И. Бурдонов, А. Косачев
Тестирование соответствия (conformance testing)
ИСПРАН
239 слайдов

Сейчас мы рассмотрим один широко распространённый частный случай, для которого стирается различие между синхронным и асинхронным тестированием. Это автоматы Мили. Кроме того, на этом примере можно продемонстрировать общую структуру тестовой системы, способы извлечения модели из спецификации, генерации тестов и методы тестирования.

АВТОМАТ МИЛИ

ВВЕДЕНИЕ

СТРУКТУРА ТЕСТОВОЙ СИСТЕМЫ

ТЕСТИРОВАНИЕ С ОТКРЫТЫМ СОСТОЯНИЕМ

МЕДИАТОРЫ

ФАКТОРИЗАЦИЯ

СЛАБЫЙ ДЕТЕРМИНИЗМ

НЕДЕТЕРМИНИЗМ

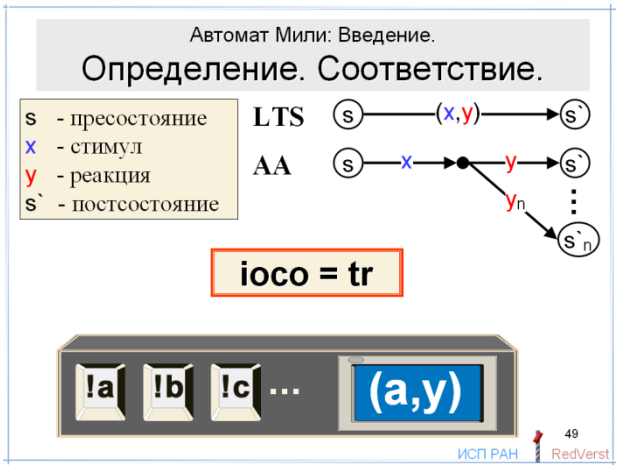
Определение и соответствие.

Хотя автомат Мили является частным случаем асинхронного автомата, он появился намного раньше и давно используется. В автомате Мили на каждый стимул выдаётся ровно одна реакция так, что любая трасса оказывается строго чередующейся последовательностью стимулов и ответных реакций. В таком автомате на переходе написан не один стимул или одна реакция, а всегда пара <стимул-реакция>, то есть это LTS без τ-переходов, алфавит которой – декартовое произведение множества стимулов на множество реакций.

Автомат Мили можно преобразовать в асинхронный автомат, если такой переход заменить последовательностью из двух переходов: сначала по стимулу из пресостояния в промежуточное состояние, потом по реакции из промежуточного состояния в постсостояние.

Мы будем считать, что промежуточное состояние одно для всех переходов из данного пресостояния по данному стимулу.

Если мы говорим, что наша модель – это автомат Мили, значит, автоматом Мили является как спецификационная, так и реализационная модель. Если автомат Мили рассматривать как частный случай более общей модели асинхронного автомата, то для спецификации просто проверяется, что она есть автомат Мили, а для реализации мы имеем реализационную гипотезу: реализационный автомат – это не любой асинхронный автомат, а автомат Мили.



Все состояния асинхронного автомата, представляющего автомат Мили, разделяются на стационарные, в которых только принимаются стимулы, и посылающие, в каждом из которых только выдаются реакции. Переход по стимулу всегда ведёт из стационарного состояния в посылающее, а переход по реакции – из посылающего в стационарное.

Поэтому для автомата Мили отношение **ioco** оказывается эквивалентным отношению трассовой сводимости **tr**. Нам не нужен тайм-аут при ожидании реакций, поскольку заранее известно, что после стимула будет реакция, а после реакции будет стационарное состояние. Алгоритм генерации полного тестового набора для **ioco** легко модифицируется – в сторону упрощения – для трассовой сводимости.

Для автомата Мили, конечно, можно использовать общую машину тестирования асинхронного автомата без встроенных тайм-аутов (машина **A**). Однако, поскольку передача стимулов и приём реакций строго чередуются, удобнее совместить последовательное нажатие кнопки стимула и кнопки приёма всех реакций.

Тогда у нас будет для каждого стимула одна кнопка, нажатие которой означает передать этот стимул и принять реакцию. В терминах автомата Мили как LTS это означает, что на кнопке написано множество символов вида (a,y), где стимул a фиксирован, а реакция y пробегает всё множество реакций.

При нажатии такой кнопки, если стимул принимается, то высвечивается пара (a,y), где a – посланный стимул, а y – принятая реакция.

Если стимул не определён в данном состоянии, кнопка остаётся утопленной, но на дисплей ничего не высвечивается.

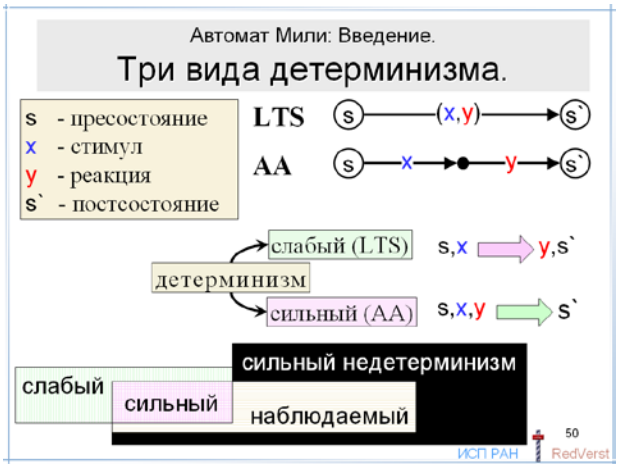
Дальше мы рассмотрим различные стратегии тестирования трассовой сводимости для конечного автомата Мили. Мы будем двигаться от простого к сложному, начиная с довольно сильных реализационных гипотез и/или сильных тестовых возможностей и простого устройства тестовой системы, и постепенно снимая ограничения на реализацию, довольствуясь меньшими тестовыми возможностями и усложняя структуру тестовой системы.

Три вида детерминизма.

Для автомата Мили можно дать два определения детерминизма: сильное и слабое аналогично сильному и слабому детерминизму для асинхронных автоматов.

Автомат *сильно детерминирован*, если пресостояние и стимул однозначно определяют реакцию и постсостояние. В таком автомате в каждом состоянии определено не более одного перехода по каждому стимулу.

В *слабо детерминированном* автомате Мили может быть несколько переходов из одного пресостояния по одному стимулу, различающихся реакциями, но тройка пресостояние, стимул и реакция однозначно определяют постсостояние.

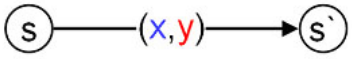


Очевидно, что сильный детерминизм влечёт слабый детерминизм и наблюдаемый детерминизм. Обратное, вообще говоря, не верно. Если реализация слабо, но не сильно, детерминирована, то она наблюдаемо недетерминирована.

Сильно недетерминированным будем называть автомат, который не является слабо детерминированным. Такой автомат может быть наблюдаемо детерминированным, то есть пара пресостояние и стимул однозначно определяют реакцию, но неоднозначно определяют постсостояние. Здесь важно, что даже тройка пресостояние, стимул и реакция неоднозначно определяют постсостояние.

Спецификация в пред- и постусловиях.

Одной из наиболее адекватных и удобных для человека формой спецификации автомата Мили является спецификация в пред- и постусловиях. Предусловие – это предикат от пресостояния и стимула, который описывает, в каких состояниях какие стимулы определены в спецификационном автомате. Постусловие – это предикат от пресостояния, стимула, реакции и постсостояния, который в предположении, что предусловие выполнено, говорит, какие реакции могут быть в ответ на этот стимул, поданный на автомат в этом пресостоянии, и в какие постсостояния может перейти автомат.



$$\mathbf{PRE}(s,x) = \text{true} \qquad \mathbf{POST}(s,x,y,s`) = \text{true}$$

Такая спецификация описывает автомат неявно. Чтобы узнать, какие переходы определены в данном пресостоянии, нужно решить уравнение **PRE**`(пресостояние,стимул)=`**`true`** относительно стимула. А чтобы после этого узнать, какие могут быть реакции и постсостояния, нужно решить уравнение
POST`(пресостояние,стимул,реакция,постсостояние)=`**`true`** относительно реакции и постсостояния. Понятно, что решать подобные уравнения в общем случае очень трудно. Для примера, который мы посмотрим на следующем слайд, всё очень просто, но что делать, если постусловие вот такое:

POST: $\exists a,b,c \in$ **Натуральные** $a^{\text{постсостояние}}+b^{\text{постсостояние}}=c^{\text{постсостояние}}$.

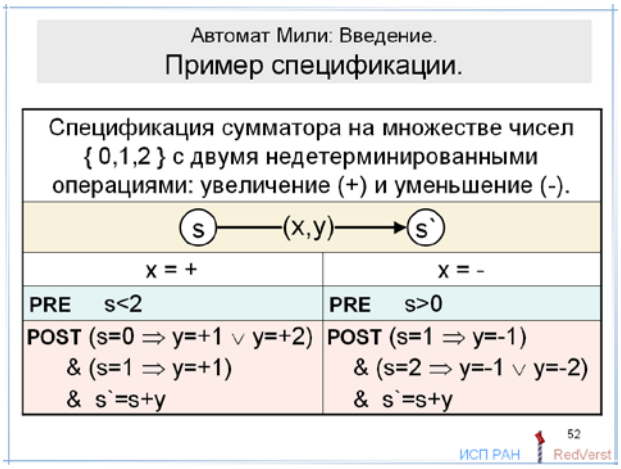
Пример спецификации.

В качестве примера рассмотрим сумматор над ограниченным множеством чисел `0`, `1` и `2` с двумя недетерминированными операциями: ***увеличение*** и ***уменьшение***, которые на слайде обозначены как **+** и **–**.

Состоянием является содержимое сумматора: `0`, `1` или `2`.

При ***увеличении*** содержание сумматора строго увеличивается; эта операция имеет предусловие: состояние должно быть меньше максимального числа `2`.

Соответственно, при ***уменьшении*** содержание сумматора строго уменьшается; эта операция имеет предусловие: состояние должно быть больше минимального числа `0`.

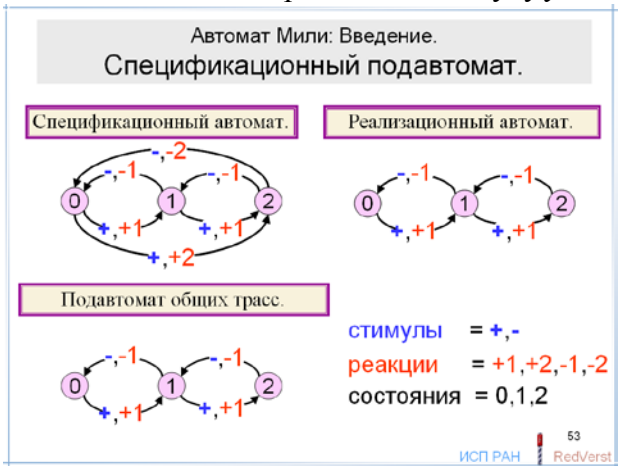


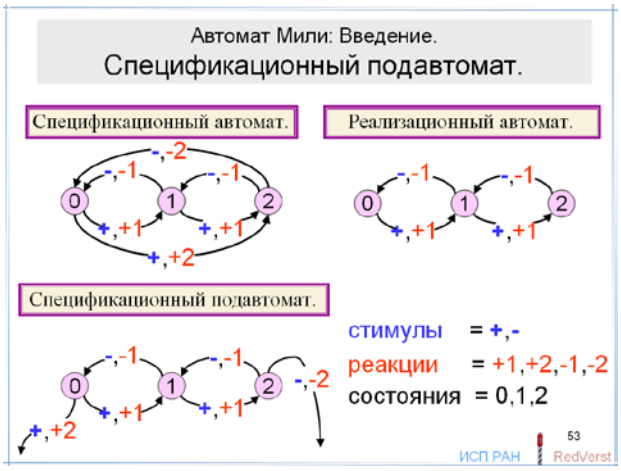
Каждая операция возвращает величину, на которую изменился сумматор.

Спецификационный подавтомат.

Теперь мы рассмотрим проблему извлечения модели из спецификации. К счастью, для этого во многих случаях мы можем обойтись без решения или почти без решения уравнений пред- и постусловия. Идея заключается в том, чтобы извлекать модель из спецификации в процессе тестирования, и делать это автоматически. Модель, которую мы будем извлекать из спецификации, не будет полностью описывать спецификационный автомат. Это будет подавтомат, зависящий от реализации. А именно: та часть спецификационного автомата, которая необходима и достаточна для проверки трассовой сводимости данной реализации.

Для нашего примера спецификационный автомат, если его извлечь из пред- и постусловий, выглядит следующим образом. В состоянии `0` есть два перехода по стимулу ***увеличение***, а в состоянии `2` – два перехода по стимулу ***уменьшение.***





Пример реализации выбран такой, чтобы сумматор всегда изменялся ровно на `1`. Реализационный автомат отличается от спецификационного автомата отсутствием переходов, связанных с изменением сумматора на `2`. Хотя этот автомат не всюду определён по стимулам, запрет блокирующего deadlock`a сохраняется: после любой общей трассы реализация принимает все стимулы, которые определены в спецификации после этой трассы.

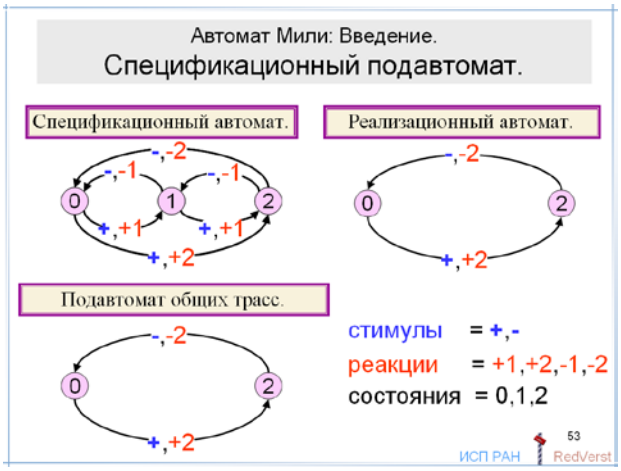
Очевидно, при тестировании, пока не обнаружена ошибка, могут проходиться только общие трассы реализации и спецификации. Ошибка заключается в том, что после некоторой общей

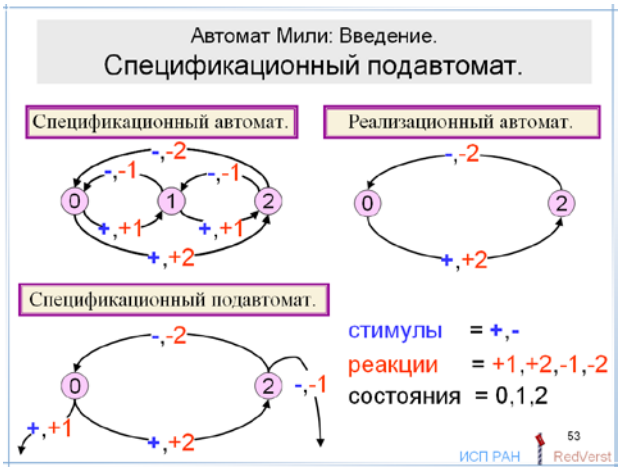
трассы σ и стимула x, которым трасса σ может продолжаться в спецификации, реализация возвращает запрещённую спецификацией реакцию y, то есть в спецификации нет трассы σ(x,y). Чтобы проверить реакцию, нам нужно знать все переходы спецификации по стимулу x, определённые во всех спецификационных состояниях, достижимых по трассе σ.

Поэтому спецификационный подавтомат определяется множеством состояний спецификации, достижимых по общим трассам, и всеми выходящими из них переходами. Если через пресостояние перехода проходит общая трасса, но никакая общая трасса не проходит через сам переход, то такой переход будем называть висящим. Висящие переходы не могут проходиться при тестировании, но они добавляются для проверки конформности.

В нашем примере есть два висящих перехода: это увеличение и уменьшение сумматора на 2. Напомним, что выделение того или иного подавтомата зависит от реализации.

