I}\cap\boldsymbol{C}_A) = \cup\boldsymbol{I} \setminus \cup(\boldsymbol{I}\cap\boldsymbol{C}_B)$.
Покажем, что если $\cup\boldsymbol{I} \setminus \cup(\boldsymbol{I}\cap\boldsymbol{C}_A) = \cup\boldsymbol{I} \setminus \cup(\boldsymbol{I}\cap\boldsymbol{C}_B)$, то $\cup(\boldsymbol{I}\cap\boldsymbol{C}_A) = \cup(\boldsymbol{I}\cap\boldsymbol{C}_B)$. Пусть это не верно, например, трасса $\sigma\in\cup(\boldsymbol{I}\cap\boldsymbol{C}_A) \setminus \cup(\boldsymbol{I}\cap\boldsymbol{C}_B)$. Тогда эта трасса принадлежит некоторой реализации из $\boldsymbol{I}$, которая конформна для A. Но тогда эта реализация не принадлежит $\boldsymbol{C}_B$, то есть в ней есть ошибка из B. Эта ошибка принадлежит $\cup(\boldsymbol{I}\cap\boldsymbol{C}_A)$, следовательно, не принадлежит $\cup\boldsymbol{I} \setminus \cup(\boldsymbol{I}\cap\boldsymbol{C}_A)$. Также эта ошибка принадлежит $\cup\boldsymbol{I}$, но не принадлежит $\cup(\boldsymbol{I}\cap\boldsymbol{C}_B)$, следовательно, принадлежит $\cup\boldsymbol{I} \setminus \cup(\boldsymbol{I}\cap\boldsymbol{C}_B)$, что противоречит равенству $\cup\boldsymbol{I} \setminus \cup(\boldsymbol{I}\cap\boldsymbol{C}_A) = \cup\boldsymbol{I} \setminus \cup(\boldsymbol{I}\cap\boldsymbol{C}_B)$. Теперь покажем, что $\boldsymbol{I}\cap\boldsymbol{C}_A=\boldsymbol{I}\cap\boldsymbol{C}_B \Leftrightarrow \cup(\boldsymbol{I}\cap\boldsymbol{C}_A) = \cup(\boldsymbol{I}\cap\boldsymbol{C}_B)$.
Если $\boldsymbol{I}\cap\boldsymbol{C}_A=\boldsymbol{I}\cap\boldsymbol{C}_B$, то, очевидно, $\cup(\boldsymbol{I}\cap\boldsymbol{C}_A) = \cup(\boldsymbol{I}\cap\boldsymbol{C}_B)$. Покажем, что если $\cup(\boldsymbol{I}\cap\boldsymbol{C}_A) = \cup(\boldsymbol{I}\cap\boldsymbol{C}_B)$, то $\boldsymbol{I}\cap\boldsymbol{C}_A=\boldsymbol{I}\cap\boldsymbol{C}_B$. Пусть это не верно, тогда есть некоторая реализация, принадлежащая, например, $\boldsymbol{I}\cap\boldsymbol{C}_A \setminus \boldsymbol{I}\cap\boldsymbol{C}_B$. Тогда эта реализация принадлежит $\boldsymbol{I}$ и не принадлежит $\boldsymbol{C}_B$, следовательно, в ней есть некоторая ошибка из B. Тем самым, эта ошибка принадлежит $\cup(\boldsymbol{I}\cap\boldsymbol{C}_A)$. Но эта ошибка не может принадлежать $\cup(\boldsymbol{I}\cap\boldsymbol{C}_B)$, что противоречит равенству $\cup(\boldsymbol{I}\cap\boldsymbol{C}_A) = \cup(\boldsymbol{I}\cap\boldsymbol{C}_B)$.

Теперь пусть $\boldsymbol{I}$ – это отображение, задающее для каждой спецификации S класс тестируемых реализаций $\boldsymbol{I}_S$. Будем говорить, что для отображения $\boldsymbol{I}$ спецификация B *может быть использована вместо спецификации* A, если 1) $\boldsymbol{I}_A \subseteq \boldsymbol{I}_B$, 2) $\boldsymbol{I}_A\cap\boldsymbol{C}_A=\boldsymbol{I}_A\cap\boldsymbol{C}_B$. Первое условие говорит о том, что любая реализация, которую мы могли тестировать для проверки конформности спецификации A, можно тестировать для проверки конформности спецификации B. Второе условие (эквивалентность спецификаций на классе $\boldsymbol{I}_A$) говорит о том, что спецификации A и B определяют одинаковую конформность реализаций на классе тестируемых реализаций для спецификации A.

Ошибка обнаруживается набором тестов, если она является трассой одного из тестов набора. Любой набор тестов, который полон на классе тестируемых реализаций $\boldsymbol{I}$, очевидно, задаёт множество обнаруживаемых ошибок (множество всех трасс всех его тестов), эквивалентное на классе $\boldsymbol{I}$ множеству всех ошибок.

4. Гипотеза о безопасности

В наших работах [1,2,5,6] мы ввели понятие безопасного тестирования. Это такое тестирование, при котором не проходятся трассы реализации, которые считаются опасными. Гипотеза о безопасности определяет класс реализаций, которые можно безопасно тестировать для проверки конформности данной спецификации.

4.1. Общий вид гипотезы о безопасности

Будем говорить, что задана *гипотеза о безопасности*, если для каждой реализации с множеством трасс I определено префикс-замкнутое подмножество $\boldsymbol{SafeTraces}(I) \subseteq I$ трасс, которые называются безопасными трассами. Тестирование данной реализации называется безопасным, если в процессе него могут получаться только безопасные трассы этой реализации. Реализация безопасна для теста T, если тестирование с помощью этого теста безопасно для этой реализации: $\boldsymbol{exp}(T) \cap I \subseteq \boldsymbol{SafeTraces}(I)$. Каждый тест T определяет класс безопасных реализаций $\boldsymbol{SafeImpl}(T) = \{I \mid \boldsymbol{exp}(T) \cap I \subseteq \boldsymbol{SafeTraces}(I)\}$. Набор тестов $\boldsymbol{T}$ определяет класс реализаций, безопасных для каждого теста из набора: $\boldsymbol{SafeImpl}(\boldsymbol{T}) = \cap\{\boldsymbol{SafeImpl}(T) | T \in \boldsymbol{T}\}$. Спецификация S определяет класс безопасных реализаций как класс реализаций, безопасных для полного теста S или, что то же самое, для набора примитивных тестов, построенных по ошибкам спецификации: $\boldsymbol{SafeImpl}(S) = \boldsymbol{SafeImpl}(\{\{\sigma\} | \sigma \in S\})$. В общем случае, если предполагается безопасное тестирование реализаций из заданного класса $\boldsymbol{I}$, тестироваться будут безопасные реализации из класса $\boldsymbol{I} \cap \boldsymbol{SafeImpl}(S)$.

4.2. Гипотеза о конечном времени ожидания наблюдения

Для того чтобы каждый сеанс тестирования был конечным по времени, нужно, чтобы были конечными времена ожидания кнопок и наблюдений на экране дисплея: 1) кнопка должна появляться на экране дисплея через конечное время после ее нажатия, 2) если оператор ждет наблюдений, то какое-нибудь наблюдение должно появиться на экране дисплея через конечное время.

Первое условие гарантированно выполнено в данной модели взаимодействия с реализацией. А второе условие может и не выполняться. Сформулируем требование к реализации, чтобы выполнялось это второе условие для данного теста.

Гипотеза о наблюдениях – λ-гипотеза[2]: если трасса реализации является префиксом трассы теста и продолжается в префикс-замыкании теста наблюдением, то в реализации она также должна продолжаться каким-нибудь (не обязательно тем же самым) наблюдением при любом поведении реализации. Это означает: 1) в реализации в каждом стабильном состоянии (состоянии, в котором не начинаются τ-переходы) после этой трассы имеется переход по какому-нибудь наблюдению, 2) после трассы нет дивергенции. λ-гипотеза является частным случаем гипотезы о безопасности.

