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

ДОПУЩЕНИЕ ПОЛНОТЫ

Тестовый контекст. 3
ПРОБЛЕМЫ асинхронного тестирования. 3
Почему соответствие не сохраняется? 4
Проблема самоприменимости спецификации. 5
Продолжение, блокировка и разрушение. 5

βγδ-ТРАССЫ

βγδ-машина тестирования. 8
Оператор параллельной композиции с разрушением. 9
Безопасные трассы. 9
Гипотеза о безопасности и конвергентности. 10
Отношение $ioco_{\beta\gamma\delta}$. 10
Спецификация. 11
Генерация тестов. 11
Конечность теста и перечислимость полного тестового набора. 12
βγδ-трассы: интерфейс теста и медиатора. 16

ПОПОЛНЕНИЕ СПЕЦИФИКАЦИЙ

Переопределение блокировок стимулов. 17
Основные виды пополнений. 18
Классическая семантика отношения *ioco*. Проблема самоприменимости спецификации. 18
Классическая семантика отношения *ioco*. Проблема несохранения соответствия. 20
Комбинированный вариант группы RedVerst. 20

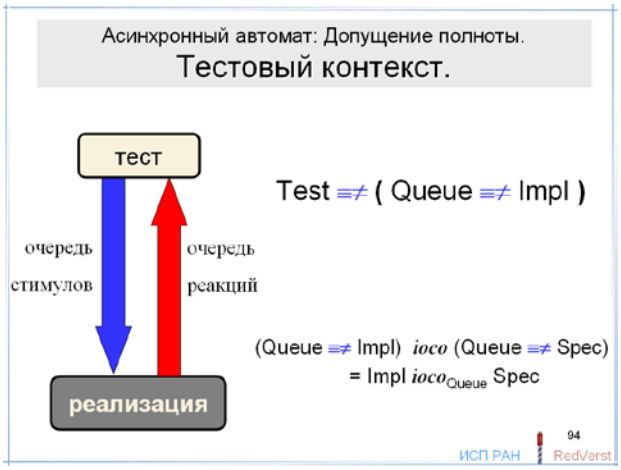
СПЕЦИФИКАЦИЯ В ПРЕД- И ПОСТУСЛОВИЯХ

Пред- и постусловия. 24
Спецификация без τ-переходов. 25
Проблема ложной стационарности. 26
Спецификация разрушения. 26
Блокировка в стационарных состояниях. 27
Привязанные реакции. 28

Тестовый контекст.

Напомним, что асинхронное тестирование – это тестирование через посредника – среду передачи.

Мы рассматриваем случай асинхронного тестирования, когда среда передачи – это две неограниченные очереди: входная очередь стимулов и выходная очередь реакций.



ПРОБЛЕМЫ асинхронного тестирования.

Как мы уже говорили, с асинхронным тестированием связаны две основные проблемы.

Проблема *вседозволенности* заключается в том, что некоторые ошибки, которые обнаруживаются при синхронном тестировании, не могут быть обнаружены при асинхронном тестировании. Иными словами, соответствие реализации и спецификации ослабляется. Это нормальная ситуация, она означает, что асинхронное тестирование менее точное, чем синхронное. Возмущение вносит среда передачи, она рассинхронизовывает тест и реализацию.

Вторая проблема – это проблема *несохранения соответствия*: асинхронное тестирование может обнаружить ошибку, которая не обнаруживается при синхронном тестировании. В этом случае происходит незаконное усиление соответствия. В чём причина несохранения соответствия?

Почему соответствие не сохраняется?

Причина, по которой соответствие не сохраняется, заключается в следующем: неспецифицированный стимул не подаётся в реализацию при *синхронном* тестировании, но может быть подан в неё при *асинхронном* тестировании.

Именно это мы и наблюдали в нашем примере.

Почему так происходит?

Дело в том, что при асинхронном тестировании используется не исходная спецификация, а её композиция со средой. Для неограниченной входной очереди такая композиция оказывается всюду определённой по стимулам, то есть в ней нет неспецифицированных стимулов. Поэтому-то все стимулы и подаются в реализацию.