Например, рассмотрим реализацию, которая изменяет сумматор на максимально возможную величину.





Мы получаем другой подавтомат общих трасс и, соответственно, другой спецификационный подавтомат. Здесь нет переходов по изменению сумматора на 1 в состояниях 0 и 2, эти переходы становятся висящими.

Мы ставим задачу построения спецификационного подавтомата в процессе тестирования. Разумеется, для того, чтобы в конечном тестовом эксперименте можно было выделить спецификационный подавтомат, этот подавтомат должен быть *конечным*: в нём должно быть конечное число состояний и переходов. Формально нам не требуется конечность всего спецификационного или реализационного автомата.

В нашем примере мы могли бы, не меняя функциональных требований, сделать спецификацию бесконечной – она бесконечно продолжалась бы после висящих переходов, то есть после реакций, отсутствующих в реализации. Правда, число таких висящих переходов, да и вообще всех переходов, определённых в каждом состоянии спецификационного подавтомата, должно быть конечно. Соответственно, реализацию также легко сделать бесконечной, добавив в неё новые стимулы, отсутствующие в спецификации; реализация бесконечно продолжалась бы после переходов по этим стимулам. Таких стимулов может быть бесконечно много – всё равно они нас не интересуют.

Четыре компонента тестовой системы.

Для определения стимулов, которые нам нужно подавать на реализацию в известном пресостоянии, мы можем осуществлять перебор всех стимулов из алфавита стимулов и каждый проверять на *предусловие*. Это фильтрация по стимулам. Программа, которая осуществляет такой перебор с фильтрацией, называется *итератором стимулов*.

В нашем примере итератор стимулов перебирает всего два стимула: **+** и **-**.

Реакцию нам вообще не нужно вычислять, поскольку при тестировании мы получаем её от реализации в ответ на стимул. Уж какая реакция пришла – такая пришла; тест ждёт всех реакций.

Другое дело постсостояние – его нужно определить. Во-первых, чтобы проверить постусловие, и, во-вторых, чтобы использовать как пресостояние при подаче следующего стимула. Компонент тестовой системы, который определяет постсостояние, называется *генератором постсостояния*. В общем виде он имеет вид:

```
постсостояние = функция_от(пресостояние,стимул,реакция).
```

Конечно, написать такую функцию – это то же самое, что решить уравнение постусловия относительно постсостояния. Для недетерминированной (ни сильно, ни слабо) спецификации эта функция будет к тому же многозначна, то есть генератор должен вернуть множество возможных постсостояний.

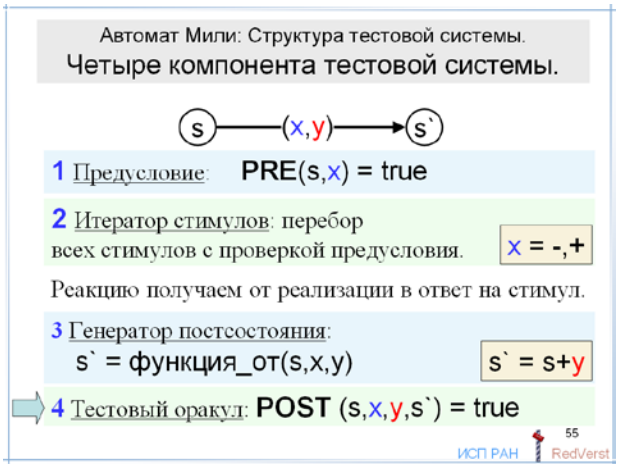
Однако в некоторых случаях такую функцию написать можно. В частности, если спецификационный автомат детерминирован, хотя бы слабо, то постсостояние определяется однозначно, и обычно не составляет труда определить эту функцию уже при написании постусловия. Тогда генератор постсостояния пишется легко, его даже можно генерировать из спецификации. Другой вариант опирается на возможность наблюдения состояния реализации, но об этом мы поговорим позже.

В нашем примере спецификация слабо детерминирована, а постсостояние вычисляется предельно просто: оно равно сумме пресостояния и реакции.

Заметим, что генератор постсостояния не проверяет, что при заданных пресостоянии, стимуле и реакции такое постсостояние будет удовлетворять постусловию. Эту проверку делает компонент, который называется *тестовым оракулом*. Если постусловие ложно, то это квалифицируется как ошибка реализации; в противном случае тестирование продолжается, а постсостояние становится новым пресостоянием спецификации.

На нашем примере это хорошо видно.

Если возможных постсостояний несколько, постусловие отфильтровывает их. Ошибка фиксируется, если ни одно из возможных постсостояний не удовлетворяет постусловию. Если оказывается, что несколько возможных постсостояний удовлетворяют постусловию, то возникает проблема недетерминизма: мы не знает, какое постсостояние выбрать в качестве текущего для продолжения тестирования.



Три проблемы перечисления.

Прежде чем двигаться дальше, отметим три проблемы перечисления, которые возникают при тестировании. Это перечисление стимулов, постсостояний и реакций, которое нужно выполнять в состоянии спецификации.

Перечисление стимулов осуществляет итератор стимулов с фильтрацией по предусловию. При заданном пресостоянии он должен последовательно выдавать все стимулы, удовлетворяющие предусловию. Иными словами, перечисление стимулов – это перечисление решений уравнения предусловия для заданного пресостояния относительно стимула. Это общая задача для всех видов тестирования.

Перечисление постсостояний осуществляет генератор постсостояния с фильтрацией по постусловию. Для автомата Мили при заданном пресостоянии, стимуле и реакции он перечисляет решения уравнения постусловия относительно постсостояния. При сильном детерминизме постсостояние однозначно определяется парой пресостоянием и стимулом. При слабом детерминизме оно определяется тройкой, то есть зависит ещё и от реакции. Если автомат сильно недетерминирован, постсостояние неоднозначно определяется даже тройкой пресостояние, стимул и реакция. Именно в этом случае требуется перечисление постсостояний для автомата Мили.

Реакцию мы получаем от реализации, и поэтому нам не нужно её вычислять. Другое дело, что для полноты тестирования нам нужно получить все имеющиеся в реализации реакции на

данный стимул при разных прогонах теста данной трассы. Как это делать – отдельная проблема.

Пятый компонент тестовой системы: обходчик конечного автомата. Условия обхода.

Итак, мы обозначили четыре компонента тестовой системы: итератор стимулов, проверка предусловия, генератор постсостояния и тестовый оракул, проверяющий постусловие. Для извлечения модели нам нужен ещё один, пятый компонент, отвечающий за общую стратегию тестирования. Он называется *обходчиком автомата*.

Обходчик автомата решает задачу «везде побывать, во все двери постучаться»: в каждом состоянии спецификации, в которое мы можем попасть по трассе, полученной в результате тестирования реализации без обнаружения ошибки, то есть общей для реализации и спецификации, мы должны попробовать каждый переход, определённый в этом состоянии в спецификации. Эта проверка либо обнаружит ошибку, либо добавит переход в строящийся подавтомат спецификации.

Автомат Мили называется сильно связным, если из каждого состояния в каждое другое состояние ведёт хотя бы один маршрут (цепочка переходов). В таком автомате, если он конечен, существует маршрут, начинающийся в начальном состоянии и проходящий через все переходы. Такой маршрут называется обходом. Сильно связность и конечность является достаточным и *почти* необходимым условием существования обхода.

Это условие становится необходимым для алгоритмов обхода заранее неизвестного автомата. Таким образом, если подавтомат конечен и сильно связен, задача теоретически может быть решена одним тестовым примером, без рестарта – не отрывая пера от бумаги. В противном случае может потребоваться несколько тестовых примеров, покрывающих в совокупности все переходы подавтомата спецификации. Для бесконечного автомата, очевидно, число тестов также будет бесконечно.

При обходе автомата мы сталкиваемся с проблемой недетерминизма. Дело в том, что для обхода нам нужно уметь выбирать в данном состоянии выходящий из него переход по своему усмотрению. Однако при фиксированном спецификационном пресостоянии переход определяется тремя параметрами: стимулом, реакцией и постсостоянием, а тест умеет управлять только стимулом.

Поэтому сначала мы выдвинем два условия: 1) при тестировании реакция будет только одна, и 2) при этой реакции постсостояние тоже будет одно.

Второе условие означает, что спецификация слабо детерминирована. Может возникнуть вопрос: насколько сильным является это ограничение? Иными словами, имеют ли слабо детерминированные спецификации такую же спецификационную мощность, как и любые спецификации, или меньшую? Известно, что для любого недетерминированного порождающего автомата существует детерминированный автомат, порождающий то же множество трасс. Если недетерминированный автомат конечен, то соответствующий детерминированный автомат тоже конечен. Это утверждение известно как основная теорема о регулярных множествах.

Нас, однако, интересует не столько сам спецификационный автомат, сколько описывающая его спецификация, которая всегда есть конечная запись. Если автомат конечен, то, очевидно, его

всегда можно описать спецификацией конечного размера. Однако конечная спецификация может описывать не только конечные автоматы. Не может ли получиться так, что конечная спецификация описывает некоторый недетерминированный бесконечный автомат, а соответствующий ему детерминированный автомат не описывается конечной спецификацией? Для решения этой задачи нам, очевидно, не хватает формального понимания того, что такое сама конечная спецификация. Здесь есть некоторые «ножницы» между моделью автомата, описываемой спецификацией, и самой спецификацией.

Но вернёмся к тестированию.

Если спецификация слабо детерминирована, то условие единственности реакции означает реализационную гипотезу о том, что для данной реализации спецификационный подавтомат сильно детерминирован, если не считать висящих переходов. Это свойство будем называть *спецификационным детерминизмом реализации*. При запрете блокирующего deadlock`а из этого свойства следует, что реализация наблюдаемо детерминирована, по крайней мере, на общих трассах. Однако эта гипотеза требует от реализации большего, а именно, определённого соответствия состояний. Будем говорить, что два состояния реализации и спецификации соответствуют друг другу, если они достижимы по некоторой общей трассе, соответственно, в реализационном автомате и спецификационном автомате (и, следовательно, подавтомате).

Спецификационный детерминизм можно определить так: если два реализационных состояния соответствуют одному и тому же спецификационному состоянию, то для любого стимула, определённого в этом спецификационном состоянии, эти реализационные состояния, принимая этот стимул, не могут возвратить разные, но обе правильные реакции. Если возвращаемая реакция не допускается спецификацией, то это означает ошибку несоответствия.

Итак, мы требуем, чтобы для данной реализации спецификационный подавтомат был сильно детерминированным, конечным и сильно связным.

В нашем примере мы выбрали как раз такую реализацию.

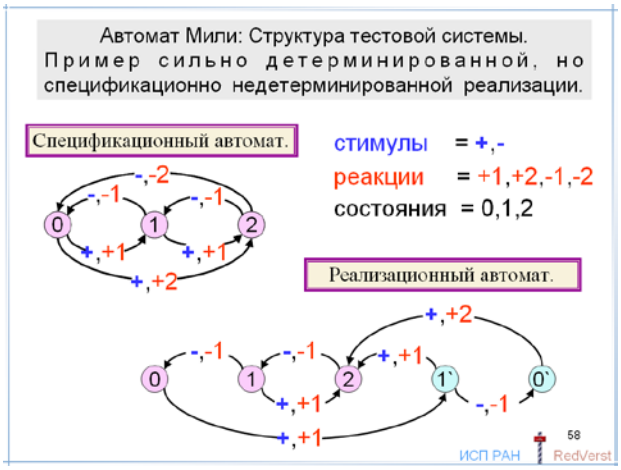
Пример сильно детерминированной, но спецификационно недетерминированной реализации.

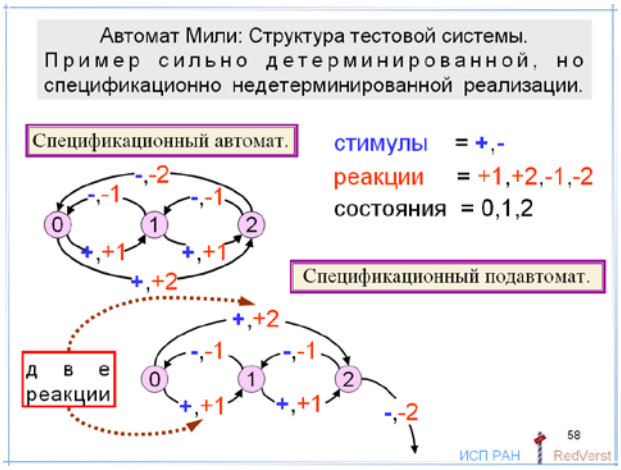
Спецификационный детерминизм, вообще говоря, более сильное требование, чем сильный детерминизм.

В нашем примере мы можем скопировать два состояния: **`0`** и **`1`**. Копии обозначены штрихом и выделены цветом. Эта реализация уже имеет не `3`, а `5` состояний, но она всё ещё спецификационно детерминирована.

Однако, если в состоянии **`0`**` для стимула **`+`** выбрать реакцию не `+1`, а `+2`, что допускает спецификация, то получится спецификационно недетерминированная реализация. Реализационные состояния **`0`** и **`0`**` соответствуют спецификационному состоянию **`0`**. Спецификация разрешает в этом состоянии выдавать в ответ на стимул **`+`** как реакцию `+1`, так и реакцию `+2`. И реализация в состоянии **`0`** выдаёт `+1`, а в состоянии **`0`**` выдаёт `+2`.

Получается, что эта реализация сильно детерминирована и конформна спецификации. Однако спецификационный подавтомат недетерминирован. В нём из одного состояния выходят два перехода по одному стимулу, но с разными реакциями.



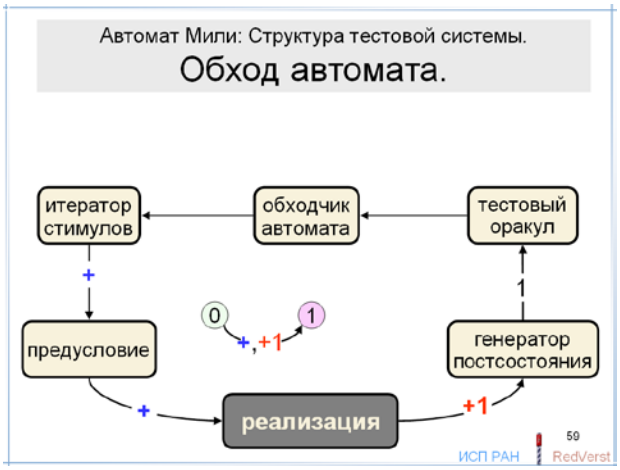


Здесь возникает вопрос: насколько практична наша гипотеза о спецификационном детерминизме? Прежде всего, можно заметить: если и реализация, и спецификация обе сильно детерминированы, то гипотеза верна. В этом случае спецификационному недетерминизму просто неоткуда взяться. Если же, как в нашем примере, сильно детерминирована только реализация, а спецификация даёт выбор реакций, то есть слабо детерминирована, то лазейка для спецификационного недетерминизма остаётся. Вопрос в том, что означает этот выбор? Если это трактуется как внешний выбор, то есть выбор между разными реализациями, гипотеза верна. В одной реализации всегда выбирается реакция `+1`, а в другой `+2`. Гипотеза говорит о том, что выбор не должен быть внутренним, то есть выбором между состояниями одной реализации: в одном состоянии `+1`, а в другом состоянии `+2`. Во многих случаях мы можем быть уверены, что разные реакции, разрешаемые спецификацией, предполагают разные реализации, а не разные состояния одной реализации.

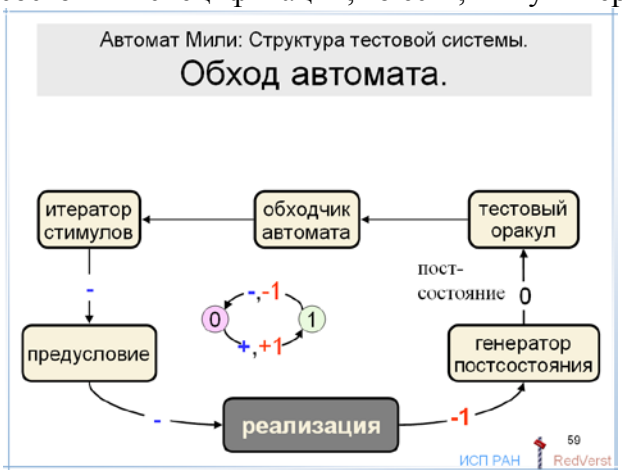
Обход автомата.

