Полное тестирование с открытым состоянием ограниченно недетерминированных систем

Бурдонов И.Б., Косачев А.С.
Институт системного программирования РАН,
{igor,kos}@ispras.ru

Аннотация. В статье представлен подход к проблеме полноты тестирования, под которым понимается проверка соответствия реализации требованиям, описываемым спецификацией. Полнота означает, что тесты обнаруживают все возможные ошибки в реализации. На практике тестирование должно заканчиваться за конечное время. Требования полноты и конечности в общем случае противоречат друг другу. Тем не менее, конечные полные тесты удаётся построить для ограниченных классов реализаций и спецификаций при наличии специальных тестовых возможностей. Предложены алгоритмы тестирования и дана оценка их сложности для конечных спецификаций и конечных реализаций с ограниченным недетерминизмом при тестировании с открытым состоянием.

Введение

Тестирование – это проверка в процессе эксперимента соответствия (конформности) реализации требованиям, заданным в виде спецификации. Тестирование полное, если оно однозначно отвечает на вопрос: есть в реализации ошибки или нет, где под ошибкой понимается нарушение требований, то есть неконформность. Для практического применения тестирование должно заканчиваться за конечное время. В общем случае тестирование оказывается либо неполным, либо бесконечным. Решение проблемы можно искать, сужая класс рассматриваемых реализаций и используя дополнительные тестовые возможности. В некоторых случаях удаётся построить тесты, которые конечны и полны, но только для заданных классов реализаций и спецификаций при дополнительных тестовых возможностях.

Основными причинами бесконечности полного тестирования являются объём реализации и/или спецификации и недетерминизм реализации.

Если объём требований бесконечен, нельзя их все проверить за конечное время: конечное тестирование по бесконечной спецификации заведомо неполно. Если хотя бы одно из требований нужно проверять в реализации в бесконечном числе ситуаций, то за конечное время это также нельзя сделать. Для полноты конечного тестирования реализация также должна быть конечной.

Однако конечности реализации недостаточно, если её объём неизвестен: ни в какой момент времени мы не можем знать, все ли ситуации, возможные в реализации, проверены или нет. Нам нужно оценить объём реализации либо

заранее, предполагая не только его конечность, но и ограниченность, либо в процессе тестирования. В последнем случае нужны дополнительные тестовые возможности, позволяющие наблюдать в том или ином виде протестированную часть реализации и делать выводы о наличии или отсутствии других частей.

Если поведение реализации недетерминировано без всяких ограничений, ни в какой момент времени мы не можем знать, продемонстрировала ли реализация все варианты своего поведения или нет. Для полноты конечного тестирования на недетерминизм реализации приходится налагать те или иные ограничения.

В статье рассматривается тестирование *конечной реализации* по *конечной спецификации* с двумя дополнительными предположениями: 1) *тестирование с открытым состоянием* – у нас есть возможность наблюдать состояния реализации, в которых она оказывается в процессе тестирования, 2) *реализация ограниченно-недетерминирована* – если одно и то же тестовое воздействие повторяется в одном и том же состоянии реализации достаточное, известное заранее число раз, то реализация демонстрирует все возможные варианты поведения. Для этого случая мы предлагаем алгоритмы конечного и полного тестирования и даём оценки числа тестовых воздействий и объёма вычислений.

1-ый раздел статьи содержит основные положения теории конформности, изложенной в [8,10,11]. Во 2-ом разделе обсуждаются проблемы практического тестирования и пути их решения. В 3-ем разделе предлагаются алгоритмы обхода, а в 4-ом разделе – алгоритмы тестирования реализации, доказывается конечность и полнота тестирования и приводятся оценки сложности.

1. Теория конформности

1.1. Семантика взаимодействия и безопасное тестирование

Верификация конформности понимается как проверка соответствия исследуемой системы заданным требованиям. В модельном мире система отображается в реализационную модель (реализацию), требования – в спецификационную модель (спецификацию), а их соответствие – в бинарное отношение конформности. Если требования выражены в терминах взаимодействия системы с окружающим миром, возможно тестирование как проверка конформности в процессе тестовых экспериментов, когда тест подменяет собой окружение системы. Само отношение конформности и его тестирование основаны на той или иной модели взаимодействия.

Мы рассматриваем семантики взаимодействия, которые основаны только на внешнем, наблюдаемом поведении системы и не учитывают её внутреннее устройство, отображаемое на уровне модели в понятии *состояния*. Мы можем наблюдать только такое поведение реализации, которое, во-первых,

«спровоцировано» тестом (управление) и, во-вторых, наблюдаемо во внешнем взаимодействии. Такое взаимодействие может моделироваться с помощью, так называемой, машины тестирования [8,10,11,17,18,25]. Она представляет собой «чёрный ящик», внутри которого находится реализация (Рис.1). Управление сводится к тому, что оператор машины, выполняя тест (понимаемый как инструкция оператору), нажимает кнопки на клавиатуре машины, «разрешая» реализации выполнять те или иные действия, которые могут им наблюдаться. Наблюдения (на «дисплее» машины) бывают двух типов: наблюдение некоторого *действия*, разрешённого оператором и выполняемого реализацией, и наблюдение *отказа* как отсутствия каких бы то ни было действий из числа тех, что разрешены нажатыми кнопками. Мы будем обозначать действия строчными буквами, а отказы (как множества действий) – прописными.

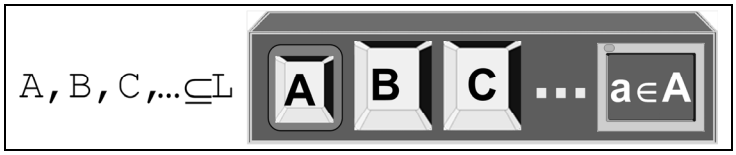

Рис.1. Машина тестирования

Подчеркнём, что при управлении оператор разрешает реализации выполнять именно множество действий, а не обязательно одно действие. Каждой кнопке соответствует своё множество разрешаемых действий. После наблюдения (действия или отказа) кнопка отжимается, и все внешние действия запрещаются. Далее оператор может нажать другую (или ту же самую) кнопку.

Тестовые возможности определяются тем, какие «кнопочные» множества есть в машине, а также, для каких кнопок возможно наблюдение отказа. Тем самым, семантика взаимодействия определяется алфавитом внешних (наблюдаемых) действий $\mathtt{L}$ и двумя наборами кнопок машины тестирования: с наблюдением соответствующих отказов – семейство $\mathbf{R}\subseteq\mathcal{P}(\mathtt{L})$ и без наблюдения отказа – семейство $\mathbf{Q}\subseteq\mathcal{P}(\mathtt{L})$. Предполагается, что $\mathbf{R}\cap\mathbf{Q}=\varnothing$ и $\cup\mathbf{R}\cup\cup\mathbf{Q}=\mathtt{L}$. Такую семантику мы называем **R**/**Q**-семантикой.

В [8,11] показано, что достаточно ограничиться семантиками, в которых все отказы наблюдаемы: $\mathbf{Q}=\varnothing$. Любая **R**/**Q**-семантика эквивалентна $\mathbf{R}\cup\mathbf{Q}/\varnothing$-семантике: для любой спецификации в **R**/**Q**-семантике существует (и может быть построена при некоторых, практически приемлемых, ограничениях) спецификация в $\mathbf{R}\cup\mathbf{Q}/\varnothing$-семантике, определяющая тот же класс конформных реализаций и не меньший класс реализаций, поддающихся тестированию. Поэтому мы будем рассматривать только **R**-семантики, то есть $\mathbf{Q}=\varnothing$.

Для выполнимости действия необходимо, чтобы оно было определено в реализации и разрешено оператором. Если этого условия также и достаточно, то в системе нет приоритетов. Мы ограничимся только такими системами.

Кроме внешних действий реализация может совершать внутренние (ненаблюдаемые) действия, обозначаемые символом τ. Эти действия всегда разрешены. Предполагается, что любая конечная последовательность любых действий совершается за конечное время, а бесконечная – за бесконечное время. Бесконечная последовательность τ-действий («зацикливание») называется *дивергенцией* и обозначается символом Δ. Кроме этого мы вводим специальное, также не регулируемое кнопками, действие, которое называем *разрушением* и обозначаем символом γ. Оно моделирует любое запрещённое при тестировании поведение реализации. Дивергенция сама по себе не опасна, но при попытке выхода из неё, когда оператор нажимает любую кнопку, он не знает, нужно ли ждать наблюдения или бесконечно долго будут выполняться только внутренние действия. Поэтому оператор не может ни продолжать тестирование, ни закончить его. Тестирование, при котором не возникает разрушения и попыток выхода из дивергенции, называется безопасным.

