Спецификация и тестирование систем с асинхронным интерфейсом

А.В. Хорошилов (khoroshilov@ispras.ru)

Аннотация. В работе рассматривается метод спецификации и тестирования систем с асинхронным интерфейсом при помощи технологии UniTesK. Определяются математические модели, лежащие в основе метода, и подходы к решению основных задач тестирования для систем с асинхронным интерфейсом. Предлагается унифицированная архитектура теста, определяющая архитектуру всех тестовых систем, построенных в соответствии с предложенным методом. Описывается реализация поддержки метода в наборе инструментов CTesK и опыт применения метода, полученный в шести проектах по тестированию различного программного обеспечения.

1. Введение

1.1. Формальные методы и тестирование программного обеспечения

В последнее время наблюдается бурное развитие компьютерных технологий. Они проникают во все сферы деятельности человека и оказывают все большее влияние на его жизнь. Системы управления транспортом и автоматизированные линии производства, банковские платежные системы и телекоммуникационные сети, медицинские системы обеспечения жизнедеятельности и интеллектуальные дома – все это является неотъемлемой частью современного мира.

Однако, чем большее значение отводится компьютерным системам, тем выше становится цена их ошибок. Как показывает практика, наиболее уязвимым местом компьютерных систем с этой точки зрения является программное обеспечение. Так, ошибка в одном программном модуле многомиллионного межпланетного корабля Mars Climate Orbiter привела к его гибели в атмосфере Марса после более чем девятимесячного полета [1,2].

Более того, некорректное поведение современных программных систем влечет не только огромные убытки, но и приводит к гибели людей. Например, ошибки в программном обеспечении медицинского оборудования Therac-25 привели к получению повышенной дозы радиации и последующей смерти 7 пациентов [3,4]. А арифметическая ошибка в программном обеспечении комплекса противовоздушной обороны Patriot не позволила вовремя обнаружить иракскую ракету, что привело к гибели 28 солдат во время ирако-американской

войны 1991 года [5]. И это только малая часть уже имевших место прецедентов.

С нарастанием сложности и важности задач, решаемых программными системами, проблема обеспечения их качества становится все острее. Много надежд на существенный прогресс в этом направлении связывается с развитием *формальных методов*.

В широком смысле, под формальными методами понимаются любые попытки использования математических подходов к разработке программного обеспечения с целью повышения его качества[6]. Как правило, для этого используются математические модели различных сущностей, участвующих в процессе разработки. Примерами таких моделей могут служить модель исходных требований к разрабатываемой системе, модель архитектуры системы или модель реализации системы.

Одним из основных направлений в области формальных методов являются методы доказательства корректности программ, такие как методы аналитической верификации и проверки моделей [7,8,9,10]. В этих методах доказательство корректности программ строится по следующей схеме. Для данной программной системы создаются математическая модель требований к системе и математическая модель реализации этой системы. После чего доказывается наличие отношения «удовлетворяет» между этими двумя моделями, что и рассматривается как доказательство корректности программы (Рис. 1). Существует также целый набор вариаций подхода. Например, в качестве модели требований может выступать модель программной системы более высокого уровня абстракции, или доказательство корректности может сводиться к доказательству непротиворечивости одной из математических моделей.

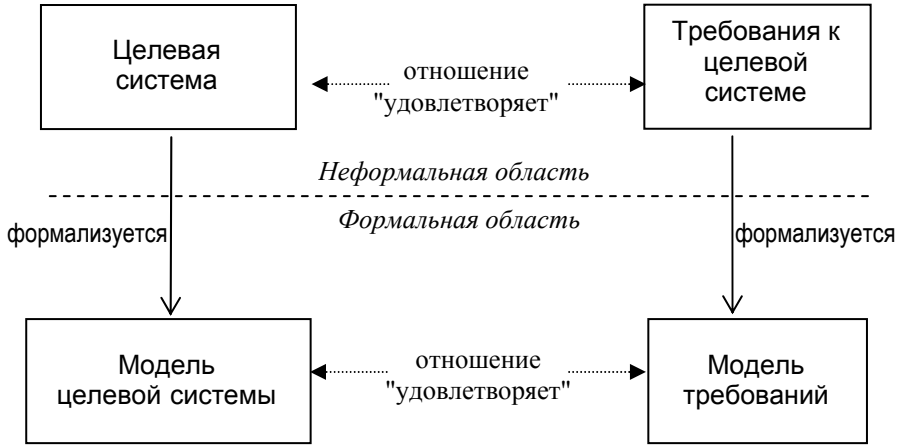

Рис. 1. Схема работы методов аналитической верификации и проверки моделей

Однако, несмотря на большие усилия, вложенные в развитие данного направления, на настоящий момент так и не появилось методов доказательства корректности программ, которые смогли бы предоставить приемлемые решения для обеспечения качества реальных программных систем. В результате, одним из основных средств обеспечения качества программных систем было и остается тестирование: при разработке программ с повышенными требованиями к надежности доля затрат на тестирование в бюджете проекта может достигать 80%.

Многочисленные исследования в области формальных методов нашли свое отражение и в развитии новых технологий тестирования. Одно из наиболее активно развивающихся направлений – тестирование на основе моделей, уже успело показать свои преимущества в целом ряде крупных индустриальных проектов [11,12,13,14,15,16,17].

Тестирование на основе моделей позволяет систематизировать и автоматизировать процесс разработки тестовых наборов посредством использования различного рода математических моделей. И в тех случаях, когда небольшого ручного тестирования оказывается недостаточно для обеспечения требуемого уровня качества программного обеспечения, тестирование на основе моделей позволяет добиться существенного повышения качества тестирования с затратами меньшими, чем при использовании других подходов.

1.2. Технология UniTesK

Комплексный подход, позволяющий автоматизировать целый спектр задач тестирования на основе математических моделей, представляет собой технология тестирования UniTesK [18,19,20], разработанная в Институте Системного Программирования РАН. Данная технология использует широко известный подход программных контрактов [21] для формального описания требований к программному обеспечению и уникальные методы генерации сложных последовательностей тестовых воздействий на основе неявного описания конечного автомата. Эффективность такого подхода была подтверждена в многочисленных проектах [22], и, в частности, при разработке тестового набора для ядра операционной системы канадского телекоммуникационного гиганта Nortel Networks [23].

Технология тестирования UniTesK основывается на базовых принципах формальных методов, таких как формальные спецификации требований к программной системе, и в то же время остается доступной для применения в крупных индустриальных проектах. Ключевым элементов в достижении этого является переход от доказательства корректности программной системы к проверке корректности реального поведения системы на тестовых данных. Для этого вместо построения модели реализации программной системы и доказательства ее корректности относительно модели требований на всех возможных входных данных используется наблюдение за реальным поведением программной системы на конкретных тестовых данных,

построение по результатам наблюдения модели поведения системы и доказательство корректности этой модели относительно модели требований (Рис. 2).

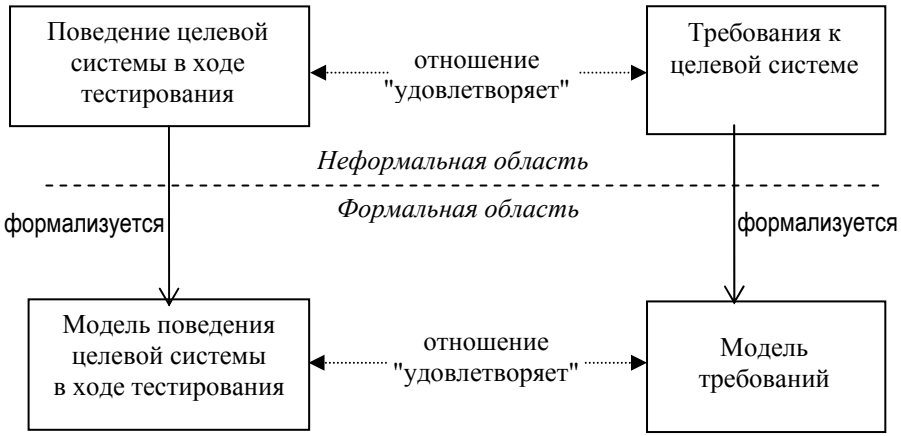

Рис. 2. Схема работы технологии UniTesK относительно формальных методов

Такое решение обладает двумя существенными достоинствами, которые обусловлены тем, что модель поведения программной системы на конкретных данных значительно проще, чем модель реализации этой системы, отражающая поведение программной системы на всех возможных входных данных.

Первое достоинство решения, предлагаемого технологией UniTesK, заключается в значительном сглаживании перехода между областью неформальных объектов и областью их математических моделей. Этот переход является одним из наиболее критикуемых мест формальных методов [24]. Применительно к методам доказательства корректности программ проблема данного перехода заключается в том, что доказательство корректности математической модели реализации программной системы не гарантирует корректность самой реализации. И единственным средством для проверки соответствия между реализацией системы и ее моделью является их сопоставление, выполняемое человеком. Для реальных программных систем такое сопоставление является большой проблемой в виду размеров модели и сложности связей между ее составляющими. Модель поведения системы на конкретных данных, используемая в технологии UniTesK, является значительно более простой, чем модель реализации, а значит сопоставление модели поведения с реальным поведением системы является значительно более простым, чем сопоставление модели реализации с самой реализацией.

Второе достоинство решения, предлагаемого технологией UniTesK, состоит в упрощении методов доказательства корректности. Доказать, что поведение

программной системы на конкретных тестовых данных является корректным, значительно проще, чем доказать то же самое для всех возможных входных данных. Наиболее важным практическим результатом данного факта является то, что область применимости методов доказательства корректности модели поведения программной системы на конкретных данных значительно превосходит область применимости методов доказательства корректности моделей реализации программных систем в целом. Поэтому во многих случаях, когда размеры программных систем не позволяют использовать методы доказательства корректности программ, тестирование на основе моделей по технологии UniTesK успешно применяется и доказывает свои преимущества.

Расплатой за эти преимущества является неполнота тестирования по сравнению с доказательством корректности программ. Если в результате доказательства корректности модели реализации, при допущении корректности соответствия между реализацией и ее моделью, следует корректность реализации на всех входных данных, то в результате доказательства корректности модели поведения программной системы на конкретных тестовых данных, при допущении корректности соответствия между реальным поведением и построенной моделью, следует корректность реализации только на тех тестовых данных, которые использовались в процессе тестирования.

С другой стороны, если сравнивать технологию UniTesK с обычными методами разработки тестов, то наличие формальных спецификаций требований и методов их автоматической проверки, позволяет значительно упростить процесс разработки высококачественных тестовых наборов. Это достигается за счет того, что добавление новых тестовых данных не требует решения задачи оценки корректности поведения тестируемой системы, так как эта оценка выполняется автоматически на основе формальной спецификации требований.

Более того, задача генерации тестовых данных также может быть в значительной степени автоматизирована на основе использования формальных спецификаций. И эта возможность также реализована в технологии UniTesK в виде уникальных методов генерации последовательностей тестовых воздействий на основе неявного описания конечного автомата.

Таким образом, технология UniTesK не предоставляет замену методам доказательства корректности программ, но позволяет воспользоваться формальным математическим базисом для разработки качественных тестов с меньшим уровнем затрат по сравнению с обычными методами разработки тестов.

1.3. Системы с асинхронным интерфейсом

Изначально технология UniTesK использовала для формального описания требований к программному обеспечению подход классических программных контрактов [21]. Этот подход показал себя с лучшей стороны при разработке тестового набора для ядра операционной системы канадского телекоммуникационного гиганта Nortel Networks [23]. В то же время,

проявился и ряд ограничений, связанных с ограниченной областью применимости классических программных контрактов.

Подход классических программных контрактов, основанный на описании инвариантов данных, а также предусловий и постусловий интерфейсных операций, предполагает синхронность всех взаимодействий с программной системой. То есть, работа программной системы рассматривается как последовательность вызовов интерфейсных операций, осуществляемых последовательно: следующий вызов происходит только после завершения предыдущей вызванной операции. Кроме того, классические программные контракты основываются на предположении, что все взаимодействия с программной системой инициируются окружением этой системы, а сама система не может инициировать никаких взаимодействий.

С другой стороны, эти предположения не выполняются для широкого класса программных систем, для которых возможность одновременного участия в нескольких взаимодействиях или инициации взаимодействий с окружением является существенной составляющей функциональности. Такие системы далее мы будем называть *системами с асинхронным интерфейсом*.

Например, телекоммуникационные протоколы и драйверы устройств могут участвовать одновременно в нескольких взаимодействиях со своим окружением, и, кроме того, они могут инициировать эти взаимодействия самостоятельно. Другим примером систем с асинхронным интерфейсом, являются компоненты, использующие межпроцессное и межпотоковое взаимодействие, компоненты, создающие и удаляющие процессы или потоки управления, а также компоненты, содержащие интерфейсные функции, которые блокируются до наступления некоторых событий.

Формальные спецификации требований к системам с асинхронным интерфейсом на основе классических программных контрактов не могут полностью описать все существенные особенности их функционирования. Поэтому тестирование таких систем при помощи технологии UniTesK в рамках подхода классических программных контрактов не может быть выполнено с требуемым уровнем качества.

В настоящей работе представлен метод спецификации и тестирования систем с асинхронным интерфейсом, разработанный в рамках развития технологии тестирования UniTesK. Структура работы построена следующим образом. Первая глава содержит введение, описывающее контекст работы и дающее общий взгляд на последующее изложение. Во второй главе рассматриваются математические модели и унифицированная архитектура теста, используемые в технологии UniTesK для тестирования синхронных систем. Третья глава посвящена методу спецификации и тестирования систем с асинхронным интерфейсом. В ней определяется понятие систем с асинхронным интерфейсом и исследуются ограничения классических программных контрактов, не позволяющие использовать их для проведения полноценного тестирования асинхронных систем. Затем определяются математические модели, позволяющие корректным образом сформулировать основные задачи

тестирования для систем с асинхронным интерфейсом, и рассматриваются алгоритмы, решающие эти задачи. В заключении главы обсуждается унифицированная архитектура асинхронной тестовой системы, определяющая архитектуру всех тестовых систем, предназначенных для тестирования систем с асинхронным интерфейсом рассматриваемым методом. В четвертой главе обсуждается инструментальная поддержка тестирования систем с асинхронным интерфейсом на основе технологии UniTesK, реализованная в наборе инструментов CTesK на платформе языка программирования C. Пятая глава описывает опыт применения технологии UniTesK для тестирования систем с асинхронным интерфейсом. В шестой главе представлено заключение, содержащее краткий обзор настоящей работы.

Разработка метода тестирования систем с асинхронным интерфейсом являлась составной частью исследований, проводимых группой UniTesK Lab Института Системного Программирования РАН в рамках развития технологии тестирования UniTesK. Исследование методов спецификации и тестирования систем с асинхронным интерфейсом было выполнено автором совместно с И.Б. Бурдоновым, А.С. Косачевым и В.В. Куляминым. Также автор выражает свою признательность руководителю группы UniTesK Lab А.К. Петренко и коллегам, участвовавшим в реализации системы тестирования CTesK и апробации рассматриваемого метода: В. Омельченко, А. Коптелову, Е. Рогову, Н. Пакулину и Г. Ключникову.

2. Архитектура UniTesK для систем с синхронным интерфейсом

Настоящая глава состоит из четырех разделов. Первые три раздела посвящены рассмотрению математических моделей и принципов решения основных задач тестирования:

- задачи оценки корректности поведения тестируемой системы;
- задачи генерации тестовых данных;
- задачи оценки качества тестирования.

Четвертый раздел посвящен обсуждению унифицированной архитектуры теста, определяющей архитектуру всех тестовых систем, построенных по технологии UniTesK для систем с синхронным интерфейсом.

Математические модели, описанные в данной главе, несколько отличаются от традиционных описаний технологии UniTesK [18,19,25], что обусловлено попыткой взглянуть на технологию UniTesK с практической точки зрения.

2.1. Основные понятия

Перед началом рассмотрения технологии UniTesK введем несколько базовых терминов.

Целевой системой будем называть программную систему, которую необходимо протестировать. В качестве синонима целевой системы также будем использовать словосочетание тестируемая система. В англоязычной

литературе этим терминам соответствуют сокращения SUT (System Under Test) и IUT (Implementation Under Test).

Тестовой системой будем называть комплекс программ, предназначенный для тестирования целевой системы. В основе тестовой системы, разработанной согласно технологии UniTesK, лежит *унифицированная архитектура теста*, которая определяет набор компонентов теста, обладающих ясным разделением функций и четкими интерфейсами. Эти компоненты составляют ядро тестовой системы и отвечают за организацию процесса тестирования и выполнение всех связанных с этим задач.

2.2. Оценка корректности поведения тестируемой системы

Технология UniTesK предназначена для автоматизации процесса разработки и выполнения функциональных тестов. При этом под *функциональным тестированием* понимается процесс взаимодействия с целевой системой, направленный на проверку выполнения этой системой требований, предъявляемых к ее функциональности. В этом определении два ключевых момента: взаимодействия и требования.

Типичным примером взаимодействия с тестируемой системой является вызов ее интерфейсной функции и получения результата ее работы. Другим примером взаимодействия может быть посылка HTTP запроса и получение ответа на этот запрос. То есть взаимодействием является любой обмен информацией с целевой системой, в том числе и направленный только в одну сторону.

Требования к функциональности целевой системы описывают ту функциональность, которую данная система должна предоставлять своему пользователю. Поскольку пользователь общается с целевой системой посредством взаимодействий, то и его требования к системе выражаются в терминах взаимодействий. Другими словами, требования определяют какие взаимодействия являются корректными (ожидаемыми), а какие – нет.

Например, пользователь библиотеки математических функций ожидает, что вызвав функцию sqrt с параметром x, он получит квадратный корень x, вычисленный с заданной точностью.

В настоящей работе под *оценкой корректности поведения целевой системы* понимается проверка того, что поведение целевой системы, наблюдаемое в процессе тестирования, удовлетворяет требованиям, предъявляемым к функциональности системы.

2.2.1. Формализация задачи

Проблема оценки корректности поведения целевой системы является одной из основных задач функционального тестирования. В технологии UniTesK эта задача решается следующим образом.

Поведение целевой системы в процессе тестирования представляется формальным образом в виде модели поведения, которая состоит из набора взаимодействий целевой системы со своим окружением. Требования,

предъявляемые к функциональности целевой системы, описываются формально в виде модели требований. На декартовом произведении всех возможных моделей поведения и всех возможных моделей требований определяется формальное отношение “удовлетворяет”, которое соответствует неформальному пониманию отношения “удовлетворяет” между поведением целевой системы и функциональными требованиями. В результате, на уровне математической модели, задача оценки корректности поведения целевой системы эквивалентна проверке наличия отношения “удовлетворяет” между моделью поведения целевой системы и моделью требований. Рассмотренный подход проиллюстрирован на Рис. 2.

Рассмотрим формальные определения моделей поведения и требований.

2.2.2. Модель поведения

Интерфейсом целевой системы будем называть тройку (X, Y, V), где

- X – множество стимулов,
- Y – множество реакций,
- V – множество состояний.

Каждое из этих множеств может быть бесконечным[1].

Взаимодействием с целевой системой, обладающей интерфейсом (X, Y, V), будем называть четверку $(v, x, y, v') \in V \times X \times Y \times V$. Первый элемент взаимодействия v мы будем называть пресостоянием, второй x – стимулом, y – реакцией, v' – постсостоянием.

Понятие взаимодействия можно интерпретировать следующим образом. В состоянии v вызывается интерфейсная операция с некоторыми значениями параметров x, на что целевая система возвращает выходные параметры y и переходит в состояние, соответствующее состоянию v'.

Например, для функции вычисления квадратного корня возможны следующие взаимодействия: (ε, sqrt(0.0), 0.0, ε), (ε, sqrt(4.0), 2.0, ε) или (ε, sqrt(10.0), 1.0, ε). В этом примере множество состояний V состоит из единственного элемента ε, множество стимулов X и реакций Y совпадают с множеством всех неотрицательных действительных чисел.

Определение 1.

Моделью поведения целевой системы с интерфейсом (X, Y, V) будем называть конечное или бесконечное мультимножество взаимодействий $\{ (v_i, x_i, y_i, v'_i) \}$.

Модель поведения соответствует набору взаимодействий с целевой системой, имевших место в процессе тестирования. Например, предположим, что в процессе тестирования функция вычисления квадратного корня была вызвана трижды с параметрами 0.0, 4.0 и 10.0 и вернула значения 0.0, 2.0 и 1.0 соответственно. Тогда моделью поведения целевой системы будет мультимножество взаимодействий, произошедших в процессе тестирования $\{ (\varepsilon, sqrt(0.0), 0.0, \varepsilon), (\varepsilon, sqrt(4.0), 2.0, \varepsilon), (\varepsilon, sqrt(10.0), 1.0, \varepsilon) \}$.

[1] В данной работе рассматриваются только не более чем счетные множества.

2.2.3. Модель требований

Абстрактным автоматом будем называть четверку (V, X, Y, E), где

- V – множество состояний,
- X – множество стимулов,
- Y – множество реакций,
- $E \subseteq V \times X \times Y \times V$ – множество переходов.

Переходы автомата (v, x, y, v') состоят из пресостояния v, стимула x, реакции y и постсостояния v'. Множества V, X, Y и E могут быть бесконечными.

Определение 2.

Моделью требований к целевой системе с интерфейсом (X, Y, V) будем называть абстрактный автомат, множества состояний, стимулов и реакций которого совпадают с V, X и Y соответственно.

Будем говорить, что взаимодействие (v, x, y, v') является *корректным* относительно модели требований A = (V, X, Y, E), если оно принадлежит множеству переходов E.

Будем говорить, что модель поведения целевой системы $\{ (v_i, x_i, y_i, v'_i) \}$ *удовлетворяет* модели требований A = (V, X, Y, E), если $\forall i \; (v_i, x_i, y_i, v'_i) \in E$.

Рассмотрим пример с функцией вычисления квадратного корня. Что является моделью требований к этой функции? Интерфейс этой системы (X, Y, V) был определен в предыдущем разделе. Множество переходов E определим как все переходы $(\varepsilon, x, y, \varepsilon) \in V \times X \times Y \times V$, для которых выполнено следующее ограничение: $| x - y^2 | < \varepsilon_{погр}$, где $\varepsilon_{погр}$ – некоторое положительное действительное число, меньшее единицы. Тогда взаимодействия (ε, sqrt(0.0), 0.0, ε) и (ε, sqrt(4.0), 2.0, ε) будут корректными относительно данной модели функциональных требований, а взаимодействие (ε, sqrt(10.0), 1.0, ε) – нет.

2.2.4. Программные контракты

В качестве основной модели в технологии UniTesK выступает модель требований. Эта модель создается первой и именно она является основой для разработки остальных моделей. Причина такого подхода заключается в том, что именно требования являются исходными данными как для процесса разработки целевой системы, так и для процесса тестирования.

Для описания модели требований в технологии UniTesK используется широко известный подход программных контрактов [21]. Основная идея этого подхода состоит в следующем. В целевой системе выделяется набор интерфейсных операций, то есть операций, которые определяют функциональность системы. Каждая такая операция имеет набор входных и выходных параметров. И каждое взаимодействие с целевой системой рассматривается как вызов интерфейсной операции с некоторым набором значений входных параметров и получение от нее набора значений выходных параметров.

Спецификация интерфейсной операции состоит из предусловия и постусловия. *Предусловие* операции определяет, какие значения входных параметров являются допустимыми для передачи целевой системе. Например, предусловие функции sqrt может ограничивать множество допустимых значений входного параметра только неотрицательными числами.

Постусловие операции определяет, какие выходные значения параметров являются корректными для данных значений входных параметров. То есть, постусловие описывает требования к целевой системе при воздействии на нее посредством данной интерфейсной операции.

Если поведение целевой системы зависит от истории ее предыдущих взаимодействий, то в спецификации требований это моделируется введением так называемого *модельного состояния*. Модельное состояние образуется набором переменных, хранящих информацию об истории взаимодействий с целевой системой. Эти данные используются при спецификации интерфейсных операций.

В этом случае предусловие операции определяет, какие параметры являются допустимыми в данном состоянии, а постусловие описывает, какие выходные значения параметров и постзначения переменных модельного состояния являются корректными при данных значениях входных параметров и презначениях этих переменных.

Презначениями переменных модельного состояния называются значения этих переменных до вызова данной интерфейсной операции, а *постзначениями* – значения после вызова.

2.2.5. Описание модели требований

Рассмотрим подход программных контрактов более детально и определим способ описания модели требований, применяемый в технологии UniTesK.

Сигнатурой интерфейсной операции I называется тройка (In, Out, Var), где

- $In = \{ x_i \mid i = 1, \ldots, in(I) ; in(I) \geq 0 \}$ – упорядоченный набор входных параметров операции,
- $Out = \{ y_i \mid i = 1, \ldots, out(I) ; out(I) \geq 0 \}$ – упорядоченный набор выходных параметров операции,
- $Var = \{ v_i \mid i = 1, \ldots, var(I) ; var(I) \geq 0 \}$ – набор переменных состояния, к которым обращается данная операция.

Каждый входной параметр x_i может принимать значения из непустого множества допустимых значений X_i, каждый выходной параметр y_i может принимать значения из непустого множества допустимых значений Y_i, а каждая переменная состояния v_i может принимать значения из непустого множества допустимых значений V_i. Эти непустые множества допустимых значений далее будут называться *типами* параметров и переменных.

Рассмотрим пример операции вычисления квадратного корня. Сигнатурой этой операции является тройка ({ x_1 }, { y_1 }, {}). Множеством допустимых значений единственного входного параметра x_1 и единственного выходного параметра y_1 является множество действительных чисел **R**. Пустое множество

переменных состояния означает, что данная операция никак не зависит ни от одной переменной состояния.

Введем следующие обозначения:

$X_I = X_1 \times \ldots \times X_{in(I)}$, если $in(I) > 0$, и $X_I = \{ \varepsilon \}$, в противном случае;

$Y_I = Y_1 \times \ldots \times Y_{out(I)}$, если $out(I) > 0$, и $Y_I = \{ \varepsilon \}$, в противном случае;

$V_I = V_1 \times \ldots \times V_{var(I)}$, если $var(I) > 0$, и $V_I = \{ \varepsilon \}$, в противном случае.

Спецификацией интерфейсной операции I с сигнатурой (In, Out, Var) называется пара предикатов (pre_I, $post_I$), в которой

- pre_I – предикат на множестве $V_I \times X_I$, называемый *предусловием*,
- $post_I$ – предикат на множестве $V_I \times X_I \times Y_I \times V_I$, называемый *постусловием*.

Спецификация интерфейсной операции утверждает, что если презначения переменных состояния и значения входных параметров удовлетворяют предусловию, то значения выходных параметров и постзначения переменных состояния должны удовлетворять постусловию для данных презначений переменных состояния и значений входных параметров.

Например, спецификацию операции вычисления квадратного корня (pre_{sqrt}, $post_{sqrt}$) можно задать следующим образом:

$$pre_{sqrt} (v, x_1) \equiv (x_1 \geq 0)$$

$$post_{sqrt} (v, x_1, y_1, v') \equiv | x_1 - y_1^2 | < \varepsilon_{погр}$$

Здесь, как и ранее, $\varepsilon_{погр}$ обозначает некоторое положительное действительное число.

Спецификацией целевой системы называется непустой набор спецификаций ее интерфейсных операций $\{ Spec_{Ii} \mid i = 1, \ldots, k; k > 0 \}$.

В технологии UniTesK для описания требований к функциональности целевой системы (модели требований) также используется подход программных контрактов. Описание модели требований задается в виде спецификации $Spec = \{ Spec_{Ii} \mid i = 1, \ldots, k \}$, а соответствующая ей модель требований **MR**[Spec] = (V, X, Y, E) определяется согласно следующим правилам:

1. Если в спецификации присутствует хотя бы одна переменная состояния, то множество состояний V является декартовым произведением множеств допустимых значений всех переменных состояния

$$V = \prod_{v \in Var_{Spec}} V_v ,$$

где Var_{Spec} является объединением переменных состояния всех интерфейсных операций, принадлежащих спецификации

$$Var_{Spec} = \bigcup_{i=1}^{k} Var_{I_i} .$$

2. Если в спецификации не участвует ни одной переменной состояния, то множество состояний состоит из единственного элемента ε:

$$V = \{ \varepsilon \}.$$

3. Множество стимулов X является дизъюнктивным объединением декартовых произведений множеств допустимых значений входных параметров всех интерфейсных операций спецификации

$$X = \coprod_{i=1}^{k} X_{I_i} .$$

Мы будем считать, что каждый элемент дизъюнктивного объединения помечается именем интерфейсной операции и таким образом эти элементы не пересекаются между собой.

4. Множество реакций Y является дизъюнктивным объединением декартовых произведений множеств допустимых значений выходных параметров всех интерфейсных операций спецификации

$$Y = \coprod_{i=1}^{k} Y_{I_i} .$$

Мы будем считать, что каждый элемент дизъюнктивного объединения помечается именем интерфейсной операции и таким образом эти элементы не пересекаются между собой.

5. Множество переходов E состоит из всех переходов, удовлетворяющих обобщенным предусловию и постусловию спецификации

$$E = \{ (v, x, y, v') \in V \times X \times Y \times V \mid pre(v, x) \wedge post(v, x, y, v') \},$$

где предикат $pre = \coprod_{i=1}^{k} pre_{I_i}$, а предикат $post = \coprod_{i=1}^{k} post_{I_i}$

2.2.6. Описание модели поведения

Требования к функциональности целевой системы являются исходными данными для процесса тестирования[2]. Соответственно, модель требований к целевой системе описывается во время разработки теста и в процессе тестирования не изменяется.

Поведение целевой системы в процессе тестирования не является статическим объектом. Оно появляется в результате взаимодействия целевой системы со своим окружением, и поэтому модель поведения целевой системы в процессе

[2] Здесь под процессом тестирования понимается процесс выполнения уже разработанного теста. Если под процессом тестирования понимать процесс разработки теста, то данное утверждение будет не совсем верно.

тестирования не может быть описана статическим образом. Только динамически, во время проведения тестирования, тестовая система наблюдает за поведением тестируемой системы и строит модель этого поведения.

Так как модель поведения используется для сравнения с моделью требований, то эти модели должны быть согласованы между собой. Это означает, что обе модели должны быть построены на основе единого интерфейса целевой системы (X, Y, V).

Если модель требований задана спецификацией Spec, то таким образом интерфейс целевой системы (X, Y, V) фиксирован согласно определению **MR**[Spec]. Состояние системы определяется значениями переменных состояния, описанных в спецификации, стимул задается интерфейсной операцией и набором значений входных параметров этой операции, а реакция идентифицируется интерфейсной операцией и набором значений выходных параметров этой операции.

В этом случае модель поведения целевой системы определяется в терминах заданной спецификации. Модель поведения состоит из набора отдельных, не связанных между собой взаимодействий. Каждое взаимодействие характеризуется:

- набором значений переменных до вызова интерфейсной функции;
- идентификатором интерфейсной операции и набором значений входных параметров этой операции;
- идентификатором интерфейсной операции и набором значений выходных параметров этой операции;
- набором значений переменных после вызова интерфейсной функции.

Все эти данные должны быть определены при построении модели поведения целевой системы. Каким образом это происходит, мы рассмотрим при обсуждении унифицированной архитектуры теста, но прежде мы обратимся к вопросу взаимосвязи моделей требований и поведения и их прообразам в реальном мире.

2.2.7. Моделирование требований и поведения

Вопрос построения моделей требований и поведения является очень непростым, как и всякий вопрос, связанный с переходом от неформальной сущности к формальной модели. Для целого ряда частных случаев можно представить правила и рекомендации по построению моделей. Тем не менее, в общем случае решение этой задачи во многом основывается на опыте и интуиции разработчика тестов.

Для иллюстрации взаимосвязи моделей требований и поведения с реальными объектами, рассмотрим один из вариантов построения этих моделей в том случае, когда целевая система представляет собой систему с прикладным программным интерфейсом.

Предположим, что целевая система предоставляет своим пользователям прикладной программный интерфейс, состоящий из n функций. Каждая

интерфейсная функция имеет набор входных и выходных параметров. Кроме того, вызов интерфейсных функций может изменять внутреннее состояние целевой системы или ее окружение, и таким образом влиять на результаты следующих вызовов интерфейсных функций.

Тогда для каждой функции прикладного интерфейса можно определить соответствующую ей интерфейсную операцию. Сигнатура интерфейсной операции определяется по следующим правилам[3]:

- входные параметры интерфейсной операции и их типы соответствуют входным параметрам функции прикладного интерфейса,
- выходные параметры интерфейсной операции и их типы соответствуют выходным параметрам функции прикладного интерфейса,
- переменные состояния соответствуют тем частям внутреннего состояния целевой системы и/или ее окружения, от которого зависит ожидаемое поведение функции прикладного интерфейса. Мы здесь использовали слово "ожидаемое", чтобы отметить, что в данном случае нас интересует поведение не конкретной реализации целевой системы, а поведение ожидаемое от нее пользователем.

Требования к поведению целевой системы после вызова функции прикладного интерфейса описывается в виде спецификации соответствующей интерфейсной операции. Таким образом, мы задаем модель требований к целевой системе посредством спецификации, состоящей из спецификаций отдельных интерфейсных операций.

Определив сигнатуры интерфейсных операций, участвующих в спецификации, мы тем самым определили интерфейс целевой системы (X, Y, V). Согласно **MR**[Spec] этот интерфейс определяется следующим образом:

- множество стимулов X состоит из вызовов функций прикладного интерфейса со всеми допустимыми наборами входных параметров,
- множество реакций Y состоит из всех возможных возвращаемых значений интерфейсных функций, где под возвращаемым значением подразумевается кортеж значений выходных параметров данной функции,
- множество состояний V состоит из значений, соответствующих различным значениям внутреннего состояния целевой системы и/или ее окружения. При этом учитываются только те значения, от которых зависит ожидаемое поведение хотя бы одной функции прикладного интерфейса.

[3] Данные правила приводятся только для целей иллюстрации, поэтому их определение дано на поверхностном уровне, не рассматриваются вопросы, касающиеся области их применимости, и т.д.

Взаимодействием с целевой системой в этом случае является четверка (v, x, y, v'), в которой:

- пресостояние v соответствует состоянию целевой системы и/или ее окружения до вызова функции прикладного интерфейса,
- стимул x соответствует вызову функции прикладного интерфейса с определенным набором значений входных параметров,
- реакция y соответствует набору значений выходных параметров, который вернула вызванная функция,
- постсостояние v' соответствует состоянию целевой системы и/или ее окружения, в котором она оказалась после данного вызова.

Соответственно, модель поведения целевой системы состоит из множества таких четверок и строится тестовой системой в ходе тестирования.

Рассмотрим данный подход на примере системы, реализующей функциональность целочисленного стека. Прикладной программный интерфейс этой системы состоит из трех функций:

- функция помещения числа в стек имеет один входной параметр – число, и не имеет выходных параметров,
- функция выборки элемента с вершины стека не имеет входных параметров и имеет один выходной параметр – число, находящееся на вершине стека,
- функция получения размера стека не имеет входных параметров и имеет один выходной параметр – число, равное текущему размеру стека.

Для определенности будем считать, что целые числа, участвующие в этом примере, заданы множеством **Int**.

Для каждой функции прикладного программного интерфейса определим соответствующую ей интерфейсную операцию.

Сигнатуру операции помещения числа в стек определим как ($\{ x \}, \{\}, \{ v_{stack} \}$), где тип входного параметра x – **Int**, а переменной состояния v_{stack} – **Int**-list. Спецификацию этой операции ($\mathrm{pre}_{\mathrm{push}}, \mathrm{post}_{\mathrm{push}}$) зададим следующими предикатами:

$$\mathrm{pre}_{\mathrm{push}} (v_{stack}, x) \equiv \textbf{true}$$

$$\mathrm{post}_{\mathrm{push}} (v_{stack}, x, v_{stack}') \equiv (v_{stack}' = \langle x \rangle \circ v_{stack})$$

Сигнатуру операции выборки элемента с вершины стека определим как ($\{\}, \{ y \}, \{ v_{stack} \}$), где тип выходного параметра y – **Int**, а переменной состояния v_{stack} – **Int**-list. Спецификацию этой операции ($\mathrm{pre}_{\mathrm{pop}}, \mathrm{post}_{\mathrm{pop}}$) зададим следующими предикатами:

$$\mathrm{pre}_{\mathrm{pop}} (v_{stack}, y) \equiv \mathrm{size}(v_{stack}) > 0$$

$$\mathrm{post}_{\mathrm{pop}} (v_{stack}, y, v_{stack}') \equiv (y = \mathrm{head}(v_{stack})) \wedge (v_{stack}' = \mathrm{tail}(v_{stack}))$$

Сигнатуру операции получения размера стека определим как ({}, { y }, { v_{stack} }), где тип выходного параметра y – **Int**, а переменной состояния v_{stack} – **Int**-list. Спецификацию этой операции (pre_{size}, $post_{size}$) зададим следующими предикатами:

$$pre_{size} (v_{stack}, y) \equiv \textbf{true}$$

$$post_{size} (v_{stack}, y, v_{stack}') \equiv (y = size(v_{stack})) \wedge (v_{stack}' = v_{stack})$$

Спецификации этих трех операций составляют спецификацию системы, реализующей функциональность целочисленного стека. На основе данной спецификации мы можем построить модель требований, как это определено в **MR**[Spec]. Наиболее важным для нас является интерфейс целевой системы, который определяется неявным образом при построении модели требований.

В данном случае интерфейс целевой системы есть (X_{stack}, Y_{stack}, V_{stack}), где[4]:

- X_{stack} = **Int** ∪ { `pop` } ∪ { `size` } – множество стимулов состоит из целых чисел, соответствующих вызову функции помещения числа в стек с данным числом в качестве параметра,
- элемента `pop`, соответствующего вызову функции выборки элемента с вершины стека,
- элемента `size`, соответствующего вызову функции получения размера стека,
- Y_{stack} = { `push` } X **Int** X **Int** – множество реакций состоит из элемента `push`, соответствующего отсутствующим выходным параметрам функции помещения числа в стек,
- целых чисел, соответствующих значению единственного выходного параметра функции выборки элемента с вершины стека,
- целых чисел, соответствующих значению единственного выходного параметра функции получения размера стека,
- V_{stack} = **Int**-list – множество состояний состоит из списков целых чисел, соответствующих текущему состоянию стека (первый элемент списка соответствует вершине стека).

Рассмотрим, как будет выглядеть модель поведения целевой системы для одного из частных случаев. Предположим, что в ходе тестирования мы вызвали целевую систему четыре раза:

- вызвали функцию помещения числа в стек с параметром 0, когда стек был пустым, и получили стек, содержащий число 0;
- вызвали функцию получения размера стека и получили 1, в качестве значения выходного параметра, значение стека не изменилось;

[4] Множества X_{stack}, Y_{stack} и V_{stack} определяются с точностью до изоморфизма.

- вызвали функцию выборки элемента с вершины стека и получили 0, в качестве значения выходного параметра, значение стека не изменилось;
- вызвали функцию получения размера стека и получили 1, в качестве значения выходного параметра, значение стека не изменилось.

Модель поведения для данного теста будет состоять из четырех взаимодействий:

{
(⟨ ⟩, 0, `push`, ⟨ 0 ⟩),
(⟨ 0 ⟩, `size`, 1, ⟨ 0 ⟩),
(⟨ 0 ⟩, `pop`, 0, ⟨ 0 ⟩),
(⟨ 0 ⟩, `size`, 1, ⟨ 0 ⟩)
}

Если описать эту модель в терминах спецификации целевой системы, то мы получим следующее:

{
(v_{stack} = ⟨ ⟩, `push(0)`, `push()`, v_{stack} = ⟨ 0 ⟩),
(v_{stack} = ⟨ 0 ⟩, `size()`, `size(1)`, v_{stack} = ⟨ 0 ⟩),
(v_{stack} = ⟨ 0 ⟩, `pop()`, `pop(0)`, v_{stack} = ⟨ 0 ⟩),
(v_{stack} = ⟨ 0 ⟩, `size()`, `size(1)`, v_{stack} = ⟨ 0 ⟩)
}

Пресостояние и постсостояние задается значением переменной состояния v_{stack}, стимул записывается в виде идентификатора интерфейсной операции со списком значений ее входных параметров, а реакция – в виде идентификатора интерфейсной операции со списком значений ее выходных параметров.

2.2.8. Модели требований и поведения в унифицированной архитектуре теста

Унифицированная архитектура теста определяет набор компонентов тестовой системы, область ответственности каждого компонента, а также основные правила взаимодействия этих компонентов. В данном разделе мы рассмотрим унифицированную архитектура теста, в той ее части, которая связана с оценкой корректности поведения целевой системы.

В технологии UniTesK поведение целевой системы рассматривается как набор отдельных, не связанных между собой взаимодействий. Соответственно, задача оценки корректности поведения целевой системы декомпозируется на частные задачи оценки корректности отдельных взаимодействий.

За оценку корректности взаимодействий отвечает специальный компонент тестовой системы – *оракул*. Этот компонент генерируется автоматически на

основе описания модели требований в виде спецификации целевой системы. Модель поведения целевой системы строится динамически и за это отвечает другой компонент тестовой системы – *медиатор*. Оракул и медиатор являются одними из основных элементов унифицированной архитектуры теста. Рассмотрим их подробнее.

Медиаторы отвечают в тестовой системе за всю работу с целевой системой. Основными задачами медиатора являются оказание тестового воздействия на целевую систему и построение модели поведения целевой системы. Эти задачи тесно связаны, так как в синхронном случае любое взаимодействие начинается с оказания тестового воздействия или стимула. Ответная реакция целевой системы является другой составляющей взаимодействия. Кроме этого, взаимодействие характеризуется пресостоянием и постсостоянием.

Унифицированная архитектура теста предполагает неявное построение пресостояния и постсостояния. Тестовая система содержит *модельное состояние*, в котором хранятся текущие значения переменных состояния, определенных в спецификации целевой системы. Соответственно, значения переменных состояния хранящиеся в модельном состоянии до оказания стимула считаются пресостоянием, а значения этих переменных после оказания стимула – постсостоянием. За синхронизацию модельного состояния с внутренним состоянием целевой системы отвечает медиатор.

Исходя из этого, работа медиатора строится по следующему алгоритму. Получив указание подать стимул $x \in X$, медиатор преобразует его в вызов интерфейсной функции целевой системы со значениями входных параметров, соответствующих данному стимулу. Затем медиатор получает ответную реакцию целевой системы, преобразуют ее в модельное представление $y \in Y$ и возвращает ее обратно. Кроме этого, медиатор синхронизирует модельное состояние с внутренним состоянием тестируемой системы после оказания тестового воздействия.