Тестирование по всюду определённой по стимулам спецификации называется *строгим*. Если же спецификация не всюду определённая, то при тестировании неспецифицированные стимулы не подаются и, тем самым, не проверяется поведение реализации на этих стимулах. Такое тестирование называется *слабым*.

Таким образом, причина несохранения соответствия заключается в том, что синхронное тестирование оказывается слабым, а асинхронное – сильным.

Если бы исходная спецификация была всюду определённой по стимулам, то проблемы несохранения соответствия не было бы.

Это наводит на мысль доопределить в спецификации все неспецифицированные стимулы.

Самый главный вопрос, который здесь возникает: что означает неспецифицированный стимул?

Можно считать, что если стимул не специфицирован, то это означает некоторое поведение по этому стимулу, которое подразумевается по умолчанию. Модель асинхронного автомата должна быть уточнена однозначной интерпретацией неспецифицированного стимула.

Такого рода интерпретации называются *допущениями полноты*, а соответствующая процедура изменения спецификации – её *пополнением*.

Проблема самоприменимости спецификации.

Проблема неспецифицированных стимулов связана не только с проблемой несохранения соответствия, но и с проблемой *самоприменимости спецификации*. Эту проблему можно ещё назвать проблемой понимания спецификации разработчиком реализации.

Казалось бы, спецификацию можно рассматривать как одну из возможных реализаций. Это самая полная реализация, если иметь в виду, что в ней реализовано не только то, что должно быть реализовано, но и всё то, что может быть реализовано, то есть все те возможности, которые спецификация предлагает на выбор, по принципу *may&must*.

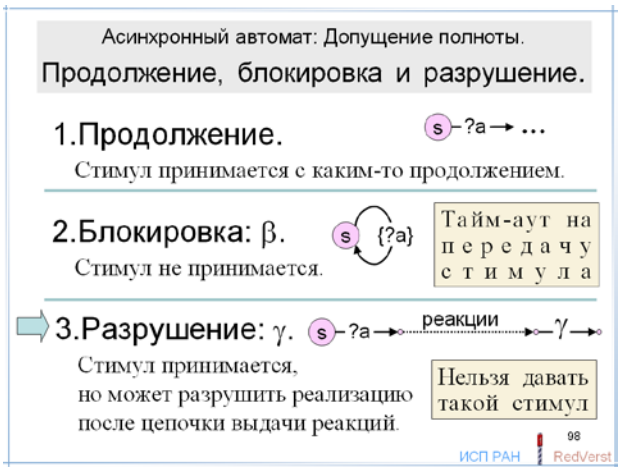
Разработчик, казалось бы, имеет право создать реализацию совпадающей со спецификацией. Если в стабильном состоянии стимул не специфицирован, то разработчик свободен в выборе, принимать стимул и что-то делать после его приёма или не принимать, блокировать стимул.

Однако для отношения ***ioco*** мы запретили блокирующий deadlock: реализация не может блокировать стимул в стабильном состоянии. Поэтому формально спецификация, имеющая неспецифицированные стимулы в стабильных состояниях, не может быть своей собственной реализацией.

Если же снять запрет блокирующего deadlock`а, то спецификация будет не конформна сама себе по отношению ***ioco***. Действительно, в недетерминированной спецификации некоторая трасса σ может закончиться в двух разных стабильных состояниях. При этом может оказаться, что в одном состоянии некоторый стимул x определён, а в другом – нет. В этом случае ***ioco*** требует, чтобы после трассы σ принимался стимул x, и при тестировании мы можем дать этот стимул, но принимается он только в одном состоянии, а в другом – блокируется, и требования отношения ***ioco*** не выполнены.

Поэтому и здесь возникает необходимость однозначной интерпретации неспецифицированного стимула, то есть того или иного допущения полноты.

Продолжение, блокировка и разрушение.



Существует три типа допущений полноты.

