Формализация тестового эксперимента

И.Б.Бурдонов, А.С.Косачев, В.В.Кулямин

Статья посвящена формальным методам тестирования соответствия (конформности) исследуемой системы заданным требованиям (спецификации). Операционная семантика взаимодействия задаётся с помощью специальной машины тестирования, формально определяющей те или иные тестовые возможности. Выделяется набор теоретически достаточно мощных и практически значимых возможностей, сводимый к наблюдению внешних действий и отказов (отсутствие внешних действий). Нововведениями являются: 1) Параметризация семантики семействами наблюдаемых и ненаблюдаемых отказов, что позволяет учитывать те или иные ограничения на (правильное) взаимодействие. 2) Разрушение как запрещённое действие, которое возможно, но не должно выполняться при правильном взаимодействии. 3) Моделирование дивергенции Δ-действием, которого тоже следует избегать в правильном взаимодействии. Предлагаются основанные на такой семантике понятие безопасного тестирования, реализационная гипотеза о безопасности и отношение безопасной конформности, отвечающее принципу независимости наблюдений: поведение реализации правильно или неправильно независимо от других её поведений. Для более узкого класса взаимодействий могут также использоваться версия семантики, основанная на трассах готовности, и соответствующее отношение конформности. Формулируется ряд утверждений о связи отношений конформности в различных семантиках. Определяются преобразование пополнения, решающее проблему рефлексивности отношения, и монотонное преобразование, решающее проблему монотонности (сохранение конформности при композиции).

1. Введение

Правильность исследуемой системы в самом широком смысле понимается как её соответствие заданным требованиям. Такое соответствие будем называть реальной конформностью. Для верификации (проверки) конформности 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 реальная конформность отображается в модельную конформность. Последняя понимается как обычное математическое соответствие, то есть подмножество декартового произведения множеств реализаций и спецификаций.

Верификация конформности основана на предположении, что моделирование выполнено «правильно». Это даёт возможность считать реальную и модельную конформности эквивалентными: система конформна требованиям тогда и только тогда, когда реализация, как модель системы, конформна спецификации, как модели требований. Конечно, это утверждение тавтологично в том смысле, что, если бы мы захотели дать формальное определение «правильности» моделирования, то оно свелось бы к эквивалентности реальной и модельной конформностей. Тем самым, верификация конформности проверяла бы ровно то, что она должна предполагать. Чтобы выйти из этого порочного круга, нужно понять саму природу моделирования.

Моделирование требований – это процесс их уточнения и формализации. Хотя результатом становится формальное описание требований (спецификация), сам процесс вряд ли может быть формализован, поскольку связывает объекты разной природы: неформальный и формальный. Только наша интуиция может подсказать, правильно ли мы записали неформальные требования в виде формальной спецификации. Если интуиция нас подвела, это может привести (и часто приводит) к выявлению ложных ошибок в исследуемой системе. Тогда приходится пересматривать моделирование требований, то

есть выявлять ошибку не в системе, а в спецификации. После исправления спецификации верификацию конформности приходится повторять.

Моделирование отношения конформности имеет ту же природу: это уточнение и формализация нашего интуитивного представления о том, что означает утверждение «система удовлетворяет требованиям». Формализация предполагает также, что мы абстрагируемся от тех деталей системы, требований и их отношения, которые несущественны с точки зрения интуитивно понимаемой «правильности» системы.

Для верификации конформности нам должны быть заданы в той или иной, но формальной форме, спецификация и отношение конформности. Если реализация, как модель исследуемой системы, также известна, то возможна статическая, аналитическая верификация. Такая верификация сводится к проверке того, что пара <реализация, спецификация> принадлежит допустимому множеству таких пар, определяемому отношением модельной конформности.

В отличие от требований и реальной конформности, исследуемая система – это обычно программный и/или аппаратный комплекс, который на известном уровне абстракции может быть понят вполне формально. Тем самым, отображение системы в реализацию связывает два формальных объекта, что даёт потенциальную возможность определить (и выполнять) это отображение формально. Иногда такое отображение действительно возможно и применяется на практике. Проблема, однако, в том, что, как правило, система и реализация задаются на слишком различных уровнях абстракции, что для сложной системы делает практически невыполнимым её анализ и формальное преобразование в реализацию. При моделировании мы абстрагируемся от большинства деталей устройства и поведения системы, выделяя существенное и несущественное с точки зрения заданных спецификации и модельной конформности. Выполняя такое моделирование не- или полу-формально, мы получаем ещё один источник ложных ошибок. Поэтому часто приходится вообще отказаться от анализа системы и признать, что реализация нам неизвестна, а в лучшем случае известен лишь *класс возможных реализаций*. Кроме того, на практике часто встречаются ситуации, когда исследуемая система вообще не может быть подвергнута формальному анализу. Например, удалённая система, доступ к которой возможен только по каналу связи, или программа, исходный текст которой на языке программирования утерян (формально можно было бы анализировать машинный код, но практически это почти всегда невозможно), или аппаратура как «чёрный ящик».

Здесь возникает вопрос: как проверять конформность, если реализация неизвестна? Это оказывается возможным тогда, когда требования к системе выражены в терминах взаимодействия системы с её окружением. Становится возможным тестирование (динамическая верификация) как проверка конформности в эксперименте. Тест подменяет собой окружение и, взаимодействуя с системой, наблюдает её поведение в этом взаимодействии. Из практических соображений тест должен заканчиваться за конечное время, что вовсе не обязательно для произвольного окружения. Поэтому для того, чтобы сымитировать любое поведение окружения, используется набор тестов, который потенциально может быть бесконечным. Конечно, на практике применяются только конечные наборы тестов, но такое тестирование будет полным (правильно определяющим конформность или неконформность) только при определённых ограничениях на тестируемую систему. Такие ограничения, выраженные в модельной форме, уточняют класс возможных реализаций и называются *реализационными гипотезами* или *моделью ошибок*, если они сводятся к утверждению о том, какие ошибки (типы неконформности) в системе бывают, а какие нет.

Естественно, возникает задача формализовать процесс тестирования. Для этого мы должны, прежде всего, предполагать, что модель системы (реализация) существует, даже если она неизвестна. Это, так называемая, *тестовая гипотеза* [Bern91]. Далее мы должны формализовать само взаимодействие и моделировать реальные тесты модельными

тестами. В модельном мире определяется формальное отношение «реализация *проходит* набор (модельных) тестов». После этого набор модельных тестов объявляется *полным* для заданной спецификации, если он удовлетворяет условию: реализация конформна спецификации тогда и только тогда, когда реализация проходит этот набор тестов. Что мы должны сделать в модельном мире? Во-первых, *доказать* существование полного набора тестов для каждой спецификации (по крайней мере, из заданного класса спецификаций) и известного класса возможных реализаций, и, во-вторых, найти способ *генерации* тестов.

Для практического применения нам нужно определить процедуру *трансляции* модельного теста в реальный тест. Здесь ситуация обратная моделированию системы или требований: мы не определяем модель реального объекта, а строим реальный объект по его заданной модели. После этого полный набор реальных тестов, то есть полученных трансляцией полного набора модельных тестов, прогоняется на исследуемой системе. В реальном мире также определяется отношение «система проходит набор (реальных) тестов». Наконец, делается заключение, что система конформна требованиям тогда и только тогда, когда она проходит полный набор реальных тестов. Разумеется, этот вывод основан на предположении, что не только моделирование требований и отношения конформности, но также трансляция тестов и соответствие реального и модельного отношений *проходит* «правильные».

Далее мы не будем останавливаться на вопросе о «правильности» моделирования и трансляции тестов, предполагая, что они «правильные». Своё внимание мы сосредоточим на модельном мире, разумеется, не забывая о мире реальном, который служит источником всех тех практических ограничений, которые нам придётся учитывать.

Формализация взаимодействия – это ключевой момент в тестировании конформности. От того, какой вид взаимодействия мы рассматриваем, непосредственно зависит отношение конформности и допустимые классы реализаций и спецификаций. Мы будем задавать операционную семантику взаимодействия с помощью так называемой *машины тестирования*. Взаимодействие реализации с окружением всегда подчиняется некоторым ограничениям. Нас интересует не любое взаимодействие, а только такое, которое допускается на практике Это определяется, во-первых, той операционной обстановкой в которой взаимодействие происходит, и, во-вторых, требованиями к самому окружению. Первое означает, что окружение *не может*, а второе – что окружение *не должно* взаимодействовать с реализацией произвольным образом. Иными словами, какого-то взаимодействия не бывает, а какое-то нас не интересует, поскольку мы проверяем правильность реализации, а не её окружения. Абстрагируясь от деталей операционной обстановки и требований к окружению, мы описываем ограничения на взаимодействие с помощью подходящей машины тестирования. Такая машина определяет тот уровень абстракции и те ограничения на взаимодействия, при которых формулируется отношение конформности в терминах такого взаимодействия. С точки зрения тестирования машина задаёт имеющиеся в нашем распоряжении тестовые возможности по управлению взаимодействием и наблюдению поведения реализации. Иными словами, интересующие нас отношения конформности – это те отношения, для проверки которых необходимо и достаточно тестовых возможностей, описываемых машиной тестирования.

Анализ тестовых возможностей мы будем вести по трём аспектам: 1) что мы *хотим* проверять при тестировании; 2) что мы *теоретически можем* проверять с помощью данных тестовых возможностей («мощность тестирования»); 3) на какие тестовые возможности и при каких условиях мы *практически можем* рассчитывать.

Остановимся немного на первом аспекте, непосредственно определяющем тип рассматриваемого отношения конформности при заданных тестовых возможностях. Речь идёт о том, какие поведения реализации в той или иной ситуации спецификация считает правильными, а какие нет. Мы будем предполагать, что реализации отвечает принципу независимости: любое поведение реализации в данной ситуации правильно или неправильно *независимо* от других её поведений. Тогда можно считать, что спецификация

определяет в каждой ситуации t множество *разрешаемых* поведений $\mathbf{\Sigma}_p(t)$. Если реализация демонстрирует множество поведений $\mathbf{I}(t)$, то конформность является предпорядком (рефлексивное и транзитивное отношение) и означает вложенность $\mathbf{I}(t)\subseteq\mathbf{\Sigma}_p(t)$. Для каждого наблюдаемого поведения реализации $i\in\mathbf{I}(t)$ нужно проверить, что $i\in\mathbf{\Sigma}_p(t)$. Такое поведение наблюдается при однократном прогоне теста, поэтому можно при завершении прогона теста выносить вердикт ***pass*** (проходит) или ***fail*** (не проходит). Реализация проходит тест, если при любом его прогоне выносится вердикт ***pass*** (не бывает вердикта ***fail***). Реализация проходит набор тестов, если она проходит каждый тест из набора.

Таким образом, мы не рассматриваем конформности, для проверки которых недостаточно анализировать каждое отдельное поведение реализации и требуется (дополнительный) анализ множества наблюдаемых поведений. В частности, мы не рассматриваем конформности, требующие обязательного наличия в реализации некоторых поведений. Если спецификация в ситуации t определяет множество *обязательных* поведений $\mathbf{\Sigma}_r(t)$, то конформность требует обратной вложенности $\mathbf{I}(t)\supseteq\mathbf{\Sigma}_r(t)$. Для проверки такой конформности требуется анализ всего множества наблюдаемых поведений $\mathbf{I}(t)$ и проверка $s\in\mathbf{I}(t)$ для каждого $s\in\mathbf{\Sigma}_r(t)$. Разновидностью таких конформностей являются эквивалентности, когда любое разрешённое поведение обязательно должно быть в реализации, то есть $\mathbf{\Sigma}_r(t)=\mathbf{\Sigma}_p(t)$. Для такого рода конформностей недостаточно вердикта в конце каждого прогона теста в том смысле, что предикат конформности не является конъюнкцией вердиктов всех прогонов всех тестов из набора (если считать ***pass***=***true*** и ***fail***=***false***).

Мы рассмотрим две известные машины тестирования, предложенные Милнером [Miln81] и Ван Глаббеком [Glab90,Glab93], с их разнообразными модификациями. Параллельно будем определять машину с ограниченным управлением, которая предоставляет минимальные тестовые возможности, доступные в большинстве практических случаев. Другие тестовые возможности либо «избыточны» (позволяют проверять то, что нам не нужно), либо в настоящее время не проработаны в теории, либо «непрактичны».

В качестве примера мы будем использовать широко распространенный частный вид взаимодействия, который сводится к обмену дискретными порциями информации (сообщениями) между реализацией и окружением. Такие системы называют системами ввода-вывода [LT87,Phal94,Tret96]. Сообщение, передаваемое из окружения в систему, называется *стимулом* (*input*), а сообщение, передаваемое из системы в окружение, – *реакцией* (*output*). Следуя нотации алгебры процессов CCS (Calculus of Communicating Systems [Miln80, Miln89]), мы обозначаем стимулы как $?m$, а реакции – как $!m$, где m – символ сообщения.

2. Реализация как модель тестируемой системы

Поскольку наша задача – формализация взаимодействия реализации с окружением, нам безразлично, как реализация устроена «внутри». Для нас важен лишь её внешний интерфейс, позволяющий оказывать на неё тестовые воздействия и наблюдать её внешнее поведение, то есть поведение, проявляющееся во взаимодействии. В соответствии с таким подходом машину тестирования можно представлять себе как «чёрный ящик», внутри которого находится тестируемая реализация и который снабжён разного рода «устройствами» для выполнения тестовых воздействий и наблюдения внешнего поведения реализации. Эти воздействия и наблюдения совершает оператор машины, поведение которого моделирует поведение окружения. Под тестом можно понимать тот или иной вариант поведения окружения, поэтому можно считать, что поведение оператора

определяется тестом. Отличие теста от такого же окружения в том, что в конце работы теста выносится вердикт ***pass*** или ***fail***.

Для наглядности и в качестве иллюстрации мы будем использовать модель исследуемой системы и её окружения, которая называется *системой размеченных переходов* – LTS (*Labelled Transition System*). LTS – это ориентированный граф с выделенной начальной вершиной, дуги которого помечены некоторыми символами. Формально, LTS – это совокупность S=LTS(V_S,L,E_S,s_0), где V_S – непустое множество состояний (вершин графа), L – алфавит символов, называемых внешними действиями, τ – символ, называемый внутренним действием, $E_S \subseteq V_S \times (L \cup \{\tau\}) \times V_S$ – множество переходов (помеченных дуг графа), $s_0 \in V_S$ – начальное состояние (начальная вершина графа). Переход из состояния s в состояние s\` по действию z обозначается $s$—z→s\`. Обозначим s—z↛ $=_{def}$ ∄s\` $s$—z→s\`. Сразу отметим, что разные LTS могут быть неразличимы при тестировании, если они в любом взаимодействии демонстрируют одинаковое внешнее поведение.

Внешнее поведение рассматривается как последовательность дискретных внешних действий, выполнение которых оператор может наблюдать. Для машины задаётся алфавит L *внешних* действий. Можно считать, что внешнее действие $z \in L$ является обозначением для множества действий машины, выполнение каждого из которых наблюдаемо оператором как z, то есть для него они неразличимы между собой. Вместе с тем выполнение в одной и той же ситуации одного или другого действия, когда они оба видимы извне как z, не означает, что последствия выполнения этих действий также будут внешне неразличимы. Также отметим, что фиксация алфавита L не означает, что тестируемая система не может выполнять ещё какие-то действия, которые могли бы наблюдаться извне. Просто нас не интересует поведение системы при выполнении этих других действий, и поэтому они не включены в алфавит L. Например, при тестировании методов объекта, вычисляющих арифметические функции, нас может не интересовать, какие ещё методы определены в классе этого объекта. В LTS-модели с алфавитом L выполнению внешнего действия $z \in L$ соответствует переход s—z→s\`. LTS, находясь в состоянии s, совершает этот переход, что наблюдается оператором как выполнение действия z, и оказывается в состоянии s\`. Сами состояния s и s\` не наблюдаемы.

Кроме внешних, наблюдаемых действий, машина может иметь *внутреннюю активность*, которая внешне (во взаимодействии) не наблюдаема. Её обозначают символом τ. Внутренняя активность может быть как конечной, то есть она прекращается через какое-то время, так и бесконечной. Бесконечную внутреннюю активность называют *дивергенцией*. Без ограничения общности можно считать, что τ-активность представляет собой последовательность дискретных τ-действий, естественно, тоже внутренних, ненаблюдаемых. В общем, символом τ обозначается множество всех внутренних действий машины, которые не наблюдаемы оператором и тем самым для него они неразличимы между собой. Конечная τ-активность – это конечная, а дивергенция – это бесконечная последовательность выполнения τ-действий. В LTS-модели выполнению τ-действия соответствует переход s—τ→s\`.

