И.Б.Бурдонов, А.С.Косачев.
Полное тестирование с открытым состоянием ограниченно недетерминированных систем.
Труды Института системного программирования РАН, N 17, 2009, стр.161-192.
32 стр.

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. Тестирование полное, если оно обнаруживает все возможные ошибки в реализации. Для практического применения тестирование должно заканчиваться за конечное время. Требования полноты и конечности тестирования в общем случае взаимно противоречат друг другу. Однако для ограниченных классов реализаций и спецификаций, а также при использовании дополнительных тестовых возможностей, удаётся построить конечные полные тесты. Предлагаются алгоритмы тестирования и даётся оценка их сложности для конечных спецификаций и конечных реализаций с ограниченным недетерминизмом при тестировании с открытым состоянием.

1. Введение

Тестирование понимается как проверка соответствия (конформности) реализации требованиям, сформулированным в виде спецификации. Тестирование считается полным, если оно однозначно отвечает на вопрос: есть в реализации ошибки или нет, где под ошибкой понимается нарушение требований, то есть неконформность реализации спецификации. Для практического применения тестирование должно заканчиваться за конечное время. К сожалению, во многих случаях тестирование либо неполно, либо бесконечно. Решение проблемы можно искать, сужая класс рассматриваемых реализаций и/или предполагая дополнительные тестовые возможности. В некоторых случаях удаётся построить частные алгоритмы полного и конечного тестирования, но применимые только для рассматриваемого класса реализаций и использующие дополнительные тестовые возможности.

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

Если объём требований, описываемых спецификацией, бесконечен, мы не сможем их все проверить за конечное время, то есть конечное тестирование будет заведомо неполным. Если объём требований конечен, но каждое или некоторые из них нужно проверять в бесконечном числе ситуаций, то за конечное время мы также не сможем это сделать. Для полноты конечного тестирования нужно, чтобы реализация имела конечное число неэквивалентных ситуаций, в которых нужно проверять требования спецификации, то есть реализация также должна быть конечной.

Однако конечности реализации недостаточно, если её объём неизвестен: ни в какой момент времени мы не можем знать, все ли имеющиеся в реализации ситуации мы проверили или нет. Нам нужно уметь оценить объём реализации либо заранее, предполагая, что он не только конечен, но и ограничен, либо в процессе тестирования. В последнем случае предполагается наличие дополнительных тестовых возможностей, позволяющих наблюдать в том или ином виде протестированную часть реализации и делать выводы о наличии или отсутствии других частей.

Если поведение реализации недетерминировано без каких-либо ограничений, ни в какой момент времени мы не можем знать, продемонстрировала ли реализация все варианты своего

недетерминированного поведения или нет. Поэтому для полноты конечного тестирования на недетерминизм реализации приходится налагать те или иные ограничения.

В данной статье мы рассматриваем тестирование *конечной реализации* по *конечной спецификации* с двумя дополнительными предположениями: 1) *тестирование с открытым состоянием* – у нас есть возможность наблюдать состояния реализации, в которых мы оказываемся 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.

Статья состоит из четырёх основных разделов. Во 2-ом разделе кратко излагаются основные положения теории конформности, которые были изложены в работах авторов [11,13,14]. В 3-ом разделе обсуждаются проблемы практического тестирования и пути их решения. В 4-ом и 5-ом разделах описываются алгоритмы тестирования, доказывается конечность и полнота тестирования и приводятся оценки сложности.

2. Теория конформности

2.1. 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 будем рассматривать такие семантики взаимодействия, которые основаны только на внешнем, наблюдаемом поведении системы и не учитывают её внутреннее устройство, отображаемое на уровне модели в понятии *состояния*. В этом случае говорят о тестировании методом «чёрного ящика» или функциональном тестировании. Мы можем наблюдать только такое поведение реализации, которое, во-первых, «спровоцировано» тестом (управление) и, во-вторых, наблюдаемо во внешнем взаимодействии. Такое взаимодействие может моделироваться с помощью, так называемой, машины тестирования [11,13,14,20,21,31]. 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 какие-то кнопки на клавиатуре машины, «приказывая» или «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

Следует подчеркнуть, что при управлении оператор разрешает реализации выполнять именно множество действий, а не обязательно одно действие. Например, при тестировании реактивных систем, основанных на обмене стимулами и реакциями, посылка одного стимула из теста в реализацию может интерпретироваться как разрешение реализации выполнять только одно действие – приём этого стимула. Однако приём тестом ответной реакции должен означать разрешение реализации выдавать любую реакцию как раз для того, чтобы проверить, правильна эта реакция или нет. Мы будем считать, что оператор нажимает одну кнопку, но на кнопке «написано», вообще говоря, не одно действие, а множество разрешаемых действий. Когда происходит наблюдение (действие или отказ) кнопка автоматически отжимается, и все внешние действия считаются запрещёнными. Далее оператор может нажимать другую (или ту же самую) кнопку.

В то же время «кнопочное» множество – это, вообще говоря, не любое подмножество множества всех действий. В вопросе о том, какие множества действий могут разрешаться тестом, а какие нет, среди исследователей существует большое разнообразие точек зрения. Например, для реактивных систем обычно считается, что нельзя (или бессмысленно) смешивать посылку стимулов с приёмом реакций (Ян Тритманс). Но существует и прямо противоположный подход: нельзя «тормозить» выдачу реакций реализацией, поэтому, даже посылая стимул, тест должен быть готов к приёму любой реакции (А.Ф. Петренко в [34]).

Также следует подчеркнуть, что наблюдение отказа возможно не при любой кнопке. И здесь разные исследователи опираются на разные предположения. Для тех же реактивных систем долгое время считалось, что тест может наблюдать отсутствие реакций (*quiescence*, стационарность), например, по тайм-ауту, но не видит, принимает реализация посланный ей стимул или нет (*input refusal*, блокировка стимула). С другой стороны, в последние годы появляется всё больше и больше работ, в которых такие блокировки стимулов допускаются или допускаются частично [8-13,24,25,29]. Также и реакции, если они принимаются тестом по разным «выходным каналам», можно принимать не все, а лишь те, которые относятся к одному или нескольким выбранным каналам [24,25].

Итак, семантика взаимодействия определяется тем, какие (в принципе) существуют наблюдаемые действия – алфавит внешних действий L, какие множества действий может разрешать тест – набор кнопок машины тестирования, и для каких из этих кнопок наблюдаемы соответствующие отказы – семейство $\mathfrak{R}\subseteq\mathcal{P}(L)$, а для каких нет – семейство $\mathfrak{Q}\subseteq\mathcal{P}(L)$. Предполагается, что $\mathfrak{R}\cap\mathfrak{Q}=\varnothing$ и $\cup\mathfrak{R}\cup\cup\mathfrak{Q}=L$. Такую семантику мы называем $\mathfrak{R}/\mathfrak{Q}$-семантикой.

В [11,14] показано, что достаточно ограничиться семантиками, в которых все отказы наблюдаемы, то есть $\mathfrak{Q}=\varnothing$. Любая $\mathfrak{R}/\mathfrak{Q}$-семантика оказывается эквивалентной $\mathfrak{R}\cup\mathfrak{Q}/\varnothing$-семантике в том смысле, что для любой спецификации, заданной в $\mathfrak{R}/\mathfrak{Q}$-семантике, существует (и может быть построена при некоторых, практически приемлемых, ограничениях) спецификация в $\mathfrak{R}\cup\mathfrak{Q}/\varnothing$-семантике, определяющая тот же класс конформных реализаций и не меньший класс реализаций, поддающихся тестированию. По

этой причине в данной статье мы будем рассматривать только $\mathfrak{R}$-семантики, предполагая, что все отказы наблюдаемы ($\mathfrak{Q}=\varnothing$).

Семантика взаимодействия предполагает, что выполняться может только то действие, которое определено в реализации и разрешено оператором машины тестирования. Если система может выполнять любое определённое и разрешённое действие, и выбор выполняемого действия не детерминирован, то говорят, что в системе нет приоритетов. Наличие приоритетов означает, что не все определённые и разрешённые действия могут выполняться, то есть выполнимость действия зависит также от того, какие ещё действия определены и/или разрешены. В данной статье мы ограничимся системами без приоритетов.

Кроме внешних, наблюдаемых действий реализация может совершать внутренние, ненаблюдаемые (и, следовательно, неразличимые оператором) действия, которые обозначаются символом τ. Выполнение таких действий не регулируется оператором – они всегда разрешены. Предполагается, что любая конечная последовательность любых действий совершается за конечное время, а бесконечная последовательность – за бесконечное время. Бесконечная последовательность τ-действий («зацикливание») называется *дивергенцией* и обозначается символом Δ. Кроме этого мы вводим специальное, не регулируемое кнопками, действие, которое называем *разрушением* и обозначаем символом γ. Оно моделирует любое запрещённое или недекларированное поведение реализации. Например, в терминах пред- и постусловий, поведение программы определено (постусловием) только в том случае, когда выполнено предусловие обращения к ней. Если же предусловие нарушено, поведение программы считается полностью неопределённым. Семантика разрушения предполагает, в частности, что в результате такого поведения система может быть разрушена.

При тестировании мы должны избегать попыток выхода из дивергенции и разрушения. Такое тестирование называется безопасным. Опасность разрушения подразумевается его семантикой. Поясним случай дивергенции. В целом её опасность означает, что после нажатия кнопки оператор может не получить никакого наблюдения, и, не зная об этом, не может ни продолжать тестирование, ни закончить его. Само по себе возникновение дивергенции не опасно, однако, нажимая после этого любую кнопку, оператор, не наблюдая внешнего действия или отказа, не знает, случится ли такое наблюдение в будущем, или реализация так и будет бесконечно долго выполнять свои внутренние действия.