В некоторых случаях значения переменных модельного состояния могут быть вычислены на основе достоверных знаний о внутреннем состоянии тестируемой системы. Тогда тестовая система может во время тестирования не только проверять требования к внешнему поведению целевой системы, но и контролировать ее внутреннее состояние. Это позволяет упростить локализации ошибок, так как некорректное изменение внутреннего состояния одной интерфейсной операцией может проявиться на внешнем поведении целевой системы только после многих вызовов других операций.

Тестирование с использованием достоверных знаний о внутреннем состоянии целевой системы называется *тестированием с открытым состоянием*. В противном случае, тестирование называется *тестированием со скрытым состоянием*. Технология UniTesK поддерживает оба этих подхода.

При тестировании с открытым состоянием медиатор вычисляет модельное состояние непосредственно на основе знаний о внутреннем состоянии целевой системы. Если при тестировании системы, реализующей функциональность

целочисленного стека, медиатор читает текущее состояние стека, не используя тестируемые функции, то это пример тестирования с открытым состоянием.

Если же доступ к внутреннему состоянию невозможен, то модельное состояние вычисляется, исходя из предположения о корректном поведении целевой системы и дополнительных знаний о ее реализации. В примере со стеком медиатор может не иметь доступа к текущему состоянию стека и тогда он будет вынужден вычислять это состояние, основываясь на предположении, что целевая система работает корректно.

Если требования к целевой системе определяют единственное постсостояние, для данных пресостояния, стимула и реакции, то проблем с неопределенностью модельного состояния после оказания тестового воздействия не возникает. В противном случае, в медиаторах необходимо использовать дополнительные знания о конкретной реализации, чтобы правильно вычислить текущее модельное постсостояние.

Согласно унифицированной архитектуре теста основным потребителем сервисов, предоставляемых медиатором, является оракул. Именно он просит медиатор подать стимул x и получает от него реакцию y. Задача оракула заключается в оценке корректности отдельного взаимодействия с целевой системой относительно модели требований.

Технически работа оракулов организована следующим образом. Оракул запоминает все необходимые части текущего модельного состояния v и передает стимул x медиатору. Медиатор оказывает тестовое воздействие, моделируемое посредством стимула x, получает реакцию целевой системы, преобразует ее в модельное представление y и синхронизирует текущее модельное состояние с внутренним состоянием реализации. Оракул, зная сохраненное состояние v, стимул x, реакцию y и текущее состояние v', выносит вердикт о корректности данного взаимодействия.

Оракул генерируется автоматически на основе описания модели требований в терминах предусловий и постусловий интерфейсных операций. Медиатор описывается разработчиком тестов вручную или с помощью техники шаблонов.

Часть архитектуры тестовой системы, разработанной согласно технологии UniTesK, связанная с организацией взаимодействия между медиатором и оракулом представлена на Рис. 3. Здесь стрелками обозначены направления передачи данных между компонентами тестовой системы.

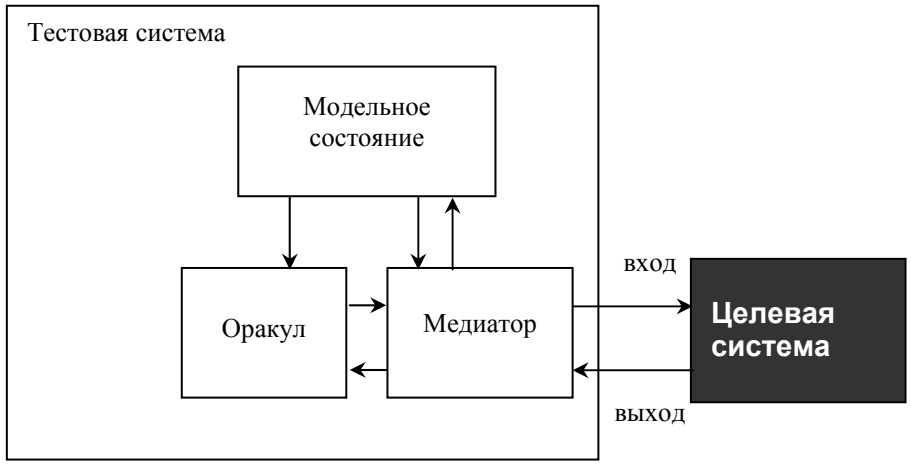

Рис. 3. Механизм работы оракула и медиатора

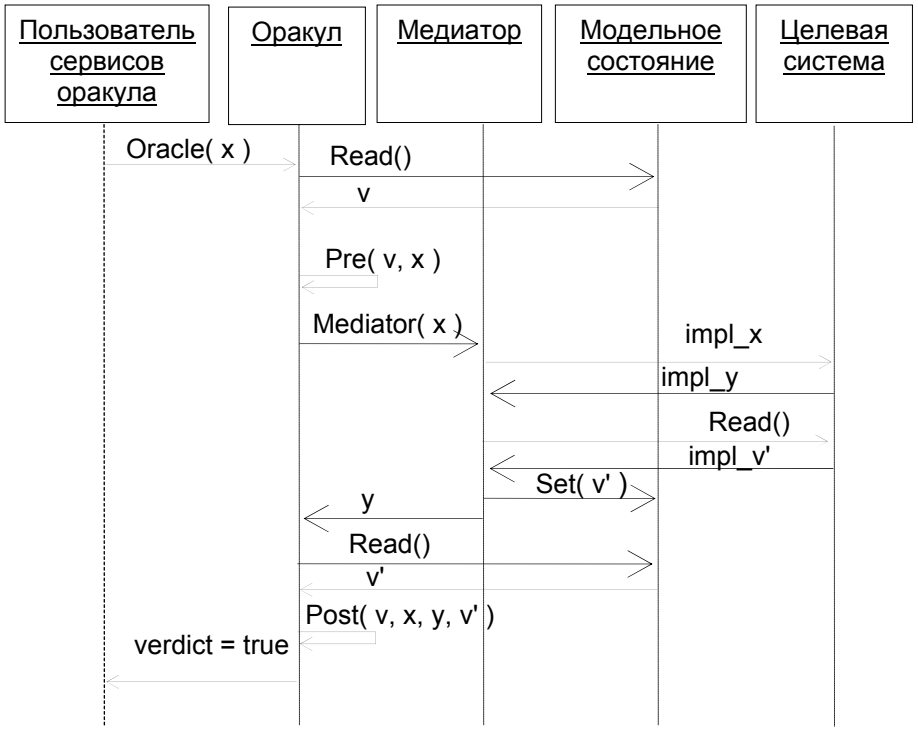

Рис. 4. Основной вариант работы оракула и медиатора при тестировании с открытым состоянием

На Рис. 4 и 5 представлены диаграммы взаимодействия UML, иллюстрирующие основные варианты работы этой части тестовой системы при тестировании с открытым и со скрытым состоянием соответственно. Работа оракула начинается с получения указания осуществить тестовое воздействие, моделируемое посредством стимула x. От кого именно поступает данное указание в настоящий момент не рассматривается. Дальнейшая последовательность взаимодействий оракула, медиатора и целевой системы происходит так, как рассматривалось ранее.

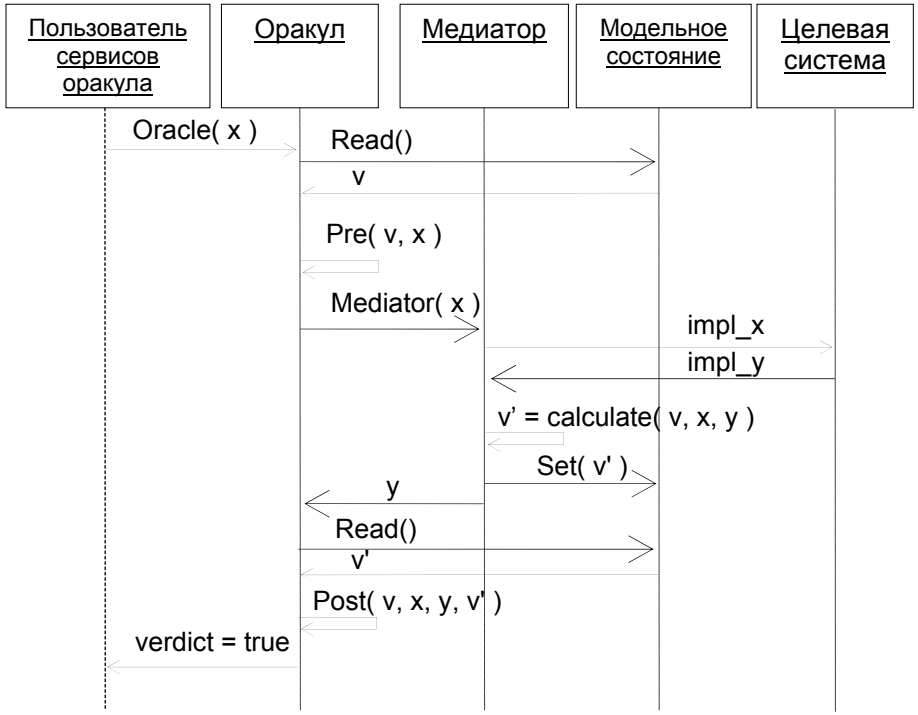

Рис. 5. Основной вариант работы оракула и медиатора при тестировании со скрытым состоянием

2.3. Генерация тестовых данных

В этом разделе мы остановимся на проблеме генерации тестовых данных. Предположим, что мы автоматизировали процесс оценки корректности поведения целевой системы, и для каждого взаимодействия с тестируемой системой можем автоматически вынести вердикт. Но как организовать эти взаимодействия?

Описывать их вручную – очень утомительно и накладно. Автоматически генерировать последовательности тестовых воздействий – возможно, но

качество, как правило, оказывается весьма невысоким. Технология UniTesK предлагает следующее компромиссное решение.

UniTesK разбивает задачу генерации тестовых воздействий на две подзадачи: генерацию семантически значимых значений отдельных типов данных, являющихся параметрами тестовых воздействий, и организацию сложных последовательностей взаимодействий, достигающих необходимый уровень покрытия.

Первая подзадача не может быть решена автоматически, так как в общем случае сводится к решению системы уравнений произвольной сложности. С другой стороны, для разработчика теста, как правило, не является большой проблемой указать множество семантически значимых значений определенного типа. Поэтому технология UniTesK перекладывает ответственность за решение этой задачи на разработчика, предоставляя ему существенную помощь в решении второй подзадачи.

Для организации сложных последовательностей тестовых воздействий на целевую систему в технологии UniTesK разработан специальный механизм построения тестового сценария. Этот механизм обеспечивает систематизированный обход заданного пространства тестовых ситуаций, посредством осуществления тестовых воздействий. При этом от разработчика тестов требуется только задание состояния сценария, согласованного с набором тестовых воздействий.

В основе механизма построения тестового сценария лежит граф. Состояния этого графа отражают различные "интересные" тестовые ситуации, а его дуги соответствуют заданным последовательностям тестовых воздействий. Тестовая система строит обход графа, осуществляя тестовые воздействия, приписанные к его дугам и, таким образом, решает задачу организации сложных последовательностей взаимодействий. Одним из вариантов построения графа является обобщение абстрактного автомата, описывающего требования, на основании наблюдения за поведением тестируемой системы. В этом случае граф тестового сценария является высокоуровневой моделью целевой системы, извлекаемой из наблюдения за ее поведением в процессе тестирования.

Формальные определения модели построения тестового сценария будут рассмотрены в последующих разделах.

2.3.1. Управляющие автоматы

Первым шагом в формализации понятия тестового сценария является определение управляющих автоматов.

Определение 3.

Управляющим автоматом будем называть пятерку (V, X, Y, E, Q), где

- S – множество состояний,
- X – множество стимулов,
- Y – множество реакций,
- $E \subseteq V \times X \times Y \times V$ – множество переходов,

- $Q \subseteq V$ – множество заключительных состояний.

Как следует из определения управляющий автомат является абстрактным автоматом, расширенным множеством заключительных состояний Q.

Стимул $s \in X$ называется *допустимым* в состоянии $u \in V$ абстрактного автомата A = (V, X, Y, E) (или управляющего автомата (V, X, Y, E, Q)), если существует переход (v, x, y, v') $\in$ E, такой что v = u и x = s.

Управляющий автомат называется *полностью определенным*, если в любом состоянии для любого допустимого в этом состоянии стимула и для любой возможной реакции существует переход с данными пресостоянием, стимулом и реакцией.

Управляющие автоматы предназначены для управления другими системами. Входные символы автомата (или стимулы) рассматриваются как действия над управляемой системой, а выходные символы (или реакции) – как ответная реакция управляемой системы. Часто, в качестве управляемой системы выступает абстрактный автомат, множества стимулов и реакций которого совпадают с соответствующими множествами управляющего автомата.

Управляющий автомат начинает свою работу в некотором начальном состоянии. Он выбирает один из допустимых в данном состоянии стимулов и подает его управляемой системе. Затем управляющий автомат получает ответную реакцию и переходит в состояние, которое приписано переходу с данными пресостоянием, стимулом и реакцией. Если таких состояний несколько, то новое состояние выбирается недетерминированным образом. Если новое состояние является заключительным, то работа управляющего автомата завершается. В противном случае, все вышеописанные действия повторяются в новом состоянии.

Последовательность переходов (e_1, e_2, …, e_n) абстрактного автомата A = (V, X, Y, E) (или управляющего автомата (V, X, Y, E, Q)) называется *путем*, если для каждого $i = 1, \ldots, n-1$ постсостояние перехода e_i совпадает с пресостоянием перехода e_{i+1}. При этом мы будем говорить, что путь ведет из пресостояния перехода e_1 в постсостояние перехода e_n.

Бесконечным путем мы будем называть бесконечную последовательность переходов, любой префикс которой является конечным путем.

Функционированием управляющего автомата (V, X, Y, E, Q) с начальным состоянием $v_0 \in V$ называется конечный или бесконечный путь $\{ e_i = (v_i, x_i, y_i, v'_i) \}$, удовлетворяющий следующим условиям:

- если начальное состояние является заключительным ($v_0 \in Q$), то путь является пустой последовательностью,
- если начальное состояние не является заключительным ($v_0 \notin Q$), то пресостояние первого перехода v_1 совпадает с начальным состоянием v_0, ($v_1 = v_0$),
- если путь является конечным и его длина больше нуля, то постсостояние последнего перехода является заключительным ($v_n \in Q$),

- если заключительное состояние присутствует в пути, то путь является конечным и это состояние есть постсостояние последнего перехода.

Управляющие автоматы будут использоваться далее для определения понятия тестового сценария.

2.3.2. Тестовый сценарий

Тестом целевой системы с интерфейсом (X, Y, V) называется конечная или бесконечная последовательность взаимодействий $\{ (v_i, x_i, y_i, v'_i) \}$. Множество всех тестов целевой системы с интерфейсом (X, Y, V) будем обозначать символом $\aleph(X, Y, V)$.

Моделью поведения целевой системы в ходе теста $\{ (v_i, x_i, y_i, v'_i) \}$ будем называть мультимножество взаимодействий $\{ (v_i, x_i, y_i, v'_i) \}$.

Тест $\{ (v_i, x_i, y_i, v'_i) \}$ целевой системы с интерфейсом (X, Y, V) будем называть *успешным* относительно модели требований A = (V, X, Y, E), если модель поведения в ходе теста удовлетворяет модели требований A. В противном случае, тест будет называться *неуспешным*.

Определение 4.

Тестовым сценарием для целевой системы с интерфейсом (X, Y, V) называется пара (C, s_0), где

- C – управляющий автомат (S, A, B, E, Q), в котором
- множество стимулов A = X X V,
- множество реакций B = (Y × V) X { ε },
- множество переходов включает в себя только переходы (s, a, b, s'), в которых либо a ∈ X и b ∈ (Y × V), либо a ∈ V и b = ε;
- $s_0 : V \rightarrow S$ – функция инициализации.

Тестовый сценарий управляет процессом тестирования, выполняя одно из двух возможных действий: либо подавая стимул x ∈ X, либо устанавливая новое состояние v ∈ V. Реакцией на первое действие является реакция целевой системы y и ее постсостояние v', а на второе – пустая реакция ε. Функция инициализации определяет начальное состояние управляющего автомата в зависимости от начального состояния целевой системы.

Множество всех тестовых сценариев для целевой системы с интерфейсом (X, Y, V) будем обозначать символом $\Im(X, Y, V)$. Множество тестовых сценариев для целевой системы с интерфейсом (X, Y, V) и с заданным множеством состояний управляющего автомата S будет обозначаться $\Im(X, Y, V, S)$.

Пусть $\{ e_j = (s_j, a_j, b_j, s'_j) \}$ является функционированием тестового сценария σ для целевой системы с интерфейсом (X, Y, V). Тогда $\{ e_{k(i)} = (s_{k(i)}, a_{k(i)}, b_{k(i)}, s'_{k(i)}) \}$ будем обозначать подпоследовательность последовательности $\{ e_j \}$ составленную из переходов, действия которых связаны с подачей стимула ($a_{k(i)} \in X$).

Результатом применения тестового сценария σ = (C = (S, A, B, E, Q), s_0) к целевой системе, находящейся в начальном состоянии $v_0 \in V$, является тест $\{ (v_i, x_i, y_i, v'_i) \}$, для которого существует такое функционирование управляющего автомата C с начальным состоянием $s_0(v_0)$ $\{ e_j = (s_j, a_j, b_j, s'_j) \}$, что длина теста совпадает с длиной подпоследовательности $\{ e_{k(i)} \}$ и для каждого взаимодействия (v_i, x_i, y_i, v'_i) выполнены следующие условия:

- пресостояние v_i совпадает с начальным состоянием целевой системы v_0, если *k(i)* = 1;
- ранее установленным состоянием $a_{k(i)-1}$, если *k(i)* > 1 и $a_{k(i)-1} \in V$;
- постсостоянием после предыдущей подачи стимула v_{i-1}, если *k(i)* > 1 и $a_{k(i)-1} \in X$;
- стимул x_i совпадает с $a_{k(i)}$;
- реакция y_i совпадает с первым элементом реакции $b_{k(i)} = (y_{k(i)}, v_{k(i)})$;
- постсостояние v'_i совпадает со вторым элементом реакции $b_{k(i)} = (y_{k(i)}, v_{k(i)})$.

Такой тест мы будем обозначать T(σ, v_0).

Тестовый сценарий определяет последовательность тестовых действий, которая может варьироваться в зависимости от поведения целевой системы в процессе тестирования. Результатом работы тестового сценария является тест, состоящий из последовательности взаимодействий с целевой системой.

Далее, мы введем вспомогательное определение тестового сценария с внутренними переходами, которое будет использоваться для описания тестовых сценариев в более компактной форме.

Тестовым сценарием с внутренними переходами для целевой системы с интерфейсом (X, Y, V) называется пара (C, s_0), где

- C – управляющий автомат (S, A, B, E, Q), в котором
- множество стимулов A = X X V X { ε },
- множество реакций B = (Y × V) X { ε },
- множество переходов включает в себя только переходы (s, a, b, s'), в которых либо a ∈ X и b ∈ (Y × V), либо a ∈ V X { ε } и b = ε;
- $s_0 : V \rightarrow S$ – функция инициализации.

Основной особенностью данного определения является дополнительный элемент множества стимулов ε, который обозначает внутренний переход управляющего автомата.

Каждому тестовому сценарию с внутренними переходами $\sigma_\varepsilon = ((S, A, B, E, Q), s_0)$ соответствует тестовый сценарий σ(σ_ε) = ((S, A', B, E', Q'), s_0), в котором

- множество стимулов A' = X X V,
- множество переходов E' состоит из переходов (s, a, b, s') ∈ S × A' × B × S, для которых существует путь $s_1 \xrightarrow{a_1,b_1} s2 \xrightarrow{a_2,b_2} \ldots$

$\xrightarrow{a_{n-1},b_{n-1}}$ s_n в управляющем автомате (S, A, B, E, Q), удовлетворяющий следующим условиям:

- начальное состояние пути $s_1 = s$,
- конечное состояние пути $s_n = s'$,
- $\exists\, i \in \{ 1, \ldots, n\text{-}1 \}$: $(a_i = a) \wedge (b_i = b) \wedge \forall\, j \in \{ 1, \ldots, n\text{-}1 \}\ \ (i \neq j) \Rightarrow (a_j = \varepsilon) \wedge (b_j = \varepsilon)$;
- множество заключительных состояний Q' состоит из состояний множества Q, а также из состояний $s \in S$, из которых не выходит ни одного перехода во множестве E', но существует путь $s_1 \xrightarrow{a_1,b_1} s2 \xrightarrow{a_2,b_2} \ldots \xrightarrow{a_{n-1},b_{n-1}} s_n$ в управляющем автомате (S, A, B, E, Q), в котором:
- начальное состояние $s_1 = s$,
- конечное состояние $s_n \in Q$,
- $\forall\, i \in \{ 1, \ldots, n \}\ \ (a_i = \varepsilon) \wedge (b_i = \varepsilon)$.

Определение 5.

Механизмом построения тестового сценария называется функция, значением которой является тестовый сценарий.

Механизмы построения тестового сценария применяются для создания тестовых сценариев на основе каких-либо данных. Примером такого механизма может быть последовательная композиция набора тестовых сценариев.

В технологии UniTesK реализовано несколько механизмов построения тестового сценария. Наиболее важный из них – автоматный механизм построения тестового сценария, который является настолько гибким и удобным, что именно он используется для построения подавляющего большинства тестовых сценариев.

2.3.3. Автоматный механизм построения тестового сценария

Ориентированным графом будем называть тройку G = (VG, XG, EG), в которой:

- VG – множество вершин графа;
- XG – множество стимулов графа;
- $EG \subseteq VG \times XG \times VG$ – множество дуг графа, каждая из которых состоит из начальной вершины, стимула и конечной вершины.

Ориентированный граф G = (VG, XG, EG)называется *конечным*, если множества VG, XG, и EG являются конечными.

Стимул $s \in XG$ называется *допустимым* в вершине $u \in VG$ ориентированного графа G = (VG, XG, EG), если существует дуга (v, x, v') $\in$ EG, такая что v = u и x = s.

Ориентированный граф G = (VG, XG, EG) называется *детерминированным*, если для каждой вершины $u \in VG$ и стимула $s \in XG$ существует не более одной дуги (v, x, v') $\in$ EG, такой что v = u и x = s.

Последовательность дуг (e_1, e_2, …, e_n) ориентированного графа G = (VG, XG, EG) называется *маршрутом*, если для каждого $i = 1, \ldots, n\text{-}1$ конечная вершина перехода e_i совпадает с начальной вершиной перехода e_{i+1}. При этом мы будем говорить, что маршрут ведет из начальной вершины перехода e_1 в конечную вершину перехода e_n.

Бесконечным маршрутом мы будем называть бесконечную последовательность переходов, любой префикс которой является конечным маршрутом.

Ориентированный граф G = (VG, XG, EG) называется *сильно-связным*, если для любых двух его вершин v и v' существует маршрут (e_1, e_2, …, e_n), ведущий из вершины v в вершину v'.

Обходом конечного ориентированного графа G = (VG, XG, EG) называется маршрут $v_1 \xrightarrow{x_1} v_2 \xrightarrow{x_2} \ldots \xrightarrow{x_{n-1}} v_n$, в котором для каждой дуги (v, x, v') $\in$ EG существует $i \in \{ 1, \ldots, n\text{-}1 \}$, такое что $v = v_i$, $x = x_i$ и $v' = v_{i+1}$.

Определение 6.

Алгоритмом движения на ориентированных графах называется функция α, которая по ориентированному графу G = (VG, XG, EG), текущей вершине $v \in VG$ и пройденному маршруту (e_1, e_2, …, e_n) определяет следующее действие $a \in XG \cup \{ \tau \}$. Следующее действие может быть либо стимулом x, допустимым в текущей вершине, либо специальным значением τ, символизирующим пустое действие.

Функционированием алгоритма движения α на ориентированном графе G = (VG, XG, EG) с начальной вершины $v_0 \in V$ называется конечный или бесконечный маршрут (e_1, e_2, …, e_n, …) в графе G, удовлетворяющий следующим условиям:

- начальная вершина v_1 первой дуги маршрута $e_1 = (v_1, x_1, v'_1)$ совпадает с начальной вершиной v_0;
- в каждом дуге маршрута $e_i = (v_i, x_i, v'_i)$ стимул x_i равен $\alpha(A, v_i, (e_1, \ldots, e_{i-1}))$ – значению алгоритма движения α на автомате A, начальной вершине v_i и пройденном маршруте (e_1, …, e_{i-1});
- если дуга $e_i = (v_i, x_i, v'_i)$ является последней в последовательности, то $\alpha(A, v'_i, (e_1, \ldots, e_i)) = \tau$.

Алгоритм движения α называется *алгоритмом обхода* на классе конечных ориентированных графов Æ, если на любом графе G из класса Æ любое функционирование алгоритма движения α является конечным обходом графа G.

Алгоритм движения α называется *неизбыточным*, если для любых двух конечных ориентированных графов $G_1 = (VG_1, XG_1, EG_1)$ и $G_2 = (VG_2, XG_2, EG_2)$, для любой вершины $v \in VG_1 \cap VG_2$ и для любого пройденного маршрута (e_1, …, e_n), из равенства множеств допустимых стимулов в текущей вершине и в каждой вершине из пройденного маршрута

следует равенство $\alpha(A_1, v, (e_1, \ldots, e_n)) = \alpha(A_2, v, (e_1, \ldots, e_n))$. Другими словами, алгоритм движения является неизбыточным, если он зависит только от текущей вершины, пройденного маршрута и множества допустимых стимулов в текущей вершине и в каждой вершине из пройденного маршрута.

Преимущество неизбыточных алгоритмов заключается в том, что для начала их работы не требуется полное описание ориентированного графа. Вся информация о вершинах и дугах графа, используемая неизбыточным алгоритмом, может быть извлечена в процессе движения по графу.

В работах [25,26,27] представлено исследование неизбыточных алгоритмов обхода различных классов графов. В частности, в [26] рассмотрен неизбыточный алгоритм обхода α_{dfsm}[5] на классе детерминированных сильно-связных конечных ориентированных графах. Этот алгоритм используется в технологии UniTesK в качестве основного алгоритма для построения последовательностей тестовых воздействий.

Неизбыточным описанием ориентированного графа называется тройка (VG, XG, π), где

- VG – множество вершин графа,
- XG – множество стимулов графа,
- $\pi : VG \to \wp(XG)$ – функция, определяющая множество допустимых стимулов в данной вершине графа.

Неизбыточное описание ориентированного графа содержит все необходимое для работы неизбыточного алгоритма движения. С другой стороны, для разработчика теста значительно проще создать такое описание, чем описывать граф в явном виде. Поэтому в технологии UniTesK неизбыточное описание графа нашло свое применение при определении тестовых сценариев.

Автоматным тестовым сценарием для целевой системы с интерфейсом (X, Y, V) называется семерка $(IG, vg_0, \alpha, S, \rho, \gamma, \eta)$, где

- $IG = (VG, XG, \pi)$ – неизбыточное описание ориентированного графа, называемого *графом сценария*,
- $vg_0 : V \to VG$ – функция инициализации графа сценария,
- α – неизбыточный алгоритм движения по графу сценария,
- S – множество состояний сценария,
- $\rho : S \to S$ – функция рестарта сценария,
- $\gamma : VG \times XG \to \Im(X, Y, V, S)$ – отображение, ставящее в соответствие каждым вершине графа сценария и стимулу, допустимому в этой вершине, тестовый сценарий, который мы будем называть *сценарным воздействием*,
- $\eta : VG \times XG \times V \times S \to VG$ – отображение, ставящее в соответствие конечную вершину дуги графа сценария для данных начальной вершины и стимула графа сценария и состояний

[5] В указанной работе этот алгоритм обозначается символом $\mathbb{A}_2$.

сценария и целевой системы после выполнения сценарного воздействия.

Автоматным механизмом построения тестового сценария называется функция, преобразующая автоматный тестовый сценарий $(IG, vg_0, \alpha, S, \rho, \gamma, \eta)$ в тестовый сценарий, посредством вспомогательного определения тестового сценария с внутренними переходами $\sigma_\varepsilon = ((S', A, B, E, Q), s_0)$ по следующим правилам:

- $S' = VG \times S \times \{ \varepsilon \} \times V \times EG^*$ – состояние тестового сценария состоит из:
- текущей вершины графа сценария vg,
- текущего состояния сценария s, которое может принимать дополнительное значение ε, которое означает неинициализированное состояние,
- текущего состояния целевой системы v,
- пройденного маршрута в графе сценария path. Здесь множество EG^* обозначает множество всех конечных списков элементов из множества дуг $EG = VG \times XG \times VG$.
- множество стимулов $A = X \times V \times X \times \{ \varepsilon \}$, согласно определению тестового сценария с внутренними переходами;
- множество реакций $B = (Y \times V) \times \{ \varepsilon \}$, согласно определению тестового сценария с внутренними переходами;
- $E = \{ ((vg, s, v, path), a, b, (vg', s', v', path')) \in S' \times A \times B \times S' \mid$
  $(\alpha(IG, vg, path) \neq \tau) \wedge (v' = state(a, b, v)) \wedge (s \notin \gamma_Q(vg, xg)$
  $\Rightarrow (vg' = vg) \wedge (path' = path) \wedge (s, a, b, s') \in \gamma_E(vg, xg))$
  $\wedge (s \in \gamma_Q(vg, xg) \Rightarrow (vg' = \eta(vg, xg, v, s))$
  $\wedge (path' = path \wedge (vg, xg, vg')) \wedge (a = \varepsilon) \wedge (b = \varepsilon) \wedge (s' = \rho(s))$
  $)$
  $\}$, где использовались следующие сокращения
- $xg \equiv \alpha(IG, vg, path)$;
- $state(a, b, v) = a$, если $a \in V$;
  $state(a, b, v) = v_b$, если $b = (y_b, v_b) \in Y \times V$;
  $state(a, b, v) = v$, иначе;
- $Q = \{ (vg, s, v_0, path) \in S' \mid \alpha(IG, vg, path) = \tau \}$ – заключительными состояния сценария являются состояния, в которых алгоритм движения возвращает пустое действие τ;
- $s_0(v_0) = (vg_0(v_0), \gamma_{S0}(vg, \alpha(IG, vg_0(v_0), \langle\rangle))(v_0), v_0, \langle\rangle)$, если $\alpha(IG, vg_0(v_0), \langle\rangle) \neq \tau$,
  $s_0(v_0) = (vg_0(v_0), \varepsilon, v_0, \langle\rangle)$, если $\alpha(IG, vs_0(v_0), \langle\rangle) = \tau$,
  – инициализирующая функция, которая устанавливает следующие начальные значения:
- текущая вершина графа сценария определяется при помощи функции инициализации графа сценария;
- текущее состояние сценария s принимает

- значение, вычисленное посредством инициализирующей функции первого сценарного воздействия, если таковое найдено при помощи алгоритм движения α,
- неинициализированное значение, в противном случае;
- текущее состояние целевой системы инициализируется параметром функции,
- пройденный маршрут инициализируется пустым списком.

Далее для построения тестового сценария автоматный механизм использует определенное ранее преобразование тестового сценария с внутренними переходами в тестовый сценарий без внутренних переходов.

Работа автоматного механизма построения тестового сценария устроена следующим образом. Граф сценария описывает некоторым образом пространство тестовых ситуаций. Автоматный механизм двигается по данному графу, используя заданный алгоритм движения.

Алгоритм движения выбирает стимул, который необходимо подать в текущей вершине. Автоматный механизм определяет тестовый сценарий, который приписан паре вершина-стимул графа сценария, и выполняет его. По результатам выполнения механизм вычисляет новую вершину графа сценария и повторяет этот цикл до тех пор, пока алгоритм движения не завершит свою работу.

Все тестовые сценарии, приписанные к графу сценария, разделяют общее состояние, которое позволяет им накапливать информацию о процессе тестирования. После завершения каждого сценария автоматный механизм обновляет это состояние при помощи функции рестарта сценария. Благодаря этому, один и тот же тестовый сценарий может успешно применяться несколько раз подряд.

Пара отображений γ и η определяет дуги графа сценария, основываясь на поведении целевой системы. Таким образом, граф сценария строится динамически посредством применения этих отображений. В него входят те дуги, которые были "пройдены" в процессе тестирования.

2.3.4. Сценарные функции

В данном разделе мы рассмотрим метод описания тестовых сценариев, приписанных к дугам графа сценария. В основе метода лежит понятие сценарной функции, которое представляет собой параметризованное семейство тестовых сценариев.

Сценарной функцией для целевой системы с интерфейсом (X, Y, V) называется шестерка (GS, PS, IS, C, is_0, π), где:

- GS – множество внешних состояний;
- PS – множество параметризующих состояний;
- IS – множество внутренних состояний;
- C = (CS, A, B, E, Q) – управляющий автомат, в котором:

- $CS = GS \times PS \times IS$ – состояние автомата состоит из декартова произведения всех видов состояний сценарной функции,
- $A = X \; X \; V$,
- $B = (Y \times V) \; X \; \{ \varepsilon \}$,
- $E \subseteq CS \times A \times B \times CS$,
- $Q \subseteq CS$;
- $is_0 : V \times GS \to IS$ – функция инициализации сценарной функции;
- $\pi : V \times GS \to \wp(PS)$ – функция, определяющая множество параметризующих состояний для данных внешнего состояния и состояния целевой системы. Это множество мы будем называть *множеством параметров*.

Множество всех сценарных функций для целевой системы с интерфейсом (X, Y, V) и имеющее множество внешних состояний GS мы будем обозначать $\Sigma(X, Y, V, GS)$.

Автоматным тестовым сценарием со сценарными функциями для целевой системы с интерфейсом (X, Y, V) называется шестерка (GS, gs_0, VG, vg, α, $\{ \Sigma_i \}_{i=1}^{k}$), где:

- GS – множество глобальных состояний сценария;
- $gs_0 : V \to GS$ – функция инициализации глобального состояния;
- VG – множество вершин графа автомата;
- $vg : V \times GS \to VG$ – функция вычисления вершины графа сценария;
- α – неизбыточный алгоритм движения по графу сценария,
- $\{ \Sigma_i \}_{i=1}^{k}$ – конечный набор сценарных функций $\Sigma_i \in \Sigma(X, Y, V, GS)$.

Множества параметров всех сценарных функций должны быть согласованы с вершинами графа сценария:

$$\forall i \in \{ 1, \ldots, k \} \; \forall v_1, v_2 \in V \; \forall gs_1, gs_2 \in GS$$

$$vg(v_1, gs_1) = vg(v_2, gs_2) \Rightarrow \pi_{\Sigma i}(v_1, gs_1) = \pi_{\Sigma i}(v_2, gs_2).$$

Другими словами, для любой пары значений состояния целевой системы и глобального состояния сценария, отображающейся в одну вершину графа сценария, множества параметров всех сценарных функций должны совпадать.

Исходя из этого ограничения, можно определить множество параметров сценарной функции Σ_i в данной вершине графа vg как:

- $\pi\Sigma_i(v, gs)$, если $\exists v \in V \; \exists gs \in GS : \; vg(v, gs) = vg$;
- $\{\}$, иначе.

В автоматном тестовом сценарии со сценарными функциями набор сценарных функций используется не только для описания тестовых сценариев приписанных к дугам графа, но и для описания множества стимулов графа. Далее, мы определим семантику автоматных тестовых сценариев со сценарными функциями посредством описания механизма преобразования таких сценариев в тестовый сценарий.

Автоматным механизмом сценарных функций называется функция, преобразующая автоматный тестовый сценарий со сценарными функциями (GS, gs_0, VG, vg, α, $\{ \Sigma_i \}_{i=1}^{k}$) в тестовый сценарий, посредством применения автоматного механизма построения тестового сценария к автоматному тестовому сценарию (IG', vg_0', α', S', ρ', γ', η'), определенному по следующим правилам:

- неизбыточное описание ориентированного графа IG' = (VG', XG', π') состоит из:
  - множества вершин, совпадающего с множеством вершин графа автомата VG (VG' = VG),
  - множества стимулов, являющегося дизъюнктивным объединением параметризующих состояний всех сценарных функций (XG' = $\bigvee_{i=0}^{k} PS_i$),
  - функции, определяющей множество допустимых стимулов $\pi' = \bigvee_{i=0}^{k} \pi_{\Sigma_i}(vg)$;
- функция инициализации графа сценария vg_0': $vg_0'(v) \equiv vg(v, gs_0(v))$;
- неизбыточный алгоритм $\alpha' = \alpha$;
- множество состояний сценария S' = GS × V × State, где State = $\bigvee_{i=0}^{k} (PS_i \times IS_i)$ X {ε};
- функция рестарта сценария ρ': $\rho'((gs,v,state)) \equiv (gs, v, \varepsilon)$;
- сценарное воздействие $\gamma'(vg, ps_i) = ((S', A', B', E', Q'), s'_0) \in \Im(X, Y, V, S')$ определяется для каждой пары (vg, ps_i) следующим образом:
  - A' = X X V,
  - B' = (Y × V) X { ε },
  - E' = { ((gs, v, state), a, b, (gs', (ps'_i, v', is'_i))) :
    ((state = ε) ⇒
    ((gs, ps_i, is_i0(v, gs)), a, b, (gs', ps'_i, is'_i)) ∈ E_i
)
    ∧ ((∃ is_i ∈ IS_i state = (ps_i, is_i)) ⇒
    ((gs, ps_i, is_i), a, b, (gs', ps'_i, is'_i)) ∈ E_i
)
    ∧ (a ∈V) ⇒ v' = a
    ∧ (b = (y_b, v_b) ∈ Y × V) ⇒ v' = v_b
    },
  - Q' = { (gs, v, state) ∈ S' :
    ((state = ε) ⇒
    (gs, ps_i, is_i0(v, gs)) ∈ Q_i
)
    ∧ ((∃ is_i ∈ IS_i state = (ps_i, is_i)) ⇒
    (gs, ps_i, is_i) ∈ Q_i
)
    },
  - $s'_0(v_0) = (gs_0(v_0), v_0, \varepsilon)$;
- отображение $\eta'(vg_1, ps_i, v, (gs, v2, state)) = vg(v, gs)$.

Таким образом, автоматный тестовый сценарий со сценарными функциями определяет тестовый сценарий, являющийся последовательной композицией тестовых сценариев приписанных к дугам графа сценария. Правила выбора последовательности дуг графа определяются алгоритмом движения, который учитывает при этом поведение целевой системы.

2.3.5. Граф автоматного тестового сценария

Неизбыточное описание графа сценария не является описанием единственного графа. Оно только определяет набор элементов, из которых можно построить граф. Фактический граф определяется в процессе функционирования автоматного тестового сценария.

Предположим, что σ_{aut} = (IG, vg_0, α, S, ρ, γ, η) – автоматный тестовый сценарий для целевой системы с интерфейсом (X, Y, V). Предположим, что σ = (C, s_0) – тестовый сценарий, полученный посредством применения автоматного механизма построения тестового сценария к σ_{aut}. Тогда *функционированиями автоматного тестового сценария* σ_{aut} на целевой системе с начальным состоянием $v_0 \in V$ будем называть все функционирования { e_j = (s_j, a_j, b_j, s'_j) } управляющего автомата C с начальным состоянием $s_0(v_0)$.

Заметим, что согласно определению автоматного механизма построения тестового сценария состояниями управляющего автомата s_j являются четверки (vg_j, s_j, v_j, $path_j$) ∈ VG × S X { ε } × V × EG^*.

Графом автоматного тестового сценария σ_{aut} = (IG = (VG, XG, π), vg_0, α, S, ρ, γ, η) при функционировании σ_{aut} { e_j = (s_j, a_j, b_j, s'_j) } называется ориентированный граф G' = (VG', XG', EG'), в котором:

- множество вершин VG' = VG;
- множество стимулов XG' = XG;
- множество дуг EG' = { eg ∈ VG × XG × VG : eg ∈ $\bigvee_j elems(path_j)$ },
  где под *elems*($path_j$) понимается множество всех элементов списка $path_j$.

Лемма.

Граф автоматного тестового сценария (IG, vg_0, α, S, ρ, γ, η) при любом функционировании $\{ e_j = (s_j, a_j, b_j, s'_j) \}$ удовлетворяет неизбыточному описанию IG.

Действительно. Граф состоит из дуг пути в графе, пройденного неизбыточным алгоритмом движения по графу, который выбирает стимулы только из множества, допустимых в текущем состоянии.

2.3.6. Механизм построения тестового сценария dfsm

Основным механизмом построения тестовых сценариев в технологии UniTesK является механизм построения тестового сценария dfsm. Этот механизм основан на неизбыточном алгоритме обхода на классе детерминированных сильно-связных конечных ориентированных графов α_{dfsm}, представленном в работе [26].

Тестовым сценарием dfsm для целевой системы с интерфейсом (X, Y, V) называется автоматный тестовый сценарий со сценарными функциями, в котором в качестве алгоритма движения по графу сценария используется алгоритм обхода α_{dfsm}.

Механизмом построения тестового сценария dfsm называется функция, преобразующая тестовый сценарий dfsm в тестовый сценарий посредством применения автоматного механизма сценарных функций.

Как показано в [26], любое конечное функционирование тестового сценария dfsm приводит

- либо к обнаружению нарушения требований детерминированности или сильно-связности графа сценария,
- либо к построению обхода этого графа.

С другой стороны, бесконечное функционирование тестового сценария dfsm возможно только в случае бесконечности графа сценария, или в случае бесконечного функционирования одной из сценарных функций.

Таким образом, если поведение целевой системы удовлетворяет требованиям, то при выполнении следующих условий

- граф сценария при любом корректном функционировании целевой системы является детерминированным и сильно-связным;
- всякая сценарная функция при любом корректном функционировании целевой системы завершается за конечное число шагов;

тестовый сценарий dfsm завершает свою работу и путь, пройденный по графу сценария, является обходом этого графа.

2.3.7. Тестовый сценарий в унифицированной архитектуре теста

В данном разделе мы вернемся к рассмотрению унифицированной архитектуры теста. В первую очередь, нас будет интересовать место тестового сценария и координация его работы с другими компонентами теста.

Тестовый сценарий представляет собой управляющий автомат, начальное состояние которого определяется на основе начального состояния целевой системы. В процессе функционирования тестовый сценарий взаимодействует с целевой системой двумя способами.

Первый вид взаимодействия соответствует подаче стимула $x \in X$ и получению реакции $y \in Y$ и постсостояния целевой системы $v' \in V$. Для осуществление такого взаимодействия тестовый сценарий обращается к оракулу, передавая ему стимул $x \in X$. Далее работа происходит по описанной ранее схеме. Оракул запоминает все необходимые части текущего модельного состояния v и передает стимул x медиатору. Медиатор оказывает тестовое воздействие, моделируемое посредством стимула x, получает реакцию целевой системы, преобразует ее в модельное представление y и синхронизирует текущее модельное состояние с внутренним состоянием реализации. Оракул, зная сохраненное состояние v, стимул x, реакцию y и текущее состояние v', выносит вердикт о корректности данного взаимодействия.