Будем предполагать, что время выполнения любого внешнего или τ-действия конечно и ограничено снизу ненулевым значением. Тогда конечная последовательность действий выполняется за конечное время, а бесконечная последовательность действий (в частности, дивергенция) выполняется бесконечно долго.

Взаимодействие всегда предполагает участие обеих сторон: реализации и окружения. Например, в системах ввода-вывода передача сообщения означает, что окружение посылает стимул, а реализация принимает, или реализация выдаёт реакцию, а окружение её получает. Иногда говорят, что передача реакции инициируется реализацией, и, коли

реакция выдана, окружение обязано её принять. В других случаях говорят, что стимул, посланный окружением, реализация не должна отвергать. Это можно рассматривать как ограничение на допустимое поведение окружения или реализации. В общем, наличие такого рода ограничений не отменяет общего принципа взаимности взаимодействия.

В машине тестирования этот принцип означает: машина может выполнять только те действия, которые определены в реализации и разрешены оператором. В LTS-реализации действие z определено, если в текущем состоянии s есть хотя бы один переход вида $s\text{—}z\rightarrow s`$. Тестовое воздействие сводится к тому, что оператор указывает, какие действия машине разрешено выполнять. При этом τ-действия, не являющиеся актами взаимодействия, всегда считаются разрешёнными, оператор не может их запретить. Разрешение внешних действий можно понимать и так, что оператор даёт машине приказ выполнить одно из этих действий на её выбор. Например, окружение принимает все реакции, но, какая именно реакция будет передана, определяет реализация.

Хотя выполняться может только определённое и разрешённое действие, в общем случае не всякое такое действие выполнимо. Выполнимость действия z есть предикат от множества определённых действий и множества разрешённых действий. Этот предикат может быть различным в различные моменты времени (в различных состояниях LTS-реализации) и для различных действий (для различных переходов LTS-реализации). Таким способом в машине могут быть заданы приоритеты выполнения действий. В данной статье мы будем рассматривать, как правило, *машины без приоритетов*: всякое определённое и разрешённое действие выполнимо.

Правило недетерминированного выбора: если есть несколько выполнимых действий, то выполняется только одно из них, выбираемое недетерминированным образом. В этом кроется причина возможного недетерминизма системы, поскольку мы абстрагируемся от тех факторов («погодных условий»), которые определяют выбор того или иного действия. Заметим, что даже в том случае, когда все выполнимые действия помечены одним и тем же символом ($z\in L$ или τ), они могут быть различными, в том числе и по своим наблюдаемым последствиям. В LTS-реализации это означает, что в текущем состоянии s может быть определено несколько переходов вида $s\text{—}z\rightarrow s`$, отличающихся только конечным состоянием $s`$.

Для машины без приоритетов выбор выполняемого действия при взаимодействии моделируется в LTS с помощью оператора композиции, заимствуемого из той или иной алгебры процессов. Нам удобнее использовать оператор композиции алгебры процессов CCS [Miln80, Miln89]. Для этого на универсуме внешних действий определяется инволюция (биекция, обратная сама себе) «подчёркивание», которая каждому внешнему действию z ставит в соответствие *противоположное* действие $\underline{z}$ так, что $\underline{\underline{z}}=z$. Заметим, что $z\in L$ не обязательно влечёт $\underline{z}\in L$. Результатом композиции двух LTS $\mathbf{S}=LTS(V_{\mathbf{S}},L,E_{\mathbf{S}},s_0)$ и $\mathbf{T}=LTS(V_{\mathbf{T}},M,E_{\mathbf{T}},t_0)$ является третья LTS $\mathbf{S}\rceil\lfloor\mathbf{T}=LTS(V_{\mathbf{S}}\times V_{\mathbf{T}},L\rceil\lfloor M,E,s_0t_0)$. Её состояния – это пары состояний LTS-операндов, начальное состояние – пара начальных состояний. В алфавит композиции попадают те действия из алфавита каждого LTS-операнда, для которых в алфавите другого LTS-операнда не нашлось противоположных действий: $L\rceil\lfloor M=(L\backslash\underline{M})\cup(M\backslash\underline{L})$. Композиционные переходы делятся на асинхронные и синхронные переходы. Асинхронному переходу соответствует переход в одном из LTS-операндов, помеченный внешним действием композиционного алфавита или символом τ; измениться может состояние только этого LTS-операнда. Синхронному переходу соответствует пара переходов в LTS-операндах по противоположным действиям, которая в композиции становится τ-переходом; измениться могут состояния обоих LTS-операндов. Формально

множество композиционных переходов E – это наименьшее множество, порождаемое следующими правилами вывода:

$$l\in(L\setminus\underline{M})\cup\{\tau\} \quad \& \; s\text{—}l\rightarrow s` \qquad\qquad \vdash st\text{—}l\rightarrow s`t;$$

$$m\in(M\setminus\underline{L})\cup\{\tau\} \qquad\qquad \& \; t\text{—}m\rightarrow t` \quad \vdash st\text{—}m\rightarrow st`;$$

$$z\in L\cap\underline{M} \qquad \& \; s\text{—}z\rightarrow s` \;\&\; t\text{—}\underline{z}\rightarrow t` \quad \vdash st\text{—}\tau\rightarrow s`t`.$$

Если оператор в машине с LTS-реализацией $\mathbf{S}$ разрешает действие $z\in L$, то в LTS-тесте $\mathbf{T}$ этому соответствует переход вида $t\text{—}\underline{z}\rightarrow t`$. Если оператор может «передумать» и, не дожидаясь выполнения внешнего действия, разрешить другое множество внешних действий, то в LTS-тесте $\mathbf{T}$ этому соответствует $\tau$-переход вида $t\text{—}\tau\rightarrow t`$. Обычно считается, что при тестировании система <реализация-тест> замкнута, то есть они взаимодействуют только друг с другом и не взаимодействуют с остальным внешним миром. В этом случае $M=\underline{L}$ и $L\rceil\lfloor M=\varnothing$, то есть в композиции $\mathbf{S}\rceil\lfloor\mathbf{T}$ остаются только τ-переходы (наследованные от операндов или синхронные).

Заметим, что на практике реальный тест (точнее, тестер, выполняющий тест и моделируемый оператором машины тестирования) может взаимодействовать с тестируемой системой не напрямую, а через некоторую среду передачи. Стандартным примером является среда из очеред стимулов и очереди реакций в системах ввода-вывода. Кроме того, тестируемая система может взаимодействовать не только с тестом и/или средой передачи, но и некоторой другой частью окружения. Например, у нас может не быть возможности «перехватывать» всё внешнее взаимодействие системы. В этом случае в модельном мире мы должны считать, что тестируется композиция реализации со средой передачи и другой частью окружения. Тогда говорят, что реализация погружена в тестовый контекст, а тестирование называют *асинхронным* или *тестированием в контексте*. Фактически, здесь меняется само отношение конформности. Тестер также может получать команды «сверху» (изменение режима тестирования или аварийное завершение работы), что в модельном мире можно понимать как замену теста на композицию теста с «вышестоящей инстанцией». Далее будем считать, что система <реализация-тест> замкнута.

3. Управление и наблюдение внешних действий

Итак, в терминах машины тестирования тестовое воздействие сводится к указанию множества разрешённых внешних действий. Способ такого указания зависит от устройства машины.

Машина Ван Глаббека называется *генеративной* машиной: каждому внешнему действию $z\in L$ соответствует переключатель, который имеет два положения: *free* (разрешение действия) и *blocked* (запрещение действия). Можно устанавливать любые переключатели в любые положения. Множество переключателей в положении *free* – это и есть множество разрешённых действий $P\subseteq L$.

Машина Милнера называется *реактивной* машиной и работает «по приказу»: для каждого внешнего действия $z\in L$ в ней имеется кнопка, нажатие которой разрешает выполнять только это действие z. Это налагает ограничения на работу окружения: в LTS-модели теста $\mathbf{T}$ в каждом состоянии должно быть не более одного перехода вида $t\text{—}\underline{z}\rightarrow t`$, где $z\in L\cap\underline{M}$. Отсутствие переходов означает, что в данный момент времени (в данном состоянии теста) оператор не нажимает никакой кнопки. После выполнения внешнего действия машина *приостанавливается* до нажатия следующей кнопки.

Мы вводим машину тестирования *с ограниченным управлением*, которую будем называть также *параметризованной* машиной. Как и реактивная машина, она работает «по

приказу», но кнопка разрешает не одно внешнее действие, а множество действий $P\subseteq L$; такую кнопку будем обозначать “P”. Предполагается, что каждое внешнее действие $z\in L$ разрешается хотя бы одной кнопкой, то есть существует такая кнопка “P”, что $z\in P$. Мы говорим, что наша машина – это машина с *ограниченным* управлением потому, что в ней не для каждого множества $P\subseteq L$ может существовать кнопка “P”. Машина параметризуется набором кнопок, то есть семейством подмножеств внешних действий $\mathfrak{R}\subseteq\mathcal{P}(L)$. Нажатая кнопка автоматически отжимается, если машина выполняет внешнее действие или оператор нажимает другую кнопку.

Сравним эти три машины. Параметризованную машину можно понимать как модификацию генеративной машины, в которой не любое положение переключателей допустимо. Если реактивную машину модифицировать, разрешив одновременно нажимать несколько кнопок, то такая реактивная машина будет работать аналогично генеративной машине. С учётом этих модификаций все три машины можно считать эквивалентными за одним исключением.

Таким исключением является поведение машины после выполнения внешнего действия до нового нажатия кнопок или изменения положения переключателей. Реактивная машина приостанавливается до тех пор, пока не будут ещё раз нажаты те же или другие кнопки. Генеративная машина продолжает работать, выполняя внутренние действия и внешние действия, разрешённые переключателями в положении *free*. В параметризованной машине реализован третий вариант: машина продолжает работать, но выполняться могут только внутренние действия (кнопка отжата). Посмотрим, как поведение одной машины может имитироваться в другой машине.

Ван Глаббек показал, что приостановка реактивной машины может имитироваться в генеративной машине специальным переключателем (on/off-switch), блокирующим выполнение не только всех внешних, но и внутренних действий. Для имитации оператор должен сразу после наблюдения внешнего действия нажать блокирующий переключатель, а после установки основных переключателей убрать блокировку. Заметим, однако, что оператор моделирует окружение, которое может быть как очень быстрым, так и очень медленным. Поэтому имитация поведения реактивной машины на самом деле ничего не добавляет, оставляя и то поведение, которое есть в генеративной машине. В общем понятно, что блокирующий переключатель не увеличивает мощность тестирования.

Обратно, поведение генеративной машины, когда хотя бы одно внешнее действие разрешено, имитируется в реактивной машине последовательностью нажатия одного и того же набора кнопок. Запрет всех внешних действий (все переключатели в положении *blocked*) Ван Глаббек предлагает имитировать в реактивной машине нажатием кнопки «лишнего» действия, про которое точно известно, что оно никогда не может выполняться. В алфавит L добавляется «лишнее» действие, а в машину – разрешающая его кнопка.

Нам достаточно показать, как поведение генеративной машины можно имитировать в параметризованной машине и наоборот. Поведение генеративной машины имитируется в параметризованной машине быстрым нажатием той же самой кнопки после выполнения внешнего действия. Заметим, что, если оператор не успел достаточно быстро нажать кнопку, ничего страшного не случится, поскольку машина успеет выполнить только одно или несколько τ-действий, которые (*в машине без приоритетов*) она может выполнить и в том случае, когда кнопка была нажата немедленно. Иными словами, мы требуем, чтобы оператор мог работать быстро, но не заставляем его всегда работать быстро. Это согласуется с тем, что оператор должен моделировать любую скорость работы окружения.

Для имитации поведения параметризованной машины в генеративной машине можно использовать упоминаемую Ван Глаббеком модификацию генеративной машины, в которой переключатели автоматически сбрасываются в положении *blocked* каждый раз, когда выполняется внешнее действие. Это в точности соответствует в параметризованной

машине тому, что после выполнения внешнего действия до нового нажатия какой-либо кнопки могут выполняться только τ-действия.

Для наблюдения внешних действий в генеративной и параметризованной машине имеется дисплей. Когда машина выполняет внешнее действие z, символ z высвечивается на дисплее. В параметризованной машине дисплей гаснет при нажатии следующей кнопки. Поскольку генеративная машина «непрерывного действия», в ней возможно наблюдение последовательности $\langle z, z, z, \dots \rangle$ без изменения положения переключателей. Чтобы оператор мог различать разные выполнения одного действия, между двумя последовательными действиями дисплей должен гаснуть на короткий интервал времени t_0. В реактивной машине дисплея нет, но кнопка выполняемого действия «проваливается», машина приостанавливается и для продолжения тестирования нужно отжать «провалившуюся» кнопку и нажать ту же самую или новую кнопку (или несколько кнопок в модифицированной машине). Очевидно, эти различия несущественны.

Подводя итоги, отметим два важных момента (для определённости, в терминах параметризованной машины).

Переключение без наблюдения. В машине без приоритетов выполнимость τ-действий не зависит от множества разрешённых внешних действий (кнопок и переключателей). Поэтому нет смысла переключать кнопки без наблюдения: такое переключение не увеличивает мощность тестирования. Действительно, если была нажата кнопка “P”, а потом без наблюдения нажата другая кнопка “Q”, то в этом интервале времени машина могла выполнять только τ-действия. Но эти τ-действия она могла бы выполнять и в том случае, когда вместо кнопки “P” сразу нажималась кнопка “Q”, а второй раз, естественно, не нажималась. Тем самым, любое поведение (трассу внешних действий), которое можно наблюдать в первом случае, можно было бы наблюдать и во втором случае.

При наличии приоритетов переключение без наблюдения необходимо для полноты тестирования, поскольку различные множества разрешённых действий по-разному влияют на выполнение τ-действий, что приводит к внешне различимым поведениям.

Тестовые истории и трассы наблюдений. Собирая полную информацию о процессе тестирования, оператор мог бы записывать последовательность выполняемых им действий и наблюдений. Такую последовательность нажимавшихся кнопок и наблюдаемых внешних действий будем называть *тестовой историей*, а её подпоследовательность наблюдений – *трассой наблюдений*, *наблюдаемой трассой* или просто *трассой*. В общем случае всё, что мы можем узнать при тестировании о реализации, сводится к множеству всех возможных тестовых историй. При наличии приоритетов нам недостаточно трасс, и приходится работать с историями.

Однако в машине без приоритетов выполнимость внешнего действия не зависит от того, какие ещё внешние действия разрешены. Если действие $z \in P$ наблюдалось после нажатия кнопки “P”, то оно точно также могло наблюдаться после нажатия любой другой кнопки “Q” такой, что $z \in Q$. Поэтому множество всех тестовых историй однозначно восстанавливается по множеству всех трасс наблюдений. Такое множество трасс будем называть *трассовой моделью реализации* или просто *трассовой реализацией*. В LTS-модели $\mathbf{I}$ трасса определяется как последовательность внешних действий, которыми помечены переходы маршрута LTS, начинающегося в начальном состоянии, с пропуском τ-переходов. LTS-реализации различимы при взаимодействии только с точностью до их трассовой модели $\mathbf{I} = \boldsymbol{traces}(\mathbf{I})$. Поэтому спецификацию можно рассматривать как описание требований к множеству трасс реализации. Для выбранного нами типа конформности спецификация задаёт множество разрешаемых трасс, которому должны принадлежать все трассы конформной реализации. Сама спецификация может быть задана как множество трасс соответствующей LTS $\mathbf{\Sigma} = \boldsymbol{traces}(\mathbf{S})$; это множество трасс будем

называть *трассовой моделью спецификации* или просто *трассовой спецификацией*. Предполагается, что трассовая спецификация однозначно определяет множество разрешаемых трасс, хотя, как увидим ниже, может с ним не совпадать.

Трассовая модель (как реализации, так и спецификации) является деревом (*prefix-closed*): если наблюдается (разрешается) некоторая трасса, то любой её префикс также наблюдается (разрешается). Это означает, что тестирование имеет смысл вести «до первой ошибки»: если после наблюдения разрешённой трассы σ наблюдается внешнее действие z, а трасса $\sigma\cdot\langle z\rangle$ запрещена, то тестирование можно завершить с вердиктом ***fail***, поскольку любое продолжение $\sigma\cdot\langle z\rangle\cdot\lambda$ также запрещено. Более того, как мы увидим ниже, продолжение тестирования в этой ситуации может быть «опасным».