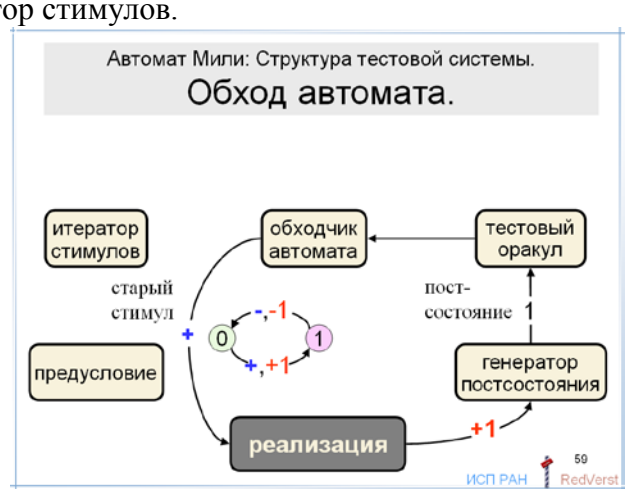
Теперь наша тестовая система состоит из пяти компонентов. Посмотрим, как она работает на нашем примере.

Сначала обходчик запрашивает у итератора стимулов новый стимул, определённый в текущем состоянии. Итератор, выбирает первый стимул и проверяет его через постусловие. Если стимул отвергается, итератор выбирает следующий стимул. Стимул, прошедший проверку предусловия, поступает в реализацию. В ответ мы получаем реакцию. После этого генератор постсостояния, зная пресостояние, стимул и реакцию, вычисляет постсостояние. Затем эта четвёрка проверяется тестовым оракулом. Если постусловие выполнено, обходчик автомата добавляет переход в строящийся подавтомат.

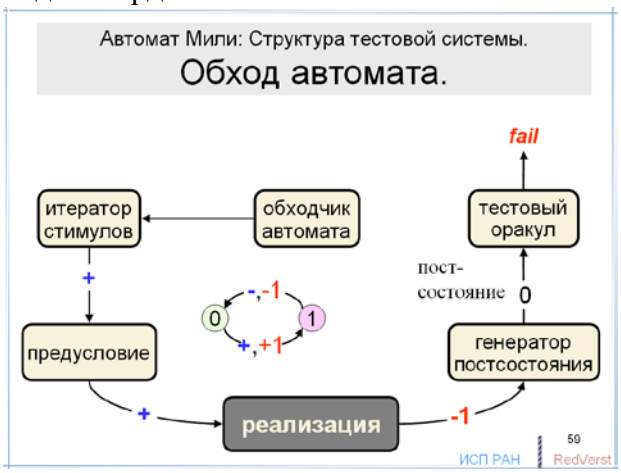


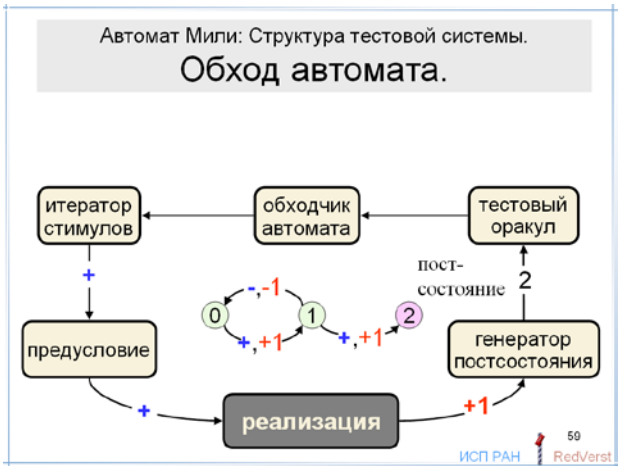
Обходчик может подавать в реализацию стимул, который раньше уже подавался в текущем состоянии спецификации, то есть, минуя итератор стимулов.



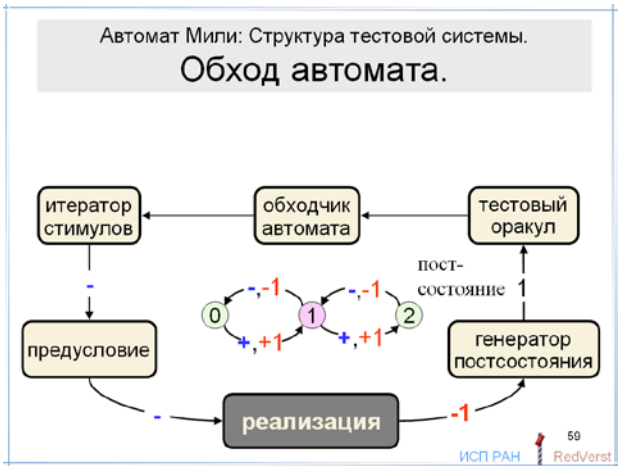


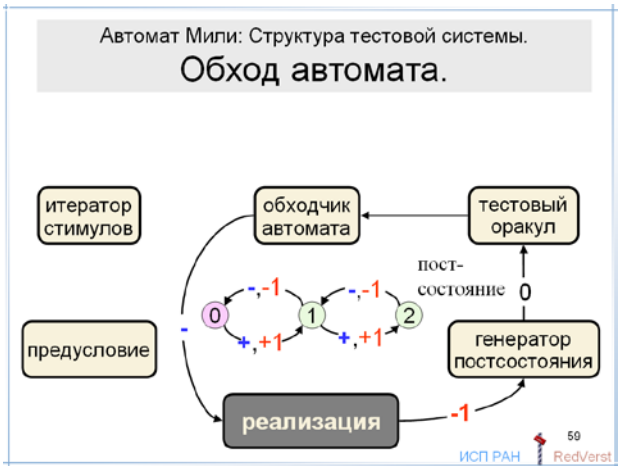
Если реализация вернула неправильную реакцию, то это ловится тестовым оракулом, который выдаёт вердикт ***`fail`***.



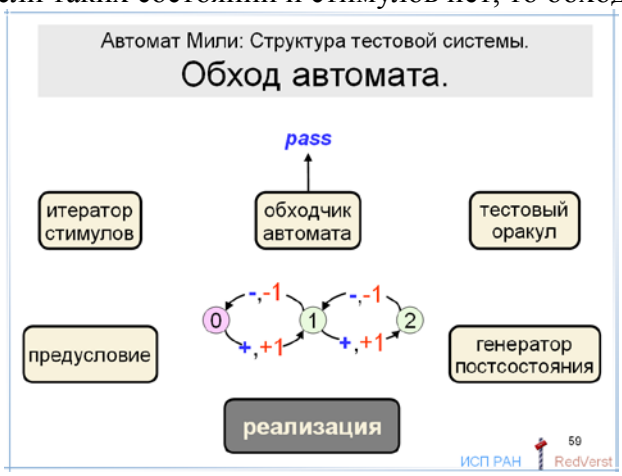


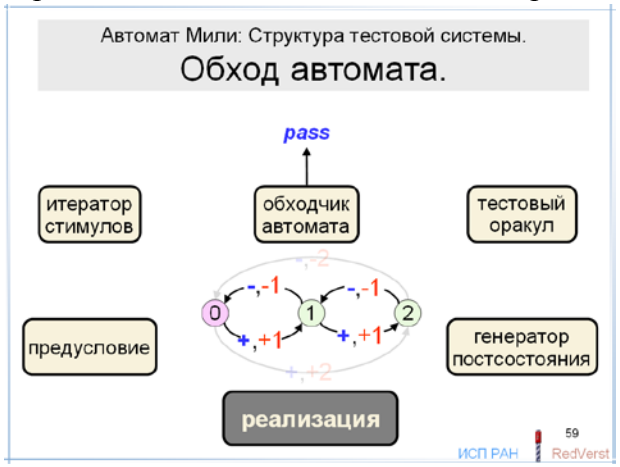
Тестирование продолжается, по крайней мере, до тех пор, пока хотя бы в одном состоянии построенного подавтомата есть определённый в нём, но ещё не опробованный стимул.





Если таких состояний и стимулов нет, то обход совершён, и подавтомат полностью построен.





Теперь мы видим, что мы построили именно подавтомат: в нём нет спецификационных переходов, изменяющих состояние сумматора на 2.

Мы не будем рассказывать о самом алгоритме обхода, это вопрос интересный, но технический. Об этом можно прочитать в нашей статье в журнале «Программирование». Важно лишь отметить, что это обход неизвестного заранее спецификационного подавтомата. О том, куда ведёт и какой реакцией помечен переход, начинающийся в данном пресостоянии и помеченный данным стимулом, мы узнаём только тогда, когда оказываемся в этом пресостоянии и подаём этот стимул. Только тогда мы получаем в ответ заранее неизвестную реакцию и определяем заранее неизвестное постсостояние. И только тогда мы можем этот переход нарисовать.

Обход автомата и полнота тестирования.

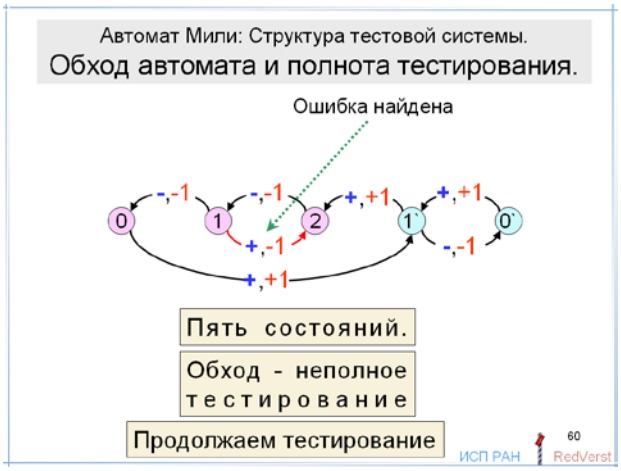
Итак, мы построили спецификационный подавтомат в процессе тестирования с одновременным обходом этого подавтомата. Такое тестирование, очевидно, значимое, но оно может оказаться ещё не полным.

Тем не менее, даже такое тестирование практически полезно. Когда мы делали свой первый проект с Нортелем по спецификации и тестированию интерфейса ядра операционной системы, мы применяли как раз такой метод. И было найдено около 200 ошибок, хотя предполагалось, что их почти нет.

Обход становится полным тестированием только при очень сильной реализационной гипотезе, которая фактически предполагает выполненным ровно то, что из неё должно следовать, а

именно: реализация такова, что обход порождаемого ею спецификационного подавтомата оказывается полным тестированием. Такая гипотеза далеко не всегда может быть принята.

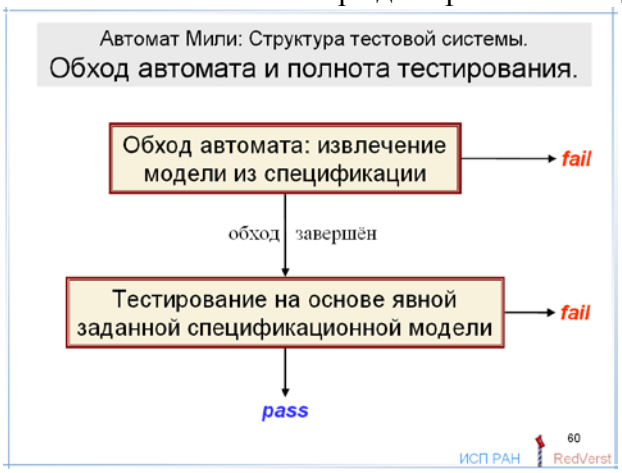
В качестве примера можно рассмотреть спецификационно детерминированную реализацию нашего сумматора с пятью состояниями. После того как будет закончен обход спецификационного подавтомата, окажется не проверенным переход из состояния **1** в состояние **2** по стимулу **+**. Эта реализация конформна спецификации. Но в ней легко сделать ошибку, которую мы не обнаружим при обходе, заменив в не пройденном переходе правильную реакцию `+1` на неправильную реакцию `-1`.



Хотя достаточно было бы дать ещё четыре стимула **++-+**, чтобы ошибку обнаружить. Похожий случай у нас был на практике.

Конечно, во всём этом нет ничего страшного, поскольку наша цель заключалась не в том, чтобы построить полный тест, а в том, чтобы извлечь из спецификации нужную спецификационную подмодель, что мы и сделали. Мы можем делать двухэтапное тестирование:

- сначала делаем обход с построением подавтомата,
- а потом применяем любые методы тестирования соответствия, основанные на явном задании спецификационной модели. В частности, мы можем применять общий алгоритм генерации полного тестового набора для трассовой сводимости.



Такое двухэтапное тестирование полезно не только для полноты тестирования, но и для построения быстрого теста, предназначенного специально для обнаружения конкретной ошибки. Если при обходе мы обнаружили ошибку на некотором переходе, то по пройденному к этому моменту автомату мы можем построить тест как цепочку переходов без циклов, начинающуюся в начальном состоянии и заканчивающуюся проверкой этого перехода. При желании можно даже строить такую цепочку минимальной длины. После того как в реализации эта ошибки исправлена, мы можем вместо того, чтобы снова делать полный обход, прогонять

этот быстрый тест, чтобы проверить, что ошибка действительно исправлена. Это называется *спрямлением* теста для построения контрпримера.

Полное тестирование явно заданного конечного автомата.

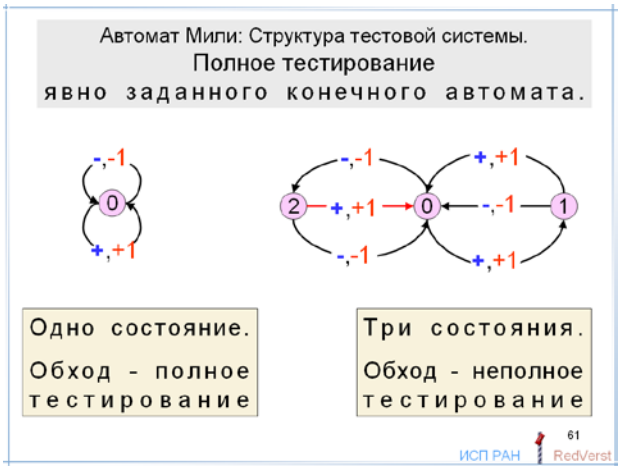
С другой стороны, специально для сильно детерминированных автоматов Мили создана довольно хорошо разработанная теория тестирования соответствия, доведённая до практически применимых алгоритмов с неплохими оценками сложности. В основном предлагаемые алгоритмы стремятся построить один полный тест, то есть полное тестирование ведётся без рестарта – не отрывая пера от бумаги. Правда, такие алгоритмы налагают дополнительные ограничения, как на спецификацию, так и на реализацию.

Спецификация должна быть *минимальной* – в ней не должно быть эквивалентных состояний. Это не страшно, поскольку существуют алгоритмы и даже инструменты минимизации явно заданного автомата Мили.

Ограничения на реализацию означают соответствующую реализационную гипотезу и сводятся, в основном, к ограничению числа состояний реализации. Если **модель ошибок**, то есть наши предположения о том, какие могут быть ошибки, утверждает, что *ошибки не увеличивают числа состояний*, то число состояний в реализации не больше, чем в спецификации. Такие ошибки могут быть двух типов: *не та реакция* или *не то постсостояние*, но не дополнительное постсостояние.

Алгоритмы работают не только тогда, когда число состояний реализации не превосходит числа состояний спецификации, но и тогда, когда имеется ограниченное число дополнительных состояний. Нужно только учитывать, что каждое дополнительное состояние увеличивает время тестирования в число раз, равное числу стимулов. Это утверждение доказано Василевским в 1973 г. Кстати, мы обнаружили в интернете *42* ссылки на эту работу Василевского в англоязычной литературе и *ни одной* в русскоязычной.