В качестве результата взаимодействия тестовый сценарий получает реакцию $y \in Y$ и постсостояния целевой системы $v' \in V$. Последнее передается неявным образом через модельное состояние. Тестовый сценарий, получив эти данные, обновляет свое состояние и принимает решение о следующем взаимодействии.

Ситуация, когда оракул выносит отрицательный вердикт о корректности данного взаимодействия, обрабатывается согласно настройкам тестового сценария. Обычно тестирование прекращается при получении некоторого константного числа отрицательных вердиктов.

Второй вид взаимодействия соответствует подаче стимула $v \in V$ и “пустой” реакции целевой системы ε. В этом случае, тестовый сценарий обращается непосредственно к медиатору с указанием перевести целевую систему в состояние, соответствующее модельному состоянию $v \in V$. Медиатор выполняет данное указание и соответствующим образом обновляет модельное состояние.

Считается, что медиатор может успешно выполнить данное указание в любых условиях. При нарушении этого предположения тестовый сценарий не может продолжать свою работу и тестирование аварийно завершается.

Важно отметить, что тестовый сценарий может успешно обходиться только взаимодействиями первого вида. Это позволяет использовать технологию UniTesK для тестирования методом черного ящика, при отсутствии какого-либо доступа к внутреннему состоянию целевой системы. В таких случаях, автоматные тестовые сценарии демонстрируют все свои достоинства, так как они предоставляют возможность неявным образом итерировать значения внутреннего состояния целевой системы тогда, когда сделать это явно не представляется возможным.

С другой стороны, наличие возможности явной установки состояния позволяет технологии UniTesK использовать все преимущества белого тестирования, в тех случаях, когда это представляется необходимым.

Унифицированная архитектура теста, дополненная тестовым сценарием, представлена на Рис. 6. Здесь стрелками обозначены направления передачи данных между компонентами тестовой системы.

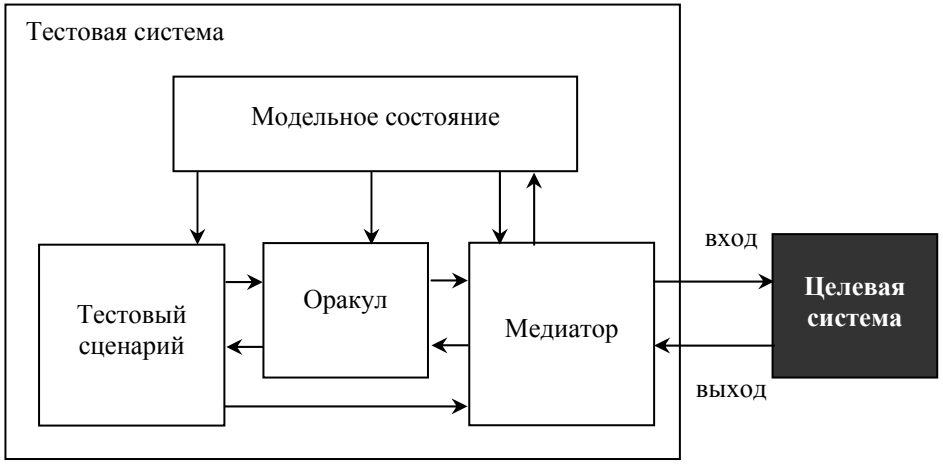

Рис. 6. Тестовый сценарий в универсальной архитектуре теста

Работа автоматного тестового сценария в случае тестирования со скрытым состоянием представлена на Рис. 7.

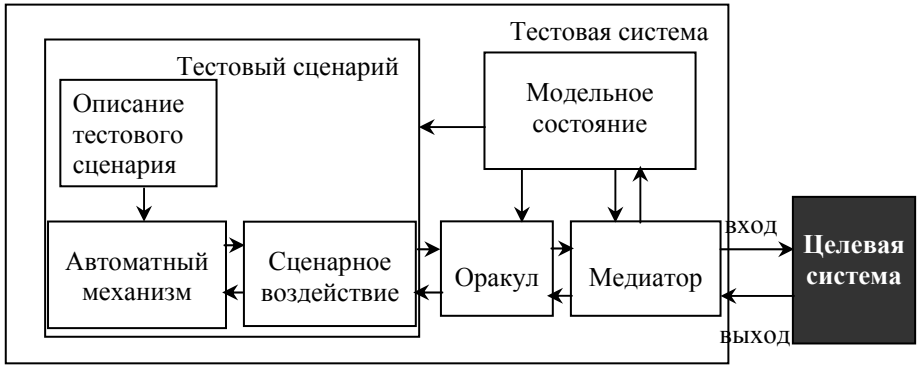

Рис. 7. Автоматный тестовый сценарий в универсальной архитектуре теста

Автоматный механизм на основе описания тестового сценария выбирает очередное сценарное воздействие и инициирует его выполнение. Сценарное воздействие, которое является «компактным» тестовым сценарием, управляет

последовательностью взаимодействий с целевой системой. Эти взаимодействия осуществляются посредством обращения к оракулу, который динамически оценивает корректность поведения целевой системы в рамках данного взаимодействия.

По завершению работы сценарного воздействия автоматный механизм выбирает следующее сценарное воздействие или принимает решение о завершении тестирования. В первом случае инициируется выполнение очередного сценарного воздействия и всё описанное повторяется.

При использовании автоматных тестовых сценариев со сценарными функциями, и в частности тестовых сценариев dfsm, сценарное воздействие описывается при помощи сценарных функций. Таким образом, выполнение сценарного воздействия является выполнением одной из сценарных функций с определенным значением ее параметризующего состояния.

2.4. Оценка качества тестирования

Мы рассмотрели, как работает тестовая система в масштабе одного взаимодействия с целевой системой и как организуется последовательность таких взаимодействий. При этом в тени осталась одна очень важная проблема, имеющая место в любом процессе тестирования – проблема оценка качества тестирования.

В подавляющем большинстве случаев проверить целевую систему на всех допустимых тестовых данных невозможно. Поэтому приходится ограничиваться некоторым конечным набором тестов. В такой ситуации неизбежно встает вопрос о качестве тестирования, проведенного посредством данного тестового набора.

Существует несколько подходов к решению этой задачи. Наиболее распространенный из них использует оценку качества на основе исходного кода целевой системы. Например, оценивается процент инструкций выполненных во время тестирования или процент покрытия возможных ветвлений потока управления.

Этот подход доказал свою важность и значимость. Однако, использование только оценки покрытия исходного кода применительно к функциональному тестированию часто оказывается недостаточным. Например, стопроцентное покрытие тестами исходного кода не гарантирует, что все требования к функциональности системы были надлежащим образом реализованы. Если какое-то требование было упущено и не нашло своего отражения в исходном коде, то тесты могут показывать идеальный процент покрытия, хотя в системе присутствует серьезный изъян.

С другой стороны, функциональное тестирование имеет своей целью проверить выполнение целевой системой требований, предъявляемых к ее функциональности. Поэтому измерение качества тестов, сформулированное в терминах этих требований, также является необходимым для получения адекватной оценки. Технология UniTesK предлагает именно такой подход,

измерение качества тестирования в котором основывается на требованиях к функциональности системы.

Модель требований представляет собой абстрактный автомат A = (V, X, Y, E). Для оценки покрытия при тестировании конечных автоматов традиционно используются оценки числа покрытых состояний или переходов по сравнению с их общим числом. Но так как абстрактный автомат, как правило, является бесконечным, то такие методы оценки покрытия оказываются неприемлемыми. Для решения этой проблемы в технологии UniTesK используются метрики покрытия модели требований.

2.4.1. Метрики покрытия модели требований

Метрикой покрытия модели требований A = (V, X, Y, E) называется конечное множество подмножеств переходов модели требований $M \subseteq 2^E$. *Элементами покрытия* называются элементы метрики M, являющиеся подмножествами E.

Будем говорить, что тест $\{ (v_i, x_i, y_i, v'_i) \}$ *покрыл* элемент покрытия $C_j \in M$, если тест содержит хотя бы одно взаимодействие (v_i, x_i, y_i, v'_i) входящее во множество C_j. *Покрытием* метрики M тестом $\{ (v_i, x_i, y_i, v'_i) \}$ будем называть набор элементов метрики M, покрытых данным тестом.

Таким образом, метрика покрытия определяет конечный набор элементов, в терминах которых осуществляется оценка качества тестирования. В технологии UniTesK некоторые метрики генерируются автоматически на основе структуры постусловий интерфейсных функций. В дополнение к этим метрикам, разработчик тестов может определять свои собственные метрики покрытия. Тестовая система автоматически оценивает качество тестирования в терминах всех доступных метрик и по результатам теста генерирует отчеты о покрытии элементов каждой метрики.

Метрика покрытия M называется *управляемой*, если для любых двух взаимодействий $I_1 = (v_1, x_1, y_1, v'_1)$ и $I_2 = (v_2, x_2, y_2, v'_2)$ выполнено следующее утверждение:

$$(v_1 = v_2) \wedge (x_1 = x_2) \Rightarrow (\forall C_i \in M \ (I_1 \in C_i) \Leftrightarrow (I_2 \in C_i))$$

Для управляемых метрик характерно то, что принадлежность взаимодействия к тому или иному элементу покрытия зависит только от той части взаимодействия, которая управляется тестовой системой, то есть от пресостояния и стимула. Работая с управляемыми метриками, тестовая система может не только автоматически подсчитывать их покрытие, но и оптимизировать генерацию тестовых данных для достижения определенного покрытия по некоторой управляемой метрике.

2.4.2. Описание метрик покрытия

Метрики покрытия в технологии UniTesK описываются совместно с описанием предусловий и постусловий интерфейсных операций целевой системы. Это позволяет локализовать описание требований и выделение элементов покрытия на основе этих требований. Элементы покрытия, как правило, соответствуют

ветвям функциональности в поведении интерфейсной операции или ситуациям, интересным с точки зрения тестирования, таким как граничные условия.

Определение метрик только для интерфейсных операций не позволяет описывать произвольные метрики покрытия модели требований. Но практический опыт показал, что имеющихся возможностей более чем достаточно для измерения качества тестирования приложений любых типов.

Определим способ описания метрик формально. Предположим, что в целевой системе выделено k интерфейсных операций: $I = \{ I_i \mid i = 1, \ldots, k \}$ и требования к ее поведению были описаны в спецификации $Spec = \{ Spec_{Ii} \mid i = 1, \ldots, k \}$.

Тогда сюръективная функция $\mu : V_I \times X_I \times Y_I \times V_I \to R$ называется *метрикой покрытия интерфейсной операции* I с сигнатурой (In, Out, Var)[6], если множество R является конечным. Каждой метрике покрытия интерфейсной операции μ соответствует метрика покрытия модели требований **MR**[Spec], которую мы будем обозначать **M**[μ].

$$\mathbf{M}[\mu] = \{ C_r \in 2^E \mid r \in R \},$$
где $C_r = \{ (v, x, y, v') \in V \times X \times Y \times V \mid \mu(v, x, y, v') = r \}$.

Лемма.

Метрика покрытия **M**[μ] является управляемой тогда и только тогда, когда третий и четвертый аргументы функции μ являются несущественными.

Кроме того, в технологии UniTesK поддерживается другой вариант описания метрик покрытия. В этом варианте каждый элемент покрытия описывается предикатом, определяющим принадлежность взаимодействия через определенную интерфейсную операцию (v, x, y, v') к данному элементу покрытия. Такой способ описания позволяет задавать метрики покрытия модели требований, элементы которых могут пересекаться между собой. Но, как и в первом способе, существуют ограничения на описываемые метрики, связанные с привязкой каждой метрики только к одной интерфейсной операции.

Конечный набор предикатов $\mu' = \{ c_i \mid i = 1, \ldots, m; \ c_i : V_I \times X_I \times Y_I \times V_I \to Bool \}$ называется *расширенной метрикой покрытия интерфейсной операции* I с сигнатурой (In, Out, Var). Каждой расширенной метрике покрытия интерфейсной операции μ' соответствует метрика покрытия модели требований **MR**[Spec], которую мы будем обозначать **M**[μ'].

$$\mathbf{M}[\mu'] = \{ C_i \in 2^E \mid i = 1, \ldots, m \},$$
где $C_i = \{ (v, x, y, v') \in V \times X \times Y \times V \mid c_i(v, x, y, v') \}$.

Лемма.

Метрика покрытия **M**[μ'] является управляемой тогда и только тогда, когда третий и четвертый аргументы всех предикатов c_i являются несущественными.

[6] Определение обозначений X_I, Y_I и V_I смотрите на стр. 12.

2.4.3. Метрики покрытия в унифицированной архитектуре теста

В процессе тестирования, при вызове интерфейсной операции, тестовая система вычисляет для каждой метрики, заданной для данной операции, все элементы покрытия, покрытые данным вызовом. Информация о покрытых элементах попадает в *трассу теста*. По трассе одного или нескольких тестов генерируются отчеты о покрытии интерфейсных операций, согласно определенным для них метрикам покрытия.

Компонент тестовой системы, который отвечает за вычисление покрытых элементов в рамках одного вызова интерфейсной операции, – оракул. Он обладает всей информацией, необходимой для выполнения этого действия. Для известных ему пресостояния, стимула, реакции и постсостояния оракул вычисляет покрытые элементы и помещает результат вычисления в трассу. Для метрики покрытия, описанной функцией μ, оракул вычисляет значение этой функции, которое однозначно характеризует покрытый элемент. Для расширенной метрики покрытия оракул вычисляет значения всех предикатов c_i и индексы предикатов, принявших положительное значение, помещаются в трассу в качестве описания покрытых элементов.

Помимо информации о покрытых элементах в трассу попадает информация о вызовах интерфейсных операций, значениях их параметров, вердиктах оракула, а также о других событиях, произошедших в процессе тестирования. По завершении процесса тестирования трасса используется для анализа результатов тестирования. На ее основе генерируются отчеты о покрытии, отчеты о найденных несоответствиях между поведением целевой системы и требованиями к нему, а также отчеты об ошибках, имевших место в тестовой системе.

Архитектура тестовой системы, дополненная работой с метриками покрытия, представлена на Рис. 8.

2.4.4. Управляемые метрики покрытия и оптимизация тестового набора

Помимо статической оценки качества тестирования управляемые метрики покрытия используются для оптимизации генерации тестовых данных непосредственно во время тестирования. Технология UniTesK предоставляет возможность не только складывать информацию о покрытии в трассу, но и использовать ее в процессе работы теста.

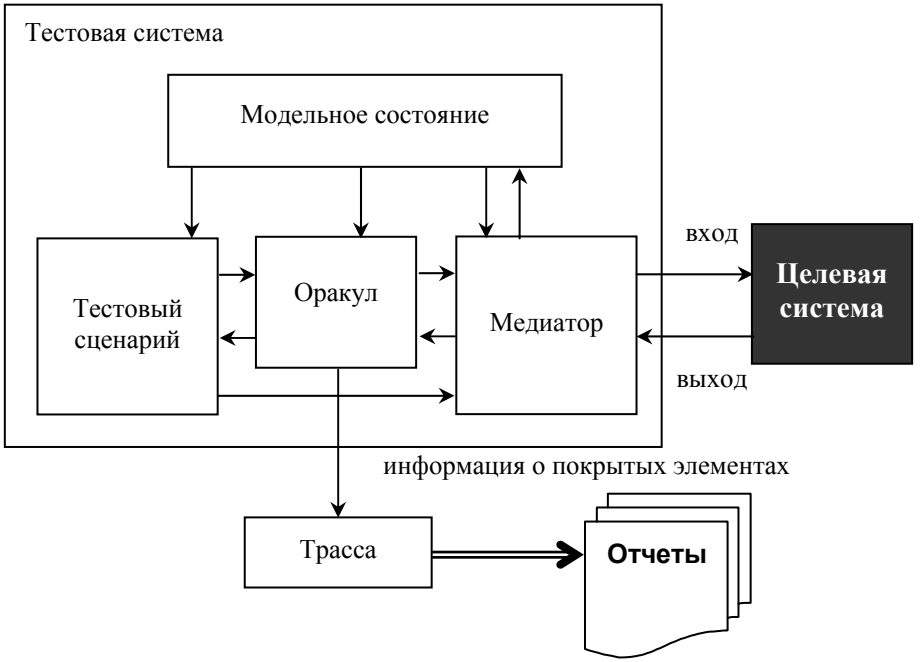

Рис. 8. Универсальная архитектура теста с учетом оценки качества тестирования

Предположим, что перед тестом стоит задача достичь определенного уровня покрытия по некоторому набору метрик. В этом случае тесту не имеет смысла подавать целевой системе стимулы, которые не приводят к улучшению покрытия. Чтобы оптимизировать таким образом тестовый набор, достаточно собирать в глобальном состоянии сценария информацию о покрытых элементах и отбрасывать в сценарных функциях те стимулы, которые не приводят к улучшению покрытия.

2.5. Унифицированная архитектура теста

В предыдущих разделах мы определили математические модели, используемые в технологии UniTesK. Также мы рассмотрели, как эти модели отражаются в унифицированной архитектуре теста. А теперь мы взглянем на унифицированную архитектуру в целом, соединив все разрозненные описания в единое целое.

Унифицированная архитектура теста в целом представлена на Рис. 9. Она определяет набор компонентов теста, составляющих ядро тестовой системы.

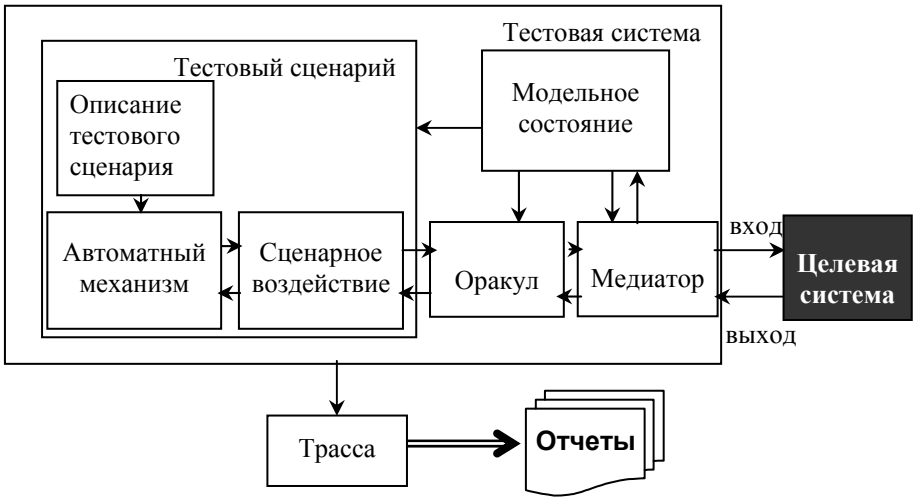

Рис. 9. Процесс работы тестовой системы, построенной по технологии UniTesK в рамках классических программных контрактов

Помимо тестовой системы на диаграмме представлены:

- целевая система – программная система, которую необходимо протестировать,
- трасса – набор информации о ходе тестирования, собираемой тестовой системой во время работы теста,
- отчеты – набор отчетов о различных аспектах процесса тестирования, генерируемый инструментами семейства UniTesK, на основе трассы теста.

Тестовая система содержит четыре компонента.

Тестовый сценарий управляет всем процессом тестирования. Он интерактивно, в ходе взаимодействия с целевой системой, генерирует последовательность воздействий на целевую систему. Для этого автоматный механизм осуществляет обход графа, заданного в описании сценария. Прохождение по дуге этого графа сопровождается выполнением сценарного воздействия, приписанного данной дуге. Сценарное воздействие, которое является «компактным» тестовым сценарием, управляет последовательностью взаимодействий с целевой системой. По завершению работы сценарного воздействия автоматный механизм либо принимает решение о завершении тестирования, либо выбирает следующую дугу и инициирует следующее сценарное воздействие.

Все воздействия на целевую систему генерируемые тестовым сценарием осуществляется через оракул. **Оракул**

- проверяет допустимость данного воздействия в текущем модельном состоянии,

- выполняет это воздействие посредством обращения к медиатору,
- проверяет корректность поведения целевой системы в рамках данного взаимодействия,
- вычисляет покрытые элементы для всех заданных метрик покрытия и помещает эту информацию в трассу теста.

Медиатор получает от оракула указание осуществить тестовое воздействие, заданное стимулом $x \in X$. Для решения этой задачи медиатор

- преобразует стимул x в вызов интерфейсной функции целевой системы со значениями входных параметров, соответствующих данному стимулу, и осуществляет этот вызов,
- получает ответную реакцию целевой системы,
- преобразует ее в модельное представление – некоторую реакцию $y \in Y$,
- синхронизирует модельное состояние с внутренним состоянием тестируемой системы после оказания тестового воздействия,
- возвращает модельное представление ответной реакции $y \in Y$ оракулу.

Единственный пассивный компонент тестовой системы – это **модельное состояние**. Основная задача модельного состояния заключается в хранении текущих значений переменных состояния, определенных в спецификации целевой системы. Эти значения составляют модельное представление внутреннего состояния целевой системы. Поэтому только медиатор, который непосредственно работает с целевой системой, может изменять модельное состояние. Все остальные компоненты могут только читать модельное состояние.

3. Тестирование систем с асинхронным интерфейсом

Настоящая глава начинается с определение систем с асинхронным интерфейсом и выделения ограничений рассмотренных методов и моделей, которые не позволяют использовать их для проведения полноценного тестирования систем с асинхронным интерфейсом. Далее будут определены математические модели, которые позволяют корректным образом сформулировать основные задачи тестирования для систем с асинхронным интерфейсом, и рассмотрены алгоритмы, решающие эти задачи. На основе предложенных моделей и алгоритмов будет построена унифицированная архитектура асинхронной тестовой системы, определяющая архитектуру всех тестовых систем, предназначенных для тестирования систем с асинхронным интерфейсом рассматриваемым методом.

3.1. Системы с асинхронным интерфейсом

Основополагающим камнем в математических моделях технологии UniTesK являются понятия взаимодействия и модели поведения целевой системы. Именно в этих терминах описывается процесс тестирования и на их основе строятся все остальные модели.

Однако, у данных моделей есть несколько ограничений. Во-первых, в них предполагается, что любое взаимодействие между целевой системой и ее окружением может инициироваться только окружением. Во-вторых, предполагается, что все взаимодействия происходят строго последовательно, то есть следующее взаимодействие начинается только после завершения предыдущего.

Примером целевой системы, для которой эти предположения нарушаются, является подсистема, реализующая стек протоколов. Она получает пакеты от вышестоящего уровня, формирует из них пакеты нижележащего уровня и посылает по сети. С другой стороны, эта подсистема получает пакеты из сети, преобразует их и передает вышестоящему приложению.

Передача пакетов в сеть и приложению являются примерами взаимодействий, которые инициируются не окружением, а самой целевой системой. Конечно, эти взаимодействия можно моделировать как часть взаимодействий, инициированных окружением. То есть можно рассматривать пакет, отправленный в сеть, как часть реакции на просьбу послать сообщение от приложения или как часть реакции на пришедший пакет из сети.

Но в этом случае особенно критичными становится второе ограничение. Рассмотрим взаимодействие, которое инициируется приложением вышестоящего уровня. Приложение просит стек протоколов послать сообщение. Завершением этого взаимодействия будем считать передачу в сеть всех пакетов, необходимых для посылки данного сообщения. Предположение о последовательности взаимодействий не позволяет тестовой системе инициировать следующее взаимодействие до завершения предыдущего. Таким образом, рассмотренная выше архитектура будет ограничена тестированием только тех ситуаций, в которых стек протоколов в каждый момент времени обрабатывает только один запрос о посылке сообщения.

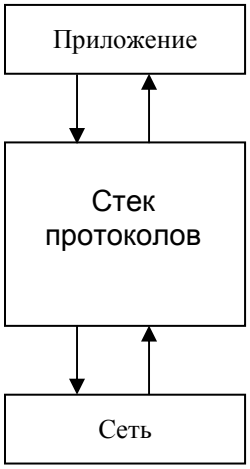

Рис. 10. Стек протоколов

Программные системы, для которых нарушаются вышеприведенные ограничения наиболее часто встречаются среди драйверов устройств, телекоммуникационных приложений, подсистем, реализующих межпроцессное или межпотоковое взаимодействие, и других программ системного уровня. Такого рода программы характеризуются асинхронными взаимодействиями с окружением.

Программные системы, которые

- самостоятельно инициируют взаимодействия с окружением или
- могут одновременно участвовать в нескольких взаимодействиях с окружением

мы будем называть *системами с асинхронным интерфейсом*.

Технология UniTesK в рамках подхода классических программных контрактов может быть применима для тестирования систем с асинхронным интерфейсом, однако качество разработанных тестов будет ограничиваться тестированием только в последовательном, синхронном режиме. В следующих разделах настоящей главы мы рассмотрим решения, которые предлагает технология UniTesK для обеспечения полноценного тестирования систем с асинхронным интерфейсом.

3.2. Оценка корректности поведения тестируемой системы

3.2.1. Модель поведения

Основной задачей модели поведения целевой системы является адекватное описание поведения целевой системы с тем, чтобы тестовая система имела возможность оценить корректность этого поведения относительно требований, предъявляемых к нему.

Основным отличием систем с асинхронным интерфейсом являются

- взаимодействия, инициируемые целевой системой;
- возможность участия целевой системы в нескольких взаимодействиях одновременно.

Чтобы учесть взаимодействия, инициируемые целевой системой, мы расширим понятие интерфейса целевой системы.

Асинхронным интерфейсом целевой системы называется тройка (X, Y, Z), где

- X – множество стимулов,
- Y – множество реакций,
- Z – множество отложенных реакций.

Каждое из этих множеств может быть бесконечным.

Множество отложенных реакций Z предназначено для описания взаимодействий, инициируемых целевой системой.

Другое отличие в определении асинхронного интерфейса, по сравнению с синхронным, заключается в отсутствии множества состояний. Причина этого состоит в том, что при асинхронных взаимодействиях состояние целевой системы в общем случае является недоступным для внешнего наблюдения, так как оно может изменяться в любой момент ввиду участия в целевой системы в параллельных взаимодействиях.

Асинхронным воздействием на целевую систему, обладающую интерфейсом (X, Y, Z), называется пара $(x, y) \in X \times Y$. Первый элемент воздействия x будем называть стимулом, а второй y – непосредственной реакцией.

Асинхронной реакцией целевой системы, обладающей интерфейсом (X, Y, Z), называется элемент множества Z.

Асинхронные воздействия и асинхронные реакции будем называть *асинхронными взаимодействиями* с целевой системой.

Асинхронные взаимодействия с целевой системой разбиваются на два класса. Первый класс составляют взаимодействия, инициируемые тестовой системой. Их мы называем асинхронными воздействиями. Асинхронное воздействие состоит из стимула, моделирующего данные передаваемые от тестовой системе к целевой, и непосредственной реакции, содержащей данные передаваемые в обратном направлении в рамках того же взаимодействия.

Второй класс асинхронных взаимодействий содержит взаимодействия, инициируемые целевой системой. Такие взаимодействия мы называем асинхронными реакциями (или отложенными реакциями). Асинхронные реакции используются для моделирования данных передаваемых только в направлении от целевой системы.

Таким образом, моделировать взаимодействия с целевой системой предлагается следующим способом. Если взаимодействие инициируется окружением целевой системы, то для его описания используются асинхронные воздействия. При этом данные, передаваемые в направлении к целевой системе, описываются стимулом, а данные, передаваемые от целевой системы, описываются непосредственной реакцией.

Если взаимодействие инициируется целевой системой и данные в рамках этого взаимодействия передаются только в направлении от целевой системы, то такие взаимодействия моделируются асинхронными реакциями.

Если же взаимодействие инициируется целевой системой, но данные в рамках этого взаимодействия передаются и от целевой системы, и к ней, то при моделировании таких взаимодействий предлагается разбивать их на меньшие составляющие, которые описываются посредством асинхронных воздействий и асинхронных реакций. Например, такое взаимодействие можно разбить на асинхронную реакцию, инициируемую целевой системой, и на набор последующих асинхронных воздействий, включающих в себя данные, передаваемые к целевой системе.

Суммарная информация по предлагаемым способам моделирования взаимодействий целевой системы с ее окружением представлена в Таб. 1.

Вид	Инициатор взаимодействия	Направление передачи данных	Способ моделирования
1	Окружение целевой системы	к целевой системе и от нее	Асинхронное воздействие
2	Целевая система	только от целевой системы	Асинхронная реакция
3	Целевая система	от целевой системы и к ней	Асинхронная реакция и последующие асинхронные воздействия

Таб. 1. Виды взаимодействий с целевой системой.

Определение 7.

Асинхронной моделью поведения целевой системы с интерфейсом (X, Y, Z) называется частично-упорядоченное мультимножество асинхронных взаимодействий (P, π).

Модель поведения состоит из, возможно, бесконечного набора асинхронных взаимодействий, имевших место в процессе тестирования. Частичный порядок, заданный на этом наборе, определяет в каком порядке происходили взаимодействия. $p_1 \ \pi \ p_2$ означает, что взаимодействие p_1 произошло раньше, чем взаимодействие p_2. Если взаимодействия несравнимы, то это значит, что их взаимный порядок – неизвестен.

Множество всех асинхронных моделей поведения целевой системы с интерфейсом (X, Y, Z) мы будем обозначать Ω(X, Y, Z).

3.2.2. Модель требований

Автоматом с отложенными реакциями [28] называется пятёрка (V, X, Y, Z, E), где

- V – множество состояний,
- X – множество стимулов,
- Y – множество непосредственных реакций,
- Z – множество отложенных реакций,
- $E \subseteq V \times ((X \times Y) \ X \ Z) \times V$ – множество переходов.

Множества V, X, Y, Z и E могут быть бесконечными.

Переходы автомата с отложенными реакциями делятся на два класса. В первый класс входят переходы (v, (x, y), v'), помеченные стимулом x и непосредственной реакцией y, а во второй – переходы (v, z, v'), помеченные отложенной реакцией z. Первый и последний элементы перехода (v, p, v') мы будем называть пресостоянием и постсостоянием соответственно.

Последовательность переходов (e_1, e_2, …, e_n) автомата с отложенными реакциями A = (V, X, Y, Z, E) называется *путем*, если для каждого $i = 1, \ldots, n-1$ постсостояние перехода e_i совпадает с пресостоянием перехода e_{i+1}. При этом мы будем говорить, что путь ведет из пресостояния перехода e_1 в постсостояние перехода e_n.

Бесконечным путем мы будем называть бесконечную последовательность переходов, любой префикс которой является конечным путем.

Определение 8.

Асинхронной моделью требований к целевой системе с интерфейсом (X, Y, Z) называется автомат с отложенными реакциями, множества стимулов, непосредственных и отложенных реакций которого совпадают с X, Y и Z соответственно.

Будем говорить, что асинхронное взаимодействие $p \in (X \times Y) \ X \ Z$ является *корректным* относительно модели требований A = (V, X, Y, Z, E) в состоянии

$v \in V$, если существует такое состояние $v' \in V$, что тройка (v, p, v') принадлежит множеству переходов E.

Определение 9.

Будем говорить, что асинхронная модель поведения целевой системы (P, π) *удовлетворяет* асинхронной модели требований $A = (V, X, Y, Z, E)$ с начальным состоянием $v_0 \in V$, если существует конечный или бесконечный путь $(e_1, e_2, \ldots, e_n, \ldots)$ в автомате A такой, что:

- пресостояние перехода e_1 совпадает с состоянием v_0,
- существует взаимнооднозначное соответствие κ между мультимножеством P и мультимножеством переходов $\{ e_i \}$, такое что:
- для каждого элемента $p \in P$ второй элемент перехода $\kappa(p)$ совпадает с p,
- для любой пары $p_1, p_2 \in P$ из $p_1 \pi p_2$ следует, что индекс перехода $\kappa(p_1)$ в пути меньше индекса перехода $\kappa(p_2)$.

Другими словами, асинхронная модель поведения удовлетворяет модели требований с данным начальным состоянием, если взаимодействия из мультимножества P можно упорядочить, не вступая в противоречие с частичным порядком π, таким образом, чтобы в автомате с отложенными реакциями нашелся путь, помеченный данными взаимодействиями в данном порядке и начинающийся в данном начальном состоянии.

Важно отметить, что асинхронная модель требований предполагают, что для любого набора взаимодействий, осуществлявшихся параллельно, существует эквивалентное с точки зрения конечного результата последовательное выполнение этих взаимодействий. Если данная гипотеза, называемая **гипотезой простого параллелизма** [20], не выполняется, то необходимо разбить взаимодействия на более мелкие, для которых гипотеза будет верна. Если никакое разбиение не позволяет добиться выполнения данной гипотезы, то рассматриваемый метод является неприменимым для тестирования такой системы. На настоящий момент автору неизвестен пример программной системы, для которой невозможно выбрать асинхронный интерфейс, удовлетворяющий гипотезе простого параллелизма.

Проверка наличия отношения “удовлетворяет” для асинхронных моделей значительно сложнее, по сравнению с синхронным случаем. Поэтому необходимо провести дополнительный анализ алгоритма этой проверки. Но прежде чем рассматривать этот алгоритм, мы остановимся на способах описания модели требований и модели поведения, так как способы задания моделей оказывают существенное влияния на алгоритмы работы с ними.

3.2.3. Описание асинхронной модели требований

Для описания асинхронных моделей требований используется немного модифицированный подход программных контрактов. Основные идеи этого способа состоят в следующем.

Для целевой системы проводится анализ ее интерфейса. В ходе анализа выявляются виды взаимодействий, в которых целевая система может принимать участие, и их параметры. Для каждого вида определяется, кто является инициатором взаимодействия данного вида, и какие данные передаются в ходе взаимодействия по направлению к целевой системе, а какие – от нее.

При этом накладывается ограничение на взаимодействия, инициируемые целевой системой. В процессе таких взаимодействий данные должны передаваться только от целевой системы. Если это ограничение не выполняется, то необходимо разбить проблемные взаимодействия на меньшие составляющие.

В результате мы получим набор взаимодействий, инициируемых окружением целевой системы, и набор взаимодействий, инициируемых целевой системой. Взаимодействия первого типа имеют данные, передаваемые к целевой системы, мы будем называть их *входными*, и данные, передаваемые от целевой системы, мы будем называть их *выходными*. Взаимодействия второго типа содержат только данные, передаваемые от целевой системы, которые мы также будем называть *выходными*.

Каждому виду взаимодействий ставится в соответствие интерфейсная операция. Для взаимодействий первого типа интерфейсная операция имеет набор входных параметров, описывающих входные данные, и набор выходных параметров, описывающих выходные данные. Такие интерфейсные операции мы будем называть *интерфейсными операциями-стимулами*. Для взаимодействий второго типа интерфейсная операция имеет только выходные параметры, описывающие выходные данные. Интерфейсные операции такого типа мы будем называть *интерфейсными операциями-реакциями*.

Для описания требований к поведению целевой системы в рамках асинхронных взаимодействий используется спецификации соответствующих интерфейсных операций. Предусловие операции определяет какие значения входных параметров являются допустимыми для передачи целевой системе, а постусловие операции определяет какие выходные значения параметров являются корректными для данных значений входных параметров. Таким образом, постусловие описывает требования к поведению целевой системы в ходе взаимодействий данного вида.

Для описания зависимости поведения в рамках данного взаимодействия от предыдущих взаимодействий используется модельное состояние. Модельное состояние образуется набором переменных, хранящих все необходимые для этого данные об истории взаимодействий с целевой системой.

Предположим, что для целевой системы были выделены k видов взаимодействий, инициируемых окружением, и m видов взаимодействий, инициируемых целевой системой. Тогда *асинхронной спецификацией* целевой системы называется набор спецификаций интерфейсных операций, соответствующих каждому выделенному виду взаимодействий: $\{ Spec_{Si} \mid i = 1, \ldots, k; k > 0 \} \cup \{ Spec_{Rj} \mid j = 1, \ldots, m; m \geq 0 \}$.

Асинхронная модель требований **MRA**[Spec] = (V, X, Y, Z, E), которая описывается данной асинхронной спецификацией Spec = { $\text{Spec}_{Si} \mid i = 1, \ldots, k;\ k > 0$ } $\cup$ { $\text{Spec}_{Rj} \mid j = 1, \ldots, m;\ m \geq 0$ } определяется согласно следующим правилам:

1. если в спецификации присутствует хотя бы одна переменная состояния, то множество состояний V является декартовым произведением множеств допустимых значений всех переменных состояния

$$V = \prod_{v \in Var_{Spec}} V_v ,$$

где Var_{Spec} является объединением переменных состояния всех интерфейсных операций, принадлежащих спецификации

$$\text{Var}_{\text{Spec}} = \bigcup_{i=1}^{k} Var_{S_i} \cup \bigcup_{j=1}^{m} Var_{R_j} .$$

2. если в спецификации не участвует ни одной переменной состояния, то множество состояний состоит из единственного элемента ε:

$$V = \{ \varepsilon \}.$$

3. множество стимулов X является дизъюнктивным объединением декартовых произведений множеств допустимых значений входных параметров всех интерфейсных операций-стимулов спецификации

$$X = \bigsqcup_{i=1}^{k} X_{S_i} .$$

Мы будем считать, что каждый элемент дизъюнктивного объединения помечается именем интерфейсного операций и таким образом эти элементы не пересекаются между собой.

4. множество непосредственных реакций Y является дизъюнктивным объединением декартовых произведений множеств допустимых значений выходных параметров всех интерфейсных операций-стимулов спецификации

$$Y = \bigsqcup_{i=1}^{k} Y_{S_i} .$$

Мы будем считать, что каждый элемент дизъюнктивного объединения помечается именем интерфейсного операций и таким образом эти элементы не пересекаются между собой.

5. множество отложенных реакций Z является дизъюнктивным объединением декартовых произведений множеств допустимых значений выходных параметров всех интерфейсных операций-реакций спецификации

$$Z = \bigsqcup_{j=1}^{m} Y_{R_j} .$$

Мы будем считать, что каждый элемент дизъюнктивного объединения помечается именем интерфейсной операции и таким образом эти элементы не пересекаются между собой.

6. множество переходов E состоит из всех переходов, удовлетворяющих обобщенным предусловию и постусловию спецификации

$$E = \{ (v, (x, y), v') \in V \times ((X \times Y) \text{ X } Z) \times V \mid \text{preS}(v, x) \wedge \text{postS}(v, x, y, v') \} \cup \{ (v, z, v') \in V \times ((X \times Y) \text{ X } Z) \times V \mid \text{preR}(v) \wedge \text{postR}(v, z, v') \},$$

где предикаты $\text{preS} = \bigwedge_{i=1}^{k} pre_{S_i}$, $\text{postS} = \bigwedge_{i=1}^{k} post_{S_i}$,

$\text{preR} = \bigwedge_{j=1}^{m} pre_{R_j}$, $\text{postR} = \bigwedge_{j=1}^{m} post_{Rj}$.

3.2.4. Описание асинхронных взаимодействий в модели поведения

Как и в синхронном случае, модель поведения целевой системы не может быть описана статическим образом. Только динамически, во время тестирования, тестовая система наблюдает за поведением целевой системы и строит модель этого поведения.

Чтобы модель поведения была согласована с моделью требований, необходимо чтобы она была построена на основе единого интерфейса целевой системы (X, Y, Z). Если модель требований задана спецификацией Spec, то таким образом интерфейс целевой системы (X, Y, Z) фиксирован согласно определению **MRA**[Spec]. Стимул задается интерфейсной операцией-стимулом и набором значений входных параметров этой операции, непосредственная реакция идентифицируется интерфейсной операцией-стимулом и набором значений выходных параметров этой операции, а отложенная реакция идентифицируется интерфейсной операцией-реакцией и набором значений выходных параметров этой операции.

В этом случае асинхронная модель поведения целевой системы определяется в терминах заданной спецификации. Асинхронная модель поведения состоит из набора асинхронных взаимодействий и частичного порядка над ним. Набор взаимодействий определяется посредством регистрации всех зафиксированных взаимодействий в специальном компоненте тестовой системы – регистраторе взаимодействий. А частичный порядок над этим набором задается посредством комбинации двух моделей: модели каналов и модели временных меток.

Асинхронные взаимодействия, инициированные тестовой системой, регистрируются после получения непосредственной реакции от целевой системы. Такие взаимодействия характеризуются:

- идентификатором интерфейсной операции-стимула,
- набором значений входных параметров этой операции,
- набором значений выходных параметров этой операции.

Асинхронные взаимодействия, инициированные целевой системой, регистрируются после их завершения. Такие взаимодействия характеризуются:

- идентификатором интерфейсной операции-реакции,
- набором значений выходных параметров этой операции.

Способы задания частичного порядка над мультимножеством асинхронных взаимодействий мы рассмотрим в следующих двух разделах.

3.2.5. Модель каналов

Модель каналов предназначена для задания частичного порядка на наборе асинхронных взаимодействий.

Определение 10.

Пусть D – конечное или бесконечное множество. *Моделью каналов* на множестве D мы будем называть конечное или бесконечное множество упорядоченных подмножеств множества D $Ch = \{ (D_i, <_i) \mid i = 1, \ldots, n, \ldots \}$, такое что

- подмножества D_i взаимно не пересекаются ($\forall i, j \quad i \neq j \Rightarrow D_i \cap D_j = \varnothing$),
- объединение всех подмножеств дает множество D ($\bigcup D_i = D$).

Модель каналов задает частичный порядок на множестве D, определяемый следующим образом:

$$d_1 \, \pi_{Ch} \, d_2 \Leftrightarrow \exists i: \; d_1 <_i d_2.$$

Множества D_i модели каналов мы будем называть *каналами*.

Модель каналов во многих случаях является удобным механизмом для задания частичного порядка, так как она позволяет описывать наиболее естественным способом порядок взаимодействий, происходивших в одном “канале”.

Например, если сетевой протокол, с помощью которого осуществляется взаимодействие, обеспечивает сохранение порядка сообщений, то все сообщения посылаемые через один “сокет” на другой “сокет” между собой строго упорядочены. Тем не менее, никакой информации о порядке доставки этих сообщений относительно сообщений, посылаемых по другим парам “сокетов”, не известно.

В таких ситуациях, модель каналов позволяет отнести все упорядоченные сообщения к одному каналу и задать для этого канала соответствующий порядок. Чтобы сделать это, при регистрации каждого взаимодействия в тестовой системе предлагается указывать в качестве одной из характеристик

взаимодействия – идентификатор канала, к которому относится данное взаимодействие. Порядок взаимодействий в рамках одного канала задается порядком регистрации взаимодействий. Считается, что взаимодействие, которое было зарегистрировано раньше другого, относящегося к тому же каналу, произошло раньше, чем второе.

Но одной модели каналов оказывается недостаточно для описания произвольного частичного порядка. Поэтому существует еще один механизм для определения частичного порядка на асинхронной модели поведения целевой системы.

3.2.6. Модель временных меток

Модель временных меток предназначена для возможности задания дополнительных ограничений на порядок взаимодействий в асинхронной модели поведения целевой системы.