4. Остановка машины и наблюдение отказов

Когда в машине нет выполнимых действий, она останавливается. Если оператор может это обнаруживать, то он получает новый вид наблюдения: машина стоит в состоянии, в котором не выполнимо ни одно действие, разрешённые действия образуют множество P. Такое множество P называется *отказом* (*refusal set*). Теперь в тестовые истории и трассы наблюдений мы будем помещать не только действия из алфавита L, но и наблюдаемые отказы как подмножества действий $P\subseteq L$.

В машине без приоритетов остановка возможна только при отсутствии внутренней активности. LTS-реализация останавливается в состоянии s без τ-переходов $s\overset{\tau}{\not\longrightarrow}$; такое состояние называется *стабильным*. Отказ P может наблюдаться в таком стабильном состоянии s, в котором $\forall z\in P\ s\overset{z}{\not\longrightarrow}$. Правило «Переключение без наблюдения» соответственно корректируется: под наблюдениями понимаются не только внешние действия, но и отказы. Возможность восстановления тестовых историй по трассам с отказами также сохраняется, поскольку отказ P можно наблюдать только при нажатии кнопки “P”. Для получения трасс с отказами (*failure traces*) по LTS-реализации достаточно добавить отказы в алфавит внешних действий LTS, в каждом её стабильном состоянии добавить переходы-петли по наблюдаемым в этом состоянии отказам, и взять множество трасс внешних действий полученной LTS: **I**=***Ftraces***(**I**).

Для наблюдения отказов в генеративной и реактивной машинах имеется *зелёная лампочка*, которая горит, если машина выполняет какое-либо внешнее или внутреннее действие, и гаснет, когда машина останавливается. Отказ вычисляется как множество действий, написанных на переключателях в положении *free* в генеративной машине, или на тех кнопках, которые нажимает оператор в реактивной машине. Возможен другой режим работы зелёной лампочки, когда она горит только при выполнении внутренних действий. Если зелёная лампочка гаснет, то это либо остановка, либо выполнение внешнего действия. В реактивной машине остановка обнаруживается, если кнопка не проваливается. В генеративной машине – если дисплей погашен в течении времени $t>t_0$, чтобы убедиться, что это остановка, а не промежуток между выполнением двух внешних действий.

Хотя для обнаружения остановки годятся оба режима работы зелёной лампочки, они различаются мощностью тестирования. Второй режим позволяет делать отдельное наблюдение внутренней активности (непустой последовательности внутренних действий), когда зелёная лампочка горит, а дисплей пуст. Заметим, что сами внутренние действия, по-прежнему, неразличимы между собой, в том числе неразличимы одно τ-действие и любая конечная последовательность τ-действий. Если наблюдение внутренней активности обозначить τ, то различаются трассы $\langle z,\tau,z`\rangle$ и $\langle z,z`\rangle$, а также трассы $\langle z,\tau,P\rangle$ и

$\langle z,P\rangle$, где $z,z`\in L$ и $P\subseteq L$, которые в первом режиме не различимы. Что касается практичности такой тестовой возможности, то это зависит от условий взаимодействия. Например, когда мы запускаем программу на своём компьютере, а она долго не отвечает, мы можем «подсмотреть», тратит ли программа процессорное время, изменяет ли какие-то переменные или файлы. С другой стороны, если программа предназначена для удалённого взаимодействия, то при соответствующем удалённом тестировании такой возможности нет. Однако более важный вопрос заключается в том, нужна ли нам такая тестовая возможность? Трудно представить себе ситуации, когда наличие или отсутствие конечной внутренней активности между внешним действием (или началом работы) и следующим внешним действием или отказом должно трактоваться как ошибка.

Для обнаружения остановки вовсе не обязательна возможность наблюдать (внутреннюю) активность реализации. Например, пусть время выполнения любого внешнего действия и любой конечной внутренней активности ограничено сверху известным значением t_1. Тогда истечение тайм-аута $t>t_1$ при отсутствии внешних действий означает либо дивергенцию, либо остановку машины. Если мы уверены, что дивергенции нет, то остаётся только остановка. В системах ввода-вывода такой тайм-аут может устанавливаться при приёме всех реакций без посылки стимулов. Соответствующий отказ называют *стационарностью* – это множество всех реакций, обозначаемое символом δ [Tret96,Vaan91]. В LTS-реализации такой отказ наблюдается в *стационарном состоянии* (*quiescent state*) – стабильном состоянии, в котором нет переходов по реакциям. Если посылка одного стимула $?x$ предусматривает достоверное (безошибочное и, значит, не требующее тестирования) «уведомление о вручении», которое должно придти за время, ограниченное сверху известным значением, то при отсутствии дивергенции мы можем наблюдать отказ $\{?x\}$, называемый *блокировкой стимула* (*input refusal*). В LTS-реализации блокировка $\{?x\}$ наблюдается в стабильном состоянии, в котором нет перехода по стимулу $?x$. Подчеркнём, что истечение тайм-аута само по себе не позволяет различить дивергенцию и остановку машины. Наблюдение остановки возможно только при условии, что в машине нет дивергенции. Эту проблему мы рассмотрим позже.

Другой вариант: реализация может сама изнутри чёрного ящика сообщать машине не только о выполнении внешнего действия, но и об остановке. Например, для наблюдения стационарности вместо тайм-аута можно использовать достоверное «уведомление о прекращении передачи реакций». Для наблюдения блокировки стимула вместо тайм-аута можно использовать не только достоверное «уведомление о вручении», но и достоверное «уведомление о невручении».

В параметризованной машине мы предлагаем абстрагироваться от способа обнаружения остановки. Будем считать, что, если при нажатии кнопки “P” может быть обнаружена остановка, то это делает сама машина и высвечивает на дисплей специальный символ θ. В LTS-тесте этому соответствует добавление в алфавит символа θ и возможность определять θ-переходы. При композиции θ-переход выполним тогда и только тогда, когда никакие другие переходы не выполнимы. Для машины без приоритетов это записывается следующим дополнительным правилом вывода:

Deadlock $\&\ t\text{—}\theta\rightarrow t` \quad \vdash st\text{—}\tau\rightarrow st`$, где

Deadlock $=_{\text{def}}\ s\text{—}\tau\not\rightarrow\ \&\ t\text{—}\tau\not\rightarrow\ \&\ \forall z\in L\cap\underline{M}\ (s\text{—}z\not\rightarrow\ \vee\ t\text{—}\underline{z}\not\rightarrow)$.

Для замкнутой системы <реализация-тест> имеем $M=\underline{L}$ и $L\rceil\lfloor M=\varnothing$. Поэтому при остановке реализации в состоянии s возможен любой отказ $P\subseteq\{z\in L\,|\,s\text{—}z\not\rightarrow\}$. В машине, параметризованной семейством $\mathfrak{R}$, должно быть также $P\in\mathfrak{R}$.

Теперь уточним параметризацию нашей машины, учитывая отказы. Мы будем считать, что одни отказы могут наблюдаться при остановке, а другие нет. Например, в системах ввода-вывода у нас может быть тайм-аут ожидания реакций, но не быть никакого уведомления о вручении или невручении стимула. В этом случае стационарность наблюдается (при отсутствии дивергенции), а блокировки стимулов не наблюдаются. Такие системы ввода-вывода рассматриваются, например, для отношений конформности ***iot***, ***ioconf***, ***ior*** и ***ioco***, которые мы рассмотрим ниже.

Таким образом, семейство кнопок разбивается на два подсемейства $\mathfrak{R}\subseteq\mathcal{P}(L)$ – кнопки с наблюдаемыми отказами, и $\mathfrak{Q}\subseteq\mathcal{P}(L)$ – кнопки с ненаблюдаемыми отказами, суммарно покрывающие алфавит: $(\cup(\mathfrak{R}))\cup(\cup(\mathfrak{Q}))=L$. Будем считать, что $\mathfrak{R}\cap\mathfrak{Q}=\varnothing$, поскольку при наличии кнопки “P” с наблюдаемым отказом добавление кнопки “P” с ненаблюдаемым отказом не увеличивает мощность тестирования. Машину, параметризованную семействами $\mathfrak{R}$ и $\mathfrak{Q}$ будем называть $\mathfrak{R}/\mathfrak{Q}$-машиной и говорить, что она определяет тестовую $\mathfrak{R}/\mathfrak{Q}$-семантику взаимодействия. Трассы с отказами из $\mathfrak{R}$ будем называть $\mathfrak{R}$-трассами, а множество таких трасс LTS – $\mathfrak{R}$-моделью.

F*-моделью* будем называть $\mathcal{P}(L)$-модель. Для реализации её ***F***-модель – это множество трасс наблюдений на $\mathcal{P}(L)/\varnothing$-машине, которую будем называть ***F****-машиной*. Для ***F***-модели **T** подмножество её $\mathfrak{R}$-трасс является $\mathfrak{R}$-моделью (**!**)[1], которую будем обозначать $\mathbf{T}_{\mathfrak{R}}=\mathbf{T}\cap(L\cup\mathfrak{R})^*$. Верно и обратное: любая $\mathfrak{R}$-модель является подмножеством $\mathfrak{R}$-трасс некоторой ***F***-модели (**!**). Далее реализацию и спецификацию будем понимать как ***F***-модели и обозначать как **I** и **Σ**, соответственно. По LTS-моделям строятся соответствующие ***F***-модели взятием всех их трасс со всеми отказами **I**=***Ftraces***(**I**) и **Σ**=***Ftraces***(**S**). Как по спецификации **Σ** определяется множество разрешённых трасс, мы увидим ниже.

Отметим, что Ван Глаббек и Милнер не рассматривали таких вариантов своих машин: у них зелёная лампочка либо есть и все отказы наблюдаемы, либо нет и все отказы не наблюдаемы. Множество наблюдаемых трасс: либо ***F***-модель (все подмножества алфавита являются наблюдаемыми отказами), либо $\varnothing/\{L\}$-модель (нет наблюдаемых отказов, трассы содержат только внешние действия).

5. Протокол взаимодействия и безопасное тестирование

Остановка машины может вызывать или не вызывать тупик при взаимодействии. Для определённости будем рассматривать $\mathfrak{R}/\mathfrak{Q}$-машину. Тупик не возникает в двух случаях. 1) Окружение может продолжить работу не дожидаясь, чем закончится последний шаг взаимодействия: внешним действием или отказом, и закончится ли вообще. Оператор переключает кнопку при отсутствии наблюдений. В состоянии LTS-окружения/теста определён τ-переход. 2) При остановке возникает наблюдаемый отказ и окружение реагирует на него. Оператор наблюдает отказ, после чего нажимает (другую) кнопку. Состояние LTS-окружения/теста стабильно и в нём определён θ-переход.

Возникновение тупика можно считать нормальным завершением взаимодействия, если это совпадает с завершением работы окружения. Оператор не нажимает и не будет

---

[1] Значком (**!**) мы обозначаем утверждения, доказательство которых в данной статье не приводится из экономии места.

нажимать кнопки: все внешние действия запрещены, никаких наблюдений больше не будет. LTS-окружение (тест) находится в терминальном состоянии (состоянии без переходов). При тестировании выносится вердикт ***pass*** или ***fail***.

Тупик в нетерминальном состоянии окружения мы будем рассматривать как ошибку взаимодействия: окружение/тест «желает» продолжения, но оно невозможно. Заметим, что в этом случае нажата какая-то кнопка. «Виновником» такой ошибки может быть как реализация, так и окружение. Допустим, что реализация «не виновата». Тогда, если отказ наблюдаемый, то окружение «виновато» в том, что не реагирует на него, хотя могло бы это делать. Если отказ ненаблюдаемый, то «вина» окружения заключается в том, что оно позволило этому произойти: оператор не должен был нажимать кнопку “P”, если отказ P ненаблюдаемый и может возникнуть в данной ситуации. Если окружение всё делает «правильно», а ошибка взаимодействия возникает, то «виновата» реализация: нажимаемая кнопка не позволяет наблюдать отказ, а реализация останавливается. Проблема, однако, в том, что ошибка взаимодействия, по определению, не проверяема при тестировании: возможность проверки как раз означала бы, что отказ наблюдаем.

Если нажатие кнопки “P” после трассы σ не может вызывать ошибки взаимодействия (ненаблюдаемого отказа), то будем называть такую кнопку *безопасной в реализации после трассы* σ. Это означает, что в ***F***-модели реализации $\mathbf{I}$ трасса $\sigma \in \mathbf{I}_{\mathfrak{R}}$ не продолжается отказом P, если $P \in \mathfrak{Q}$ (в LTS-реализации трасса σ не заканчивается в стабильном состоянии, в котором нет переходов по действиям из P):

$$P \ \textbf{\textit{safe in}} \ \mathbf{I} \ \textbf{\textit{after}} \ \sigma =_{\text{def}} P \in \mathfrak{R} \vee \sigma \cdot \langle P \rangle \notin \mathbf{I}.$$

Протоколом взаимодействия будем называть правило, определяющее, какие $\mathfrak{Q}$-кнопки после каких трасс наблюдений разрешено нажимать, а какие нет. Первые будем называть *безопасными*, а вторые – *опасными после трассы в данном протоколе*. $\mathfrak{R}$-кнопки объявляются безопасными (в смысле отсутствия ненаблюдаемых отказов) после любой трассы. Таких протоколов может быть много. Нас интересует не любое взаимодействие, а только то, которое происходит по правилам некоторого заданного i-го протокола. Будем говорить, что реализация отвечает i-ой *гипотезе о безопасности*, если при взаимодействии с ней любого окружения, выполняющего правила i-го протокола, не возникает ошибок взаимодействия. Каждая i-ая гипотеза о безопасности определяет свой i-ый класс *безопасно-тестируемых* реализаций. Безопасно-тестируемость реализации в заданном i-ом протоколе невозможно проверить при тестировании, поскольку такая проверка означала бы проверку наличия или отсутствия ошибки взаимодействия. Поэтому соответствующая гипотеза о безопасности является предусловием тестирования. В свою очередь, тест (оператор машины) должен соблюдать i-ый протокол взаимодействия, и такое тестирование будем называть *безопасным*.

Только в этом контексте можно говорить о конформности или неконформности реализации: каждый i-ый протокол взаимодействия определяет свой, i-ый класс конформных реализаций как подкласс i-ого класса безопасно-тестируемых реализаций. Цель безопасного тестирования по i-ому протоколу – проверка конформности реализации при условии, что реализация взята из i-го класса безопасно-тестируемых реализаций. Заметим, что реализация, конформная в одном протоколе, может оказаться не безопасно-тестируемой в другом протоколе.

Теперь уточним определение конформности как вложенности наблюдаемых трасс реализации во множество трасс, разрешаемых спецификацией. Мы должны параметризовать конформность заданным i-ым протоколом взаимодействия. Во-первых, нас интересуют не все реализации, а только те, которые удовлетворяют i-ой гипотезе о безопасности. Во-вторых, нас интересуют не все трассы таких реализаций, которые могут

наблюдаться на данной $\mathfrak{R}/\mathfrak{Q}$-машине, а только те, которые наблюдаемы при взаимодействии по i-ому протоколу. Во множестве таких трасс спецификация выделяет подмножество разрешённых трасс. Теперь разрешённые трассы – это не все трассы, которые могут быть в конформных реализациях, а только те из них, которые могут наблюдаться при безопасном тестировании. Мы будем считать, что спецификации соответствует $\boldsymbol{F}$-модель $\boldsymbol{\Sigma}$, и множество разрешённых трасс одно и то же в любом протоколе взаимодействия – это подмножество $\mathfrak{R}$-трасс $\boldsymbol{\Sigma}_{\mathfrak{R}}$. Остальные трассы, которые могут быть в конформных реализациях, но не наблюдаются при безопасном тестировании, определяются протоколом. Для того, чтобы множество разрешённых трасс совпадало с $\boldsymbol{\Sigma}_{\mathfrak{R}}$, ограничиваемся такими протоколами, которые удовлетворяют следующим правилам.

Первое правило: Все $\mathfrak{R}$-кнопки объявляются безопасными после любой $\mathfrak{R}$-трассы.