Чтобы не было недоразумений, нужно сказать, что само по себе отсутствие в реализации «лишних» состояний не гарантирует ни того, что реализация изоморфна спецификационному подавтомату, ни того, что обход спецификационного подавтомата будет полным тестированием.



Вот два примера реализации нашего сумматора.

Обе реализации конформны спецификации. В них ослаблены предусловия операций: в каждом состоянии можно увеличивать и уменьшать.

Левая реализация имеет всего одно состояние. Она не изоморфна спецификационному подавтомату, но обход является полным тестированием.

Правая реализация имеет три состояния, как и спецификация, но, во-первых, она также не изоморфна спецификационному подавтомату, а, во-вторых, при обходе спецификационного подавтомата мы не проверяем один реализационный переход, в котором как раз и могла бы быть ошибка.

Смешанный подавтомат.

Другой способ достичь полноты тестирования предполагает дополнительные тестовые возможности и позволяет достичь полноты тестирования уже при обходе. Эти тестовые возможности следующие: в каждый момент времени мы можем узнать состояние реализации. Такое тестирование называется *тестированием с открытым состоянием*.

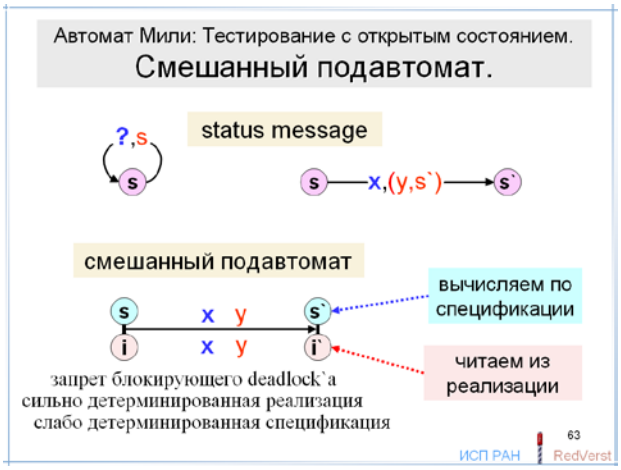
Это можно понимать как наличие специальной операции, которая называется *status message*.

Предполагается, во-первых, что такая операция определена в каждом состоянии реализации. В нашей модели это означает, что в каждом состоянии автомата Мили определён переход-петля, стимул которого – это запрос состояния, а реакция – значение состояния. Во-вторых, предполагается, что эта операция достоверна: это действительно петля и возвращаемая реакция действительно значение состояния. Эту операцию мы не тестируем, предполагая, что она работает правильно. Иными словами, наличие такой тестовой возможности одновременно предполагает выполненной реализационную гипотезу о достоверности операции *status message*.

Другое понимание – это когда мы считаем, что значение постсостояния мы получаем вместе с реакцией как её часть. Какое понимание выбрать – это лишь вопрос интерпретации и удобства отображения в модели. На практике у нас может быть, например, специальная операция в классе объекта, возвращающая состояние объекта. Или состояние реализации – это поля данных объекта, которые мы можем просто прочитать. Опять-таки, мы используем реализационную гипотезу о том, что состояние реализации полностью хранится именно в этих

полях данных и ни в каком другом месте – других полях данных этого объекта, в других полях данных других объектов, в глобальных переменных и т.п.

Что нам даёт открытость реализационных состояний?



Мы может строить и обходить в процессе тестирования не спецификационный, а смешанный подавтомат, состояния которого – это пара состояний: вычисленное спецификационное состояние и считанное из реализации реализационное состояние. Выполняя переход, мы считываем реализационное постсостояние и вычисляем спецификационное постсостояние. Последнее нужно не только для того, чтобы проверить постусловие, но и для того, чтобы осуществить итерацию стимулов, то есть определить, какой следующий стимул посылать в реализацию. Эту итерацию мы делаем отдельно для каждого реализационного состояния, входящего в пару с данным спецификационным состоянием.

Такой смешанный подавтомат, конечно, тоже может быть «меньше» всей реализации, поскольку в реализации могут быть дополнительные стимулы, которых нет в спецификации, и поэтому мы их не тестируем при обходе.

Хотя состояние подавтомата – это пара состояний реализации и спецификации, однако, это не даёт нам право не читать реализационное постсостояние, если мы вычислили то спецификационное постсостояние, в котором уже были. Пример спецификации сумматора с *тремя* состояниями и реализации с *пятью* состояниями показывает, что одному спецификационному состоянию может соответствовать несколько реализационных. Точно также, мы не можем отказаться от вычисления спецификационного постсостояния в случае, если попали в то реализационное состояние, в котором уже были. Пример реализации сумматора с *двумя* состояниями показывает, что одному реализационному состоянию может соответствовать несколько спецификационных, для каждого из которых нужна своя итерация стимулов.

В общем случае мы имеем некоторое соответствие между состояниями реализации и спецификации, причём такое, что для перехода из одного состояния должен найтись так же помеченный переход из соответствующего состояния, и концы этих переходов также соответствуют друг другу. Такого рода отношения между LTS называются *симуляциями* или, если они симметричны, *бисимуляциями*. Для LTS общего вида существует несколько таких симуляций: строгая, слабая, branchig, delay и другие.

В слабо детерминированных автоматах Мили нет τ-переходов и отношение, которое здесь может быть, является разновидностью строгой симуляции. Отличие в том, что в этих автоматах различаются стимулы и реакции, и соответствия типа **`ioco`** несимметричны относительно них. Реализационное и спецификационное состояние соответствуют друг другу, если мы можем оказаться в этих состояниях после общей трассы в реализации и спецификации.

Если пресостояния `s` и `i` соответствуют друг другу, а стимул `x` определён в `s`, то, в силу запрета блокирующего deadlock`a, этот стимул также определён в состоянии `i`. Для *сильно* детерминированной реализации каждый такой стимул `x` однозначно определяет реакцию `y` и

постсостояние `i`` в реализации, тем самым, однозначно определяя пару `(x,y)`. Эта пара, в свою очередь, для *слабо* детерминированной спецификации либо приводит к нарушению постусловия, либо однозначно определяет постсостояние `s`` и переход в спецификации.

Это показывает, что обход смешанного подавтомата остаётся детерминированным: выбирая стимул, мы можем получить только одну реакцию, реализация может перейти только в одно состояние, а при этих условиях, если не обнаружена ошибка, спецификационное состояние также единственно.

Все трассы автомата, который мы строим и обходим, являются общими для реализации и спецификации. Когда обход завершён, оказывается, что мы прошли все общие трассы. Любая трасса реализации, ответвляющаяся от общей трассы, делает это по «лишнему» стимулу, которого нет после этой трассы в спецификации и который мы, тем самым, не должны тестировать. Аналогично, любая трасса спецификации, ответвляющаяся от общей трассы, делает это по «лишней» реакции, которая не обязана быть в реализации, и поэтому мы не можем требовать появления такой реакции при тестировании.

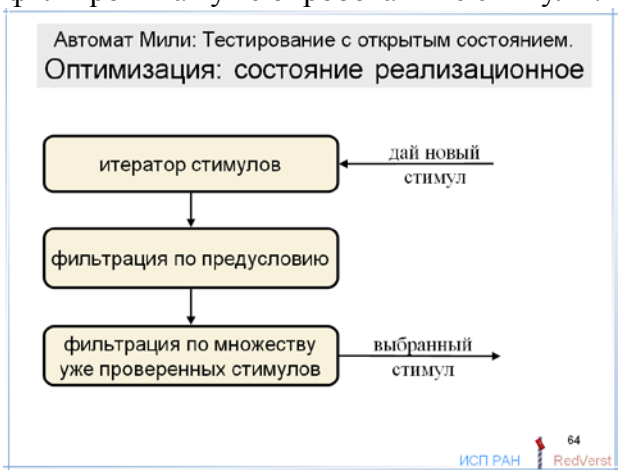
Тем самым, обход оказывается полным тестированием.

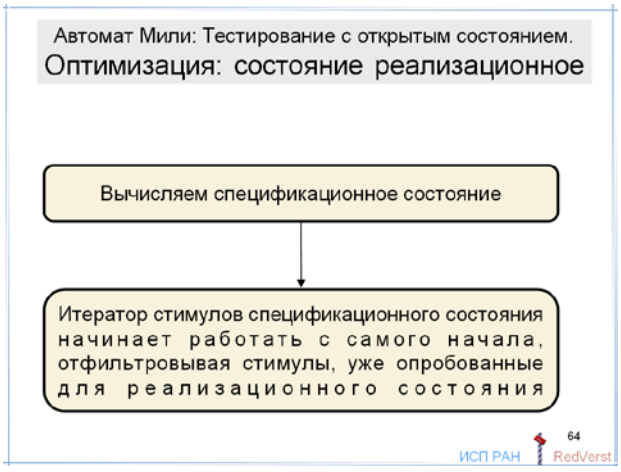
Оптимизация

Легко заметить, что для парных состояний мы делаем избыточное тестирование. Если реализационное состояние входит в пару с `10` спецификационными состояниями и в `5` из них определён стимул `x`, то получится, что мы, по крайней мере, `5` раз подавали стимул `x` в этом реализационном состоянии, трактуя этот стимул как новый. Он новый для каждого из этих `5` спецификационных состояний, но реализационное состояние одно и то же, поэтому и результат в сильно детерминированной реализации будет один и тот же. Поэтому достаточно было бы это делать `1` раз.

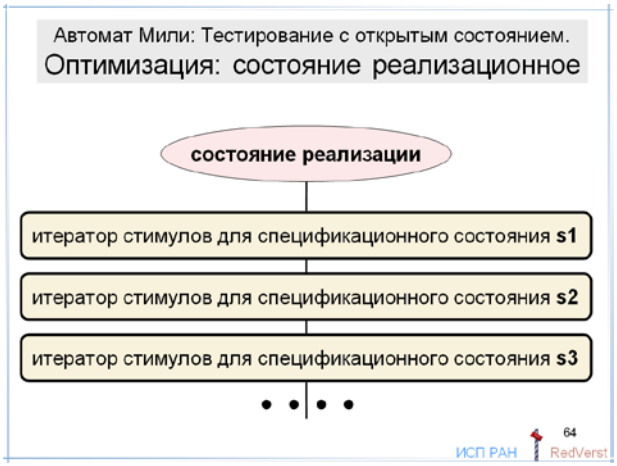
Оптимизация заключается в том, чтобы хранить только реализационное состояние и отфильтровывать не только те стимулы, которые нарушают предусловие, но и те, которые мы уже опробовали в данном реализационном состоянии. Это можно сделать двумя способами.

Первый способ заключается в следующем. Как только мы попадаем в данное реализационное состояние, мы, как и ранее, вычисляем соответствующее спецификационное состояние. Но делаем мы это не для того, чтобы искать или создавать новую пару состояний, а для того, чтобы начать итерацию стимулов для вычисленного спецификационного состояния с самого начала, отфильтровывая уже опробованные стимулы.





Другой способ оптимизации лучше объяснить на примере объектно-ориентированной реализации тестовой системы. В этом случае итератор стимула – это объект, который мы можем создавать для каждого реализационного состояния и каждого спецификационного состояния, когда это спецификационное состояние оказывается в паре с реализационным состоянием.



Вместо создания пары состояний мы храним вместе с реализационным состоянием множество таких созданных итераторов. Когда мы попадаем в это реализационное состояние, мы вычисляем спецификационное состояние и ищем во множестве его итератор. Если такого итератора ещё нет, создаём его и он начинает работать с начала. Если же такой итератор уже есть, то мы вызываем его. В любом случае делаем фильтрацию по уже опробованным стимулам.

ПРОБЛЕМЫ: Взрыв состояний.

Такое тестирование с открытым состоянием порождает одну из проблем, имеющих общее название «взрыв состояний». Дело в том, что наш смешанный подавтомат может иметь слишком много состояний. Это происходит за счёт реализационных состояний, входящих в пару. Поэтому оптимизация, когда вместо пары используется только реализационное состояние, ничего не меняет. Спецификация – это всё-таки некоторая абстракция, то есть мы отвлекаемся от множества несущественных деталей; поэтому спецификационных состояний может быть приемлемое количество. Реализационное же состояние – это просто набор бит. Если суммарный объём полей данных, представляющих состояние, равен 100 битам, то мы имеем 2^{100} реализационных состояний. Все они формально разные. Понятно, что для таких величин тест будет работать неприемлемо долго.

Здесь мы сталкиваемся с классическим противоречием двух желаний: желание абстрагироваться от реализационных деталей и желания иметь полное тестирование, учитывающее все реализационные детали.

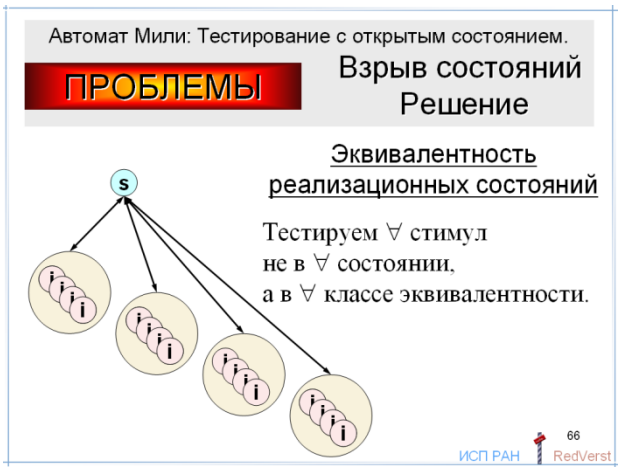
ПРОБЛЕМЫ: Взрыв состояний. Решение.

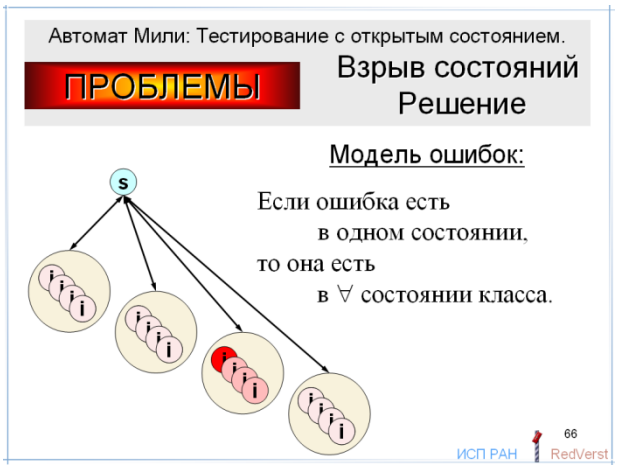
Решение проблемы можно найти, вводя эквивалентность реализационных состояний, реализационную эквивалентность. До этого места у нас был договор: мы умеем читать реализационное состояние. Но мы не договаривались, что мы умеем его понимать. Именно поэтому для нас все эти 2^{100} состояний различны, и мы формально вынуждены стремиться тестировать поведение реализации в каждом из них, если мы его можем достигнуть.

Предположим, что мы кое-что знаем об устройстве реализации, что она не совсем уж «чёрный» ящик. На этом этапе нам не нужно знание её алгоритмов, закодированных в программном коде;

нам нужно знать только, как устроено её состояние. Мы можем сообразить, что из этих `100` бит `30` несущественны, поскольку не влияют на выполнение нужных нам операций, то есть обработку стимулов. Может быть, эти `30` бит нужны для других операций, но мы эти другие операции не тестируем. Далее, оказывается, что байтовое поле служит для хранения не `256` разных значений, а только пяти символов в коде ASCII. И так далее.

В целом мы получаем отношение эквивалентности на множестве реализационных состояний, и нам не нужно тестировать каждый стимул в каждом состоянии, а только каждый стимул в каждом классе эквивалентности. Разумеется, такой вывод можно сделать только на основе соответствующей реализационной гипотезы: те состояния, которые мы признали эквивалентными, действительно эквивалентны.





Это значит, в терминах модели ошибок: если ошибка проявляется для одного из состояний, то она проявляется и для каждого эквивалентного состояния.
Если для одного состояния нет ошибок, то их нет для всех эквивалентных состояний. Во многих случаях такая гипотеза вполне приемлема.