Пусть (TM, π_{TM}) – частично-упорядоченное множество временных меток. Тогда *временным интервалом* мы будем называть пару временных меток (tm_1, tm_2), в которой вторая метка больше либо равна первой (т.е. $tm_1 = tm_2 \vee tm_1 \, \pi_{TM} \, tm_2$).

Множество всех временных интервалов будет обозначаться TI(TM). Определим на нем частичный порядок π_{TI} следующим образом:

$$(tm_1, tm_2) \, \pi_{TI} \, (tm'_1, tm'_2) \Leftrightarrow tm_2 \, \pi_{TM} \, tm'_1$$

Определение 11.

Пусть D – конечное или бесконечное множество, а (TM, π_{TM}) – множество временных меток. Тогда *моделью временных меток* на множестве D мы будем называть отображение $\tau : D \rightarrow TI(TM)$.

Модель временных меток задает на множестве D частичный порядок π_τ :

$$d_1 \, \pi_\tau \, d_2 \Leftrightarrow \tau(d_1) \, \pi_{TI} \, \tau(d_2).$$

В качестве множества временных меток TM в технологии UniTesK используется множество $(F \times Nat) \cup \{ -\infty, +\infty \}$, где F – конечное или бесконечное множество систем координат, Nat – множество натуральных чисел, а $-\infty$ и $+\infty$ – специально выделенные значения.

Для описания частичного порядка π_{TM} применяются следующие правила:

- $\forall f \in F \quad \forall n \in Nat \quad -\infty \, \pi_{TM} (f, n)$,
- $\forall f \in F \quad \forall n \in Nat \quad (f, n) \, \pi_{TM} +\infty$,
- $\forall f \in F \quad \forall n_1, n_2 \in Nat \quad n_1 < n_2 \Rightarrow (f, n_1) \, \pi_{TM} (f, n_2)$,
- $\forall f_1, f_2 \in F \quad \forall n_1, n_2 \in Nat \quad (f_1, n_1) \, \pi_{TM} (f_2, n_2)$, если эта зависимость была зафиксирована разработчиком теста и она не противоречит определению частичного порядка (иррефлексивность, антисимметричность и транзитивность).

Для установки зависимостей последнего типа тестовая система предоставляет специальную функцию, с помощью которой в процессе тестирования можно

зафиксировать зависимости между временными метками. Множество систем координат F также определяется динамически, в процессе тестирования.

Модель временных меток предоставляет удобный способ для описания частичного порядка на множестве взаимодействий при использовании в следующей интерпретации. Временная метка рассматривается как идентификатор некоторого момента времени. Каждый момент времени описывается натуральным числом в некоторой системе временных координат. В рамках одной системы координат все моменты времени упорядочены. А про моменты времени, принадлежащие различным системам координат, заранее ничего не известно. Но если какая-нибудь информация появляется, то она фиксируется согласно четвертому правилу.

Каждому взаимодействию ставится в соответствие временной интервал, первая временная метка, которого описывает момент начала взаимодействия, а вторая – момент его завершения. Таким образом, считается, что одно взаимодействие достоверно произошло раньше другого, если временная метка завершения первого взаимодействия меньше временной метки начала второго. В качестве концов интервала допускается использование специальных значений $-\infty$ и $+\infty$, предназначенных для описания открытых интервалов.

Способ выделения натуральных чисел, используемых для описания момента времени, может варьироваться. Это может быть текущее значение системного таймера или просто некоторое абстрактное число. Различные системы временных координат, например, могут соответствовать различным машинам в сети.

Определять модель временных меток предлагается следующим образом. Во-первых, в процессе тестирования фиксируется информация об отношении порядка между временными метками, принадлежащими различным системам координат. А во-вторых, при регистрации взаимодействий указывается временной интервал, соответствующий данному взаимодействию.

3.2.7. Описание асинхронной модели поведения

Суммируя информацию о построении асинхронной модели поведения целевой системы в процессе тестирования, можно сказать, что этот процесс состоит из решения двух задач:

- регистрации взаимодействий,
- фиксирования достоверно известной информации о порядке, в котором происходили эти взаимодействия.

Регистрация взаимодействий осуществляется в специальном компоненте тестовой системы – регистраторе взаимодействий. А для решения второй задачи предлагается при регистрации взаимодействий указывать идентификатор канала, к которому относится данное взаимодействие, и временной интервал, в котором оно происходило. Кроме того, требуется фиксировать известные ограничения на порядок временных меток, принадлежащих различным системам координат.

В результате этого тестовая система будет иметь набор асинхронных взаимодействий D, модель каналов Ch и модель временных меток τ. На основе этой информации будет построена асинхронная модель поведения (P, π), в которой

- мультимножество взаимодействий P совпадает с D,
- частичный порядок π является транзитивным замыканием объединения частичных порядков π_{Ch} и π_{τ}.

3.2.8. Алгоритм проверки корректности поведения

Предположим, что нам даны конечная асинхронная модель поведения целевой системы (P, π) и асинхронная модель требований $A = (V, X, Y, Z, E)$ с начальным состоянием $v_0 \in V$. Как проверить, находятся ли эти модели в отношении “удовлетворяет” или нет?

Для этого необходимо проверить существование пути $(e_1, e_2, \ldots, e_n)$ в автомате A, начинающегося в состоянии v_0 и помеченного взаимодействиями из P так, чтобы это не противоречило частичному порядку π.

Асинхронная модель требований $A = (V, X, Y, Z, E)$ задается асинхронной спецификацией $\{ Spec_{Si} \mid i = 1, \ldots, k; k > 0 \} \cup \{ Spec_{Rj} \mid j = 1, \ldots, m; m \geq 0 \}$, которая определяет переходы автомата неявным образом. Это осложняет решение поставленной задачи, так как вычисление состояний, в которые можно попасть из данного состояния по переходам помеченным данным символом, сводится к решению системы уравнений.

В данном разделе мы не будем рассматривать пути решения этой проблемы. Здесь мы будем считать, что нам задана вспомогательная функция $\gamma : V \times ((X \times Y) X Z) \rightarrow 2^V$, вычисляющая множество состояний, которые модель требований допускает в качестве постсостояния перехода с данным пресостоянием v и асинхронным взаимодействием p:

$$\gamma(v, p) = \{ v' \in V \mid (v, p, v') \in E \}$$

Но даже в этом случае для проверки корректности поведения требуется рассмотреть все возможные линеаризации асинхронной модели поведения и для каждой из них проверить существование соответствующего пути в автомате A.

Определение 12.

Линейный порядок $<$ на частично-упорядоченном множестве (P, π) называется *линеаризацией*, если он сохраняет частичный порядок π, то есть

$$\forall\, p_1, p_2 \in P \;\; p_1 \,\pi\, p_2 \Rightarrow p_1 < p_2.$$

Для последующего рассмотрения введем вспомогательную функцию Γ^*, вычисляющую по состоянию в автомате и последовательности асинхронных взаимодействий, множество состояний автомата, достижимых из данного по последовательности дуг, помеченных данной последовательностью взаимодействий. Формальное определение функции Γ^* следующее:

$$\Gamma^*(v_0, (p_1, p_2, \ldots, p_n)) = \begin{cases} \Gamma(\ldots\Gamma(\Gamma(\{v_0\}, p_1), p_2), \ldots p_n), & \text{для } n > 0 \\ \{v_0\}, & \text{для } n = 0 \end{cases}, \text{ где}$$

$$\Gamma(V, p) = \bigcup_{v \in V} \gamma(v, p).$$

Лемма.

Для последовательности асинхронных взаимодействий $(p_1, p_2, \ldots, p_n)$ существует путь $(e_1, e_2, \ldots, e_n)$ в автомате A, начинающийся в состоянии v_0 и помеченный данными взаимодействиями, тогда и только тогда, когда $\Gamma^*(v_0, (p_1, p_2, \ldots, p_n)) \neq \varnothing$.

Доказательство.

Действительно, пусть существует путь $v_0 \xrightarrow{p_1} v_1 \xrightarrow{p_2} \ldots \xrightarrow{p_n} v_n$ в автомате A, начинающийся в состоянии v_0 и помеченный взаимодействиями $(p_1, p_2, \ldots, p_n)$. Докажем индукцией по длине пути, что $v_n \in \Gamma^*(v_0, (p_1, p_2, \ldots, p_n))$, а следовательно $\Gamma^*(v_0, (p_1, p_2, \ldots, p_n)) \neq \varnothing$.

Базис индукции. (n = 1)

Итак, пусть $v_0 \xrightarrow{p_1} v_1$. Тогда $v_1 \in \gamma(v_0, p_1) \Rightarrow v_1 \in \Gamma(\{v_0\}, p_1) = \Gamma^*(v_0, (p_1))$.

Шаг индукции.

Предположим, что мы доказали утверждение для путей длины n-1. Докажем его для n.

Пусть существует путь $v_0 \xrightarrow{p_1} v_1 \xrightarrow{p_2} \ldots \xrightarrow{p_n} v_n$ в автомате A, начинающийся в состоянии v_0 и помеченный взаимодействиями $(p_1, p_2, \ldots, p_n)$. Тогда по предположению индукции $v_{n-1} \in \Gamma^*(v_0, (p_1, p_2, \ldots, p_{n-1}))$. Но $v_{n-1} \xrightarrow{p_n} v_n$, то есть $v_n \in \gamma(v_{n-1}, p_n) \Rightarrow v_n \in \Gamma(V_{n-1}, p_n)$, где V_{n-1} – любое множество, содержащее v_{n-1}, в том числе $V_{n-1} = \Gamma^*(v_0, (p_1, p_2, \ldots, p_{n-1}))$. Следовательно $v_n \in \Gamma(\Gamma^*(v_0, (p_1, p_2, \ldots, p_{n-1})), p_n) = \Gamma^*(v_0, (p_1, p_2, \ldots, p_n))$.

В обратную сторону.

Пусть $\Gamma^*(v_0, (p_1, p_2, \ldots, p_n)) \neq \varnothing$. Докажем индукцией по n, что в автомате A существует путь $v_0 \xrightarrow{p_1} v_1 \xrightarrow{p_2} \ldots \xrightarrow{p_n} v_n$, начинающийся в состоянии v_0 и помеченный взаимодействиями $(p_1, p_2, \ldots, p_n)$.

Базис индукции. (n = 1)

Пусть $\Gamma^*(v_0, (p_1)) \neq \varnothing$. Тогда существует $v_1 \in \Gamma^*(v_0, (p_1)) = \Gamma(\{v_0\}, p_1) = \gamma(v_0, p_1)$. То есть переход $v_0 \xrightarrow{p_1} v_1$ принадлежит автомату A и является искомым путем.

Шаг индукции.

Предположим, что мы доказали утверждение для n-1. Докажем его для n.

Пусть $\Gamma^*(v_0, (p_1, p_2, \ldots, p_n)) \neq \varnothing$.

Тогда $\forall k \in \{1,\ldots,n-1\}\ \exists v_k \in \Gamma^*(v_0, (p_1, p_2, \ldots, p_k))$: $\Gamma^*(v_k, (p_{k+1}, p_{k+2}, \ldots, p_n)) \neq \varnothing$.

Предположим, что это не так.

То есть $\exists k \in \{1,\ldots,n-1\}\ \forall v_k \in \Gamma^*(v_0, (p_1, p_2, \ldots, p_k))\ \Gamma^*(v_k, (p_{k+1}, p_{k+2}, \ldots, p_n)) = \varnothing$.

Тогда $\Gamma^*(v_0, (p_1, p_2, \ldots, p_n)) = \Gamma(\ldots\Gamma(\Gamma^*(v_0, (p_1, p_2, \ldots, p_k)), p_{k+1}), \ldots p_n) =$

$= \bigcup_{v_k \in \Gamma^*(v_0, (p_1, p_2, \ldots, p_k))} \Gamma(\ldots\Gamma(\{v_k\}, p_{k+1}), \ldots p_n) =$

$\bigcup_{v_k \in \Gamma^*(v_0, (p_1, p_2, \ldots, p_k))} \Gamma^*(v_k, (p_{k+1}, \ldots, p_n)) = \bigcup_{v_k \in \Gamma^*(v_0, (p_1, p_2, \ldots, p_k))} \varnothing = \varnothing$, что противоречит исходному предположению.

Данное утверждение верно и для k = 1. То есть найдется такое состояние $v_1 \in \Gamma^*(v_0, (p_1))$, что $\Gamma^*(v_1, (p_2, p_3, \ldots, p_n)) \neq \varnothing$. Следовательно, по предположению индукции в автомате A существует путь $v_1 \xrightarrow{p_2} v_2 \xrightarrow{p_3} \ldots \xrightarrow{p_n} v_n$, начинающийся в состоянии v_1 и помеченный взаимодействиями $(p_2, p_3, \ldots, p_n)$.

Так как $v_1 \in \Gamma^*(v_0, (p_1)) = \Gamma(\{v_0\}, p_1) = \gamma(v_0, p_1)$, то переход $v_0 \xrightarrow{p_1} v_1$ также принадлежит автомату A. Следовательно в автомате A существует путь $v_0 \xrightarrow{p_1} v_1 \xrightarrow{p_2} \ldots \xrightarrow{p_n} v_n$, начинающийся в состоянии v_0 и помеченный взаимодействиями $(p_1, p_2, \ldots, p_n)$.

■

Как следует из леммы, проверка существования пути в автомате сводится к последовательному применению функции Γ, вычислительная сложность которого сильно зависит от числа состояний, допускаемых моделью требований в качестве постсостояния перехода с заданными пресостоянием и асинхронным взаимодействием. В дальнейших оценках мы будем считать, что асинхронная модель требований допускает не более одного такого состояния, то есть модель является детерминированной.

Асинхронная модель требований $A = (V, X, Y, Z, E)$ называется *детерминированной*, если для каждого состояния $v \in V$ и асинхронного взаимодействия $p \in (X \times Y) \times Z$ существует не более одного перехода $(u, x, u') \in E$, такого что $u = v$ и $x = p$. Другими словами, в детерминированных моделях $\forall v, p\ |\gamma(v, p)| \leq 1$.

Для детерминированных моделей проверка существования пути, начинающегося в заданном состоянии v_0 и помеченного заданной последовательностью взаимодействий $(p_1, p_2, \ldots, p_n)$, в худшем случае сводится к n-кратному вычислению функции γ. Таким образом, сложность этой проверки оценивается сверху n-кратной сложностью вычисления функции γ.

Для оценки корректности асинхронной модели поведения такую проверку необходимо выполнить для каждой возможной линеаризации мультимножества асинхронных взаимодействий. А в худшем случае число линеаризаций составляет n!, где n – мощность множества P. То есть общая оценка сложности составляет n·n!.

Заметим, что многие линеаризации имеют общее начало. Этим свойством можно воспользоваться, чтобы не дублировать проверку существования путей, соответствующих общему началу линеаризаций.

Определим для множества P дерево возможных линеаризаций, как дерево в котором

- из корня выходит n дуг, из вершин первого уровня – n-1 дуга, …, из вершин n-го уровня – 1 дуга и из вершин n+1-го уровня – ни одной дуги.
- каждая дуга помечена элементом из P,
- дуги, выходящие из вершины, путь к которой из корня дерева помечен последовательностью ($p_1, p_2, \ldots, p_k$), помечены элементами P, которые составляют множество $P \setminus \{ p_1, p_2, \ldots, p_k \}$.

Пример дерева возможных линеаризаций для P = { 1, 2, 3 } представлен на Рис. 11.

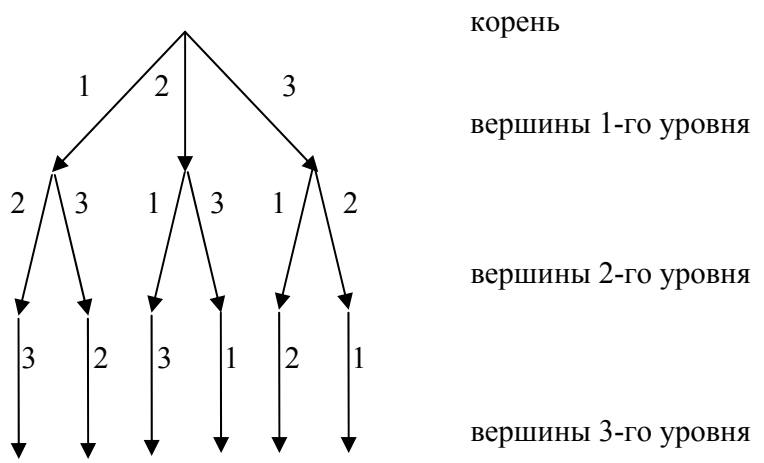

Рис. 11. Дерево возможных линеаризаций для P = { 1, 2, 3 }.

Заметим, что пути, ведущие из корня в листья, помечены последовательностями элементов из P, каждая из которых является линеаризацией множества P, а все вместе они образуют множество всех возможных линеаризаций.

Припишем к каждой вершине дерева возможных линеаризаций подмножество P, составленное из элементов $\{ p_1, p_2, \ldots, p_k \}$, которыми помечен путь из корня дерева к данной вершине.

Деревом возможных линеаризаций частично-упорядоченного множества (P, π) мы будем называть дерево, полученное из дерева возможных линеаризаций множества P удалением всех поддеревьев, корню которых приписано множество не являющееся идеалом множества (P, π). Дуга, ведущая в корень удаляемого поддерева, также удаляется.

Идеалами частично-упорядоченного множества (P, π) называются такие подмножества P, для которых выполнено условие: $y \in I \wedge x \, \pi \, y \Rightarrow x \in I$.

Лемма.

Пути из корня в листья дерева возможных линеаризаций частично-упорядоченного множества (P, π) помечены последовательностями элементов из P, составляющими множество всех возможных линеаризаций (P, π).

Доказательство.

Покажем, что любой путь из корня в лист помечен последовательностью являющейся линеаризаций. Действительно, по построению дерева, любая такая последовательность является упорядочиванием всех элементов P. Покажем,

что определяемый ею линейный порядок сохраняет исходный частичный порядок π.

Предположим, что это не так. То есть $\exists\, p_1, p_2 \in P$: $p_1 \,\pi\, p_2 \wedge p_2 < p_1$.

Из того, что $p_2 < p_1$ следует, что дуга, помеченная p_2, встречается в пути, раньше чем дуга, помеченная p_1. Следовательно к вершине, в которую ведет дуга, помеченная p_2, приписано множество, содержащее p_2 и не содержащее p_1. Такое множество не является идеалом, то есть рассматриваемые дуги не могут принадлежать дереву возможных линеаризаций. Противоречие показывает, что наше предположение не верно и любая последовательность на пути из корня в лист является линеаризаций.

Докажем, что для любой линеаризации (P, π) найдется путь в дереве возможных линеаризаций, помеченный последовательностью соответствующей данной линеаризации.

Для этого мы рассмотрим путь в дереве возможных линеаризаций множества P, помеченный соответствующей последовательностью элементов $(p_1, p_2, \ldots, p_n)$, и покажем, что этот путь будет присутствовать в дереве возможных линеаризаций частично-упорядоченного множества (P, π).

Предположим, что это не так и одна из вершин этого пути была удалена из дерева. Это могло произойти только в случае, когда данная вершина принадлежит поддереву, к корню которого приписано множество, не являющееся идеалом (P, π). Так как рассматриваемый путь ведет из корня в лист дерева, то корень любого поддерева, содержащего вершину из данного пути, также принадлежит этому пути. Следовательно, к одной из вершин рассматриваемого пути приписано множество, не являющееся идеалом (P, π).

Это множество состоит из элементов, приписанных к пути из корня дерева в эту вершину. Так как такой путь в дереве единственен, то он совпадает с началом пути $(p_1, p_2, \ldots, p_n)$. Следовательно, $\exists\, k$: $\{p_1, p_2, \ldots, p_k\}$ – не является идеалом (P, π). То есть существуют такие i и j, что $p_i \in \{p_1, p_2, \ldots, p_k\} \wedge p_j \,\pi\, p_i \wedge p_j \notin \{p_1, p_2, \ldots, p_k\}$.

Отсюда, $i \le k < j \wedge p_j \,\pi\, p_i$. Следовательно, последовательность $(p_1, p_2, \ldots, p_n)$ не является линеаризацией, что противоречит нашему предположению.

■

Если не дублировать проверку существования путей, соответствующих общему началу линеаризаций, то число необходимых вычислений функции γ будет совпадать с числом дуг в дереве возможных линеаризаций. Число дуг в дереве для множества размерностью n составляет:

$$n + n\cdot(n-1) + \ldots + n\cdot(n-1)\cdot\ldots\cdot 1 = \sum_{k=0}^{n-1}\frac{n!}{k!} = n!\sum_{k=0}^{n-1}\frac{1}{k!} \le n!\sum_{k=0}^{\infty}\frac{1}{k!} \le e\cdot n!,$$

где e – число Эйлера.

Основной сложностью в реализации этого алгоритма является организация перебора возможных линеаризаций асинхронной модели поведения (P, π). Наиболее простой подход, основанный на переборе всех перестановок в лексикографическом порядке, требует $O(n\cdot n!)$ операций, так как для проверки того, является ли очередная перестановка линеаризацией, в этом случае требуется ~n операций.

С другой стороны, существуют другие алгоритмы перебора перестановок, которые позволяют сократить затраты на проверку того, является ли очередная перестановка линеаризацией или нет. Но применение нетривиального перебора линеаризаций должно обеспечивать не только сложность O(n!), но и избегать перебора k! заведомо неподходящих перестановок, в тех случаях, когда известно, что (n–k)-ый элемент в какой-то перестановке нарушает условия линеаризации.

Алгоритм перебора линеаризаций, удовлетворяющий всем этим требованиям, был построен на основе алгоритма перебора перестановок 1.11, описанного в [29]. Этот алгоритм обеспечивает перебор перестановок таким образом, что каждая последующая перестановка отличается от предыдущей транспозицией одной пары элементов. В результате, проверка соответствия очередной перестановки условиям линеаризации требует в большинстве случаев константного времени. И в то же время, данный алгоритм позволяет избежать перебора k! заведомо неподходящих перестановок.

Алгоритм проверки корректности поведения, представленный ниже, основан на обходе дерева возможных линеаризаций в глубину. В нем используется следующая вспомогательная функция `int B(int m, int i)`, сложность вычисления которой ограничена константой:

```
int B(int m, int i)
{
 if ((m%2 == 0) && (m > 2))
  {if ( i < m-1 )
     return i;
   else
     return m-2;
  }
 else
   return m-1;
}
```

Сам алгоритм представлен далее.

Алгоритм проверки корректности поведения.

Входные данные.

Асинхронная модель поведения (P, π) задана:

- массивом асинхронных взаимодействий P (|P| = n);
- двухмерным массивом частичного порядка order[n,n]

- order[i,j] = 1, если P[i] π P[j];
- order[i,j] = 0, в противном случае.

Асинхронная модель требований A = (V, X, Y, Z, E) задана функцией возможных постсостояний $\gamma : V \times ((X \times Y) \; X \; Z) \to 2^V$.

Начальное состояние $v_0 \in V$.

Данные алгоритма.

bound : Nat[n] := { n, n–1, …, 2, 1 }

массив счетчиков числа точек ветвления дерева возможных линеаризаций.

perm : Nat[n] := { 0, 1, …, n–1 }

массив, хранящий текущую перестановку взаимодействий.

current : Nat := 0

индекс текущего элемента перестановки.

path : (V-set)[n+1] := { $\{v_0\}$, ∅, …, ∅ }

массив множеств состояний текущего пути в модели требований. Для детерминированных моделей требований этот массив может быть заменен на массив состояний V, так как все его элементы будут множествами с размерностью не более 1.

processed : Nat := 0

индекс наибольшего элемента пути, содержащего актуальное множество состояний модели требований.

order_failed : Bool := false

флаг, содержащий истинное значение только в том случае, когда текущая перестановка не является линеаризацией частичного порядка π.

current2 : Nat := 0

вспомогательная переменная для хранения индекса второго взаимодействий, участвовавшего в последней транспозиции.

tmp : Nat := 0

вспомогательная переменная.

j : Nat := 0

вспомогательная переменная.

Алгоритм.

1. Алгоритм вычисляет является ли текущая перестановка perm линеаризацией частичного порядка π, записывает результат в переменную order_failed и переходит к шагу 2, если текущая перестановка не противоречит частичному порядку π, или к шагу 3 – в противном случае. При этом предполагается, что упорядочение P[perm[0]], …, P[perm[current]] не противоречит частичному порядку π и поэтому достаточно проверить элементы перестановки, начиная с current.


```
for(;current<n-1;current++)
 {for(j=current+1;j<n;j++)
   {if (order[perm[j],perm[current]])
     {order_failed = true;
      goto 3;
     }
   }
 }
order_failed = false;
```

2. Алгоритм строит пути в модели требований, соответствующие последовательности взаимодействий в текущей перестановке, заполняя массив path. При этом предполагается, что начальные отрезки пути path[0], …, path[processed] уже построены ранее. Если на одном из шагов множество состояний модели требований оказывается пустым, то алгоритм переходит к шагу problem, установив значение переменной current равное индексу элемента перестановки для которого было получено пустое множество. Если же path[n] ≠ ∅, то алгоритм завершает свою работу с положительным вердиктом. Найдена линеаризация (P[perm[0]], P[perm[1]],…, P[perm[n-1]]), для которой множество $\Gamma^*(v_0, (P[perm[0]], P[perm[1]], …, P[perm[n-1]]))$ = path[n] не пусто.

```
for(;processed<n;processed++)
 {path[processed+1] = ∅;
  foreach v : V in path[processed]
    path[processed+1] = path[processed+1]
                              ∪ γ( v,
P[perm[processed]] );
  if (path[processed+1] == ∅)
   {current = processed;
    for(j=current+1;j<n;j++)
      bound[j] = n-j;
    goto 3;
   }
 }
КОНЕЦ (асинхронная модель поведения является
корректной
       относительно данной асинхронной
модели требований)
```

3. Если индекс текущего элемента перестановки больше либо равен n-2 или , то алгоритм переходит к шагу 9.

```
if ((current ≥ n-2)||(bound[current] == 1))
  goto 9;
```

4. Если индекс текущего элемента перестановки является нечетным при обратной нумерации элементов перестановки [n-1 → 0, …, 0 → n-1], то алгоритм переходит к шагу 7.

```
if ((n-current) % 2 == 0)
  goto 7;
```

5. Алгоритм осуществляет циклический сдвиг хвоста перестановки на единицу

perm[current+2] → perm[current+1]

…

perm[n-1] → perm[n-2]

perm[current+1] → perm[n-1]

```
tmp = perm[current+1];
for(j=current+1;j<n-1;j++)
  perm[j] = perm[j+1];
perm[n-1] = tmp;
```

6. Если предыдущая перестановка не противоречила частичному порядку π, то алгоритм вычисляет не противоречит ли новая перестановка этому порядку, записывает результат в переменную order_failed и переходит к шагу 9.

```
if (!order_failed)
 {for(j=current+1;j<n-1;j++)
   {if (order[perm[n-1],perm[j]])
     {order_failed = true;
      break;
     }
   }
 }
goto 9;
```

7. Алгоритм осуществляет транспозицию элементов перестановки с индексами current+1 и current+2.

```
tmp = perm[current+1];
perm[current+1] = perm[current+2];
perm[current+2] = tmp;
```

8. Если предыдущая перестановка не противоречила частичному порядку π, то алгоритм вычисляет не противоречит ли новая перестановка этому порядку и записывает результат в переменную order_failed.

```
if (!order_failed)
 {if
(order[perm[current+2],perm[current+1]])
```



```
    order_failed = true;
 }
```

9. Если текущий счетчик числа точек ветвления равен 1, то алгоритм переходит к шагу 14.

```
if (bound[current] == 1)
  goto 14;
```

10. Алгоритм осуществляет транспозицию элементов перестановки с индексами current и n–B(n–current,n–current–bound[current]+1), сохраняя индекс второго элемента в переменной current2.

```
current2 = n-B(n-current,n-current-
bound[current]+1);
tmp = perm[current];
perm[current] = perm[current2];
perm[current2] = tmp;
```

11. Алгоритм уменьшает текущий счетчик числа точек ветвления на единицу.

```
bound[current]--;
```

12. Если предыдущая перестановка противоречила частичному порядку π, то алгоритм переходит к шагу 1.

```
if (order_failed)
  goto 1;
```

13. В противном случае алгоритм вычисляет не противоречит ли новая перестановка частичному порядку π, записывает результат в переменную order_failed и переходит к шагу 2, если текущая перестановка не противоречит частичному порядку π, или к шагу 3 – в противном случае. В последнем случае алгоритм устанавливает значение переменной current равное наименьшему индексу элемента перестановки, для которого нарушается условие линеаризации.

```
for(j=current+1;j<=current2;j++)
 {if (order[perm[j],perm[current]])
   {order_failed = true;
    goto 3;
   }
 }
for(j=current+1;j<current2;j++)
 {if (order[perm[current2],perm[j]])
   {order_failed = true;
    current = j;
    goto 3;
   }
 }
goto 2;
```

14. Если индекс текущего элемента перестановки равен 0, то алгоритм завершает свою работу с отрицательным вердиктом.

```
if (current == 0)
  КОНЕЦ (асинхронная модель поведения
является
  не корректной относительно данной
асинхронной
  модели требований)
```

15. В противном случае алгоритм переинициализирует текущий счетчик числа точек ветвления и уменьшает индекс текущего элемента перестановки на единицу.

```
bound[current] = n-current;
current--;
```

16. Если индекс текущего элемента перестановки оказался меньше индекса наибольшего элемента пути, содержащего актуальное множество состояний модели требований, то значение последнего устанавливается равным значению первого.

```
if (current < processed)
  processed = current;
```

17. Алгоритм переходит к шагу 9.

```
goto 9;
```

Доказательство корректности данного алгоритма представлено в [30].

Частичный порядок π модели поведения рассматривается в алгоритме в виде двухмерной матрицы n×n, содержащей значение 1 для пар принадлежащих частичному порядку и 0 – в противном случае. Для построения этой матрицы на основе моделей каналов и временных меток можно использовать алгоритм построения транзитивного замыкания Уоршала, который требует $O(n^3)$ операций [29]. Существуют также алгоритмы построения транзитивного замыкания с лучшей оценкой (например, $O(n^{\log 7} \cdot \log n)$), однако они имеют преимущества только при очень больших значениях n.

Обратим внимание на тот факт, что оценка сложности алгоритма O(n!) получена для класса детерминированных моделей поведения, для недетерминированных моделей сложность становится еще больше. Но даже при использовании детерминированных моделей требования применимость алгоритма сильно ограничена: при числе взаимодействий превышающем несколько десятков время работы становится неприемлемо большим. Поэтому при применении рассматриваемого метода необходимо учитывать имеющиеся ограничения на размерность модели поведения, участвующей в задаче оценки корректности.

3.2.9. Требования к полноте набора асинхронных реакций

Асинхронная модель требований позволяет оценить являются ли корректными взаимодействия с целевой системой, произошедшие в процессе тестирования.

В то же время, для асинхронных систем существуют требования другого вида, которые также необходимо проверять. Это требования к полноте набора асинхронных реакций.

Например, рассмотрим ситуацию, когда в ответ на некоторое воздействие целевая система должна выдать две асинхронных реакции, но выдает только одну. В этом случае, асинхронная модель поведения всегда будет удовлетворять модели требований, так как все взаимодействия по отдельности являются корректными. В то же время, набор всех взаимодействий с точки зрения пользователя является некорректным, так как в нем отсутствует часть необходимых асинхронных реакций.

Такого рода несоответствие между формальной оценкой корректности поведения и ожиданиями пользователя происходит по двум причинам:

- из-за отсутствия в асинхронных моделях требований ограничений на момент, когда должна появиться ожидаемая асинхронная реакция;
- из-за отсутствия в асинхронных моделях поведения явного подтверждения того факта, что временная граница получения реакции была пройдена.

В результате этого, при отсутствии одной из ожидаемых асинхронных реакций оценка корректности дает положительный вердикт, так как считается, что данная реакция еще может быть получена позднее.

Для выявления такого рода некорректностей можно предложить следующий подход. К множеству асинхронных реакций асинхронного интерфейса (X, Y, Z) добавляется вспомогательная асинхронная реакция *done*, семантика которой заключается в отражении того факта, что время, отведенное на получение асинхронных реакций, вышло. Соответственно, в асинхронную спецификацию добавляется спецификация псевдо интерфейсной операции $Spec_{done}$, описывающая корректные переходы модели требований, помеченные реакцией *done*. Эти переходы предлагается интерпретировать следующим образом. Если из данного состояния модели требований $v \in V$ существует переход, помеченный асинхронной реакцией $z \in Z$, то

- если из этого состояния существует переход, помеченный реакцией *done*, то мы будем говорить, что асинхронная реакция z является *опциональной* в состоянии v, то есть целевая система может выдать данную реакцию, а может и не выдать.
- если из этого состояния не существует перехода, помеченного реакцией *done*, то мы будем говорить, что асинхронная реакция z является *обязательной* в состоянии v, то есть целевая система обязана выдать данную реакцию.

Для использования псевдо реакции *done* тестовая система должна при завершении тестирования выждать время, необходимое для получения всех асинхронных реакций, и добавить в построенную асинхронную модель поведения псевдо реакцию *done* таким образом, чтобы согласно частичному порядку эта псевдо реакция была больше всех остальных взаимодействий. При

таком подходе удается разрешить рассмотренную выше проблему с проверкой неполноты набора асинхронных реакций.

Заметим, что это возможно только тогда, когда время получения всех асинхронных реакций ограничено, так как в противном случае появление псевдо реакции *done* в асинхронной модели поведения становится невозможным.

Данное ограничение может быть ослаблено, до требования ограниченности времени получения всех обязательных асинхронных реакций, где под обязательными реакциями понимаются реакции, которые являются обязательными хотя бы в одном состоянии модели требований.

Для описания более детальных ограничений, таких как требования вида “асинхронная реакция должна появиться не позже, чем” и “ асинхронная реакция должна появиться не раньше, чем” необходимо отражение выделенных моментов времени как в модели требований, так и в модели поведения. Для решения этой задачи можно предложить два подхода.

Первый подход предполагает введение псевдо асинхронных реакций, предназначенных для симулирования наступления временных границ (нижний или верхней) для каждой конкретной асинхронной реакции. В модели требований такие псевдо реакции должны использоваться для описания соответствующих ограничений на момент появления реакции, а в модели поведения они должны отражать соответствующие моменты времени.

Второй подход основывается на использовании модели временных меток. Для этого к каждому асинхронному взаимодействию приписывается временной интервал, то есть в асинхронном интерфейсе целевой системы множества Y и Z заменяются на $Y \times TI(TM)$ и $Z \times TI(TM)$ соответственно. В этом случае, в спецификациях интерфейсных операций временные метки учитываются при описании требований ко времени получения реакций, посредством сохранения в модельном состоянии информации о времени осуществления соответствующих асинхронных воздействий. Во втором подходе также может потребоваться использование множества псевдо асинхронных реакций $\{ done \} \times TI(TM)$ для отражения момента завершения сбора асинхронных реакций.

Следует также отметить, что работа с временными свойствами обладает целым рядом проблем, связанных со сложностью отслеживания временных характеристик работы целевой системы. Поэтому такая работа должна проводиться с учетом возможных неточностей в измерении. Для этого основным инструментом работы со временем сделан временной интервал, описывающий не единственный момент во времени, а целый интервал, ограниченный минимальной и максимальной временными метками.

Рассмотренные методы организации проверки дополнительных требований к полноте набора асинхронных реакций и времени их появления построены на использовании ранее определенных моделей поведения и требований. Изменения касались только технологии использования этих моделей. Поэтому

для реализации дополнительных проверок никаких изменений в моделях поведения и требований и алгоритмах работы с ними не требуется.

3.2.10. Модели требований и поведения в унифицированной архитектуре асинхронного теста

Компонент тестовой системы, ответственный за взаимодействие с целевой системой и построение модели ее поведения, мы, как и ранее, будем называть *медиатором*. Компонент, решающий задачу оценки корректности поведения тестируемой системы в ходе работы теста, мы будем называть *оракулом*. Рассмотрим более детально их внутреннее устройство и способ взаимосвязи.

Отличия в архитектуре тестовой системы для асинхронных систем диктуются основными особенностями этих систем:

- наличием взаимодействий, инициируемых целевой системой;
- возможностью одновременного участия целевой системы в нескольких взаимодействиях.

При тестировании асинхронных систем их поведение уже не может рассматриваться как набор отдельных, не связанных между собой взаимодействий. Между этими взаимодействиями существуют зависимости, которые необходимо учитывать и при построении модели поведения, и при оценке ее корректности.

Отсюда появляется необходимость в компоненте, в котором будет собираться информация обо всех взаимодействиях с тестируемой системой и зависимостях между ними. Этот компонент называется регистратором взаимодействий. Информация, собираемая в нем, представляет собой модель поведения целевой системы в процессе работы теста. В дальнейшем она будет использоваться для оценки корректности поведения.

Информация о взаимодействиях попадает в регистратор взаимодействий двумя путями. Взаимодействия, инициируемые тестовой системой, регистрируются *медиаторами воздействий*. Помимо регистрации взаимодействий эти медиаторы ответственны за осуществление воздействия на тестируемую систему. Работа медиатора воздействия строится по следующему алгоритму. Получив указание подать стимул $x \in X$, медиатор преобразует его в соответствующее воздействие на целевую систему и осуществляет его. Затем медиатор воздействия получает ответную реакцию целевой системы, преобразуют ее в модельное представление $y \in Y$ и регистрирует взаимодействие $(x, y) \in X \times Y$ в регистраторе взаимодействий.

Взаимодействия, инициируемые целевой системой, регистрируются *кетчерами*, задачей которых является получении информации обо всех таких взаимодействиях, преобразование каждого взаимодействия в модельное представление $z \in Z$ и регистрация его в регистраторе взаимодействий.

Для передачи тестовой системе известной информации о последовательности, в которой происходили взаимодействия, используются модель каналов и модель временных меток. Модель каналов строится следующим образом. При регистрации каждого взаимодействия указывается идентификатор канала, к

которому оно относится. При этом считается, что все взаимодействия, зарегистрированные с одинаковым идентификатором канала, образуют один канал. Линейный порядок в рамках этого канала определяется последовательностью регистрации взаимодействий: если из двух взаимодействий, зарегистрированных с одним идентификатором канала, одно было зарегистрировано раньше другого, то первое меньше второго в линейном порядке соответствующего канала.

Модель временных меток определяется похожим способом. При регистрации каждого взаимодействия ему приписывается временной интервал. Кроме того, в процессе работы теста имеющаяся информация о взаимном порядке временных меток, принадлежащих различным системам координат, должна быть передана регистратору взаимодействий. Таким образом, в регистраторе взаимодействий будет собрано все необходимое для определения модели временных меток.

У медиатора, использовавшегося для тестирования синхронных систем, была еще одна важная функция – синхронизация модельного состояния с состоянием реализации. При тестировании с открытым состоянием синхронизация осуществлялась посредством чтения внутреннего состояния целевой системы и преобразования его в модельное представление. При тестировании с закрытым состоянием синхронизация проводилась путем вычисления модельного состояния, удовлетворяющего модели требований, с учетом дополнительных знаний медиатора о деталях поведения тестируемой реализации.

Тестирование асинхронных систем, как правило, можно проводить только с закрытым состоянием, так как из-за асинхронности поведения не представляется возможным прочитать состояние целевой системы, в котором она находилась после определенного взаимодействия. Поэтому при тестировании асинхронных систем модельное состояние не рассматривается как часть модели поведения целевой системы. Оно используется только в модели требований для определения корректности поведения.

В алгоритме оценки корректности поведения функция γ описывает множество допустимых модельных состояний после данного взаимодействия при заданном начальном состоянии системы. Модель требований может допускать большое количество таких состояний, абстрагируясь от деталей реализации описываемых требований. Анализ всех этих состояний может требовать большого количества ресурсов. В то же время, при тестировании конкретной целевой системы может быть известно о деталях поведения данной реализации, на основе которых можно ограничить множество допустимых модельных состояний только теми состояниями, которые будут соответствовать корректному поведения данной реализации. Так как такое ограничение является реализационнозависимым, то оно описывается в медиаторе, а именно в специальном компоненте медиатора, называемом *медиатором состояния*. Подробнее, назначение медиатора состояния будет рассмотрено при обсуждении работы оракула.

Устройство медиатора, предназначенного для тестирования асинхронных систем, представлена на Рис. 12. Как и ранее, стрелками здесь обозначены направления передачи данных между компонентами тестовой системы.

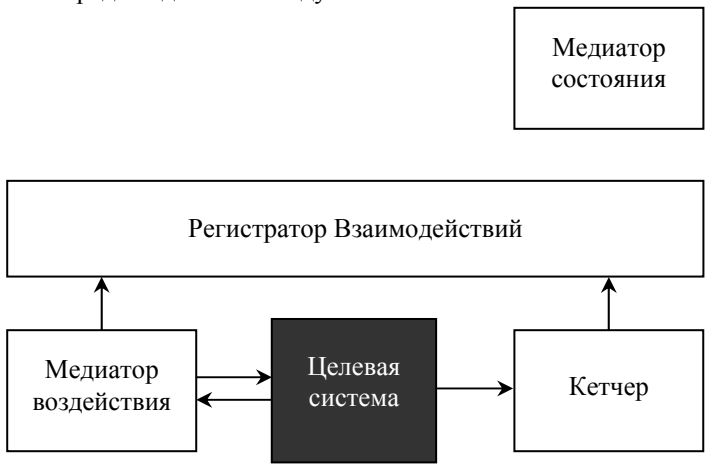

Рис. 12. Медиатор в универсальной архитектуре асинхронного теста

Оракул решает задачу оценки корректности поведения тестируемой системы в ходе работы теста. Алгоритм решения этой задачи был представлен в предыдущем разделе. Он учитывает в своей работе все имевшие место взаимодействия с тестируемой системой и связи между ними. Поэтому компонент оракула, реализующий этот алгоритм, мы будем называть *гипероракулом*. Гипероракул получает от регистратора взаимодействий информацию обо всех взаимодействиях, зарегистрированных в процессе работы теста, и дальше действует согласно алгоритму оценки корректности поведения.

Алгоритм оценки корректности поведения предполагает, что модель требований задана посредством функции $\gamma : V \times ((X \times Y) \, X \, Z) \rightarrow 2^V$, однако в действительности модель поведения описывается асинхронной спецификацией $\{ \mathrm{Spec}_{Si} \mid i = 1, \ldots, k;\ k > 0 \} \cup \{ \mathrm{Spec}_{Rj} \mid j = 1, \ldots, m;\ m \geq 0 \}$. Поэтому необходимо рассмотреть каким образом гипероракул организует вычисление функции γ.