Второе правило: Все трассы из $\boldsymbol{\Sigma}_{\mathfrak{R}}$ могут быть проверены при тестировании. Это означает: если трасса $\sigma\cdot\langle z\rangle\in\boldsymbol{\Sigma}_{\mathfrak{R}}$, то протокол должен объявить безопасной после σ *хотя бы одну* кнопку “P”, разрешающую действие z, то есть $z\in P$. Заметим, что $\mathfrak{R}$-кнопки безопасны по любому протоколу, но мы не делаем ограничений сверху на множество безопасных $\mathfrak{Q}$-кнопок. В частности, можно все $\mathfrak{Q}$-кнопки, разрешающие действие z, объявить безопасными после трассы σ. Это правило позволяет проверить, продолжается ли в реализации трасса σ внешним действием z или нет. Если после нажатия кнопки “P” наблюдается действие z, то тестирование можно продолжить, чтобы сравнить поведение реализации и спецификации после трассы $\sigma\cdot\langle z\rangle$. Если наблюдается действие $z`$, которого нет в спецификации $\sigma\cdot\langle z`\rangle\notin\boldsymbol{\Sigma}$, то это ошибка (неконформность).

Третье правило: Протокол не налагает на реализацию лишних ограничений, то есть все $\mathfrak{Q}$-кнопки, которые протокол не объявил безопасными по первому и второму правилам, объявляются опасными и, тем самым, гипотеза о безопасности разрешает им быть опасными в реализации. В частности, если в спецификации трасса σ не продолжается действиями, разрешаемыми некоторой $\mathfrak{Q}$-кнопкой “Q”, то есть $\forall z\in Q\ \sigma\cdot\langle z\rangle\notin\boldsymbol{\Sigma}$, то кнопка “Q” опасна после трассы σ по любому протоколу. Заметим, что, если бы протокол объявил безопасной такую кнопку, то любая безопасно-тестируемая реализация, имеющая разрешённую трассу σ, оказалась бы неконформной.

Теперь, кроме разрешённой трассы $\sigma\in\boldsymbol{\Sigma}_{\mathfrak{R}}\cap\mathbf{I}$, в конформной реализации могут быть ненаблюдаемые для данного протокола трассы. Во-первых, после трассы σ может быть ненаблюдаемый отказ $Q\in\mathfrak{Q}$, если кнопка “Q” объявлена протоколом опасной после σ и не нарушаются гипотеза о безопасности и условие конформности: для каждой кнопки “P”, безопасной по протоколу после σ, $\exists u\ (u\in P\backslash Q \vee u=P\in\mathfrak{R})\ \&\ \sigma\cdot\langle u\rangle\in\mathbf{I}\cap\boldsymbol{\Sigma}$. Во-вторых, допускается любая трасса вида $\sigma\cdot\langle z\rangle\cdot\lambda$, где внешнее действие z разрешается только кнопками, объявленными протоколом опасными после σ. Это непосредственно следует из того, что опасные кнопки не нажимаются при безопасном тестировании и, следовательно, наличие или отсутствие таких продолжений трассы σ не может быть проверено. Разумеется эти дополнительные трассы реализации не могут наблюдаться при безопасном тестировании по данному протоколу.

В качестве примера рассмотрим $\mathfrak{R}/\mathfrak{Q}$-машину с $\mathfrak{R}=\varnothing$ и $\mathfrak{Q}=\{\{a,b\},\{b,c\}\}$ и спецификацию $\boldsymbol{\Sigma}_{\mathfrak{R}}=\{\epsilon,\cdot\langle b\rangle\}$. Возможны три типа протокола взаимодействия в зависимости от того, какие кнопки считаются безопасными в самом начале (после пустой

трассы): 1) только “$\{a,b\}$”, 2) только “$\{b,c\}$”, 3) любые. Реализация $\mathbf{I}$ безопасно-тестируема, если: 1) $\langle b\rangle\in\mathbf{I} \vee \langle a\rangle\in\mathbf{I}$, 2) $\langle b\rangle\in\mathbf{I} \vee \langle c\rangle\in\mathbf{I}$, 3) $\langle b\rangle\in\mathbf{I} \vee \langle a\rangle\in\mathbf{I} \ \& \ \langle c\rangle\in\mathbf{I}$. Реализация конформна, если: 1) $\langle b\rangle\in\mathbf{I} \ \& \ \langle a\rangle\notin\mathbf{I}$, 2) $\langle b\rangle\in\mathbf{I} \ \& \ \langle c\rangle\notin\mathbf{I}$, 3) $\langle b\rangle\in\mathbf{I} \ \& \ \langle a\rangle\notin\mathbf{I} \ \& \ \langle c\rangle\notin\mathbf{I}$. В этом примере в любой конформной реализации вместе с любой *непустой* трассой σ может быть любое продолжение $\sigma\cdot\lambda$.

6. Гипотеза о безопасности и безопасная конформность

Будем задавать протокол взаимодействия в форме отношения «кнопка безопасна после $\mathfrak{R}$-трассы спецификации» P ***safe by*** $\mathbf{\Sigma}$ ***after*** σ, которое должно отвечать описанным выше правилам: $\forall\sigma\in\mathbf{\Sigma}_{\mathfrak{R}}\ \forall P\in\mathfrak{R}\ \forall z\in L\ \forall Q\in\mathfrak{Q}$

1) P ***safe by*** $\mathbf{\Sigma}$ ***after*** σ,
2) $\sigma\cdot\langle z\rangle\in\mathbf{\Sigma} \Rightarrow \exists P\in\mathfrak{R}\cup\mathfrak{Q}\ z\in P \ \& \ P$ ***safe by*** $\mathbf{\Sigma}$ ***after*** σ,
3) Q ***safe by*** $\mathbf{\Sigma}$ ***after*** $\sigma \Rightarrow \exists v\in Q\ \sigma\cdot\langle v\rangle\in\mathbf{\Sigma}$.

Безопасной трассой реализации будем называть её $\mathfrak{R}$-трассу, в которой каждое встречающееся внешнее действие z разрешается хотя бы одной кнопкой, безопасной в реализации после префикса трассы, непосредственно предшествующего этому действию:

SafeIn$(\mathbf{I})$ $=_{\text{def}}$ $\{\sigma\in\mathbf{I}_{\mathfrak{R}} \mid \forall\mu\ \forall z\in L$
$(\mu\cdot\langle z\rangle\leq\sigma \Rightarrow \exists P\in\mathfrak{R}\cup\mathfrak{Q}\ P$ ***safe in*** $\mathbf{I}$ ***after*** $\mu \ \& \ z\in P\})$.

Гипотеза о безопасности требует: если $\mathfrak{R}$-трасса σ спецификации безопасна в реализации, то любая кнопка, безопасная после σ в спецификации, безопасна после σ в реализации:

$\mathbf{I}$ ***safe for*** $\mathbf{\Sigma}$ $=_{\text{def}}$ $\forall\sigma\in\mathbf{\Sigma}_{\mathfrak{R}}\cap$***SafeIn***$(\mathbf{I})$ $\forall P\in\mathfrak{R}\cup\mathfrak{Q}$
$(P$ ***safe by*** $\mathbf{\Sigma}$ ***after*** $\sigma \Rightarrow P$ ***safe in*** $\mathbf{I}$ ***after*** $\sigma)$.

Тестирование ведётся «до первой ошибки»: прекращается сразу после обнаружения неконформности. Теперь продолжение тестирования после получения запрещённой трассы не только бессмысленно (не влияет на вердикт), но и небезопасно, поскольку гипотеза о безопасности уже не даёт никаких гарантий.

Обозначим внешние действия и отказы, наблюдаемые в модели T после трассы σ при нажатии кнопки P: ***obs***(σ,P,T) $=_{\text{def}}$ $\{z \mid \sigma\cdot\langle z\rangle\in T \ \& \ (z\in P \vee z=P \ \& \ P\in\mathfrak{R})\}$.

Будем говорить, что реализация *безопасно конформна* спецификации, если она безопасно-тестируема и любое наблюдение в ответ на нажатие безопасной кнопки, разрешается спецификацией. Такую конформность будем обозначать ***saco*** (***SA****fe* ***CO****nformance*).

$\mathbf{I}$ ***saco*** $\mathbf{\Sigma}$ $=_{\text{def}}$ $\mathbf{I}$ ***safe for*** $\mathbf{\Sigma}$ $\&$ $\forall\sigma\in\mathbf{\Sigma}\cap$***SafeIn***$(\mathbf{I})$ $\forall P$ ***safe by*** $\mathbf{\Sigma}$ ***after*** σ
obs$(\sigma,P,\mathbf{I})\subseteq$***obs***$(\sigma,P,\mathbf{\Sigma})$.

Генерация полного набора тестов для отношения ***saco*** сводится к тому, чтобы после каждой трассы σ, имеющейся в спецификации и наблюдаемой в реализации, нажать каждую кнопку “P”, безопасную после этой трассы. Если после этого наблюдается действие $z\in P$ или отказ $P\in\mathfrak{R}$, продолжающие трассу σ в спецификации, то тестирование либо заканчивается с вердиктом ***pass***, либо продолжается нажатием очередной кнопки. В противном случае тест выносит вердикт ***fail***.

Все определения безопасности и конформности, сделанные выше, применяются к LTS-моделям с помощью взятия их ***F***-моделей. Например:

$\mathbf{I}$ ***saco*** $\mathbf{S}$ $=_{\text{def}}$ ***Ftraces***$(\mathbf{I})$ ***saco*** ***Ftraces***$(\mathbf{S})$.

7. Пополнение спецификаций

Теперь обратим внимание на различие в определении безопасных кнопок для реализации и спецификации: ***safe in*** и ***safe by***. В реализации $\mathfrak{Q}$-кнопка “P” безопасна после трассы, если трасса не продолжается отказом P, а в спецификации – если трасса продолжается хотя бы одним действием $z \in P$ (да и то не обязательно, если есть другие безопасные кнопки, разрешающие все действия, которые продолжают трассу в спецификации и разрешаются кнопкой “P”). В первом случае трасса не может продолжаться отказом P, а во втором случае – должна продолжаться действием $z \in P$, но может продолжаться и отказом P. Это означает, что мы игнорируем $\mathfrak{Q}$-отказы в спецификации после трассы, которая продолжается действиями из этого отказа. Фактически, во всех определениях мы используем для реализации $\mathfrak{R} \cup \mathfrak{Q}$-проекцию $\mathbf{I}_{\mathfrak{R} \cup \mathfrak{Q}}$, а для спецификации $\mathfrak{R}$-проекцию $\mathbf{\Sigma}_{\mathfrak{R}}$. Это различие является причиной серьёзных проблем: *нерефлексивности* и *немонотонности* отношения конформности.

Сначала рассмотрим первую проблему: одна и та же модель, рассматриваемая как реализация, может быть не безопасно-тестируема и, следовательно, неконформна самой себе, рассматриваемой как спецификация. Более того, в спецификации могут быть $\mathfrak{R}$-трассы, которых не может быть ни в какой конформной реализации. Пример приведён на Рис 1 для LTS ввода-вывода. Здесь в LTS-спецификации S кнопка “?x” безопасна после трассы $\langle ?x \rangle$ и, следовательно, после трассы $\langle \delta, ?x, \delta \rangle$ (отказ не меняет состояния LTS). Поэтому, если в реализации есть трасса $\langle \delta, ?x, \delta \rangle$, то в ней есть и трасса $\langle \delta, ?x, \delta, ?x \rangle$. Поскольку приём реакций безопасен, должна наблюдаться хотя бы одна из трасс $\langle \delta, ?x, \delta, ?x, !a \rangle$, $\langle \delta, ?x, \delta, ?x, !b \rangle$ или $\langle \delta, ?x, \delta, ?x, \delta \rangle$. Но тогда в реализации есть хотя бы одна из трасс $\langle ?x, \delta, ?x, !a \rangle$, $\langle \delta, ?x, ?x, !b \rangle$ или $\langle ?x, ?x, \delta \rangle$. Каждая из этих трасс является продолжением безопасной трассы спецификации наблюдением, порождаемым безопасной кнопкой “δ”, но отсутствующим в спецификации, что противоречит конформности.

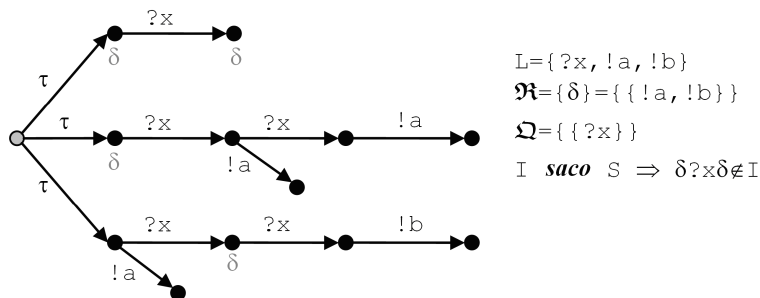

Рис 1.

Нерефлексивность конформности прямо противоречит интуиции: если реализацию «списать» со спецификации, то мы получим заведомо ошибочную реализацию. Заметим, что, если $\mathfrak{Q}=\varnothing$, то ***safe in*** = ***safe by***, и отношение ***saco*** рефлексивно (и транзитивно, то есть является предпорядком) (**!**). Это наводит на мысль перейти от $\mathfrak{R}/\mathfrak{Q}$-семантики к $\mathfrak{R} \cup \mathfrak{Q}/\varnothing$-семантике, когда все отказы наблюдаемы. Переход делается с помощью преобразования спецификации $\mathcal{C}$, которое называется *пополнением*. Его можно определить как для трассовой, так и для LTS-модели. В обоих случаях должна сохраняться конформность, то есть множество конформных реализаций. Для трассовой

модели: $\forall$I I $\boldsymbol{saco}_{\mathfrak{R}/\mathfrak{Q}}$ $\Sigma \Leftrightarrow$ I $\boldsymbol{saco}_{\mathfrak{R}\cup\mathfrak{Q}/\varnothing}$ $\mathcal{C}(\Sigma)$. Здесь в нижнем индексе мы указали, для какой тестовой семантики рассматривается отношение конформности. LTS-пополнение определяется аналогично, но, поскольку одной трассовой модели может соответствовать несколько LTS с тем же множеством трасс, оно не единственно.

В пополненной спецификации $\mathcal{C}(\Sigma)$ может появиться новая трасса σ, если она допустима в конформной реализации. Необходимое условие этого: во множестве $\boldsymbol{D}(\sigma)$ подтрасс, получаемых из σ удалением некоторых (в частности, всех) отказов, найдётся подтрасса $\sigma`\in\Sigma$. В $\mathcal{C}(\Sigma)$ трасса $\sigma$ продолжается внешним действием z, если после каждой подтрассы $\sigma`\in\boldsymbol{D}(\sigma)\cap\Sigma$ действие z либо не разрешается безопасными в Σ кнопками, либо разрешается и тогда $\sigma`\cdot\langle z\rangle\in\Sigma$. Также в $\mathcal{C}(\Sigma)$ трасса $\sigma$ продолжается $\mathfrak{R}$-отказом R, если им продолжается в $\Sigma$ каждая подтрасса $\sigma`\in\boldsymbol{D}(\sigma)\cap\Sigma$. Наконец, в $\mathcal{C}(\Sigma)$ трасса σ продолжается $\mathfrak{Q}$-отказом P, если для любой подтрассы $\sigma`\in\boldsymbol{D}(\sigma)\cap\Sigma$ и любой кнопки “R”, безопасной в $\Sigma$ после $\sigma`$, либо существует такое действие $z\in R\backslash P$, что $\sigma`\cdot\langle z\rangle\in\Sigma$, либо $R\in\mathfrak{R}$ и $\sigma`\cdot\langle R\rangle\in\Sigma$.

Такого рода пополнение аналогично, так называемому, *демоническому пополнению* для систем ввода-вывода без наблюдаемых блокировок стимулов [JJTV99,BRT03,BRT03r]. Отличие в том, что мы в пополненной части модели допускаем любые отказы, в том числе (ненаблюдаемые) блокировки. Кроме того, в [BRT03,BRT03r] предлагается LTS-преобразование, которое не полностью сохраняет конформность ***ioco*** (ослабляет её). В [JJTV99] предлагается трассовое преобразование, сохраняющее конформность, но этой конформностью является не ***ioco***, а ***ioconf***. Эти и некоторые другие отношения конформности, встречающиеся в литературе, мы рассмотрим ниже. Обзор различных видов пополнений можно найти в [BP94, LY96].

Заметим, что пополненную спецификацию уже нельзя рассматривать в $\mathfrak{R}/\mathfrak{Q}$-семантике, поскольку в ней она не эквивалентна исходной спецификации (Рис 2).

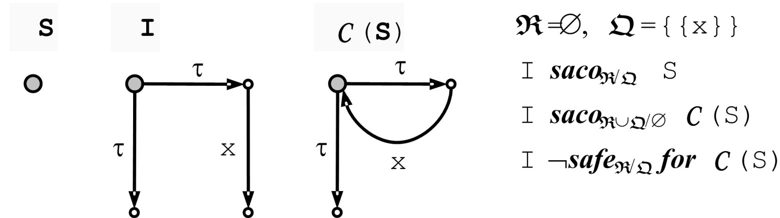

Рис 2.