1.2. LTS-модель и её трассы

В качестве модели реализации и спецификации мы используем *систему помеченных переходов* (LTS – Labelled Transition System) – ориентированный граф с выделенной начальной вершиной, дуги которого помечены некоторыми символами. Формально, LTS – это совокупность $\mathbf{S}=\mathrm{LTS}(V_S, L, E_S, s_0)$, где V_S – непустое множество состояний (вершин графа), L – алфавит внешних действий, $E_S \subseteq V_S \times (L \cup \{\tau,\gamma\}) \times V_S$ – множество переходов (помеченных дуг графа), $s_0 \in V_S$ – начальное состояние (начальная вершина графа). Переход из состояния s в состояние s' по действию z обозначается $s \text{—}z\rightarrow s'$. Обозначим $s \text{—}z\rightarrow \triangleq \exists s'\ s \text{—}z\rightarrow s'$ и $s \text{—}z\not\rightarrow \triangleq \nexists s'\ s \text{—}z\rightarrow s'$. Маршрутом LTS называется последовательность смежных переходов: начало каждого перехода, кроме первого, совпадает с концом предыдущего перехода.

Выполнение LTS в машине тестирования сводится к выполнению того или иного перехода, определённого в текущем состоянии и разрешаемого нажатой кнопкой (τ- и γ-переходы всегда разрешены).

Состояние *стабильно*, если из него не выходят τ- и γ-переходы, и *дивергентно*, если в нём начинается бесконечная цепочка τ-переходов (в частности, τ-цикл). Отказ $P \in \mathbf{R}$ порождается стабильным состоянием, из которого нет переходов по действиям из P. Переход $s \text{—}z\rightarrow s'$ *разрушающий*, если из s' по цепочке (возможно пустой) τ-переходов достижимо начало γ-перехода.

Добавим в каждом стабильном состоянии LTS **S** виртуальные петли, помеченные порождаемыми отказами, и Δ-переходы из дивергентных состояний. В полученной LTS рассмотрим маршруты, начинающиеся в начальном состоянии и не продолжающиеся после Δ- и γ-переходов. **R**-трассой LTS **S** назовём последовательность пометок на переходах такого маршрута с пропуском символов τ. Множество **R**-трасс LTS **S** обозначим $\boldsymbol{T}(\mathbf{S})$.

1.3. Гипотеза о безопасности и безопасная конформность

Определим отношение безопасности «*кнопка безопасна в LTS после **R**-трассы*»: нажатие кнопки P после **R**-трассы σ не вызывает попытку выхода из дивергенции (после трассы нет дивергенции) и не вызывает разрушение (после действия, разрешаемого кнопкой). При безопасном тестировании будут нажиматься только безопасные кнопки. Формально, кнопка P безопасна:

<u>в состоянии s</u>: P ***safe*** s, если состояние s не дивергентно и в каждом состоянии s`, достижимом из s по τ-переходам, нет γ-переходов, а все переходы $s` \text{—} z \rightarrow s``$, где $z \in P$ неразрушающие;

<u>во множестве состояний S</u>: P ***safe*** $S \triangleq \forall s \in S$ P ***safe*** s;

<u>после трассы σ</u>: P ***safe*** **S** ***after*** σ $\triangleq$ P ***safe*** (**S** ***after*** σ),

где **S** ***after*** σ – множество состояний LTS **S** после трассы σ, то есть состояний, достижимых из начального состояния по трассе σ.

Безопасность кнопок влечёт безопасность действий и отказов после трассы. Отказ R ***safe*** (**S** ***after*** σ), если после трассы σ безопасна кнопка R. Действие z ***safe*** (**S** ***after*** σ), если оно разрешается $z \in P$ некоторой кнопкой P ***safe*** (**S** ***after*** σ). **R**-трасса безопасна, если 1) LTS не разрушается с самого начала, то есть в ней нет трассы $\langle\gamma\rangle$, 2) каждый символ трассы безопасен после непосредственно предшествующего ему префикса трассы. Множества безопасных трасс LTS **S** обозначим $\boldsymbol{Safe}(\mathbf{S})$.

Требование безопасности тестирования выделяет класс *безопасных* реализаций, которые могут быть безопасно протестированы для проверки их конформности заданной спецификации. Этот класс определяется следующей *гипотезой о безопасности*: реализация **I** *безопасна* для спецификации **S**, если 1) в реализации нет разрушения с самого начала, если этого нет в спецификации, 2) после общей безопасной трассы реализации и спецификации любая кнопка, безопасная в спецификации, безопасна после этой трассы в реализации:

$$\mathbf{I}\ \textit{safe for}\ \mathbf{S} \triangleq (\langle\gamma\rangle \notin \boldsymbol{T}(\mathbf{S}) \Rightarrow \langle\gamma\rangle \notin \boldsymbol{T}(\mathbf{I}))\ \&\ \forall \sigma \in \boldsymbol{Safe}(\mathbf{S}) \cap \boldsymbol{T}(\mathbf{I})\ \ \forall P \in \mathbf{R}$$
$$(P\ \textit{safe}\ \mathbf{S}\ \textit{after}\ \sigma \Rightarrow P\ \textit{safe}\ \mathbf{I}\ \textit{after}\ \sigma).$$

Гипотеза о безопасности не проверяема при тестировании и является его предусловием. Определим отношение *конформности*: реализация **I** *конформна* спецификации **S**, если она безопасна и выполнено *тестируемое условие*: любое наблюдение, возможное в реализации в ответ на нажатие безопасной (в спецификации) кнопки, разрешается спецификацией:

$$\mathbf{I}\ \textit{saco}\ \mathbf{S} \triangleq \mathbf{I}\ \textit{safe for}\ \mathbf{S}\ \&\ \forall\sigma\in \textit{Safe}(\mathbf{S})\cap \textit{T}(\mathbf{I})\ \ \forall P\ \textit{safe}\ \mathbf{S}\ \textit{after}\ \sigma\ \textit{obs}(\mathbf{I}\ \textit{after}\ \sigma, P)\subseteq \textit{obs}(\mathbf{S}\ \textit{after}\ \sigma, P),$$

где $\textit{obs}(M,P) \triangleq \{u \mid \exists m\in M\ u\in P\ \&\ m \text{—}u\!\!\rightarrow \vee\ u=P\ \&\ \forall z\in P\ m \text{—}z\!\!\not\rightarrow\}$ – множество наблюдений, которые возможны в состояниях из множества M при нажатии кнопки P.

1.4. Генерация тестов

В терминах машины тестирования тест – это инструкция оператору машины, состоящая из терминальных и нетерминальных пунктов. В каждом пункте указывается кнопка, которую оператор должен нажимать, и для каждого наблюдения, возможного после нажатия этой кнопки, – пункт инструкции, который должен выполняться следующим, или вердикт (***pass*** или ***fail***), если тестирование нужно закончить. В [8,10,11] такая инструкция соответствует формальному определению *управляемого* LTS-теста, который однозначно определяет поведение оператора (без лишнего недетерминизма).

Реализация *проходит* тест, если её тестирование всегда заканчивается с вердиктом ***pass***. Реализация проходит набор тестов, если она проходит каждый тест из набора. Набор тестов *значимый*, если каждая конформная реализация его проходит; *исчерпывающий*, если каждая неконформная реализация его не проходит; *полный*, если он значимый и исчерпывающий. Задача заключается в генерации полного набора тестов по спецификации.

Полный набор тестов всегда существует, в частности, им является набор всех *примитивных* тестов. Такой тест строится по одной выделенной безопасной **R**-трассе спецификации. Для этого в трассу вставляются кнопки, которые оператор должен нажимать: перед каждым отказом R вставляется кнопка R, а перед каждым действием z – какая-нибудь безопасная (после префикса трассы) кнопка P, разрешающая действие z. Безопасность трассы гарантирует безопасность кнопки R и наличие такой безопасной кнопки P. Выбор кнопки P может быть неоднозначным: по одной безопасной трассе спецификации можно сгенерировать, вообще говоря, несколько разных примитивных тестов. Если наблюдение, полученное после нажатия кнопки, продолжает трассу, тест продолжается. Последнее в трассе наблюдение и любое наблюдение, «ведущее в сторону», всегда заканчивают тестирование. Вердикт ***pass*** выносится, если

полученная **R**-трасса есть в спецификации, а вердикт ***fail*** – если нет. Такие вердикты соответствуют *строгим* тестам, которые, во-первых, значимые (не ловят ложных ошибок) и, во-вторых, не пропускают обнаруженных ошибок. Любой строгий тест сводится к некоторому множеству примитивных тестов, которые обнаруживают те же самые ошибки.