Определим формально множество $\boldsymbol{SafeTraces}_{\lambda}(I)$ безопасных трасс реализации I для λ-гипотезы. λ-трассой реализации будем называть трассу реализации, которая заканчивается в стабильном состоянии, где нет переходов по наблюдениям, или в дивергентном состоянии. Трассу реализации будем называть безопасной, если любой ее строгий префикс, за которым в трассе следует наблюдение, не является λ-трассой.

[2] Символом λ принято обозначать ситуацию, когда возникает *deadlock* или дивергенция [11]

λ-гипотеза не меняет актуальность трасс: все трассы актуальны. λ-гипотеза меняет конформность трасс: на классе безопасных реализаций, определяемом этой гипотезой, трасса μ неконформна, если для каждого наблюдения *u* трасса μ*u* является ошибкой. Для определения первичных ошибок спецификации в случае λ-гипотезы применяется следующая **процедура λ-нормализации спецификации**: систематически применяем три действия:

1) Если для каждого наблюдения *u* трасса σ*u*∈*S*, то добавляем в *S* трассу σ.
2) Если *p* – кнопка и σ*p*∈*S*, то добавляем в *S* трассу σ.
3) Если μ∈*S*, σ∈*S* и μ<σ, то удаляем из *S* трассу σ.

Полученное множество трасс – это множество первичных ошибок в случае λ-гипотезы.

4.3. Гипотеза о разрушении

Другой разновидностью гипотезы о безопасности является, так называемая, гипотеза о разрушении. Под разрушением понимается любое поведение реализации, которое нежелательно во время тестирования [1,2,5]. Причины нежелательности того или иного поведения могут быть самыми разными, мы не налагаем здесь никаких ограничений. Для изображения разрушения в LTS-модели реализации некоторые её переходы по наблюдениям или τ-переходы (ненаблюдаемое поведение) заменяются γ-переходами, то есть переходами, помеченными специальным символом разрушения – γ.

Теперь под моделью реализации мы будем понимать LTS в алфавите с добавленным символом γ. Поскольку нас не интересует поведение реализации после разрушения, под трассой будем понимать последовательность кнопок и наблюдений, быть может, заканчивающуюся разрушением.

Гипотеза о разрушении – γ-гипотеза: для спецификации *S* любая трасса из ***exp***(*S*) не продолжается в реализации разрушением. γ-гипотеза является частным случаем гипотезы о безопасности.

Определим формально множество $\boldsymbol{SafeTraces}_{\gamma}(I)$ безопасных трасс реализации *I* для γ-гипотезы: трассу реализации будем называть безопасной, если любой ее префикс не продолжается в реализации разрушением.

γ-гипотеза меняет актуальность трасс: трасса актуальная, если она не имеет вида μγ, где μ∈***exp***(*S*). γ-гипотеза не меняет конформность актуальных трасс: на классе безопасных реализаций, определяемом этой гипотезой, неконформна та и только та актуальная трасса, префикс которой является ошибкой. Поскольку γ-гипотеза не меняет конформность актуальных трасс, процедура нормализации не требуется: все ошибки спецификации являются первичными.

λ- и γ-гипотезы в совокупности определяют класс безопасных реализаций $\boldsymbol{SafeImpl}_{\lambda\gamma}(S) = \boldsymbol{SafeImpl}_{\lambda}(S) \cap \boldsymbol{SafeImpl}_{\gamma}(S)$.

5. Моделирование других семантик

В этом разделе мы покажем, каким образом некоторые известные конформности типа редукции сводятся к общей редукции в ***B*/*O***-семантике, описанной выше. В 1993 г. ван Глаббек опубликовал обобщающую и систематизирующую работу [11], в которой определены 30 типов наблюдений. Та или иная комбинация этих типов наблюдений соответствует той или иной семантике взаимодействия. Не все комбинации допустимы,

выделяется 155 возможных семантик и соответствующих конформностей. Мы рассмотрим только те наблюдения, которые соответствуют конформности типа редукции (то есть не симуляции), и покажем, как эти наблюдения и соответствующие семантики отображаются в нашей ***B***/***O***-модели и LTS-реализации.

Предполагается, что реализация может выполнять внешние, наблюдаемые действия из некоторого алфавита ***A***. Что касается ненаблюдаемого поведения реализации, то ван Глаббек исходит из основного допущения о τ-активности: τ-активность может быть между любыми двумя действиями (и до первого наблюдаемого действия) в реализации. В LTS-модели реализации переходы помечены символами из множества $\boldsymbol{A}\cup\{\tau\}$.

Клавиатура машины тестирования ван Глаббека состоит из переключателей – по одному на каждое внешнее действие. Переключатель имеет два положения: *free* и *blocked*. Реализация может выполнять внешнее действие *a* только в том случае, когда переключатель "*a*" находится в положении *free*; τ-действия всегда разрешены независимо от положения переключателей. Предполагается, что оператор машины может в любой момент времени устанавливать любые переключатели в любое положение и обладает достаточно быстрой реакцией, чтобы делать это достаточно быстро, то есть сразу после того или иного наблюдения. Внешние действия, которые совершает реализация, отображаются на экране дисплея. Машина ван Глаббека генеративная: реализация работает, выполняя разрешённые переключателями внешние действия и τ-действия до тех пор, пока такие действия у неё есть. Если в данный момент времени (в данном состоянии) реализации может выполнить несколько действий, то выбор выполняемого действия происходит недетерминированным образом – в зависимости от погодных условий.

В машине ван Глаббека тестовое воздействие – это изменение положения переключателей, которому соответствует множество переключателей в положении *free*, то есть подмножество $p\subseteq\boldsymbol{A}$. В ***B***/***O***-машине такому тестовому воздействию соответствует отдельная кнопка "*p*"[3], а каждое действие $a\in\boldsymbol{A}$ является наблюдением из ***O***. Иными словами, будем предполагать, что $\{\text{“}p\text{”}|p\subseteq\boldsymbol{A}\}\subseteq\boldsymbol{B}$ и $\boldsymbol{A}\subseteq\boldsymbol{O}$.

Если никаких других наблюдений, кроме внешних действий, нет, то такая семантика называется *trace semantics* [12]. Для моделирования этой семантики в ***B***/***O***-семантике нужно сделать следующее преобразование исходной LTS-реализации *S*. Для каждого состояния *s* и каждой кнопки "*p*", где $p\subseteq\boldsymbol{A}$, вводим новое состояние s_p. В этом состоянии проводим переход $s_p\text{—}a\rightarrow t_p$, где $a\in\boldsymbol{A}\cup\{\tau\}$, тогда и только тогда, когда $a\in p\cup\{\tau\}$ и был переход $s\text{—}a\rightarrow t$. Также проводим переход $s_p\text{—}q\rightarrow s_q$ для каждой кнопки "*q*", где $q\subseteq\boldsymbol{A}$ и $q\neq p$ (переходы-петли по кнопкам в ***B***/***O***-семантике можно не изображать), этот переход соответствует изменению положения переключателей. Новым начальным состоянием становится состояние $s_{0\varnothing}$, где s_0 начальное состояние исходной LTS *S* (предполагается, что сразу после включения машины переключатели находятся в положении *blocked*). Заметим, что если в состоянии *s* не было переходов по действиям из $p\cup\{\tau\}$, то в состоянии s_p не будет переходов по действиям из $\boldsymbol{A}\cup\{\tau\}$ (будут определены только переходы по кнопкам). В результате преобразования получится новая LTS S_T.

Заметим, что для *trace semantics* изменение положения переключателей излишне: достаточно было бы с самого начала установить все переключатели в положение *free* и не

[3] Мы обозначаем кнопку с использованием кавычек "*p*" для $p\subseteq A$ для того, чтобы отличить кнопку "*p*" от отказа *p*, который вводится ниже.

менять их. Это эквивалентно отсутствию переключателей в предположении, что все действия тем самым разрешены (так описана машина тестирования для *trace semantics* в [10]). Такой режим работы имеет ту же мощность тестирования, то есть будет получаться то же множество трасс, что и при всех возможных изменениях положения переключателей. Для моделирования в ***B***/***O***-семантике достаточно в LTS S_T оставить только состояния вида s_A, новым начальным состоянием является не состояние $s_{0\varnothing}$, а состояние s_{0A}. В результате преобразования получится LTS S_{T*}. Очевидно, LTS S_{T*} изоморфна исходной LTS *S*.