Вычисление функции γ на основе асинхронной спецификации является задачей эквивалентной решению системы уравнений произвольной сложности. И так как автоматически решить эту задачу не представляется возможным, то тестовая система полагается в этом случае на подсказку пользователя. В качестве такой подсказки предлагается использовать функцию $\gamma' : V \times ((X \times Y) \, X \, Z) \rightarrow 2^V$, которая вычисляет конечное множество состояний, включающее в себя все потенциальные элементы $\gamma(v, p)$:

$$\forall v \in V \ \forall p \in ((X \times Y) X Z) \quad |\gamma'(v, p)| < \infty \wedge \gamma(v, p)) \subseteq \gamma'(v, p)$$

Обладая такой подсказкой, тестовая система получает возможность рассмотреть конечное множество потенциальных решений и вычислить искомое множество $\gamma(v, p)$, отсеяв элементы $\gamma'(v, p)$, не удовлетворяющие заданной асинхронной спецификации.

Заметим, что определение функции γ' очень близко к описанию ограничения на множество допустимых модельных состояний, осуществляемое в медиаторе состояния. Действительно, в медиаторе состояния в целях оптимизации предполагается ограничить множество допустимых состояний, что соответствует определению дополнительной функции $\gamma'' : V \times ((X \times Y) X Z) \to 2^V$, учитывающей детали поведения тестируемой реализации и вычисляющей подмножество множества $\gamma(v, p)$:

$$\forall v \in V \ \forall p \in ((X \times Y) X Z) \quad \gamma''(v, p) \subseteq \gamma(v, p)$$

Предполагается, что подмножество $\gamma''(v, p)$ включает в себя не все допустимые модельные состояния, а только те, которые являются корректными, учитывая детали поведения тестируемой реализации. Другими словами, использование функции γ'' позволяет уточнять абстрактную модель требований дополнительными требованиями для конкретной реализации.

В унифицированной архитектуре асинхронного теста предлагается совместить определения уточнения требований и подсказки путем описания в медиаторе состояния функции $\gamma^* : V \times ((X \times Y) X Z) \to 2^V$, результатом которой является конечное множество состояний. Это множество может быть шире множества допустимых состояний, вычисленных γ'' с учетом дополнительных требований. Но только за счет состояний, не удовлетворяющих исходной модели требований.

$$\forall v \in V \ \forall p \in ((X \times Y) X Z)$$
$$\gamma''(v, p) \subseteq \gamma^*(v, p) \subseteq \gamma'(v, p) \ \wedge \ \gamma^*(v, p) \cap \gamma(v, p) = \gamma''(v, p)$$

В простейшем случае, при отсутствии уточнения требований, функция γ^* совпадает с функцией подсказки γ'.

Далее мы будем считать, что размерность множества $\gamma^*(v, p)$ не превышает 1. Это требование является ограничением на область применимости архитектуры. Заметим, что данное ограничение выполняется для всех детерминированных моделей требований и, кроме того, оно выполняется для недетерминированных моделей требований, если требования к тестируемой реализации могут быть уточнены до достижения детерминированности.

Итак, гипероракул вычисляет вместо функции γ ее модификацию $\gamma''(v, p)$, и это вычисление происходит следующим образом:

- Гипероракул устанавливает *модельное состояние* равным v.
- Гипероракул вызывает *оракул воздействия* для взаимодействия p.
- Оракул воздействия вычисляет предусловие взаимодействия p: $pre_{S \cup R}(v,p)$.

- Если нарушено предусловие интерфейсной операции-стимула ($p \in X \times Y$), то тестовая система завершает свою работу[7].
- Если нарушено предусловие интерфейсной операции-реакции ($p \in Z$), то оракул воздействия возвращает отрицательный вердикт и результатом вычисления γ'' является пустое множество.
- Оракул воздействия сохраняет все необходимые части текущего модельного состояния v и вызывает медиатор состояния для взаимодействия p.
- Медиатор состояния читает текущее модельное состояние v и вычисляет $\gamma^*(v, p)$.
- Если $\gamma^*(v, p)$ пусто, то медиатор состояния возвращает отрицательный вердикт и результатом вычисления γ'' является пустое множество.
- В противном случае, множество $\gamma^*(v, p)$ состоит из одного элемента v'. Тогда медиатор состояния записывает v' в модельное состояние и возвращает управление оракулу воздействия с положительным вердиктом.
- Оракул воздействия вычисляет постусловие взаимодействия p для сохраненного состояния v и текущего модельного состояния v': $post_{S \cup R}(v, p, v')$.
- Если постусловие нарушено, то оракул воздействия возвращает отрицательный вердикт и результатом вычисления γ'' является пустое множество
- В противном случае оракул воздействия возвращает положительный вердикт и результатом вычисления γ'' является множество $\{ v' \}$, где v' – текущее значение модельного состояния.

Диаграмма последовательности взаимодействия, описывающая успешное вычисление функции γ'' представлена на Рис. 13.

[7] Более детальное обсуждение проблемы нарушения предусловия интерфейсных операций-стимулов смотрите ниже, в разделе «Нарушение предусловий асинхронных воздействий».

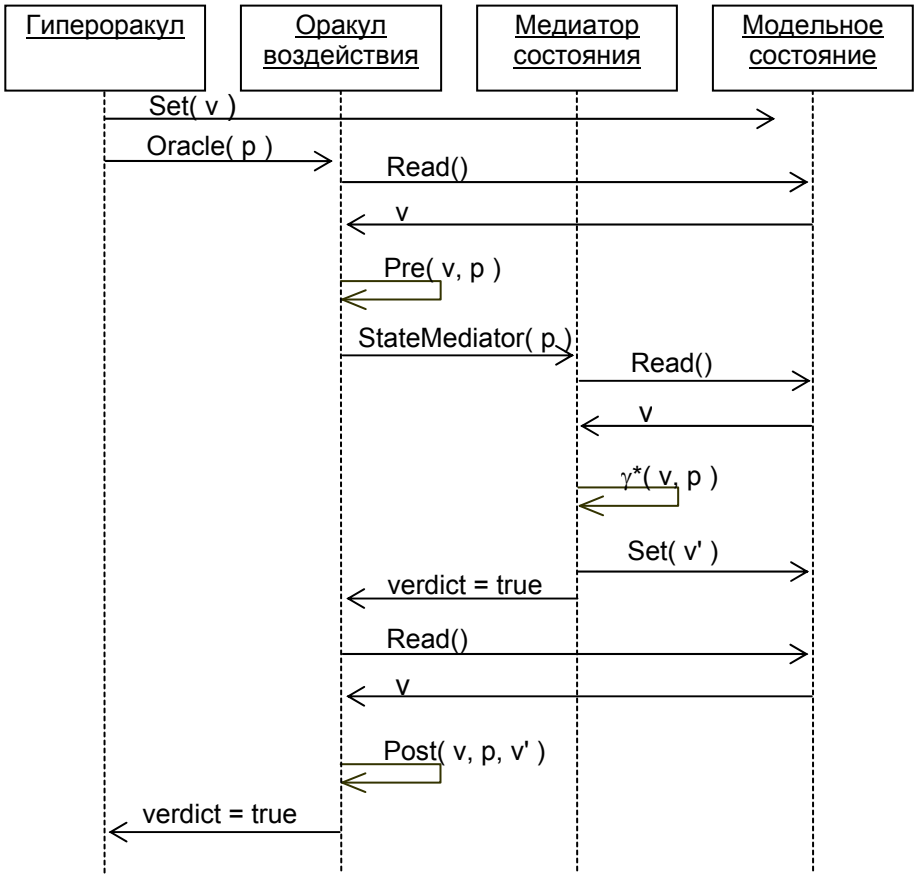

Рис. 13. Диаграмма последовательности взаимодействий, описывающая ход вычисления функции γ'' с непустым результатом

Такая архитектура вычисления функции γ'' позволяет без изменений использовать оракулы воздействий для тестирования как синхронных, так и асинхронных систем.

Схема работы асинхронного теста по оценке корректности поведения целевой системы представлена на Рис. 14. Как и ранее, стрелками здесь обозначены направления передачи данных между компонентами тестовой системы.

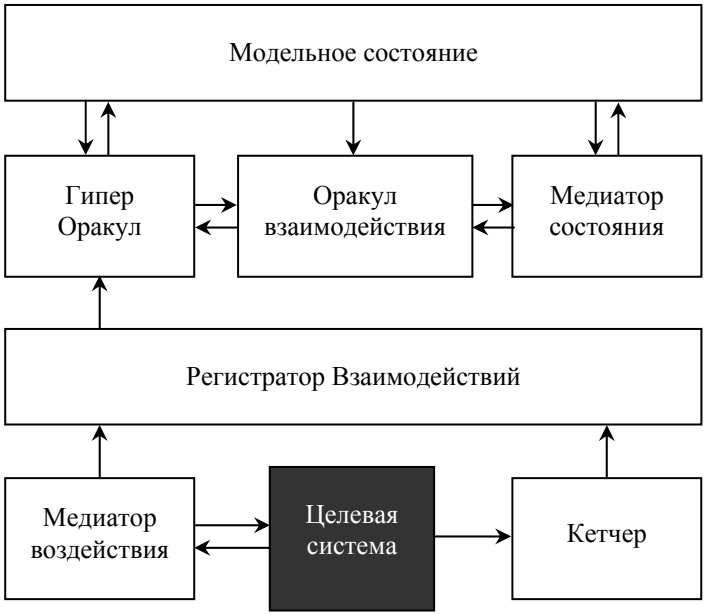

Рис. 14. Схема работы асинхронного теста по оценке корректности поведения целевой системы

3.3. Асинхронные тестовые сценарии

3.3.1. Генерация тестовых данных для асинхронных систем

В данном разделе мы рассмотрим проблему генерации тестовых данных для асинхронных систем. Решение этой проблемы сильно осложняется большой временной сложностью алгоритмов оценки корректности поведения асинхронных систем. Область применимости этих алгоритмов ограничена асинхронными моделями поведения, состоящими из нескольких десятков взаимодействий.

Одного теста такого размера явно не достаточно для достижения приемлемого качества тестирования. Построение наборов независимых тестов небольшого размера приведет к потере одного из основных достоинств технологии UniTesK – автоматическому построению сложных последовательностей тестовых воздействий.

Компромиссное решение, позволяющее совместить генерацию тестовых последовательностей с ограничениями на размеры модели поведения, подвергающейся оценки корректности, было предложено в работе [27]. Идея этого подхода заключается в использовании автоматных тестовых сценариев для построения обхода графа, в котором каждой дуге приписан асинхронный тест небольшого размера. Для реализации такого подхода требуется, чтобы целевая система после получения любого набора асинхронных воздействий за конечное время переходила в состояние, в котором она не должна выдавать ни одной асинхронной реакции до получения следующего воздействия.

С учетом этого ограничения данный подход позволяет сохранить автоматическое построение сложных последовательностей тестовых воздействий, несмотря на ограниченную применимость алгоритма оценки корректности поведения.

В последующих разделах мы рассмотрим вопрос построения тестов для систем с асинхронным интерфейсом более формально.

3.3.2. Взаимодействующие автоматы

Определение 13.

Взаимодействующим автоматом будем называть шестерку (S, s_0, X, Y, E, Q), где

- S – множество состояний,
- $s_0 \in S$ – начальное состояние,
- X – множество стимулов,
- Y – множество реакций,
- $E \subseteq S \times (X \cup Y \cup \{\tau\}) \times S$ – множество переходов, которые мы будем называть
  - *посылающими*, если второй элемент перехода является стимулом,
  - *принимающими*, если второй элемент перехода является реакцией,
  - *внутренним*, если второй элемент перехода является τ;
- $Q \subseteq S$ – множество заключительных состояний.

Взаимодействующий автомат может посылать стимулы и получать реакции, изменяя при этом свое состояние. При возможности выбора очередного перехода этот выбор осуществляется в зависимости от доступности реакций для получения. Выбор перехода среди различных доступных переходов осуществляется недетерминировано.

Первый элемент перехода взаимодействующего автомата мы будем называть пресостоянием перехода, второй – сообщением, а третий – постсостоянием.

Заметим, что множества стимулов и реакций могут пересекаться ($X \cap Y \neq \varnothing$), и поэтому в определении множества переходов используется операция дизъюнктивного объединения. Для обозначения того факта, что сообщение является стимулом мы будем использовать символ !, помещаемый следом за элементом множества X (например, x!), а для обозначения принадлежности сообщения к множеству реакций будет использоваться символ ?: (например, y?).

Конечная последовательность переходов (e_1, e_2, …, e_n) взаимодействующего автомата A = (S, s_0, X, Y, E, Q) называется *путем*, если для каждого $i = 1, \ldots, n-1$ постсостояние перехода e_i совпадает с пресостоянием перехода e_{i+1}. При этом мы будем говорить, что путь ведет из пресостояния перехода e_1 в постсостояние перехода e_n.

Бесконечным путем мы будем называть бесконечную последовательность переходов, любой префикс которой является конечным путем.

Путь (e_1, e_2, …, e_n) взаимодействующего автомата A = (S, s_0, X, Y, E, Q) называется *функционированием*, если пресостоянием перехода e_1 является

начальное состояние s_0, а постсостояние перехода e_n принадлежит множеству конечных состояний Q.

Линейно-упорядоченное мультимножество стимулов и реакций (T, π) называется *трассой* взаимодействующего автомата A = (S, s_0, X, Y, E, Q), если существует такое функционирование (e_1, e_2, …, e_n) автомата A, что

- мультимножество T совпадает с мультимножеством всех сообщений функционирования (e_1, e_2, …, e_n), отличающихся от τ;
- порядок π совпадает с порядком этих сообщений в функционировании (e_1, e_2, …, e_n).

Множество всех возможных трасс взаимодействующего автомата A мы будем обозначать Traces(A).

Определение 14.

Дадим определение *распределенного взаимодействующего автомата* в рекурсивной форме.

- взаимодействующий автомат A = (S, s_0, X, Y, E, Q) является распределенным взаимодействующим автоматом;
- если A_1, A_2, …, A_n – распределенные взаимодействующие автоматы, а M – множество сообщений, то система взаимодействующих автоматов с множеством внутренних сообщений M $(A_1,A_2,\ldots,A_n)\backslash M$ является распределенным взаимодействующим автоматом.

Множеством стимулов распределенного взаимодействующего автомата DA будем называть

- множество стимулов X, если автомат DA является взаимодействующим автоматом A = (S, s_0, X, Y, E, Q),
- $\bigcup_i X_i \setminus M$, если автомат DA является системой взаимодействующих автоматов A_1, A_2, …, A_n с множеством внутренних сообщений M.

Аналогично, множеством реакций распределенного взаимодействующего автомата DA будем называть

- множество реакций Y, если автомат DA является взаимодействующим автоматом A = (S, s_0, X, Y, E, Q),
- $\bigcup_i Y_i \setminus M$, если автомат DA является системой взаимодействующих автоматов A_1, A_2, …, A_n с множеством внутренних сообщений M.

В системе взаимодействующих автоматов с множеством внутренних сообщений M все взаимодействующие автоматы обмениваются сообщениями из множества M только между собой и не могут обмениваться ими с окружением.

Предположим, что взаимодействующий автомат A = (S, s_0, X, Y, E, Q) входит в состав системы взаимодействующих автоматов $SA = (A_1,A_2,\ldots,A_n)\backslash M$. Тогда мы будем говорить, что сообщение $m \in X \cup Y$ автомата A *сокрыто* в SA, если $m \in M$ и не существует системы взаимодействующих автоматов $SA' = (A'_1,A'_2,\ldots,A'_{n'})\backslash M'$, входящей в SA, такой что $m \in M'$ и A входит в состав SA'.

Предположим, что взаимодействующий автомат $A = (S, s_0, X, Y, E, Q)$ входит в состав распределенного взаимодействующего автоматов DA. Переходы автомата A $(s_1, m, s_2) \in E$ будем называть *внутренними* для DA, если сообщение m сокрыто в одной из систем взаимодействующих автоматов, входящей в состав DA. В противном случае, переходы будем называть *внешними* для DA.

Функционированием распределенного взаимодействующего автомата DA называется тройка (E, π, μ), где

- E – набор функционирований $\{ (e_{i,1}, e_{i,2}, \ldots, e_{i,n(i)}) \}$ всех взаимодействующих автоматов, входящих в DA;
- π – частичный порядок на множестве всех переходов $\{ e_{i,j} \}_{i, j=1,\ldots,n(i)}$ набора E;
- $\mu : \{ e_{i,j} \} \to \{ e_{i,j} \}$ – отображение на этом же множестве переходов;

при условии, что E, π и μ удовлетворяют следующим ограничениям:

- областью определения μ является множество всех принимающих переходов $e_{i,j} = (s_1, m?, s_2)$ автомата A, сообщение m которых сокрыто в некоторой системе взаимодействующих автоматов SA, входящей в состав DA;
- для каждого такого перехода $e_{i,j} = (s_1, m?, s_2)$ значением $\mu(e_{i,j})$ является посылающий переход $e_{i',j'} = (s'_1, m!, s'_2)$ автомата A', входящего в состав той же системы взаимодействующих автоматов SA, но не совпадающего с A ($A \neq A'$);
- частичный порядок π сохраняет порядок переходов, относящихся к функционированию одного взаимодействующего автомата, то есть
  $\forall e_{i,j}, e_{i',j'} \in \{ e_{i,j} \} \ (i = i') \wedge (j < j') \Rightarrow e_{i,j} \ \pi \ e_{i',j'}$;
- переходы, связанные отношением μ, являются неразличимыми относительно частичного порядка π, то есть
  $\forall e_{i,j}, e_{i',j'} \in \{ e_{i,j} \} \ e_{i',j'} = \mu(e_{i,j}) \Rightarrow$
  $\forall e \in \{ e_{i,j} \} \ ((e \ \pi \ e_{i,j}) \Leftrightarrow (e \ \pi \ e_{i',j'})) \wedge ((e_{i,j} \ \pi \ e) \Leftrightarrow (e_{i',j'} \ \pi \ e))$.

Распределенный взаимодействующий автомат функционирует таким образом, что его компоненты обмениваются внутренними сообщениями между собой, а остальными сообщениями как между собой так и с окружением. Функционирование такого автомата состоит из набора функционирований составляющих его взаимодействующих автоматов, а также из частичного порядка, определяющего порядок осуществления переходов во времени, и отображения μ, связывающее каждое получение внутреннего сообщения с его посылкой.

Заметим, что одной посылке сообщения может соответствовать несколько получений этого сообщения, то есть посылка сообщений может быть широковещательной. Все взаимодействия между автоматами осуществляются синхронным образом, а асинхронные взаимодействия должны моделироваться явно, например, при помощи дополнительного автомата моделирующего работу канала связи или буфера.

Частично упорядоченное мультимножество стимулов и реакций (T, π_T) называется *трассой* распределенного взаимодействующего автомата DA, если существует такое функционирование (E, π, μ) автомата DA, что

- мультимножество T совпадает с мультимножеством всех сообщений набора функционирований $\{ (e_{i,1}, e_{i,2}, \ldots, e_{i,n(i)}) \}$, отличающихся от τ и не являющихся внутренними для DA;
- частичный порядок π_T является сужением порядка π на переходах из $\{ (e_{i,1}, e_{i,2}, \ldots, e_{i,n(i)}) \}$, соответствующих этим сообщениям, то есть
  $\forall t_1, t_2 \in T \ (t_1 \ \pi_T \ t_2) \Leftrightarrow (e(t_1) \ \pi \ e(t_2))$.

Множество всех возможных трасс распределенного взаимодействующего автомата DA мы будем обозначать Traces(DA).

3.3.3. Асинхронные функции

Для вычисления функций в распределенных взаимодействующих автоматах реализуется посредством использования выделенных взаимодействующих автоматов, получающих входные параметры функции и возвращающих результат.

Асинхронной функцией AF из множества входных значений IS во множество выходных значений OS называется взаимодействующий автомат $SA = (S, s_0, X, Y, E, Q)$, в котором

- $PY = \{ call_{AF,is} \mid is \in IS \} \subseteq Y$ – множество входных значений параметризует выделенное подмножество реакций автомата,
- $RX = \{ result_{AF,os} \mid os \in OS \} \subseteq X$ – множество выходных значений параметризует выделенное подмножество стимулов автомата,
- $RX \cap PY = \varnothing$,
- $\forall (s, m, s') \in E \ (s = s_0) \Rightarrow m \in PY$,
- $\forall (s, m, s') \in E \ m \in PY \Rightarrow \forall m' \in PY \ \exists s'' \in S: (s, m', s'') \in E$,
- $\forall (s, m, s') \in E \ (m \in RX) \Rightarrow \forall (s', m', s'') \in E \ m' \in PY$,
- $\forall$ пути $(e_1, e_2, \ldots, e_n)$ в SA $(n > 1)$
  $e_1 = (s_1, py_1, s'_1) \wedge e_n = (s_n, py_n, s'_n) \Rightarrow \exists j \in \{ 2, ..., n-1 \}$
  $e_j = (s_j, rx, s'_j)$,
- $\forall q \in Q \ \forall (s, m, s') \in E \ (s' = q) \Rightarrow m \in RX$.

Данные ограничения определяют ту специфическую роль, которую играют сообщения из PY и RX в работе автомата SA. Эта роль заключается в том, что функционирование автомата SA состоит из последовательных циклов, каждый из которых начинается получением параметризующей реакции $call_{AF,ip}$ и завершается посылкой результирующего стимула $result_{AF,op}$. Внутри этого цикла автомат SA произвольным образом взаимодействует со своим окружением, но только посредством сообщений, не входящих в $PY \cup RX$.

Множество всех асинхронных функций, у которых множество входных значений равно IS, а множество выходных значений равно OS, мы будем обозначать как Ã(IS,OS).

Чтобы работа распределенного взаимодействующего автомата завершилась, необходимо, чтобы все составляющие его автоматы перешли в заключительное состояние. Для этого вместо асинхронных функций обычно используют их замыкания по выделенному множеству терминирующих реакций.

Замыканием взаимодействующего автомата (S, s_0, X, Y, E, Q) по множеству реакций R мы будем называть взаимодействующий автомат (S', s'_0, X', Y', E', Q'), в котором

- S' = S X {final},
- $s'_0 = s_0$,
- X' = X,
- Y' = Y X R,
- E' = E ∪ { (s, r, final) | ∀ s ∈ S, ∀ r ∈ R },
- Q' = Q ∪ {final}.

Замыкание взаимодействующего автомата дополняет исходный автомат специальным завершающим состоянием и множеством переходов, ведущих в это состояние изо всех остальных состояний по получению реакции из множества R.

3.3.4. Асинхронные тесты

Асинхронным тестом целевой системы с асинхронным интерфейсом (X, Y, Z) называется конечное или бесконечное частично упорядоченное мультимножество асинхронных взаимодействий (P, π), которое совпадает с моделью поведения целевой системы в ходе этого теста.

Асинхронный тест (P, π) целевой системы с асинхронным интерфейсом (X, Y, Z) будем называть *успешным* относительно асинхронной модели требований A = (V, X, Y, Z, E) с начальным состоянием $v_0 \in V$, если модель поведения в ходе теста (P, π) удовлетворяет модели требований A с начальным состоянием v_0. В противном случае, тест будет называться *неуспешным*.

Определение 15.

Асинхронным тестовым сценарием для целевой системы с асинхронным интерфейсом (X, Y, Z) называется распределенный взаимодействующий автомат DA, в котором

- множество стимулов совпадает с X;
- множество реакций совпадает с Y X Z;
- для любого взаимодействующего автомата A' = (S', s'_0, X', Y', E', Q'), входящего в состав DA, для любого внешнего для DA посылающего перехода (s_1, x!, s_2) ∈ E', сообщение x которого принадлежит X ∩ X', любой переход (s_2, m, s_3) ∈ E', начинающийся в состоянии s_2, является принимающим и внешним для DA, а его сообщение m принадлежат множеству Y;

- для любого взаимодействующего автомата A' = (S', s'_0, X', Y', E', Q'), входящего в состав DA, для любого внешнего для DA принимающего перехода (s_2, y?, s_3) ∈ E', сообщение y которого принадлежит Y ∩ Y', любой переход (s_1, m, s_2) ∈ E', завершающийся в состоянии s_2, является посылающим и внешним для DA.

Состояния взаимодействующих автоматов, из которых выходят только принимающие переходы, помеченные сообщениями из Y, будут далее называться *промежуточными*.

Асинхронный тестовый сценарий предназначен для управлением процессом тестирования систем с асинхронным интерфейсом. Такой тестовый сценарий подает тестовые стимулы целевой системе и получает от нее непосредственные и отложенные реакции, используя эту информацию для продолжения тестирования.

Множество всех асинхронных тестовых сценариев для целевой системы с асинхронным интерфейсом (X, Y, Z) будем обозначать символом ℑ(X, Y, Z).

Асинхронный тест (P, π) называется *результатом применения* тестового сценария DA для целевой системы с асинхронным интерфейсом (X, Y, Z) к соответствующей целевой системе, если (P, π) ∈ Traces(DA).

Асинхронный тестовый сценарий определяет правила посылки тестовых стимулов, которые могут зависеть от ответа целевой системы. Результатом работы сценария является асинхронный тест, состоящий из набора взаимодействий с целевой системой и информации о порядке, в котором происходили эти взаимодействия. Этот порядок в точности соответствует порядку взаимодействий между тестовым сценарием и целевой системой, наблюдаемому в процессе работы тестового сценария.

При наличии некоторой среды между тестовой системой и целевой системой, например, сетевого соединения, которое не гарантирует сохранения порядка сообщений, предполагается, что эта среда включается в тестовый сценарий и моделируется в нем явно. В качестве альтернативы включения среды между тестовой системой и целевой системой в состав тестового сценария можно использовать включение этой среды в состав целевой системы, но в этом случае все требования к поведению целевой системы необходимо формулировать с учетом поведения этой среды.

Определение 16.

Механизмом построения асинхронного тестового сценария будем называть функцию, значением которой является асинхронный тестовый сценарий.

Также как и для обычных тестовых сценариев, для асинхронных тестовых сценариев мы определим автоматный механизм их построения, который обеспечивает автоматическое формирование последовательности тестовых воздействий на основе обхода ориентированного графа.

3.3.5. Автоматный механизм построения асинхронного тестового сценария

Асинхронным автоматным тестовым сценарием для целевой системы с асинхронным интерфейсом (X, Y, Z) называется пятерка (DA, A_{fsm}, IG, vg_0, α), где

- DA – асинхронный тестовый сценарий для целевой системы с асинхронным интерфейсом (X, Y, Z),
- $A_{fsm} = (S_{fsm}, s_0, X_{fsm}, Y_{fsm}, E_{fsm}, Q_{fsm})$ – выделенный взаимодействующий автомат, входящий в состав DA, и называемый *ведущим автоматом* асинхронного тестового сценария DA,
- IG = (VG, XG, π) – неизбыточное описание ориентированного графа, называемого *графом сценария*,
- $vg_0 \in VG$ – начальное состояние графа сценария,
- α – неизбыточный алгоритм движения по графу сценария;

и для которой выполнены следующие ограничения:

- $S_{fsm} = (VG \times EG^* \times (XG \; X \; \{ \varepsilon \})) \; X \; \{ final \}$ – состояние ведущего автомата является либо выделенным состоянием final, либо содержит:
  - текущую вершину графа сценария vg,
  - пройденный маршрут в графе сценария path (здесь множество EG^* обозначает множество всех конечных списков элементов из множества дуг $EG = VG \times XG \times VG$),
  - последний посланный стимул графа, на который не был получен ответ, либо ε, обозначающее отсутствие такового стимула;
- $s_0 = (vg_0, \langle\rangle, \varepsilon)$ – начальное состояние ведущего автомата состоит из
  - начальной вершины графа сценария,
  - пройденного пути, равного пустому списку,
  - специального значения ε;
- $X_{fsm} = \{ start_{vg,xg} \mid vg \in VG, \; xg \in XG \} \cup \{ stop \}$ – множество стимулов ведущего автомата состоит из набора стимулов, помеченных вершиной и стимулом графа, и из дополнительного стимула, символизирующего завершение работы, причем все эти стимулы являются внутренними для DA;
- $Y_{fsm} = \{ finish_{vg} \mid vg \in VG \} \cup \{ abort \}$ – множество реакций ведущего автомата состоит из набора реакций, помеченных вершинами графа, и из дополнительной реакции, символизирующей аварийное завершение работы, причем все эти реакции являются внутренними для DA;

- $E_{fsm} = \{ ((vg, path, \varepsilon), start_{vg,xg}, (vg, path, xg)) \mid xg = \alpha(IG, vg, path) \}$
  $\cup$
  $\{ ((vg, path, \varepsilon), stop, final) \mid \alpha(IG, vg, path) = \tau \}$
  $\cup$
  $\{ ((vg, path, xg), finish_{vg*}, (vg', path', \varepsilon) \mid$
  $vg' = vg^* \wedge$
  $path' = path \; ^\wedge \; (vg, xg, vg')$
  $\}$
  $\cup$
  $\{ ((vg, path, xg), abort, final) \}$
- $Q_{fsm} = \{ final \}$ – множество заключительных состояний ведущего автомата состоит из единственного состояния final.

Автоматным механизмом построения асинхронного тестового сценария называется функция, которая преобразует асинхронный автоматный тестовый сценарий (DA, A_{fsm}, IG, vg_0, α) в асинхронный тестовый сценарий DA.

Работа автоматного тестового сценария устроена следующим образом. Граф сценария описывает некоторым образом пространство тестовых ситуаций. Ведущий автомат двигается по данному графу, используя заданный алгоритм движения.

Алгоритм движения выбирает стимул графа, который необходимо подать в текущей вершине. Ведущий автомат посылает сообщение, соответствующее текущей вершине и выбранному стимулу. Это сообщение обрабатывается другими взаимодействующими автоматами, входящими в состав DA, и преобразуется в набор воздействий на целевую систему. По завершении обработки этого сообщения, ведущему автомату посылается либо новая вершина графа, либо уведомление об аварийном завершении работы. В первом случае, ведущий автомат повторяет рассмотренный цикл до тех пор, пока алгоритм движения не завершит свою работу. Во втором случае, работа тестового сценария завершается.

3.3.6. Асинхронные сценарные функции

Для описания стимулов графа автоматного сценария в асинхронном случае так же, как и в синхронном, используется понятие сценарной функции. В данном случае, в качестве сценарной функции выступает обычная асинхронная функция, а множество допустимых стимулов определяется на основе множества ее входных параметров.

Асинхронной сценарной функцией мы будем называть асинхронную функцию, в которой множество входных значений может быть произвольным, а множество выходных значений Bool состоит из двух элементов true и false.

Множество всех асинхронных сценарных функций, у которых множество входных значений равно IS, мы будем обозначать как $\Sigma(IS)$.

Асинхронным автоматным тестовым сценарием со сценарными функциями для целевой системы с асинхронным интерфейсом (X, Y, Z) называется семерка (DA, A_{fsm}, VG, XG, vg_0, α, $\{ \Sigma_i \}_{i=1}^{k}$), где

- DA – асинхронный тестовый сценарий для целевой системы с асинхронным интерфейсом (X, Y, Z),
- $A_{fsm} = (S_{fsm}, s_0, X_{fsm}, Y_{fsm}, E_{fsm}, Q_{fsm})$ – выделенный взаимодействующий автомат, входящий в состав DA, и называемый *ведущим автоматом* асинхронного тестового сценария DA,
- VG – множество состояний графа сценария,
- XG – множество стимулов графа сценария,
- vg_0 – начальное состояние графа сценария,
- α – неизбыточный алгоритм движения по графу сценария,
- $\{ \Sigma_i \}_{i=1}^{k}$ – конечный набор асинхронных сценарных функций $\Sigma_i = (S_i, s_{0,i}, X_i, Y_i, E_i, Q_i) \in \Sigma(IS_i)$;

и для которой выполнены следующие ограничения:

- замыкания взаимодействующих автоматов Σ_i по множеству { stop, abort } входят в состав DA, причем все эти вхождения и вхождение A_{fsm} в DA различаются между собой,
- $\forall\, i \in \{ 1, \ldots, k \}$ $IS_i \subseteq VG \times XG$,
- $\forall\, i, j \in \{ 1, \ldots, k \}$ $i \neq j \Rightarrow XG_i \cap XG_j = \varnothing$,
  где $XG_i = \{ xg \in XG \mid \exists\, vg \in VG:\ (vg, xg) \in IS_i \}$,
- $X_{fsm} = \mathbf{Y}_{i=1}^{k} PY_i \cup \{ stop \}$, где $PY_i = \{ call_{i,is} \mid is \in IS_i \}$,
- сообщения из $\mathbf{Y}_{i=1}^{k} PY_i$ присутствуют только в ведущем автомате и автоматах-сценарных функциях,
- пятерка (DA, A_{fsm}, IG, vg_0, α), в которой IG состоит из
  - множества состояний графа сценария VG,
  - множества стимулов графа сценария XG,
  - функции π: $\pi(vg) = \{ xg \in XG \mid \exists\, i \in \{ 1, \ldots, k \}:$ $(vg, xg) \in IS_i \}$,

  является асинхронным автоматным тестовым сценарием для целевой системы с асинхронным интерфейсом (X, Y, Z).

3.3.7. Стационарное тестирование асинхронных систем

Тестирование систем с асинхронным интерфейсом значительно осложнено большой временной сложностью алгоритмов оценки корректности поведения тестируемой системы. Область применимости этих алгоритмов ограничена асинхронными моделями поведения, состоящими из нескольких десятков взаимодействий.

В большинстве случаев одного теста такого размера оказывается недостаточно для достижения требуемого качества тестирования. Как следствие, появляется

необходимость в построении набора тестов небольшого размера. Если разрабатывать такие наборы тестов независимо друг от друга, то при этом теряется одно из основных достоинств технологии UniTesK – автоматическое построении сложных последовательностей тестовых воздействий.

Компромиссное решение, позволяющее совместить генерацию тестовых последовательностей с ограничениями на размеры модели поведения, подвергающейся оценки корректности, было предложено в работе [27]. Идея этого подхода заключается в использовании автоматных тестовых сценариев для построения обхода графа, в котором каждой дуге соответствует асинхронный тестовый сценарий небольшого размера.

Оценка корректности поведения целевой системы, в этом случае, выполняется каждый раз после завершения работы локального тестового сценария. Так как размеры такого сценария небольшие, то алгоритм оценки корректности поведения обрабатывает его результаты за приемлемое время. По завершении этой проверки автоматный тестовый сценарий продолжает обход графа, то есть выбирает следующую дугу и выполняет соответствующий локальный тестовый сценарий.

В результате, такое решение сохраняет генерацию последовательности тестовых воздействий, позволяющую автоматически выполнять набор тестовых воздействий в различных состояниях графа, и в то же время обходит ограничения алгоритма оценки корректности поведения.

В настоящей работе мы ограничимся рассмотрением случае, когда во время проверки корректности поведения после завершения локального сценария тестируемая система не выполняет никаких активных действий. Это означает, что при корректном поведении целевой системы ожидается, что после завершения работы локального тестового сценария целевая система не будет посылать никаких отложенных реакций. Или другими словами, предполагается, что тестируемая система находится в стационарном состоянии.

Определение 17.

Стационарными состояниями асинхронной модели требований (V, X, Y, Z, E) называются такие состояния $v \in V$, из которых выходят только переходы, помеченные стимулом и непосредственной реакцией, то есть $\forall\, (v, p, v') \in E$ $p \in X \times Y$.

Определение 18.

Тестируемые системы мы будем называть *пассивными*, если после подачи произвольного набора стимулов в произвольном состоянии тестируемая система за конечное время переходит в стационарное состояние.

На первый взгляд пассивные тестируемые системы составляют лишь небольшую часть всех асинхронных систем, но с точки зрения предлагаемого подхода бóльшая часть "активных" систем может быть переведена в класс пассивных. Действительно, если асинхронная система проявляет свою активность через некоторые промежутки времени, то есть периодически

посылает сообщения, не инициированные внешним воздействием, то такое поведение можно моделировать как переход системы в стационарное состояние, получение ею специального стимула "время" и последующую посылку отложенных реакций. В этом случае, тестируемая система может рассматриваться как пассивная, если времени до очередной отложенной реакции достаточно, чтобы тестовая система завершила оценку корректности поведения и инициировала следующий локальный тестовый сценарий. Таким образом, класс пассивных асинхронных систем является достаточно большим и далее мы будем рассматривать тестирование только таких целевых систем.

Тестирование пассивных целевых систем, построенное по рассмотренным выше принципам, мы будем называть *стационарным тестированием*.

Если после завершения работы локального тестового сценария некоторые ожидаемые отложенные реакции не были получены, то состояние модели требований не будет стационарным. Таким образом, проверка стационарности состояния модели требований после завершения работы локального тестового сценария может служить решением проблемы проверки полноты набора отложенных реакций, обсуждавшийся ранее в разделе «Требования к полноте набора асинхронных реакций» (с. 67).

С другой стороны, непосредственная проверка стационарности не слишком удобна при наличии опциональных отложенных реакций, которые могут быть посланы тестируемой системой, а могут и отсутствовать. Для описания таких реакций разработчику теста придется вводить псевдостимул *done*, описывающий завершение ожидания опциональных реакций.

Чтобы упростить работу разработчика тестов мы определим неявные трансформации асинхронной модели требований и асинхронной модели поведения, построенной по результатам работы локального сценария. При помощи данных трансформаций мы решим проблему проверки полноты набора отложенных реакций, включающего как обязательные, так и опциональные реакции.

Стабилизирующей трансформацией $ST[A,V_S]$ асинхронной модели требований $A = (V, X, Y, Z, E)$ относительно множества стабильных состояний $V_S \subseteq V$ будем называть асинхронную модель требований $A' = (V', X', Y', Z', E')$, в которой:

- множество состояний $V' = V \times \{T,F\}$, первую составляющую которого мы будем называть *основной*,
- множество стимулов $X' = X \cup \{done\}$,
- множество непосредственных реакций $Y' = Y \cup \{\varepsilon\}$,
- множество отложенных реакций $Z' = Z$,
- множество переходов $E' = \{ ((v_1,T), p, (v_2,F)) \mid (v_1, p, v_2) \in E \wedge p \in X \times Y \} \cup \{ ((v_1,F), p, (v_2,F)) \mid (v_1, p, v_2) \in E \} \cup \{ ((v,F), (done,\varepsilon), (v,T)) \mid v \in V_S \}$.

Заметим, что все состояния стабилизирующей трансформации вида (v,T) гарантированно являются стационарными, так как из них, по определению, не

выходит переходов, помеченных отложенными реакциями. Семантика множества стабильных состояний заключается в том, что стабильными состояниями считаются такие состояния, в которых не ожидается ни одной обязательной отложенной реакции.

Стабилизирующей трансформацией $ST[(P,\pi)]$ асинхронной модели поведения (P, π) будем называть асинхронную модель поведения (P', π'), в которой:

- $P' = P \cup \{(done,\varepsilon)\}$,
- $\pi' = \pi \cup \{ (p, (done,\varepsilon)) \mid p \in P \}$.

Стабилизирующая трансформация модели поведения содержит дополнительное псевдовзаимодействие $(done,\varepsilon)$, которое является максимальным элементом относительно частичного порядка π'.

Стабилизирующие трансформации обоих моделей единообразно модифицируют асинхронный интерфейс целевой системы, добавляя в него псевдостимул *done* и непосредственную реакцию ε. Таким образом, если исходные модели были построены для общего асинхронного интерфейса (X, Y, Z), то их стабилизирующие трансформации будут для общего асинхронного интерфейса $(X \cup \{done\}, Y \cup \{\varepsilon\}, Z)$.

При реализации стационарного тестирования на основе асинхронного автоматного тестового сценария со сценарными функциями, в качестве локальных тестовых сценариев будут выступать автоматы-сценарные функции, а построение обхода графа сценария будет выполнять ведущий автомат. Для оценки корректности поведения целевой системы после работы локального сценария мы введем дополнительный взаимодействующий автомат, который будем называть *стабилизирующим*. Этот автомат будет получать сообщения о завершении очередного цикла работы автомата-сценарной функции ($rx \in RX_i$), инициировать проверку корректности поведения целевой системы и в случае ее успешного завершения вычислять новую вершину графа сценария и передавать ее ведущему автомату. В случае обнаружения некорректности поведения целевой системы стабилизирующий автомат будет посылать сообщение abort, завершая таким образом работу тестового сценария.

Входными данными для алгоритма проверки корректности поведения являются конечная асинхронная модель поведения целевой системы (P, π) и асинхронная модель требований $A = (V, X, Y, Z, E)$ с начальным состоянием $v_0 \in V$. Модель поведения содержит информацию о взаимодействиях с целевой системой в процессе работы автомата-сценарной функции и строится динамически. Модель требований задается заранее и является неизменной на протяжении всей работы тестового сценария. А вот начальное состояние модели требований изменяется в процессе тестирования и соответствует текущему состоянию целевой системы.

Единственность такого состояния накладывает дополнительное ограничение на работу тестового сценария. От него требуется, чтобы все корректные линеаризации асинхронной модели поведения P завершались в одном и том же состоянии, то есть

$$\left| \bigcup_{(p_1,\ldots,p_n)\ -\ линеаризация\ P} \Gamma^*(v_0,(p_1,\ldots,p_n)) \right| \le 1$$

В случае выполнении данного ограничения при нахождении корректной линеаризации переменная алгоритма path[n] будет содержать всегда одно и то же состояние, которое мы будем называть *конечным состоянием линеаризации*. Это состояние будет выступать в качестве начального состояния модели требований для следующего цикла работы автомата-сценарной функции.

Заметим, что алгоритм проверки корректности поведения легко обобщается на случай, когда начальное состояние модели требований задано неоднозначно. То есть, когда вместо конкретного начального состояния v_0 задано множество возможных начальных состояний V_0. Для этого требуется только изменить инициализацию первого элемента массива path: вместо инициализации path[0] множеством, содержащим начальное состояние $\{v_0\}$, использовать инициализацию множеством возможных начальных состояний V_0.

Но в настоящей работе мы не будем рассматривать способы построения тестовых сценариев без данного ограничения, а оставим эту задачу в качестве направления дальнейшего развития.

3.3.8. Стационарный автоматный тестовый сценарий

Рассмотрим реализацию предложенного подхода более формально.

Стационарным автоматным тестовым сценарием относительно асинхронной модели требований A = (V, X, Y, Z, E) с множеством стабильных состояний $V_S \subseteq V$ и начальным состоянием $v_0 \in V_S$, называется пятерка (SA, A_{St}, A_{IR}, A_{HO}, A_{GS}), где

- SA = (DA, A_{fsm}, VG, XG, vg_0, α, $\{ \Sigma_i \}_{i=1}^{k}$) – асинхронный автоматный тестовый сценарий со сценарными функциями для целевой системы с асинхронным интерфейсом (X, Y, Z),
- A_{St} = (S_{St}, $s_{0,St}$, X_{St}, Y_{St}, E_{St}, Q_{St}) – выделенный взаимодействующий автомат, входящий в состав DA, и называемый *стабилизирующим автоматом* асинхронного тестового сценария DA,
- $A_{IR} \in$ Ã(Unit, ℘(Ω(X,Y,Z))) – асинхронная функция, замыкание которой по множеству { stop, abort } входит в состав DA,
- $A_{HO} \in$ Ã(℘(Ω(X,Y,Z)) × V, Bool × V)[8] – асинхронная функция, замыкание которой по множеству { stop, abort } входит в состав DA,
- $A_{GS} \in$ Ã(Unit, VG) – асинхронная функция, замыкание которой по множеству { stop, abort } входит в состав DA;

[8] Здесь и далее идентификатор Unit будет обозначать множество, состоящее из единственного элемента ε, а идентификатор Bool – множество, состоящее из двух элементов true и false.