2. Проблемы практического тестирования

2.1. Недетерминизм и глобальное тестирование

Если приоритетов нет, в каждый момент времени реализация может выполнять любое определённое в ней и разрешённое оператором внешнее действие, а также определённые и всегда разрешённые внутренние действия. Если таких действий несколько, выбирается одно из них недетерминированным образом. Здесь предполагается, что недетерминизм – это явление того уровня абстракции, которое определяется тестовыми возможностями по наблюдению и управлению, то есть семантикой взаимодействия. Иными словами, поведение реализации зависит от неких не учитываемых нами факторов – «погодных условий», которые определяют выбор действия вполне детерминировано.

Для полноты тестирования приходится предполагать, что любые погодные условия могут быть воспроизведены в тестовом эксперименте, причём для каждого теста. Если это возможно, тестирование называется *глобальным* [25]. Мы абстрагируемся от количества вариантов погодных условий, здесь важна только потенциальная возможность проверить поведение системы при любых погодных условиях и любом поведении оператора. На практике используется только конечное число прогонов каждого теста, и без дополнительных условий мы не можем быть уверены, что провели тестовые испытания каждого теста для всех погодных условий. Возможны различные решения этой проблемы.

Одно из них – специальные тестовые возможности по «управлению погодой». Здесь мы выходим за рамки модели, которая как раз и абстрагировалась от внешних факторов, то есть от погоды. Тестирование становится зависящим не только от спецификации, но и от деталей реализации, от того, что называют операционной обстановкой, в которой работает реализация. Для каждого варианта операционной обстановки создаётся свой набор тестов. В некоторых частных случаях на этом пути можно получить практические выгоды.

Другое решение – специальные гипотезы о реализации. Они предполагают, что если реализация ведёт себя правильно при некоторых выделенных погодных условиях, то она будет вести себя правильно при любых погодных условиях [4].

Третье решение основано на знании о распределении вероятностей различных погодных условий. Тестирование полно с той или иной вероятностью [14].

Четвёртое решение предполагает, что в любой ситуации (после любой трассы) число неэквивалентных погодных условий ограничено известным числом `t`: после `t` прогонов теста гарантированно воспроизводится поведение реализации при всех возможных погодных условиях [16,24]. Это можно назвать *ограниченным недетерминизмом (`t`-недетерминизмом)*. При `t=1` все тестируемые реализации детерминированы. Это используется на практике [2,3,26], когда известно, что интересующие нас реализации детерминированы.

2.2. Бесконечность полного набора тестов

Поскольку тесты конечные, полный набор, как правило, содержит бесконечное число тестов, в частности, бесконечен набор примитивных тестов. (Полный набор конечен только для моделей с конечным поведением, то есть конечным числом трасс: конечная LTS-спецификация без циклов.) Однако на практике используются только конечные наборы конечных тестов. Возможны различные решения этой проблемы. Все они сводятся к специальным тестовым возможностям и/или реализационным гипотезам. Такие гипотезы, по сути, предполагают, что если реализация ведёт себя правильно на тестах данного конечного набора, то она ведёт себя правильно на всех тестах полного набора. Обоснованием может служить каким-то образом полученное знание об устройстве реализаций из рассматриваемого подкласса. Такие конечные наборы тестов на классе всех реализаций только значимы (не ловят ложных ошибок), но зато полны на подклассе, определяемом реализационной гипотезой.

Теория конформности предполагает общий критерий покрытия: проверка всех требований спецификации во всех возможных реализационных ситуациях. Часто используются специальные критерии покрытия, чтобы проверить не все ситуации, а только некоторые интересующие нас классы ситуаций (основанные на модели возможных ошибок) [19,20,27]. Достаточно общий подход сводится к тому, что вместо исходной спецификации используется более грубая, так называемая, тестовая модель, которую получают факторизацией спецификации по эквивалентности переходов, что обычно сводится к эквивалентности состояний и/или действий [1]. Иногда при факторизации исчезает недетерминизм, что заодно решает и эту проблему. Этот подход оправдан, если достаточно мотивирована гипотеза о том, что ошибки, возможные в реализации, обнаруживаются при тестировании по факторизованной спецификации (по конечному набору, удовлетворяющему критерию покрытия).

Примером служит тестирование конечного автомата по спецификации, заданной также в виде конечного автомата [2,3,4,12,13,21,22,23]. Проверяемая конформность – это эквивалентность автоматов. Обычно считается, что спецификация и реализация детерминированы, и в реализации не больше

состояний, чем в спецификации (с точностью до эквивалентности состояний), или не больше, чем на заданную величину.

2.3. Тестирование с открытым состоянием

Если имеется операция, позволяющая достоверно опросить текущее состояние реализации (*status message*), то говорят о тестировании с открытым состоянием. Как известно, для проверки эквивалентности конечных автоматов с открытым состоянием полное тестирование сводится к обходу графа переходов автомата с опросом каждого проходимого состояния [2,4-7,15,22].

Для LTS-модели опрос состояния можно понимать в терминах машины тестирования как нажатие специальной кнопки после каждого наблюдения. Это эквивалентно тому, что постсостояние после наблюдения появляется на дисплее машины как часть наблюдения. Важно подчеркнуть, что, если после выполнения действия реализация оказывается в нестабильном состоянии i, то постсостояние на дисплее не обязательно равно i, а может быть любым состоянием, достижимым из i по пустой трассе ϵ, то есть по цепочке τ-переходов. Состояние $i \in (\mathbf{I}\ \textit{after}\ \epsilon)$ будем называть начально-достижимым. Тем самым, мы не требуем остановки внутренней работы реализации после наблюдения до опроса состояния. Точно также мы не требуем остановки после опроса состояния до следующего тестового воздействия или завершения тестирования. Несколько опросов состояний подряд позволяют наблюдать движение реализации по τ-переходам, но мы думаем, что такая возможность не увеличивает мощности тестирования конформности ***saco***.

Старт (или рестарт) системы понимается как одно из тестовых воздействий, входящих в **R**. Отличие в том, что после старта (рестарта) гарантированно получается начально достижимое состояние. В общем случае рестарт определён не в каждом состоянии реализации. Будем считать, что если рестарт определён, то наблюдается действие «рестарт», иначе – отказ по рестарту. LTS реализации пополняется переходами по рестарту, которые «сбрасывают» трассу: трасса маршрута – это трасса его постфикса после последнего рестарта.

2.4. Условия конечности тестирования

Сформулируем ограничения на реализации и спецификации, делающие возможным конечное полное тестирование и рассматриваемые в статье. Мы предполагаем тестирование с открытым состоянием конформности ***saco*** и общий критерий покрытия. Для каждого ограничения укажем его ослабленные варианты, которые для простоты изложения в дальнейшем не рассматриваются.

t-недетерминизм реализации. Мы предполагаем, что в любом состоянии i реализации после t нажатий любой кнопки будут получены все возможные пары (наблюдение, постсостояние), то есть будут пройдены все маршруты, начинающиеся в i и имеющие трассу ⟨наблюдение⟩. Ограничение можно ослабить, рассматривая только достижимые состояния реализации. Для данной спецификации важны только те состоянии реализации, которые достижимы по безопасным трассам спецификации. Из t-недетерминизма следует, что для любой кнопки (с любым кнопочным множеством) в каждом состоянии определено не более t переходов по действиям, разрешаемым кнопкой.

Для решения проблемы бесконечности полного тестового набора наложим ограничения на размеры **R**-семантики, спецификации и реализации.

Число кнопок конечно. Без этого ограничения нам придётся, вообще говоря, выполнять бесконечное число тестовых воздействий после каждой пройденной трассы. Для данной спецификации ограничение ослабляется, если учитывать только кнопки, безопасные после трассы в спецификации.

Есть алгоритм разрешения кнопки относительно всех действий, который для каждого действия z и каждой кнопки P за конечное время определяет $z \in P$ или $z \notin P$. Это нужно для проверки безопасности кнопки в спецификации после пройденной трассы: кнопка безопасна, если нет разрушающих переходов из состояний после трассы по действиям из кнопки. Алгоритм легко строится, если все кнопочные множества конечны (тогда конечен и алфавит действий).

Число состояний спецификации конечно. При бесконечным числе состояний бесконечно полное тестирование, например, самой спецификации. Ограничение можно ослабить, если считать только состояния после безопасных трасс. Для данной реализации нужны не все такие состояния спецификации, однако мы предполагаем, что спецификация используется для тестирования любой реализации (с учётом ограничений на недетерминизм и размер реализации).