Для доказательства существования трассового пополнения достаточно взять объединение всех конформных трассовых реализаций: $\mathcal{C}(\Sigma)=\cup(\{\mathbf{I}|\mathbf{I}\ \boldsymbol{saco}\ \Sigma\})$ (**!**). Для LTS-пополнения используется объединение LTS: добавляется новое начальное состояние, из которого проводятся τ-переходы в начальные состояния объединяемых LTS. Соответствующие трассовые модели объединяются в теоретико-множественном смысле: $\boldsymbol{Ftraces}\circ\mathcal{MC}(\mathbf{S})=\cup(\{\mathbf{I}|\mathbf{I}\ \boldsymbol{saco}\ \boldsymbol{Ftraces}(\mathbf{S})\})$. Конечно, это преобразование не алгоритмизуемо, но существует алгоритмизуемое пополнение (**!**).

8. Монотонность конформности

Вторая проблема – это, так называемая, *немонотонность конформности*, то есть несохранение конформности при композиции LTS: композиция реализаций, конформных своим спецификациям, оказывается неконформной композиции этих спецификаций. Эта проблема не может быть поставлена в теории $\mathfrak{R}$-трасс, поскольку там не определена

композиция трассовых моделей. Проблема монотонности носит более общий характер, чем проблема пополнения, и остаётся даже в том случае, когда нет $\mathfrak{Q}$-кнопок (**!**).

Решением проблемы является *монотонное* LTS-преобразование $\mathcal{M}$, сохраняющее конформность реализаций: композиция реализаций, конформных своим *преобразованным* спецификациям, конформна композиции этих *преобразованных* спецификаций. Естественно, композиция LTS-пополнения и монотонного LTS-преобразования $\mathcal{MC}$ является LTS-пополнением, которое можно назвать *монотонным пополнением*. Объединение конформных LTS-реализаций оказывается именно таким пополнением (**!**). Существует и алгоритмизуемое монотонное преобразование (**!**).

9. Примеры тестовых семантик и отношений конформности

Рассмотрим несколько встречающихся в литературе тестовых семантик (вместе с соответствующими отношениями конформности) и интерпретируем их как $\mathfrak{R}/\mathfrak{Q}$-семантики. Обзоры этих семантик и конформностей можно найти у Ван Глаббека [Glab90,Glab93] и Тритманса [Tret96]. Для каждой из этих семантик, интерпретируемой как $\mathfrak{R}/\mathfrak{Q}$-семантика, однозначно выделяется одна гипотеза о безопасности, поскольку в них каждое действие z, не разрешаемое $\mathfrak{R}$-кнопками, разрешается одной $\mathfrak{Q}$-кнопкой. Все такие действия, продолжающие трассу спецификации, определяют все $\mathfrak{Q}$-кнопки, которые должны быть объявлены безопасными после этой трассы.

<u>Трассовая семантика</u> (*trace semantics*). Задаётся $\mathfrak{R}/\mathfrak{Q}$-машиной с $\mathfrak{R}=\varnothing$ и $\mathfrak{Q}=\{L\}$. Трассы содержат только внешние действия. Обычно трассовый предпорядок (***tr*** – *trace preorder*) означает простую вложенность трасс. Однако мы трактуем остановку с ненаблюдаемым отказом как ошибку взаимодействия, и рассматриваем только безопасное тестирование. Гипотеза о безопасности требует: если наблюдаемая трасса в спецификации продолжается внешним действием, то она не может заканчиваться в терминальных состояниях LTS-реализации. Оператору запрещено нажимать кнопку “L” только после трассы, заканчивающейся в LTS-спецификации только в терминальных состояниях.

<u>Семантика завершённых трасс</u> (*completed trace semantics*). Задаётся $\mathfrak{R}/\mathfrak{Q}$-машиной с $\mathfrak{R}=\{L\}$ и $\mathfrak{Q}=\varnothing$. Единственный отказ L наблюдаем в терминальном состоянии реализации, когда никакое другое продолжение невозможно. Трассы содержат внешние действия и отказ L, но после него может следовать только тот же отказ L. Любая реализация безопасно-тестируема. Соответствующая конформность называется тестовым предпорядком (***te*** – *testing preorder*).

<u>Семантика трасс с отказами</u> (*failure trace semantics* или *refusal semantics*). Задаётся ***F***-машиной: $\mathfrak{R}=\mathcal{P}(L)$ и $\mathfrak{Q}=\varnothing$. Трассы содержат внешние действия и произвольные отказы (***Ftraces***). Любая реализация безопасно-тестируема. Соответствующая конформность называется предпорядком трасс с отказами (*failure trace preorder* или ***rf*** – *refusal preorder*). Встречается также разновидность этой семантики, когда разрешено может быть только *конечное* множество внешних действий. Этому соответствует $\mathfrak{R}/\mathfrak{Q}$-машина, в которой $\mathfrak{R}$ – семейство всех *конечных* подмножеств алфавита L.

Мы не останавливаемся на соответствующих эквивалентностях (*trace*, *completed trace* и *failure trace* или *refusal equivalence*), поскольку они не отвечают принципу независимости поведения реализации.

Для систем ввода-вывода имеется ряд специфических семантик. Все они основаны на следующих ограничениях: 1) окружение не может выбирать, какую реакцию ему

принимать, а какую нет: если реакции принимаются, то принимается любая реакция; 2) имеется возможность послать один стимул, не посылая других стимулов.

Семантика для систем ввода-вывода без блокировки стимулов. Задаётся $\mathfrak{R}/\mathfrak{Q}$-машиной с $\mathfrak{R}=\{\delta\}=\{\{!y\}|!y\in L\}$ и $\mathfrak{Q}=\{\{?x\}|?x\in L\}$. Трассы содержат стимулы, реакции и единственный отказ – стационарность δ. Отношение ***ior*** (*input-output refusal relation* или *repetitive quiescence relation*), означает простую вложенность трасс $\mathbf{I}_{\mathfrak{R}}\subseteq\mathbf{\Sigma}_{\mathfrak{R}}$. Для того, чтобы тестирование было безопасным, любая трасса $\sigma\in\mathbf{I}_{\mathfrak{R}}\cap\mathbf{\Sigma}_{\mathfrak{R}}$ должна в LTS-реализации заканчиваться только в таких стабильных состояниях, в которых определены переходы по всем стимулам. Однако, если такая трасса в спецификации не продолжается стимулом ?x, то реализация заведомо неконформна. Из-за этого спецификация, содержащая, например, только трассу $\langle ?x,!y\rangle$ и все её префиксы, не имеет конформных реализаций.

Отмеченная неудовлетворительность отношения ***ior*** преодолевается в отношении ***ioco*** (*input-output conformance*), предложенном Тритмансом [Tret92,Tret96]. Это отношение опирается на гипотезу о всюду-определённости реализации по стимулам (*input enabledness*): в каждом достижимом стабильном состоянии LTS-реализации принимаются все стимулы. Однако, на самом деле, при тестировании стимул ?x не посылается в реализацию после трассы σ, которая в спецификации не продолжается стимулом ?x, то есть для конформности по ***ioco*** безразлично поведение реализации после приёма такого стимула. Если $\mathbf{I}$ ***ioco*** $\mathbf{\Sigma}$ и $\sigma\cdot\langle ?x\rangle\in\mathbf{I}$, то реализация $\mathbf{I}$`, отличающаяся от $\mathbf{I}$ только тем, что она блокирует стимул ?x после трассы σ, также безопасно-тестируема (не возникает ненаблюдаемых отказов). Эти две реализации обе безопасно-тестируемы и не различимы при безопасном тестировании. Тем не менее, реализация $\mathbf{I}$ конформна, а реализация $\mathbf{I}$` неконформна, поскольку не входит в домен отношения ***ioco***. Иными словами, гипотеза всюду-определённости реализации по стимулам неоправданно сильная. Наша гипотеза о безопасности более точно выражает предусловие тестирования реализации: если трасса σ в спецификации продолжается стимулом ?x, то в реализации трасса σ не должна заканчиваться в состоянии, где стимул блокируется. Для отношения ***saco*** не может быть двух реализаций, неразличимых при безопасном тестировании, одна из которых конформна, а другая нет.

Семантика для систем ввода-вывода с блокировками стимулов. Задаётся $\mathfrak{R}/\mathfrak{Q}$-машиной с $\mathfrak{R}=\{\delta\}\cup\{\{?x\}|?x\in L\}$ и $\mathfrak{Q}=\varnothing$. Трассы содержат стимулы, реакции и отказы: стационарность δ и блокировки стимулов. Любая реализация безопасно-тестируема. Соответствующая конформность типа ***saco*** называется ***ioco***$_{\beta\delta}$ [BK05-1,BK05-2,BKK06,BKK07]. Именно к этой семантике нужно переходить при пополнении спецификации в семантике без блокировок стимулов.

Семантика для систем с мультиканалами стимулов и реакций (*MIOTS – Multi Input-Output Transition Systems*). Такие системы рассматриваются в работах Херинка и Тритманса, где предлагается отношение конформности ***mioco*** [HT97,Heer98]. Множество стимулов разбивается на подмножества – входные каналы L_I, и для каждого входного канала L_I реализация должна принимать либо любой стимул из L_I, либо ни одного. Множество реакций также разбивается на подмножества – выходные каналы L_U, но в отличие от стимулов не налагается ограничений на выдачу реакций реализацией. Каждому каналу соответствует своё θ-наблюдение. Оно означает: для входного канала – отказ по всем стимулам этого канала, для выходного канала – отсутствие реакций этого канала («частичная» стационарность). Заметим, что здесь тест уже не обязан принимать все или ни одной реакции, но обязан принимать все или ни одной реакции из каждого

выходного канала. В $\mathfrak{R}/\mathfrak{Q}$-машине каждому стимулу `?x` соответствует $\mathfrak{R}$-кнопка `{?x}`, каждому выходному каналу L_U – $\mathfrak{R}$-кнопка L_U, $\mathfrak{Q}$-кнопок нет. Указанное выше ограничение на приём стимулов является ограничением на реализацию (и спецификацию) и не затрагивает устройства машины. Любая реализация безопасно-тестируема.

Отношение ***mioco*** почти совпадает с отношением ***saco*** для такой машины за одним исключением. Отношение ***mioco*** разрешает реализации всегда принимать стимул независимо от того, принимается стимул в спецификации или только блокируется. По-видимому, такая «либеральность» объясняется происхождением ***mioco*** от ***ioco***, в которой реализация считается всюду определённой по стимулам. Для ***mioco*** это уже не так – реализация может блокировать стимул, если это разрешает спецификация, но по-прежнему нельзя потребовать, чтобы реализация не принимала стимул, то есть только блокировала его.

Иногда в системах ввода-вывода на окружение налагают ограничение: оно не должно «тормозить реакции», выдаваемые реализацией [PYH03,LG05]. В терминах LTS это значит, что в каждом стабильном состоянии окружения/теста должны быть определены переходы по всем реакциям.

Семантика для систем ввода-вывода без «торможения реакций» и без блокировок. Задаётся $\mathfrak{R}/\mathfrak{Q}$-машиной с $\mathfrak{R}=\{\delta\}$ и $\mathfrak{Q}=\{\{\delta,?x\}\,|\,?x\in L\}$. В отличие от аналогичной семантики с торможением реакций гипотеза о безопасности ослабляется: если трасса σ в спецификации продолжается стимулом `?x` *или реакциями*, то в реализации трасса σ не должна заканчиваться в *стационарном* состоянии, где этот стимул `?x` блокируется. Посылая стимул в безопасно-тестируемую реализацию, мы должны быть готовы к тому, что будем наблюдать не приём стимула, а выдачу реакции. Если в реализации есть бесконечная цепочка выдачи реакций (*осцилляция* [PYH03]), то мы не сможем послать стимул `?x` с гарантированным приёмом его реализацией. Более того, если в этой цепочке нет состояний с приёмом стимула `?x`, мы об этом никогда не узнаем, непрерывно нажимая на одну и ту же кнопку $\{\delta,?x\}$.

Семантика для систем ввода-вывода без «торможения реакций» и с блокировками. Задаётся $\mathfrak{R}/\mathfrak{Q}$-машиной с $\mathfrak{R}=\{\delta\}\cup\{\{\delta,?x\}\,|\,?x\in L\}$ и $\mathfrak{Q}=\varnothing$. Любая реализация безопасно-тестируема. Блокировка стимула наблюдается только одновременно со стационарностью (в стационарных состояниях LTS-реализации). Проблема осцилляции здесь также имеет место: мы не сможем ничего узнать о блокировках стимулов в нестационарных состояниях.

Для большинства описанных выше семантик с $\mathfrak{R}$-отказами изучались разновидности, когда любой отказ или отказ определённого вида может наблюдаться только в конце трассы. После наблюдения такого отказа дальнейшие наблюдения невозможны. В $\mathfrak{R}/\mathfrak{Q}$-машине (как и в реактивной машине) этому могла бы соответствовать блокировка клавиатуры при наблюдении отказа. В генеративной машине Ван Глаббек предлагает считать, что переключатели нельзя перебрасывать из положения *blocked* в положение *free*, что в общем эквивалентно блокировке клавиатуры по мощности тестирования. Для семантики завершённых трасс ничего не меняется, поскольку отказ `L` в любом случае означает, что никаких действий больше не будет. Для остальных семантик отношения конформности меняются:

rf $\rightarrow$ ***conf*** или *failure preorder* для трасс внешних действий
и пар <трасса внешних действий, отказ> (*failure pairs*),

ior $\rightarrow$ ***iot*** (*input-output testing relation*) для трасс внешних действий
и трасс внешних действий, заканчивающихся стационарностью (*quiescent traces*),

ioco $\rightarrow$ ***ioconf*** (с той же расшифровкой *input-output conformance*).

Отношение ***rioco*** в [LG05] с самого начала ориентировано на то, что блокировки стимулов (в отличие от стационарности) располагаются только в конце трассы.

Все эти разновидности только уменьшают мощность тестирования, но не вполне понятны причины, по которым на практике наблюдение отказа не даёт возможность продолжения тестирования. Далее мы не будем на этом останавливаться.

10. Разрушение

Мы вводим новый вид действия машины, которое называем *разрушением* и обозначать символом γ. Под γ-действием понимается любое нежелательное поведение системы, в том числе и реальное разрушение системы, которого нельзя допускать при тестировании (например, кнопка немедленного саморазрушения для систем оборонного назначения). Разрушение, как и отказ, является одним из способов интерпретации неспецифицированного поведения. Отличие в том, что отказ предполагает «защиту» системы от нежелательных обращений, а разрушение ничего не гарантирует.

Исключение из рассмотрения обращений, разрушающих реализацию, часто мотивируют тем, что реализация должна проверять корректность параметров обращения [Tret96,Heer98]. Если параметры некорректны, реализация либо игнорирует обращение, либо сообщает об ошибке. Такое требование к реализации естественно, если это «система общего пользования»: в ней должна быть предусмотрена «защита от дурака». Однако часто требуется тестировать внутренние компоненты или подсистемы, доступ к которым строго ограничен и взаимные проверки корректности обращений излишни. Такое часто встречается для обращений с параметрами сложной внутренней структурой и нетривиальными условиями корректности, когда накладные расходы на проверку неоправданно увеличивают трудозатраты на создание системы, её объём и время выполнения. Альтернативой в этом случае является строгая спецификация предусловий вызова операций [Hoare69]. Например, вызов операции освобождения участка памяти, который не был ранее получен через операцию запроса памяти, является нарушением предусловия. Тестированию подлежит не поведение компонентов в ответ на некорректные обращения от других компонентов, а правильность обращения компонентов друг к другу. Последнее означает, фактически, проверку поведения каждого компонента (по его постусловию) только для таких обращений, которые удовлетворяют предусловию компонента. Иными словами, поскольку мы хотим убедиться в правильности реализации, а не окружения, нас не интересует поведение реализации при вызове её операций с нарушением предусловий, и такое поведение не регламентируется спецификацией.

Семантика γ-действия предполагает, что после него любые наблюдения недостоверны. Поскольку нас интересуют только достоверные наблюдения, будем считать, что после разрушения никакие наблюдения невозможны. Разрушение считается условно-наблюдаемым действием: как событие оно может возникать при взаимодействии, но мы ограничимся только таким тестированием, при котором этого не бывает. В этом смысле разрушение аналогично ненаблюдаемому отказу. Тем самым, нас интересует только безопасное взаимодействие окружения с реализацией, не приводящее к разрушению.