Можно также отметить, что из-за внутренних действий нажатие пустой кнопки (кнопки с пустым множеством разрешаемых действий) не эквивалентно отсутствию нажатой кнопки. В обоих случаях все внешние действия запрещены, однако наблюдение отказа означает, что оператор узнаёт об остановке машины, когда она не может выполнять также и внутренние действия. Пустая кнопка не может вызвать разрушение после действия (никакого действия быть не может), но она опасна, если есть дивергенция, как и любая другая кнопка.

2.2. LTS-модель и трассовая модель

В качестве модели реализации и спецификации мы используем *систему помеченных переходов* (LTS – Labelled Transition System). LTS – это 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, τ – символ внутреннего действия, γ – символ разрешения, $E_S\subseteq V_S\times(L\cup\{\tau,\gamma\})\times V_S$ – 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 $s—z\rightarrow s`$. Обозначим $s—z\rightarrow$ $=_{def}$ $\exists s`$ $s—z\rightarrow s`$ и $s—z\nrightarrow$ $=_{def}$ $\nexists s`$ $s—z\rightarrow s`$.

Выполнение LTS, помещённой в «чёрный ящик» машины тестирования, сводится к выполнению того или иного перехода, определённого в текущем состоянии и разрешаемого нажатой кнопкой (τ- и γ-переходы разрешены при нажатии любой кнопки и при отсутствии нажатой кнопки).

Состояние называется *стабильным*, если в нём не определены τ- и γ-переходы, и *дивергентным*, если в нём начинается бесконечная цепочка τ-переходов (в частности, τ-цикл, в том числе, τ-петля). Если состояние не дивергентно, оно называется *конвергентным*. Отказ P порождается стабильным состоянием, в котором нет переходов по действиям из P. Переход $s—z\rightarrow s`$ называется *разрушающим*, если из состояния $s`$ достижимо по цепочке (быть может, пустой) τ-переходов состояние, в котором определён γ-переход.

Для получения трасс LTS достаточно добавить в каждом стабильном состоянии виртуальные петли, помеченные порождаемыми состоянием отказами, а также добавить Δ-переходы во всех дивергентных состояниях. После этого рассматриваются все конечные маршруты LTS, начинающиеся в начальном состоянии и не продолжающиеся после Δ- или γ-перехода. Трассой маршрута считается последовательность пометок его переходов с пропуском τ-переходов. Такие трассы мы называем *полными* или ***F**-трассами*, а множество **F**-трасс LTS S – *полной трассовой моделью* или ***F**-моделью*, и обозначаем $F(S)$. **F**-трасса, все отказы которой принадлежат семейству $\mathfrak{R}$, называется $\mathfrak{R}$-трассой. Это те трассы, которые могут наблюдаться на машине тестирования в $\mathfrak{R}$-семантике. Множество всех $\mathfrak{R}$-трасс LTS, то есть проекция её **F**-модели на алфавит, состоящий из всех внешних действий, отказов, символов Δ и γ, называется $\mathfrak{R}$-моделью, соответствующей «взгляду» на реализацию в $\mathfrak{R}$-семантике.

2.3. Гипотеза о безопасности и безопасная конформность

Безопасное тестирование, прежде всего, предполагает формальное определение на уровне модели отношения безопасности «*кнопка безопасна в модели после $\mathfrak{R}$-трассы*», которое означает, что нажатие кнопки P после $\mathfrak{R}$-трассы σ не может означать попытку выхода из дивергенции (после трассы нет дивергенции) и не может вызывать разрушение (после действия, разрешаемого кнопкой). При безопасном тестировании будут нажиматься только безопасные кнопки.

Определим формально отношения безопасности кнопки:

<u>в состоянии s</u>:

P ***safe*** s, если состояние s не дивергентно и в каждом состоянии q, достижимом из s по τ-переходам нет γ-переходов и все переходы из q по действию $z\in P$ неразрушающие;

<u>во множестве состояний S</u>:

P ***safe*** S $=_{def}$ $\forall s\in S$ P ***safe*** s;

<u>после трассы σ</u>:

P ***safe*** S ***after*** σ $=_{def}$ $\forall u\in P$ $\sigma\cdot\langle u,\gamma\rangle\notin F(S)$ & $\sigma\cdot\langle\Delta\rangle\notin F(S)$,

что эквивалентно 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 определяет безопасность действий и отказов после $\mathfrak{R}$-трассы. Отказ R $\mathbf{safe}$ $(S$ $\mathbf{after}$ $\sigma)$, если после трассы σ безопасна кнопка R. Действие z $\mathbf{safe}$ $(S$ $\mathbf{after}$ $\sigma)$, если оно разрешается $z\in P$ некоторой кнопкой P $\mathbf{safe}$ $(S$ $\mathbf{after}$ $\sigma)$. Теперь мы можем определить *безопасные трассы*. $\mathfrak{R}$-трасса безопасна, если 1) модель не разрушается с самого начала (сразу после включения машины ещё до нажатия первой кнопки), то есть в ней нет трассы $\langle\gamma\rangle$, 2) каждый символ трассы безопасен после непосредственно предшествующего ему префикса трассы. Множества безопасных трасс модели S обозначим $\mathbf{Safe}(S)$.

Требование безопасности тестирования выделяет класс *безопасных* реализаций, то есть таких, которые могут быть безопасно протестированы для проверки их конформности или неконформности 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:

$$
\begin{aligned}
I\ \mathbf{safe\ for}\ S =_{def}\ & (\langle\gamma\rangle\notin \mathbf{F}(S) \Rightarrow \langle\gamma\rangle\notin \mathbf{F}(I)) \\
\&\ & \forall\sigma\in \mathbf{Safe}(S)\cap \mathbf{F}(I)\ \forall P\in\mathfrak{R} \\
& (P\ \mathbf{safe}\ S\ \mathbf{after}\ \sigma \Rightarrow P\ \mathbf{safe}\ I\ \mathbf{after}\ \sigma).
\end{aligned}
$$

Следует отметить, что гипотеза о безопасности не проверяема при тестировании и является его предусловием. После этого можно определить отношение (безопасной) *конформности*: реализация I *безопасно конформна* (или просто *конформна*) спецификации S, если она безопасна и выполнено *тестируемое условие*: любое наблюдение, возможное в реализации в ответ на нажатие безопасной (в спецификации) кнопки, разрешается спецификацией:

$$
\begin{aligned}
I\ \mathbf{saco}\ S =_{def}\ & I\ \mathbf{safe\ for}\ S \\
\&\ & \forall\sigma\in \mathbf{Safe}(S)\cap \mathbf{F}(I)\ \forall P\ \mathbf{safe}\ S\ \mathbf{after}\ \sigma \\
& \mathbf{obs}(I\ \mathbf{after}\ \sigma, P)\subseteq \mathbf{obs}(S\ \mathbf{after}\ \sigma, P),
\end{aligned}
$$

где $\mathbf{obs}(M,P) =_{def} \{u \mid \exists m\in M\ u\in P\ \&\ m\xrightarrow{u} \lor\ u=P\ \&\ \forall z\in P\ m\not\xrightarrow{z}\}$ – множество наблюдений, которые возможны в состояниях из множества M при нажатии кнопки P.

2.4. Параллельная композиция и генерация тестов

Взаимодействие двух систем моделируется в LTS-теории оператором параллельной композиции. Мы используем оператор композиции, аналогичный тому, который определяется в алгебре процессов CCS (Calculus of Communicating Systems) [30,32]. Будем считать, что для каждого внешнего действия z определено противоположное действие $\underline{z}$ так, что $\underline{\underline{z}}=z$. Например, посылке стимула из теста соответствует приём теста в реализации, а выдаче реакции реализацией соответствует приём этой реакции в тесте. Параллельное выполнение двух LTS в алфавитах A и B понимается так, что переходы по противоположным действиям z и $\underline{z}$, где $z\in A$ и $\underline{z}\in B$, выполняются синхронно, то есть, в обеих LTS одновременно, причём в композиции это становится τ-переходом. Такие действия называются синхронными. Остальные внешние действия $z\in A\setminus\underline{B}$ и $z\in B\setminus\underline{A}$, а также действия τ и γ называются асинхронными. Переход по такому действию выполняются в одной из LTS при сохранении состояния другой LTS. Результатом

композиции двух LTS **I** и **T** становится LTS **I**⇂⇃**T** в алфавите A⇂⇃B $=_{def}$ (A\\B)∪(B\\A). Её состояния – это пары состояний it LTS-операндов, начальное состояние – это пара начальных состояний, а переходы порождаются следующими правилами вывода:

(1) z∈{τ,γ}∪A\\B & i—z→i\` ⊢ it—z→i\`t,

(2) z∈{τ,γ}∪B\\A & t—z→t\` ⊢ it—z→it\`,

(3) z∈A∩B & i—z→i\` & t—z→t\` ⊢ it—τ→i\`t\`.

Тестирование понимается как замкнутая композиция LTS-реализации **I** в алфавите A и LTS-теста **T** в противоположном алфавите B=A. Для обнаружения отказов в тесте (но не в реализации!) допускаются специальные θ-переходы, которые срабатывают тогда и только тогда, когда никакие другие переходы не могут выполняться:

(4) t—θ→t\` & ***Deadlock***(i,t) ⊢ it—τ→it\`,

где ***Deadlock***(i,t) = i—τ↛ & i—γ↛ & t—τ↛ & t—γ↛ & (∀z∈A∩B i—z↛ ∨ t—z↛).

Мы будем предполагать, что в тесте нет разрушения (требование t—γ↛ всегда выполнено).

Поскольку алфавиты реализации и теста противоположны, композиционный алфавит пуст и в композиционной LTS есть только τ- и γ-переходы. При безопасном тестировании γ-переходы недостижимы. Выполнению теста соответствует прохождение τ-маршрута, начинающегося в начальном состоянии композиции **I**⇂⇃**T**. Тест заканчивается, когда достигается терминальное состояние теста. Каждому такому терминальному состоянию назначается вердикт ***pass*** или ***fail***. Реализация *проходит* тест, если состояния теста с вердиктом ***fail*** недостижимы. 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.