Преобразование LTS-реализации при таком режиме тестирования, когда кнопки не используются, не требуется.

Кроме этого, в машине ван Глаббека может быть *зелёная лампочка*, которая горит тогда, когда в реализации имеется какая-то активность (выполняется переход по внешнему действию или по τ-действию). Если зелёная лампочка гаснет, то это означает, что реализация остановилась: в ней нет τ-активности, а все внешние действия, которые она могла бы выполнять, заблокированы положением соответствующих переключателей. Это даёт новые наблюдения – *отказы* (*refusal set*). Отказ $p \subseteq \boldsymbol{A}$ определён в состоянии *s* тогда, когда в этом состоянии нет переходов по действиям из $p \cup \{\tau\}$. Трассы, в которые, кроме действий, входят отказы, называются *failure traces*, а соответствующая семантика взаимодействия – *failure trace semantics*, которая обозначается как *FT*. Для моделирования *FT*-семантики в ***B***/***O***-семантике достаточно в LTS-реализации S_T в каждом состоянии s_p, в котором не определены по действиям из $\boldsymbol{A} \cup \{\tau\}$, добавить переход-петлю по отказу *p*. В результате преобразования получается LTS S_{FT}.

Также в машине ван Глаббека могут быть, так называемые, *лампочки меню* – по одной для каждого действия $a \in \boldsymbol{A}$. Лампочка ‘*a*’ для действия *a* горит, если в реализации в текущий момент времени (в текущем состоянии) определён переход по действию *a* (независимо от того, разрешено это действие переключателями или нет). Понятно, что, если реализация выполняет какие-то действия (внешние или внутренние), то лампочки меню постоянно мигают. Поэтому они дают достоверную информацию только в том случае, когда реализация стоит, что определяется по зелёной лампочке. В этом случае мы получаем новое наблюдение – *множество готовности* (*ready set*). Множество готовности $r \subseteq \boldsymbol{A}$ определено в состоянии *s* тогда, когда это состояние стабильно (нет τ-переходов) и множество внешних действий, по которым в состоянии определены переходы, равно *r*. Трассы, содержащие действия и множества готовности, называются *ready traces*, а соответствующая семантика взаимодействия – *ready trace semantics*, которая обозначается как *RT*. Очевидно, что по множеству готовности *r* можно вычислить все отказы в этом состоянии: это все подмножества $\boldsymbol{A} \backslash r$. Поэтому в *RT*-семантике отказы не считаются отдельными наблюдениями и не рассматриваются смешанные трассы, содержащие и множества готовности и отказы. Для моделирования *RT*-семантики в ***B***/***O***-семантике достаточно в LTS-реализации S_T в каждом состоянии s_p, в котором не определены переходы по действиям из $\boldsymbol{A} \cup \{\tau\}$, добавить переход-петлю по множеству готовности для состояния *s* в исходной LTS *S*. Заметим, что множество готовности для состояния *s* в исходной LTS *S* равно объединению множеств готовности для всех состояний LTS S_T вида s_q, где *q* пробегает все кнопки. Иными словами, это множество всех внешних действий, которые LTS-реализация S_T, находящаяся в состоянии s_p, может выполнить после того или иного изменения положения переключателей машины ван Глаббека (после нажатия той или иной кнопки ***B***/***O***-машины). В результате преобразования получается LTS S_{RT}.

Ван Глаббек рассматривает также режим работы машины, когда переключатели нельзя переводить из положения *blocked* в положение *free*, кроме начальной установки переключателей при включении машины. В этом режиме после остановки реализации никакого продолжения её работы быть не может, поэтому наблюдение отказа или множества готовности может быть только в конце трассы. Обычно считается, что наблюдается пара: трасса действий и отказ (*failure pair*) или множество готовности (*ready pair*) в конце сеанса тестирования. Соответствующие семантики называются *failure semantics* (*F*) и *readiness semantics* (*R*). Для моделирования *F*- или *R*-семантики в нашей модели в LTS-реализации S_{FT} или S_{RT} переходы-петли по отказам или множествам готовности заменяются переходами в терминальное состояние. В результате преобразования получается LTS S_F или S_R.

Если только конечное число переключателей можно установить в положение *free*, то соответствующие семантики с отказами обозначаются FT^- и F^-. Для моделирования таких семантик в ***B***/***O***-семантике в LTS-реализации S_{FT} или S_F удаляются переходы по бесконечным кнопкам. В результате преобразования получается LTS S_{FT}- или S_F-.

При наличии лампочек меню рассматриваются семантики, когда, кроме того, что только конечное число переключателей можно установить в положение *free*, также только конечное число лампочек меню может быть активировано. В этом случае лампочки меню интерпретируются как лампочки-кнопки: чтобы активировать такую лампочку-кнопку, её нужно отжать; неактивированная лампочка ничего не показывает. Теперь, если реализация стоит, то на вопрос о том, есть ли в текущем её состоянии действие $a \in \boldsymbol{A}$, может быть три ответа: 1) есть – переключатель "*a*" в положении *blocked*, лампочка-кнопка '*a*' отжата и горит, 2) нет – переключатель "*a*" в положении *blocked*, лампочка-кнопка '*a*' отжата, но не горит, либо переключатель "*a*" в положении *free* (отжатая лампочка-кнопка '*a*' не может гореть), 3) не известно – переключатель "*a*" в положении *blocked*, лампочка-кнопка '*a*' не отжата. Поэтому полная информация о действиях в текущем состоянии описывается не одним множеством готовности *r*, а парой множеств: множеством r^+ действий с ответом «есть» и множеством r^- действий с ответом «нет». В отличие от случая, когда все лампочки меню всегда активированы, эти два множества r^+ и r^- в объединении могут не давать множество $\boldsymbol{A}$ всех действий. Тестовое воздействие теперь – это не только установление тех или иных переключателей в положение *free*, но и активизация тех или иных лампочек меню. Соответствующие семантики с множествами готовности обозначаются RT^- и R^-. Для моделирования таких семантик в ***B***/***O***-семантике в LTS-реализации S_{RT} или S_R вместо состояния s_p создаётся множество состояний вида s_{px}, где x – конечное множество активированных лампочек меню. Вместо перехода по действию $s_p \text{—} a \rightarrow t_p$ проводятся переходы вида $s_{px} \text{—} a \rightarrow t_{px}$. Вместо перехода по кнопке $s_p \text{—} q \rightarrow s_q$ проводятся переходы вида $s_{px} \text{—} qy \rightarrow s_{qy}$, где y – конечное множество активированных лампочек меню, и только для конечной кнопки q. Вместо перехода-петли по множеству готовности $s_p \text{—} r \rightarrow s_p$ проводятся переходы-петли по парам конечных множеств $s_{px} \text{—} r^+r^- \rightarrow s_{px}$; очевидно, что $x \setminus r^- = r^+$ и $p \subseteq r^-$. В результате преобразования получается LTS S_{RT}- или S_R-.

Ван Глаббек рассматривает также два специальных наблюдения: 0 и *S*. Наблюдение 0 возникает, когда гаснет зелёная лампочка, что означает остановку реализации. Наблюдение *S* возникает, когда реализация переходит в стабильное состояние. В *FT*- или *F*-семантике наблюдение 0 возникает всякий раз, когда возникает какой-нибудь отказ, а

наблюдение *S* возникает всякий раз, когда возникает отказ ∅. Поэтому в этих семантиках оба этих наблюдения излишни.

Наблюдение 0 полезно, когда переключателей нет, что интерпретируется как фиксация всех переключателей в положении *free* (также как для *trace semantics*). В этом случае наблюдение 0 означает переход реализации в терминальное состояние, что соответствует отказу ***A*** в *FT*- или *F*-семантике. Понятно, что после наблюдения *0* не может быть никаких наблюдений, поэтому наблюдение *0* может только заканчивать трассу. Такие трассы называются *completed traces*. Для моделирования в ***B***/***O***-семантике достаточно (как для *trace semantics* при отсутствии переключателей) в LTS-реализации S_{T*} в каждом терминальном состоянии добавить переход-петлю по наблюдению *0*. В результате преобразования получается LTS S_{T0}.