Стимул принимается. Предполагается, что неспецифицированный стимул можно подавать в реализацию, и реализация его принимает. Поведение после приёма стимула может быть различным. Дальше мы рассмотрим несколько вариантов такого поведения. В каждом из этих вариантов в спецификации определяется переход по приёму стимула с нужным дальнейшим поведением.

Блокировка. До сих пор мы запрещали блокирующий deadlock. Однако запрет на блокировку стимулов реализацией делает невозможным тестирование систем, в функциональность которых входит такая блокировка. Примером может служить ограниченная FIFO-очередь: передача стимула – это помещение символа в конец очереди, передача реакции – выборка символа из головы очереди.

Допущение полноты

Спецификация очереди требует, чтобы стимул блокировался тогда и только тогда, когда очередь полностью заполнена. Такая очередь часто реализуется как очередь сообщений, когда операция *послать* (send) завершается с соответствующим кодом ответа, если сообщение не может быть передано немедленно. Иногда операция *послать* снабжается тайм-аутом (встроенным или передаваемым как параметр). Если сообщение не передано до истечения тайм-аута, операция *послать* заканчивается с соответствующим кодом ответа, что соответствует абстракции θ-действия.

Блокировка. Стимул можно подавать в реализацию, но реализация не принимает стимул. Возникает блокирующий deadlock. В этом случае никакого перехода по стимулу добавить в спецификацию нельзя. Поэтому этот вид умолчания можно считать в некотором смысле основным, а остальные умолчания интерпретировать как переопределение блокировки.

При тестировании конвергентной реализации блокировку можно обнаружить с помощью тайм-аута на передачу стимула.

Разрушение. Следующий тип умолчания предложен в группе RedVerst, хотя о такой семантике неспецифицированного стимула, как об одной из возможных, говорится в обзоре Lee и Yannakakis, посвящённом тестированию конечных автоматов Мили. Этот тип умолчания основан на понимании предусловия как абсолютного запрета на передачу в реализацию стимулов, нарушающих предусловие. Предполагается, что поведение реализации по поводу этого стимула полностью не определено: она может его блокировать, принимать с тем или иным продолжением, но самое главное, реализация может быть *разрушена*, что считается недопустимым. В некоторых языках спецификации для такого поведения имеется соответствующее ключевое слово, например, в RSL (Raise Specification Language) – это слово **swap**.

Исключение из рассмотрения блокируемых и разрушающих стимулов часто мотивируют тем, что реализация после приёма стимула должна проверять его корректность. Если стимул некорректен, реализация либо просто игнорирует его, либо сообщает об ошибке ответной реакцией. Такое требование к реализации естественно, если это система «общего пользования», в ней должна быть предусмотрена «защита от дурака». Однако часто требуется тестировать внутренние компоненты или подсистемы, доступ к которым строго ограничен и взаимные проверки корректности обращений излишни. Такое часто встречается для стимулов со сложной внутренней структурой и нетривиальными условиями корректности, когда накладные расходы на проверку неоправданно увеличивают трудозатраты на создание системы, её объём и время выполнения. Альтернативой в этом случае является строгая спецификация предусловий взаимодействующих компонентов. Тестированию подлежит не поведение компонентов в ответ на некорректные стимулы, а правильность обращения компонентов друг к другу. Последнее означает, фактически, проверку поведения каждого компонента (по его постусловию) только для таких стимулов, которые удовлетворяют предусловию компонента.

Есть и ещё класс ситуаций, когда полезно специфицировать разрушение. Таким способом мы можем маркировать то поведение реализации, которое по каким-то причинам не хотим тестировать. Причиной может быть незаконченность реализации – какие-то части пока не полностью реализованы. Другая причина – желание проверить только часть поведения, но зато максимально полно; здесь мы пытаемся сократить время тестирования за счёт исключения из него неинтересующей нас части поведения. При тестировании, совмещённом с реальной работой системы, мы можем исключать поведение, перегружающее систему и ставящее под угрозу реальную работу. Например, можно тестировать распределение памяти, но нельзя приводить систему в состояние критически малого объёма свободной памяти, хотя поведение системы в этой ситуации может быть особенно интересным в другом отношении.