Обычно ограничиваются, так называемыми, *управляемыми* тестами, то есть тестами без лишнего недетерминизма. Для этого множество внешних действий, для которых определены переходы в данном состоянии теста, должно быть одним из «кнопочных» множеств 𝔑-семантики (точнее, множеством противоположных действий, поскольку при композиции CCS тест определяется в противоположном алфавите). Оператор, исполняя тест, однозначно определяет, какую кнопку ему нужно нажимать в данном состоянии теста. Для обнаружения отказа в состоянии теста должен быть также определён θ-переход.

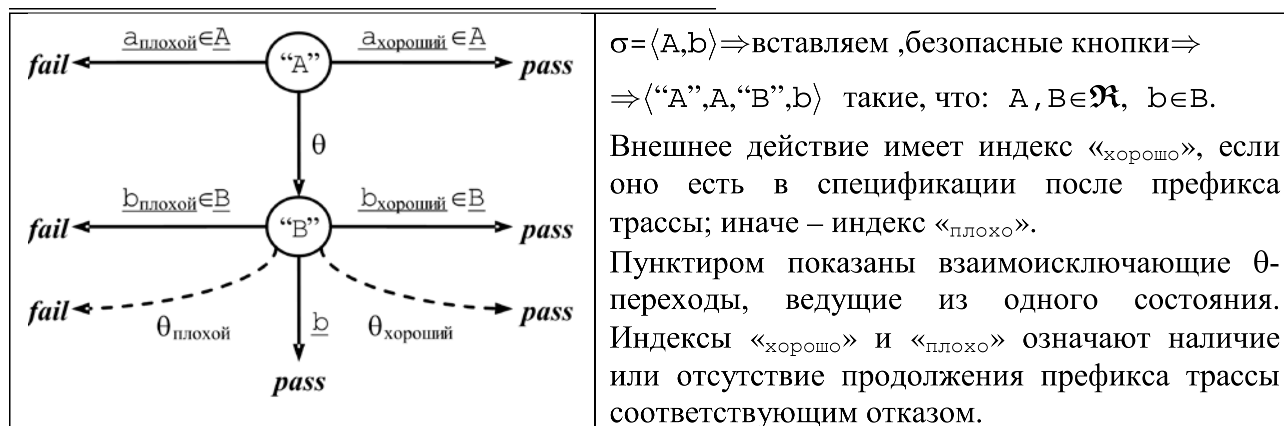

Рис.2. Примитивный тест для безопасной 𝔑-трассы σ

Полным набором всегда является набор всех *примитивных* тестов. Примитивный тест строится по одной выделенной безопасной 𝔑-трассе спецификации. Для этого сначала в трассу 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, то есть по одной безопасной трассе спецификации можно сгенерировать, вообще говоря, множество разных примитивных тестов. Для различения кнопок и отказов (и то и другое – подмножества внешних действий) мы будем кнопки заключать в кавычки и писать "P", а не просто P. По полученной последовательности действий, отказов и кнопок строится LTS-тест (Рис.2). Его состояниями становятся расставленные кнопки, начальное состояние – это первая в последовательности кнопка, символы переходов из состояния-кнопки – это действия, противоположные тем, которые могут наблюдаться после нажатия этой кнопки, или символ θ. Если это не последняя кнопка, то один переход ведёт в состояние, соответствующее следующей кнопке. Остальные переходы ведут в терминальные состояния. Вердикт ***pass*** назначается тогда, когда соответствующая 𝔑-трасса есть в спецификации, а вердикт ***fail*** – когда нет. Такой вердикт соответствует *строгим* тестам, которые, во-первых, значимые (не ловят ложных ошибок) и, во-вторых, не пропускают обнаруженных ошибок. Любой строгий тест можно заменить на объединение примитивных тестов, которое обнаруживает те же самые ошибки.

3. Проблемы практического тестирования

3.1. Недетерминизм и глобальное тестирование

Как уже было сказано, в каждый момент времени реализация может выполнять любое определённое в ней и разрешённое оператором внешнее действие, а также определённые и всегда разрешённые внутренние действия (мы избегаем разрушения при безопасном тестировании). Если таких действий несколько, выбирается одно из них недетерминированным образом. Здесь предполагается, что недетерминизм поведения реализации – это явление того уровня абстракции, которое определяется нашими тестовыми возможностями по наблюдению и управлению, то есть семантикой взаимодействия. Иными словами, поведение реализации недетерминировано, поскольку оно зависит от неких не учитываемых нами факторов – «погодных условий», которые определяют выбор выполняемого действия детерминировано.

Для того, чтобы тестирование могло быть полным, мы должны предположить, что любые погодные условия могут быть воспроизведены в тестовом эксперименте, причём для каждого

теста. Если такая возможность есть, тестирование называется *глобальным* [31]. Мы абстрагируемся от количества вариантов погодных условий. Здесь нам важна только потенциальная возможность проверить поведение системы при любых погодных условиях и любом поведении оператора. Конечно, на практике используется только конечное число прогонов каждого теста. Без дополнительных условий мы не можем быть уверены, что провели тестовые испытания каждого теста для всех возможных погодных условий. Возможны различные решения этой проблемы.

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

Другое решение – специальные реализационные гипотезы. Для конечного числа прогонов теста предполагают, что, если реализация ведёт себя правильно при некоторых погодных условиях, то она будет вести себя правильно при любых погодных условиях [4].

Третье решение основано на том, что нам известно распределение вероятностей тех или иных погодных условий. В этом случае тестирование оказывается полным с той или иной вероятностью [17].

Близкое к этому четвёртое решение предполагает, что в каждой ситуации (после трассы) возможно лишь конечное число погодных условий (с точностью до эквивалентности) и существует такое число `t`, что после `t` прогонов теста гарантированно будет проверено поведение реализации при всех возможных в этой ситуации погодных условиях [19,30]. Это можно назвать *ограниченным (числом* `t`*) недетерминизмом*.

Наконец, существует и более радикальное решение – просто запретить недетерминизм реализации, то есть реализационная гипотеза ограничивает класс реализаций только детерминированными реализациями. Очевидно, что это эквивалентно ограниченному недетерминизму при `t=1`. При всей своей наивности, это достаточно распространённый практический приём [33] (он применялся и в ИСП РАН в рамках системы UniTESK). Обоснованием может служить то, что во многих случаях заранее известно, что интересующие нас реализации детерминированы.

3.2. 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; в частности, в LTS-спецификации не должно быть циклов.) Однако на практике используются только конечные наборы конечных (по времени выполнения) тестов. Возможны различные решения этой проблемы. В конечном счёте все они сводятся к специальным тестовым возможностям и/или реализационным гипотезам. Такие гипотезы, по сути, предполагают ровно то, что хотелось бы «доказать»: если реализация ведёт себя правильно на тестах данного конечного набора, то она будет вести себя правильно на всех тестах полного набора. Обоснованием может служить некоторое (не важно как полученное) «знание» о том, как могут быть устроены тестируемые реализации (тестируем не все возможные реализации, а только такие). На классе всех реализаций такие конечные наборы тестов будут только значимыми (не ловят

ложных ошибок), а полными – только на подклассе, определяемом реализационной гипотезой.

Теория конформности предполагает некоторый общий критерий покрытия, требующий проверки всех возможных ситуаций. Часто используются специальные критерии покрытия, чтобы проверить не все ситуации, а только некоторые интересующие нас классы ситуаций (основанные на модели возможных ошибок) [22,23,35]. Теоретически конечный набор можно получить фильтрацией по критерию покрытия перечислимого полного набора. Поэтому теория конформности должна быть развита до уровня алгоритмов перечисления полного набора. Однако на практике обычно используются более прямые методы построения нужного конечного набора. Достаточно общий подход сводится к тому, что вместо исходной спецификационной модели используется более грубая, так называемая, тестовая модель. Тестовая модель – это результат факторизации исходной LTS-спецификации по отношению эквивалентности переходов, что обычно сводится к эквивалентности состояний и/или действий [1]. Иногда при факторизации исчезает недетерминизм, что заодно решает и эту проблему. Разумеется, чтобы такой подход был оправданным, нужны мотивированные реализационные гипотезы о том, что ошибки, возможные в реализации, обнаруживаются при тестировании по факторизованной спецификации (вообще по конечному набору, удовлетворяющему критерию покрытия).

Примером практического тестирования может служить тестирование конечного автомата по спецификации, заданной также в виде конечного автомата [2,3,4,15,16,26,27,28]. Конформность, которая при этом проверяется, это эквивалентность автоматов. В большинстве работ предполагается, что спецификация конечна и детерминирована. Реализационная гипотеза предполагает, что реализация также детерминирована, а число её состояний не превосходит числа состояний спецификации (с точностью до эквивалентности состояний), или превосходит не более, чем на заданную величину.

3.3. Тестирование с открытым состоянием

Тестированием с открытым состоянием называется тестирование, при котором у нас есть специальная операция, позволяющая достоверно и напрямую опросить текущее состояние реализации (*status message*).

Как известно, при тестировании с открытым состоянием конечных автоматов на эквивалентность полное тестирование сводится к обходу графа переходов автомата и применению операции опроса в каждом проходимом состоянии [2,4-7,18,27].