и для которой выполнены следующие ограничения:

- вхождения замыканий взаимодействующих автоматов A_{IR}, A_{HO} и A_{GS} и вхождения A_{St}, A_{fsm} и Σ_i в DA различаются между собой,
- S_{St} = ({ s_0, s_1, s_2, s_4, s_5, s_6, final } ∪ Ω(X,Y,Z)) × V,
- $s_{0,St}$ = (s_0, v_0),
- X_{St} = { $call_{IR}$, $call_{GS}$, abort } ∪ { $call_{HO,P,v}$ | P ∈ Ω(X,Y,Z), v ∈ V },
- $Y_{St} = \bigcup_{i=1}^{k} RX_i$ ∪ { stop, abort } ∪ { $result_{HO,verdict,v}$ | verdict ∈ Bool, v ∈ V } ∪ { $result_{IR,P}$ | P ∈ Ω(X,Y,Z) },
  где RX_i = { $result_{i,true}$, $result_{i,false}$ },
- E_{St} = { ((s_0, v) $result_{i,true}$?, (s_1, v)) | i = 1, …, k }
  ∪ { ((s_1, v), $call_{IR}$!, (s_2, v)) }
  ∪ { ((s_2, v), $result_{IR,P}$?, (P, v)) | P ∈ Ω(X,Y,Z) }
  ∪ { ((P, v), $call_{HO,P,v}$!, (s_4, v)) | P ∈ Ω(X,Y,Z) }
  ∪ { ((s_4, v), $result_{HO,true,v'}$?, (s_5, v')) }
  ∪ { ((s_4, v), $result_{HO,false,v'}$?, (s_6, v)) }
  ∪ { ((s_5, v), $call_{GS}$!, (s_0, v)) }
  ∪ { ((s_6, v), abort!, (final, v)) }
  ∪ { ((s_0, v), $result_{i,false}$?, (s_6, v)) | i = 1, …, k }
  ∪ { ((s_0, v), stop?, (final, v)) }
  ∪ { ((s_0, v), abort?, (final, v)) }
- Q_{St} = { (final, v) | v ∈ V },
- Y_{fsm} = { $result_{GS,vg}$ | vg ∈ VG } ∪ { abort },
- асинхронная функция A_{IR} возвращает асинхронную модель поведения, описывающую все взаимодействия с целевой системой, зафиксированные за время работы сценарной функции,
- асинхронная функция A_{HO} получает асинхронную модель поведения P и текущее состояние модели v и возвращает true, если стабилизирующая трансформация этой модели ST[P] удовлетворяет стабилизирующей трансформации $ST[A,V_S]$ с начальным состоянием (v,T), и false – в противном случае, причем, в первом случае она также возвращает основную составляющую конечного состояния линеаризации,
- асинхронная функция A_{GS} возвращает текущую вершину графа автомата,
- все стимулы и реакции стабилизирующего автомата являются внутренними для DA:
  $(X_{St} \cup Y_{St}) \cap (X \cup Y \cup Z) = \varnothing$,
- всем стимулам, вызывающим асинхронные функции из стабилизирующего автомата, соответствуют единственные в рамках DA реакции, принадлежащие соответствующим асинхронным функциям,

- всем реакциям получения результата асинхронных функций из стабилизирующего автомата, соответствуют единственные в рамках DA стимулы, принадлежащие соответствующим асинхронным функциям.

Стационарный автоматный тестовый сценарий определяет правила организации стационарного тестирования на основе асинхронного автоматного тестового сценария со сценарными функциями. Основную роль здесь играет стабилизирующий автомат, который после завершения каждого цикла работы сценарной функции запрашивает модель поведения, описывающую все взаимодействия с целевой системой за время этого цикла, и проверяет корректность этой модели поведения. В случае успешной проверки стабилизирующий автомат возвращает управление управляющему автомату с указанием новой вершины в графе автомата. В противном случае тест завершается посылкой стимула abort.

Работа по оценке корректности модели поведения выполняется в асинхронной функции A_{HO}, которая получает на вход текущее состояние модели требований и асинхронную модель поведения. К модели поведения применяется стабилизирующая трансформация и затем проверяется корректность этой трансформации относительно стабилизирующей трансформации модели требований $ST[A,V_S]$ с начальным состоянием, равным текущему состоянию модели требований. В дополнение к оценке корректности A_{HO} возвращает основную составляющую конечного состояния линеаризации, которая становится новым текущим состоянием модели требований.

Однако, конечное состояние линеаризации может быть найдено только в том случае, когда все успешные линеаризации модели поведения ST[P] завершаются в одном и том же состоянии. Если это условие не выполняется, то асинхронная функция A_{HO} не может выполнить свою задачу и, соответственно, стационарный автоматный тестовый сценарий не может быть применен.

Далее, мы обратимся к вопросу применимости стационарного автоматного тестового сценария и определим достаточные условия для его корректной работы.

Асинхронная модель требований $A = (V, X, Y, Z, E)$ называется *стационарно-детерминированной* относительно множества стабильных состояний $V_S \subseteq V$, если для любого конечного мультимножества асинхронных взаимодействий P и для любого состояния $v \in V_S$

$$\left| \biguplus_{(p_1,\ldots,p_n)\ -\ линеаризация\ P} \Gamma^*(v,(p_1,\ldots,p_n)) \cap V_S \right| \le 1$$

Лемма.

Если асинхронная модель требований $A = (V, X, Y, Z, E)$ является стационарно-детерминированной относительно множества стабильных состояний $V_S \subseteq V$, то произвольный стационарный автоматный тестовый

сценарий относительно модели требований A с множеством стабильных состояний V_S и начальным состоянием $v_0 \in V_S$ применим независимо от поведения целевой системы.

Доказательство.

1. Сначала мы докажем по индукции, что асинхронная функция A_{HO} в качестве текущего состояния модели требований всегда получает стабильное состояние.

Функция A_{HO} получает на вход состояние модели требований, которое хранится в стабилизирующем автомате сценария. В начале работы сценария это состояние инициализируется начальным состоянием v_0, которое является стабильным по определению ($v_0 \in V_S$).

В дальнейшем, это состояние может измениться только в переходе $((s_4, v), result_{HO,true,v'}?, (s_5, v'))$, при получении результата от асинхронной функции A_{HO}, так как согласно четырнадцатому ограничению стационарного автоматного тестового сценария реакции получения результата асинхронной функции A_{HO} из стабилизирующего автомата $result_{HO,true,v'}$ соответствует единственный в рамках DA стимул, принадлежащий соответствующей асинхронной функции. При этом новое состояние совпадает с конечным состоянием линеаризации, которое возвращается функцией A_{HO}.

В качестве индуктивного перехода покажем, что для любой асинхронной модели поведения P, если стабилизирующая трансформация этой модели ST[P] удовлетворяет стабилизирующей трансформации $ST[A,V_S]$ с начальным состоянием (v,T), где v стабильно ($v \in V_S$), и конечное состояние линеаризации (v',x') существует, то оно также является стабильным ($v' \in V_S$).

Действительно, по определению, $\{ (v',x') \} = \Gamma^*((v,T), (p_1, p_2, \ldots, p_n))$ для некоторой линеаризации $(p_1, p_2, \ldots, p_n)$ модели поведения ST[P]. Так как псевдовзаимодействие (*done*,ε) является максимальным элементом в стабилизирующей трансформации ST[P], то $p_n = (done,\varepsilon)$. С другой стороны, по определению стабилизирующей трансформации $ST[A,V_S]$, все переходы, помеченные (*done*,ε), могут быть только вида ((w,F), (*done*,ε), (w,T)), где $w \in V_S$. Следовательно

$$\forall\ W \subseteq V \times \{T,F\} \quad \Gamma(W, p_n) \subseteq V_S \times \{T\},$$

и

$$\{(v',x')\} = \Gamma^*((v,T), (p_1, p_2, \ldots, p_n)) = \Gamma(\ldots \Gamma(\{(v,T)\}, p_1), \ldots p_n) \subseteq V_S \times \{T\}.$$

То есть $v' \in V_S$, что и требовалось доказать.

2. В качестве завершающего шага мы докажем, что для любой асинхронной модели поведения P, если стабилизирующая трансформация этой модели ST[P] удовлетворяет стабилизирующей трансформации $ST[A,V_S]$ с начальным состоянием (v,T), где v стабильно ($v \in V_S$), то конечное состояние линеаризации (v',x') существует. Из этого следует, что асинхронная функция A_{HO} всегда может выполнить свою задачу, так как согласно первому пункту она всегда получает на вход только стабильные состояния модели требований.

Предположим, что это не так. То есть существует асинхронная модель поведения P, стабильное состояние модели требований $v \in V_S$ и два состояния (v',x') и (v",x") такие, что

$\{ (v',x'), (v",x") \} \subseteq \bigcup_{(p_1,\ldots,p_n) - \text{линеаризация } ST[P]} \Gamma^*_{ST[A,V_S]}((v,T), (p_1,\ldots,p_n))$ и (v',x') ≠ (v",x").

Тогда существует две линеаризации модели поведения ST[P] ($p_{1,1}, p_{1,2}, \ldots, p_{1,n}$) и ($p_{2,1}, p_{2,2}, \ldots, p_{2,n}$) такие, что

$(v', x') \in \Gamma^*_{ST[A,V_S]}((v,T), (p_{1,1}, p_{1,2}, \ldots, p_{1,n}))$ и

$(v",x") \in \Gamma^*_{ST[A,V_S]}((v,T), (p_{2,1}, p_{2,2}, \ldots, p_{2,n}))$.

Повторяя рассуждения из первого пункта, можно показать, что x' = x" = T, $p_{1,n} = p_{2,n} = (done,\varepsilon)$, $v' \in V_S$, $v" \in V_S$ и

$(v', F) \in \Gamma^*_{ST[A,V_S]}((v,T), (p_{1,1}, p_{1,2}, \ldots, p_{1,n-1}))$ и

$(v",F) \in \Gamma^*_{ST[A,V_S]}((v,T), (p_{2,1}, p_{2,2}, \ldots, p_{2,n-1}))$.

То есть в асинхронной модели требований $ST[A,V_S]$ существует путь, начинающийся в (v,T), помеченный ($p_{1,1}, p_{1,2}, \ldots, p_{1,n-1}$) и заканчивающийся в (v', F):

$(v,T) \xrightarrow{p_{1,1}} (v_2,x_2) \xrightarrow{p_{1,2}} \ldots \xrightarrow{p_{1,n-1}} (v', F)$.

Так как $\{ p_{1,1}, p_{1,2}, \ldots, p_{1,n-1} \} \subseteq ((X \times Y) \times Z)$, то из определения $ST[A,V_S]$ следует, что в исходной модели требований A существует путь

$v \xrightarrow{p_{1,1}} v_2 \xrightarrow{p_{1,2}} \ldots \xrightarrow{p_{1,n-1}} v'$.

Следовательно, $v' \in \Gamma^*_A(v, (p_{1,1}, p_{1,2}, \ldots, p_{1,n-1}))$.

Аналогично, $v" \in \Gamma^*_A(v, (p_{2,1}, p_{2,2}, \ldots, p_{2,n-1}))$.

И так как мультимножества взаимодействий $\{ p_{1,1}, p_{1,2}, \ldots, p_{1,n-1} \}$ и $\{ p_{2,1}, p_{2,2}, \ldots, p_{2,n-1} \}$ совпадают, то существует мультимножество взаимодействий $P^* = \{ p_{1,1}, p_{1,2}, \ldots, p_{1,n-1} \}$:

$\{ v', v" \} \subseteq \bigcup_{(p_1,\ldots,p_{n-1}) - \text{линеаризация } P^*} \Gamma^*_A(v,(p_1,\ldots,p_{n-1}))$, причем v' ≠ v" и $v' \in V_S$, $v" \in V_S$.

То есть $\left| \bigcup_{(p_1,\ldots,p_{n-1}) - \text{линеаризация } P^*} \Gamma^*_A(v,(p_1,\ldots,p_{n-1})) \cap V_S \right| > 1$

и асинхронная модель требований A = (V, X, Y, Z, E) не является стационарно-детерминированной относительно множества стабильных состояний $V_S \subseteq V$.

Следовательно наше предположение не верно и v' = v", что и требовалось доказать.

■

Таким образом, стационарные автоматные тестовые сценарии могут применяться для тестирования относительно стационарно-детерминированных моделей требований, которые представляют широкий класс моделей требований. С другой стороны, требование стационарной-детерминированности не является необходимым, и стационарные автоматные тестовые сценарии могут работать с другими моделями при условии, что в процессе тестирования не встретится ситуация с отсутствием конечного состояния линеаризации по причине наличия нескольких успешных линеаризаций, заканчивающихся в различных состояниях.

Асинхронная функция A_{HO} может быть реализована в предположении, что модель требований является стационарно-детерминированной. В этом случае она может возвращать в качестве конечного состояния постсостояние первой успешной линеаризации. Также возможна реализация функция A_{HO}, которая будет верифицировать корректность модели требований, находя все успешные линеаризации и проверяя, что все их постсостояния совпадают.

Последовательной композицией набора асинхронных моделей поведения $\{(P_i,\pi_i)\}_{i=1}^k$ мы будем называть асинхронную модель поведения (P, π), в которой мультимножество P получено объединением элементов мультимножеств P_i ($P = \bigcup_{i=1}^k P_i$), а частичный порядок π составлен из частичных порядков π_i:

$$\bigcup_{i=1}^k \pi_i \cup \{ (p_1, p_2) \mid p_1 \in P_i \wedge p_2 \in P_j \wedge i < j \}.$$

Последовательная композиция моделей поведения представляет собой объединенную модель поведения, в которой присутствуют все взаимодействия из этих моделей поведения, причем считается, что любое взаимодействие из *i*-ой модели поведения происходило позже всех взаимодействий, принадлежащих предыдущим моделям.

Лемма.

Предположим, что модель требований A = (V, X, Y, Z, E) является стационарно-детерминированной относительно множества стабильных состояний $V_S \subseteq V$. Предположим также, что в процессе работы стационарного автоматного тестового сценария (SA, A_{St}, A_{IR}, A_{HO}, A_{GS}) относительно модели требований A с множеством стабильных состояний V_S и начальным состоянием $v_0 \in V_S$ асинхронная функция A_{IR} последовательно возвращала

стабилизирующему автомату асинхронные модели поведения $\{(P_i, \pi_i)\}_{i=1}^{k}$. Тогда, все вердикты функции A_{HO}, посланные стабилизирующему автомату, были положительными, в том, и только в том случае, когда последовательная композиция стабилизирующих трансформаций моделей поведения $\{ST[(P_i, \pi_i)]\}_{i=1}^{k}$ удовлетворяет стабилизирующей трансформации модели требований ST[A] с начальным состоянием v_0.

Доказательство.

1. Докажем утверждение леммы в прямую сторону. Предположим, что все вердикты функции A_{HO}, посланные стабилизирующему автомату, были положительными, и покажем, что последовательная композиция $\{ST[(P_i, \pi_i)]\}_{i=1}^{k}$ удовлетворяет ST[A] с начальным состоянием v_0.

В пункте 2 предыдущей леммы мы показали, что если асинхронная функция A_{HO} получает на вход асинхронную модель поведения P_i и состояние модели требований v и возвращает положительный вердикт и конечное состояние линеаризации v', то в модели требований ST[A] существует путь $(v,T) \xrightarrow{p_{i,1}} \dots \xrightarrow{p_{i,n(i)-1}} (v',F) \xrightarrow{p_{i,n(i)}} (v',T)$ для некоторой линеаризации $(p_{i,1}, p_{i,2}, \dots, p_{i,n(i)-1}, p_{i,n(i)})$ модели $ST[P_i]$ и $p_{i,n(i)} = done$.

Как мы уже отмечали, функция A_{HO} получает на вход состояние модели требований, которое хранится в стабилизирующем автомате сценария. Это состояние инициализируется в начале работы сценария начальным состоянием v_0, а в дальнейшем оно изменяется только в переходах вида $((s_4, v), result_{HO,true,v'}?, (s_5, v'))$. Эти переходы осуществляются при получении результата от асинхронной функции A_{HO}, так как согласно четырнадцатому ограничению стационарного автоматного тестового сценария каждой реакции получения результата асинхронной функции A_{HO} из стабилизирующего автомата $result_{HO,true,v'}$ соответствует единственный в рамках DA стимул, принадлежащий соответствующей асинхронной функции.

Отсюда следует, что в модели требований ST[A] существует путь

$$(v_0,T) \xrightarrow{p_{1,1}} \dots \xrightarrow{p_{1,n(1)-1}} (v'_1,F) \xrightarrow{p_{1,n(1)}}$$
$$(v'_1,T) \xrightarrow{p_{2,1}} \dots \xrightarrow{p_{2,n(2)-1}} (v'_2,F) \xrightarrow{p_{2,n(2)}} (v'_2,T) \dots$$
$$\dots (v'_{k-1},T) \xrightarrow{p_{k,1}} \dots \xrightarrow{p_{k,n(k)-1}} (v'_k,F) \xrightarrow{p_{k,n(k)}} (v'_k,T).$$

Покажем, что последовательность

$$(p_{1,1}, \dots, p_{1,n(1)-1}, p_{1,n(1)}, p_{2,1}, \dots, p_{2,n(2)-1}, p_{2,n(2)}, \dots, p_{k,1}, \dots, p_{k,n(k)-1}, p_{k,n(k)})$$

является линеаризацией (P, π), где (P, π) обозначается последовательная композиция $\{ST[(P_i, \pi_i)]\}_{i=1}^{k}$.

Действительно, все элементы мультимножества P присутствуют в этой последовательности, а порядок элементов последовательности не противоречит частичному порядку π.

Предположим, что последнее утверждение не верно. Тогда в последовательности существуют два таких элемента $p_{i,r}$ и $p_{j,s}$, что $p_{j,s}\ \pi\ p_{i,r}$, но $p_{i,r}$ расположен раньше, чем $p_{j,s}$, то есть $(i < j) \vee (i = j) \wedge (r < s)$.

В первом случае, если $i < j$, то $p_{i,r}\ \pi\ p_{j,s}$, согласно определению частичного порядка π, и следовательно $\neg\ p_{j,s}\ \pi\ p_{i,r}$.

Во втором случае, если $i = j$ и $r < s$, то из $p_{j,s}\ \pi\ p_{i,r}$ следует, что $p_{j,s}\ \pi_i\ p_{i,r}$. Но тогда $(p_{i,1}, p_{i,2}, \dots, p_{i,n(i)})$ не может быть линеаризацией модели $ST[P_i]$, так как порядок элементов $p_{j,s}$ и $p_{i,r}$ противоречит частичному порядку $ST[\pi_i]$. Следовательно, наше предположение не верно и порядок элементов последовательности не противоречит частичному порядку π.

Таким образом, рассматриваемая последовательность является линеаризацией последовательной композиции $\{ST[(P_i, \pi_i)]\}_{i=1}^{k}$, а из существования пути

$$(v_0,T) \xrightarrow{p_{1,1}} \dots \xrightarrow{p_{1,n(1)-1}} (v'_1,F) \xrightarrow{p_{1,n(1)}}$$
$$(v'_1,T) \xrightarrow{p_{2,1}} \dots \xrightarrow{p_{2,n(2)-1}} (v'_2,F) \xrightarrow{p_{2,n(2)}} (v'_2,T) \dots$$
$$\dots (v'_{k-1},T) \xrightarrow{p_{k,1}} \dots \xrightarrow{p_{k,n(k)-1}} (v'_k,F) \xrightarrow{p_{k,n(k)}} (v'_k,T).$$

в автомате A следует, что последовательная композиция $\{ST[(P_i, \pi_i)]\}_{i=1}^{k}$ удовлетворяет модели требований A с начальным состоянием v_0.

2. Докажем утверждение леммы в обратную сторону. Предположим, что первые k-1 вердикта функции A_{HO}, посланные стабилизирующему автомату, были положительными, а k-й вердикт – отрицательный. Покажем, что в этом случае последовательная композиция $\{ST[(P_i, \pi_i)]\}_{i=1}^{k}$ не удовлетворяет ST[A] с начальным состоянием v_0.

Предположим, что это не так и последовательная композиция $\{ST[(P_i, \pi_i)]\}_{i=1}^{k}$ удовлетворяет ST[A] с начальным состоянием v_0. То есть в модели требований ST[A] существует путь

$$(v_0,T) \xrightarrow{p_{1,1}} \dots \xrightarrow{p_{1,n(1)-1}} (v_{1,n(1)-1}, x_{1,n(1)-1}) \xrightarrow{p_{1,n(1)}} (v_{1,n(1)}, x_{1,n(1)}) \xrightarrow{p_{2,1}} \dots$$
$$\dots (v_{k-1,n(k-1)}, x_{k-1,n(k-1)}) \xrightarrow{p_{k,1}} \dots \xrightarrow{p_{k,n(k)-1}} (v_{k,n(k)-1}, x_{k,n(k)-1}) \xrightarrow{p_{k,n(k)}} (v_{k,n(k)}, x_{k,n(k)}),$$

где $(p_{1,1}, \dots, p_{1,n(1)-1}, p_{1,n(1)}, p_{2,1}, \dots, p_{2,n(2)-1}, p_{2,n(2)}, \dots, p_{k,1}, \dots, p_{k,n(k)-1}, p_{k,n(k)})$ – некоторая линеаризацией последовательной композиции $\{ST[(P_i, \pi_i)]\}_{i=1}^{k}$.

Согласно определению стабилизирующей трансформации максимальным элементом в SP[(P_i,π_i)] является псевдовзаимодействие (*done*,ε). Следовательно $\forall\ i \in \{1,\ldots,k\}\ \ p_{i,n(i)} = (done,\varepsilon)$.

По определению стабилизирующей трансформации модели требований ST[A] все переходы, помеченные (*done*,ε), имеют вид ((w,F), (*done*,ε), (w,T)), где $w \in V_S$. То есть

$$\forall\ i \in \{1,\ldots,k\}\ \ (v_{i,n(1)-1},\ x_{i,n(i)-1}) \xrightarrow{p_{i,n(i)}} (v_{i,n(i)},\ x_{i,n(i)}) \equiv (v'_i,F) \xrightarrow{(done\varepsilon)} (v'_i,T),$$

где $v'_i \in V_S$.

Докажем по индукции, что $\forall\ i \in \{1, \ldots, k-1\}$ в результате i-го вызова функции A_{HO} из стабилизирующего автомата, та возвращает состояние модели требований равное v'_i.

База индукции. Согласно определению стабилизирующего автомата функция A_{HO} в первый раз получает модель поведения (P_1,π_1) и состояние $v_0 \in V_S$. Если $k > 1$, то функция возвращает положительный вердикт и основную составляющую конечного состояния линеаризации (v', x'). Так как модель требований A является стационарно-детерминированной, то конечное состояние линеаризации существует, причем $v' = v'_1$, так как $(v_0,T) \xrightarrow{p_{1,1}} \ldots \xrightarrow{p_{1,n(1)-1}} (v'_1,F) \xrightarrow{(done\varepsilon)} (v'_1,T)$ является успешной линеаризацией модели (P_1,π_1).

Предположение индукции. Предположим, что после i-го вызова функции A_{HO} из стабилизирующего автомата, та возвращает состояние модели требований v'_i.

Шаг индукции. Докажем, что если после i+1-го вызова функция A_{HO} возвращает положительный вердикт и состояние модели требований w, то $w = v'_{i+1}$.

По определению стабилизирующего автомата при i+1-ом вызове функция A_{HO} получает модель поведения (P_{i+1},π_{i+1}) и состояние $v'_i \in V_S$. Если $i+1 < k$, то по условиям леммы функция возвращает положительный вердикт и основную составляющую конечного состояния линеаризации w. Заметим, что в автомате ST[A] существует путь $(v'_i,T) \xrightarrow{p_{i,1}} \ldots \xrightarrow{p_{i,n(i)-1}} (v'_{i+1},F) \xrightarrow{(done\varepsilon)} (v'_{i+1},T)$, который является одной из успешных линеаризаций модели (P_{i+1},π_{i+1}). Следовательно состояние w совпадает с состоянием v'_{i+1}.

Рассмотрим работу функции A_{HO} при k-ом вызове. Стабилизирующий автомат передает ей модель поведения (P_k,π_k) и состояние $w \in V$, полученное от функции A_{HO} на предыдущем вызове, если $k > 1$, или $v_0 \in V$, если $k = 1$. В первом случае, как мы показали по индукции, состояние модели требований w равно v'_{k-1}.

Исходя из предположения второго пункта леммы, существует линеаризация модели поведения SP[(P_k,π_k)] ($p_{k,1}$, …, $p_{k,n(k)-1}$, $p_{k,n(k)}$), такая что в автомате ST[A] существует путь $(v'_{k-1},T) \xrightarrow{p_{k,1}} \ldots \xrightarrow{p_{k,n(k)-1}} (v'_k,F) \xrightarrow{(done\varepsilon)} (v'_k,T)$. И так как модель требований A является стационарно-детерминированной, то согласно предыдущей лемме конечное состояние линеаризации существует и совпадает с (v'_k,T). Следовательно, функция A_{HO}, по определению, должна вернуть положительный вердикт и состояние v'_k. Но эта функция вернула на k-ом шаге отрицательный вердикт, то есть наше предположение не верно, и последовательная композиция $\{ST[(P_i,\pi_i)]\}_{i=1}^{k}$ не удовлетворяет модели требований ST[A] с начальным состоянием v_0.

■

Из данной леммы можно сделать вывод, что стационарный автоматный тестовый сценарий, проверяя корректность небольших моделей поведения, описывающих только часть поведения целевой системы в процессе тестирования, в итоге проверяет корректность поведения целевой системы в процессе всего тестирования в целом. Таким образом, декомпозиция задачи оценки корректности осуществлена корректным образом.

3.3.9. Асинхронный тестовый сценарий dfsm

Основным механизмом построения тестовых сценариев в синхронном случае является механизм построения тестового сценария dfsm. Этот механизм основан на неизбыточном алгоритме обхода на классе детерминированных сильно-связных конечных ориентированных графов α_{dfsm}[26]. Для стационарного тестирования асинхронных систем данный алгоритм обхода также будет выступать в качестве одного из основных.

Определение 19.

Асинхронным тестовым сценарием dfsm относительно асинхронной модели требований A = (V, X, Y, Z, E) с множеством стабильных состояний $V_S \subseteq V$ и начальным состоянием $v_0 \in V_S$, называется стационарный автоматный тестовый сценарий, в котором в качестве алгоритма движения по графу сценария используется алгоритм обхода α_{dfsm}.

Любое конечное функционирование алгоритма обхода α_{dfsm} приводит

- либо к обнаружению нарушения требований детерминированности или сильно-связности графа сценария,
- либо к построению обхода этого графа.

Таким образом, если граф сценария при любом корректном функционировании целевой системы является детерминированным и сильно-связным, то в результате успешной работы асинхронного тестового сценария dfsm путь, пройденный по графу сценария, является обходом этого графа. В этом случае, в каждом достижимом состоянии графа сценария будет вызвана каждая

допустимая в данном состоянии асинхронная сценарная функция. Это позволяет строить асинхронные тестовые сценарии на основе так называемой автоматной композиции, которая продемонстрировала свои достоинства в ходе применения технологии UniTesK для тестирования синхронных систем различного рода.

Однако, при тестировании асинхронных систем недетерминированное поведение встречается значительно чаще, так как в случае параллельных взаимодействий результат может определяться последовательностью внутренних событий целевой системы, не поддающихся наблюдению и тем более управлению со стороны тестовой системы. Поэтому потребность в использовании алгоритмов обхода, способных работать с недетерминированными графами, для асинхронных тестовых сценариев выше, чем в синхронном случае. Один из таких алгоритмов, мы рассмотрим в следующем разделе.

3.3.10. Алгоритм обхода ndfsm

Основная идея алгоритма обхода ndfsm заключается в следующем. Если в недетерминированном графе существует детерминированный сильносвязный полный остовный подграф, то алгоритм движения может использовать этот подграф для построения обхода, не сталкиваясь непосредственно с проблемами недетерминизма.

Ориентированный граф $G' = (VG', XG', EG')$ называется *остовным подграфом* ориентированного графа $G = (VG, XG, EG)$, если $VG' = VG$, $XG' = XG$ и $EG' \subseteq EG$.

Остовный подграф $G' = (VG', XG', EG')$ ориентированного графа $G = (VG, XG, EG)$ называется *полным остовным подграфом*, если

$$\forall (v, xg, v') \in EG\ (v, xg, v'') \in EG\ (v, xg, v') \in EG' \Rightarrow (v, xg, v'') \in EG'.$$

Напомним, что алгоритмом движения на ориентированных графах называется функция α, которая по ориентированному графу $G = (VG, XG, EG)$, текущей вершине $v \in VG$ и пройденному маршруту $(e_1, e_2, \ldots, e_n)$ определяет следующее действие $a \in XG \cup \{ \tau \}$.

Пусть $\mathbf{e} = (e_1, e_2, \ldots, e_n)$ – пройденный маршрут в графе $G = (VG, XG, EG)$. Тогда мы будем использовать следующие обозначения. Множеством пройденных вершин $VG_{\mathbf{e}}$ мы будем называть множество вершин пройденного маршрута:

$$VG_{\mathbf{e}} = \{ vg \in VG \mid \exists e_i: e_i = (v_i, xg_i, v'_i) \wedge (v_i = vg \vee v'_i = vg) \}$$

Пройденным подграфом мы будем называть граф $G'_{\mathbf{e}} = (VG_{\mathbf{e}}, XG, \{ e_1, e_2, \ldots, e_n \})$.

Пусть $e_i = (v_i, xg_i, v'_i)$. Тогда детерминированным пройденным подграфом мы будем называть граф $G''_{\mathbf{e}} = (VG_{\mathbf{e}}, XG, \{ e_i \mid \forall j \in \{ 1, \ldots, n \} (v_i \neq v_j \vee xg_i \neq xg_j \vee v'_i = v'_j) \})$.

Будем говорить, что стимул xg пройден в вершине vg, если существуют такая вершина $vg' \in VG$ и дуга $e_i \in \mathbf{e}$, что $e_i = (vg, xg, vg')$.

Если в вершине vg все допустимые стимулы являются пройденными, то такая вершина будет называться завершенной.

Для обеспечения детерминированности функции мы будем считать, что множество стимулов XG является фундированным, то есть множество линейно упорядоченно и в нем отсутствует бесконечно убывающая последовательность элементов.

Определение 20.

Алгоритмом движения ndfsm мы будем называть функцию α_{ndfsm}, которая по неизбыточному описанию ориентированного графа $G = (VG, XG, \pi)$, текущей вершине $v \in VG$ и пройденному маршруту $\mathbf{e} = (e_1, e_2, \ldots, e_n)$ вычисляет действие $a \in XG \cup \{ \tau \}$ следующим образом:

- Если текущая вершина не является завершенной, то функция возвращает минимальный стимул из $\pi(v)$, который не является пройденным в текущей вершине;
- Иначе, если в детерминированном пройденном подграфе существует маршрут, ведущий из текущей вершины в какую-либо из незавершенных пройденных вершин, то функция возвращает стимул, приписанный к дуге e_i пройденного маршрута, которая обладает минимальным индексом среди всех дуг, являющихся первыми дугами в кратчайших маршрутах детерминированного пройденного подграфа, ведущих из текущей вершины в какую-либо из незавершенных пройденных вершин;
- В противном случае функция возвращает τ.

Корректность определения следует из фундированности множества стимулов и конечности пройденного подграфа.

При работе с недетерминированными графами алгоритм движения не может гарантировать обход всех дуг графа, так как алгоритм не может выбрать конкретную дугу среди набора дуг, начинающихся в одной вершине и помеченных одним стимулом. Этот выбор происходит недетерминированно и независимо от работы алгоритма движения. Поэтому для недетерминированных графов вводится понятие обхода по стимулам, которое совпадает с понятием обхода на классе детерминированных графов и является более слабым на классе недетерминированных графов.

Обходом по стимулам [27] конечного ориентированного графа $G = (VG, XG, EG)$ называется маршрут $v_1 \xrightarrow{x_1} v_2 \xrightarrow{x_2} \ldots \xrightarrow{x_{n-1}} v_n$, в котором для каждой дуги $(v, x, v') \in EG$ существует $i \in \{ 1, \ldots, n-1 \}$, такое что $v = v_i$ и $x = x_i$.

Алгоритм движения α называется *алгоритмом обхода по стимулам* на классе конечных ориентированных графов Æ, если на любом графе G из класса Æ

любое функционирование алгоритма движения α является конечным обходом по стимулам графа G.

Лемма.

Алгоритм движения ndfsm является алгоритмом обхода по стимулам на классе конечных графов, обладающих детерминированным сильносвязным полным остовным подграфом.

Доказательство.

Предположим, что это не так. Тогда существует конечный граф $G = (VG, XG, EG)$, обладающий детерминированным сильносвязным полным остовным подграфом, и функционирование $\mathbf{e} = (e_1, e_2, \ldots, e_n, \ldots)$ алгоритма ndfsm на этом графе такие, что **e** не является конечным обходом по стимулам графа G. Такое может быть в двух случаях: если функционирование **e** является конечным, но не является обходом по стимулам графа G, и если функционирование **e** является бесконечным.

1. Предположим, что функционирование $\mathbf{e} = (e_1, e_2, \ldots, e_n)$ является конечным, но не является обходом по стимулам графа G. Тогда из определения функционирования следует, что $\alpha_{ndfsm}(A, v'_n, \mathbf{e}) = \tau$[9]. Следовательно вершина v'_n является завершенной и в детерминированном пройденном подграфе $G''_{\mathbf{e}}$ не существует маршрута, ведущего из вершины v'_n в какую-либо из незавершенных пройденных вершин.

С другой стороны **e** не является обходом по стимулам графа G, то есть в графе G существует такая дуга $(v, x, v') \in EG$, что не существует такого $i \in \{ 1, \ldots, n \}$, что $v = v_i$ и $x = x_i$.

Рассмотрим детерминированный сильносвязный полный остовный подграф $G' = (VG', XG', EG')$ графа G. Так как подграф является остовным, то обе вершины v и v'_n принадлежат VG'. Так как граф G конечный, а его подграф G' – сильносвязный, то существует конечный маршрут $\mathbf{f} = (f_1, f_2, \ldots, f_k)$, ведущий из v'_n в v. Пусть $f_i = (w_i, y_i, w'_i)$. Тогда для всех $i \in \{ 1, \ldots, k \}$ из вершины w_i в графе G выходит одна единственная дуга f_i, помеченная стимулом y_i. Предположим, что существует вторая такая дуга: $(w_i, y_i, w''_i) \in EG$. Тогда эта дуга также принадлежит и подграфу G' ($(w_i, y_i, w''_i) \in EG'$), так как подграф G' является полным остовным подграфом. А это противоречит тому, что подграф G' является детерминированным. Следовательно, наше предположение относительно существования второй дуги $(w_i, y_i, w''_i) \in EG$ неверно и дуга (w_i, y_i, w'_i) является единственной дугой в графе G, выходящей из w_i и помеченной стимулом y_i.

Докажем индукцией по пути $\mathbf{f} = (f_1, f_2, \ldots, f_k)$, что все вершины w'_i являются завершенными для $i \in \{ 1, \ldots, k \}$.

<u>База индукции.</u> Вершина $w_1 = v'_n \in VG_{\mathbf{e}}$ является завершенной, по определению функционирования и функции α_{ndfsm}.

[9] В рамках данного доказательства мы считаем, что $e_i = (v_i, x_i, v'_i)$.

<u>Предположение индукции.</u> Пусть вершина $w'_{i-1} = w_i$ является завершенной.

<u>Шаг индукции.</u> Так как вершина w_i является завершенной, стимул y_i является допустимым в этой вершине и дуга (w_i, y_i, w'_i) является единственной дугой, выходящей из w_i и помеченной стимулом y_i, то дуга (w_i, y_i, w'_i) также является пройденной, то есть $(w_i, y_i, w'_i) \in \mathbf{e}$. Отсюда следует, что дуга (w_i, y_i, w'_i) принадлежит детерминированному пройденному подграфу $G''_{\mathbf{e}}$, так как эта дуга является пройденной и детерминированной. На этом основании можно сделать вывод, что вершина w'_i является завершенной, так как в противном случае мы нашли маршрут $(f_1, f_2, \ldots, f_i)$ в детерминированном пройденном подграфе $G''_{\mathbf{e}}$, ведущий из v'_n в одну из незавершенных пройденных вершин.

Мы доказали, что все вершины w'_i являются завершенными. Следовательно и вершина $w'_k = v \in VG_{\mathbf{e}}$ также является завершенной, что противоречит нашему предположению о том, что **e** не является обходом по стимулам графа G. Следовательно это предположение не верно и функционирование **e** не является конечным.

2. Предположим, что функционирование $\mathbf{e} = (e_1, e_2, \ldots, e_n, \ldots)$ является бесконечным. Тогда для всех $i \geq 0$ $\alpha_{ndfsm}(A, v'_i, (e_1, \ldots, e_i)) \neq \tau$.

Определим функцию β_G на графе G, устанавливающую соответствие между маршрутом $(e_1, \ldots, e_i)$ в графе G и парой натуральных чисел (b_1,b_2), следующим образом:

- b_1 равно числу дуг графа G, отсутствующих в маршруте $(e_1, \ldots, e_i)$, $|EG \setminus \{ e_1, \ldots, e_i \}|$;
- b_2 равно
  - 0, если $i = 0$, вершина v'_i является незавершенной или в детерминированном пройденном подграфе нет маршрута, ведущего из вершины v'_i и в какую-либо из незавершенных пройденных вершин;
  - минимальной длине маршрута в детерминированном пройденном подграфе, начинающегося в вершине v'_i и заканчивающегося в какой-либо из незавершенных пройденных вершин, в противном случае.

Определим на множестве пар натуральных чисел линейный порядок:

$$(b_1,b_2) < (b'_1,b'_2) \Leftrightarrow b_1 < b'_1 \vee (b_1 = b'_1 \wedge b_2 < b'_2)$$

Отметим, что множество пар натуральных чисел с данным линейным порядком образуют фундированное множество, то есть множество, в котором отсутствует бесконечно убывающая последовательность элементов.

Согласно определению $\beta_G(()) = (|EG|,0)$.

Покажем, что для любого префикса $(e_1, \ldots, e_i)$ функционирования **e**

$$\beta_G((e_1, \ldots, e_i, e_{i+1})) < \beta_G((e_1, \ldots, e_i))$$

Для всех $i \geq 0$ $\alpha_{ndfsm}(A, v'_i, (e_1, \ldots, e_i)) \neq \tau$. Следовательно, либо вершина v'_i не является завершенной, либо в детерминированном пройденном подграфе

существует маршрут, ведущий из вершины v'_i в какую-либо из незавершенных пройденных вершин.

Рассмотрим первый случай. Если вершина v'_i не является завершенной, то α_{ndfsm} возвращает минимальный стимул из $\pi(v)$, который не является пройденным в вершине v'_i. На этом основании можно сделать вывод, что e_{i+1} является дугой, не принадлежащей $\{ e_1, \ldots, e_i \}$. Поэтому первый компонент $\beta_G((e_1, \ldots, e_{i+1}))$ на единицу меньше первого компонента $\beta_G((e_1, \ldots, e_i))$, откуда следует, что $\beta_G((e_1, \ldots, e_i, e_{i+1})) < \beta_G((e_1, \ldots, e_i))$.

Рассмотрим второй случай. Если вершина v'_i является завершенной и в детерминированном пройденном подграфе существует маршрут, ведущий из вершины v'_i в какую-либо из незавершенных пройденных вершин, то α_{ndfsm} возвращает стимул, приписанный к дуге e_j пройденного маршрута, которая обладает минимальным индексом среди всех дуг, являющихся первыми дугами в кратчайших маршрутах детерминированного пройденного подграфа, ведущих из текущей вершины в какую-либо из незавершенных пройденных вершин. При этом дуга e_{i+1} может либо совпасть с дугой e_j, либо нет.

Если дуга e_{i+1} совпадает с дугой e_j, то уменьшается на единицу второй компонент функции β_G, так как кратчайший маршрут в детерминированном пройденном подграфе станет на одну дугу короче, или же в результате перехода будет достигнута незавершенная вершина. При этом первая компонента функции β_G останется неизменной, поэтому $\beta_G((e_1, \ldots, e_i, e_{i+1}))$ будет меньше $\beta_G((e_1, \ldots, e_i))$.

Если дуга e_{i+1} не совпадает с дугой e_j, то эта дуга не принадлежит $\{ e_1, \ldots, e_i \}$, так как в противном случае дуга e_j не была бы частью детерминированного пройденного подграфа. Следовательно первый компонент $\beta_G((e_1, \ldots, e_{i+1}))$ на единицу меньше первого компонента $\beta_G((e_1, \ldots, e_i))$, то есть $\beta_G((e_1, \ldots, e_i, e_{i+1})) < \beta_G((e_1, \ldots, e_i))$.

Мы показали, что для всех $i \geq 0$ $\beta_G((e_1, \ldots, e_i, e_{i+1})) < \beta_G((e_1, \ldots, e_i))$. Следовательно функционирование **e** не может быть бесконечным, так как в противном случае последовательность $\beta_G((e_1, \ldots, e_i))$ образует бесконечно убывающую подпоследовательность в фундированном множестве.

■

Алгоритм движения ndfsm позволяет заменять стандартный алгоритм dfsm, в тех случаях когда отдельные сценарные функции имеют недетерминированную природу, а другие гарантировано обеспечивают детерминированное движение по графу.

Определение 21.

Асинхронным тестовым сценарием ndfsm относительно асинхронной модели требований $A = (V, X, Y, Z, E)$ с множеством стабильных состояний $V_S \subseteq V$ и начальным состоянием $v_0 \in V_S$, называется стационарный автоматный

тестовый сценарий, в котором в качестве алгоритма движения по графу сценария используется алгоритм обхода α_{ndfsm}.

3.3.11. Параллельные воздействия на целевую систему

Асинхронные тестовые сценарии, рассмотренные ранее, не предоставляют разработчику тестов никакой автоматизации в осуществлении параллельных тестовых воздействий на целевую систему. Чтобы выполнять параллельные тестовые воздействия в стационарных автоматных тестовых сценариях необходимо вручную описывать правила их осуществления.