Число переходов спецификации конечно. Это требуется для проверки безопасности кнопки после пройденной трассы. Нужно перебрать все переходы, ведущие из состояний после трассы, и, если переход разрушающий, проверить, принадлежит ли действие, которым переход помечен, данной кнопке. Проверка выполняется за конечное время в двух случаях: 1) число переходов после трассы конечно; 2) кнопка конечна, множество переходов перечислимо, а спецификация задана так, что переходы по одному действию перечисляются подряд. Поскольку число состояний конечно, конечно число переходов по данному действию. В процессе перебора переходов отмечаем те действия, переходы по которым неразрушающие и которые принадлежат кнопке, до тех пор, пока не обнаружим разрушающий переход по действию из

кнопки или пока не отметим все действия из кнопки. Если число переходов бесконечно, а спецификация задана иным образом, конечное время проверки не гарантируется. То же самое имеет место, если множество переходов и кнопка оба бесконечны. Ограничение можно ослабить, если учитывать только те переходы, которые продолжают маршруты с безопасными трассами.

Число состояний реализации конечно. В противном случае не гарантируется, что ошибка будет обнаружена за конечное время, а при отсутствии ошибок тестирование гарантированно бесконечное. Это ограничение можно ослабить, если учитывать только те состояния реализации, которые достижимы по безопасным трассам спецификации. При нарушении ослабленного ограничения конформные реализации будут всегда тестироваться бесконечно долго.

Сильно-связность реализации. Для полноты тестирования нужно проверить все переходы реализации, лежащие на маршрутах с трассами, безопасными в спецификации. При тестировании LTS, порождаемой такими переходами, выполняется её обход: проходится маршрут, содержащий все достижимые переходы. Для существования обхода LTS должна быть сильно-связной: из каждого состояния достижимо по переходам каждое другое состояние. Более точно, LTS должна представлять собой цепочку сильно-связных компонентов, и из каждого непоследнего компонента должен вести ровно один переход в следующий компонент [2]. Однако, без дополнительных условий, обход алгоритмизуем, только если каждый компонент в цепочке, кроме последнего, – это одно состояние без петель. LTS всегда сильно-связна, если рестарт определён в каждом состоянии. Далее будем считать, что LTS сильно-связна.

3. Алгоритм обхода LTS

Сначала рассмотрим задачу обхода LTS-реализации без достижимых разрушения и дивергенции, для которой выполнены ограничения: LTS конечна, сильно-связна и t-недетерминирована, а число кнопок конечно.

3.1. LTS обхода

Обозначим: **I** – LTS-реализация. При обходе будем строить LTS, которую назовём пройденной LTS и обозначим **G**. Переход i—z→i` по внешнему действию z добавляется, когда после нажатия кнопки P в состоянии i наблюдается действие z∈P и постсостояние i`. Это означает, что в **I** имеется маршрут, начинающийся в i и заканчивающийся в i`, и имеющий трассу ⟨z⟩, то есть содержащий один переход по z, и τ-переходы. Если наблюдается отказ с *тем же самым* постсостоянием i`=i, то переходы не добавляются. Если отказ P наблюдается с *другим* постсостоянием i`≠i, то

добавляется переход i—τ→i`. Это означает, что в **I** имеется τ-маршрут, начинающийся в i и заканчивающийся в i`, причём состояние i` стабильно, и в нём есть отказ P. Вместе с переходом будем хранить кнопку, нажатие которой вызвало этот переход (любую из таких кнопок).

Обход заканчивается, когда в **G** нельзя добавить ни одного нового перехода. Легко показать, что после этого LTS **I** и **G** имеют одно и то же множество достижимых состояний и одно и то же множество **R**-трасс. Любой алгоритм, выполняющий обход LTS **G**, гарантирует обход LTS **I**.

Для каждой кнопки P и каждого пройденного состояния i определим счётчик c(P,i) числа нажатий кнопки P в состоянии i. Будем называть кнопку P *полной в состоянии* i, если 1) c(P,i)=1, в **G** нет τ-переходов из i и нет перехода i—z→i`, где z∈P, или 2) c(P,i)=t. Это означает, что уже получены все возможные переходы из состояния i при нажатии кнопки P. В случае 1 в состоянии i после нажатия кнопки P наблюдался отказ P в этом же состоянии, повторное нажатие кнопки P даст тот же отказ. В случае 2 после t нажатий кнопки получены все возможные переходы. Состояние i будем называть *полным*, если каждая кнопка полна в нём. Это означает, что все возможные переходы в **G** из состояния i уже получены.

Вначале в **G** только одно состояние i∈{**I** ***after*** ϵ} и ∀P∈**R** c(P,i)=0.

3.2. Общая схема алгоритма обхода

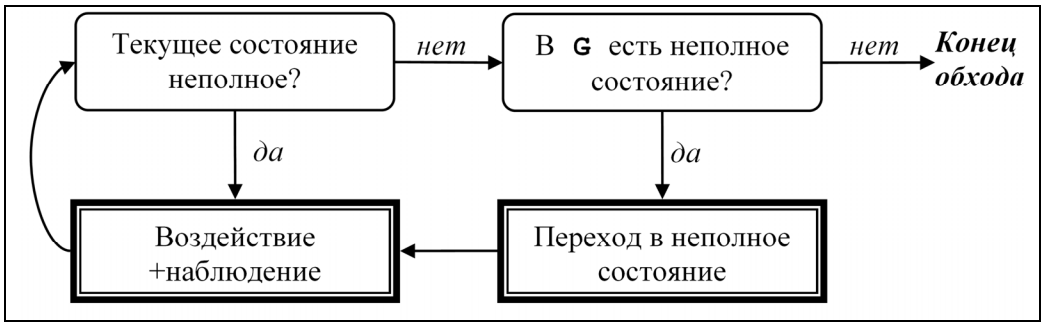

Рис.2. Общая схема алгоритма обхода

На Рис.2 изображена общая схема алгоритма. Если текущее состояние i неполное, то некоторая кнопка P неполна в i. В этом случае нажимаем эту кнопку и получаем наблюдение o (действие или отказ) и постсостояние i`, которое становится новым текущим состоянием. Увеличиваем счётчик c(P,i):=c(P,i)+1, и в **G** добавляем переход i—o→i`, если o действие, или переход i—τ→i`, если o отказ и i`≠i.

Если текущее состояние полное, то в нём все тестовые воздействия выполнены нужное число раз и получены все возможные наблюдения и постсостояния. Для продолжения обхода нужно перейти в любое неполное состояние. Если таких состояний нет, алгоритм заканчивается, поскольку все пройденные состояния полные и, следовательно, все достижимые переходы пройдены.

Рассмотрим переход в неполное состояние. В графе LTS **G** всегда существует лес деревьев, покрывающих все состояния и ориентированных к своим корням, которыми являются все неполные состояния. Выберем любой такой лес и для каждого его перехода i—○→i` через P(i) обозначим связанную с ним кнопку. Будем двигаться, нажимая в каждом текущем состоянии i кнопку P(i). Из-за недетерминизма мы можем оказаться не в i`, а в другом состоянии i``, где будем нажимать кнопку P(i``).

Покажем, что неполное состояние достигается за конечное число шагов. Будем вести доказательство от противного: пусть мы совершаем бесконечное число шагов, оказываясь только в полных состояниях. Длину пути по лесу деревьев от состояния i до неполного состояния назовём расстоянием r_i. Поскольку число состояний конечно, в каких-то полных состояниях мы окажемся бесконечное число раз. Выберем из них состояние i с минимальным расстоянием r_i. Поскольку мы бесконечное число раз выходим по тем или иным переходам из состояния i, нажимая одну и ту же кнопку P(i), то в силу ограниченности недетерминизма мы должны были бесконечное число выходить из i по переходу леса i—○→i`. Тогда в состоянии i` мы были тоже бесконечное число раз, но $r_{i`}=r_i-1$, что противоречит минимальности расстояния r_i. Мы пришли к противоречию, что и требовалось.

3.3. Оценка числа тестовых воздействий

На Рис.5 тестовые воздействия выполняются в блоках в жирной рамке. Пусть n – число состояний, b – число кнопок (с учётом рестарта, если он определён хотя бы в одном состоянии), t – ограничение недетерминизма.

Блок «Воздействие+наблюдение». Тестовое воздействие в каждом состоянии осуществляется не более bt раз (меньше, если наблюдаются отказы с тем же постсостоянием). Суммарно имеем не более btn тестовых воздействий.

Блок «Переход в неполное состояние». Оценим однократную работу блока при числе полных состояний c. Рассмотрим все полные состояния, в которых мы бываем. Если состояние имеет расстояние r=1, то в силу t-недетерминизма мы выходим из этого состояния не более t раз. Поэтому из состояния с

расстоянием $r=2$ мы выходим не более t^2 раз. В общем, из состояния с расстоянием r мы выходим не более t^r раз. Пусть a_r число состояний с расстоянием r. Тогда число тестовых воздействий не больше, чем
$\boldsymbol{f}(c)=\sum_r(a_r t^r)\le t+t^2+\ldots+t^c=(t^{c+1}-t)/(t-1)$ при $t>1$, или c при $t=1$.