Наблюдение *S* полезно, когда вместо переключателей для каждого внешнего действия имеется один переключатель, который разрешает или блокирует сразу все внешние действия. Если все действия запрещены, а зелёная лампочка гаснет, это означает, что реализация оказалась в стабильном состоянии, что и отображается наблюдением *S*. Оно эквивалентно наблюдению отказа ∅ в *FT*- или *F*-семантике. Если все действия разрешены, а зелёная лампочка гаснет, это означает, что реализация оказалась не просто в стабильном, а в терминальном состоянии, то есть имеет место наблюдение 0, «поглощающее» наблюдение *S*. В трассы, кроме действий, может входить наблюдение *S*, а заканчиваться такие трассы могут наблюдением *0*. Для моделирования в ***B***/***O***-семантике достаточно в LTS-реализации S_T оставить только состояния вида s_A и $s_\varnothing$, после чего добавить переходы-петли по наблюдению 0 во всех состояниях вида s_A, где нет переходов по действиям из $\boldsymbol{A}\cup\{\tau\}$, и переходы-петли по наблюдению *S* во всех стабильных состояниях вида $s_\varnothing$. В результате преобразования получается LTS S_{T0S}.

Если не считать ограничения по конечности в FT^-, F^-, RT^- и R^--семантиках и рассмотренные выше два специальных случая с наблюдениями *0* и *S*, ван Глаббек не рассматривает конформности, которые получаются при различных ограничениях на то, какие переключатели могут находиться в положении *free*, то есть на множества разрешаемых внешних действий. Также считается, что если есть зелёная лампочка, то никаких ограничений на её работу нет: она работает всегда, независимо от положения переключателей. Иными словами, либо наблюдаются все возможные отказы, либо никакие отказы не наблюдаются. В то же время многие конформности как раз основаны на такого рода ограничениях.

Одной из таких конформностей, не попавшей в классификацию ван Глаббека, является популярное сегодня отношение ***ioco*** (***i****nput-****o****utput* ***co****nformance*), предложенное Яном Тритмансом в 1996 г. [14,15]. Предполагается, что алфавит внешних действий ***A*** разбит на два непересекающихся подмножества стимулов (*input*) *X* и реакций (*output*) *Y*. Разрешен может быть либо один стимул, либо все реакции. Говорят, что оператор может либо посылать в реализацию один стимул, либо ждать от неё любой реакции. При этом зелёная лампочка работает только при ожидании реакций, тем самым, имеется только один отказ, означающий отсутствие реакций, называемый *quiescence* (молчание) и обозначаемый символом $\delta=Y$. Кроме того, ***ioco***-семантика требует не генеративной, а реактивной машины. Вместо переключателей имеются кнопки, которые автоматически отжимаются после выполнения реализацией разрешённого внешнего действия. Для получения следующего внешнего действия нужно ещё раз нажать другие или те же самые кнопки. Впрочем, различие между генеративной и реактивной машинами на самом деле несущественно, как показал ван Глаббек в той же работе [11].

Обобщением такого подхода является предложенная нами ***R***/***Q***-семантика [1,5,6]. Она задаётся двумя непересекающимися семействами множеств действий $\boldsymbol{R}\subseteq 2^{\boldsymbol{A}}$ и $\boldsymbol{Q}\subseteq 2^{\boldsymbol{A}}$, покрывающими весь алфавит: $(\cup\boldsymbol{R})\cup(\cup\boldsymbol{Q})=\boldsymbol{A}$. ***R***/***Q***-машина тестирования реактивная. Каждому множеству $p\in\boldsymbol{R}\cup\boldsymbol{Q}$ соответствует кнопка "*p*", разрешающая все действия из *p*. Зелёная лампочка работает только, если $p\in\boldsymbol{R}$, иными словами, наблюдаются только отказы из ***R***. Заметим, что в ***R***/***Q***-семантике допускаются также переходы по разрушению γ в реализации.

Отношение ***ioco*** – это частный случай ***R***/***Q***-семантики, когда $\boldsymbol{R}=\{\{x\}|x\in X\}$ – семейство всех множеств, каждое из которых состоит из одного стимула, и $\boldsymbol{Q}=\{Y\}$ – семейство, состоящее из одного множества всех реакций. Кроме того, для ***ioco*** предполагается, что в реализации нет разрушения.

Для моделирования ***R***/***Q***-семантики в ***B***/***O***-семантике нужно преобразовать исходную LTS-реализацию *S* следующим образом. Для каждой кнопки "*p*", где $p\in\boldsymbol{R}\cup\boldsymbol{Q}$, и каждого состояния *s* добавляем новое состояние s_p и новый переход s—"p"→s_p. В каждом новом состоянии s_p проводим переход s_p—a→t, если $a\in p$ и имеется переход s—a→t. Также проводим переход s_p—a→t_p, если $a\in\{\tau,\gamma\}$ и имеется переход s—a→t. Если $p\in\boldsymbol{R}$ и в состоянии s_p нет переходов по действиям из $\boldsymbol{A}\cup\{\tau\}$, то проводим переход s_p—p→s. Такой переход, очевидно, проводится тогда и только тогда, когда в состоянии *s* был ***R***-отказ *p*. После этого удаляем все переходы по внешним действиям из старых состояний, оставляя τ- и γ-переходы.

У нас получится LTS $S_{\boldsymbol{R}/\boldsymbol{Q}}$, состояния которой делятся на «старые» и «новые», это нужно для моделирования «реактивности» ***R***/***Q***-машины на генеративной ***B***/***O***-машине. Переход по кнопке "*p*" ведёт из старого состояния в новое: из *s* в s_p. Переход по внешнему действию *a* – из нового состояния в старое: из s_p в *t*, причем только по такому действию *a*, которые разрешаются кнопкой "*p*", то есть $a\in p$. Переход по отказу $p\in\boldsymbol{R}$ ведёт из состояния s_p в состояние *s*. А τ- и γ-переходы ведут как из старых состояний в старые (из *s* в *t*), так и из соответствующих новых в соответствующие новые (из s_p в t_p).

Более подробно о моделировании ***R***/***Q***-семантики в ***B***/***O***-семантике можно прочитать в нашей работе [8].

Итак, мы видим, что все рассмотренные выше семантики моделируются в ***B***/***O***-семантике с помощью соответствующего преобразования исходной LTS-реализации. Тем самым, в ***B***/***O***-семантике исходная конформность рассматривается не на классе всех LTS-реализаций, допускаемых ***B***/***O***-семантикой, а на подклассе преобразованных LTS-реализаций. Из-за этого и возникают множественное следование ошибок и эквивалентные множества ошибок, не совпадающие со спецификацией, то есть возникают различные эквивалентные спецификации. Каждая такая спецификация получается из некоторого полного набора тестов, если взять все ***fail***-трассы тестов набора как ошибки. Или, иными словами, существуют полные наборы тестов, различие между которыми не устраняется тривиальной оптимизацией, аналогичной нормализации спецификаций. В этом и состоит проблема оптимизации тестов для различных семантик, которая оказывается частным случаем проблемы оптимизации тестов на том или ином классе реализаций.

6. Приоритеты

Как замечает сам ван Глаббек [11] в его машине тестирования предполагается отсутствие приоритетов между действиями: реализация может выполнять действие a, если переключатель "a" находится в положении *free*, независимо от положения остальных переключателей, то есть от того, какие ещё действия разрешены. Правило недетерминированного выбора предписывает реализации выбирать на выполнение любое действие, которое в ней определено и которое разрешено положением переключателей. При этом τ-активность всегда может выполняться независимо от положения переключателей[4]. В то же время для реальных программных и аппаратных систем это правило не всегда адекватно отражает требуемое поведение системы. Ниже мы рассмотрим несколько примеров таких систем.