Для общего случая LTS тестирование с открытым состоянием можно понимать в терминах машины тестирования как наличие специальной кнопки для опроса текущего состояния реализации. Мы будем нажимать эту кнопку после каждого наблюдения. Это эквивалентно тому, что постсостояние после наблюдения высвечивается на дисплее машины как часть наблюдения (вместе с отказом или выполненным действием). Важно подчеркнуть, что, если после выполнения действия реализация оказывается в нестабильном состоянии `i`, то высвечиваемое постсостояние не обязательно равно `i`, а может быть любым состоянием, достижимым из `i` по τ-переходам. Это означает, что мы не требуем приостановки внутренней работы реализации после наблюдения до опроса состояния. Точно также мы не требуем приостановки после опроса состояния до следующего тестового воздействия или завершения тестирования. Выполнение нескольких опросов состояний подряд даёт возможность наблюдать движение реализации по τ-переходам, но мы думаем, что такая возможность не увеличивает мощности тестирования конформности ***saco***.

Старт (или рестарт) системы понимается как специальное тестовое воздействие, входящее в $\mathfrak{R}$. Его отличие от других тестовых воздействий лишь в том, что оно даёт постсостояние, гарантированно совпадающее с начальным или достижимым из него по цепочке τ-переходов. Состояние из множества I ***after*** ϵ будем называть начально-достижимым. Вообще говоря, можно считать, что рестарт определён не в каждом состоянии реализации; в стабильных состояниях наблюдается отказ по рестарту. Будем считать, что если рестарт выполняется, то наблюдается действие «рестарт». Соответственно, LTS реализации пополняется переходами по рестарту. Такой переход «сбрасывает» трассу: трассой маршрута считается трасса его постфикса после последнего рестарта.

3.4. Условия конечности тестирования

Сформулируем ограничения на реализации и спецификации, при которых возможно конечное полное тестирование и которые будут рассматриваться в данной статье. Мы предполагаем тестирование с открытым состоянием конформности ***saco*** и общий (не специальный) критерий покрытия. Для каждого ограничения (выделенного подчёркиванием) мы укажем возможные его ослабленные варианты, но для простоты изложения в дальнейшем эти варианты рассматривать не будем.

Ограниченный недетерминизм реализации. Для решения проблемы недетерминизма мы предполагаем, что реализация обладает ограниченным недетерминизмом: существует такое число t, что в любом состоянии i реализации при нажатии любой кнопки t раз будут получены все возможные пары (наблюдение, постсостояние). Более точно, будут пройдены все маршруты, начинающиеся в состоянии i и имеющие трассу $\langle$наблюдение$\rangle$. Это ограничение, очевидно, можно ослабить, рассматривая только достижимые состояния реализации. Более того, для данной спецификации нас интересуют только те состоянии реализации, которые достижимы по трассам, безопасным в спецификации. Заметим, что из ограниченного недетерминизма следует, что для любой кнопки (даже с бесконечным кнопочным множеством) в каждом состоянии реализации может быть определено не более t переходов по действиям, разрешаемым этой кнопкой.

Для решения проблемы бесконечности полного тестового набора мы наложим ограничения на размеры $\mathfrak{R}$-семантики, спецификации и реализации.

Число кнопок конечно. Это ограничение связано с тем, что при тестировании мы должны нажимать все кнопки, безопасные в спецификации после пройденной трассы. Если все кнопки безопасны, то мы должны выполнить бесконечное число тестовых воздействий.

Есть алгоритм разрешения кнопки относительно всех действий, который для каждого действия z и каждой кнопки P за конечное время определяет $z \in P$ или нет. В частности, это всегда можно сделать, если все кнопочные множества конечны (тогда конечен и алфавит действий). 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.

Число состояний спецификации конечно. Спецификация с бесконечным числом состояний, очевидно, требует бесконечного тестирования хотя бы в том случае, когда тестируется сама спецификация. Это ограничение можно ослабить, поскольку нас будут интересовать только такие состояния, которые достижимы по безопасным трассам спецификации. Конечно, при тестировании данной реализации не все такие состояния спецификации окажутся достижимыми, однако мы предполагаем, что спецификацию можно использовать для

тестирования любой реализации (с учётом ограничений на недетерминизм и размер реализации).

Число переходов спецификации конечно. Это нужно для проверки безопасности кнопки после пройденной трассы. Это ограничение можно ослабить, поскольку нас будут интересовать только такие переходы, которые лежат на маршрутах с безопасными трассами. Однако этого мало для проверки безопасности любой кнопки после любой безопасной трассы. Для такой проверки нужно перебирать все переходы, ведущие из состояний после трассы, и, если переход разрушающий, проверять, не принадлежит ли действие, которым переход помечен, данной кнопке. Безопасность кнопки можно проверить за конечное время в двух случаях. 1) Если число переходов после трассы конечно. 2) Если конечна кнопка, множество переходов перечислимо, и спецификация задана таким образом, что переходы по одному и тому же действию перечисляются подряд. Заметим, что для каждого действия число переходов по этому действию конечно, поскольку конечно число состояний. Тогда в процессе перебора переходов мы можем отмечать 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, может случиться так, что мы не определим безопасность кнопки за конечное время. То же самое имеет место, если множество переходов и кнопка оба бесконечны.

Число состояний реализации конечно. Если реализация имеет бесконечное число состояний, то может случиться так, что при тестировании ошибка будет обнаружена за конечное время, однако во многих случаях это не гарантировано; кроме того, конформные бесконечные реализации могут тестироваться бесконечно долго. Это ограничение тоже можно ослабить, поскольку нас будут интересовать только такие состояния реализации, которые достижимы по безопасным трассам заданной спецификации. При нарушении ослабленного ограничения бесконечные конформные реализации будут всегда тестироваться бесконечно долго.

Сильно-связность реализации. Полнота тестирования может быть достигнута только тогда, когда в процессе тестирования реализация выполняет все переходы, лежащие на маршрутах с трассами, безопасными в спецификации. Если рассмотреть LTS, порождаемую всеми такими переходами, то при тестировании выполняется её обход, то есть проходится маршрут, содержащий все переходы этой LTS. Для этого LTS должна быть сильно-связной: из каждого состояния можно попасть по переходам в каждое другое состояние. При наличии рестарта к таким переходам относятся и переходы по рестарту. Более точно, для обхода необходимо и достаточно, чтобы LTS представляла собой цепочку сильно-связных компонентов, в которой из каждого непоследнего компонента ведёт ровно один переход в следующий компонент [2]. Однако в этом случае алгоритм обхода возможен лишь тогда, когда LTS детерминирована и все компоненты в цепочке, кроме последнего, содержат по одному состоянию без петель. В дальнейшем будем считать, что LTS сильно-связна.

4. Алгоритм обхода реализации

Сначала рассмотрим задачу обхода реализации, в которой нет разрушения и дивергенции. Более точно: разрушение и дивергенция недостижимы из начального состояния реализации. Кроме того, нас будут интересовать только такие переходы реализации, которые достижимы из начального состояния. Также будем считать, что выполнены ограничения, касающиеся реализации и перечисленные в предыдущем подразделе: множества кнопок и состояний реализации конечны, реализация сильно-связна и ограниченно недетерминирована.

4.1. LTS обхода

Обозначим: $\mathbf{I}$ – реализационная LTS. В процессе обхода мы будем строить *LTS обхода*, которая порождается всеми маршрутами реализации, начинающимися в начальном состоянии. В каждый момент обхода часть из этих маршрутов оказывается пройденной, что означает построение части LTS обхода, которую мы будем называть пройденной LTS и обозначать $\mathbf{G}$. Её состояния – это состояния реализации, которые мы уже наблюдали. Переход $i\text{—}z\rightarrow i`$ по внешнему действию $z$ добавляется, когда в реализационном состоянии $i$ мы нажимаем некоторую кнопку $P$ и наблюдаем действие $z\in P$ и постсостояние $i`$. Это означает, что в реализации имеется маршрут, начинающийся и заканчивающийся в состояниях i и $i`$, соответственно, и имеющий трассу $\langle z\rangle$, то есть содержащий один переход по $z$, а все остальные переходы – это $\tau$-переходы. Если после нажатия кнопки $P$ в состоянии $i$ мы наблюдаем отказ с *тем же самым* постсостоянием $i`=i$, то никаких переходов не добавляется. Если же отказ P наблюдается с *другим* постсостоянием $i`\neq i$, то добавляется переход $i\text{—}\tau\rightarrow i`$. Это означает, что в реализации имеется τ-маршрут, начинающийся и заканчивающийся в состояниях i и $i`$, соответственно, причём состояние $i`$ стабильно, и в нём есть отказ P. Вместе с каждым переходом мы будем хранить кнопку, нажатие которой вызвало этот переход; таких кнопок может быть несколько, нам достаточно хранить любую из них.

Обход заканчивается, когда LTS обхода полностью построена, то есть в $\mathbf{G}$ нельзя добавить ни одного нового перехода. Легко показать, что LTS $\mathbf{I}$ и LTS обхода имеют одно и то же множество достижимых состояний и одно и то же множество $\mathfrak{R}$-трасс. Кроме того, любой алгоритм, выполняющий обход (неизвестной заранее) LTS обхода, гарантирует проход по каждому переходу реализации $\mathbf{I}$.

Для каждой кнопки P и каждого пройденного состояния i определим счётчик $c(P,i)$ числа нажатий кнопки P в состоянии i. Будем говорить, что кнопка P *полна в состоянии* i, если 1) $c(P,i)=1$, в $\mathbf{G}$ нет τ-переходов из i и нет перехода $i\text{—}z\rightarrow i`$, где $z\in P$, или 2) $c(P,i)=t$. Это означает, что мы уже получили все возможные переходы из состояния i при нажатии кнопки P. В первом случае в состоянии i мы нажимали кнопку P и наблюдали отказ P в этом же состоянии. Поэтому нет смысла ещё раз нажимать кнопку P: ничего другого мы наблюдать не можем. Во втором случае после t нажатий кнопки мы получили все возможные переходы.

Состояние i будем называть *полным*, если каждая кнопка полна в нём. Это означает, что все возможные переходы в $\mathbf{G}$ из состояния i мы уже получили.