В данном разделе мы рассмотрим возможные варианты осуществления параллельных тестовых воздействий в автоматизированном режиме. В качестве таких вариантов были выделены следующие возможности:

- параллельная работа независимых тестовых сценариев;
- параллельное выполнение сценарных функций в рамках работы одного тестового сценария;
- организация параллельных воздействий на уровне сценарных функций.

Наиболее высокий уровень на котором можно организовать параллелизм – это уровень тестовых сценариев. Несколько тестовых сценариев можно запустить независимо друг от друга при условии, что каждый из них будет работать со своей собственной копией модельного состояния. Если отсутствие учета влияния параллельно работающих тестовых сценариев не может привести к ошибкам в оценке корректности поведения целевой системы, то тогда такой независимый запуск сценариев является корректным и может быть использован для того, чтобы удостовериться, что такие параллельные активности в целевой системе не приводят к аномальному поведения.

Следующий уровень параллелизма может быть достигнут при параллельном запуске нескольких сценарных функций в рамках одного тестового сценария. Одним из вариантов в рамках данного подхода является алгоритм повторного обхода графа сценария, предложенный в [28]. Идея этого алгоритма заключается в разбиении работы тестового сценария на две фазы. Первая фаза представляет собой обычную работу асинхронного тестового сценария dfsm, по результатам которой осуществляется построение графа тестового сценария. Во время второй фазы выполняется повторный обход графа сценария, во время которого в каждой вершине графа происходит параллельное выполнение всех пар сценарных функций с учетом дополнительных условий их допустимости.

Другим вариантом организации параллельной работы сценарных функций может быть комбинация нескольких тестовых сценариев, работающих с общим модельным состоянием. В этом случае работа теста также должна выполняться в двух режимах. В одном режиме работает только одна сценарная функция одного из сценариев, а в другом происходит параллельный запуск нескольких сценарных функций.

Общий недостаток рассмотренных выше подходов заключается в слишком большом удалении уровня параллелизма от целевой системы. Параллелизм на

уровне тестовых сценариев и сценарных функций не может гарантировать параллельного выполнения функций целевой системы и, таким образом, не позволяет обеспечить даже минимального уровня тестирования одновременной работы компонентов целевой системы.

Возможность одновременной работы компонентов целевой системы и способы внешнего управления такой работой являются индивидуальными для каждой конкретной целевой системы. Поэтому для обеспечения эффективных параллельных воздействий на целевую систему предлагается использовать специальный управляющий механизм, зависящий от особенностей конкретной тестируемой системой.

Обращения к такому управляющему механизму можно осуществлять непосредственно из асинхронных сценарных функций. В результате разработчик тестового сценария получает возможность управлять не только набором модельных воздействий, предназначенным для одновременного выполнения, но и перебирать различные параметры такого выполнения.

Рассмотрим пример системы с асинхронным интерфейсом, в которой асинхронность проявляется как возможность блокировки вызовов отдельных интерфейсных функций вплоть до наступления некоторого события, например, вызова другой интерфейсной функции системы в параллельном потоке управления. Такая целевая система предполагает работу с общими данными из различных потоков управления. Поэтому проверка корректности синхронизации доступа к общим данным является необходимой составляющей качественного тестирования. Для осуществления такой проверки требуется организовать параллельное выполнение кода, осуществляющего доступ к общим данным, в нескольких потоках управления и обеспечить переключение этих потоков в процессе выполнения тестируемого кода.

В данном примере, управляющий механизм может быть построен на основе имеющейся информации о работе планировщика потоков операционной системы. В качестве указаний со стороны сценарных функций механизм может получать набор целевых функций, которые необходимо выполнять параллельно, а также итерируемые значения параметров, управляющих переключениями потоков.

Таким образом, предлагается отказаться от разработки универсального способа организации параллельных воздействий, так как нет оснований полагать, что такой метод может обеспечить требуемый уровень качества тестирования. Вместо этого предлагается разрабатывать специализированные механизмы организации параллельных воздействий, общие для определенных классов целевых систем, и обращаться к этим механизмам непосредственно из сценарных функций.

Такое решение оказывает минимальное влияние на архитектуру тестового набора. Оно влечет появление опционального компонента, ответственного за организацию параллельных воздействий. Через этот компонент некоторые асинхронные сценарные функции могут управлять генерацией тестовых

воздействия, в то время как другие сценарные функции могут осуществлять тестовые воздействия самостоятельно.

3.3.12. Тестирование с открытым стационарным состоянием

Тестирование асинхронных систем всегда происходит в режиме со скрытым состоянием. Причина этого заключается в том, что в процессе осуществления тестовых воздействий у тестовой системы нет достоверной информации о текущем состоянии целевой системы. Даже если тестовая система имеет возможность прочитать внутреннее состояние целевой системы, то так как взаимодействия происходят асинхронно, то прочитанное состояние может быть неконсистентно или быть измененным в результате взаимодействия, о котором тестовой системе еще не известно.

Поэтому если существует возможность чтения консистентного состояния целевой системы, то такое чтение должно оформляться как отдельная интерфейсная операция. Спецификация этой операции должна проверять, что прочитанное состояние соответствует текущему модельному состоянию. В этом случае, ситуация, когда взаимодействия, связанные с чтением состояния, будут подвергаться процессу линеаризации как все остальные взаимодействия и ни одно из возможных упорядочиваний не приведет к положительному вердикту, будет означать, что прочитанное состояние не согласуется с асинхронной моделью требований.

В отличие от тестирования с открытым состоянием, где чтение состояния целевой системы осуществляется автоматически медиатором, чтение состояния через интерфейсную операцию должно каждый раз инициироваться вручную из сценарных функций. В то же время существуют такие ситуации, когда это чтение полезно автоматизировать.

Например, полезно воспользоваться возможностью получить информацию о внутреннем состоянии целевой системы в конце каждой активной фазы работы стационарного автоматного тестового сценария. В момент после завершения времени ожидания, когда все взаимодействия уже зарегистрированы и целевая система находится в стационарном состоянии, можно прочитать состояние целевой системы, не опасаясь за его консистентность. А полученные сведения могут оказаться полезными как гипероракулу для выбора правильной линеаризации зарегистрированных взаимодействий, так и разработчику теста для упрощения локализации ошибочного поведения.

Тестирование систем с асинхронным интерфейсом, при котором тестовая система получает достоверную информацию о внутреннем состоянии целевой системы, во время нахождения той в стационарном состоянии, называется *тестированием с открытым стационарным состоянием*.

Для тестирования с открытым стационарным состоянием стационарный автоматный тестовый сценарий может предоставлять сервис автоматического инициирования чтения состояния целевой системы каждый раз, когда целевая система попадает в стационарное состояние. При этом тестовая система только инициирует чтение, а то как это чтение реализуется определяется

разработчиком теста. Единственное, что требует тестовая система – это то, что чтение должно осуществляться посредством вызова псевдо интерфейсной операции как это было рассмотрено ранее.

3.3.13. Нарушение предусловий асинхронных воздействий

Для описания модели требований как синхронных, так и асинхронных систем используются спецификации интерфейсных операций. Каждая спецификация операции состоит из предусловия и постусловия. Предусловие описывает какие значения входных параметров являются допустимыми в текущем модельном состоянии, а постусловие определяет какие значения выходных параметров и постсостояния модели являются корректными для данных значений входных параметров и пресостояния модели. Другими словами, предусловие описывает обязательства пользователя целевой системы перед этой системой, а постусловие – обязательства целевой системы перед своим пользователем.

В процессе тестирования синхронных систем, перед вызовом каждой интерфейсной операции, тестовая система проверяет предусловие этой операции на соответствующих значениях входных параметров. Таким образом предотвращается возможность передачи целевой системе некорректных данных, которые могут привести к невозможности продолжать тестирование. Например, если в требованиях к целевой системе сказано, что ее поведение на некоторых входных данных не определено, а тестовая система вызвала интерфейсную операцию с такими значениями, то любое дальнейшее поведение целевой системы не будет противоречить требованиям к ней, и поэтому продолжать тестирование становится бессмысленно.

При тестировании асинхронных систем проверка предусловий интерфейсных операций существенно осложняется тем фактом, что в момент асинхронного взаимодействия тестовой системе может быть неизвестно текущее модельное состояние. Асинхронный тестовый сценарий может осуществлять несколько тестовых воздействий одновременно, получая в то же время отложенные реакции, и поэтому не владеть достоверной информацией о состоянии тестируемой системы в момент взаимодействия.

С другой стороны, в процессе оценки корректности поведения целевой системы становится доступной информация о возможном состоянии модели требований перед вызовом каждой операции. Причем таких состояний может быть несколько в зависимости от рассматриваемой линеаризации модели поведения. И если хотя бы для одной линеаризации нарушается предусловие какой-либо интерфейсной операции-стимула, то это означает, что возможен такой сценарий, при котором тестовая система не выполнила своих обязательств перед целевой системой и дальнейшее поведение последней не определено. В этом случае, тестовая система считает, что асинхронный тестовый сценарий построен некорректным образом, так как допустил такую ситуацию, и завершает свою работу.

Нарушение предусловия интерфейсной операции-реакции существенно отличается от нарушения предусловия операции-стимула. Так как предусловие операции описывает обязательства стороны, инициирующей соответствующее

ей взаимодействие, то предусловие интерфейсной операции-реакции описывает обязательства целевой системы. В данном случае нарушение предусловия ничем не отличается от нарушения постусловия, что стирает границу между ними.

Апостериорная проверка предусловий при тестировании асинхронных систем имеет несколько недостатков. Во-первых, если тестовая система не осуществляет полный перебор всех возможных линеаризаций, то потенциальные нарушения предусловий интерфейсных операций-стимулов могут быть не найдены. А во-вторых, отсутствие какой-либо априорной проверки не позволяет предотвращать некорректные воздействия со стороны тестовой системы.

Чтобы смягчить второй недостаток можно использовать следующий подход. В предусловии интерфейсных операций-стимулов выделяется статическая составляющая, которая определяет ограничения на допустимые значения входных параметров, независящие от состояния. Эти ограничения можно проверять перед каждым вызовом интерфейсной операции и использовать для предотвращения некорректных воздействий на тестируемую систему. В то же время, все предусловие целиком будет проверяться только на этапе оценки корректности поведения целевой системы.

Если предусловие интерфейсной операции задано предикатом pre_I на множестве $V_I \times X_I$, то *статической составляющей* предусловия мы будем называть предикат $static(pre_I)$ на множестве X_I, определяемый следующим образом:

$$static(pre_I)(x) = T \Leftrightarrow \forall\, v \in V_I \;\; pre_I(v,x) = T$$

Ответственность за предварительную проверку статической составляющей предусловия необходимо возложить на медиатор воздействия, так как именно через этот компонент тестовой системы проходят все воздействия на целевую систему.

3.3.14. Тестовый сценарий в унифицированной архитектуре асинхронного теста

В данном разделе мы рассмотрим место асинхронного тестового сценария в унифицированной архитектуре асинхронного теста, а также основные принципы его взаимодействия с другими компонентами теста.

Работа асинхронного тестового сценария разбивается на два этапа. На первом этапе тестовый сценарий осуществляет набор тестовых воздействий на целевую систему посредством обращения к медиаторам воздействий. Обратно от медиаторов воздействий сценарий получает непосредственные реакции целевой системы. При этом все воздействия на целевую систему, а также полученные от нее отложенные реакции регистрируются в регистраторе взаимодействий. Для формирования тестовых воздействий тестовый сценарий может обращаться к модельному состоянию, но не может изменять его.

На втором этапе асинхронный тестовый сценарий обращается к гипероракулу, чтобы оценить корректность поведения целевой системы и синхронизировать

модельное состояние с состоянием целевой системы. После этого тестовый сценарий может осуществить следующий набор тестовых воздействий или принять решение о завершении тестирования. Схема взаимодействия асинхронного тестового сценария с другими компонентами тестовой системы представлена на Рис. 15.

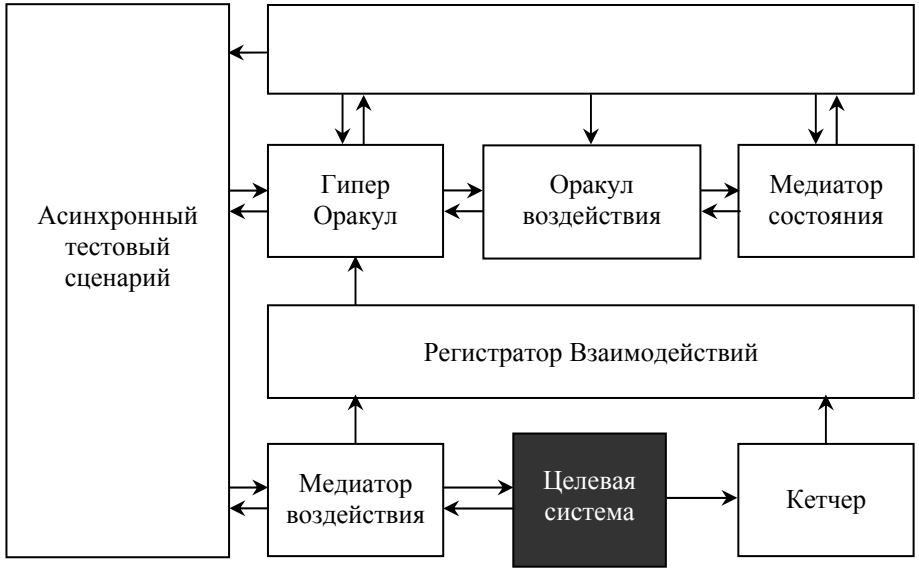

Рис. 15. Место асинхронного тестового сценария в унифицированной архитектуре асинхронного теста

Стационарный автоматный тестовый сценарий чередует первый и второй этапы неоднократно. Для формирования тестовых воздействий на первом этапе он использует локальные асинхронные тестовые сценарии, которые описываются как набор параметризуемых асинхронных сценарных функций. Комбинация локальных сценариев осуществляется на основании алгоритма движения по графу сценария, определяемого, как и в синхронном случае, неявным образом. К каждой дуге графа сценария приписывается локальный асинхронный тестовый сценарий, заданный в виде асинхронной сценарной функции с фиксированными значениями параметров. Переход по дуге графа соответствует одному циклу работы асинхронного тестового сценария, на котором последовательно выполняются первый и второй этапы.

3.4. Оценка качества тестирования

В подавляющем большинстве случаев проверить целевую систему на всех допустимых тестовых данных невозможно. Поэтому приходится ограничиваться некоторым конечным набором тестов. В такой ситуации

неизбежно встает вопрос о качестве тестирования, проведенного посредством данного тестового набора.

Для систем с асинхронным интерфейсом ситуация осложняется тем, что, как правило, такие системы используют для своей работы более одного потока управления. Поэтому для более полной оценки качества тестирования требуется дополнительно учитывать различные аспекты работы многопотоковых программных систем. В качестве таких аспектов могут выступать моменты переключения потоков управления, если работа целевой системы осуществляется на одном процессоре, или моменты межпроцессорной синхронизации, если процессоров несколько. Однако все эти дополнительные аспекты сильно зависят от особенностей устройства конкретных целевых систем, и поэтому рассмотрение способов их оценки находится за рамками данной работы.

Измерение покрытия исходного кода, также как и в синхронном случае, является очень важным и полезным способом оценки качества тестирования систем с асинхронным интерфейсом. Но оценка качества тестирования остается неполной, если она не учитывает покрытия требований к тестируемой функциональности. Для получения такой оценки мы будем использовать асинхронную модель требований и определим метрики покрытия в терминах этой модели.

3.4.1. Метрики покрытия асинхронной модели требований

Метрикой покрытия асинхронной модели требований $A = (V, X, Y, Z, E)$ называется конечное множество подмножеств переходов модели требований $M \subseteq 2^E$. *Элементами покрытия* называются элементы метрики M, являющиеся подмножествами E.

Частично-упорядоченное множество (P', π') называется *префиксным подмножеством* частично-упорядоченного множества (P, π), если:

- множество P' является подмножеством P ($P' \subseteq P$);
- частичный порядок π' является подмножеством частичного порядка π' ($\pi' \subseteq \pi$);
- все пары элементов из P', являющиеся упорядоченными в (P, π), также являются упорядоченными в (P', π'): $(p_1,p_2) \in \pi \wedge p_1 \in P' \wedge p_2 \in P' \Rightarrow (p_1,p_2) \in \pi'$;
- все элементы P меньшие элемента P' также принадлежат P': $(p_1,p_2) \in \pi \wedge p_2 \in P' \Rightarrow p_1 \in P'$.

Предположим, что асинхронный тест (P, π) является неуспешным относительно асинхронной модели требований $A = (V, X, Y, Z, E)$ с начальным состоянием $v_0 \in V$. Будем говорить, что частично-упорядоченное множество (P', π') является *граничным успешным подтестом* асинхронного теста (P, π), если:

- (P', π') является префиксным подмножеством (P, π);

- (P', π') является успешным относительно асинхронной модели требований A = (V, X, Y, Z, E) с начальным состоянием $v_0 \in V$;
- существует такое частично-упорядоченное множество (P'', π''), что
  - (P'', π'') является префиксным подмножеством (P, π);
  - (P'', π'') является неуспешным относительно асинхронной модели требований A = (V, X, Y, Z, E) с начальным состоянием $v_0 \in V$;
  - P'' включает в себя P', но P'' больше P' ровно на один элемент ($P' \subset P'' \ \wedge \ | P'' \setminus P' | = 1$).

Заметим, что всякое максимальное успешное префиксное подмножество асинхронного теста (P, π) является его граничным успешным подтестом, а обратное утверждение не верно.

Если асинхронный тест (P, π) является успешным относительно асинхронной модели требований A = (V, X, Y, Z, E) с начальным состоянием $v_0 \in V$, то путь в автомате A, удовлетворяющий требованиям определения 9 будем называть *успешным*.

Если асинхронный тест (P, π) является успешным относительно асинхронной модели требований A = (V, X, Y, Z, E) с начальным состоянием $v_0 \in V$, то будем говорить, что тест (P, π) *покрыл* элемент покрытия $C_j \in M$, если любой успешный путь (e_1, e_2, …, e_n, …) теста (P, π) в автомате A содержит хотя бы один переход e_i, входящий во множество C_j.

Если асинхронный тест (P, π) является неуспешным относительно асинхронной модели требований A = (V, X, Y, Z, E) с начальным состоянием $v_0 \in V$, то будем говорить, что тест (P, π) *покрыл* элемент покрытия $C_j \in M$, если любой успешный путь (e_1, e_2, …, e_n, …) любого граничного успешного подтеста теста (P, π) в автомате A содержит хотя бы один переход e_i, входящий во множество C_j.

Покрытием метрики M асинхронным тестом (P, π) будем называть набор элементов метрики M, покрытых данным тестом.

Таким образом, асинхронная метрика покрытия, так же как и в синхронном случае, представляет собой конечный набор подмножеств переходов модели требований. И оценка качества тестирования осуществляется в терминах покрытия элементов из этого множества. Основное отличие асинхронного случая заключается в том, что определение того, является ли данный элемент метрики покрытым данным тестом, значительно сложнее ввиду необходимости рассмотрения различных путей в модели требований.

Асинхронная метрика покрытия M называется *управляемой*, если все переходы из ее элементов, помечены стимулом и непосредственной реакцией, и для любых двух переходов $I_1 = (v_1, x_1, y_1, v'_1)$ и $I_2 = (v_2, x_2, y_2, v'_2)$ выполнено следующее утверждение:

$$(v_1 = v_2) \wedge (x_1 = x_2) \ \Rightarrow \ \forall C_i \in M \ \ (I_1 \in C_i) \Leftrightarrow (I_2 \in C_i)$$

Управляемые асинхронные метрики характеризуются тем, что принадлежность перехода к тому или иному элементу покрытия зависит только от той части перехода, которая управляется тестовой системой, то есть от пресостояния и стимула. Тем не менее, работая с управляемыми асинхронными метриками, тестовая система имеет меньше возможностей, чем при синхронном тестировании, так как только находясь в стационарном состоянии тестовая система может определить принадлежность текущего стимула к тому или иному элементу покрытия.

3.4.2. Описание асинхронных метрик покрытия

Асинхронные метрики покрытия описываются совместно с описанием предусловий и постусловий интерфейсных операций целевой системы. Это позволяет локализовать описание требований и выделение элементов покрытия на основе этих требований. Элементы покрытия, как правило, соответствуют ветвям функциональности в поведении интерфейсной операции или ситуациям, интересным с точки зрения тестирования, таким как граничные условия.

Предположим, что для целевой системы были выделены k видов взаимодействий, инициируемых окружением, $S = \{ S_i \mid i = 1, \dots , k \}$ и m видов взаимодействий, инициируемых целевой системой, $R = \{ R_j \mid j = 1, \dots , m \}$. A асинхронная модель требований задана посредством асинхронной спецификации $Spec = \{ Spec_{Si} \mid i = 1, \dots, k; \ k > 0 \} \cup \{ Spec_{Rj} \mid j = 1, \dots, m; \ m \geq 0 \}$.

Тогда метрика покрытия интерфейсной операции $\mu : V_{Si} \times X_{Si} \times Y_{Si} \times V_{Si} \rightarrow R$, являющаяся сюръективной функцией в конечное множество R, используется для описания асинхронной метрики интерфейсной операции-стимула S_i. Этой метрике покрытия соответствует асинхронная метрика покрытия асинхронной модели требований **MRA**[Spec], которую мы будем обозначать **MA**[μ].

$$\mathbf{MA}[\mu] = \{C_r \in 2^E \mid r \in R\},$$
$$\text{где } C_r = \{(v, x, y, v') \in V \times X \times Y \times V \mid \mu(v, x, y, v') = r\}.$$

Аналогично определяются метрики покрытия интерфейсных операций-реакций. Метрика покрытия интерфейсной операции $\mu : V_{Rj} \times Unit \times Y_{Rj} \times V_{Rj} \rightarrow R$ используется для описания асинхронной метрики интерфейсной операции-реакции R_j. Этой метрике μ соответствует асинхронная метрика покрытия **MA**[μ] асинхронной модели требований **MRA**[Spec]:

$$\mathbf{MA}[\mu] = \{ C_r \in 2^E \mid r \in R \},$$
$$\text{где } C_r = \{ (v, z, v') \in V \times Z \times V \mid \mu(v, \varepsilon, z, v') = r\}.$$

Также как и в синхронном случае, определение метрик интерфейсных операций не дает возможности задать произвольную асинхронную метрику покрытия. Этот способ позволяет описывать только метрики покрытия, переходы которых помечены одной интерфейсной операцией. Но как показывает практический опыт этого оказывается достаточно для измерения качества тестирования систем с асинхронным интерфейсов различных типов.

Второй способ описания метрик покрытия при помощи расширенных метрик покрытия интерфейсных операций также может быть использован при тестировании асинхронных систем. Асинхронная метрика покрытия, заданная посредством расширенной метрики покрытия интерфейсной операции S_i $\mu' = \{ c_p \mid p = 1, \ldots, n; c_p : V_{Si} \times X_{Si} \times Y_{Si} \times V_{Si} \to \text{Bool} \}$, обозначается $\mathbf{MA}[\mu']$:

$$\mathbf{M}[\mu'] = \{ C_p \in 2^E \mid p = 1, \ldots, n \},$$
$$\text{где } C_p = \{ (v, x, y, v') \in V \times X \times Y \times V \mid c_p(v, x, y, v') \}.$$

Также как и асинхронная метрика покрытия, заданная посредством расширенной метрики покрытия интерфейсной операции R_j $\mu' = \{ c_p \mid p = 1, \ldots, n; c_p : V_{Rj} \times \text{Unit} \times Y_{Rj} \times V_{Rj} \to \text{Bool} \}$:

$$\mathbf{M}[\mu'] = \{ C_p \in 2^E \mid p = 1, \ldots, n \},$$
$$\text{где } C_p = \{ (v, z, v') \in V \times Z \times V \mid c_p(v, \varepsilon, z, v') \}.$$

3.4.3. Оценка качества тестирования в унифицированной архитектуре асинхронного теста

Оценка качества тестирования асинхронных систем проводится по той же схеме, что и при тестировании синхронных систем. В процессе выполнения теста вычисляется принадлежность каждого отдельного перехода в асинхронной модели требований ко всем заданным разработчиком теста асинхронным метрикам покрытия. Эта информация сохраняется в трассе теста. И уже после завершения тестирования выполняется анализ трассы и генерация отчетов о покрытии асинхронных метрик покрытия.

В унифицированной архитектуре асинхронного теста за рассмотрение каждого отдельного перехода в асинхронной модели требований отвечает оракул воздействия. Поэтому именно на него накладывается дополнительная ответственность за вычисление принадлежности этого перехода к элементам метрик покрытия и помещение этой информации в трассу теста.

Если асинхронная метрика покрытия задана функцией μ, то оракул воздействия вычисляет значение этой функции и помещает его в трассу, так как оно однозначно характеризует покрытый элемент метрики. Если асинхронная метрика покрытия задана расширенной метрики покрытия, то оракул воздействия вычисляет значения всех предикатов c_i и индексы предикатов, принявших положительное значение, помещаются в трассу в качестве описания покрытых элементов.

Но при тестировании систем с асинхронным интерфейсом информации о принадлежности отдельных переходов к элементам метрики недостаточно для определения покрытия этой метрики. Чтобы вычислить покрытие необходимо учитывать все успешные пути в модели требований, а также принадлежность переходов этим путям. Эта информация помещается в трассу теста гипероракулом, так как именно этот компонент тестовой системы управляет процессом линеаризации и обладает всеми необходимыми данными.

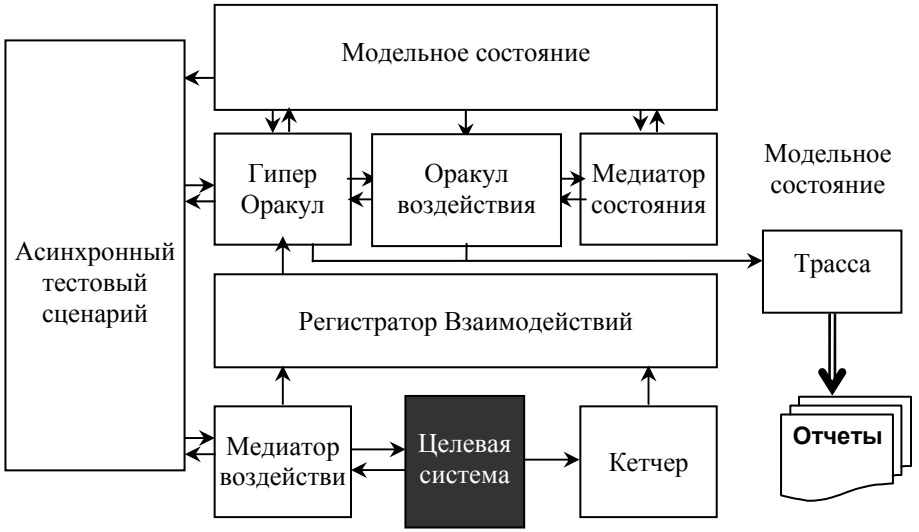

Рис. 16. Оценка качества тестирования в унифицированной архитектуре асинхронного теста

Принципы измерения покрытия асинхронных метрик в процессе тестирования представлены на Рис. 16. Гипероракул и оракул воздействия помещают информацию, необходимую для вычисления покрытия заданных пользователем асинхронных метрик покрытия, в трассу теста. В последствии трасса анализируется специальными инструментами, которые генерируют отчеты о качестве тестирования в терминах метрик покрытия асинхронной модели требований.

3.5. Унифицированная архитектура асинхронного теста

В заключение главы, посвященной тестированию систем с асинхронным интерфейсом, мы рассмотрим унифицированную архитектуру асинхронного теста в целом, еще раз сформулируем ответственности каждого из ее компонентов и обсудим правила взаимодействия компонентом между собой.

Процесс работы тестовой системы, построенной на основе унифицированной архитектуры асинхронного теста, представлен на Рис. 17. Управляет процессом тестирования асинхронный тестовый сценарий, который определяет какие тестовые воздействия и каким образом необходимо оказывать на целевую систему, и в какой момент необходимо оценивать корректность поведения целевой системы.

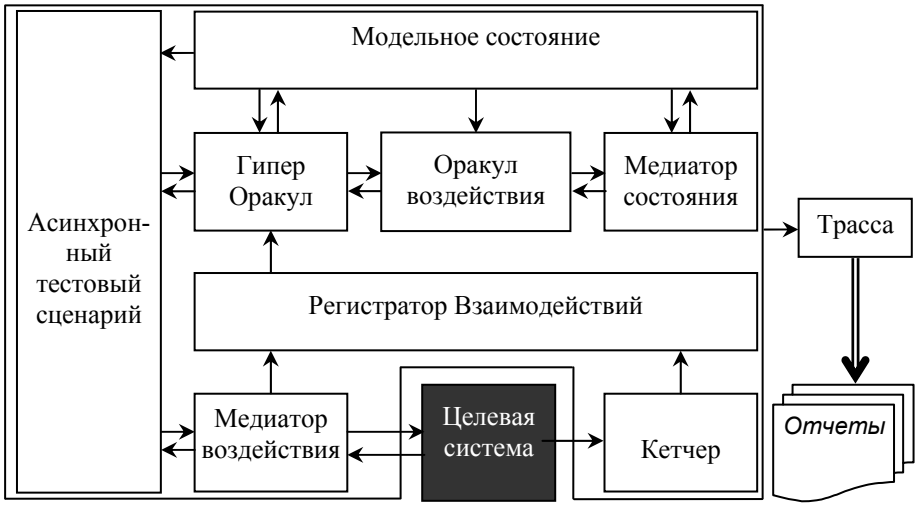

Рис. 17. Унифицированная архитектура асинхронного теста

Рассматриваемый в настоящей работе метод предполагает, что асинхронный тестовый сценарий является стационарным автоматным тестовым сценарием. Такой сценарий осуществляет обход графа, выполняя проход по каждой дуге в два этапа. На первом этапе выполняется «компактный» тестовый сценарий, соответствующий этой дуге. «Компактный» сценарий осуществляет набор тестовых воздействий на целевую систему, обращаясь для этого к медиатору воздействия. Медиатор воздействия выполняет необходимое воздействие на целевую систему, получает от нее непосредственную реакцию и регистрирует осуществленное взаимодействие с целевой системой в регистраторе взаимодействий. Непосредственная реакция при этом возвращается в «компактный» сценарий и может быть использована для формирования последующих тестовых воздействий. Также для этих целей может быть использовано модельное состояние, которое является доступным для чтения в локальном сценарии.

Другим активным участником первого этапа прохода по дуге является кетчер, который ответственен за сбор информации обо всех взаимодействиях, инициируемых целевой системой, и за регистрацию этих взаимодействий в регистраторе взаимодействий.

Переход ко второму этапу происходит после того, как «компактный» сценарий завершает свою работу. На втором этапе стационарный автоматный тестовый сценарий обращается к гипероракулу, чтобы оценить корректность поведения целевой системы и синхронизировать модельное состояние с состоянием целевой системы.

Работа гипероракула устроена следующим образом. Гипероракул получает от регистратора взаимодействий набор всех взаимодействий, зарегистрированных

во время первого этапа, а также имеющуюся информацию о порядке, в котором происходили эти взаимодействия. Эта информация представляет собой асинхронную модель поведения целевой системы, построенную по результатам наблюдения за целевой системой на первом этапе. Гипероракул проверяет корректность данной модели поведения относительно асинхронной модели требований, заданной до начала тестирования при помощи асинхронной спецификации. Для этого гипероракул действует согласно алгоритму оценки корректности поведения целевой системы. Он рассматривает все возможные линеаризации асинхронной модели поведения, обращаясь для проверки корректности каждого отдельного взаимодействия к оракулу воздействия. Оракул воздействия, в свою очередь, использует медиатор состояния в качестве подсказки для решения своей задачи. Медиатор состояния изменяет модельное состояние, чтобы указать, в какое состояние может перейти асинхронная модель требований по переходу, помеченному данными символами. После чего оракул воздействия фильтрует полученные переходы на основании их принадлежности к модели требований.

В процессе своей работы гипероракул изменяет модельное состояние, сохраняя и восстанавливая его при переходе от рассмотрения одной линеаризации к другой. В дополнении к своей основной области ответственности – оценке корректности поведения целевой системы, гипероракул отвечает за сохранение всей информации о процессе линеаризации в трассе теста. Также в трассу теста сохраняется информация о покрытии элементов асинхронных метрик покрытия, определенных разработчиком теста. За вычисление покрытых элементов и за ее сохранение в трассе отвечает оракул воздействия. На основе трассы теста в последствии проводится оценка качества тестирования и анализ найденных несоответствий между поведением целевой системы и моделью требований к нему.

После завершения второго этапа прохода по дуге графа стационарный автоматный тестовый сценарий при помощи алгоритма движения принимает решение либо осуществить переход по следующей дуге, либо завершить тестирование.

3.6. Результаты главы

В данной главе представлен метод автоматизированного тестирования систем с асинхронным интерфейсом. В начале главы было дано определение системам с асинхронным интерфейсом и рассмотрены ограничения подхода классических программных контрактов, не позволяющие полностью описать требования к системам с асинхронным интерфейсом. В последующих разделах были последовательно рассмотрены основные задачи тестирования применительно к асинхронным системам и сформулированы подходы по их решению, предлагаемые настоящим методом. На основе предложенных решений была разработана унифицированная архитектура асинхронной тестовой системы, определяющая архитектуру всех тестовых систем, построенных в соответствии с рассматриваемым методом.

4. Инструментальная поддержка тестирования систем с асинхронными интерфейсами

В настоящей главе мы рассмотрим инструментальную поддержку описываемого подхода к тестированию систем с асинхронным интерфейсом. Данная поддержка реализована в виде расширения набора инструментов CTesK, поддерживающих процесс тестирования по технологии UniTesK на платформе языка C.

4.1. Процесс тестирования в технологии UniTesK

Процесс тестирования, построенный по технологии UniTesK, разбивается на три этапа, в ходе которых решаются различные задачи, и которые требуют различных средств автоматизации. Первый этап, этап разработки тестового набора, является основополагающим для всего последующего тестирования. На этом этапе осуществляется проектирование и разработка автоматизированного тестового набора. Унифицированная архитектура теста UniTesK предоставляет базовые архитектурные решения, в рамках который происходит разработка тестового набора для каждой конкретной целевой системы.

Второй этап процесса тестирования, этап исполнения тестов, заключается в проведении тестирования целевой системы посредством выполнения автоматизированного тестового набора, разработанного на предыдущем шаге. Результатом второго этапа является трасса исполнения тестового набора, которая содержит информацию о событиях, происходивших в процессе тестирования.

Эта информация является основой для проведения третьего этапа, этапа анализа результатов тестирования, на котором осуществляется изучение различных аспектов проведенного тестирования, и, в частности, происходит:

- выделение и анализ ошибочного поведения целевой системы и тестового набора;
- оценка качества тестирования и анализ тестового покрытия.

Общие решения по автоматизации этих этапов технологии UniTesK заключаются в следующем. Автоматизированный тестовый набор разрабатывается на платформе одного из доступных языков программирования. Для этого языка создается набор инструментов, поддерживающих процесс тестирования по технологии UniTesK на платформе соответствующего языка программирования.

Для упрощения работы пользователя на этапе разработки тестового набора язык программирования расширяется небольшим набором конструкций, позволяющих описывать основные компоненты тестовой системы в более удобной и компактной форме. Например, для описания спецификаций интерфейсных операций в виде предусловий и постусловий вводится специальный вид функций. Также вводятся специальные конструкции для описания медиаторов, сценарных функций и тестовых сценариев. Код тестового набора, написанный на расширении языка программирования,

транслируется инструментами UniTesK в код на базовом языке программирования, после чего тестовый набор компилируется в исполнимую форму стандартными средствами разработки.

Технология UniTesK не предусматривает специальной поддержки этапа исполнения тестов, так как на этом этапе происходит выполнение автоматизированного тестового набора, которое является полностью автоматическим и не требует вмешательства человека. Результаты выполнения тестового набора сохраняются в виде трассы теста, формат которой является унифицированным для всех проекций технологии UniTesK на различные языки программирования.

Поэтому на этапе анализа результатов тестирования используются общие инструменты анализа результатов выполнения тестового набора, построенного по технологии UniTesK. Таких инструментов два. Генератор статических отчетов создает статистические отчеты в формате HTML. Эти отчеты содержат информацию о достигнутом покрытии метрик покрытия, графе автоматного тестового сценария и найденных несоответствиях между поведением реализации и моделью требований. Динамический анализатор трассы позволяет пользователю анализировать результаты тестирования в интерактивном режиме, предоставляя различные представления трассы (граф тестового сценария, MSC диаграмма [31], структурированное представление).

4.2. Проекция технологии UniTesK на язык программирования C

Процесс тестирования по технологии UniTesK на платформе языка программирования C поддерживается семейством инструментов CTesK. Работа семейства инструментов CTesK устроена по принципам, рассмотренным в предыдущем разделе.

Для описания основных элементов архитектуры тестовой системы UniTesK используется спецификационное расширение языка C (SEC), которое позволяет сделать эти описания более компактными и удобными. SEC расширяет синтаксис языка программирования C небольшим числом дополнительных конструкций, основными среди которых являются:

- спецификационные типы;
- подтипы (инварианты типов);
- инварианты переменных;
- спецификационные функции;
- медиаторные функции;
- сценарные функции;
- тестовые сценарии.

Спецификационные типы являются дополнительным видом типов SEC. Значения спецификационных типов располагаются в динамической памяти, управление которой осуществляется автоматически. Вместе с каждым таким

значением хранится таблица функций для создания, сравнения, копирования и удаления значений соответствующего типа.

Подтипы также представляют собой дополнительный вид типов, множеством значений которых является подмножество значений других типов, ограниченное инвариантом подтипа. Другим видом инвариантов, поддерживаемых в спецификационном расширении языка C, являются *инварианты переменных*, которые позволяют определить множество допустимых значений глобальных переменных программы.

Спецификационные функции предназначены для описания модели требований. Каждая спецификационная функция определяет спецификацию одной интерфейсной операции. Спецификационная функция состоит из сигнатуры и тела.

Сигнатура спецификационной функции задает сигнатуру, соответствующей интерфейсной операции. Параметры спецификационной функции и их типы описываются как параметры обычной функции языка C. Для описания набора переменных модельного состояния, с которыми работает данная функция используется дополнительная конструкция языка SEC, получившая название *ограничения доступа*. Ограничение доступа содержит список переменных модельного состояния с указанием вида доступа, осуществляемого к каждой переменной. Всего существует три вида доступа:

- доступ для чтения;
- доступ для записи;
- доступ для обновления.

Доступ для чтения означает, что функция читает значение переменной, но не может изменять его. Доступ для записи говорит о том, что функция записывает новое значение в переменную без предварительного чтения ее предыдущего значения. Доступ для обновления используется, когда функция и читает исходное значение переменной, и записывает в нее новое значение.

Тело спецификационной функции состоит из предусловия, постусловия и набора метрик покрытия. Предусловие представляется собой составной оператор, помеченный ключевым словом SEC **`pre`**. Код внутри составного оператора эквивалентен коду функции языка C, имеющей те же параметры, что и спецификационная функция, и возвращающей значение типа **`bool`**. Таким образом, этот код задает предикат на множестве значений параметров функции и переменных модельного состояния, называемый предусловием в спецификации интерфейсной операции.

Постусловие представляется собой составной оператор, помеченный ключевым словом SEC **`post`**. Код внутри составного оператора эквивалентен коду функции языка C, имеющей возвращаемое значение типа **`bool`**, а также набор параметров, состоящий из

- значений параметров спецификационной функции после вызова целевой операции (доступных по имени соответствующего параметра спецификационной функции);

- значений параметров спецификационной функции до вызова целевой операции (доступных по имени соответствующего параметра спецификационной функции, предваренному оператором @);
- возвращаемого значения спецификационной функции (доступного по имени спецификационной функции);
- значений переменных модельного состояния до вызова целевой операции (доступных по имени соответствующей переменной, предваренному оператором @).

Кроме того, непосредственно по имени переменной модельного состояния в постусловии доступны значения этой переменной после вызова целевой операции. Таким образом, код постусловия задает предикат на множестве значений параметров функции и переменных модельного состояния до и после вызова целевой функции, называемый постусловием в спецификации интерфейсной операции.

Набор метрик покрытия описывается последовательностью составных операторов, помеченных ключевым словом SEC **`coverage`** и идентификатором покрытия. Код внутри каждого отдельного составного оператора эквивалентен коду функции языка C, имеющей те же параметры, что и спецификационная функция, и возвращающей значение перечислимого типа. Этот перечислимый тип не определяется явным образом, а получается объединением всех идентификаторов, использованных в операторах **`return`** внутри данного составного оператора. Таким образом, составной оператор определяет метрику покрытия интерфейсной операции, а идентификатор покрытия задает имя этой метрики.

Тело спецификационной функции содержит одно предусловие, одно постусловие и ноль или более метрик покрытия. Предусловие и постусловие определяют спецификацию интерфейсной операции. Совокупность спецификаций интерфейсных операций, задаваемых всеми спецификационными функциями, образует модель требований.

Для задания модели поведения целевой системы используется другой вид функций SEC, называемых *медиаторными*. Каждой медиаторной функции соответствует некоторая спецификационная функция, которая получила название базовой спецификационной функции. Сигнатура медиаторной функции состоит из ключевого слова SEC **`mediator`**, идентификатора медиаторной функции, ключевого слова **`for`** и сигнатуры базовой спецификационной функции. Тело медиаторной функции представляет собой тело функции языка C, имеющей те же параметры и тот же тип возвращаемого значения, что и базовая спецификационная функция.

Медиаторная функция должна выполнить следующую последовательность действий:

- осуществить воздействие на целевую систему, моделируемое соответствующей спецификационной функцией;

- получить ответ от целевой системы:
- синхронизировать переменные модельного состояния с внутренним состоянием целевой системы;
- возвратить значение, моделирующее ответ целевой системы.

Каждое выполнение медиаторной функции в процессе тестирования рассматривается как выполнение одного взаимодействия между тестовой системой и целевой системой. Набор значений переменных модельного состояния перед началом работы медиаторной функции рассматривается как модельное состояние до вызова интерфейсной операции; набор значений параметров функции в совокупности с идентификатором спецификационной функции – как стимул; набор значений выходных параметров функции, возвращаемое значение и идентификатор спецификационной функции – как реакция; набор значений переменных модельного состояния после завершения работы медиаторной функции – как модельное состояние после вызова интерфейсной операции. Совокупность всех вызовов медиаторных функций в процессе тестирования образует модель поведения целевой системы.

Медиаторные функции не могут быть вызваны непосредственно. Все тестовые воздействия на целевую систему выглядят как вызовы спецификационных функций. Обработка таких вызовов происходит следующим образом:

- оракул, генерируемый из спецификационной функции, проверяет выполнение предусловия для полученных значений параметров и текущих значений переменных модельного состояния;
- если предусловие выполнено, то оракул запоминает значения переменных модельного состояния и входных параметров и вызывает медиаторную функцию;
- оракул получает управление от медиаторной функции, получая при этом значения выходных параметров и новые значения переменных модельного состояния;
- оракул проверяет постусловие для запомненных и новых значений параметров и переменных, осуществляя таким образом оценку корректности поведения целевой системы одновременно с выполнением тестового воздействия.