Для того чтобы ввести приоритеты в машину ван Глаббека, нужно пометить в LTS-реализации каждый переход не только действием a, но и множеством разрешённых действий p, то есть парой (a,p). При этом предполагается, что $a \in p \cup \{\tau\}$. Что происходит при изменении положения переключателей, когда множество разрешённых действий меняется с p на q? Мы предполагаем, что влияние этого изменения на поведение реализации происходит в два этапа. На первом этапе блокируется выполнение переходов по действиям, но по-прежнему остаются разрешёнными τ-переходы, помеченные «старым» множеством p, то есть переходы, помеченные парой (τ,p). В то же время предполагается, что на этом этапе реализация может выполнить только конечное число таких переходов. После этого, на втором этапе, разрешаются только переходы помеченные «новым» множеством q, то есть переходы, помеченные парой (a,q), где $a \in q \cup \{\tau\}$.

Поясним смысл первого этапа. При отсутствии приоритетов по основному допущению о τ-активности τ-переходы всегда разрешены независимо от положения переключателей. Поэтому при изменении положения переключателей нет различия между τ-переходами, которые выполняются непосредственно перед этим изменением и непосредственно после него. Однако при наличии приоритетов такое различие возникает. Если бы первого этапа не было, то изменение множества разрешённых действий с p на q сразу же запрещало бы (τ,p)-переходы. После прохождения реализацией некоторого маршрута M с трассой σ реализация может оказаться в состоянии s, в котором определена цепочка (τ,p)-переходов. Для полноты тестирования необходимо, чтобы хотя бы в одном сеансе тестирования эта цепочка переходов могла быть пройдена. Но для того, чтобы реализация могла пройти эту цепочку, оператор должен изменить положение переключателей только после выполнения этой цепочки. Для этого оператор должен выждать некоторый интервал времени, зависящий от длины цепочки и времен ожидания в состояниях этой цепочки. Поскольку реализация оператору неизвестна (она скрыта в чёрном ящике), значение этого интервала времени оператору неизвестно. Поэтому мы должны были бы потребовать, чтобы оператор в различных сеансах тестирования делал все возможные задержки по времени перед очередным изменением положения переключателей. Однако мы предъявляем к оператору единственное требование: хотя бы в одном сеансе тестирования эта задержка достаточно мала, то есть меньше времени ожидания реализацией в состоянии s. Наличие первого этапа гарантирует возможность прохождения всех цепочек (τ,p)-переходов после маршрута M с трассой σ без дополнительных требований к оператору. Иными словами, гарантируется прохождение каждого маршрута с трассой σ до того, как изменится множество разрешённых действий с p на q и начнут выполняться только (τ,q)-переходы.

[4] В **R**/**Q**-семантике это относится также к γ-переходам.

При наличии приоритетов меняются понятия отказа и дивергенции. Отказ p возникает при разрешённом множестве действий p в том случае, когда в текущем состоянии нет переходов, помеченных парой вида (a,p), где $a \in p \cup \{\tau\}$. Дивергенция при разрешённом множестве действий p возникает, когда в текущем состоянии начинается бесконечный τ-маршрут с бесконечным постфиксом (τ,p)-переходов. Соответственно, можно говорить о p-дивергенции и о p-дивергентных и p-конвергентных состояниях.

Как было сказано выше, в ***B*/*O***-семантике имеется приоритет тестового воздействия над поведением реализации, как наблюдаемым, так и ненаблюдаемым. Это даёт возможность представить любую семантику машины ван Глаббека с приоритетами как частный случай ***B*/*O***-семантики для конформностей типа редукции. Для такого моделирования нужно выполнить модифицированное преобразование исходной LTS-реализации S с приоритетами в соответствующую LTS S_i, где i обозначает семантику, то есть i – это T, T^*, FT, RT, F, R, FT^-, RT^-, F^-, R^-, $T0$ или $T0S$. Модификация заключается в следующем: переход $s_p — a \to t_p$, где $a \in \boldsymbol{A} \cup \{\tau\}$, определяется тогда и только тогда, когда $a \in p \cup \{\tau\}$ и в S был не переход $s — a \to t$, как раньше, а переход $s — (a,p) \to t$. Далее переходы по кнопкам, отказам, множествам готовности (или парам множеств) и наблюдениям 0 и S проводятся как обычно.

Для ***R*/*Q***-семантики с приоритетами LTS-реализация задаётся аналогичным образом: каждый переход помечается парой (a,p), где $a \in p \cup \{\tau,\gamma\}$ и $p \subseteq \boldsymbol{A}$. Разумеется, для данных ***R*** и ***Q*** при тестировании через ***R*/*Q***-машину выполняться могут только такие (a,p)-переходы, где $p \in \boldsymbol{R} \cup \boldsymbol{Q}$. Для моделирования ***R*/*Q***-семантики с приоритетами в ***B*/*O***-семантике нужно выполнить модифицированное преобразование исходной LTS-реализации S с приоритетами в LTS $S_{\boldsymbol{R/Q}}$. Модификация заключается в следующем: переходы $s_p — a \to t$, где $a \in p$, $s_p — a \to t_p$, где $a \in \{\tau,\gamma\}$, определяются, когда в S был не переход $s — a \to t$, как раньше, а переход $s — (a,p) \to t$. Переход по отказам и кнопкам проводятся как обычно. Заметим, что переход по отказу $s_p — p \to s$, где $p \in \boldsymbol{R}$, проводится тогда, когда в s был ***R***-отказ p, то есть когда в состоянии s не было переходов, помеченных парами (a,p), где $a \in p \cup \{\tau,\gamma\}$. Наконец, в старых состояниях оставляются не все τ- и γ-переходы, а только помеченные пустым множеством, то есть парой $(\tau,\varnothing)$ или $(\gamma,\varnothing)$.

Отметим одну особенность такого моделирования ***R*/*Q***-семантики с приоритетами средствами ***B*/*O***-семантики. В ***R*/*Q***-семантике, содержащей пустую кнопку ($\varnothing \in \boldsymbol{R} \cup \boldsymbol{Q}$), не различаются τ- и γ-переходы при нажатии пустой кнопки и при отсутствии нажатой кнопки: в обоих случаях множество разрешённых внешних действий одно и то же – пустое множество. Из-за этого невозможно потребовать, чтобы такой переход срабатывал только в том случае, когда никакая кнопка не нажата, или, наоборот, чтобы он срабатывал при нажатии пустой кнопки, но не мог выполняться, если никакая кнопка не нажата. При моделировании в ***B*/*O***-семантике τ-переходы $s_\varnothing — \tau \to t_\varnothing$ и $s — \tau \to t$ возникают всегда одновременно (если был переход $s — (\tau,\varnothing) \to t$). Но после такого моделирования в ***B*/*O***-семантике мы можем разрешить эту проблему, оставив только один из этих двух τ-переходов.

Подробнее ***R*/*Q***-семантика с приоритетами описана в наших работах [3,4], а её моделирование в ***B*/*O***-семантике – в нашей работе [8].

Поскольку все рассмотренные выше семантики с приоритетами моделируются в ***B***/***O***-семантике с помощью соответствующего преобразования исходной LTS-реализации, в ***B***/***O***-семантике исходная конформность рассматривается не на классе всех LTS-реализаций, допускаемых ***B***/***O***-семантикой, а на подклассе преобразованных LTS-реализаций. Из-за этого, как и в случае семантик без приоритетов, возникают множественное следование ошибок и эквивалентные множества ошибок, не совпадающие со спецификацией, то есть возникают различные эквивалентные спецификации.