В начале обхода в $\mathbf{G}$ у нас есть только одно состояние реализации $i\in\{\mathbf{I}\ \textit{after}\ \epsilon\}$ и $\forall P\in\mathfrak{R}\ c(P,i)=0$.

4.2. Общая схема алгоритма обхода

На Рис.3 изображена общая схема алгоритма. Если текущее состояние i неполное, то некоторая кнопка P неполна в i. В этом случае мы нажимаем эту кнопку, осуществляя тем самым тестовое воздействие, и получаем наблюдение o (действие или отказ) и постсостояние $i`$, которое становится новым текущим состоянием. Увеличиваем счётчик

c(P,i):=c(P,i)+1. После этого в **G** добавляется переход i—o→i`, если o действие, или переход i—τ→i`, если o отказ и i`≠i.

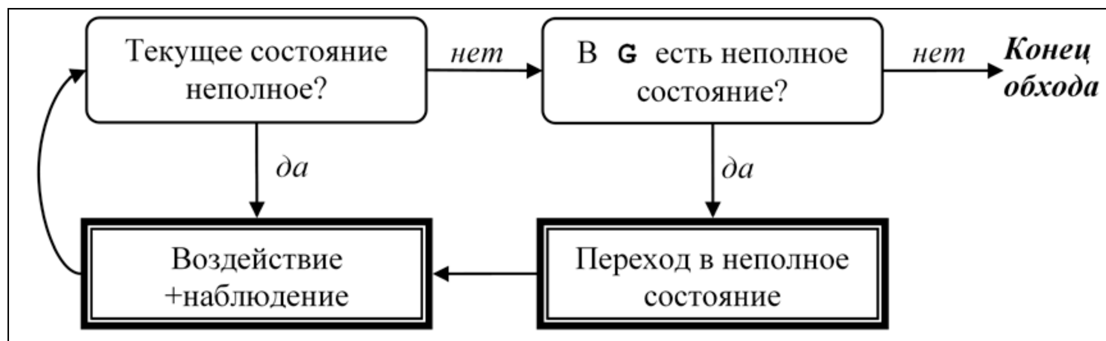

Рис.3. Общая схема алгоритма обхода

Теперь рассмотрим случай, когда текущее состояние оказалось полным. Это означает, что в этом состоянии мы выполнили нужное число раз каждое тестовое воздействие и получили все возможные наблюдения и постсостояния реализации. Чтобы продолжить обход, нам нужно перейти в любое неполное состояние. Если таких состояний нет, алгоритм заканчивается. В этом случае все пройденные состояния полны и, следовательно, все достижимые переходы пройдены.

Переход в неполное состояние выполняется следующей процедурой. В графе LTS **G** всегда существует лес деревьев, покрывающих все пройденные состояния и ориентированных к своим корням, которыми являются все неполные состояния. Выберем любой такой лес и пометим все его переходы. Для каждого полного состояния i обозначим через P(i) кнопку, связанную с помеченным переходом, выходящим из i. После этого начнём движение по **G**, нажимая в каждом текущем полном состоянии i кнопку P(i). Движение по помеченному переходу может не получиться из-за недетерминизма реализации: вместо помеченного перехода мы пройдём непомеченный переход.

Покажем, что через конечное число шагов мы окажемся в неполном состоянии. Доказательство будем вести от противного: пусть мы совершаем бесконечное число шагов, проходя только через полные состояния. Длину пути по лесу деревьев от состояния i до неполного состояния будем называть расстоянием r_i. Поскольку число состояний, тем более полных, конечно, через какие-то полные состояния мы будем проходить 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 помеченному переходу i—o→i`. Но тогда мы бесконечное число раз оказывались в состоянии i`, которое, однако, имеет меньшее расстояние $r_{i`}=r_i-1$, что противоречит минимальности расстояния r_i. Мы пришли к противоречию, что и требовалось.

4.3. Оценка числа тестовых воздействий

В общей схеме алгоритма на Рис.3 блоки, в которых выполняются тестовые воздействия, отмечены жирной рамкой. Обозначим: n – число состояний реализации, b – число кнопок, t – ограничение недетерминизма реализации.

В блоке «Воздействие+наблюдение» тестовое воздействие в каждом состоянии осуществляется не более bt раз (меньше, если наблюдаются отказы с тем же постсостоянием). Тем самым, в этом блоке осуществляется не более btn тестовых воздействий.

Рассмотрим работу в блоке «Переход в неполное состояние». Сначала оценим однократную работу блока. Обозначим через c число полных состояний во время работы блока. Рассмотрим все полные состояния, через которые мы проходили. Если такое состояние имеет расстояние $r=1$, то в силу ограниченности недетерминизма мы выходили из этого состояния не более t раз. Следовательно, из каждого состояния с расстоянием $r=2$ мы выходили не более t^2 раз. В общем, из состояния с расстоянием r мы выходили не более t^r раз. Обозначим через a_r число состояний с расстоянием r. Тогда число тестовых воздействий не превосходит:

$$\boldsymbol{f}(c) = \sum_r(a_r t^r) \le t+t^2+\ldots+t^c =$$
$$= \text{для } t>1\text{: } (t^{c+1}-t)/(t-1),$$
$$= \text{для } t=1\text{: } c.$$

Теперь оценим суммарно работу блока «Переход в неполное состояние» во время обхода. Перенумеруем все состояния в том порядке, в котором они становились полными. Рассмотрим момент времени, непосредственно предшествующий получению в i-ом состоянии j-ой тройки (кнопка, наблюдение, постсостояние). Обозначим через c_{ij} число полных состояний в этот момент времени. Если непосредственно перед получением j-ой тройки мы выполняли блок «Переход в неполное состояние», переходя в состояние i, то обозначим через y_{ij} число тестовых воздействий в этом блоке; в противном случае положим $y_{ij}=0$. 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, которые мы получаем в данном состоянии, не превосходит bt. Общая оценка числа тестовых воздействий в блоке равна:

$$\sum_{ij}(y_{ij}) \le \sum_{ij}(\boldsymbol{f}(c_{ij})) \le \sum_{ij}(\boldsymbol{f}(i-1)) \le \sum_i(bt\boldsymbol{f}(i-1)) =$$
$$= \text{для } t>1\text{: } \quad bt\sum_i((t^i-t)/(t-1)) = O(bt^n),$$
$$= \text{для } t=1\text{: } \quad b\sum_i(i-1) = O(bn^2).$$

Эта оценка достигается (по порядку) на LTS, изображённой на Рис.4. Здесь имеется только одно действие и одна кнопка, разрешающая его. Если недетерминизм реализации устроен так, что в процессе движения по LTS каждый переход, выделенный пунктиром, предваряется $t-1$ переходами в начальное состояние (выделены жирной стрелкой), нижняя оценка равна $O(t^n)$. Заметим, что на этом примере указанная оценка достигается при любом алгоритме обхода, а не только описанном выше.

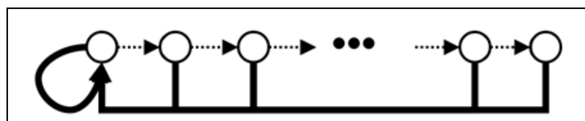

Рис.4. Пример реализации

4.4. Оценка объёма вычислений

Мы будем считать, что при опросе состояния мы получаем его уникальный идентификатор, но не предполагаем каких-либо знаний об устройстве этого идентификатора, то есть мы можем только сравнивать два идентификатора на равенство. Каждый раз, когда после тестового воздействия мы делаем опрос постсостояния, мы должны определить, является ли

это постсостояние новым или старым, и, если старым, то каким именно. Это требует $O(n)$ операций сравнения. Поэтому в любом случае объём вычислений по порядку в n раз превосходит длину обхода: для $t>1$: $O(nbt^n)$, для $t=1$: $O(bn^3)$. В этом подразделе мы убедимся, что при подходящей реализации алгоритма другие вычисления не изменяют этот порядок.

Прежде всего, отметим, что лес деревьев, требуемый для прохода в неполное состояние, может использоваться много раз без его перестройки до тех пор, пока какое-то неполное состояние не станет полным. Поэтому мы будем перестраивать этот лес только в этом случае.

Определим основные структуры данных. Будем считать, что все кнопки перенумерованы от 1 до b, и все пройденные состояния также пронумерованы. По номеру состояния i можно определить:

- $I(i)$ – идентификатор состояния, получаемый при опросе состояния;
- $B(i)$ – номер текущей кнопки, в полных состояниях $B(i)=b+1$, вначале $B(i)=1$;
- $C(i)$ – счётчик текущей кнопки, вначале $C(i)=0$;
- $T(i)$ – список различных номеров состояний, из которых в **G** выходит переход, заканчивающийся в i, вначале список $T(i)$ пуст;
- $P(i)$ – номер кнопки в лесе деревьев для достижения неполных состояний, начальное значение $P(i)=0$.

Отдельно хранится номер текущего состояния i_c.

Кроме этого, мы будем поддерживать счётчик N пройденных состояний (вначале $N=0$), счётчик полных состояний C (вначале $C=0$) и матрицу M размером $N\times N$, строками и столбцами которой являются номера состояний, а $M(i,j)$ содержит кнопку, связанную с каким-нибудь переходом $I(i)\text{—o}\rightarrow I(j)$, или 0, если такой кнопки нет.

Блок «Текущее состояние неполное?» проверяет, что $B(i_c)\leq b$. Такое сравнение – элементарная операция, а число сравнений равно числу переходов, которое имеет точную верхнюю оценку $O(btn)$.

В блоке «Воздействие+наблюдение» в текущем состоянии с номером i_c нажимается кнопка P и получается наблюдение o и постсостояние. Прежде всего, нам нужно определить номер постсостояния $i`$, что делается поиском по массиву состояний; если это новое состояние, то ему присваивается очередной номер $N+1$, создаётся его описание и расширяется матрица $M$, добавлением $N+1$-ых нулевых строки и столбца. Далее $N:=N+1$. Увеличиваем счётчик $C(i_c):=C(i_c)+1$. 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 P заносится в матрицу $M(i_c,i`):=P$. Меняется текущее состояние $i_c:=i`$.