Теперь оценим суммарную работу блока. Перенумеруем все состояния в том порядке, в котором они становились полными. Пусть в момент времени, непосредственно предшествующий получению в i-ом состоянии j-ой тройки (кнопка, наблюдение, постсостояние), имеется c_{ij} полных состояний. Если перед получением j-ой тройки мы переходили в неполное состояние i, то через y_{ij} обозначим число тестовых воздействий во время такого перехода; иначе $y_{ij}=0$. В силу доказанного $y_{ij}\le\boldsymbol{f}(c_{ij})$. Поскольку $c_{ij}\le i-1$, а функция $\boldsymbol{f}$ монотонно возрастает, $\boldsymbol{f}(c_{ij})\le\boldsymbol{f}(i-1)$. Число троек в каждом состоянии не превосходит bt. Общая оценка числа тестовых воздействий в блоке равна:
$\sum_{ij}(y_{ij}) \le \sum_{ij}(\boldsymbol{f}(c_{ij})) \le \sum_{ij}(\boldsymbol{f}(i-1)) \le \sum_i(bt\boldsymbol{f}(i-1)) =$
$= bt\sum_i((t^i-t)/(t-1)) = O(bt^n)$, если $t>1$,
$= b\sum_i(i-1) = O(bn^2)$, если $t=1$.

Эта оценка достигается по порядку на LTS, изображённой на Рис.3. Здесь имеется только одно действие и одна кнопка, разрешающая его. Если в процессе движения по LTS каждый переход, выделенный пунктиром, из-за недетерминизма предваряется $t-1$ переходами в начальное состояние (жирные стрелки), нижняя оценка равна $O(t^n)$. Заметим, что эта оценка достигается при любом алгоритме обхода, а не только описанном выше.

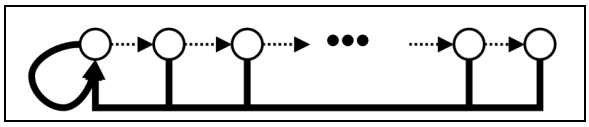

Рис.3. Пример LTS

3.4. Оценка объёма вычислений

Будем считать, что опрос состояния даёт его уникальный идентификатор, но мы не знаем устройства этого идентификатора, а можем только сравнивать два идентификатора на равенство. При опросе состояния нужно определить, является ли оно новым или старым, и, если старым, то каким именно. Это требует $O(n)$ операций сравнения. Поэтому объём вычислений по порядку не более, чем в n раз превосходит длину обхода: $O(nbt^n)$ при $t>1$ и $O(bn^3)$ при $t=1$. Убедимся, что при подходящей реализации алгоритма другие вычисления не изменяют этот порядок.

Заметим, что лес деревьев, требуемый для прохода в неполное состояние, может использоваться много раз без его перестройки, пока какое-то неполное состояние не станет полным. Будем перестраивать лес только в этом случае.

Определим основные структуры данных. Перенумеруем кнопки от `1` до `b`, и все пройденные состояния. По номеру состояния `i` можно определить:

`I(i)` – идентификатор состояния;

`B(i)` – номер текущей кнопки, вначале `B(i)=1`, если `i` полное, то `B(i)=b+1`;

`C(i)` – счётчик текущей кнопки, вначале `C(i)=0`;

`T(i)` – список различных номеров состояний, из которых выходит переход, заканчивающийся в `i`, вначале список `T(i)` пуст;

`P(i)` – номер кнопки в лесе деревьев, вначале `P(i)=0`.

Отдельно хранится номер текущего состояния i_c, счётчик `N` пройденных состояний (вначале `N=0`), счётчик полных состояний `C` (вначале `C=0`) и матрица `M` размером `N×N`, где `M(i,j)` содержит номер кнопки, связанной с каким-нибудь переходом `I(i)`—o→`I(j)`, или `0`, если такой кнопки нет.

Блок «Текущее состояние неполное?» проверяет, что $B(i_c) \le b$. Число сравнений равно числу переходов, которое не больше `O(btn)`.

Блок «Воздействие+наблюдение». В состоянии i_c нажимаем кнопку `B(i)` и получаем наблюдение `o` и постсостояние. Определяем номер постсостояния `i`` поиском по массиву состояний за `O(n)` операций. Если `i`` новое состояние, ему присваивается номер `N+1`, создаётся его описание, расширяется матрица `M`, добавлением `N+1`-ых нулевых строки и столбца, `N:=N+1`. Далее $C(i_c) := C(i_c)+1$. Если наблюдается отказ и $i` = i_c$, или $C(i_c)=t$, выбираем следующую кнопку $B(i_c) := B(i_c)+1$, $C(i_c) := 0$. Если $M(i_c, i`)=0$, то состояние i_c добавляется в список `T(i`)`, а кнопка `B(i)` заносится в матрицу $M(i_c, i`) := P$. Меняется текущее состояние $i_c := i`$. Блок работает не более `btn` раз, и суммарная оценка $O(btn^2)$.

Когда состояние i_c становится полным ($B(i_c)=b+1$), перестраиваем лес деревьев для `C:=C+1`. Создаём рабочий список `L` номеров листовых состояний леса деревьев. Сначала за один просмотр всех пройденных состояний обнуляем номера кнопок `P(i):=0` и помещаем номера неполных состояний в список `L` (`O(n)` операций). Далее для элемента `i` из головы списка `L` просматриваем список `T(i)` и для каждого состояния `j` из списка проверяем, принадлежит ли оно лесу деревьев, или ещё нет. Если не принадлежит (`P(j)=0 & B(j)=b+1`), то состояние `j` помещается в конец списка `L`, а `P(j):=M(j,i)`. Состояние `i` удаляется из `L`. Построение

заканчивается, когда список L становится пуст. Число проверяемых переходов не превышает O(btn), поэтому лес строится за O(n)+O(btn)=O(btn) операций. Лес строится, когда неполное состояние становится полным (и остаётся таким до конца), то есть не более n раз, что даёт суммарную оценку сложности построения всех лесов деревьев $O(btn^2)$.

Суммарная оценка для блока «Воздействие+наблюдение» $O(btn^2)$.

Блок «В **G** есть неполное состояние?» проверяет C≠N. Суммарно: O(btn).

Блок «Переход в неполное состояние». Для выполнения одного перехода нажимается кнопка $P(i_c)$ и опрашивается постсостояние. Определяем номер постсостояния i` за O(n) операций, меняем текущее состояние $i_c$:=i` и проверяем его полноту. Если оно неполное ($B(i_c)<b+1$), действия повторяются. Учитывая число тестовых воздействий в этом блоке, суммарно имеем оценку объёма вычислений: $O(nbt^n)$ для t>1, $O(bn^3)$ для t=1.

Поскольку для t>1 имеем $n \le t^{n-1}$, оценка объёма вычислений алгоритма:
$O(btn)+O(btn^2)+O(btn)+O(nbt^n)=O(nbt^n)$ для t>1,
$O(btn)+O(btn^2)+O(btn)+O(bn^3)=O(bn^3)$ для t=1.

3.5. Сильно-Δ-связные LTS

Описанный алгоритм обхода применим к любой конечной сильно-связной t-недетерминированной LTS. При t=1 имеем детерминированный случай с оценкой длины обхода $O(bn^2)$ и объёма вычислений $O(bn^3)$. Однако при t>1 обе оценки экспоненциальны, что вряд ли приемлемо на практике. Эти оценки можно улучшить, если ограничиться некоторыми подклассами LTS. Одним из таких подклассов является класс сильно-Δ-связных LTS.

Сильно-Δ-связность введена в [4] для решения задачи обхода графа переходов недетерминированного автомата. Тестовое воздействие – это посылка стимула в автомат, а наблюдение – получение реакции от него. В автомате нет непомеченных переходов (τ-переходов в LTS). Переформулируем основные определения и утверждения работы [4] в терминах LTS и **R**-семантики.