В LTS с приоритетами может оказаться много кратных переходов, помеченных парами (a,p_i) с одним и тем же действием a и разными множествами разрешённых действий $p_1,p_2,\ldots$ Для того, чтобы изобразить такие переходы более компактным образом, введем булевские переменные – по одной на каждое действие $a\in\boldsymbol{A}$; переменная ‘*a’* принимает значение *true*, если в машине ван Глаббека переключатель “*a*” находится в положении *free* или в ***R***/***Q***-машине нажата кнопка “*p*” и $a\in p$**.** Множество разрешённых действий p можно задавать элементарной конъюнкцией ‘*p*’ этих переменных, в которую каждая переменная a входит ровно один раз: без отрицания, если $a\in p$, или с отрицанием в противном случае. Конъюнкция ‘*p*’ принимает значение *true* тогда и только тогда, когда переключатели машины ван Глаббека разрешают множество действий p или в ***R***/***Q***-машине нажата кнопка “*p*”. После этого множество кратных переходов можно изобразить одним переходом, помеченным парой (a,π), где π предикат, эквивалентный СДНФ ‘p_1’$\vee$‘p_2’$\vee\ldots$

Рассмотрим несколько характерных примеров использования приоритетов.

Выход из дивергенции. Запрос, поступающий извне, может бесконечно долго игнорироваться системой, если он имеет тот же приоритет, что бесконечная внутренняя активность, то есть дивергенция. Заметим, что внутренняя активность может быть инициирована предыдущим запросом. Если речь идёт о составной системе, собранной из нескольких компонентов, то дивергенция может быть естественным результатом взаимодействия компонентов между собой. И в этом случае для обработки запроса, поступающего в систему (в один из её компонентов) извне, он должен иметь больший приоритет, чем внутреннее взаимодействие.

В ***B***/***O***-семантике такой запрос можно понимать как (тестовое) воздействие, то есть нажатие кнопки[5]. Переход по кнопке выполняется гарантированно через конечное время, то есть только после конечной τ-активности. Тем самым, может осуществиться выход из дивергенции, если, конечно, переход по кнопке ведёт в конвергентное состояние.

В машине ван Глаббека и в ***R***/***Q***-машине запросам соответствуют некоторые внешние действия из алфавита ***A***. Тогда τ-переход из состояния s помечается только таким множеством p разрешённых действий, которое не содержит запросов, переходы по которым определены в s**.** Если реализация находится в состоянии s, а множество действий, переключатели которых находятся в положении *free*, равно p, то дивергенция может возникнуть только в том случае, когда состояние s p-дивергентно.

Выход из осцилляции (приоритет приёма над выдачей). Под осцилляцией понимается бесконечная цепочка выдачи сообщений системой. Для того чтобы такую цепочку можно было прервать, заставив систему обрабатывать поступающий извне запрос, последний

[5] Если мы можем наблюдать приём запроса реализацией, то такому запросу, кроме кнопки, соответствует также отдельное наблюдение (см. п.8.1).

должен иметь больший приоритет, чем выдача сообщений. Обычно также подразумевается, что внутренняя активность менее приоритетна, чем приём запроса.

В ***B***/***O***-семантике естественно считать выдачу сообщения наблюдением, а запрос – (тестовым) воздействием (нажатием кнопки). Поскольку нажатие кнопки блокирует наблюдения до перехода по кнопке, выход из осцилляции будет выполнен, если такой переход по кнопке ведёт в состояние, где нет бесконечной цепочки выдачи сообщений и нет дивергенции.

В машине ван Глаббека и в ***R***/***Q***-машине запросам и выдаче сообщений соответствуют два непересекающихся подмножества алфавита ***A***. Тогда переходы по выдаче сообщений и τ-переходы из состояния *s* помечаются только теми множествами *p* разрешённых действий, которые не содержат запросов, переходы по которым определены в *s*.

Для краткости мы рассмотрим остальные примеры только для машины ван Глаббека.

Приоритет выдачи над приёмом в неограниченных очередях. Этот обратный пример характерен для неограниченной очереди, используемой в качестве буфера между взаимодействующими системами, в частности, при тестировании в контексте [13]. Здесь нужно, чтобы выборка из очереди была приоритетней постановки в очередь. В противном случае очередь имеет право только принимать сообщения и никогда их не выдавать. При тестировании в контексте для входной очереди это означает, что все входные сообщения, посылаемые тестом, не доходят до реализации, бесконечно накапливаясь в очереди. Соответственно, для выходной очереди это означает, что тест может не получать никаких ответных сообщений от реализации, хотя она их выдаёт, поскольку они «оседают» в очереди.

Пусть элементы очереди принадлежат алфавиту *Z*. Постановке элемента $x \in Z$ в очередь соответствует действие $!x \in \boldsymbol{A}$ и переключатель "*!x*". Для детерминированности постановки элементов в очередь в положение *free* разрешается устанавливать не более одного из этих переключателей. Выборке элемента $y \in Z$ из очереди соответствует действие $?y \in \boldsymbol{A}$ и переключатель "*?y*". Приоритет выборки из очереди над постановкой в очередь означает, что переход по постановке в очередь, то есть по действию *!x*, может выполняться только в том случае, когда переключатель "*!x*" находится в положении *free*, а выбирать из очереди либо нечего, то есть очередь пуста, либо запрещено, то есть переключатель "*?y*", где *y* первый элемент очереди, находится в положении *blocked*. Состояние реализации *s* – это состояние очереди, то есть $s \in Z^*$. Переход $s \text{—}(!x,p)\rightarrow s \cdot !x$ определяется тогда и только тогда, когда $p \subseteq \boldsymbol{A}$, $!x \in p$, а очередь *s* либо пуста, либо её первый элемент $y=s(1)$ не может быть выбран из очереди, поскольку переключатель "*?y*" находится в положении *blocked*, то есть $?y \notin p$. Переход $?y \cdot s \text{—}(?y,p)\rightarrow s$ определяется, как обычно, для каждого $p \subseteq \boldsymbol{A}$, если $?y \in p$.

Прерывание цепочки действий. Команда «отменить» (*cancel*) должна останавливать выполнение последовательности действий, инициированной предыдущим запросом, и вызывать цепочку завершающих действий. При отсутствии приоритетов такая команда, даже если она выдана сразу после выдачи запроса, имеет право быть выполнена только после того, как вся обработка закончится, то есть, фактически, ничего «не отменяет».

Команду *cancel* будем рассматривать как одно из внешних действий. Если в состоянии *s* определён переход по *cancel*, то все остальные переходы из состояния *s* помечаются

только теми множествами *p* разрешённых действий, которые не содержат *cancel*. Переход по *cancel*, а также все переходы в других состояниях помечаются всеми допустимыми множествами разрешённых действий. Если в состоянии *s* определён переход по *cancel*, и переключатель "*cancel*" находится в положении *free*, то выполняться будет только переход по *cancel*.

Приоритетная обработка запросов. 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.

Множество запросов разбивается на непересекающиеся подмножества $X_1, X_2, \ldots$ алфавита $\boldsymbol{A}$ так, что запросы из подмножества с большим индексом имеют больший приоритет. Переход из состояния *s* по запросу $x \in X_i$ помечается таким множеством $p \subseteq \boldsymbol{A}$, которое не содержит ни одного такого запроса $y \in X_j$, что $j > i$ и в состоянии *s* есть переход по *y*.

7. Оптимизация тестов для различных классов реализаций

В предыдущих разделах мы показали, что для ***B***/***O***-семантики и общей редукции на классе всех возможных реализаций имеются только тривиальные зависимости между ошибками, которые легко устраняются процедурой нормализации спецификации. Также мы рассмотрели две гипотезы о безопасности: λ- и γ-гипотезу, которые сужают класс тестируемых реализаций. При этом λ-гипотеза создаёт дополнительную зависимость между ошибками, которая, однако, легко устраняется дополнительной λ-нормализацией, а γ-гипотеза не создаёт дополнительной зависимости между ошибками и дополнительной нормализации не требуется. Далее мы рассмотрели примеры семантик и конформностей, которые сводятся к ***B***/***O***-семантике и общей редукции, но рассматриваются на суженных классах реализаций. Из-за такого сужения возникают уже нетривиальные зависимости между ошибками, что требует нетривиальной оптимизации тестов [6,7].