Все описанные операции можно считать элементарными, кроме определения номера постсостояния, которое требует порядка $O(n)$ операций. Данный блок алгоритма работает не более btn раз, и суммарная оценка $O(btn^2)$.

Если состояние i_c становится полным $B(i_c)=b+1$, то $C:=C+1$. Теперь нужно перестроить лес деревьев. Для этого используется 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. Это даёт O(n) операций. Далее для головного элемента i списка L просматриваем список номеров состояний T(i) и для каждого состояния j из этого списка проверяем, принадлежит ли состояние j 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). Построение заканчивается, когда список L становится пуст. Число проверяемых переходов по порядку не превышает O(btn). Поэтому лес деревьев строится за O(n)+O(btn)=O(btn) операций. Мы строим лес деревьев каждый раз, 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 сложности работы блока «Воздействие+наблюдение» $O(btn^2)+O(btn^2)=O(btn^2)$.

Блок «В **G** есть неполное состояние?» выполняет проверку C≠N. Суммарно это даёт не более O(btn) операций.

В блоке «Переход в неполное состояние» для выполнения одного перехода нажимается кнопка $P(i_c)$ и опрашивается постсостояние. Здесь опять нам нужно определить номер постсостояния i`, что делается поиском по массиву состояний не более, чем за O(n) элементарных операций. Меняем текущее состояние $i_c$:=i` и проверяем полноту нового текущего состояния. Если оно неполно, то есть $B(i_c)<b+1$ (или $P(i_c)>0$), действия повторяются. Учитывая число тестовых воздействий в этом блоке $O(bt^n)$ для t>1 и $O(n^3)$ для t=1, 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.

4.5. Сильно-Δ-связные реализации

Описанный выше алгоритм обхода применим к любой конечной сильно-связной реализации с ограниченным недетерминизмом. При t=1 мы имеем детерминированный случай с полиномиальной 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 тестируемых реализаций. Одним из таких подклассов является класс, так называемых, сильно-Δ-связных LTS.

Сильно-Δ-связность была введена в [4] для решения задачи обхода графов переходов недетерминированных автоматов. Тестовое воздействие понималось как посылка стимула в реализацию, а наблюдение как получение реакции от реализации. Автомат не содержал непомеченных переходов (аналог τ-переходов в LTS). Мы переформулируем основные определения и утверждения этой работы в терминах LTS и $\mathfrak{R}$-семантики.

Для состояния s и кнопки P Δ-переходом (s,P) будем называть множество всех переходов s—z→s`, где z∈P. Δ-маршрутом будем называть множество маршрутов D, начинающихся в одном состоянии и «ветвящихся» в каждом проходимом Δ-переходе. Формально: для любого префикса любого маршрута из D, заканчивающегося в состоянии

s, существует такая кнопка P, что множество переходов, которыми этот префикс продолжается в D, совпадает с Δ-переходом (s,P). Начальное состояние a маршрутов Δ-маршрута будем называть началом Δ-маршрута. Если B – множество концов маршрутов Δ-маршрута, начинающегося в состоянии в состоянии a, такой Δ-маршрут называется [a,B]-Δ-маршрутом. Если B состоит из одной вершины b, то это [a,b]-Δ-маршрут. Длиной Δ-маршрута будем называть максимальную длину его маршрутов. Маршрут будем называть обходом по кнопкам, если он содержит крайней мере один переход из каждого непустого Δ-перехода. Δ-обходом будем называть Δ-маршрут, все маршруты которого являются обходами по кнопкам. Алгоритмом Δ-обхода называется такой алгоритм, который независимо от того или иного недетерминированного поведения LTS выполняет обход по кнопкам; суммарно по всем вариантам недетерминированного поведения LTS обходы по кнопкам, проходимые алгоритмом, образуют Δ-обход. Δ-путём называется Δ-маршрут, все маршруты которого являются путями. Состояние b Δ-достижимо из состояния a, если существует [a,b]-Δ-путь. LTS называется сильно-Δ-связной, если любое её состояние Δ-достижимо из любого другого состояния. Заметим, что Δ-достижимость мы также рассматриваем с учётом рестартов.

Нестабильное состояние не может быть Δ-достижимым. Поэтому из сильно-Δ-связности следует отсутствие τ-переходов в реализации.
Основное утверждение работы [4] переформулируется следующим образом: Существует 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, а m – число Δ-переходов. Для реализации с ограниченным недетерминизмом $m \le bn$, и оценки имеют вид $O(bn^2)$ и $O(bn^3)$, соответственно.

Для ограниченно недетерминированных сильно-Δ-связных реализаций можно модифицировать алгоритм Δ-обхода из [4] так, чтобы он стал алгоритмом обхода, то есть проходил по каждому переходу, а не по хотя бы одному переходу из каждого Δ-перехода. Этот алгоритм базировался на локальной аппроксимации Δ-расстояния состояния a до множества состояний B, где Δ-расстояние определялось как минимальная длина [a,B]-Δ-маршрута по пройденной LTS. В качестве такого множества B выбиралось множество состояний, в которых мы нажимали ещё не все кнопки, поскольку целью был обход по кнопкам: в каждом состоянии нужно было хотя бы один раз нажать каждую кнопку. Модификация сводится к тому, что в качестве множества B выбирается множество неполных состояний. Поскольку теперь мы должны нажимать в каждом состоянии каждую кнопку хотя бы t раз, оценки увеличиваются в t раз: $O(btn^2)$ для числа тестовых воздействий и $O(btn^3)$ для объёма вычислений.

5. Алгоритм тестирования

5.1. Безопасно-достижимая LTS

Обозначим: **I** – реализационная LTS, **S** – спецификационная LTS. В процессе тестирования мы будем строить безопасно-достижимую LTS, порождаемую не всеми маршрутами реализации, начинающимися в начальном состоянии, как для обхода, а только теми из них, трассы которых безопасны в спецификации. При тестировании мы будем совершать обход именно этой LTS. Если тестирование опирается на сильно-Δ-связность LTS,

то это свойство предъявляется не ко всей реализации, а к её безопасно-достижимой LTS. Пройденную при тестировании часть LTS, по-прежнему, будем обозначать **G**.

По-прежнему с каждым пройденным состоянием i и с каждой кнопки P свяжем счётчик c(P,i) числа нажатий кнопки P в состоянии i (3.1). Кроме того для каждого состояния i будем хранить множество **S**(i), элементами которого являются множества состояний спецификации. Множество S принадлежит **S**(i), если после наблюдения некоторой трассы σ мы оказались в реализационном состоянии i, а в спецификации трасса σ безопасна и заканчивается во множестве состояний S=(**S** ***after*** σ). Будем говорить, что кнопка P *допустима в состоянии* i, если она безопасна хотя бы в одном множестве состояний S∈**S**(i). Только допустимые кнопки мы можем нажимать, когда находимся в состоянии i. Определение полного состояния меняется, поскольку теперь требуется, чтобы не каждая, а каждая допустимая в состоянии кнопка была в нём полна.

Теперь реализация **I** и полностью построенная LTS **G** имеют одно и то же множество ***Safe***(**S**)∩***F***(**I**) $\mathfrak{R}$-трасс, безопасных в спецификации. Более того, они имеют одно и то же множество безопасно-тестируемых трасс, то есть трасс вида σ·⟨o⟩, где σ∈***Safe***(**S**)∩***F***(**I**), а наблюдение o разрешается некоторой кнопкой P ***safe*** **S** ***after*** σ. Поэтому **I** ***saco*** **S** ⇔ **G** ***saco*** **S**. Также LTS **I** и **G** имеют одно и то же множество состояний, достижимых в реализации по безопасно-тестируемым трассам.

5.2. Начало работы алгоритма

В начале тестирования в **G** у нас есть только одно состояние i∈{**I** ***after*** ϵ}, соответствующее множеству состояний спецификации после пустой трассы ϵ. Полагаем **S**(i)={**S** ***after*** ϵ}. В состоянии i допустимы только те кнопки, которые допустимы в спецификации после пустой трассы, для каждой такой кнопки P полагаем c(P,i):=0.

Если мы не предполагаем стабильности начального состояния (в частности, сильно-Δ-связности реализации), нам нужно в процессе тестирования получить все состояния реализации из множества **I** ***after*** ϵ. Для этого необходимо и достаточно, чтобы рестарт был определён хотя бы в одном состоянии, достижимом по безопасной трассе спецификации. Если начальное состояние реализации нестабильно, и рестарт не определён в безопасно-достижимых состояниях, то условие сильно-связности не гарантирует полноты тестирования (в отличие от обхода). Пример приведён на Рис.5. Здесь реализация не конформна спецификации, так как в ней с самого начала может наблюдаться действие y, что запрещено спецификацией (может быть только отказ {y}). Если в начале тестирования мы оказываемся в состоянии реализации 1, то при отсутствии рестарта в состояние 0 мы можем попасть только после нажатия кнопки {x}. А согласно спецификации после любой трассы, содержащей x, кнопка {y} опасна, и мы не будем её нажимать. Следовательно, мы не обнаружим наличие действия y в состоянии 0, то есть не обнаружим ошибку.