Δ-переходом (s,P) назовём множество переходов s—z→s`, где z∈P∈**R**. [u,V]-Δ-маршрут – это такое множество D маршрутов, что u – их общее начало, V – множество их концов, и для любого префикса любого маршрута из D, заканчивающегося в состоянии s, найдётся такая кнопка P, что множество переходов, которыми этот префикс продолжается в D, совпадает с Δ-переходом (s,P). Если V={v}, то это [u,v]-Δ-маршрут. Длина Δ-

маршрута – это максимальная длина его маршрутов. Маршрут назовём обходом по кнопкам, если он содержит хотя бы один переход из каждого непустого Δ-перехода. Δ-обходом назовём Δ-маршрут, все маршруты которого – обходы по кнопкам. Алгоритм Δ-обхода при любом недетерминированном поведении LTS выполняет обход по кнопкам; в сумме все эти обходы образуют Δ-обход. Δ-путём назовём Δ-маршрут, все маршруты которого – пути. Если существует `[u,v]`-Δ-путь, `v` Δ-достижимо из `u`. LTS сильно-Δ-связна, если из любого состояния Δ-достижимо любое состояние. В сильно-Δ-связной LTS нет τ-переходов, так как нестабильное состояние не может быть Δ-достижимым. Заметим, что Δ-достижимость рассматривается с учётом рестартов.

В работе [4] предложен (в терминах LTS) алгоритм Δ-обхода сильно-Δ-связных LTS с точной верхней оценкой длины обхода $O(nm)$ и объёма вычислений $O(n^2m)$, где n – число состояний, m – число Δ-переходов. Алгоритм основан на локальной аппроксимации Δ-расстояния состояния `u` до множества состояний `V`, где Δ-расстояние – минимальная длина `[u,V]`-Δ-маршрута по пройденной LTS. В качестве `V` выбиралось множество состояний, в которых нажимались ещё не все кнопки. При `t`-недетерминизме для обхода (а не Δ-обхода) в качестве `V` выбирается множество неполных состояний. Поскольку в каждом Δ-переходе $O(t)$ переходов, а $m \le bn$, оценки становятся: $O(btn^2)$ для числа тестовых воздействий и $O(btn^3)$ для объёма вычислений.

4. Алгоритм тестирования

Прежде всего, отметим, что, если мы уверены в том, что в тестируемой реализации разрушение и дивергенция недостижимы, в частности, если их нет в спецификации, то тестирование сводится к обходу реализации и последующей аналитической верификации пройденной LTS. В общем случае алгоритмы обхода модифицируются, а верификация выполняется параллельно с обходом.

4.1. Безопасно-достижимая LTS

Обозначим: **`I`** – LTS-реализация, **`S`** – LTS-спецификация. При тестировании будем совершать обход безопасно-достижимой LTS **`G`**, переходы которой порождаются нажатиями не всех кнопок в состоянии `i`, а только тех, которые безопасны после трассы, безопасной в спецификации и заканчивающейся в реализации в состоянии `i`. Если предполагается сильно-Δ-связность, то этим свойством должна обладать не вся реализация, а её безопасно-достижимая LTS.

По-прежнему `c(P,i)` – число нажатий кнопки `P` в состоянии `i` (п.3.1). Также для состояния `i` хранится множество **`S`**`(i)`, элементами которого становятся множества `S=(`**`S`** ***after*** σ`)`, где σ∈***Safe***`(`**`S`**`)` и `i∈(`**`I`** ***after*** σ`)`.

Множество S помещается в $\mathbf{S}(i)$, когда получена трасса σ, а реализация оказалась в состоянии i. Кнопка P *допустима в* i, если она безопасна хотя бы в одном множестве $S \in \mathbf{S}(i)$. Только допустимые кнопки будут нажиматься в состоянии i. Определение полного состояния меняется: требуется, чтобы не каждая, а каждая допустимая в состоянии кнопка была в нём полна.

Реализация $\mathbf{I}$ и полностью построенная LTS $\mathbf{G}$ имеют одно и то же множество безопасно-тестируемых трасс вида $\sigma \cdot \langle o \rangle$, где $\sigma \in \mathit{Safe}(\mathbf{S}) \cap \mathit{T}(\mathbf{I})$, а наблюдение o разрешается некоторой кнопкой P ***safe*** $\mathbf{S}$ ***after*** σ. Поэтому $\mathbf{I}$ ***saco*** $\mathbf{S} \Leftrightarrow \mathbf{G}$ ***saco*** $\mathbf{S}$. Также LTS $\mathbf{I}$ и $\mathbf{G}$ имеют одно и то же множество состояний, достижимых по безопасно-тестируемым трассам.

4.2. Начало работы алгоритма

В начале тестирования в $\mathbf{G}$ есть только одно состояние $i \in (\mathbf{I}$ ***after*** $\epsilon)$ и $\mathbf{S}(i) = \{\mathbf{S}$ ***after*** $\epsilon\}$. Допустимы все кнопки P ***safe*** $\mathbf{S}$ ***after*** ϵ и $c(P,i) := 0$.

Если начальное состояние реализации i_0 нестабильно, нам нужно в процессе тестирования получить все состояния из множества $\mathbf{I}$ ***after*** ϵ. Для этого необходимо и достаточно, чтобы рестарт был определён хотя бы в одном состоянии реализации, достижимом по безопасной трассе спецификации.

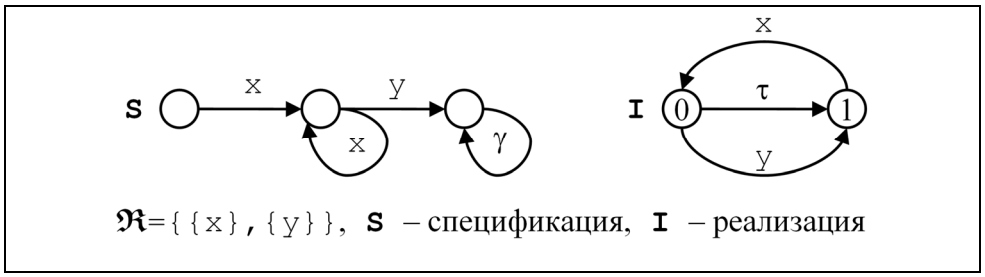

Рис.4. Пример спецификации и неконформной реализации

Если это условие не выполнено, а состояние i_0 нестабильно, то сильно-связность гарантирует обход, но не полноту тестирования. Пример приведён на Рис.4. Здесь $\mathbf{I}$ ~~***saco***~~ $\mathbf{S}$: в $\mathbf{I}$ с самого начала есть действие y, что запрещено спецификацией (должен быть отказ $\{y\}$). Если в начале тестирования мы оказываемся в состоянии 1, то без рестарта в состояние 0 мы попадём только после нажатия кнопки $\{x\}$. По спецификации после трассы, содержащей x, кнопка $\{y\}$ опасна, её нельзя нажимать. Тем самым ошибка не обнаружится.

4.3. Общая схема алгоритма

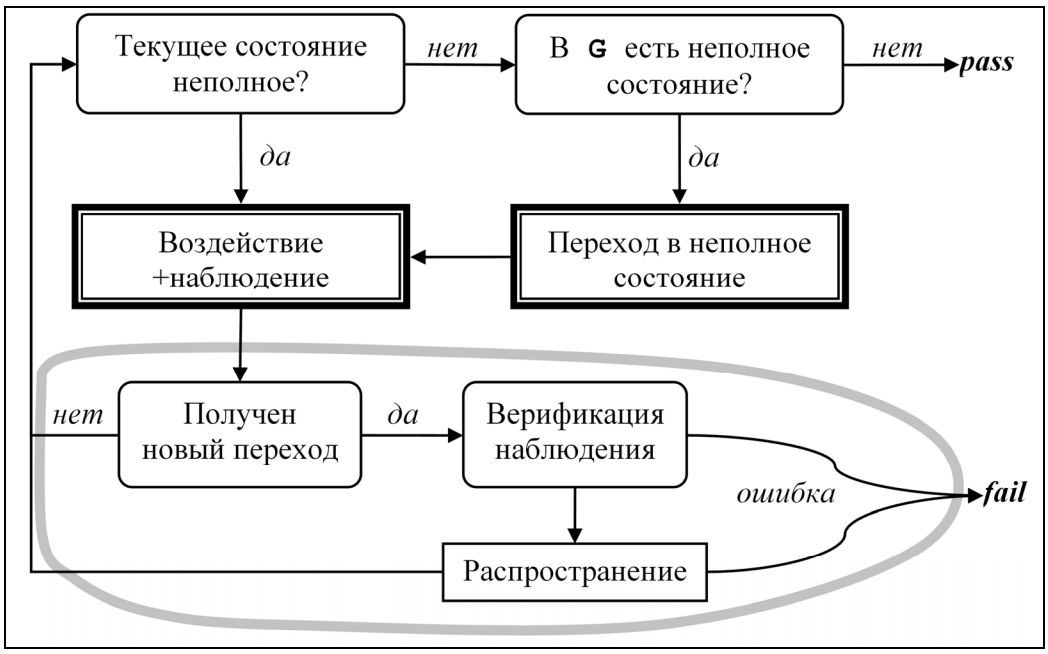

Рис.5. Общая схема алгоритма

На Рис.5 изображена общая схема алгоритма. В отличие от схемы обхода на Рис.2 здесь добавляются три блока, обведённые широкой серой линией. Кроме того, в блоке «Воздействие+наблюдение» нажимается не любая неполная кнопка, а только допустимая. Это гарантирует прохождение только трасс, безопасных в спецификации, поскольку в блоке «Переход в неполное состояние» нажимаются только те кнопки, которые уже раньше нажимались в блоке «Воздействие+наблюдение», то есть тоже допустимые.