Разрушение, как и τ-действия, не управляется кнопками: оно всегда разрешено. В безопасном тестировании оператор не должен нажимать кнопку, если это может привести к выполнению внешнего действия, после которого реализация может разрушиться. Тем самым, γ-действие возникает только после внешнего действия или в начале работы. Последний случай вырожденный: реализация конформна только такой спецификации, которая разрешает разрушение в самом начале, но тестирование излишне и недопустимо (машину нельзя «включать»). В LTS-модели переход может быть помечен не только внешним действием или символом τ, но и символом γ. При композиции LTS добавляются два новых правила:

```
s—γ→s`  ⊢ st—γ→s`t;
```

$t \text{—}\gamma\text{→} t` \quad \vdash st\text{—}\gamma\text{→} st`$.

У нас появляется новый вид трасс – трассы, заканчивающиеся разрушением. Теперь в реализации кнопка безопасна после трассы $\sigma \in \mathbf{I}_{\mathfrak{R}}$, если она не только не вызывает ненаблюдаемого отказа, но и не приводит к разрушению (условие, подчёркнутое волнистой линией): P ***safe in*** $\mathbf{I}$ ***after*** $\sigma =_{def} (P \in \mathfrak{R} \vee \sigma \cdot \langle P \rangle \notin \mathbf{I})$ $\underline{\& \ \forall z \in P \ \sigma \cdot \langle z, \gamma \rangle \notin \mathbf{I}}$.

Соответствующим образом модифицируются правила протокола взаимодействия (добавляются условия, подчёркнутые волнистой линией): $\forall \sigma \in \underline{\mathbf{\Sigma}}_{\mathfrak{R}} \ \forall R \in \mathfrak{R} \ \forall z \in L \ \forall Q \in \mathfrak{Q}$

1) R ***safe by*** $\mathbf{\Sigma}$ ***after*** σ $\underline{\Leftrightarrow \ \forall u \in R \ \sigma \cdot \langle u, \gamma \rangle \notin \mathbf{\Sigma}}$,
2) $\sigma \cdot \langle z \rangle \in \mathbf{\Sigma}$ $\underline{\& \ \exists T \in \mathfrak{R} \cup \mathfrak{Q} \ z \in T \ \& \ \forall u \in T \ \sigma \cdot \langle u, \gamma \rangle \notin \mathbf{\Sigma}}$

   $\Rightarrow \exists P \in \mathfrak{R} \cup \mathfrak{Q} \ z \in P \ \& \ P$ ***safe by*** $\mathbf{\Sigma}$ ***after*** σ,
3) Q ***safe by*** $\mathbf{\Sigma}$ ***after*** $\sigma \Rightarrow \exists v \in Q \ \sigma \cdot \langle v \rangle \in \mathbf{\Sigma}$ $\underline{\& \ \forall u \in Q \ \sigma \cdot \langle u, \gamma \rangle \notin \mathbf{\Sigma}}$.

Определение безопасной трассы реализации изменяется: 1) оно использует отношение ***safe in***, учитывающее разрушение, 2) в безопасной трассе не должно быть опасных $\mathfrak{R}$-отказов. Также появляется понятие безопасной трассы спецификации. Раньше в нём не было нужды, поскольку безопасность понималась как отсутствие ненаблюдаемых отказов, а второе правило протокола гарантировало для каждого внешнего действия в трассе наличие кнопки, разрешающей это действие и не вызывающей ненаблюдаемого отказа после предшествующего префикса трассы. Теперь мы должны учитывать возможность разрушения. Безопасных трасс нет, если есть трасса $\langle \gamma \rangle$.

$$\textbf{\textit{SafeIn}}(\mathbf{I}) =_{def} \{\sigma \in \mathbf{I}_{\mathfrak{R}} \mid \langle \gamma \rangle \notin \mathbf{I} \ \& \ \forall \mu \ \forall u$$
$$(\mu \cdot \langle u \rangle \leq \sigma \Rightarrow \exists P \in \mathfrak{R} \cup \mathfrak{Q} \ P \ \textbf{\textit{safe in}} \ \mathbf{I} \ \textbf{\textit{after}} \ \mu \ \& \ (u \in P \vee u = P)\}).$$

$$\textbf{\textit{SafeBy}}(\mathbf{\Sigma}) =_{def} \{\sigma \in \mathbf{\Sigma}_{\mathfrak{R}} \mid \langle \gamma \rangle \notin \mathbf{\Sigma} \ \& \ \forall \mu \ \forall u$$
$$(\mu \cdot \langle u \rangle \leq \sigma \Rightarrow \exists P \in \mathfrak{R} \cup \mathfrak{Q} \ P \ \textbf{\textit{safe by}} \ \mathbf{\Sigma} \ \textbf{\textit{after}} \ \mu \ \& \ (u \in P \vee u = P)\}).$$

Гипотеза о безопасности теперь требует отсутствия в реализации разрушения, если оно невозможно в спецификации в той же ситуации, то есть либо с самого начала (до нажатия кнопок), либо после безопасного продолжения безопасной трассы.

$$\mathbf{I} \ \textbf{\textit{safe for}} \ \mathbf{\Sigma} =_{def} (\gamma \notin \mathbf{\Sigma} \Rightarrow \gamma \notin \mathbf{I}) \ \& \ \forall \sigma \in \textbf{\textit{SafeBy}}(\mathbf{\Sigma}) \cap \textbf{\textit{SafeIn}}(\mathbf{I}) \ \forall P \in \mathfrak{R} \cup \mathfrak{Q}$$
$$(P \ \textbf{\textit{safe by}} \ \mathbf{\Sigma} \ \textbf{\textit{after}} \ \sigma \Rightarrow P \ \textbf{\textit{safe in}} \ \mathbf{I} \ \textbf{\textit{after}} \ \sigma).$$

Отношение конформности строится только на безопасных $\mathfrak{R}$-трассах спецификации:

$$\mathbf{I} \ \textbf{\textit{saco}} \ \mathbf{\Sigma} =_{def} \mathbf{I} \ \textbf{\textit{safe for}} \ \mathbf{\Sigma} \ \& \ \forall \sigma \in \textbf{\textit{SafeBy}}(\mathbf{\Sigma}) \cap \textbf{\textit{SafeIn}}(\mathbf{I}) \ \forall P \ \textbf{\textit{safe by}} \ \mathbf{\Sigma} \ \textbf{\textit{after}} \ \sigma$$
$$\textbf{\textit{obs}}(\sigma, P, \mathbf{I}) \subseteq \textbf{\textit{obs}}(\sigma, P, \mathbf{\Sigma}).$$

Утверждения о рефлексивности и транзитивности отношения конформности при отсутствии $\mathfrak{Q}$-кнопок и о существовании (в том числе, алгоритмизуемых) пополнения $\mathcal{C}$ и монотонного преобразования $\mathcal{M}$ остаются в силе и для γ-семантики.

$\mathbf{I} \ \textbf{\textit{saco}}_{\mathfrak{R}/\mathfrak{Q}} \ \mathbf{S} \Leftrightarrow \mathbf{I} \ \textbf{\textit{saco}}_{\mathfrak{R} \cup \mathfrak{Q}/\varnothing} \ \mathcal{C}(\mathbf{S}) \Leftrightarrow \mathbf{I} \ \textbf{\textit{saco}}_{\mathfrak{R}/\mathfrak{Q}} \ \mathcal{C}(\mathbf{S})$ (**!**).

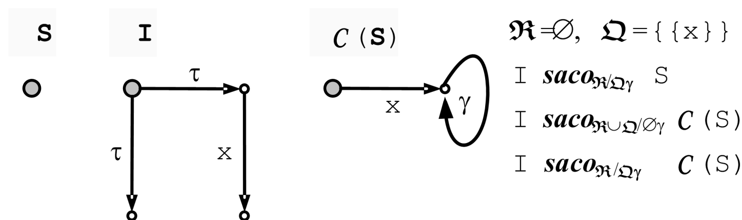

Рис 3.

Для γ-семантики существует такая пополненная спецификация, которую уже можно рассматривать не только в $\mathfrak{R}\cup\mathfrak{Q}$-семантике, но и в $\mathfrak{R}/\mathfrak{Q}$-семантике (ср. Рис 2 и Рис 3).

Заметим, что в пополнении C вместо «демонического» продолжения трассы σ всеми трассами вида $\sigma\cdot\langle z\rangle\cdot\lambda$, используется гамма-пополнение, продолжающее трассу σ только трассой $\sigma\cdot\langle z,\gamma\rangle$. Тем самым, не теряется информация об опасных кнопках: после демонического пополнения мы должны при тестировании проверять все добавленные трассы $\sigma\cdot\langle z\rangle\cdot\lambda$, хотя это и бессмысленно («всё разрешено»), а при гамма-пополнении добавленная трасса $\sigma\cdot\langle z\rangle$ не проверяется, поскольку ведёт к разрушению.

11. Дивергенция

Дивергенция (бесконечная внутренняя активность) – вовсе не обязательно ошибка «зацикливания» программы. В некоторых случаях дивергенция является правильным или неизбежным поведением. Например, тест может подменять собой не всё окружение, а только его часть. Фактически, тестируется композиция реализации с остальной частью окружения. Взаимодействие реализации с этой частью окружения может быть бесконечным, но из теста оно видимо как бесконечная внутренняя активность, то есть дивергенция (синхронные переходы композиции – это τ-переходы). Возможна и чисто «внутренняя» дивергенция, например, «ждущий тест» в операционной системе, который продолжается бесконечно долго, пока его не прервут.

На самом деле проблема не в дивергенции самой по себе, а в выходе из неё: если внешнее воздействие имеет больший приоритет, чем внутренняя активность, дивергенция прекращается. В машине с приоритетами выполнимость τ-действий зависит от нажатой кнопки, и мы можем косвенно управлять ими и, следовательно, дивергенцией. Тогда можно говорить о *выполнимой* дивергенции: при одной нажатой кнопке (или когда нет нажатой кнопки) все τ-действия бесконечной цепочки выполнимы, а при другой – нет и, следовательно, нет «зацикливания». Выйти из дивергенции, которая начинает выполняться после кнопки “A”, можно с помощью кнопки “B”, при которой дивергенция не выполнима. Заметим, что для этого требуется переключение кнопок, то есть нажатие кнопки без наблюдения (которого может не быть). Единственный случай, когда из дивергенции нельзя гарантированно выйти, – это когда дивергенция выполнима при нажатии любой кнопки. Для машины без приоритетов такой нежелательной дивергенцией является любая дивергенция.

Дивергенция «неудобна» тем, что не позволяет получить наблюдение в ответ на тестовое воздействие за конечное время и, тем самым, продолжить тестирование после наблюдения. В этом смысле дивергенция аналогична ненаблюдаемому отказу. Однако есть и существенное отличие: ненаблюдаемый отказ возникает вместо внешнего действия после нажатия кнопки, а дивергенция возникает после внешнего действия (или в начальном состоянии машины до нажатия первой кнопки) и проявляется только тогда, когда оператор ещё раз нажимает какую-либо кнопку. Иными словами, опасна не дивергенция, а выход из неё.

В машине без приоритетов поведение реализации при дивергенции предлагается моделировать специальным Δ-действием. Оно разрешается каждой кнопкой и считается условно-наблюдаемым: как событие дивергенция может возникать при взаимодействии, но мы ограничимся только таким тестированием, при котором этого не бывает. Такое действие очень похоже на γ-действие, и отличается только тем, что разрушение нельзя запретить, а Δ-действие разрешается каждой кнопкой, но запрещено, если кнопки не нажаты, что моделирует проявление дивергенции при попытке выхода из неё. Отношения безопасности меняются так (добавления отмечены волнистой линией):

P ***safe in*** $\mathbf{I}$ ***after*** $\sigma =_{def} (P\in\mathfrak{R} \vee \sigma\cdot\langle P\rangle\notin\mathbf{I})\ \&\ \forall z\in P\ \sigma\cdot\langle z,\gamma\rangle\notin\mathbf{I}\ \&\ \sigma\cdot\langle\Delta\rangle\notin\mathbf{I}$.

$\forall\sigma\in\mathbf{\Sigma}^{\downarrow}L_{\mathfrak{R}}\ \forall R\in\mathfrak{R}\ \forall z\in L\ \forall Q\in\mathfrak{Q}$

1) R ***safe by*** $\mathbf{\Sigma}$ ***after*** $\sigma \Leftrightarrow \forall u\in R\ \sigma\cdot\langle u,\gamma\rangle\notin\mathbf{\Sigma}\ \&\ \sigma\cdot\langle\Delta\rangle\notin\mathbf{\Sigma}$,

2) $\sigma\cdot\langle z\rangle\in\mathbf{\Sigma}\ \&\ \exists T\in\mathfrak{R}\cup\mathfrak{Q}\ z\in T\ \&\ \forall u\in T\ \sigma\cdot\langle u,\gamma\rangle\notin\mathbf{\Sigma}\ \&\ \sigma\cdot\langle\Delta\rangle\notin\mathbf{\Sigma}$
$\Rightarrow \exists P\in\mathfrak{R}\cup\mathfrak{Q}\ z\in P\ \&\ P$ ***safe by*** $\mathbf{\Sigma}$ ***after*** σ,

3) Q ***safe by*** $\mathbf{\Sigma}$ ***after*** $\sigma \Rightarrow \exists v\in Q\ \sigma\cdot\langle v\rangle\in\mathbf{\Sigma}\ \&\ \forall u\in Q\ \sigma\cdot\langle u,\gamma\rangle\notin\mathbf{\Sigma}\ \&\ \sigma\cdot\langle\Delta\rangle\notin\mathbf{\Sigma}$.

Гипотеза о безопасности теперь разрешает дивергенцию в реализации только в том случае, когда это разрешено в спецификации в той же самой ситуации, то есть после той же самой трассы, безопасной в спецификации.

При композиции может возникать дивергенция как бесконечная цепочка синхронных переходов даже в том случае, когда в операндах нет дивергенции. Это означает, что класс моделей без дивергенции не замкнут по композиции.

Иногда рассматривают тестирование, в котором возможно прямое (а не условное, как у нас) наблюдение дивергенции (Δ-наблюдение) или, так называемое, λ-наблюдение, означающее дивергенцию или ненаблюдаемый отказ. В обоих случаях предполагается либо возможность бесконечного наблюдения, либо ограничение сверху на время выполнения любого внешнего действия и любой конечной цепочки τ-наблюдений. Если экран пуст бесконечно долго или дольше тайм-аута после нажатия $\mathfrak{R}$-кнопки, то это означает дивергенцию, а после нажатия $\mathfrak{Q}$-кнопки – λ-наблюдение. Бесконечное наблюдение мы считаем практически нереальным и далее не рассматриваем. Если ограничение по времени используется как способ обнаружения отказа, то различие между $\mathfrak{R}$- и $\mathfrak{Q}$-кнопками стирается, и срабатывание тайм-аута в обоих случаях приходится понимать как λ-наблюдение. В дальнейшем мы будем предполагать, что дивергенция моделируется Δ-действием, а тестирование безопасно, то есть избегает ненаблюдаемых отказов, дивергенции и разрушения. Это основано на представлении о том, что при правильном взаимодействии не должно возникать тупиков, бесконечного ожидания окружением результата взаимодействия и разрушения системы.

12. Лампочки меню и трассы готовности

При тестировании могут быть дополнительные тестовые возможности, которые в машине задаются с помощью, так называемых, *лампочек меню*, соответствующих внешним действиям. Лампочка для действия z горит, когда действие z определено в машине. Если машина активна, состояние лампочек говорит о действиях, определённых, быть может, не в текущем, а в уже пройденном состоянии. Поэтому обычно считается, что состояние лампочек достоверно только тогда, когда машина стоит в стабильном состоянии s. Лампочки меню позволяют в момент остановки машины определить *множество готовности* (*ready set*)– множество всех внешних действий, определённых в состоянии s, то есть готовых к выполнению. Это множество обозначим через ***ready***(s). Трассы, в которых, кроме внешних действий, могут встречаться множества готовности, называются трассами готовности (*ready traces*), а соответствующие семантика, предпорядок или эквивалентность – семантикой, предпорядком или эквивалентностью трасс готовности (*ready trace semantics, ready trace preorder or equivalence*).