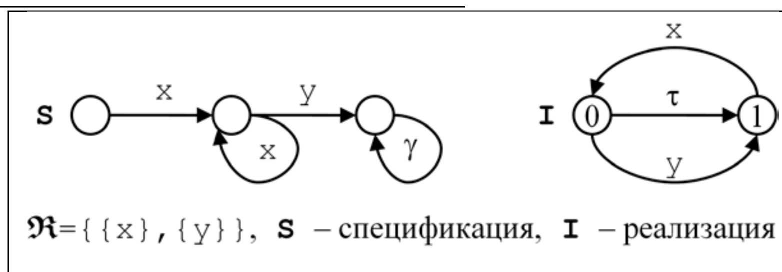

Рис.5. Пример спецификации и неконформной реализации

5.3. Общая схема алгоритма

На Рис.6 изображена общая схема алгоритма. В отличие от схемы обхода на Рис.3 здесь добавляются три блока, обведённые широкой серой линией. Кроме того, в блоке «Воздействие+наблюдение» мы нажимаем не любую кнопку, которая не полна в текущем состоянии, а только допустимую. Этим гарантируется, что при тестировании проходятся только трассы, безопасные 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.

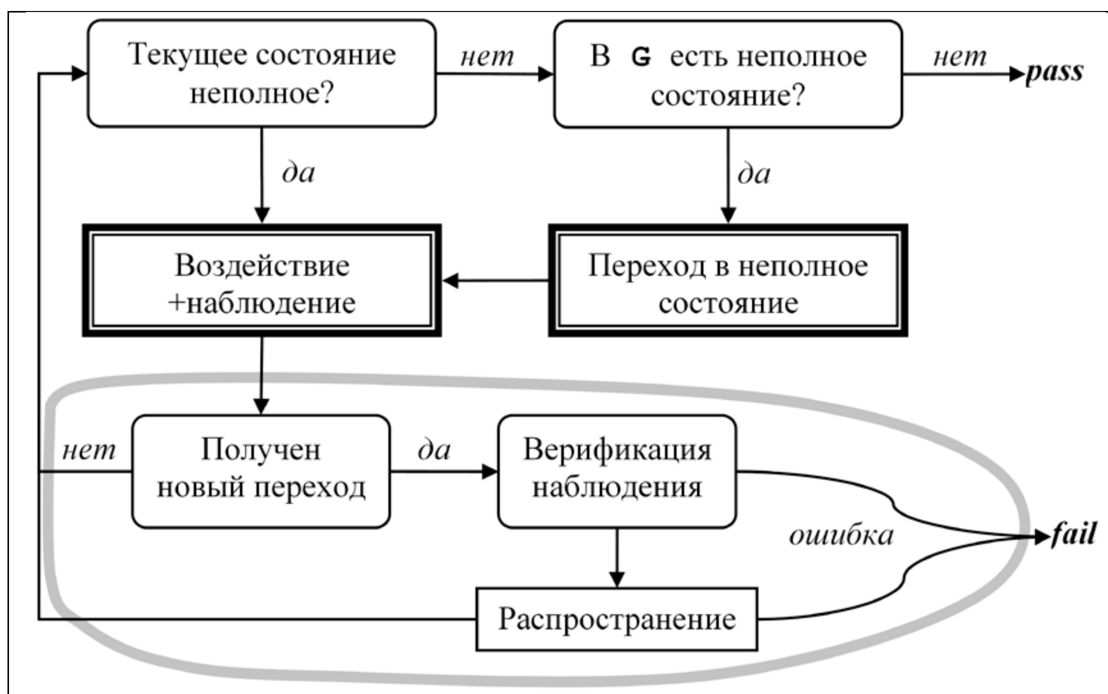

Рис.6. Общая схема алгоритма

Определим процедуру ***Post***(i—o→i`,S), которая совмещает функции верификации (оракул) и вычисление множества спецификационных постсостояний. Здесь i—o→i` – это переход LTS **G**, а S∈**S**(i). Если фиксируется ошибка (неконформность), то тестирование заканчивается с вердиктом ***fail***. Иначе вычисленное множество спецификационных 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) o – внешнее действие. Если существует кнопка P ***safe*** S такая, что o∈P, то вычисляется множество S`, состоящее из всех состояний s`, которые достижимы в

спецификации по τ-переходам из концов переходов вида s—z→s`, где $s∈S$. Если $S$`=∅, то это означает, что $o∉$***obs***(S,P), и процедура фиксирует ошибку. Если такая кнопка P не существует, то ничего не делается.

2) $o=P$ – отказ (им помечен виртуальный переход). Если P ***safe*** S, то вычисляется множество S`, состоящее из всех состояний $s$`∈S, в которых имеется отказ P. Если S`=∅, то это означает, что $o∉$***obs***(S,P), и процедура фиксирует ошибку. Если P ~~***safe***~~ S, то ничего не делается.
3) $o=τ$ – внутреннее действие. Вычисляется S`=S, ошибка не фиксируется.

Блок «Верификация наблюдения» работает после того, как в блоке «Воздействие+наблюдение» мы добавили *новый* (возможно, виртуальный) переход i—o→i`, где наблюдение $o$ отлично от рестарта и отказа по рестарту. (Если мы наблюдаем отказ по рестарту в другом состоянии $i$`≠i, то, как описано выше, добавляется переход i—τ→i`.) Для каждого множества состояний $S$∈**S**($i$) вызывается процедура ***Post***($i$—$o$→$i$`,S). Если проверка завершена успешно, выполняется блок «Распространение». В случае наблюдения рестарта постсостояние – это начально-достижимое состояние, поэтому множество **S** ***after*** ϵ добавляется к **S**(i`) при условии, что его там не было, и помечается в **S**($i$`) флагом «добавленное».

В блоке «Распространение» вызываем процедуру ***Post***(i—o→i`,$S$) для каждого уже имеющегося перехода $i$—$o$→$i$`, где наблюдение o отлично от рестарта и отказа по рестарту, и *добавленного* множества состояний S∈**S**(i), после чего снимаем с множества S в **S**(i), флаг «добавленное». Эти действия выполняются до тех пор, пока есть добавленные множества состояний. Поскольку число состояний реализации конечно, а также конечно число состояний спецификации и, следовательно, число множеств состояний спецификации, блок «Распространение» закончится за конечное время.

Важно отметить, что в этих двух блоках происходит не только верификация наблюдения, полученного после *реальной* трассы, пройденной при тестировании, но также после возможных наблюдений после *потенциальных* трасс. Более точно: верифицируются наблюдения, про которые установлено, что они возможны в реализации после трасс, про которые известно, что они безопасны в спецификации и имеются в реализации. 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 исчерпывающим.

Сначала покажем, что для каждого состояния i LTS **G** каждое множество S∈**S**(i) является концом некоторой трассы, безопасной в спецификации: ∃σ∈***Safe***(**S**)∩***F***(**I**) S=(**S** ***after*** σ) & i∈(**I** ***after*** σ). Будем вести доказательство индукцией по последовательности добавлений множества S∉**S**(i) для любого S и любого i: **S**(i):=**S**(i)∪S. 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=ϵ. Процедура ***Post***(i—o→i`,$S$) добавляет в **S**($i$`)

множество постсостояний $S`$ для перехода $i\text{—}o\rightarrow i`$ таким образом, что, если $\exists\sigma\in \textbf{\textit{Safe}}(\mathbf{S})\cap \textbf{\textit{F}}(\mathbf{I})$ $S=(\mathbf{S}\ \textbf{\textit{after}}\ \sigma)$ & $i\in(\mathbf{I}\ \textbf{\textit{after}}\ \sigma)$, то для $o\neq\tau$ имеем $o\ \textbf{\textit{safe}}\ (\mathbf{S}\ \textbf{\textit{after}}\ \sigma)$ & $S`=(\mathbf{S}\ \textbf{\textit{after}}\ \sigma\cdot\langle o\rangle)$ & $i`\in(\mathbf{I}\ \textbf{\textit{after}}\ \sigma\cdot\langle o\rangle)$, а для $o=\tau$ имеем $S`=S$ & $i`\in(\mathbf{I}\ \textbf{\textit{after}}\ \sigma)$. Утверждение доказано.

Значимость эквивалентна тому, что каждая ошибка, обнаруживаемая тестом, действительно говорит о неконформности реализации. С учётом доказанного утверждения вердикт ***fail*** выносится в процедуре ***Post*** только тогда, когда обнаруживается, что после некоторой трассы $\sigma\in \textbf{\textit{Safe}}(\mathbf{S})\cap \textbf{\textit{F}}(\mathbf{I})$ некоторая кнопка $P\ \textbf{\textit{safe}}\ \mathbf{S}\ \textbf{\textit{after}}\ \sigma$ разрешает наблюдение $o\in \textbf{\textit{obs}}(\mathbf{I}\ \textbf{\textit{after}}\ \sigma,P)$, отсутствующее в спецификации $o\notin \textbf{\textit{obs}}(\mathbf{S}\ \textbf{\textit{after}}\ \sigma,P)$, то есть когда реализация неконформна.

Исчерпываемость означает, что, если тест заканчивается с вердиктом ***pass***, то реализация конформна. Нам достаточно показать, что для каждой трассы $\sigma\in \textbf{\textit{Safe}}(\mathbf{S})\cap \textbf{\textit{F}}(\mathbf{I})$ после завершения теста каждое состояние $i\in(\mathbf{I}\ \textbf{\textit{after}}\ \sigma)$ имеется в LTS $\mathbf{G}$ и $(\mathbf{S}\ \textbf{\textit{after}}\ \sigma)\in\mathbf{S}(i)$. Будем вести доказательство индукцией по трассам. Для пустой трассы ϵ утверждение верно, поскольку в начале тестирования или после одного из рестартов мы добавляем $\mathbf{S}\ \textbf{\textit{after}}\ \epsilon$ во множество $\mathbf{S}(i)$ для каждого состояния $i\in(\mathbf{I}\ \textbf{\textit{after}}\ \epsilon)$.