Конечно, здесь можно заметить, что спецификационные состояния тоже в каком-то смысле опираются на некоторую эквивалентность реализационных состояний. Почему же нам понадобилась другая эквивалентность? Это объясняется известными «ножницами» между уровнем абстракции спецификации и уровнем абстракции признаваемой модели ошибок, которую мы кладём в основу реализационной эквивалентности. Какие-то детали могут оказаться несущественными с точки зрения функциональных требований, и потому не отражены в спецификации. Однако при поверхностном взгляде на реализацию мы видим, что эти детали составляют существенную её часть, и нет уверенности, что они не влияют на функциональность. Иными словами, только глубокое изучение реализации позволило бы подтвердить или опровергнуть гипотезу о несущественности этих деталей. Вместо глубокого изучения мы оставляем эти детали различимыми в реализационной эквивалентности, и их не влияние на функциональное поведение будем проверять тестированием. Зато другие детали можно проигнорировать, поскольку в их невлиянии на функциональность мы уверены.

Отображение открытых состояний.

Тестирование с открытым состоянием может использоваться не только и не столько для усиления полноты тестирования при обходе, на также как способ определения спецификационного постсостояния. Идея заключается в том, что, зная кое-что об устройстве

реализационного состояния, мы можем определить спецификационное постсостояние как функцию от реализационного состояния.

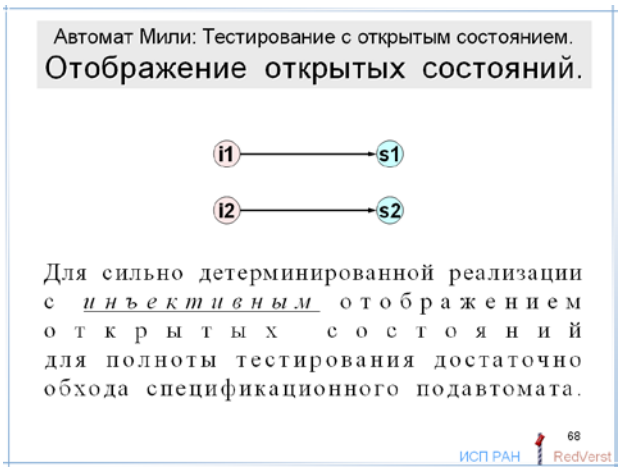
Иными словами, предлагается гипотеза о реализации: между реализационными и спецификационными состояниями существует соответствие, это соответствие является однозначным отображением реализационных состояний в спецификационные, и нам это отображение известно.

При всей внешней экстравагантности такой гипотезы, она имеет право на существование во многих случаях. Во-первых, если спецификация создаётся по коду реализации в процессе кодоинспекции, то это практически гарантирует правильное отображение. Оно будет оставаться правильным также и при модификации реализации, разумеется, в определённых пределах. Во-вторых, если, наоборот, реализация создавалась по спецификации, то есть большая доля уверенности, что разработчик достаточно ленив, чтобы не изобретать реализационные состояния совсем уж непохожими на спецификационные состояния. Он может что-то добавлять, выбирать форму представления данных и т.п., но обычно всегда остаётся связь со спецификацией, которую можно оформить как нужное нам отображение.

Если у нас есть такое отображение, то мы можем при построении спецификационного подавтомата не вычислять постсостояние как решение уравнения постусловия, а получать его как функцию от прочитанного реализационного постсостояния. Это можно делать и в тех случаях, когда решение уравнения постусловия далеко не так очевидно, как в нашем примере с сумматором. Более того, это можно делать и для недетерминированной реализации, хотя о методах тестирования такой реализации мы ещё не говорили.

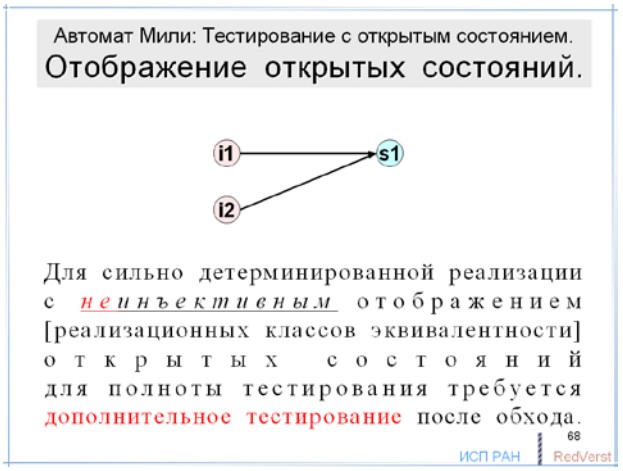
Как обычно, эта медаль имеет оборотную сторону: возникает проблема достоверности отображения состояний. Отнюдь не всегда можно считать его стопроцентно достоверным, а отказываться от отображения не хочется. Что делать?

К счастью, тестовая система обладает хорошим самоконтролем: полученное (неважно каким путём) спецификационное постсостояние не просто принимается на веру, а проверяется постусловием. Если постусловие истинно, такое постсостояние могло бы быть. Если же тестовый оракул выносит вердикт ***fail***, то это может означать ошибку не в реализации, а в отображении. Особенно это удобно, когда спецификация слабо детерминирована, потому что тогда правильное постсостояние, если оно есть, то единственно, и истинность постусловия гарантированно подтверждает нам, что постсостояние то, какое надо.



Тестирование с открытым состоянием

Если отображение инъективно, то есть разным реализационным состояниям соответствуют разные спецификационные состояния, то реализационные и спецификационные состояния, участвующие в тестировании, взаимно-однозначно соответствуют друг другу. Иными словами, они отличаются только способом их кодирования. Поэтому спецификационный подавтомат, если из него удалить висящие переходы, можно считать тем самым подавтоматом реализации, который мы и должны были проверить на трассовую сводимость. Все остальные переходы реализации либо ведут из состояний, которые недостижимы по общим трассам, либо из достижимых состояний, но по стимулам, которые не определены в спецификации после соответствующих общих трасс. Тем самым, наличие или отсутствие этих переходов в реализации не влияет на её соответствие спецификации по трассовой сводимости. Таким образом, при тестировании с открытым состоянием и инъективным отображением состояний сильно детерминированной реализации для проверки соответствия достаточно обхода спецификационного подавтомата.



При наличии реализационной эквивалентности (с соответствующей реализационной гипотезой) отображение можно понимать как отображение класса эквивалентности в спецификационное состояние. Соответственно, инъективность будет означать, что разные классы отображаются в разные состояния. Очевидно, инъективное отображение классов почти всегда является неинъективным отображением состояний (если только все классы не являются синглетонами, то есть множествами состоящими из одного элемента).

Если отображение неинъективно, то одному спецификационному состоянию могут соответствовать несколько реализационных состояний. Когда мы проверяем стимул, определённый в этом спецификационном состоянии, то может оказаться, что мы проверяем его только в одном из соответствующих состояний реализации. Аналогично обстоит дело с неинъективным отображением классов реализационной эквивалентности в спецификационные состояния. Обход спецификационного подавтомата уже не будет полным тестом.

Для полноты нужно использовать либо смешанный подавтомат, либо другие методы, о которых мы говорили раньше. Альтернативный вариант: считать эквивалентными те реализационные состояния, которые соответствуют одному спецификационному. Разумеется, это означает соответствующую реализационную гипотезу об отсутствии тех «ножниц», о которых мы говорили: между уровнем абстракции спецификации и уровнем абстракции признаваемой модели ошибок, которую мы кладём в основу реализационной эквивалентности.

Шестой компонент тестовой системы: медиатор.

До сих пор мы неявно предполагали, что алфавиты стимулов и реакций реализации и спецификации совпадают. На практике, это почти всегда не так. Поэтому требуется ещё один компонент тестовой системы – медиатор.

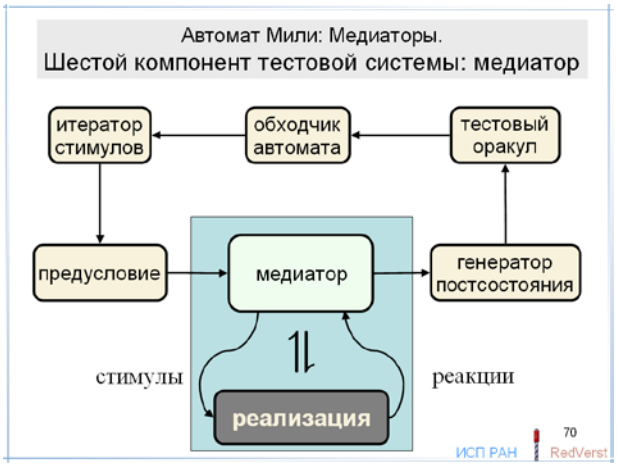
Медиатор – это программа, которая осуществляет связь между двумя моделями: спецификационной и реализационной.

Тест создаётся и работает в терминах спецификационной модели, поэтому медиатор осуществляет преобразование спецификационных стимулов в реализационные стимулы и, в обратном направлении, реализационных реакций в спецификационные реакции.

Как минимум это отображение алфавитов реализации и спецификации. В программном смысле это преобразование из одного типа в другое. Спецификация и реализация работают с разными типами, которые представляют стимулы и реакции, более того, это могут быть типы разных языков. Например, тип стимула – это имя функции или метода класса плюс типы формальных параметров, а тип реакции – это тип функции или типы ответных параметров для языков с несколькими ответными параметрами.

В более сложных случаях преобразование стимула или реакции может зависеть от состояния спецификации или реализации и вообще от предыстории взаимодействия теста и реализации.

В терминах автоматной модели, медиатор – это автомат, который непосредственно взаимодействует с реализацией, то есть, компонуется с ней с помощью оператора параллельной композиции. Тем самым тест взаимодействует не с реализацией, а с результатом такой компоновки. Это очень похоже на асинхронное тестирование. Но здесь есть одно весьма существенное отличие: медиатор – это часть тестовой системы, он создаётся разработчиком тестов. Поэтому мы не только всё знаем об этом медиаторе, но он находится под полным нашим контролем.



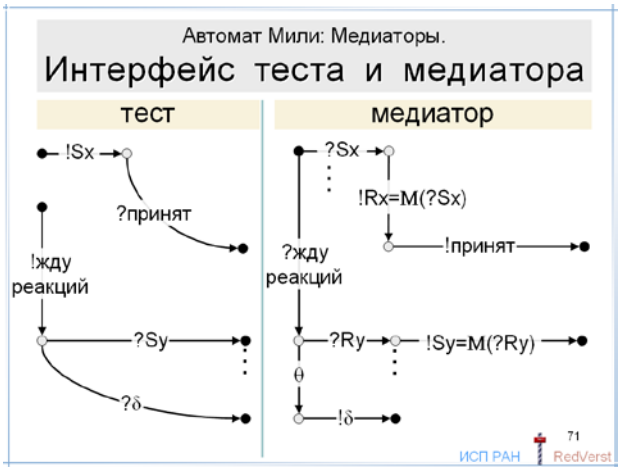
Интерфейс теста и медиатора.

Интерфейс теста и медиатора отличается от непосредственного интерфейса теста и реализации.

Передача стимула. Тест посылает спецификационный стимул в медиатор, который готов принять любой спецификационный стимул от теста.

После этого медиатор преобразует спецификационный стимул в реализационный и посылает его в реализацию.

Поскольку мы запретили блокирующиё deadlock, передача стимула в реализацию всегда проходит, и медиатор сообщает об этом тесту.



Приём реакций. Тест посылает в медиатор запрос на приём реакций. Медиатор начинает ждать от реализации всех реакций.

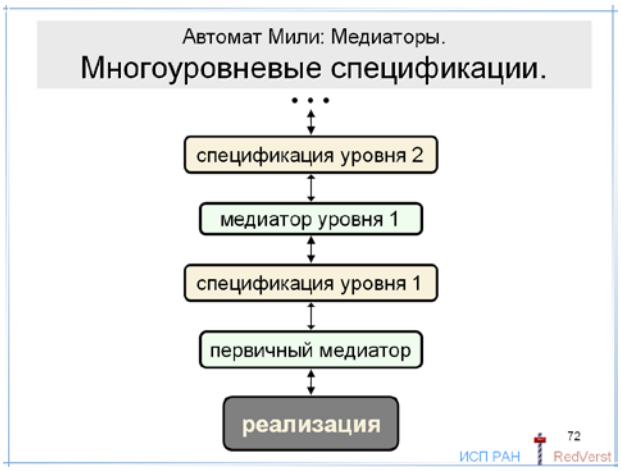
Когда приходит некоторая реализационная реакция, медиатор преобразует её в спецификационную реакцию, которую передаёт в тест.

Если реакций нет, медиатор обнаруживает это по тайм-ауту, то есть по θ-переходу, и посылает в тест сообщение о стационарности δ.

Многоуровневые спецификации.

Спецификация обычно не зависит от языка реализации и операционной среды, в которой работает реализация, поскольку, как правило, это выходит за пределы функциональности требований. Но для того, чтобы тест мог взаимодействовать с реализацией, нужна привязка к этим реализационным вещам, что как раз и делает медиатор. Для реализаций разного типа могут создаваться разные медиаторы при сохранении одной и той же спецификации. Такие медиаторы называют первичными.

При многоуровневых спецификациях используются медиаторы между соседними уровнями спецификации, осуществляющие аналогичные преобразования. Разные уровни спецификации отличаются различной степенью абстрагирования от реализационных особенностей. Более точно: спецификация более высокого уровня описывает более слабые функциональные требования к реализации. Соответственно, тестирование также может быть многоуровневым. Но эту тему мы сегодня развивать больше не будем, за исключением одного специального случая, о котором речь пойдёт позже.



Преобразование стимулов и реакций.

Для медиаторного преобразования предполагается, что спецификация более абстрактна, чем реализация: реализационный стимул или реакция соответствуют только одному стимулу или реакции спецификации.

Преобразование реакций может быть не инъективно: разные реализационные реакции преобразуются в одну спецификационную реакцию. Но здесь никаких проблем не возникает, поскольку на уровне спецификации мы не различаем эти разные реализационные реакции.

Трудности возникают, когда медиаторное преобразование стимулов неоднозначно: одному спецификационному стимулу может соответствовать множество стимулов реализации, и медиатор выбирает один из них. Тем самым, возникает вопрос: а вдруг реализация делает ошибку при другом выборе?

Медиатор стимулов как «погода».

Если вспомнить машину тестирования, то медиатор можно понимать как её переднюю панель, осуществляющую интерфейс с реализацией. Когда мы видим на дисплее символ a, то это спецификационное a, которое медиатор преобразовал из реализационных a1, a2, и так далее. Когда мы нажимаем на кнопку a, то это спецификационное a, которое медиатор преобразует в одно из реализационных a1, a2, и так далее. В этом смысле медиатор стимулов является частью погоды, от состояния которой зависит выполнение реализации. Но это такая погода, которая находится под нашим контролем: медиаторы создаются разработчиками тестовой системы, и составляют её часть.

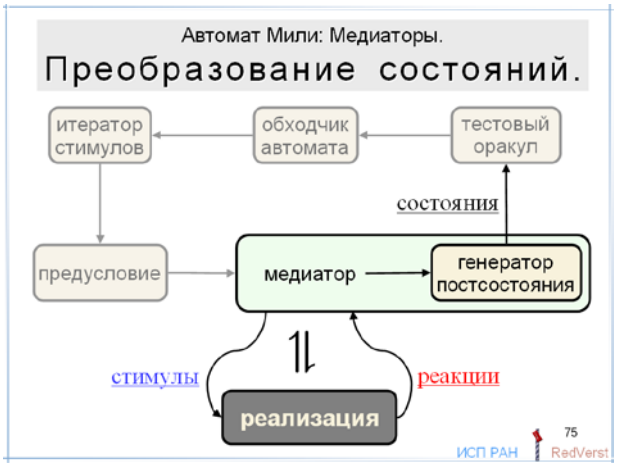
Таким образом, у нас есть выбор:

Нет плохой погоды: мы принимаем реализационную гипотезу об эквивалентности реализационных стимулов, соответствующих по медиаторному отображению одному спецификационному стимулу.