Определим процедуру ***Post***`(i—○→i`,S)`, совмещающую верификацию наблюдения (оракул) и вычисление множества постсостояний спецификации. Здесь `i—○→i`` – это переход LTS **G**, а `S∈`**S**`(i)`. Если фиксируется ошибка (неконформность), тестирование заканчивается с вердиктом ***fail***. Иначе вычисленное множество постсостояний `S`` добавляется во множество **S**`(i`)` при условии, что `S`≠∅` и `S`∉`**S**`(i`)`, и помечается во множестве **S**`(i`)` специальным флагом «добавленное». Возможны три случая:

1) `○` – внешнее действие. Если есть кнопка `P` ***safe*** `S` такая, что `○∈P`, то `S`` состоит из всех состояний `s`, достижимых в **S** по τ-переходам из концов переходов вида `s—z→s`, где `s∈S`. Если `S`=∅`, то `○∉`***obs***`(S,P)`, и фиксируется ошибка. Если такой кнопки `P` нет, то ничего не делается.
2) `○=P` – отказ (виртуальная петля). Если `P` ***safe*** `S`, то `S`` состоит из всех состояний `s`∈S`, в которых есть отказ `P`. Если `S`=∅`, то `○∉`***obs***`(S,P)`, и фиксируется ошибка. Если `P` ~~***safe***~~ `S`, то ничего не делается.
3) `○=τ` – внутреннее действие. Вычисляется `S`=S`, ошибка не фиксируется.

Блок «Верификация наблюдения» работает при добавлении *нового* (возможно, виртуального) перехода $i—o→i`$. Если $o$ рестарт, множество $\mathbf{S}\ \boldsymbol{after}\ \epsilon$ добавляем к $\mathbf{S}(i`)$ и, если его там не было, помечаем в $\mathbf{S}(i`)$ флагом «добавленное». Если $o$ отказ по рестарту, ничего не делаем. Иначе вызываем процедуру $\boldsymbol{Post}(i—o→i`,S)$ для каждого $S∈\mathbf{S}(i)$. Если ошибок нет, выполняется блок «Распространение».

В блоке «Распространение» для каждого уже имеющегося перехода $i—o→i`$, где $o$ отлично от рестарта и отказа по рестарту, и *добавленного* множества состояний $S∈\mathbf{S}(i)$ вызываем процедуру $\boldsymbol{Post}(i—o→i`,S)$, затем снимаем флаг «добавленное» с множества S в $\mathbf{S}(i)$. Эти действия повторяются, пока есть добавленные множества. Поскольку множества состояний реализации и спецификации конечны (тем самым, конечно число множеств состояний спецификации), блок «Распространение» закончится за конечное время.

Важно отметить, что верифицируются не только наблюдения, полученные после *реальных* трасс, пройденных при тестировании, но и возможные наблюдения после *потенциальных* трасс. Более точно: верифицируются наблюдения, про которые установлено, что они возможны в реализации после известных трасс, безопасных в спецификации. Это даёт существенную экономию числа тестовых воздействий, необходимых для проверки конформности: мы выполняем множество проверок без реального тестирования, основываясь на полученном знании о поведении реализации.

Докажем, что тест, работающий по описанному алгоритму, является полным, то есть значимым и исперпывающим.

Сначала покажем, что для каждого состояния i LTS $\mathbf{G}$ каждое множество $S∈\mathbf{S}(i)$ является концом некоторой трассы, безопасной в спецификации: $∃σ∈\boldsymbol{Safe}(\mathbf{S})∩\boldsymbol{T}(\mathbf{I})\ \ S=(\mathbf{S}\ \boldsymbol{after}\ σ)\ \ \&\ \ i∈(\mathbf{I}\ \boldsymbol{after}\ σ)$. Будем вести доказательство индукцией по последовательности добавлений $\mathbf{S}(i):=\mathbf{S}(i)∪\{S\}$, где $S∉\mathbf{S}(i)$, для любого S и любого i. Такое добавление делается: 1) в начале работы алгоритма, 2) в блоке «Верификация наблюдения» для рестарта, 3) в процедуре ***Post***. В случаях 1 и 2 утверждение верно для пустой трассы $σ=\epsilon$. Процедура $\boldsymbol{Post}(i—o→i`,S)$ добавляет в $\mathbf{S}(i`)$ множество постсостояний $S`$ для перехода $i—o→i`$ таким образом, что, если $∃σ∈\boldsymbol{Safe}(\mathbf{S})∩\boldsymbol{T}(\mathbf{I})\ \ S=(\mathbf{S}\ \boldsymbol{after}\ σ)\ \ \&\ \ i∈(\mathbf{I}\ \boldsymbol{after}\ σ)$, то для $o≠τ$ имеем $o\ \boldsymbol{safe}\ (\mathbf{S}\ \boldsymbol{after}\ σ)\ \ \&\ \ S`=(\mathbf{S}\ \boldsymbol{after}\ σ·⟨o⟩)\ \ \&$

i`∈(**I** ***after*** σ·⟨o⟩), а для o=τ имеем S`=S & i`∈(**I** ***after*** σ). Утверждение доказано.

Значимость: если тест обнаруживает ошибку, реализация неконформна. С учётом доказанного утверждения процедура ***Post*** выносит вердикт ***fail***, когда обнаруживает, что после трассы σ∈***Safe***(**S**)∩***T***(**I**) кнопка P ***safe*** **S** ***after*** σ разрешает наблюдение o∈***obs***(**I** ***after*** σ,P), отсутствующее в спецификации o∉***obs***(**S** ***after*** σ,P), то есть когда реализация неконформна.

Исчерпываемость: если тест выносит вердикт ***pass***, то реализация конформна. Достаточно показать, что в конце теста для каждой трассы σ∈***Safe***(**S**)∩***T***(**I**) каждое состояние i∈(**I** ***after*** σ) имеется в LTS **G** и (**S** ***after*** σ)∈**S**(i). Будем вести доказательство индукцией по трассам. Для пустой трассы ϵ утверждение верно, поскольку в начале тестирования или после рестартов мы добавляем **S** ***after*** ϵ в **S**(i) для каждого состояния i∈(**I** ***after*** ϵ).

Рассмотрим непустую трассу σ·⟨o⟩∈***Safe***(**S**)∩***T***(**I**), где o отлично от рестарта и отказа по рестарту и разрешается некоторой кнопкой P ***safe*** **S** ***after*** σ. Пусть утверждение верно для трассы σ, и докажем его для трассы σ·⟨o⟩. Пусть состояние i`∈(**I** ***after*** σ·⟨o⟩). Тогда в реализации есть состояние i∈(**I** ***after*** σ) и переход i—o→i` (если o отказ, то это виртуальная петля). По предположению шага индукции, состояние i имеется в **G** и (**S** ***after*** σ)∈**S**(i). Поскольку при вердикте ***pass*** все состояния полны, мы должны были в состоянии i после некоторого нажатия кнопки P получить переход i—o→i` и выполнить процедуру ***Post***(i—o→i`,S). А тогда (**S** ***after*** σ·⟨o⟩)∈**S**(i`), что и требовалось доказать.

4.4. Оценка числа тестовых воздействий

В отличие от обхода, при тестировании нажимаются только допустимые кнопки. Допустимость кнопки зависит от трассы. Может случиться так, что состояние i∈(**I** ***after*** σ) полное, а кнопка P ***safe*** **S** ***after*** σ недопустима в i, так как не получена (реально или потенциально) трасса σ. После получения трассы σ, состояние i становится снова неполным.

Теперь нельзя считать, что состояние, ставшее полным, таким и остаётся. При оценке суммарной работы блока «Переход в неполное состояние» нельзя считать монотонно неубывающим число полных состояний, от которого

зависит оценка числа тестовых воздействий при однократной работы блока. Дадим оценку на основе числа вызовов блока. После прохода в неполное состояние мы нажимаем неполную кнопку. Поэтому блок вызывается не более btn раз, и каждый раз выполняется не более $f(n-1)$ тестовых воздействий. Это даёт оценку $O(bnt^n)$ для $t>1$, и $O(bn^2)$ для $t=1$.