Рассмотрим непустую трассу $\sigma\cdot\langle o\rangle\in \textbf{\textit{Safe}}(\mathbf{S})\cap \textbf{\textit{F}}(\mathbf{I})$, где наблюдение o отлично от рестарта или отказа по рестарту и разрешается некоторой кнопкой $P\ \textbf{\textit{safe}}\ \mathbf{S}\ \textbf{\textit{after}}\ \sigma$. Пусть утверждение верно для трассы σ, и докажем его для трассы $\sigma\cdot\langle o\rangle$. Пусть состояние $i`\in(\mathbf{I}\ \textbf{\textit{after}}\ \sigma\cdot\langle o\rangle)$. Тогда в реализации есть состояние $i\in(\mathbf{I}\ \textbf{\textit{after}}\ \sigma)$ и переход $i\text{—}o\rightarrow i`$ (если o отказ, то это виртуальная петля). По предположению шага индукции, состояние i имеется в $\mathbf{G}$ и $(\mathbf{S}\ \textbf{\textit{after}}\ \sigma)\in\mathbf{S}(i)$. Поскольку в конце тестирования с вердиктом ***pass*** все состояния полны, мы должны были в состоянии i нажимать кнопку P и после некоторого нажатия получить переход $i\text{—}o\rightarrow i`$. Следовательно, процедура ***Post*** должна была выполняться с параметрами $(i\text{—}o\rightarrow i`,S)$. А тогда $(\mathbf{S}\ \textbf{\textit{after}}\ \sigma\cdot\langle o\rangle)\in\mathbf{S}(i`)$, что и требовалось доказать.

5.4. Оценка числа тестовых воздействий

В отличие от обхода, при тестировании нажимаются только те кнопки, которые допустимы в текущем состоянии i. Допустимость кнопки в состоянии i зависит от трассы. Может случиться так, что кнопка P недопустима в состоянии i настолько долгое время, что все допустимые кнопки уже нажимались нужное число раз, то есть состояние i стало полным. Однако потом, после получения (реальной или потенциальной) трассы σ такой, что $P\ \textbf{\textit{safe}}\ \mathbf{S}\ \textbf{\textit{after}}\ \sigma$ и $i\in(\mathbf{I}\ \textbf{\textit{after}}\ \sigma)$, кнопка P становится допустимой в состоянии i, то есть оно становится снова неполным.

Теперь мы не можем считать, что состояние, ставшее полным, остаётся таким до конца тестирования. Поэтому 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. Мы дадим оценку на основе числа вызовов блока. После прохода в неполное состояние мы нажимаем

кнопку, которую ещё не нажимали нужное число раз в этом состоянии. Поэтому блок вызывается не более btn раз, и каждый раз выполняется не более $\boldsymbol{f}(n-1)$ 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. Мы высказываем гипотезу о том, что эта оценка завышена, а точная оценка остаётся той же $O(bt^n)$. Эта гипотеза верна для $b=1$. Достаточно рассмотреть состояние i_n, которое последним из всех состояний становится полным. В состоянии, находящемся на расстоянии r от состояния i_n, мы должны были быть не более t^r раз. Обозначим через a_r число таких состояний. Суммарно во всех состояниях мы должны были быть не более $\sum_r(a_r t^r) \le t+t^2+t^3+\ldots+t^n = O(t^n)$ раз, что совпадает с числом тестовых воздействий.

Также эту гипотезу мы можем доказать для случая, когда рестарт определён в каждом (достижимом по безопасным трассам спецификации) состоянии реализации. Переход в неполное состояние выполняется как рестарт и далее переход в неполное состояние из начального состояния (если начальное состояние нестабильно, то может потребоваться несколько рестартов подряд, но это не увеличивает, а скорее уменьшает оценку). Выберем дерево, ориентированное от начального состояния и содержащее все пройденные состояния. Мы будем двигаться по дереву, нажимая соответствующие кнопки. Каждый раз, когда окажется, что мы не проходим нужный переход дерева из-за недетерминизма, мы 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, нам нужно по порядку не более t^r тестовых воздействий. Суммарно для прохода по одному разу в каждое состояние потребуется число тестовых воздействий по порядку не более $\sum_r(a_r t^r) \le t+t^2+\ldots+t^{n-1} = O(t^{n-1})$. Поскольку в каждое состояние мы должны переходить не более bt раз, общая оценка $O(bt^n)$.

5.5. Оценка объёма вычислений

Отличия от обхода следующие: 1) большее число тестовых воздействий, 2) большее число раз строится лес деревьев, 3) работа дополнительных блоков алгоритма.

1) Число тестовых воздействий умножается на n для оценки числа операций, требуемых для поиска номера состояния по его идентификатору. Суммарная оценка: для $t>1$: $O(bn^2t^n)$ или $O(bnt^n)$, если есть рестарты, и $O(bn^3)$ для $t=1$.
2) Лес деревьев теперь может строиться больше, чем n раз, но не более bn раз, так как каждое состояние переходит из неполного состояния в полное, при завершении нажатий некоторой кнопки. Суммарная оценка увеличивается в b раз: $O(b^2tn^2)$.
3) Основные вычисления в дополнительных блоках происходят в процедуре ***Post***$(i\text{—}o\rightarrow i`,S)$, однократная работа которой требует константного числа операций. Можно привести пример спецификации с k состояниями и реализации с n состояниями такой, что в LTS **G** появятся все возможные пары (i,S), то есть $n2^k$ пар. Для каждой такой пары проверяется не более bt переходов. Суммарная оценка $O(btn2^k)$.

Нужно отметить, что все проверки, которые выполняет процедура ***Post***, необходимы для верификации конформности: это проверки всех наблюдений, возможных в реализации после всех безопасных трасс спецификации.

Общий объём вычислений алгоритма: $O(bn^2t^n)+O(b^2tn^2)+O(btn2^k)$, или первое слагаемое заменяется на $O(bnt^n)$, если есть рестарты, или $O(bn^3)$ для $t=1$.

5.6. Сильно-Δ-связные реализации

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

Для трёх случаев, описанных в предыдущем подразделе, имеем следующие оценки объёма вычислений.

1) Число тестовых воздействий умножается на n для оценки числа операций, требуемых для поиска номера состояния по его идентификатору. Суммарная оценка: $O(btn^3)$.
2) При вычислении Δ-расстояний нам достаточно для каждого состояния отметить Δ-переход, по которому достигается минимальное Δ-расстояние, то есть отметить кнопку. Такое вычисление аналогично построению леса деревьев и требует число операций по порядку не более числа переходов, то есть не более $O(btn)$. Мы должны перевычислять Δ-расстояния каждый раз, когда неполное состояние становится полным. Число состояний n. Поскольку состояние становится полным, когда выполняются все нужные переходы по некоторой кнопке, это состояние становится полным не более b раз. Суммарная оценка $O(b^2tn^2)$.
3) Вычисления в процедуре ***Post***, как и в общем случае, занимают порядка $O(btn2^k)$ 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. Тестирование компонентов распределенной системы. Труды Всероссийской научной конференции «Научный сервис в сети ИНТЕРНЕТ», Изд-во МГУ, 2005.

[9] Бурдонов И.Б., Косачев А.С. Верификация композиции распределенной системы. Труды Всероссийской научной конференции «Научный сервис в сети ИНТЕРНЕТ», Изд-во МГУ, 2005.
[10] Bourdonov I., Kossatchev A., Kuliamin V. Formal Conformance Testing of Systems with Refused Inputs and Forbidden Actions. Proc. Of MBT 2006, Vienna, Austria, March 2006.
[11] Бурдонов И.Б., Косачев А.С., Кулямин В.В. Формализация тестового эксперимента. «Программирование», 2007, No. 5.
[12] 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.
[13] Бурдонов И.Б., Косачев А.С., Кулямин В.В. Теория соответствия для систем с блокировками и разрушением. «Наука», 2008.
[14] 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.
[15] http://www.ispras.ru/~RedVerst/RedVerst/Publications/TR-01-2007.pdf
[16] Василевский М.П. О распознавании неисправностей автомата. Кибернетика, т. 9, № 4, стр. 93–108, 1973.
[17] 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.
[18] 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.
[19] Edmonds J. Johnson E.L. Matching. Euler Tours, and the Chinese Postman. Math. Programming 5, 88-124 (1973).
[20] 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.
[21] 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.
[22] 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.
[23] 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.
[24] Grochtmann M., Grimm K. Classification trees for partition testing. Software Testing, Verification and Reliability, 3:63-82, 1993.
[25] Heerink L., Tretmans J. Refusal Testing for Classes of Transition Systems with Inputs and Outputs. In T.Mizuno, N.Shiratori, T.Higashino, A.Togashi, eds. Formal Description Techniques and Protocol Specification, Testing and Verification. Chapman & Hill, 1997.
[26] Heerink L. Ins and Outs in Refusal Testing. PhD thesis, University of Twente, Enschede, The Netherlands, 1998.
[27] 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.
[28] Lee D., Yannakakis M. Principles and Methods of Testing Finite State Machines – A Survey. Proceedings of the IEEE 84, No. 8, 1090–1123, 1996.

[29] 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.
[30] Lestiennes G., Gaudel M.-C. Test de systemes reactifs non receptifs. Journal Europeen des Systemes Automatises, Modelisation des Systemes Reactifs, pp. 255–270. Hermes, 2005.
[31] Milner R. A Calculus of Communicating Processes. LNCS, vol. 92, Springer-Verlag, 1980.
[32] Milner R. Modal characterization of observable machine behaviour. In G. Astesiano & C. Bohm, editors: Proceedings CAAP 81, LNCS 112, Springer, pp. 25-34.
[33] Milner R. Communication and Concurrency. Prentice-Hall, 1989.
[34] 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.
[35] Petrenko A., Yevtushenko N., Huo J.L. Testing Transition Systems with Input and Output Testers. Proc. IFIP TC6/WG6.1 15th Int. Conf. Testing of Communicating Systems, TestCom’2003, pp. 129-145. Sophia Antipolis, France, May 26-29, 2003.
[36] Zhu, Hall, May. Software unit test coverage and adequacy. ACM Computing Surveys, v. 29, n. 4, 1997.