Трассы готовности обладают свойством *генеративности*: по трассе готовности мы можем получить все трассы отказов, которые могли бы наблюдаться при выполнении реализацией той же самой цепочки переходов. Иными словами, множество трасс готовности однозначно определяет множество трасс с отказами. Однако без лампочек меню мы не можем, наблюдая трассу с отказами, определить, является ли она

дополнением (заменой отказов на их дополнения до алфавита) трассы готовности или нет. При отказе нам известно, что все разрешённые действия не определены в реализации. Однако про запрещённые действия мы не знаем, определены они или нет. Кроме того, в параметризованной машине, вообще говоря, не все дополнения множеств готовности являются наблюдаемыми отказами (соответствуют $\mathfrak{R}$-кнопкам).

Как и для семантик с отказами, встречается модификация, когда наблюдение готовности не позволяет продолжать тестирование. Тогда говорят о парах <трасса внешних действий, множество готовности> (*ready pairs*), семантике, предпорядке или эквивалентности готовности (*readiness semantics, ready preorder or equivalence*). Аналогично машине тестирования с конечными отказами, рассматривается машина, в которой можно узнать только о *конечных* множествах готовых к выполнению действий. Ван Глаббек предлагает понимать это так, что лампочка меню – это одновременно кнопка, и может гореть только, если её нажал оператор. Нажимая по одной лампочки-кнопки после остановки машины в стабильном состоянии (зелёная лампочка погасла), оператор узнаёт, определены или нет соответствующие действия. Однако, если алфавит действий бесконечен, не известно, является ли полученное множество определённых действий множеством готовности, то есть множеством *всех* определённых действий, или нет.

Тестовые возможности, моделируемые лампочками меню, увеличивают мощность тестирования, но их практичность вызывает сомнения. Наблюдаемость отказа означает, что окружение может узнать, выполнилось или нет какое-либо действие из тех, что оно потребовало от реализации, и, если выполнилось, то какое именно, и использовать это знание для коррекции дальнейшего взаимодействия. Иными словами, оно рассчитывает не только на уведомление о выполнении запрошенного действия, но и на уведомление об отказе в выполнении. Это довольно естественное поведение. Однако для определения множества готовности, окружение должно узнать, какие действия реализация в принципе *могла бы* выполнить в данном стабильном состоянии. Для этого, по-видимому, требуется специальная операция опроса, которая далеко не всегда имеется в интерфейсе реализации.

Правда, бывают и исключения: например, в графических интерфейсах такое меню определённых действий может появляться на экране, иногда в виде списка всех (или части) действий, в котором действиям, которые не могут выполняться, соответствуют «бледные кнопки». Но даже в этом примере действия в графическом меню – это только стимулы, а какие реакции реализация выдаст в ответ на нажатие той или иной кнопки из графического меню, вообще говоря, неизвестно. Иными словами, для этого примера нужно рассматривать машину тестирования, в которой лампочки меню имеются не для всех, а лишь для некоторых действий. Можно предложить более общую тестовую возможность *частичных* множеств готовности: в стабильном состоянии система сообщает статус каждого действия: 1) определено – лампочка горит зелёным цветом, 2) не определено – лампочка горит красным цветом, или 3) не известно – лампочка не горит. Далее мы будем рассматривать полные множества готовности, когда третьего случая нет.

Теперь рассмотрим вопрос о том, как окружение может использовать множества готовности. Наблюдение готовности происходит, когда машина стоит в стабильном состоянии. Это даёт окружению возможность вычислять любые отказы. Поэтому окружение может разрешать только такие множества действий, хотя бы одно из которых реализация может выполнить. При таком протоколе взаимодействия ненаблюдаемые отказы не возникают после того, как машина остановилась, хотя по-прежнему могут возникать, если кнопка нажимается после внешнего действия (или в начале работы). Отказы никаких дополнительных наблюдений не дают, поэтому и не рассматриваются смешанные трассы, в которых были бы и множества готовности и отказы. Естественно, мы по-прежнему должны уметь обнаруживать остановку машины, что делается с помощью θ-наблюдения в $\mathfrak{R}/\mathfrak{Q}$-машине после нажатия $\mathfrak{R}$-кнопки или с помощью зелёной лампочки в генеративной и реактивной машинах. При остановке в трассу помещается

наблюдаемое множество готовности. В $\mathfrak{R}/\mathfrak{Q}$-машине гипотеза о безопасности ослабляется: она касается поведения реализации только после таких трасс, которые не заканчиваются на множество готовности (пустая трасса и трасса, заканчивающаяся внешним действием). После такой трассы σ $\mathfrak{Q}$-кнопка “P” безопасна в спецификации $\boldsymbol{\Sigma}$, если она безопасна после любого продолжения этой трассы множеством готовности R, которое возможно в спецификации: $\forall R\ \sigma\cdot\langle R\rangle\in\boldsymbol{\Sigma} \Rightarrow R\cap P\neq\varnothing$.

В отличие от внешнего действия и отказа для множества готовности естественно считать, что конформность означает не равенство $R_i=R_s$ реализационного множества готовности R_i некоторому спецификационному множеству готовности R_s после той же самой трассы, а обратную вложенность $R_i\supseteq R_s$. Иными словами, если реализация после трассы может оказаться в состоянии, где определено некоторое множество действий R_i, то это должно быть разрешено спецификаций в том смысле, что в спецификации после этой трассы должно быть состояние, в котором определено не большее множество действий R_s. Это похоже на требование *обязательного* поведения: реализация не должна предлагать «меню» действий *меньше*, чем это разрешает спецификация.

Будем говорить, что трасса реализации *мажорируется* трассой спецификации, если они имеют одну длину и в соответствующих позициях стоят либо одинаковые внешние действия, либо вложенные множества готовности. При наличии разрушения нужно учитывать только неразрушающие внешние действия спецификации (разрушающие действия спецификации не проверяются и поэтому в реализации могут быть не определены). Для множеств трасс готовности по-прежнему применяется разрешительный принцип, но только трасса реализации разрешена не тогда, когда она сама есть во множестве разрешённых трасс спецификации, а когда там есть мажорирующая её трасса.

Такое отношение конформности можно назвать безопасной конформностью с трассами готовности и обозначить как ***resaco***. В $\mathfrak{R}/\mathfrak{Q}$-семантике для LTS оно связано с отношением ***saco*** через некоторое монотонное пополнение $\mathcal{MC}$:

$$I\ \boldsymbol{saco}_{\mathfrak{R}/\mathfrak{Q}}\ S \Leftrightarrow I\ \boldsymbol{saco}_{\mathfrak{R}\cup\mathfrak{Q}/\varnothing}\ \mathcal{MC}(S) \Leftrightarrow I\ \boldsymbol{saco}_{\mathfrak{R}/\mathfrak{Q}}\ \mathcal{MC}(S)$$

$$\Leftrightarrow I\ \boldsymbol{resaco}_{\mathfrak{R}\cup\mathfrak{Q}/\varnothing}\ \mathcal{MC}(S) \Leftrightarrow I\ \boldsymbol{resaco}_{\mathfrak{R}/\mathfrak{Q}}\ \mathcal{MC}(S).$$

Это утверждение пока что является гипотезой. Идея его доказательства опять могла бы заключаться в том, чтобы в качестве преобразования $\mathcal{MC}$ взять объединение конформных реализаций. Заметим, что эквивалентность $I\ \boldsymbol{resaco}_{\mathfrak{R}\cup\mathfrak{Q}/\varnothing}\ S` \Leftrightarrow I\ \boldsymbol{resaco}_{\mathfrak{R}/\mathfrak{Q}}\ S`$ верна только для спецификации $S`=\mathcal{MC}(S)$, а не для произвольной спецификации $S`$.

Множества готовности обладают важным свойством *вычислимости* при композиции LTS: стабильность и множество готовности композиционного состояния st однозначно определяются стабильностью и множествами готовности состояний-перандов s и t. При композиции LTS в алфавитах L и M имеем:

$$st\ \boldsymbol{стабильно} \Leftrightarrow s\ \boldsymbol{стабильно}\ \&\ \boldsymbol{ready}(s)\cap\underline{\boldsymbol{ready}(t)}=\varnothing$$
$$\&\ t\ \boldsymbol{стабильно}\ \&\ \underline{\boldsymbol{ready}(s)}\cap\boldsymbol{ready}(t)=\varnothing,$$
$$st\ \boldsymbol{стабильно}\ :\ \boldsymbol{ready}(st)=(\boldsymbol{ready}(s)\setminus\underline{M})\cup(\boldsymbol{ready}(t)\setminus\underline{L}).$$

Это даёт возможность определить операцию композиции трасс готовности, которая по паре трасс λ и μ возвращает множество трасс. Это множество состоит из всех трасс, в которых подтрасса асинхронных действий из $\{\gamma\}\cup L\setminus\underline{M}$ ($\{\gamma\}\cup M\setminus\underline{L}$) совпадает с подтрассой таких же действий трассы λ (μ), а каждое множество готовности является композицией соответствующих множеств готовности в трассах-операндах. Операция ***ReadyTraces*** взятия множества трасс готовности LTS оказывается *аддитивной* относительно операций композиции LTS и трасс готовности:

$$\boldsymbol{ReadyTraces}(S\rceil\lfloor T)=\cup(\boldsymbol{ReadyTraces}(S)\rceil\lfloor\boldsymbol{ReadyTraces}(T))\quad (\mathbf{!}).$$

Если дивергенция моделируется Δ-действием, мы должны также компоновать бесконечные трассы готовности: если трассы-операнды имеют бесконечные постфиксы противоположных действий, то композиционная трасса заканчивается Δ-действием.

Генеративность и аддитивность трасс готовности позволяют построить замкнутую трассовую теорию, включающую как отношение конформности (даже если оно основано на трассах с отказами, а не на трассах готовности), так и композицию трассовых моделей. Именно это, на наш взгляд, делает трассы готовности полезными в теории конформности.

13. Репликация и симуляции

Наблюдаемое поведение тестируемой системы зависит от того, когда и какие кнопки нажимает оператор. Для того, чтобы смоделировать возможные варианты поведения оператора, в машине тестирования используется, так называемая, *кнопка репликации*. Она позволяет создать несколько копий машины в данный момент времени и продолжить работу с каждой копией независимо. В параметризованной машине мы предполагаем, что такая репликация выполняется перед началом работы. В этом случае различные копии машины имитируют различные «сеансы» тестирования: после каждого сеанса машину можно заставить работать с начала для следующего сеанса. Репликация отличается от такой сеансовой работы только тем, что абстрагируется от числа возможных сеансов: оно может быть и бесконечным, и несчётным. Такая репликация необходима для потенциальной возможности получить все конечные трассы (в машине с приоритетами – истории) в конечных тестовых экспериментах.

В машинах Ван Глаббека и Милнера используется гораздо более сильная репликация в любой момент времени. С помощью такой репликации мы можем получить информацию о состоянии машины, в котором она оказалась после трассы σ, в виде множества трасс, наблюдающихся в этом состоянии. (Тем самым, состояния различаются с точностью до этого множества трасс). Если в каждой копии после наблюдения трассы σ сделана репликация, то в i-ой копии мы получаем множество продолжающих трасс Σ_i, а суммарно имеем семейство $\Sigma=\{\Sigma_i \mid i=1,2,\ldots\}$. Если репликация делается только перед началом работы (только сеансы тестирования), то мы можем узнать лишь объединение $\cup(\Sigma)$, то есть всё множество трасс, которыми продолжается трасса σ во *всех* состояниях после этой трассы. Какие их этих трасс к каким состояниям машины относятся, то есть каковы «слагаемые» $\Sigma_1,\Sigma_2,\ldots$, нам неизвестно.

Репликация в любой момент времени позволяет вводить такие отношения конформности, как симуляции (*simulations*) и бисимуляции (*bisimulations*), учитывающие так или иначе состояния системы. К таким отношениям относятся строгая (*strong*) и слабая (*weak*) симуляции, симуляция задержек (*delay simulation*), симуляция ветвления (*branching simulation*) и другие. Ван Глаббек рассматривает также различные модификации репликации, зависящие от разного рода ограничений на состояние машины, в котором можно делать репликацию (например, только в стабильных состояниях), и от числа копий машины, которые можно создавать при однократном нажатии на кнопку репликации. Эти модификации определяют соответствующие модификации симуляций и бисимуляций. Для связи трассы σ, наблюдаемой до репликации, с её продолжениями после репликации, вводятся специальные операции типа «конъюнкции». Для репликации перед началом работы такие операции излишни, поскольку трасса σ всегда пуста. Репликация в любой момент времени, на наш взгляд, может считаться практически значимой только в редких специальных случаях, и далее мы её не рассматриваем.

14. Глобальное тестирование

Наблюдаемая трасса получается, вообще говоря, недетерминированным способом, то есть она не однозначно определяется последовательностью нажатий кнопок оператором.

При одной и той же последовательности кнопок наблюдаемая трасса определяется тем, как машина выполняет на каждом шаге недетерминированный выбор одного из нескольких выполнимых действий. Недетерминизм выбора действия можно понимать как результат абстрагирования от неких не учитываемых внешних факторов – погодных условий, которые определяют этот выбор детерминировано.

Заметим, что для машины с приоритетами следует не только запоминать истории вместо трасс, но также дополнительно учитывать временные задержки, которые делает оператор между наблюдением и последующим нажатием кнопки или между двумя нажатиями кнопок при их переключении без наблюдения. Поэтому мы включаем в погодные условия также те факторы, которые влияют на «свободу воли» оператора, определяя те или иные временные задержки при нажатии кнопок. Это вполне естественно, если учесть, что оператор моделирует тестер, который является программой компьютера. Такая программа недетерминирована только на некотором уровне абстракции, когда мы отвлекаемся от других программ или аппаратуры, влияющих на её поведение.

Если машина не имеет приоритетов, нам достаточно считать, что оператор работает достаточно быстро. Погодные условия будут включать только факторы, влияющие на работу машины, а не оператора. Это, конечно, не означает, что в некоторых тестовых экспериментах оператор не может работать медленно. Это означает лишь, что любая трасса, которая может быть получена при медленной работе оператора, может быть получена при быстрой работе.

Для полноты тестирования мы должны предполагать, что любые погодные условия могут быть воспроизведены в эксперименте. В формализме машины тестирования это можно понимать так, что при репликации создаётся достаточное число копий для каждого варианта погодных условий. Если такая возможность есть, тестирование называется *глобальным*. Заметим, что мы абстрагируемся от количества вариантов погодных условий. Здесь нам важна только потенциальная возможность проверить поведение системы при любых погодных условиях и любом поведении оператора.

При репликации должна быть создана, по крайней мере, одна копия для каждого сочетания теста как инструкции оператору и варианта погодных условий. Поскольку заранее известно, какой тест прогоняется на машине, мы можем считать что копии промаркированы тестами, тем более, что при тестировании нас интересуют не вообще все возможные тесты, а лишь тесты некоторого полного набора. Таким образом для каждого теста из набора создаётся столько копий, сколько есть вариантов погодных условий. Копия с некоторым вариантом погодных условий для данного модельного теста моделирует прогон реального теста на тестируемой системе.

Конечно, на практике используются только конечные тесты, а также должны быть конечными число тестов в тестовом наборе и число прогонов каждого теста. Поскольку тесты конечные, полный набор, как правило, содержит бесконечное число тестов. Кроме того, без дополнительных условий мы не можем быть уверены, что провели тестовые испытания каждого теста для всех возможных погодных условий. Возможны различные решения этих проблем.

Одно из них – специальные тестовые возможности по управлению погодой. Для этого мы должны выйти за рамки модели, которая как раз и абстрагировалась от второстепенных деталей внешних факторов, то есть от погоды. Тем самым, тестирование становится зависящим не только от спецификации, но и от реализационных деталей, от того, что можно назвать операционной обстановкой, в которой работает реализация. Для каждого варианта такой операционной обстановки мы будем вынуждены создавать свой набор тестов. Тем не менее, в некоторых частных случаях на этом пути можно получить практические выгоды.

Другое решение – специальные реализационные гипотезы. Для конечного набора тестов предполагают, что, если реализация ведёт себя правильно на этих тестах, то она будет вести себя правильно на всех тестах полного набора. Для конечного числа прогонов

теста предполагают, что, если реализация ведёт себя правильно при некоторых погодных условиях, то она будет вести себя правильно при любых погодных условиях.

Третье решение основано на том, что нам известно распределение вероятностей тех или иных погодных условий. В этом случае тестирование оказывается полным с той или иной вероятностью [BGNV05].

Близкое к этому четвёртое решение предполагает, что в каждой ситуации (после трассы) возможно лишь конечное число погодных условий (с точностью до эквивалентности) и существует такое число `N`, что после `N` прогонов теста гарантированно будет проверено поведение реализации при всех возможных в этой ситуации погодных условиях [Miln80, FB92].