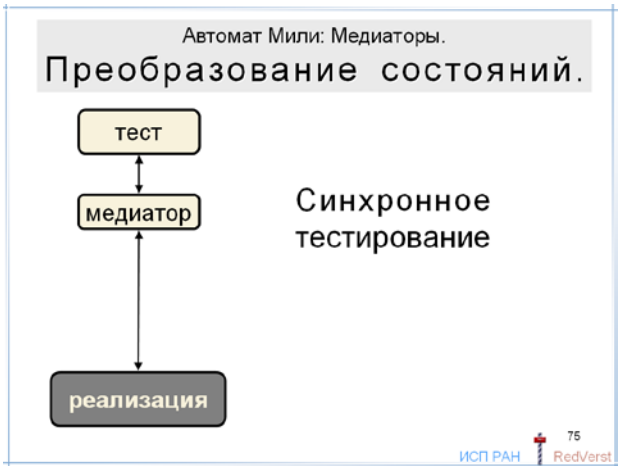
Управление погодой: медиатор должен перебирать реализационные стимулы, соответствующие одному спецификационному стимулу. Важно, что это должно учитываться при обходе подавтомата: в состоянии подавтомата оказываются определёнными не спецификационные, а соответствующие им реализационные стимулы.

Преобразование состояний.

При тестировании с открытым состоянием отображение состояний также является частью медиатора. Такое отображение может быть в первичном медиаторе, а может и не быть, если спецификационное постсостояние вычисляется по постусловию. Однако такое отображение почти всегда присутствует в медиаторах более высокого уровня, соединяющих соседние уровни спецификации. В общем случае, даже когда спецификационное состояние вычисляется, генератор постсостояния можно считать частью медиатора. Таким образом, медиатор осуществляет три преобразования: стимулов, реакций и состояний.



Можно заметить, что медиатор похож на среду передачи при асинхронном тестировании: в обоих случаях предполагается их параллельная композиция с реализацией так, что тест компонуется уже с соответствующим композиционным автоматом. Существенное отличие в том, что медиатор является частью тестовой системы и находится под полным её контролем, а среда передачи – внешняя по отношению к тестовой системе так же, как реализация.



Мы рассмотрели медиаторы для автоматов Мили, но аналогичные медиаторы создаются и для асинхронных автоматов общего вида.

При асинхронном тестировании мы имеем композицию четырёх автоматов: реализации, среды передачи, медиатора и теста.

Цели и проблемы факторизации.

Медиатор связывает реализацию, как детальную модель, и спецификацию, как обобщённую модель. Тем не менее, для тестирования спецификация часто оказывается всё ещё слишком подробной. Для того чтобы тестирование стало возможно и практично, приходится эту модель огрублять. Общий метод заключается в факторизации спецификации.

Основная цель факторизация: уменьшение размера спецификационной модели и, как следствие, размера тестов и длительности их выполнения.

В самом общем смысле спецификацию можно рассматривать как множество трасс наблюдений. Вводится эквивалентность трасс и строится фактор-спецификация на основе этой эквивалентности.

Для автомата Мили естественно строить эквивалентность трасс на базе эквивалентностей трёх видов: эквивалентности состояний, эквивалентности стимулов и эквивалентности реакций.

С факторизацией связаны две проблемы: 1) недетерминизм и 2) запрет блокирующего deadlock`a. Эти проблемы мы сейчас рассмотрим.

Проблема недетерминизма заключается в том, что при факторизации может увеличиться степень недетерминизма. Самое интересное в том, что факторизация может и уменьшить недетерминизм. Такое уменьшение недетерминизма может даже составлять главную цель факторизации в некоторых случаях.

Выбор правильной факторизации – это задача тестировщика. Это та часть создания тестовой системы, которая плохо поддаётся автоматизации, здесь требуются чисто интеллектуальные усилия.

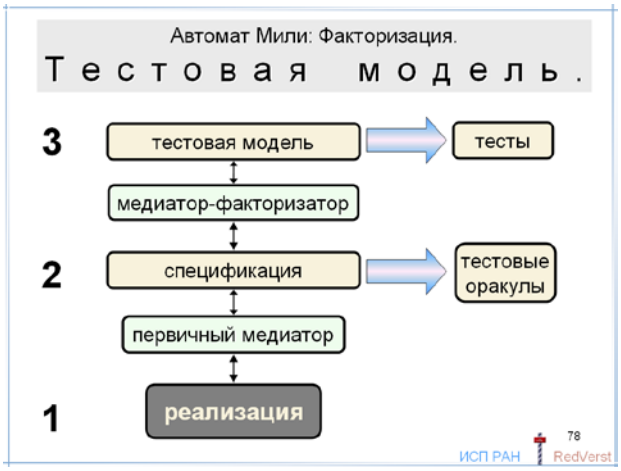
Тестовая модель.

Фактор-спецификация – это тоже спецификация, но с ослабленными требованиями. Можно было бы строить тестовую систему непосредственно по фактор-спецификации, выбросив исходную спецификацию. Но тогда мы потеряли бы возможность более точно проверять правильность реализации.

Вместо этого используется трёхуровневая структура тестовой системы.

Уровень спецификации используется для генерации тестовых оракулов. Первичный медиатор обеспечивает отслеживание текущего спецификационного состояния, стимула и реакции.

Выше вводится уровень тестовой модели. Она связана с уровнем спецификации медиатором, который и осуществляет факторизацию. Эта модель непосредственно используется для генерации тестов.



Таким образом, тестовая, более грубая, модель определяет более короткий тест, но проверка правильности получаемых от реализации реакций осуществляется, по-прежнему, тестовыми оракулами уровня спецификации.

Эквивалентность реакций.

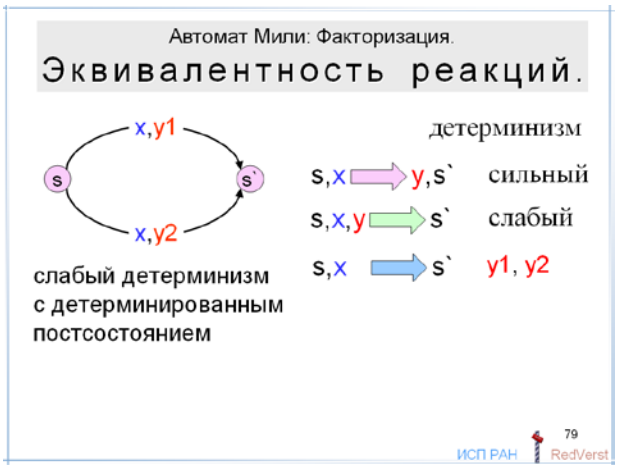
Эквивалентность реакций предполагает реализационную гипотезу о том, что для полноты тестирования достаточно проверить выдачу реализацией хотя бы одной реакции из класса эквивалентных реакций. Простейшая эквивалентность определяется просто на множестве всех реакций. В общем случае нужно говорить об эквивалентности переходов так, что эквивалентные переходы должны отличаться только реакциями.

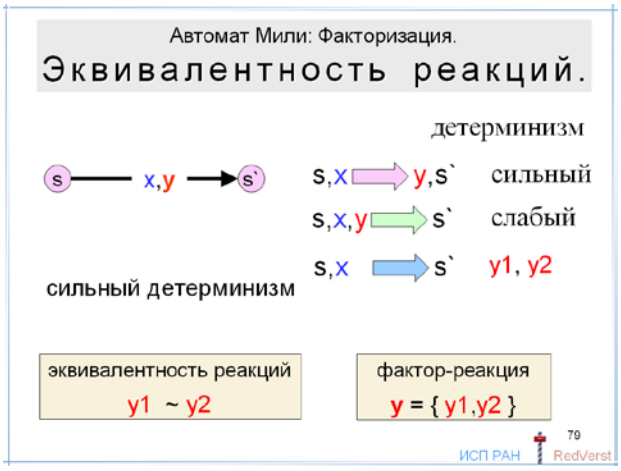
Формально, эквивалентность реакций уменьшает размер автомата, поскольку несколько переходов могут склеиться в один переход. Однако такие склеиваемые переходы отличаются только реакциями, выбором которых тест в большинстве случаев не умеет управлять. Поэтому введение эквивалентности реакций предназначено, в основном, для уменьшения недетерминизма.

Простейший случай, когда источником недетерминизма являются только реакции: пресостояние и стимул однозначно определяют постсостояние, но неоднозначно определяют реакцию. Этот вид детерминизма отличается от сильного и слабого детерминизма.

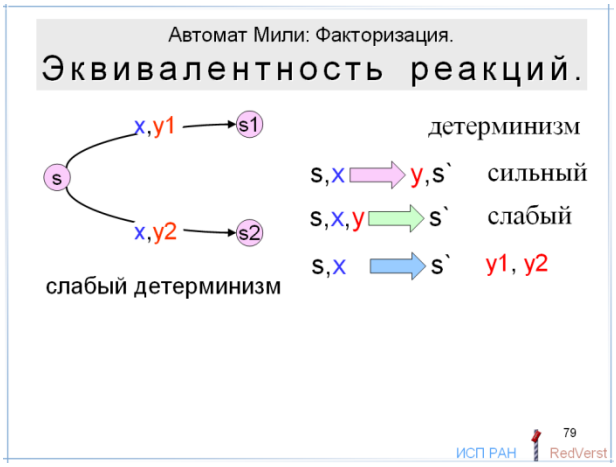
В спецификации допускаются кратные переходы, которые различаются только реакциями.

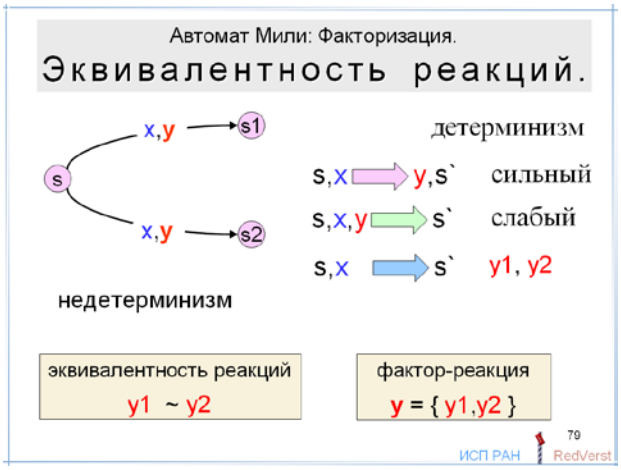
Если такие реакции объявить эквивалентными, автомат становится сильно детерминированным.





Вообще говоря, эквивалентность реакций вовсе не гарантирует уменьшения недетерминизма. Более того, в некоторых случаях она может дать обратный эффект.
Если из одного состояния вели два перехода по одному стимулу и разным реакциям в разные состояния, то это слабый детерминизм.
Склеивая эти реакции, мы получаем недетерминизм.



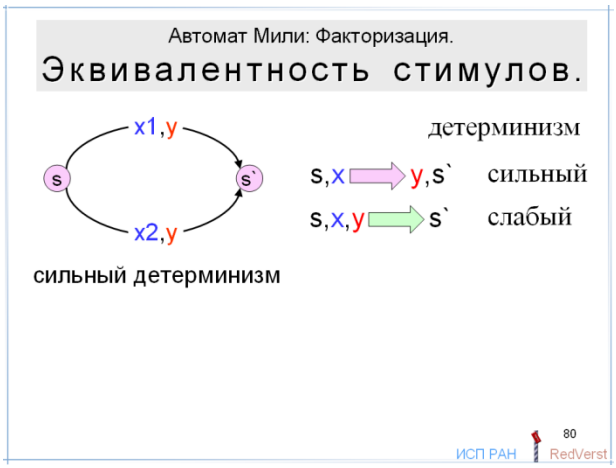


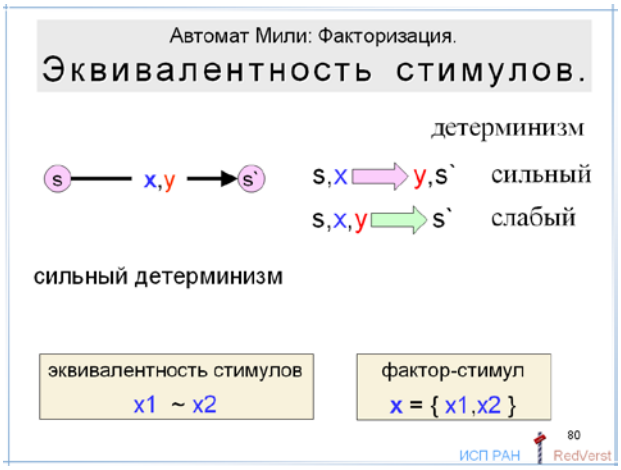
Эквивалентность стимулов.

Эквивалентность стимулов в основном предназначена для уменьшения размера автомата. Поскольку стимулами управляет тест, то чем меньше будет стимулов, тем быстрее будет проходить тестирование.

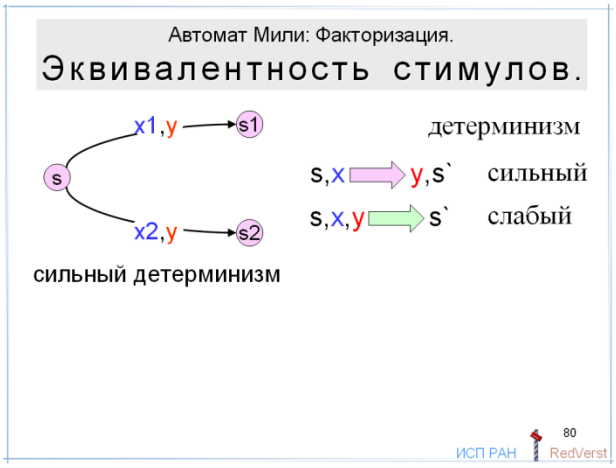
При факторизации по стимулам нужно следить за детерминизмом.

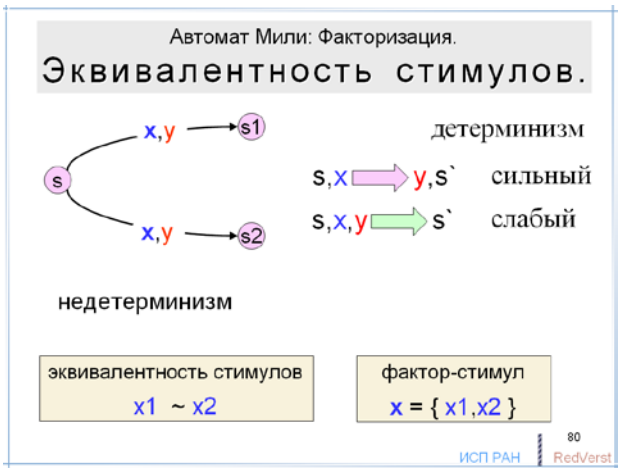
Если склеиваются стимулы кратных переходов, степень детерминизма не изменяется.





Однако если склеиваются стимулы переходов с разными постсостояниями, то это может привести к недетерминизму.





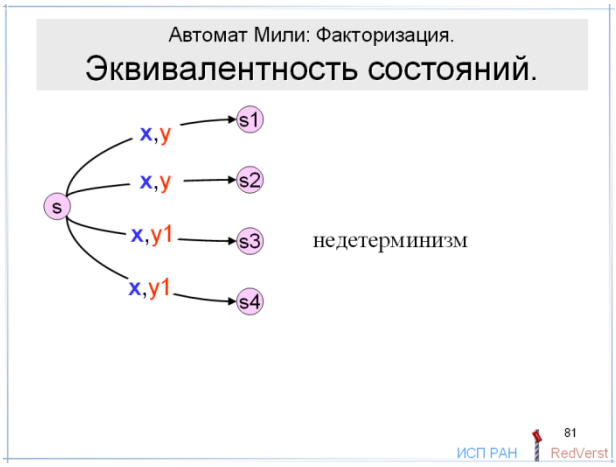
Эквивалентность состояний.