Для детерминированного случая ($t=1$) оценка не изменилась, а для $t>1$ возросла в n раз. Можно предположить, что эта оценка завышена, а точная оценка остаётся той же $O(bt^n)$. Эта гипотеза верна для случая $b=1$, который не отличается от обхода: полное состояние снова становится неполным, только если появляется новая допустимая в нём кнопка, а здесь только одна кнопка.

Также гипотезу можно доказать, если рестарт определён в каждом состоянии, достижимом при тестировании. Переход в неполное состояние можно делать из начального состояния после рестарта (или нескольких рестартов, если начальное состояние нестабильно, это не увеличивает оценку). Выберем дерево, ориентированное от начального состояния и содержащее все пройденные состояния. Будем двигаться по дереву, нажимая соответствующие кнопки. Если нужный переход дерева не проходится из-за недетерминизма, будем снова делать рестарт и начинать с начала. Обозначим через a_r число состояний, находящих на расстоянии r от начального состояния по дереву. Чтобы гарантированно попасть в такое состояние, нужно не более $O(t^r)$ тестовых воздействий. Чтобы по разу попасть в каждое состояние нужно не более $\sum_r(a_r t^r) \le t+t^2+\ldots+t^{n-1} = O(t^{n-1})$ тестовых воздействий. В каждое состояние нужно переходить не более bt раз, общая оценка $O(bt^n)$.

4.5. Оценка объёма вычислений

Рассмотрим отличия тестирования от обхода, влияющие на объём вычислений.

1) Больше тестовых воздействий. Их число умножается на n для оценки сложности поиска номеров состояний по их идентификаторам: $O(bn^2t^n)$ для $t>1$ или $O(bnt^n)$, если $b=1$ или рестарт везде определён, и $O(bn^3)$ для $t=1$.
2) Лес деревьев строится большее число раз: $O(bn)$ раз вместо $O(n)$ раз, так как неполное состояние становится полным при завершении нажатий некоторой кнопки. Оценка увеличивается в b раз: $O(b^2tn^2)$.
3) Дополнительные блоки алгоритма. Наибольший вклад даёт процедура ***Post***$(i\text{—o}\rightarrow i`,S)$, её однократная работа требует $O(1)$ операций. Есть пример спецификации с k состояниями и реализации с n состояниями, когда в LTS $\mathbf{G}$ будут все пары (i,S), то есть $n2^k$ пар. Для каждой такой пары проверяется не более bt переходов. Оценка $O(btn2^k)$.

Все проверки, выполняемые процедурой ***Post***, необходимы для верификации конформности: проверяются все наблюдения, возможные в реализации после всех безопасных трасс спецификации. Тестовые воздействия требуются только для части этих проверок ($O(btn)$), остальные делаются при распространении.

Итоговая оценка: $O(bn^2t^n)+O(b^2tn^2)+O(btn2^k)$. Если $b=1$ или рестарт везде определён, первое слагаемое будет $O(bnt^n)$, а если $t=1$, то $O(bn^3)$.

4.6. Сильно-Δ-связные реализации

Числов тестовых воздействий определяется, главным образом, проходами в неполные состояния. Алгоритм, основанный на локальной аппроксимации Δ-расстояний по LTS **G**, предполагает, что состояние только один раз становится полным. Но можно вычислять точные Δ-расстояния. По-прежнему, число тестовых воздействий при проходе в неполное состояние ограничено $O(n)$. Число проходов не более btn, поэтому суммарная оценка $O(btn^2)$.

Рассмотрим три слагаемых оценки объёма вычислений.

1) Поиск номеров состояний по их идентификаторам. Число тестовых воздействий $O(btn^2)$, поэтому объём вычислений $O(btn^3)$.
2) Вычисление Δ-расстояний. Отметим Δ-переходы (кнопки в состояниях), по которым считаются минимальные Δ-расстояния. Это похоже на построение леса деревьев и требует число операций порядка числа переходов $O(btn)$. Перевычисляем Δ-расстояния, когда неполное состояние становится полным. Состояние становится полным при завершении нажатий некоторой кнопки, то есть не более b раз. Суммарная оценка $O(b^2tn^2)$.
3) Вычисления в процедуре ***Post***: как и в общем случае, $O(btn2^k)$ операций.

Итоговая оценка объёма вычислений $O(btn^3)+O(b^2tn^2)+O(btn2^k)$.

Литература

1. Бурдонов И.Б., Косачев А.С., Кулямин В.В. Использование конечных автоматов для тестирования программ. «Программирование». 2000. No. 2.
2. Бурдонов И.Б., Косачев А.С., Кулямин В.В. Неизбыточные алгоритмы обхода ориентированных графов. Детерминированный случай. «Программирование». 2003. No. 5.
3. Кулямин В.В., Петренко А.К., Косачев А.С., Бурдонов И.Б. Подход UniTesK к разработке тестов. «Программирование», 2003, No. 6.

4. Бурдонов И.Б., Косачев А.С., Кулямин В.В. Неизбыточные алгоритмы обхода ориентированных графов. Недетерминированный случай. «Программирование». 2004. No. 1.

5. Бурдонов И.Б. Обход неизвестного ориентированного графа конечным роботом. «Программирование», 2004, No. 4.

6. Бурдонов И.Б. Проблема отката по дереву при обходе неизвестного ориентированного графа конечным роботом. «Программирование», 2004, No. 6.

7. Бурдонов И.Б. Исследование одно/двунаправленных распределённых сетей конечным роботом. Труды Всероссийской научной конференции "Научный сервис в сети ИНТЕРНЕТ". 2004.

8. Бурдонов И.Б., Косачев А.С., Кулямин В.В. Формализация тестового эксперимента. «Программирование», 2007, No. 5.

9. Бурдонов И.Б., Косачев А.С., Кулямин В.В. Безопасность, верификация и теория конформности. Материалы Второй международной научной конференции по проблемам безопасности и противодействия терроризму, Москва, МНЦМО, 2007.

10. Бурдонов И.Б., Косачев А.С., Кулямин В.В. Теория соответствия для систем с блокировками и разрушением. «Наука», 2008.

11. Бурдонов И.Б. Теория конформности для функционального тестирования программных систем на основе формальных моделей. Диссертация на соискание учёной степени д.ф.-м.н., Москва, 2008. http://www.ispras.ru/~RedVerst/RedVerst/Publications/TR-01-2007.pdf

12. Василевский М.П. О распознавании неисправностей автомата. Кибернетика, т. 9, № 4, стр. 93–108, 1973.

13. Aho A.V., Dahbura A.T., Lee D., Uyar M.Ü. An Optimization Technique for Protocol Conformance Test Generation Based on UIO Sequences and Rural Chinese Postman Tours. IEEE Transactions on Communications, 39(11):1604–1615, 1991.

14. Blass A., Gurevich Y., Nachmanson L., Veanes M. Play to Test Microsoft Research. Technical Report MSR-TR-2005-04, January 2005. 5th International Workshop on Formal Approaches to Testing of Software (FATES 2005). Edinburgh, July 2005.

15. Edmonds J. Johnson E.L. Matching. Euler Tours, and the Chinese Postman. Math. Programming 5, 88-124 (1973).

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

17. van Glabbeek R.J. The linear time – branching time spectrum. In J.C.M. Baeten and J.W. Klop, editors, CONCUR'90, Lecture Notes in Computer Science 458, Springer-Verlag, 1990, pp 278–297.

18. van Glabbeek R.J. The linear time - branching time spectrum II; the semantics of sequential processes with silent moves. Proceedings CONCUR '93, Hildesheim, Germany, August 1993 (E. Best, ed.), LNCS 715, Springer-Verlag, 1993, pp. 66-81.

19. Goodenough J.B. Gerhart S.L. Toward a theory of test data selection. IEEE Trans. Software Eng., vol. SE-1, no. 2, pp. 156- 173, June 1975.

20. Grochtmann M., Grimm K. Classification trees for partition testing. Software Testing, Verification and Reliability, 3:63-82, 1993.

21. Lee D., Yannakakis M. Testing Finite State Machines: State Identification and Verification. IEEE Trans. on Computers, Vol. 43, No. 3, March 1994, pp. 306-320.

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

23. Legeard B., Peureux F., Utting M. Automated boundary testing from Z and B. In Proc. of the Int. Conf. on Formal Methods Europe, FME'02, volume 2391 of LNCS, Copenhaguen, Denmark, pages 21--40, July 2002. Springer.

24. Milner R. A Calculus of Communicating Processes. LNCS, vol. 92, Springer-Verlag, 1980.

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

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

27. Zhu, Hall, May. Software unit test coverage and adequacy. ACM Computing Surveys, v.29, n.4, 1997.