Наконец, существует и более радикальное решение – просто запретить недетерминизм реализации, то есть реализационная гипотеза ограничивает класс реализаций только детерминированными реализациями. При всей своей наивности, это достаточно распространённый практический приём [PYB96]. Обоснованием может служить то, что во многих случаях заранее известно, что интересующие нас реализации детерминированы.

Вместо полного, но бесконечного, набора тестов на практике приходится использовать конечные наборы, которые только значимы: если тест выносит вердикт ***fail***, то реализация не конформна. Такой конечный набор строится по тому или иному *критерию покрытия*, чтобы покрыть все интересующие нас классы ситуаций (ошибок). Теоретически конечный набор можно получить фильтрацией по критерию покрытия перечислимого полного набора, хотя на практике обычно используются более прямые методы построения нужного набора. Достаточно общий подход сводится к тому, что вместо исходной спецификационной модели используется более грубая, так называемая, тестовая модель. Тестовая модель – это результат факторизации исходной LTS-спецификации по отношению эквивалентности переходов, что обычно сводится к эквивалентности состояний и/или действий [BKK00]. Иногда при факторизации недетерминизм исчезает. Разумеется, чтобы такой подход был оправданным, нужны мотивированные реализационные гипотезы о том, что все ошибки, которые возможны в реализации, обнаруживаются при тестировании по факторизованной спецификации (вообще по конечному набору, удовлетворяющему критерию покрытия) [BKK00].

Примером практического тестирования может служить тестирование конечного автомата по спецификации, заданной также в виде конечного автомата. Если у нас есть специальная операция, позволяющая достоверно и напрямую опросить текущее состояние реализации (*status message*), то, как известно, полное тестирование сводится к обходу графа переходов автомата и применению операции опроса в каждом проходимом состоянии [LY96]. Обход графа переходов используется также в случае тестирования методом «чёрного ящика», когда состояния реализации не видны. Но здесь для полноты тестирования требуются реализационные гипотезы, компенсирующие отсутствие тестовой возможности достоверного опроса состояний [BKK03-1, BKK04, KPKB03]. Этот подход переносится и на общий случай LTS для систем ввода-вывода [KKPPB03]. В частности, когда используется, так называемое, стационарное тестирование, при котором стимулы подаются в реализацию только в её стационарных состояниях (в этом случае также снимается проблема торможения реакций) [BKK03].

15. Бесконечные и отрицательные наблюдения

Бесконечное наблюдение – это возможность проводить тестовый эксперимент бесконечно долго и получать бесконечную трассу. Для отношений конформности, основанных на конечных трассах, бесконечная трасса не добавляет ничего, чего не давало бы множество всех её конечных префиксов. Исключение составляет прямое наблюдение дивергенции и λ-наблюдение, о которых речь шла выше. Бесконечные наблюдения и

отношения конформности, основанные на бесконечных трассах, обычно не рассматриваются как значимые для практического тестирования.

Отрицание $\neg\sigma$ означает, что трасса σ отсутствует во множестве всех наблюдаемых трасс, которые мы можем гарантированно получить с помощью данной машины тестирования для проверки данного отношения конформности. Отрицательное наблюдение предполагает, что мы можем (хотя бы потенциально) получить все наблюдаемые трассы. Для этого требуется репликация (хотя бы перед началом работы) и глобальное тестирование. Фактически, отрицательное наблюдение – это наблюдение, «вычисляемое» по положительным, то есть «реальным» наблюдениям. При репликации только перед началом работы всё множество наблюдаемых трасс представляет собой объединение множеств трасс для каждого теста.

Если бы мы не запрещали тупик во взаимодействии окружения и реализации, то при отсутствии дивергенции мы могли бы вычислять некоторые отказы как отрицательные наблюдения. Действительно, пусть трасса σ не продолжается ни одним действием из $P\in\mathfrak{R}\cup\mathfrak{Q}$ во множестве наблюдаемых трасс, то есть имеет место $\neg\sigma\cdot\langle z\rangle$ для каждого $z\in P$. Тогда трасса σ в реализации продолжается либо дивергенцией, либо отказом P, то есть при отсутствии дивергенции можно вычислить трассу $\sigma\cdot\langle P\rangle$. Однако отсюда не следует, что можно наблюдать все отказы: трасса σ может продолжаться в реализации как отказом P, так и некоторыми действиями $z\in P$. Для наблюдения всех отказов нужна уже репликация в любой момент времени. Кроме того, возникает проблема с продолжением после отказа. Для того, чтобы, наблюдая трассу $\sigma\cdot\mu$, узнать, есть или нет трасса $\sigma\cdot\langle P\rangle\cdot\mu$, мы должны определить, начинается ли трасса μ в стабильном состоянии после трассы σ. Если трасса σ заканчивается некоторым отказом R, то это так, но как получить сам этот отказ R? В целом получается, что, даже в том случае, когда разрешены тупики и отрицательные наблюдения отказов, нам нужен какой-то реальный способ наблюдения остановки машины: зелёная лампочка или $\mathfrak{R}$-отказ в $\mathfrak{R}/\mathfrak{Q}$-машине.

Аналогичная ситуация складывается с вычислением множеств готовности. Без репликации в любой момент времени мы можем вычислить лишь множество действий, продолжающих трассу, то есть объединение множеств готовности в стабильных состояниях плюс множество действий, определённые в нестабильных состояниях после трассы. Для определения слагаемых нужна репликация в любой момент времени и реальное наблюдение остановки машины.

Если мы запрещаем при правильном взаимодействии тупик, дивергенцию и разрушение, то отрицательные наблюдения ничего не дают для конформностей, основанных на принципе независимости: поведение реализации правильно или неправильно *независимо* от других её поведений. Окончательный вердикт равен конъюнкции вердиктов во всех прогонах всех тестов из полного набора.

16. Приоритеты

О преимуществах приоритетов в реализации и проблемах тестирования таких реализаций выше было сказано уже достаточно. Здесь хотелось бы только отметить, что θ-переход, который обычно допускается только в тесте, мог бы использоваться в реализации как один из способов задания приоритета. Фактически, такой переход является внутренним, но имеет наименьший приоритет: выполняется только тогда, когда остальные переходы не могут выполняться. В реализации θ-переход может использоваться как способ задания альтернативного поведения при возникновении тупиков. В частности, торможение реакций окружением (не принимаются реакции, посылаемые реализацией) перестаёт быть проблемой, если в реализации можно задать альтернативное поведение при таком торможении. Другая возможность: θ-действие в стационарном состоянии

реализации позволяет ей распознать отсутствие стимулов. Этот вариант мы исследовали в [BKK03], где такое θ-действие называли ε-действием. К сожалению, приоритеты почти не отражены в современной теории тестирования.

17. Заключение

Мы рассмотрели различные тестовые возможности, задаваемые с помощью машины тестирования. Среди них мы выделили набор теоретически достаточно мощных и практически значимых возможностей, определяющих $\mathfrak{R}/\mathfrak{Q}$-семантику тестирования. Эта семантика строится на наблюдениях внешних действий и отказов. Нововведениями являются: 1) Параметризация семантики семействами наблюдаемых и ненаблюдаемых отказов, что позволяет учитывать те или иные ограничения на (правильное) взаимодействие. 2) Разрушение как запрещённое действие, которое возможно, но не должно выполняться при правильном взаимодействии. 3) Моделирование дивергенции Δ-действием, которое тоже возможно, но не должно выполняться при правильном взаимодействии. Предлагаются основанные на такой семантике понятие безопасного тестирования, реализационная гипотеза о безопасности и отношение безопасной конформности (***saco***), отвечающее принципу независимости наблюдений.

Для более узкого класса взаимодействий могут также использоваться $\mathfrak{R}/\mathfrak{Q}$-***Ready***-семантика, основанная на трассах готовности, и соответствующее отношение ***resaco***. Для этих трасс, в отличие от $\mathfrak{R}$-трасс, можно определить композицию так, что трассы готовности композиции LTS совпадают с композицией трасс готовности LTS-операндов. Это свойство аддитивности вместе со свойством генеративности (по трассам готовности можно получить $\mathfrak{R}$-трассы) позволяют построить замкнутую трассовую теорию, включающую как отношение конформности (даже если оно основано на $\mathfrak{R}$-трассах), так и композицию трассовых моделей. Это делает трассы готовности полезными в теории конформности независимо от практической возможности их наблюдения.

Также мы сформулировали ряд утверждений о связи отношений ***saco*** и ***resaco*** в $\mathfrak{R}/\mathfrak{Q}$-, $\mathfrak{R}\cup\mathfrak{Q}/\varnothing$-, $\mathfrak{R}/\mathfrak{Q}$-***Ready***- (и $\mathfrak{R}\cup\mathfrak{Q}/\varnothing$-***Ready***-) семантиках. Переход к следующей семантике в этом списке осуществляется с помощью преобразования пополнения, решающего проблему рефлексивности отношения, и монотонного преобразования, решающего проблему монотонности (сохранение при композиции) конформности. Доказательство утверждений, помеченных знаком (**!**) выходит за рамки статьи. В [BKK07] эти утверждения доказаны для систем ввода-вывода, в которых наблюдаемыми отказами могут быть только стационарность и блокировки стимулов. Утверждение об отношении ***resaco*** пока остаётся гипотезой.

Литература

[Bern91] G. Bernot. 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.

[BGNV05] Andreas Blass, Yuri Gurevich, Lev Nachmanson, and Margus Veanes. Play to Test Microsoft Research, Technical Report MSR-TR-2005-04, January 2005. 5th International Workshop on Formal Approaches to Testing of Software (FATES 2005). Edinburgh, July 2005.

[BK05-1] Бурдонов И. Б., Косачев А. С. «Тестирование компонентов распределенной системы.» Труды Всероссийской научной конференции «Научный сервис в сети ИНТЕРНЕТ», Изд-во МГУ, 2005, стр.63-65.

[BK05-2] Бурдонов И. Б., Косачев А. С. «Верификация композиции распределенной системы.» Труды Всероссийской научной конференции «Научный сервис в сети ИНТЕРНЕТ», Изд-во МГУ, 2005, стр.67-69.

[BKK00] И. Б. Бурдонов, А. С. Косачев, В. В. Кулямин. Использование конечных автоматов для тестирования программ. "Программирование". 2000. No. 2. стр.12-28.

[BKK03] И. Б. Бурдонов, А. С. Косачев, В. В. Кулямин. «Асинхронные автоматы: классификация и тестирование.» Труды ИСП РАН, т. 4, 2003, с. 7-84.

[BKK03-1] И. Б. Бурдонов, А. С. Косачев, В. В. Кулямин. Неизбыточные алгоритмы обхода ориентированных графов. Детерминированнный случай. "Программирование". 2003. No. 5.

[BKK04] И. Б. Бурдонов, А. С. Косачев, В. В. Кулямин. Неизбыточные алгоритмы обхода ориентированных графов. Недетерминированнный случай. "Программирование". 2004. No. 1.

[BKK06] I. Bourdonov, A. Kossatchev, and V. Kuliamin. Formal Conformance Testing of Systems with Refused Inputs and Forbidden Actions. Proc. of MBT 2006, Vienna, Austria, March 2006.

[BKK07] И. Б. Бурдонов, А. С. Косачев, В. В. Кулямин. Теория соответствия для систем с блокировками и разрушением. "Наука", в печати.

[BP94] Gregor V. Bochmann , Alexandre Petrenko. Protocol testing: review of methods and relevance for software testing, Proceedings of the 1994 international symposium on Software testing and analysis, pp.109-124, August 17-19, 1994, Seattle, Washington, United States.

[BRT03] Machiel van der Bijl, Arend Rensink, Jan Tretmans. Compositional testing with ioco. In Formal Approaches to Software Testing: Third International Workshop, FATES 2003, Montreal, Quebec, Canada, October 6th, 2003. Editors: Alexandre Petrenko, Andreas Ulrich ISBN: 3-540-20894-1. LNCS volume 2931, Springer, pp. 86-100.

[BRT03r] M. van der Bijl, A. Rensink, J. Tretmans. Component Based Testing with ioco. CTIT Technical Report TR-CTIT-03-34, University of Twente, 2003.

[DNH84] R. De Nicola and M.C.B. Hennessy. Testing equivalences for processes. Theoretical Computer Science, 34:83–133, 1984.

[FB92] S. Fujiwara and G. von Bochmann. Testing Nondeterministic Finite State Machine with Fault Coverage. IFIP Transactions, Proceedings of IFIP TC6 Fourth International Workshop on Protocol Test Systems, 1991, Ed. by Jan Kroon, Rudolf J. Hei-jink, and Ed Brinksma, 1992, North-Holland, pp. 267-280.

[Glab90] R.J. van Glabbeek. 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.

[Glab93] 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.

[Heer98] L. Heerink. Ins and Outs in Refusal Testing. PhD thesis, University of Twente, Enschede, The Netherlands, 1998.

[Hoare69] C. A. R. Hoare. "An axiomatic basis for computer programming". Communications of the ACM, 12(10):576–585, October 1969.

[HT97] L. Heerink, J. Tretmans. *Refusal Testing for Classes of Transition Systems with inputs and Outputs.* In T. Mizuno, N. Shiratori, T. Higashino, A. Togashi, eds. Formal Description Techniques and Protocol Specification, Testing and Verification. Chapman & Hill, 1997.

[JJTV99] C. Jard, T. Jéron, L. Tanguy, C. Viho. Remote testing can be as powerful as local testing, in Formal methods for protocol engineering and distributed systems, FORTE XII/ PSTV XIX' 99, Beijing, China, J. Wu, S. Chanson, Q. Gao (eds.), pp. 25-40, October 1999.

[KKPPB03] Victor V. Kuliamin, Alexander S.Kossatchev, Alexander K. Petrenko, Nick V. Pakoulin, Igor B. Bourdonov. Integration of Functional and Timed Testing of Real-Time and Concurrent Systems. Perspectives of System Informatics // LNCS. No. 2890. 2003. pp. 450-461.

[KPKB03] В.В.Кулямин, А.К.Петренко, А.С.Косачев, И.Б.Бурдонов. Подход UniTesK к разработке тестов. "Программирование", 2003, №6, стр.25-43.

[Lan90] R. Langerak. A testing theory for LOTOS using deadlock detection. In E. Brinksma, G. Scollo, and C. A. Vissers, editors, Protocol Specification, Testing, and Verification IX, pages 87–98. North-Holland, 1990.

[LG05] G. Lestiennes, M.-C. Gaudel. Test de systemes reactifs non receptifs. Journal Europeen des Systemes Automatises, Modelisation des Systemes Reactifs, pp. 255–270. Hermes, 2005.

[LT87] Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In Proceedings of the 6th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, pp. 137-151, August 1987.

[LY96] D. Lee and M. Yannakakis. Principles and Methods of Testing Finite State Machines—A Survey, Proceedings of the IEEE 84, No. 8, 1090–1123, 1996.

[Miln80] R. Milner. A Calculus of Communicating Systems, LNCS, vol. 92, Springer-Verlag, 1980.

[Miln81] R. Milner. Modal characterisation of observable machine behaviour. In G. Astesiano & C. Bohm, editors: Proceedings CAAP 81, LNCS 112, Springer, pp. 25-34.

[Miln89] R. Milner. Communication and Concurrency. Prentice-Hall, 1989.

[Phal94] M. Phalippou. Relations d`Implantation et Hypotheses de Test sur des Automates a Entrees et Sorties. PhD thesis, L`Universite de Bordeaux I, France, 1994.

[PYB96] A. Petrenko, N. Yevtushenko, G. von Bochmann. Testing deterministic implementations from nondeterministic FSM specifications. Selected proceedings of the IFIP TC6 9-th international workshop on Testing of communicating systems, September 1996.

[PYH03] A. Petrenko, N. Yevtushenko and J.L. Huo. Testing Transition Systems with Input and Output Testers. In: Proc. IFIP TC6/WG6.1 15th Int. Conf. Testing of Communicating Systems, TestCom'2003, pp. 129-145. Sophia Antipolis, France, May 26-29, 2003.

[Tret92] J. Tretmans. A Formal Approach to Conformance Testing. PhD. Thesis, University of Twente, Enschede, The Netherlands, 1992.

[Tret96] Tretmans, J. Test Generation with Inputs, Outputs and Repetitive Quiescence. In: Software-Concepts and Tools, Vol. 17, Issue 3, 1996.

[Vaan91] F. Vaandrager. On the relationship between process algebra and Input/Output Automata. In Logic in Computer Science, pp. 387-398. Sixth Annual IEEE Symposium, IEEE Computer Society Press, 1991.