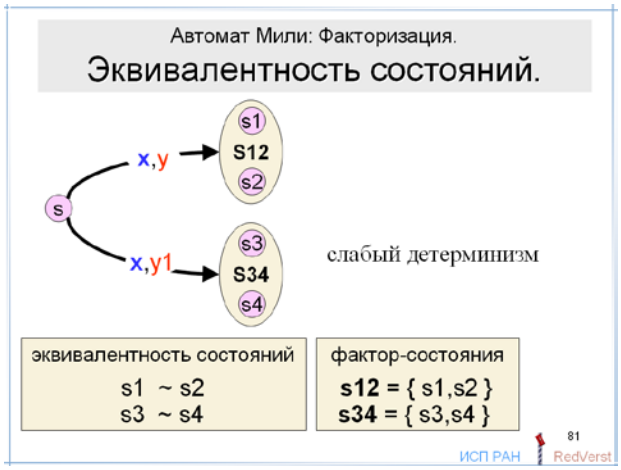
Эквивалентность состояний также предполагает соответствующую реализационную гипотезу, и может использоваться как для уменьшения размера автомата, так и для уменьшения недетерминизма.

Поскольку такая факторизация уменьшает число состояний, она уменьшает число переходов, что сокращает время тестирования.

С детерминизмом дело обстоит сложнее.

Переходы под одним и тем же стимулам и реакциям, которые вели из одного пресостояния в разные склеиваемые постсостояния, склеиваются в один фактор-переход.



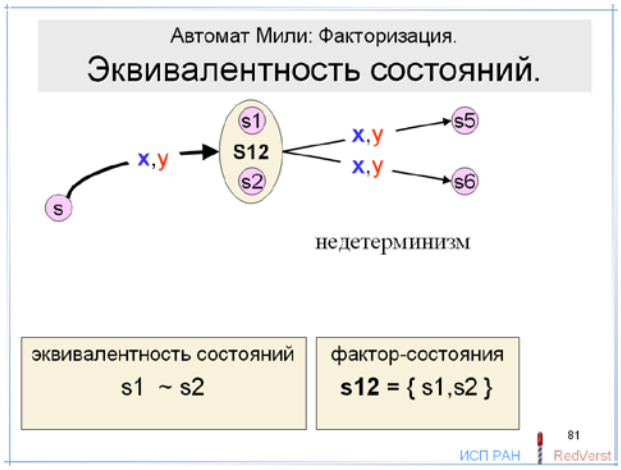


Недетерминированный автомат может стать слабо детерминированным.

Если бы в этом примере не было состояний s3,s4, то исходный автомат оставался бы недетерминированным, а фактор-автомат стал бы сильно детерминированным.

Нужно отметить, что сама по себе эквивалентность не обязательно уменьшает недетерминизм; она может даже увеличить его.

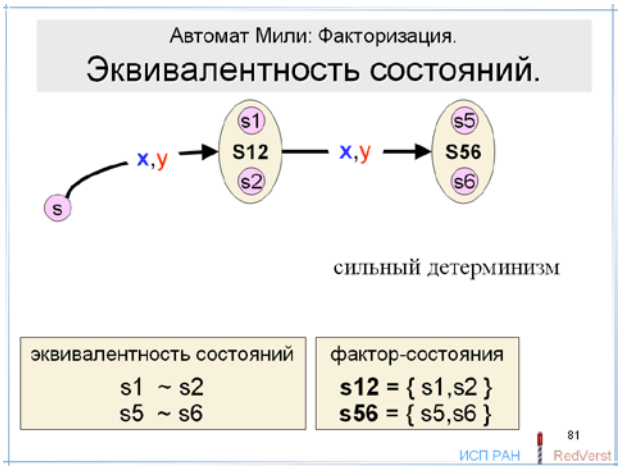
Вот пример.



Исходный автомат сильно детерминирован. После склеивания состояний s1,s2, автомат стал недетерминирован.

Но если бы мы склеили также состояния s5,s6, автомат остался бы сильно детерминированным.

Поэтому искусство построения тестовой модели заключается в том, чтобы найти хороший баланс между практически обоснованной реализационной гипотезой об эквивалентности состояний и требованиями детерминизма.

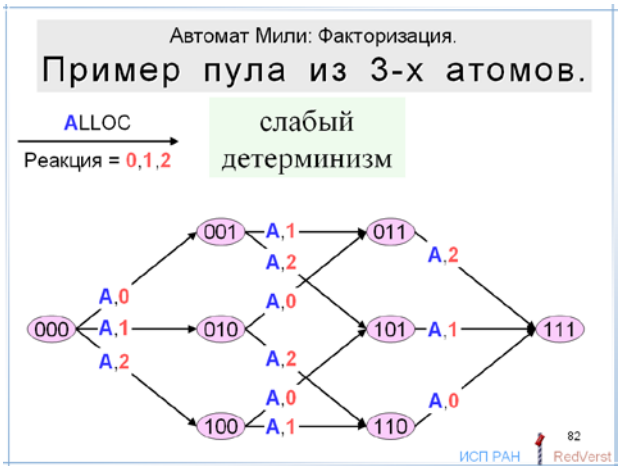


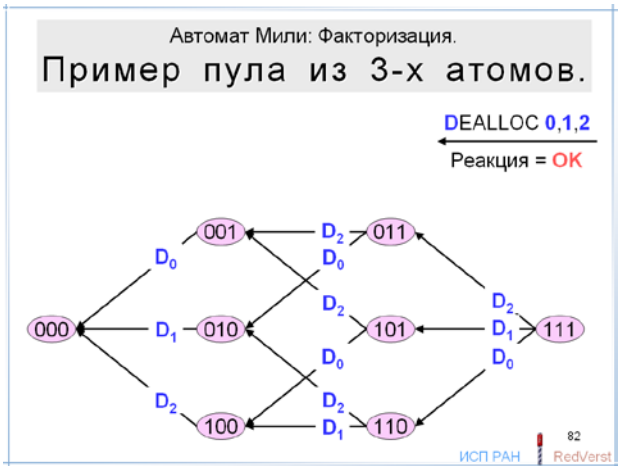
Пример пула из трёх атомов.

Обычно при факторизации используются все три эквивалентности: реакций, стимулов и состояний. В качестве примера рассмотрим пул из фиксированного числа атомов с двумя операциями: `alloc` и `dealloc`. Чтобы автомат пула уместился на экране, число атомов возьмём маленьким – всего `3` атома. Атомы имеют номера `0,1,2`. Состояние пула – это три двоичных числа: `i`-ое число равно `0`, если `i`-ый атом свободен и `1`, если он занят.

Операция `alloc` – это один стимул, в ответ мы получаем номер выделенного нам атома: `0`, `1` или `2`. Это реакция. Если есть несколько свободных атомов, пул может вернуть номер любого из них и атом с этим номером становится занятым. Тем самым, по операции `alloc` пул слабо детерминирован. Будем считать, что эта операция имеет предусловие: запрашивать атом можно только в том случае, когда в пуле есть свободные атомы.

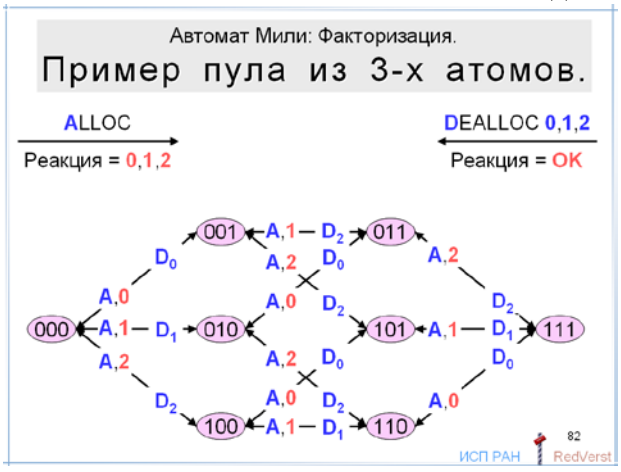
По этой операции пул слабо детерминирован.

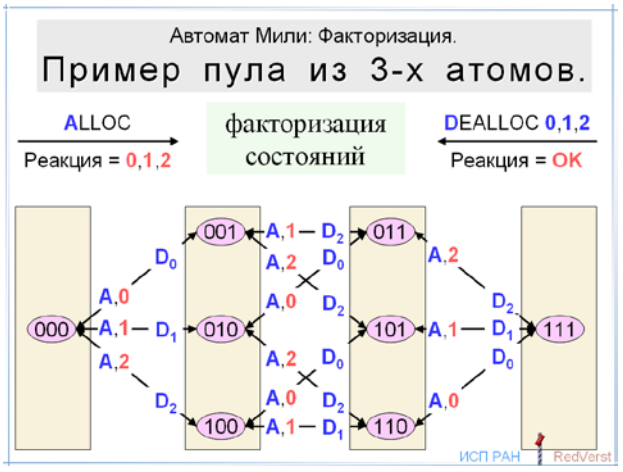




Операция `dealloc` освобождает атом, номер которого передаётся как параметр: `0`, `1` или `2`. Тем самым, здесь мы имеем три стимула. Реакция на каждый из этих стимулов одна и та же – подтверждение освобождения атома, и на слайде она не указана. По этой операции пул, очевидно, сильно детерминирован. Каждый из стимулов `dealloc` также имеет предусловие: освобождать можно только занятый атом.

Эквивалентными объявим состояния с одинаковым числом занятых атомов.

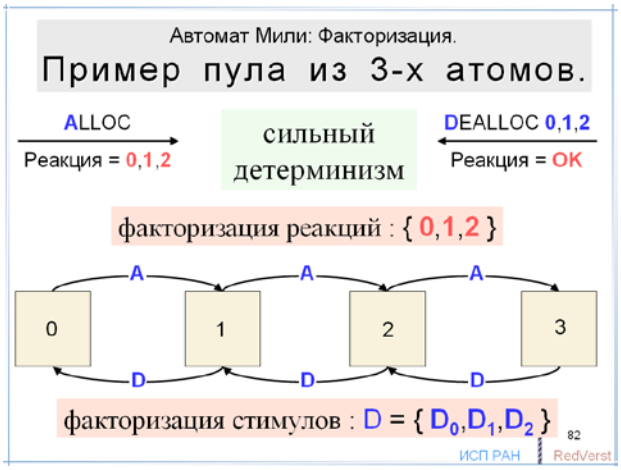




Тем самым, фактор-состояние – это, фактически, число занятых атомов: `0,1,2` или `3`.

Далее факторизуем реакции: все три реакции на `alloc` объявим эквивалентными.

Наконец, факторизуем стимулы: все три стимула, соответствующие операции `dealloc`, объявим эквивалентными.



Очень важно отметить, что факторизующий медиатор преобразует фактор-стимул `dealloc` в операцию `dealloc` с параметром – номером освобождаемого атома. Этот номер медиатор берёт из соответствующего спецификационного состояния: выбирается любой занятый атом. Какой именно атом выбрать – дело медиатора. Этот выбор может быть полностью недетерминированным, или, поскольку медиатор создаётся разработчиком теста и находится под полным контролем тестовой системы, мы можем заставить медиатор выбирать тот или иной занятый атом по каким-то удобным для нас правилам.

В результате мы видим, что получился сильно детерминированный фактор-автомат. То есть нам удалось увеличить степень детерминизма: был слабый, стал сильный.

На рисунке видно, как сильно уменьшился автомат: даже для пула из 3-х атомов число состояний уменьшилось вдвое, а число переходов – в 4 раза.

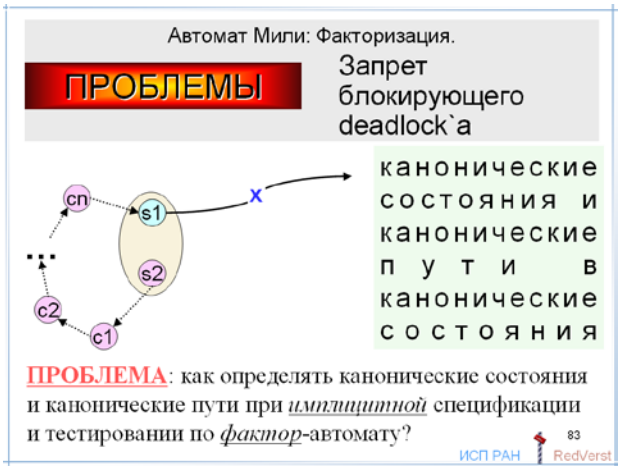
	до факторизации	после факторизации
число состояний	$2^n = 8$	$n+1 = 4$
число переходов	$n \cdot 2^n = 24$	$2 \cdot n = 6$
детерминизм	слабый	сильный

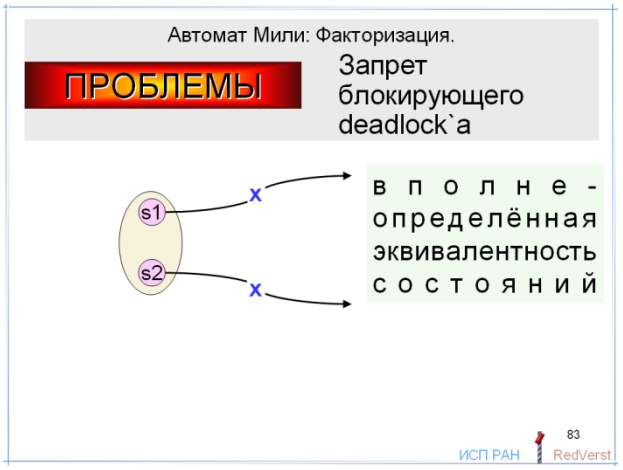
ПРОБЛЕМЫ: Запрет блокирующего deadlock`a.

А теперь рассмотрим вторую проблему факторизации. Мы договаривались запретить блокирующий deadlock.

Однако при факторизации состояний этот запрет может быть нарушен, если в один класс эквивалентности попадают два состояния, и в одном из них, `s1`, стимул `x` определён, а в другом, `s2`, – не определён.

Мы должны определить переход из такого фактор-состояния по стимулу `x` для того, чтобы проверить этот стимул в состоянии `s1`. Однако реализация может оказаться в состоянии, аналогичном состоянию `s2`, в котором стимул `x` не определён. Примером такой реализации может служить сама исходная спецификация, которая, очевидно, конформна сама себе и не приводит к блокирующему deadlock`у при тестировании по исходной спецификации. Однако при тестировании по фактор-спецификации такой deadlock как раз и возникнет.





Одно из решений этой проблемы основано на введении канонических состояний и канонических путей в канонические состояния. В нашем примере для стимула `x` каноническим будет состояние `s1`, в котором этот стимул определён.

Из состояния `s2` определяется канонический путь в состояние `s1`. Когда мы хотим в нашем фактор-состоянии дать стимул `x`, мы смотрим, в каком состоянии мы находимся. Если это состояние `s1`, то стимул можно давать. Если же это состояние `s2`, то сначала мы даём цепочку стимулов, приводящую нас по каноническому пути в состояние `s1`, а потом уже даём стимул.

Такой метод может применяться, если спецификация обладает достаточным детерминизмом и связностью, так как в противном случае гарантированного пути из `s2` в `s1` может не существовать.

Но самое главное: выделение канонических состояний и путей требует явного задания спецификации. Если же спецификация имплицитна, то мы получаем такое явное представление только постепенно в процессе тестирования.

Более того, тестирование мы хотим вести не по исходной спецификации, а по фактор-спецификации. Здесь получается своеобразный логический круг. Поэтому такой метод для имплицитной спецификации не столько решает проблему, сколько создаёт новые проблемы.

Другой метод состоит в том, чтобы ограничиться только такими эквивалентностями состояний, которые называются вполне определёнными. Это означает, что у нас просто не должно возникать ситуации, когда стимул определён в одном состоянии, но не определён в другом, эквивалентном ему, состоянии. Во многих практических случаях удаётся найти такую эквивалентность. В частности, нам это удалось в примере с пулом.

Сильно детерминированная реализация.