Всё это можно рассматривать как частный случай общей проблемы сужения класса тестируемых реализаций, которое создаёт зависимости между ошибками и даёт возможность оптимизации тестов. Рассмотрим несколько примеров такого сужения класса тестируемых реализаций, не связанных напрямую с выбором той или иной семантики или гипотезы о безопасности. В этих примерах мы покажем, что такое сужение позволяет использовать конечные полные наборы тестов. Во всех этих примерах предполагается конечность ***B***/***O***-семантики и LTS-спецификации *S*. Будем считать, что суммарное число кнопок и наблюдений не превосходит числа *m*, а число состояний детерминированной LTS-спецификации не превосходит числа *k*.

Первый пример – класс LTS-реализаций с ограниченным числом состояний. Если число состояний реализации не превосходит *n*, то для полноты тестирования достаточно ограничиться тестами длиной не более *nk*. Тогда этот набор тестов содержит не более $O(m^{nk})$ тестов.

Для доказательства этого утверждения достаточно построить композицию LTS реализации и спецификации по следующим правилам. Состояниями композиционной LTS являются пары состояний реализации и спецификации, начальное состояние – пара начальных состояний. Переход $(s,t) \text{—} a \rightarrow (s`,t`)$ определяется тогда и только тогда, когда в

реализации есть переход $s—a→s`$, а в спецификации есть переход $t—a→t`$. Реализация неконформна тогда и только тогда, когда в ней есть ошибка 1-го рода, то есть трасса спецификации. Такая трасса в детерминированной спецификации заканчивается в одном состоянии, которое объявлено конечным. В реализации есть такая трасса тогда и только тогда, когда в композиционной LTS из начального состояния достижимо состояние вида (s,t), где t – конечное состояние спецификации. Такое состояние достижимо по простому маршруту (проходящему через каждое состояние не более одного раза), длина которого не превосходит числа достижимых состояний композиционной LTS, которое, в свою очередь, не превосходит общего числа состояний, равного nk. Таким образом, реализация неконформна тогда и только тогда, когда в ней есть ошибочная трасса длиной не более nk. Иными словами, набор всех примитивных тестов длиной не более nk, является полным. Число таких последовательностей в алфавите с m символами, очевидно, равно $O(m^{nk})$.

Второй пример – конечный (с точностью до изоморфизма) класс тестируемых реализаций. Для конечной семантики класс LTS-реализаций с ограниченным числом состояний, очевидно, конечен с точностью до изоморфизма. Поэтому первый пример является частным случаем второго примера. Если семантика и спецификация конечны, то для любого конечного класса реализаций существует конечный полный набор тестов. Для доказательства достаточно заметить, что любой конечный класс реализаций $\boldsymbol{I}$ является подклассом класса реализаций, число состояний которых ограничено числом n, где n – максимальное число состояний реализаций из класса $\boldsymbol{I}$.

Третий пример – конечный подкласс неконформных реализаций класса тестируемых реализаций. В работах [16,17] такой подкласс называется *классом неисправностей*. Для конечности полноты тестового набора достаточно не конечности класса $\boldsymbol{I}$ тестируемых реализаций, а его подкласса $\boldsymbol{I}\backslash\boldsymbol{C}_S$ неисправностей. Действительно, в каждой неконформной реализации из $I\in\boldsymbol{I}\backslash\boldsymbol{C}_S$ имеется некоторая ошибка 1-го рода, выберем одну такую ошибку σ_I. Набор ошибок $S_{\boldsymbol{I}}=\{\sigma_I|I\in\boldsymbol{I}\backslash\boldsymbol{C}_S\}$ конечен и, очевидно, является полным тестом, а набор $\{\{\sigma_I\}|I\in\boldsymbol{I}\backslash\boldsymbol{C}_S\}$ примитивных тестов, построенный по этим ошибкам, является полным набором тестов для класса $\boldsymbol{I}$.

Таким образом, фактически, при тестировании мы пытаемся найти не все ошибки 1-го рода, определяемые спецификацией S, а их конечное подмножество $S_{\boldsymbol{I}}\subseteq S$. Это эквивалентно тому, что вместо спецификации S мы используем спецификацию $S_{\boldsymbol{I}}$. Иными словами, на классе реализаций $\boldsymbol{I}$ спецификации S и $S_{\boldsymbol{I}}$ эквивалентны. Правда, выполняя тестирование по спецификации S, мы можем быстрее найти ошибку, чем при тестировании по спецификации $S_{\boldsymbol{I}}$. Это объясняется тем, что неконформная реализация $I\in\boldsymbol{I}\backslash\boldsymbol{C}_S$ может содержать не только ошибку σ_I, но и какие-то ошибки, не вошедшие в конечный набор $S_{\boldsymbol{I}}$. Например, спецификация S может определять как ошибочные некоторые наблюдения с самого начала (до нажатия кнопок): $a,b_1,b_2,b_3,\ldots$, а $S_{\boldsymbol{I}}$ содержит только одну такую ошибку a. При тестировании мы можем с самого начала ждать наблюдений и, опираясь на спецификацию S, выносим вердикт ***fail***, если получаем любую ошибку $a,b_1,b_2,b_3,\ldots$, но, опираясь на спецификацию $S_{\boldsymbol{I}}$, вердикт ***fail*** выносится только для ошибки a.

Эти рассуждения дают четвёртый пример – конечное подмножество ошибок $S_{\boldsymbol{I}}\subseteq S$ такое, что каждая неконформная (то есть содержащая хотя бы одну ошибку из S) реализация из класса $\boldsymbol{I}$ содержит хотя бы одну ошибку из $S_{\boldsymbol{I}}$. Вместо конечного класса неисправностей достаточно использовать просто конечный поднабор набора ошибок, определяемого спецификацией.

Далее напомним, что класс реализаций ***I*** определяет ошибки 2-го рода: трассы, которые не встречаются в конформных реализациях класса ***I***, но встречаются в некоторых его неконформных реализациях. Такая ошибка 2-го рода σ может не быть ошибкой 1-го рода, то есть $\sigma \notin S$. Поэтому четвёртый пример является частным случаем последнего, пятого примера, когда для класса тестируемых реализаций ***I*** задан конечный набор ошибок (1-го и 2-го родов) S_I такой, что каждая неконформная (то есть содержащая хотя бы одну ошибку из S) реализация из класса ***I*** содержит хотя бы одну ошибку из S_I. Этот пример последний, поскольку его условие просто эквивалентно условию существования конечного полного набора тестов. Если такой набор тестов существует, то множество трасс всех тестов набора – это и есть множество ошибок S_I.

8. Обоснование выбранной модели взаимодействия

В этом заключительном разделе статьи мы дадим обоснование выбранной нами модели взаимодействия. Мы рассмотрим шесть вопросов, которые возникают в связи с этой моделью: 1) когда кнопка попадает в трассу: при её нажатии и/или при выполнении LTS-реализацией перехода по кнопке; 2) почему нажатие кнопки блокирует наблюдения; 3) почему оператор должен уметь нажать кнопку достаточно быстро после получения трассы; 4) почему нажатие кнопки не блокирует τ-активность; 5) почему нажатие кнопки блокирует дивергенцию, то есть разрешает только конечную τ-активность; 6) почему переход по каждой кнопке определён в каждом состоянии реализации?

8.1. Когда кнопка попадает в трассу?

Нажатие кнопки выполняется оператором машины тестирования, и в этот момент времени он знает, какая трасса уже получена. Поэтому ничто не мешает оператору отметить тот факт, что он нажал данную кнопку после наблюдения данной трассы. С другой стороны, поведение реализации, вообще говоря, зависит от той трассы, после которой оператор нажимает кнопку. Поэтому в любом случае при нажатии кнопки она попадает в трассу.

Если при выполнении LTS-реализацией перехода по кнопке p на экране дисплея также появляется кнопка p, то это аналогично тому, как при выполнении реализацией перехода по наблюдению на экране дисплея появляется это наблюдение. Это означает, что выполнение перехода по кнопке p есть, фактически, наблюдение, появление которого в трассе обозначим p`, чтобы отличить от p, означающего нажатие кнопки p.

Такое наблюдение, в принципе, ничем не отличается от других наблюдений, поэтому режим работы с наблюдаемым переходом по кнопке можно считать частным случаем общей модели взаимодействия. Чтобы в этой модели изобразить такой режим работы, достаточно в LTS-реализации каждый переход по кнопке $s—p→t$ заменить на два перехода с введением дополнительного промежуточного состояния: $s—p→s`—p`→t$.