Для генерации последовательности вызовов спецификационных функций используются *тестовые сценарии*. Описание тестовых сценариев в SEC происходит при помощи специальной конструкции, в которой указывается механизм построения тестовых сценариев и данные, необходимые для этого механизма.

В системе тестирования CTesK в качестве основного механизма построения тестовых сценариев используется механизм dfsm. Исходные данные необходимые для механизма dfsm состоят из набора переменных сценария, функции вычисления вершины графа сценария и набора сценарных функций, определяющих множество стимулов графа сценария и локальные тестовые сценарии, приписанные к каждой дуге графа.

Значения переменных сценария определяют глобальное состояние сценария, которое может использоваться для определения графа сценария, правил построения последовательности тестовых воздействий и других аспектов поведения тестового сценария. Переменные сценария описываются как обычные глобальные переменные языка C и не имеют никакого специального синтаксического выделения.

Функция вычисления вершины графа задается в виде обычной функции языка C. Эта функция не имеет параметров и возвращает значение, представляющее вершину графа, которая соответствует текущему модельному состоянию и текущему глобальному состоянию сценария.

Изменение глобального состояния сценария и вызов спецификационных функций могут быть выполнены только из сценарных функций. *Сценарные функции* являются еще одним специальным видом функций SEC. Они не имеют параметров и возвращают значение типа `bool`. Каждая сценарная функция определяет набор стимулов графа сценария и отображение из каждого стимула в «компактный» тестовый сценарий, который может осуществлять тестовые воздействия посредством вызова спецификационных функций и изменять глобальное состояние сценария.

И в дополнение к основным исходным данным механизма dfsm также определяются функции инициализации и завершения работы тестового сценария, которые несут ответственность за выделение и освобождение необходимых ресурсов, а также за инициализацию глобального состояния сценария.

Для обработки описаний компонентов тестовой системы, выполненных на спецификационном расширении языка C, в состав системы тестирования CTesK входит SEC2C транслятор, который преобразует эти описания в код на языке C. Далее этот код обрабатывается обычным C компилятором и может быть собран в исполнимую программу или программы, образующие автоматизированный тестовый набор. Во время сборки к тестовому набору подключаются библиотеки, входящие в состав системы тестирования CTesK:

- библиотека абстрактных типов данных;
- библиотека поддержки тестовой системы;
- библиотека поддержки времени исполнения SEC;
- библиотека функций трассировки событий.

Библиотека абстрактных типов данных содержит основные функции для работы со спецификационными типами и набор библиотечных спецификационных типов, представляющих базовые типы языка C, а также основные виды коллекций, таких как множества, списки, мультимножества и отображения.

Библиотека поддержки тестовой системы включает в себя реализацию тех компонентов тестовой системы, построенной по технологии UniTesK, которые остаются неизменными вне зависимости от особенностей тестируемой

системы. Основным элементом этой библиотеки является реализация механизма построения тестовых сценариев dfsm.

Библиотека поддержки времени исполнения SEC содержит реализацию всех функций, необходимых SEC2C транслятору для организации работы сгенерированного кода.

Библиотека функций трассировки событий объединяет в себе всю функциональность по работе с трассой теста. В нее входят функции по созданию, настройке и управлению трассой теста, а также функции трассировки всех событий, происходящих в процессе тестирования.

В результате сборки сгенерированного из SEC кода и библиотек системы тестирования CTesK получается одна или несколько исполнимых программ, образующих автоматизированный тестовый набор. На втором этапе процесса тестирования, этапе исполнения тестов, происходит выполнение автоматизированного тестового набора, результатом которого является трасса исполнения тестового набора. На этапе анализа результатов тестирования эта трасса анализируется при помощи двух специализированных инструментов, рассмотренных в предыдущем разделе.

4.3. Тестирование систем с асинхронным интерфейсом на платформе языка C

Процесс тестирования систем с асинхронным интерфейсом при помощи набора инструментов CTesK максимально унифицирован с процессом тестирования синхронных систем. Тестирование асинхронных систем также разбивается на три этапа, инструментальная поддержка которых построена по аналогичным принципам.

На этапе разработки тестового набора осуществляется проектирование и разработка автоматизированного тестового набора на основе унифицированной архитектуры асинхронного теста. Эта архитектура предоставляет базовые решения, в рамках которых происходит построение тестового набора для каждой конкретной целевой системы.

Первым шагом при разработке тестового набора для системы с асинхронным интерфейсом является выделение набора интерфейсных операций. Интерфейсные операции представляют собой атомарные взаимодействия с целевой системой. Для асинхронных систем такие взаимодействия разделяются на два вида: взаимодействия, инициируемые окружением целевой системы, и взаимодействия, инициируемые самой целевой системой. Анализ требований к целевой системе, ее документации и других документов с целью выделения набора интерфейсных операций и их классификации является существенно неформальным процессом и поэтому его автоматизация не осуществляется.

На втором шаге разработки тестового набора происходит формализация требований к целевой системе посредством построения асинхронной модели требований. Описание модели требований выполняется в виде предусловий и постусловий интерфейсных операций, выделенных на предыдущем шаге. В

качестве нотации для записи предусловий и постусловий используются спецификационные функции SEC. Но так как спецификационные функции предназначены для описания требований к взаимодействиям, инициируемым окружением целевой системы, то для тестирования систем с асинхронным интерфейсом в спецификационное расширение языка C добавлен дополнительный вид функций, называемых *функциями-отложенными реакциями*.

Функции-отложенные реакции имеют очень много общего со спецификационными функциями. Они также состоят из сигнатуры и тела. Сигнатура функции-отложенной реакции отличается от сигнатуры спецификационной функции использованием ключевого слова **`reaction`** вместо ключевого слова **`specification`** и требованием отсутствия параметров. Последнее является следствием того, что функции-отложенные реакции предназначены для описания требований к взаимодействиям, инициируемым целевой системой, во время которых данные передаются только в одном направлении: от целевой системы к ее окружению. Эти данные описываются в виде возвращаемого значения функции-отложенной реакции. Данные, передаваемые в обратном направлении, отсутствуют, поэтому у функций-отложенных реакций и не может быть параметров. Тело функции-отложенной реакции состоит из предусловия, постусловия и набора метрик покрытия, описываемых по тем же правилам, что и в теле спецификационной функции.

Спецификационные функции и функции-отложенные реакции описывают спецификации интерфейсных операций, выделенных на предыдущем шаге, и в совокупности определяют асинхронную модель требований. Следует отметить, что принципы описания асинхронной модели требований во многом совпадают с описанием модели в синхронном случае, что является важным моментом для решения поставленной задачи.

На третьем шаге разработки тестового набора решаются три задачи:

- осуществляется связывание интерфейсных операций с целевой системой;
- описываются правила динамического построения асинхронной модели поведения целевой системы в процессе тестирования;
- определяются правила обновления модельного состояния.

Для решения первой задачи используются медиаторные функции SEC. Каждая медиаторная функция соответствует ровно одной спецификационной функции и описывает, какие действия необходимо выполнить, чтобы осуществить взаимодействие с целевой системой, моделируемое посредством данной спецификационной функции.

Задача динамического построения асинхронной модели поведения заключается в регистрации всех взаимодействий с целевой системой, происходящих в процессе тестирования, а также взаимозависимостей между ними. Зависимости между отдельными взаимодействиями задаются при помощи каналов и временных меток. Дальнейшее построение асинхронной модели поведения

осуществляется регистратором взаимодействий, реализованном в виде библиотечного компонента тестовой системы.

Процесс регистрации взаимодействий различается в зависимости от того, кто является инициатором этого взаимодействия: целевая система или ее окружение. Если инициатором взаимодействия является окружение, а во время тестирования – это тестовая система, то взаимодействие происходит по следующей схеме. Тестовая система вызывает спецификационную функцию, которая, в свою очередь, вызывает связанную с ней медиаторную функцию, а медиаторная функция уже выполняет необходимые действия для организации взаимодействия. После завершения взаимодействия оно автоматически регистрируется без специальных указаний со стороны разработчика тестового набора. При этом разработчик может полностью управлять процессом регистрации: он может указать к какому каналу отнести текущее взаимодействие, какую временную метку ему присвоить или вообще отключить автоматическую регистрацию.

В случае, когда инициатором взаимодействия является целевая система, механизм посредством которого осуществляется взаимодействие зависит от особенностей ее реализации. Только разработчик теста, имеющий информацию об этих особенностях, может указать тестовой системе, какие взаимодействия, инициированные целевой системой, имели место в процессе тестирования. Компонент, реализующий данную функциональность, называется в унифицированной архитектуре асинхронного теста кетчером. Реализация кетчера разрабатывается для каждой целевой системы индивидуально. Кетчер никак не выделяется синтаксически и представляет собой обычный код на языке C, использующий для регистрации взаимодействий одну из функций, принадлежащих интерфейсу регистратора взаимодействий. Канал и временная метка каждого взаимодействия указываются явно при его регистрации.

Еще одним компонентом унифицированной архитектуры асинхронного теста, разрабатываемым на третьем шаге, является медиатор состояния, который определяет правила обновления модельного состояния. Для описания медиатора состояния также используются медиаторные функции SEC. При тестировании систем с синхронным интерфейсом медиаторные функции также описывали правила осуществления тестового воздействия и синхронизации переменных модельного состояния с внутренним состоянием целевой системы. Однако, там не было необходимости разделять эти действия, так как они всегда происходили во время вызова медиаторной функции. При тестировании систем с асинхронным интерфейсом ситуация изменилась. В этом случае во время вызова медиаторной функции необходимо выполнять только осуществление тестового воздействия. Соответствующая синхронизация модельного состояния должна происходить уже во время оценки корректности поведения целевой системы, причем неоднократно.

Для разделения этих двух аспектов поведения медиаторной функции предложено использовать именованные составные операторы, помеченные ключевыми словами **`call`** и **`state`** соответственно. Таким образом, описание

правил осуществления тестового воздействия и синхронизации переменных модельного состояния остается там же, где и было при синхронном тестировании, но явным образом разносится по различным составным операторам. Это позволяет автоматически извлекать медиатор воздействия и медиатор состояния из одной медиаторной функции, внося минимальные изменения в работу разработчика теста.

Так как медиатор состояния необходим не только для взаимодействий, инициируемых окружением целевой системы, но и для взаимодействий, инициируемых самой целевой системой, то медиаторные функции должны быть определены не только для всех спецификационных функций, но и для всех функций-отложенных реакций. Отличие медиаторных функций для функций-отложенных реакций заключается в том, что в их теле требуется наличие только одного составного оператора, помеченного ключевым словом **`state`**, так как никакого воздействия для отложенной реакции осуществлять не требуется.

Четвертый шаг разработки тестового набора состоит в определении метрик покрытия интерфейсных операций. В случае тестирования асинхронных систем этот шаг практически ничем не отличается от синхронного случая. Асинхронные метрики покрытия определяются так же, как и синхронные: посредством специальных именованных блоков в теле спецификационных функций.

На пятом шаге разрабатываются тестовые сценарии. Чтобы определить тестовый сценарий в SEC необходимо указать механизм построения тестового сценария и исходные данные, необходимые для этого механизма. Для систем с синхронным интерфейсом в системе тестирования CTesK реализован единственный механизм построения тестовых сценариев – механизм dfsm. В качестве исходных данных механизма dfsm выступают

- функция инициализации тестового сценария;
- функция завершения тестового сценария;
- функция вычисления вершины графа сценария;
- набор сценарных функций, задающих множество дуг графа сценария.

Для поддержки асинхронного стационарного тестирования в набор исходных данных для механизма dfsm были добавлены три дополнительных элемента:

- функция сохранения модельного состояния;
- функция восстановления модельного состояния;
- функция проверки стационарности текущего модельного состояния.

Назначение всех прежних элементов осталось без изменений, поэтому разработка асинхронных тестовых сценариев не вызывает проблем у пользователей, знакомых с разработкой синхронных тестовых сценариев. Единственное существенное изменение коснулось семантики вызова спецификационных функций: если при синхронном тестировании такой вызов ведет к немедленному обновлению модельного состояния, синхронизирующем

его с состоянием реализации, то при асинхронном тестировании модельное состояние остается неизменным вплоть до завершения очередного цикла работы сценарной функции.

Дополнительные исходные данные представляют собой обычные функции языка C. Функция сохранения модельного состояния читает значения переменных модельного состояния и сохраняет их в объекте спецификационного типа, возвращаемого из функции. Функция восстановления модельного состояния получает объект спецификационного типа, созданный ранее функцией сохранения, и восстанавливает значения переменных модельного состояния такими, какими они были на момент вызова функции сохранения. И, наконец, функция проверки стационарности текущего модельного состояния возвращает булевское значения, отражающее стационарность текущего модельного состояния.

В дополнение к механизму dfsm был реализован новый механизм построения тестовых сценариев ndfsm, отличающийся от dfsm только алгоритмом движения по графу сценария. Этот механизм упрощает разработку тестовых сценариев, так как он позволяет не беспокоиться о недетерминированности отдельных переходов графа, если другие переходы гарантированно образуют детерминированный сильносвязный полный остовный подграф.

Второй этап процесса тестирования, основанного на технологии UniTesK, этап исполнения тестов, в асинхронном случае также не предусматривает дополнительной автоматизации, как и при тестировании синхронных систем.

Третий этап, анализ результатов тестирования систем с асинхронным интерфейсом внешне не сильно отличается от анализа результатов синхронного тестирования. Генератор статических отчетов также создает статистические отчеты в формате HTML, содержащие информацию о достигнутом покрытии метрик покрытия, графе автоматного тестового сценария и найденных несоответствиях между поведением реализации и моделью требований. Отличия здесь заключаются только в ином описании найденных несоответствий. Если при синхронном тестировании можно однозначно определить какое взаимодействие с целевой системой было некорректным, то при асинхронном тестировании некорректной является вся асинхронная модель поведения. Поэтому детальный отчет о найденном несоответствии содержит информацию не об одном взаимодействии, а обо всей модели поведения, участвовавшей в неуспешной линеаризации.

Отличия в динамическом анализаторе трассы в первую очередь касаются MSC представления. Отложенные реакции в нем отображаются отличным образом от стимулов. Участки MSC диаграммы, описывающие набор взаимодействий информация о взаимном порядке которых отсутствует, отображаются специальным образом, с выделением пунктиром. А для представления различных линеаризаций модели поведения, проверявшихся алгоритмом оценки корректности, создаются отдельные MSC диаграммы.

В результате расширения функциональности набора инструментов CTesK инструментальная поддержка метода тестирования систем с асинхронным

интерфейсом была интегрирована с инструментальной поддержкой технологии UniTesK для систем с синхронным интерфейсом, что позволяет разработчикам тестов плавно переходить от тестирования синхронных систем к тестированию асинхронных систем. Помимо модификации самих инструментов был доработан комплект пользовательской документации системы тестирования CTesK, включающий в себя справочник по языку SEC, описание библиотечных компонентов и руководство пользователя. Кроме того, специально для тренинг-курса по системе тестирования CTesK был разработан цикл лекций и практических занятий, посвященный обучению слушателей технологии тестирования систем с асинхронным интерфейсом.

5. Опыт применения технологии UniTesK для тестирования систем с асинхронным интерфейсом

В данной главе мы рассмотрим опыт применения технологии UniTesK для тестирования систем с асинхронным интерфейсом. Рассмотренный метод и его реализация в системе тестирования CTesK использовались в шести проектах по тестированию различного программного обеспечения, проводившихся в Институте Системного Программирования РАН.

5.1. Реализация протокола IPv6

Впервые рассматриваемый подход использовался для тестирования реализации протокола IPv6 от Microsoft Research, которое проводилось в рамках совместного проекта ИСП РАН и Microsoft Research, Cambridge [32,33].

Асинхронность взаимодействий реализации IPv6 с нижележащими и вышележащими уровнями стека протоколов играет ключевую роль в функциональности системы. Поэтому спецификация и тестирование асинхронных аспектов поведения являлось важной составляющей проекта MSR IPv6.

Для формализации требований к целевой системе и проведения тестирования использовалась первая версия системы тестирования CTesK, в которую были включены возможности для работы с асинхронными интерфейсами на основе рассмотренного подхода.

Требования к поведению тестируемой системы извлекались из 10 RFC (RFC 2460, RFC 2461, RFC 2462, RFC 2463, RFC 2464, RFC 3513, RFC 2373, RFC 2292, RFC 2553 и RFC 2675). После изучения этих требований происходила их формализация в виде предусловий и постусловий интерфейсных операций, моделирующих взаимодействия с тестируемой системой.

Тестирование проводилось при помощи 15 стационарных асинхронных тестовых сценариев, которые эмулировали различные варианты использования функций протокола IPv6 на основе обхода детерминированных графов. Эти сценарии позволили обнаружить 4 ошибки в целевой системе, 2 из которых являлись фатальными и приводили к перезагрузке системы при получении определенной последовательности пакетов IPv6. Такая последовательность

пакетов может быть использована для организации атаки типа «Отказ в обслуживании».

На примере этого проекта были апробированы основные решения по описанию требований к системам с асинхронным интерфейсом и использованию этого описания для проведения тестирования. Инструментальная поддержка разработки спецификаций и тестов была реализована в тот момент на минимальном уровне, но важным результатов проекта стало получение подтверждения применимости предлагаемого метода организации тестирования систем с асинхронным интерфейсом к телекоммуникационному программному обеспечению. Полученный в рамках проекта опыт нашел свое применение в дальнейшей доработке технологии и в развитии ее инструментальной поддержки.

5.2. Функциональность протокола Mobile IPv6

Работа по тестированию MSR IPv6 получила продолжение в проекте по тестированию функций Mobile IPv6 на примере реализации Microsoft MIPv6 для Windows CE 4.1 и Windows XP. В этом проекте было проведено тестирование соответствия реализации базовых функций IPv6 в Windows CE на основе спецификаций и тестов, разработанных для MSR IPv6. Кроме того, проводилось тестирование на соответствие реализации ряда функций протокола Mobile IPv6 спецификациям протокола, а также соответствие реализации служебного протокола MLD (Multicast Listener Discovery) спецификации протокола.

Требования к реализации Mobile IPv6 извлекались из 13-го проекта стандарта Mobile IPv6, так как именно эту спецификацию поддерживала тестируемая реализация. Требования к реализации MLD извлекались из спецификаций первой версии протокола MLD, RFC 2710. В качестве дополнительных источников требований использовались RFC 2462 и RFC 2473. Эти требования были описаны в виде асинхронной модели требований, которая в дальнейшем использовалась для оценки корректности поведения тестируемой системы.

Для организации тестирования использовалась распределенная архитектура тестового набора, в рамках которой в процессе тестирования участвовало несколько систем, объединенных локальной сетью. На выделенной инструментальной машине решались основные задачи тестовой системы, заключающиеся в организации тестовых воздействий и оценке корректности поведения тестируемой системы. Другие машины использовались для оказания тестовых воздействий на тестируемую систему и получения реакций от нее.

5.3. Протокол MPEG-2 IPMP

Еще одним примером применения технологии UniTesK для тестирования телекоммуникационных протоколов был пилотный проект по тестированию реализации протокола MPEG-2 IPMP, выполненный ИСП РАН совместно с Morphibius Technology Inc., Канада.

Основной целью данного проекта являлся анализ возможностей использования тестирования на соответствие стандарту для обеспечения корректности взаимодействия между различными реализациями этого стандарта. Одним из направлений работ по проекту стала разработка формальных спецификаций и тестового набора для нескольких важных вариантов использования MPEG-2 IPMP:

- обработка управляющей информации протокола IPMP;
- работа с IPMP дескрипторами;
- обработка потока данных IPMP.

Выбранные аспекты функциональности IPMP включали в себя одновременное участие целевой системы в нескольких взаимодействиях, поэтому для формализации требований стандарта использовалась асинхронная модель требований. По этой же причине тестовые сценарии были построены при помощи асинхронного стационарного механизма построения тестовых сценариев.

Таким образом, данный проект еще раз подтвердил востребованность предложенного метода по спецификации требований и тестированию систем с асинхронным интерфейсом в области телекоммуникационных протоколов.

5.4. Компоненты распределенной операционной системы для сенсорных сетей

В дополнение к опыту тестирования телекоммуникационных систем предложенный подход применялся для тестирования асинхронных аспектов поведения многопроцессных и многопотоковых систем. Первым таким опытом был пилотный проект по тестированию отдельных компонентов TinyOS – распределенной операционной системы для сенсорных сетей, проведенной ИСП РАН совместно с российской компанией Luxoft.

Особенностью работы с TinyOS является то, что для написания компонентов используется специфический C-подобный язык nesC. На этом языке интерфейс компонентов операционной системы описывается двусторонним образом. С одной стороны описывается набор интерфейсных функций, предоставляемых данным компонентом, с другой стороны описывается набор функций, которые должен предоставить компонент, использующий сервисы данного компонента. Таким образом, функциональность каждой интерфейсной функции может заключаться не только в вычислении результата, но и в обращении с определенными значениями параметров к функциям пользователя, причем дальнейшее поведение может зависеть от результатов, полученных от этих пользовательских функций.

При описании требований к такого рода системам вызов интерфейсной функции тестируемого компонента рассматривался в качестве стимула на целевую систему, а вызов пользовательской функции из тестируемого компонента – в качестве реакции целевой системы. Соответственно, данные, возвращаемые в тестируемый компонент из пользовательской функции,

рассматривались в качестве стимула на целевую систему, а данные, возвращаемые из интерфейсной функции, - в качестве ее реакции.

Стимулы, моделирующие возврат управления из пользовательской функции, являлись достаточно специфичными, так как они могли быть посланы целевой системе только в ответ на специальный класс реакций, полученных с ее стороны. Тем не менее, работа с этими стимулами не вызвала никаких проблем. Ограничения на ситуации, когда можно подавать такие стимулы, были описаны в предусловии соответствующих спецификационных функций и автоматически отслеживались в процессе тестирования. Также эти ограничения учитывались при разработке тестовых сценариев, чтобы обеспечивать согласованность выполняемых тестовых воздействий с предусловиями стимулов.

В рамках проекта тестировался интерфейс TinyDB, предоставляющий доступ к показаниям распределенных сенсоров в виде операций чтения/записи виртуальной базы данных. Для него были формализованы требования к 6 интерфейсным операциям и разработаны 5 асинхронных тестовых сценариев. Результатом использования метода тестирования систем с асинхронным интерфейсом в данном проекте стала первая демонстрация возможности успешной работы этого метода за рамками телекоммуникационных протоколов.

Основным разработчиком спецификаций и тестовых сценариев во всех четырех проектах, рассмотренных выше, является Николай Пакулин.

5.5. Ядро операционной системы реального времени

Наиболее значимой работой по тестированию асинхронных аспектов поведения многопотоковых и многопроцессных систем является проект по тестированию ядра POSIX-совместимой операционной системы реального времени ОС2000. Эта операционная система предоставляет пользователю прикладной программный интерфейс, состоящий из 482 функций. В рамках проекта все функции были проанализированы и сгруппированы по подсистемам согласно реализуемой ими функциональности. Список подсистем приведен в таблице 2. Во втором столбце таблицы указано число интерфейсных функций, принадлежащих соответствующей подсистеме.

Название подсистемы	Размер	Асинхронность
Потоки управления	39	A1
Планировщик	10	A1
Сигналы	19	A1
Синхронизация	30	A2
Очереди сообщений	10	A2
Прерывания	14	A1
Время и таймеры	25	A1
Поддержка многопроцессорности	15	A3
Память	9	A3
Ввод-вывод	61	A2
Асинхронный ввод-вывод	8	A1
Файловая система	11	A3
Терминалы	10	A1
Сокеты	39	A2
IEEE 754	16	A1
Математические функции	37	S
Строковые функции	50	S
Функции протоколирования	53	A1
Вспомогательные функции	26	S

Таб. 2. Разбиение функций ОС2000 на подсистемы.

Анализ функциональности выделенных подсистем показал, что только три подсистемы из девятнадцати не требуют применения методов тестирования систем с асинхронным интерфейсом и могут быть полностью протестированы на основе методов синхронного тестирования. Эти подсистемы помечены в таблице 2 символом S. Они реализуют математические функции, функции для работы со строками и вспомогательные функции.

Остальные шестнадцать подсистем либо содержат в своем интерфейсе взаимодействия, инициируемые тестируемой системой, либо их тестирование не может быть проведено на требуемом уровне качества с использованием только последовательных взаимодействий. К первой группе относятся девять подсистем, помеченных символом A1, а ко второй группе – семь подсистем, помеченных символами A2 и A3.

Подсистема управления потоками содержит такие асинхронные реакции как:

- старт нового потока управления;
- вызов функции инициализации во время динамической инициализации переменных;

- вызов функций свертывания и функций деструкторов локальных данных потока в процессе завершения работы потока.

В планировщике основным видом асинхронных реакций является переключение между потоками управления. Также существенную роль в данной подсистеме играют такие асинхронные реакции как прерывание или доставка сигнала, которые относятся к одноименным подсистемам.

При описание требований к таймерам, асинхронному вводу-выводу и терминалам появляется необходимость в использовании сигналов, а при работе с числами с плавающей точкой (IEEE 754) и с функциями протоколирования в качестве асинхронных реакций рассматриваются вызовы функций, передаваемых в подсистему в качестве параметров интерфейсных функций.

Подсистемы, помеченные символом A2, не содержат асинхронных реакций в явном виде, но требуют рассмотрения некоторых вызовов интерфейсных функций как двух отдельных взаимодействий: вызова интерфейсной функции и возврата управления из нее. В этом случае возврат управления инициируется тестируемой системой и потому описывается как асинхронная реакция. Необходимость в разбиение одного вызова на два взаимодействия появляется в тех случаях, когда вызов интерфейсной функции может быть заблокирован до наступления некоторого события, такого как получение ожидаемых данных или освобождения ожидаемого ресурса. Если рассматривать такой блокируемый вызов как одно взаимодействие, то тестовая система должна быть заблокирована до завершения этого взаимодействия и она не сможет в это время осуществлять другие тестовые воздействия, в том числе и те, которые могут привести к разблокировке вызова. Поэтому тестирование блокируемых вызовов только синхронными методами обладает существенными ограничениями.

Рассмотрим в качестве примера подсистему синхронизации, предоставляющую возможности по синхронизации потоков управления при помощи семафоров, мьютексов и условных переменных. Основная функциональность этой подсистемы реализуется посредством блокирующихся функций, которые возвращают управление только после того как вызвавший их поток управления получает в свое распоряжение запрошенный ресурс. Например, функция pthread_mutex_lock, обеспечивающая захват мьютекса, блокируется, если мьютекс уже захвачен другим потоком, и возвращает управление только после того, как мьютекс будет освобожден и используемая стратегия планирования выберет данный поток среди всех потоков, стоящих в очереди на захват данного мьютекса. Если рассматривать вызов функции pthread_mutex_lock как атомарное взаимодействие, то при синхронном тестировании нельзя добиваться блокировки этой функции, так как у тестовой системы не будет возможности разблокировать этот вызов. Таким образом, при использовании синхронных методов основная функциональность этой функции останется недоступной для тестирования, что является неприемлемым для достижения требуемого качества тестирования.

При использовании асинхронного подхода вызов функции pthread_mutex_lock разбивается на два взаимодействия: вызов функции pthread_mutex_lock и возврат управления из нее. Первое взаимодействие рассматривается как асинхронное воздействие, а второе – как асинхронная реакция. В этом случае, блокировка функции pthread_mutex_lock не приводит к блокировке тестовой системы и тестовая система имеет возможность продолжить тестирование, выполняя блокировку произвольного числа потоков управления, освобождая мьютекс, изменяя значения его атрибутов, то есть не имея ограничений на осуществляемые тестовые воздействия.

Очереди сообщений содержат блокируемые вызовы функций чтения и записи в очередь сообщений в случаях, когда очередь пуста и полна соответственно. Поэтому для тестирования этих функций также требуется применение асинхронных методов тестирования. Аналогичная картина наблюдается в подсистемах ввода-вывода и сокетов, в которых присутствуют блокируемые вызовы функций чтения и получения сообщений.

Подсистемы, помеченные символом A3, не содержат ни явных асинхронных реакций, ни блокируемых вызовов функций. Но их реализация должна обеспечивать корректность работы при одновременном обращении к этим подсистемам из различных потоков управления. Поэтому для проведения тестирования с одновременным участием целевой системы в нескольких взаимодействиях требуется использование асинхронных методов.

Опыт этого проекта показал, что подавляющее большинство подсистем операционной системы требует использования методов асинхронного тестирования для обеспечения высокого качества тестового набора. В ходе проекта было выделено три класса функций прикладного программного интерфейса, для тестирования которых требуется применение асинхронных методов.

Первый класс функций составляют функции, приводящие к появлению явных асинхронных реакций, таких как сигналы, вызов функций обратного интерфейса, старт, приостановка и завершение потоков управления и т.д. Второй класс образуют блокируемые функции, управление из которых возвращается только при наступлении определенных условий, таких как освобождение требуемых ресурсов, поступление необходимого числа данных и т.д. В третий класс входят функции, которые должны обеспечивать корректность своей работы при одновременном вызове из различных потоков управления и для тестирования которых требуется одновременное участие целевой системы в нескольких взаимодействиях.

Одним из результатов проекта является подтверждение того, что во всех трех случаях использование предложенного метода тестирования систем с асинхронным интерфейсом позволяет преодолеть ограничения синхронного тестирования и обеспечить необходимый уровень качества тестирования. С другой стороны, проект продемонстрировал, что реализация метода тестирования асинхронных систем в системе тестирования CTesK предоставляет возможность тестирования действительно сложных целевых

систем с нетривиальной функциональностью, таких как ядро операционной системы реального времени.

5.6. Прикладные бинарные интерфейсы ОС Linux

Еще одним крупным проектом, в рамках которого метод спецификации и тестирования систем с асинхронным интерфейсом играет существенную роль, является проект по формализации требований стандартов и разработки автоматизированного тестового набора для бинарных интерфейсов операционной системы Linux, проводимый в рамках проекта по созданию Центра верификации ОС Linux [34].

В этом проекте предполагается формализация требований к 1532 функциям прикладного бинарного интерфейса ОС Linux, описанным в стандарте Linux Standard Base Core 3.1 [35]. Данные функции были сгруппированы по двухуровневой схеме, согласно реализуемой ими функциональности. На верхнем уровне функции были распределены по кластерам, а кластеры были разбиты на подсистемы. Кроме того, если размеры подсистемы оказывались достаточно большими и входящие в нее функции логически разделялись на несколько групп, подсистема разбивалась на группы функций. Список всех подсистем и групп функций, описанных в Linux Standard Base Core 3.1, представлен на сайте проекта [34].

Для каждой группы функций проведен анализа потребности в применении метода асинхронного тестирования. Из 137 групп функций только 50 групп (36%) могут быть полностью протестированы синхронным образом. Для полного описания требований к 13 группам функций (9%) необходимо использовать асинхронные реакции. 18 групп функций (13%) содержат блокирующиеся функции, а для тестирования 56 групп функций (42%) требуется одновременное участие целевой системы в нескольких взаимодействиях со своим окружением.

5.7. Результаты апробации

В настоящей главе был рассмотрен опыт применения технологии UniTesK к тестированию систем с асинхронным интерфейсом, а также его реализации в наборе инструментов CTesK. Этот опыт основывается на шести проектах по тестированию различного программного обеспечения, в ходе которых рассматриваемый подход нашел свое применение.

Результаты этих проектов продемонстрировали, что использование предложенного метода для тестирования программных систем с асинхронным интерфейсом позволило существенно повысить качество проводимого тестирования за счет появления возможности работы в таких тестовых ситуациях, которые приходилось избегать при синхронном тестировании. К этим ситуациям относятся:

- участие тестируемой системы в нескольких одновременных взаимодействиях с окружением;
- блокируемые вызовы функций;

- получение асинхронных реакций от целевой системы.

Опыт шести проектов показал, что спектр программных систем, где метод асинхронного тестирования является востребованным, достаточно широк. В него входят как телекоммуникационные протоколы и программное обеспечение для встраиваемых систем, так и программные интерфейсы операционных систем общего назначения.

Другим результатом выполненных проектов является подтверждение того, что предложенные решения по тестированию систем с асинхронным интерфейсом и их реализация в наборе инструментов CTesK позволяют успешно проводить систематизированное тестирование самых сложных асинхронных систем, таких как операционные системы и телекоммуникационные протоколы.

6. Заключение

В настоящей работе рассмотрен метод тестирования систем с асинхронным интерфейсом, являющийся составной частью технологии UniTesK. В результате исследования подхода классических программных контрактов, лежащего в основе технологии UniTesK для тестирования синхронных систем, были выявлены основные проблемы, препятствующие его применению для тестирования систем с асинхронным интерфейсом. Эти проблемы заключаются в том, что базовые предположения относительно поведения целевой системы, лежащие в основе подхода, оказываются неприменимыми к системам с асинхронным интерфейсом.

Для систем с асинхронным интерфейсом оказывается не верно, что:

- любое взаимодействие между тестируемой системой и ее окружением может инициироваться только окружением;
- все взаимодействия происходят строго последовательно, то есть следующее взаимодействие начинается только после завершения предыдущего.

По результатам исследования особенностей систем с асинхронным интерфейсом были разработаны новые математические модели, позволяющие корректным образом сформулировать основные задачи тестирования для систем с асинхронным интерфейсом, и были исследованы алгоритмы, решающие эти задачи. Ниже мы рассмотрим каждую задачу в отдельности.

Задача оценки корректности поведения целевой системы решается следующим образом:

- требования к функциональности тестируемой системы описываются в виде формальных спецификаций, которые неявным образом задают автомат с отложенными реакциями;
- поведение целевой системы, наблюдаемое в процессе тестирования, представляется в виде частично-упорядоченного множества взаимодействий между целевой системой и ее окружением;

- зафиксированное множество взаимодействий проверяется на соответствие автомату с отложенными реакциями, заданному формальными спецификациями требований.

Если множество соответствует автомату, то считается, что поведение целевой системы в процессе тестирования удовлетворяло предъявляемым к нему требованиям. В противном случае считается, что тестирование обнаружило несоответствие между поведением целевой системы и формальными спецификациями требований. Проверка соответствия множества взаимодействий автомату с отложенными реакциями осуществляется при помощи алгоритма проверки корректности поведения, описанного в одноименном разделе.

Задача генерации тестовых данных для систем с асинхронным интерфейсом существенно осложняется большой вычислительной сложностью алгоритма оценки корректности поведения. Чтобы решить эту проблему и сохранить при этом одно из основных достоинств технологии UniTesK для синхронных систем – автоматическое построение сложных последовательностей тестовых воздействий, использован подход так называемого стационарного тестирования. Идея этого подхода заключается в том, что алгоритм оценки корректности применяется не один раз в конце теста, а постоянно в процессе тестирования, по завершению некоторого шага теста.

Задача оценки качества тестирования для систем с асинхронным интерфейсом решается также как и в синхронном случае: на основе разбиения множества всех возможных взаимодействий на конечное число классов эквивалентности. Однако, понятие покрытия класса эквивалентности существенно отличается в связи с многочисленностью корректных путей в автомате с отложенными реакциями, что потребовало использования нового алгоритма подсчета покрытия классов эквивалентности по результатам работы асинхронного теста.

На основе рассмотренных математических моделей и алгоритмов была разработана унифицированная архитектура асинхронной тестовой системы, определяющая архитектуру всех тестовых систем, построенных в соответствии с предложенным методом. В этой архитектуре были выделены основные компоненты тестовой системы, проведено разграничение ответственностей между компонентами и определены правила взаимодействия между ними.

Инструментальная поддержка представленного метода была выполнена в виде расширения набора инструментов CTesK, поддерживающих технологию UniTesK на платформе языка C. Доработанный набор инструментов сохранил поддержку исходного метода тестирования синхронных систем, и в то же время получил дополнительные возможности по тестированию систем с асинхронным интерфейсом на основе представленного в настоящей работе метода.

Предложенный метод и его инструментальная поддержка были апробированы в шести проектах по тестированию различного программного обеспечения. В качестве тестируемых систем выступали реализации протоколов IPv6, Mobile IPv6, MPEG-2 IPMP, отдельные компоненты распределенной операционной

системы для сенсорных сетей TinyOS, ядро операционной системы реального времени ОС2000 и функции стандартного бинарного интерфейса операционной системы Linux. Вышеозначенные проекты показали востребованность методов тестирования систем с асинхронным интерфейсом, а также работоспособность предложенных решений и их программной реализации, обеспечивших проведение систематизированного тестирования таких сложных асинхронных систем, как операционные системы и реализации телекоммуникационных протоколов.

Литература

1. Euler E.E., Jolly S.D., Curtis H.H. The Failures of the Mars Climate Orbiter and Mars Polar Lander: A Perspective from the People Involved // Proceedings of Guidance and Control 2001. American Astronautical Society, AAS 01-074, 2001.
2. Report on the Loss of the Mars Climate Orbiter Mission. JPL D-18441, 1999.
3. Porrello A.M. Death and Denial: The Failure of the Therac-25, A Medical Linear Accelerator.
4. Leveson N.G., Clark S.T. An Investigation of the Therac-25 Accidents // Computer. July, 1993. P. 18-41.
5. Patriot missile defense: Software problems led to system failure at Dhahran, Saudi Arabia. Report GAO/IMTEC-92-26, Information Management and Technology Division, US General Accounting Office. Washington DC, Feb. 11992.
6. Berry D. M. Formal methods: the very idea, some thoughts about why they work when they work. // Science of Computer Programming #42(1). P. 11-27.
7. Floyd R. W. Assigning meanings to programs // Proceedings Symp. Appl. Math., 19. in: J.T.Schwartz (ed.), Mathematical Aspects of Computer Science. P. 19-32. American Mathematical Society, Providence, R.I., 1967.
8. Francez N. Verification of programs. Addison-Wesley Publishers Ltd., 1992.
9. Manna Z. Mathematical theory of computation. McGraw-Hill, 1974.
10. Manna Z., Pnueli A. The Temporal Logic of Reactive and Concurrent Systems. Springer-Verlag, 1991.
11. Blackburn M., Busser R., Nauman A., Knockerbocker R., Kasuda R. Mars Polar Lander Fault Identification Using Model-Based Testing // 26th Annual NASA Goddard Software Engineering Workshop. 27-29 November, 2001.
12. Dalal S. R., Jain A., Karunanithi N., Leaton J. M., Lott C. M. Model-Based Testing of a Highly Programmable System // Proceedings of ISSRE-98. 5-7 November 1998.
13. Dalal S. R., Jain A., Karunanithi N., Leaton J. M., Lott C. M., Patton G. C., Horowitz B. M. Model-Based Testing in Practice // Proceedings of the ICSE'99. May 1999.
14. Farchi E., Hartman A., Pinter S. S. Using a Model-Based Test Generator to Test for Standard Conformance // IBM Systems Journal, 2002. V. 41, No. 1, P. 89-110.
15. Gronau A., Hartman A., Kirshin A., Nagin K., Olvovsky S. A Methodology and Architecture for Automated Software Testing // IBM Research Laboratory in Haifa Technical Report, 2000.
16. Offutt A. J., Liu S., Abdurazik A. Generating Test Data from State-Based Specifications // Journal of Software Testing, Verification & Reliability. V. 13, No. 1, March 2003.
17. Al-Ghafees M., Whittaker J. A. Markov Chain-based Test Data Adequacy Criteria: a Complete Family. // IS June 2002. P. 13-37.
18. Bourdonov I., Kossatchev A., Kuliamin V., Petrenko A. UniTesK Test Suite Architecture // Proceedings of FME, 2002. LNCS 2391. P. 77-88. Springer-Verlag.

19. Кулямин В.В., Петренко А.К., Косачев А.С., Бурдонов И.Б. Подход UniTesK к разработке тестов // Программирование, 29(6). 2003. С. 25–43.
20. Баранцев А.В., Бурдонов И.Б., Демаков А.В., Зеленов С.В., Косачев А.С., Кулямин В.В., Омельченко В.А., Пакулин Н.В., Петренко А.К., Хорошилов А.В. Подход UniTesK к разработке тестов: достижения и перспективы // Труды Института системного программирования РАН, №5, 2004. [HTML] (http://www.citforum.ru/SE/testing/unitesk).
21. Meyer B. Applying 'Design by Contract' // IEEE Computer, vol. 25, October 1992. P. 40-51.
22. [HTML,PDF] (http://www.unitesk.com).
23. Burdonov I., Kossatchev A., Petrenko A., Cheng S., Wong H. Formal Specification and Verification of SOS Kernel // BNR, NORTEL Design Forum, June 1996.
24. Monin J. F., Hinchey M. G. Understanding Formal Methods. Springer-Verlag, 2003.
25. Бурдонов И.Б., Косачев А.С., Кулямин В.В. Использование конечных автоматов для тестирования программ. // Программирование, 2000, №2.
26. Бурдонов И.Б., Косачев А.С., Кулямин В.В. Неизбыточные алгоритмы обхода ориентированных графов. Детерминированный случай. // Программирование. 2003. №5. С. 11-30.
27. Бурдонов И.Б., Косачев А.С., Кулямин В.В. Неизбыточные алгоритмы обхода ориентированных графов. Недетерминированный случай. // Программирование. 2004. №1. С. 4-24.
28. Бурдонов И.Б., Косачев А.С., Кулямин В.В. Асинхронные автоматы: классификация и тестирование. // Труды ИСП РАН. 2003. №4. С. 7-84.
29. Хорошилов А.В. Спецификация и тестирование компонентов с асинхронным интерфесом. Диссертация на соискание ученой степени кандидата физико-математических наук. Москва, ИСП РАН, 2006.
30. Липский В. Комбинаторика для программистов. Москва, Мир, 1988.
31. Message Sequence Charts. ITU recommendation Z.120.
32. Agamirzian I., Groshev S.G., Khoroshilov A.V., Kluchnikov G.N., Kossatchev A.S., Omelchenko V.A., Pakoulin N.V., Petrenko A.K., Shnitman V.Z. MSR IPv6 verification project, December 2001. [HTML] (http://www.ispras.ru/~RedVerst).
33. Agamirzian I., Groshev S.G., Khoroshilov A.V., Kluchnikov G.N., Kossatchev A.S., Omelchenko V.A., Pakoulin N.V., Petrenko A.K., Shnitman V.Z. MSR IPv6 Verification within the CTesK-lite Framework, April 2002. [HTML] (http://www.ispras.ru/~RedVerst).
34. [HTML] (http://www.linuxtesting.ru).
35. Linux Standard Base Core 3.1. ISO/IEC IS-23360.[HTML] (http://refspecs.freestandards.org/lsb.shtml).