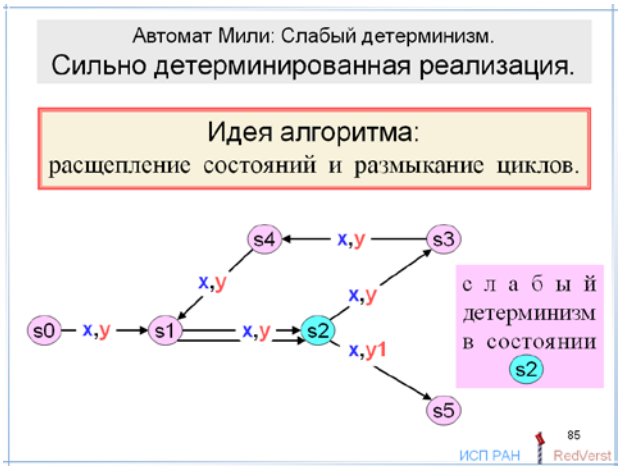
До сих пор мы стремились получить автомат, который был бы конечным, сильно связным и сильно детерминированным, поскольку только для такого автомата у нас был алгоритм обхода. Для этого мы либо предполагали, что спецификационный подавтомат сильно детерминирован, либо, если состояния реализации открыты, использовали сильно детерминированный смешанный или, после оптимизации, реализационный подавтомат. Во всех случаях сама реализация должна быть сильно детерминирована, по крайней мере, на общих трассах.

У нас остался ещё один случай сильно детерминированной реализации и слабо детерминированной спецификации. Это случай тестирования с закрытым состоянием, когда спецификационный подавтомат остаётся слабо детерминированным.

Можно предложить идею алгоритма, который, используя сильный детерминизм реализации, позволял бы по-прежнему строить и обходить сильно детерминированный автомат. Этот автомат получается из слабо детерминированного спецификационного подавтомата с помощью расщепления состояний и размыкания циклов.

Проиллюстрируем эту идею на примере. Пусть в процессе тестирования мы в какой-то момент времени построили вот такой автомат. До последнего перехода мы имели сильный детерминизм, а последний переход делает автомат слабо детерминированным, поскольку в состоянии `s2` в ответ на стимул `x` мы получаем реакцию не `y`, как раньше, а `y1`.

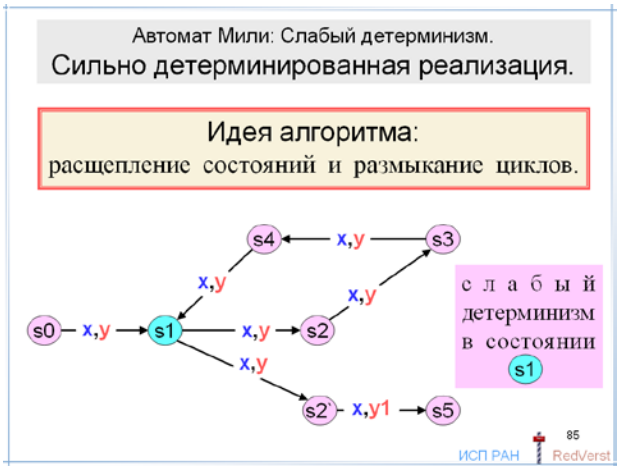
Поскольку мы знаем, что реализация сильно детерминирована, мы можем заключить, что спецификационное состояние `s2` соответствует двум разным реализационным состояниям: в одном состоянии на стимул `x` приходит реакция `y`, а в другом – `y1`.

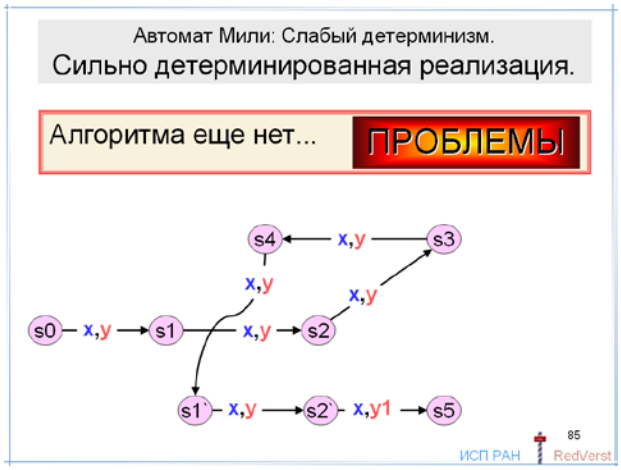


Тогда мы расщепляем состояние `s2` на два состояния, то есть, добавляем состояние `s2`. Соответственно, предпоследний переход теперь будет вести не в состояние `s2`, а в состояние `s2`. Из-за этого, в свою очередь, состояние `s1` становится слабо детерминированным.

Поэтому мы расщепляем и это состояние, добавляя состояние `s1`.

Эту процедуру расщепления состояний мы делаем до тех пор, пока не получится сильно детерминированный автомат. Это произойдёт тогда, когда будет разомкнут цикл переходов.





Для того чтобы можно было так работать, нам нужно помнить не только пройденный автомат, но и полный путь в нём от начального состояния до текущего состояния.

Следует отметить, что это только идея алгоритма. Самого алгоритма ещё нет, здесь могут быть кое-какие проблемы.

Обход по стимулам.

Если реализация не является сильно детерминированной, то никакими ухищрениями нам не удастся обходить сильно детерминированный автомат. Если реализация в некотором состоянии `i` в ответ на стимул `x` может вернуть две разные реакции `y1` и `y2`, то, какому бы состоянию `s` автомата, который мы обходим, ни соответствовало реализационное состояние `i`, мы можем получать в состоянии `s` обе реакции в ответ на стимул `x`.

Выходом, конечно, является подходящая факторизация. Но факторизация – это всегда огрубление тестирования. Может оказаться, что такого огрубления, которое делает автомат сильно детерминированным, мы себе позволить не можем.

Задача гарантированного обхода автомата, который не является сильно детерминированным, очевидно, не имеет решения. Реализация, в зависимости от погоды, может любое число раз подряд выдавать реакцию `y1` и любой конечный тест не сможет определить, возможна ли реакция `y2`, или нет.

Раз такая цель недостижима, приходится довольствоваться другой, достижимой целью.

Мы можем поставить задачу: в каждом состоянии, которого мы смогли достичь при тестировании, попробовать каждый стимул. Такой обход называется обходом по стимулам.

Для обхода по стимулам неизвестного слабо детерминированного автомата необходимо и достаточно двух условий: 1) конечность, 2) сильно-Δ-связность.

Последнее условие означает, что мы можем гарантированно попасть из каждого состояния в каждое другое состояние. Для сильно детерминированного автомата такая цепочка переходов для заданных состояний однозначно определяется последовательностью подаваемых стимулов. Однако, для слабо детерминированного автомата это не так. Здесь требуется подбирать следующий стимул в зависимости от полученной реакции. Такую цепочку иногда называют *адаптивной*. Фактически, это алгоритм, вырабатывающий очередной стимул в зависимости от пройденной трассы.

Мы не будем здесь останавливаться на точном определении сильно-Δ-связности и алгоритме обхода по стимулам. Это вопросы интересные, но технические. Об этом можно прочитать в нашей статье в журнале «Программирование».

Полнота тестирования. Сильно-Δ-связный подавтомат.

Вместо этого посмотрим лучше, как можно вести тестирование на основе такого обхода по стимулам. Мы будем считать, что спецификационный подавтомат конечен, слабо детерминирован и сильно-Δ-связен.

В процессе обхода по стимулам мы можем обнаружить ошибку, и тестирование на этом заканчивается.

Если же обход по стимулам удалось довести до конца, мы можем применять любые методы тестирования по явно заданной модели.

Однако в отличие от сильно детерминированного автомата, здесь мы можем получить новую реакцию в ответ на стимул, который уже давали раньше в данном текущем состоянии. Поскольку обход по стимулам закончен, сильно-Δ-связность гарантирует, что постсостояние этого нового перехода будет одним из уже пройденных состояний. Иными словами, у нас могут появляться только такие новые переходы, которые соединяют старые состояния и помечены старым стимулом, но новой реакцией.

При появлении такого нового перехода мы добавляем его в автомат и продолжаем тестирование, быть может, начиная его заново, поскольку автомат изменился.

Так мы и двигаемся по циклу, пока не найдём ошибку, или пока тестирование не завершится с вердиктом **pass**.

Полнота тестирования. *Не* сильно-Δ-связный подавтомат.

Если спецификационный слабо детерминированный подавтомат оказывается сильно связным, но не сильно-Δ-связным, обход не всегда возможен.

Тест либо завершает обход, либо обнаруживает, что не может его завершить.

После этого мы также можем применять дополнительное тестирование по явно заданной модели.

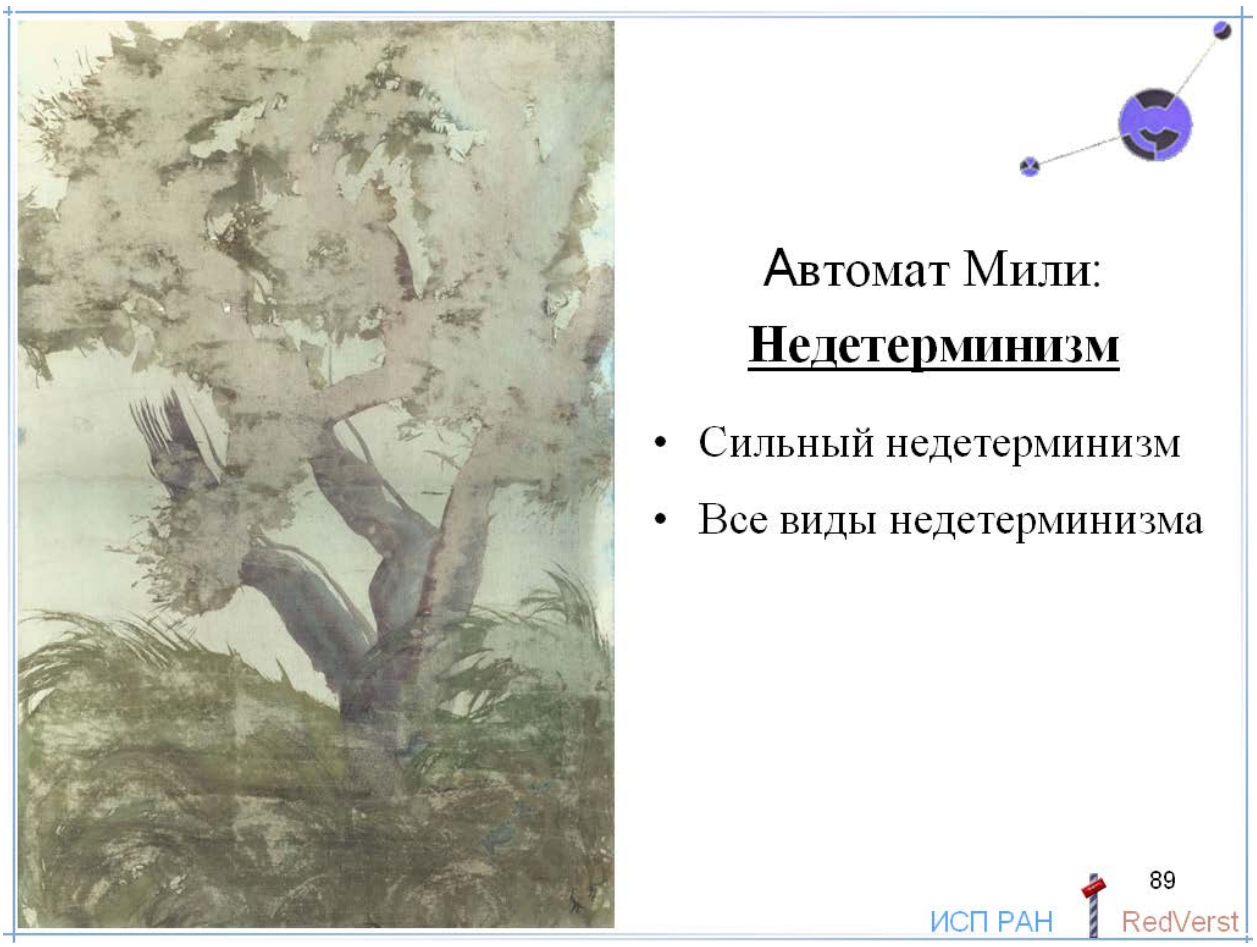
Однако, теперь, получив новую реакцию, мы можем обнаружить, что попали в новое постсостояние.

Тогда продолжаем обход, пока не закончим его или пока это возможно.

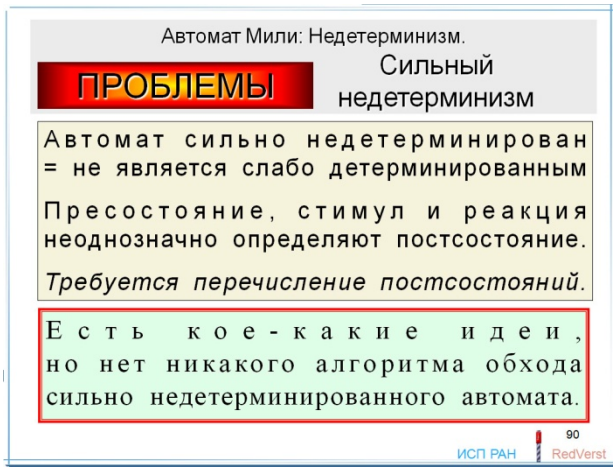
Соответственно, автомат достраивается новыми состояниями и переходами. При этом также могут проходиться новые переходы или даже обнаруживаться ошибки. Если ошибок не обнаружено, продолжаем тестирование.

Следует отметить, что для тестирования по явно заданной модели нам нужно гарантированно попадать в интересующие нас состояния. Если прогоняется несколько тестов с рестартами между ними, то есть, отрывая перо от бумаги, то нам требуется гарантированно попасть в каждое состояние из начального состояния, но не обязательно обратно. Соответственно, требование сильно-Δ-связности ослабляется.

Понятно, что, если мы не можем гарантированно попасть в некоторое нужное нам состояние, то мы не сможем опробовать в этом состоянии все стимулы. В таком случае нужно искать обходные пути: другие реализационные гипотезы или тестовые возможности, например, пытаться управлять погодой и т.п.



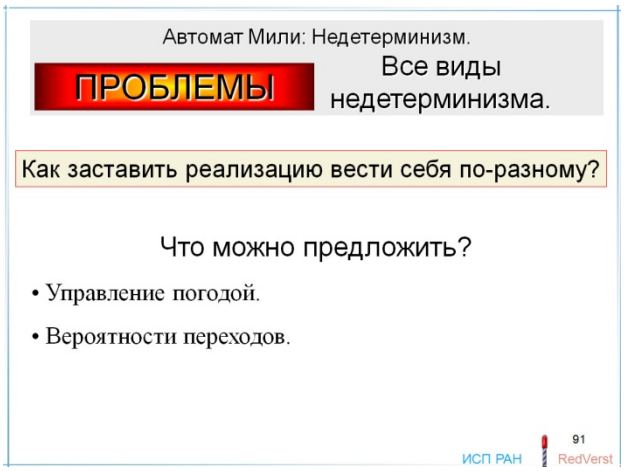
ПРОБЛЕМЫ: Сильный недетерминизм.



Автомат называется сильно недетерминированным, если он не является слабо детерминированным автоматом. Иными словами, пресостояние, стимул и реакция неоднозначно определяют постсостояние.

Для такого автомата, кроме перечисления стимулов, мы должны уметь делать перечисление постсостояний. Есть кое-какие идеи, но нет никакого алгоритма обхода сильно недетерминированного автомата.

ПРОБЛЕМЫ: Все виды недетерминизма.



В общем, для недетерминированной реализации у нас остаётся проблема полноты тестирования, поскольку реализация может не выполнять все те переходы, которые у неё есть.

Как заставить реализацию вести себя по-разному? Можно предложить два варианта.

Первый вариант – это те или иные тестовые возможности по управлению погодой в той или иной степени.

Второй вариант – введение вероятностей переходов и использование их при тестировании. Тогда тест будет проверять нужные переходы с некоторой вероятностью.