8.2. Почему нажатие кнопки блокирует наблюдения?

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

Тем самым, будет наблюдаться трасса *pu*. Следовательно, если при взаимодействии с реализацией может наблюдаться трасса *up*, то может наблюдаться и трасса *pu*.

В выбранной нами модели взаимодействия такой дополнительной зависимости между трассами нет. В то же время режим работы с отсутствием блокировки наблюдений при нажатии кнопки легко моделируется в нашей модели. Для этого достаточно систематически выполнить следующее преобразование реализации, пока оно возможно: если в реализации имеются переходы $s \text{—} p \rightarrow s_p$, $s \text{—} u \rightarrow t$ и $t \text{—} p \rightarrow t_p$, то добавим переход $s_p \text{—} u \rightarrow t_p$. Тем самым, если в состоянии s начиналась трасса *up*, заканчивающаяся в состоянии t_p, то теперь будет и трасса *pu*, заканчивающаяся в том же состоянии.

Таким образом, модель взаимодействия с блокировкой наблюдений при нажатии кнопки является более общей. Классу всех реализаций для модели без блокировки соответствует подкласс реализаций, получаемых описанной выше процедурой, для модели с блокировкой. Как и в общем случае, такое сужение класса реализаций приводит к появлению зависимостей между трассами реализации (в частности, между ошибками).

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

8.3. Зачем оператору нужно быстро нажимать кнопки?

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

Также мы хотим, чтобы любой достижимый переход в LTS-реализации мог быть выполнен при том или ином взаимодействии с ней (в зависимости от поведения оператора и погодных условий, моделирующих недетерминированное поведение реализации). Если это не так, и какой-то переход не выполняется ни при каком взаимодействии, то это эквивалентно отсутствию этого перехода в реализации. А это, в свою очередь, приводит к сужению класса рассматриваемых реализаций, что также чревато появлением дополнительных зависимостей между ошибками. Только гипотезы о безопасности запрещают выполнение тех или иных «опасных» переходов в реализации, но, как мы рассмотрели выше, это также приводит к сужению класса реализаций и, возможно, появлению дополнительных зависимостей между ошибками.

Почему мы требуем, чтобы оператор мог нажимать кнопку достаточно быстро после получения трассы, хотя и не обязан это делать всегда? Если оператор не может нажать кнопку достаточно быстро после трассы, то реализация может успеть выполнить после этой трассы один или несколько τ-переходов. Тем самым, переход по кнопке, начинающийся в состоянии до этих τ-переходов, никогда не будет выполнен.

8.4. Почему нажатие кнопки не блокирует τ-активность?

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

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

8.5. Почему нажатие кнопки блокирует дивергенцию?

Хотя нажатие кнопки не блокирует τ-активность, но разрешает только конечную τ-активность, то есть разрешает выполнять только конечное число τ-переходов. Тем самым, нажатие кнопки блокирует дивергенцию. Это необходимо для того, чтобы реализовать «выход из дивергенции», то есть приоритет тестового воздействия над дивергенцией.

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

До сих пор мы предполагали, что переход по кнопке определён в каждом состоянии LTS-реализации (по умолчанию отсутствие такого перехода в состоянии трактуется как наличие перехода-петли). На самом деле, это требование не столь принципиально, если его опустить, то это влияет лишь на условие выполнения τ-активности при нажатой кнопке. Новое условие такое: реализация может не выполнить никакого перехода по нажатой кнопке *p* только в том случае, если она после конечного числа τ-переходов будет бесконечно двигаться по бесконечному τ-маршруту, который проходит только через такие состояния, где нет переходов по кнопке *p*. В противном случае реализация выполняет конечное число τ-переходов и затем переход по кнопке *p*.

LTS-реализацию, в которой переходы по кнопкам определены не во всех состояниях, можно промоделировать с помощью LTS, в которой такие переходы есть во всех состояниях. Для этого в исходную LTS-реализацию вносятся следующие изменения.

1. Если переход по кнопке *p* отсутствует в стабильном состоянии *s*, то возникает *deadlock*: реализация не может выполнить переход по *p* или τ-переход, поскольку их нет, и не может выполнить переход по наблюдению, поскольку такие переходы блокируются нажатой кнопкой *p*, а разблокированы могут быть только после перехода по *p*. Внешне (для оператора машины тестирования) такой *deadlock* выглядит как отсутствие наблюдений. Из такого *deadlock*`а можно выйти, нажав другую кнопку, по которой в стабильном состоянии *s* есть переход. В нашей модели это реализуется

добавлением перехода $s—p→s`$, ведущего в новое состояние $s`$, в котором определяются переходы-петли $s`—q→s`$ по всем кнопкам q, по которым из состояния s нет переходов, а также переходы по кнопкам, по которым есть переходы из состояния s, ведущие туда же, куда они ведут из состояния s: переход $s_p—r→t$ проводится, когда есть переход $s—r→t$.

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

3. Если переход по кнопке p отсутствует в дивергентном, состоянии s, в котором начинается бесконечный τ-маршрут, проходящий только конечное число раз через состояния, где есть переход по кнопке p, то возможно, что реализация будет проходить именно этот бесконечный τ-маршрут. В этом случае может не выполниться никакой переход по p, и переходы по наблюдениям останутся заблокированы. Внешне (для оператора машины тестирования) такая дивергенция выглядит как отсутствие наблюдений. Из неё можно выйти, нажав другую кнопку, для которой условие этого пункта не будет выполняться. Это полностью аналогично п.1, при моделировании в нашей модели выполняются те же изменения в реализации, которые описаны в п.1.

Литература

1. Бурдонов И.Б., Косачев А.С., Кулямин В.В. Формализация тестового эксперимента. «Программирование», 2007, №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., Косачев А.С. Системы с приоритетами: конформность, тестирование, композиция. Труды Института системного программирования РАН, № 14.1, 2008.
4. Бурдонов И.Б., Косачев А.С. Системы с приоритетами: конформность, тестирование, композиция. "Программирование", 2009, №4.
5. Игорь Бурдонов. Теория конформности (функциональное тестирование программных систем на основе формальных моделей). LAP LAMBERT Academic Publishing, Saarbrucken, Germany, 2011,ISBN 978-3-8454-1747-9.
6. Бурдонов И.Б., Косачев А.С. Удаление из спецификации неконформных трасс. Препринт Института Системного Программирования РАН, 2011, №23.
7. Бурдонов И.Б., Косачев А.С. Пополнение спецификации для ***ioco***. "Программирование", 2011, №1.
8. Бурдонов И.Б., Косачев А.С. Зависимости между ошибками на классах тестируемых реализаций. Труды Института системного программирования РАН, № 23, 2013.

9. Bernot G. Testing against formal specifications: A theoretical view. In S. Abramsky and T.S.E. Maibaum, editors, TAPSOFT'91, Volume 2, pp. 99-119. Lecture Notes in Computer Science 494, Springer-Verlag, 1991.
10. 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.
11. 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.
12. C.A.R. Hoare. Communicating sequential processes. In R.M. McKeag & A.M. Macnaghten, editors, On the construction of programs – an advanced course. Cambridge University Press, 1980, pp. 229-254.
13. Revised Working Draft on "Framework: Formal Methods in Conformance Testing". JTC1/SC21/WG1/Project 54/1 //ISO Interim Meeting /ITU-T on, Paris, 1995.
14. 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.
15. Tretmans J. Test Generation with Inputs, Outputs and Repetitive Quiescence. In: Software-Concepts and Tools, Vol. 17, Issue 3, 1996.
16. Adenilso da Silva Simão, Alexandre Petrenko, Nina Yevtushenko: Generating Reduced Tests for FSMs with Extra States. TestCom/FATES 2009: 129-145.
17. Alexandre Petrenko, Nina Yevtushenko: Testing from Partial Deterministic FSM Specifications. IEEE Trans. Computers 54(9): 1154-1165 (2005).