Разрушение. Стимул принимается реализацией, но это может привести к её разрушению. Разрушение не обязательно происходит сразу после приёма стимула. Возможно, перед этим выполняется выдача некоторых реакций. Здесь предполагается, что после приёма стимула реакции выдаются по инициативе самой реализации, а окружение может вести себя пассивно, то есть, окружение только принимает эти реакции. Таким образом, можно считать, что разрушение инициировано передачей в реализацию этого стимула. Разрушение мы предлагаем моделировать переходом, помеченным специальным символом γ.

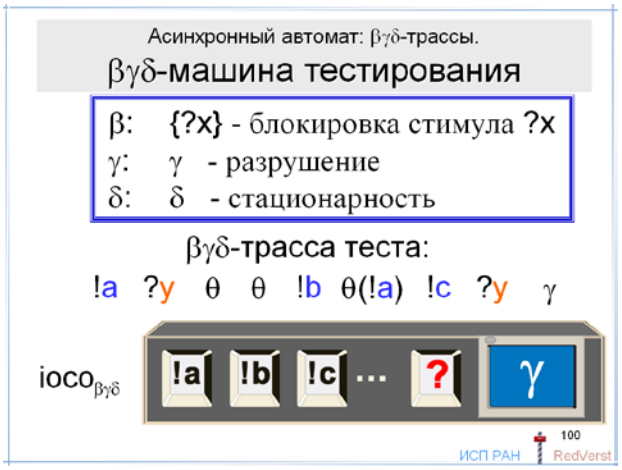
Стимул, который может привести к разрушению, мы не должны подавать в реализацию при тестировании. Именно интерпретация неспецифицированного стимула как разрушающего позволяет оставить тестирование слабым.

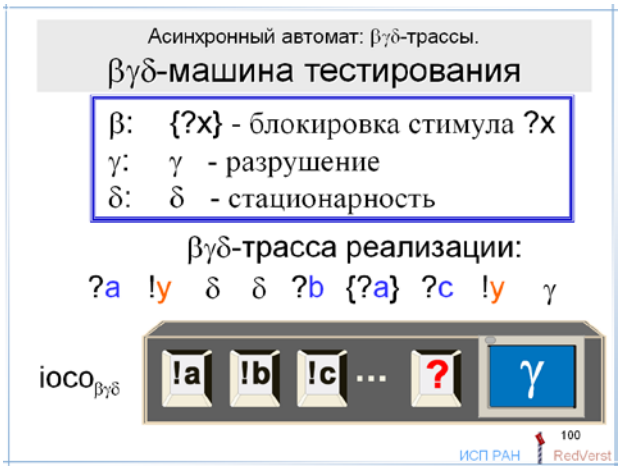
Мы будем считать, что тестируемая система не разрушается «сама по себе», то есть без приёма каких бы то ни было стимулов. Такой асинхронный автомат будем называть *безопасным*.

βγδ-машина тестирования.

Теперь мы можем рассматривать наблюдения, в которые, кроме внешних действий, входят не только стационарность, но также блокировки стимулов и разрушение. Стационарность изображается символом δ, разрушение – символом γ, а блокировка стимула `?x` – стимулом `?x` в фигурных скобках `{?x}`.

Трассы таких обогащённых наблюдений мы будем называть βγδ-трассами, а соответствие ***ioco*** заменяется более общим соответствием $\boldsymbol{ioco}_{\beta\gamma\delta}$.





Сначала посмотрим, как устроена соответствующая машина тестирования, формализующая тестовые возможности, необходимые для такого рода наблюдений. Мы будем называть её βγδ-машиной.

Эта машина похожа на машину тестирования для трасс с задержками.

Результатом эксперимента является βγδ-трасса, которую проходит тест.
Передача стимулов и приём реакций происходит так же, за исключением двух случаев.

Во-первых, в ответ на посылку стимула мы можем получить истечение тайм-аута, что означает блокировку стимула.

Во-вторых, мы можем получить разрушение машины.

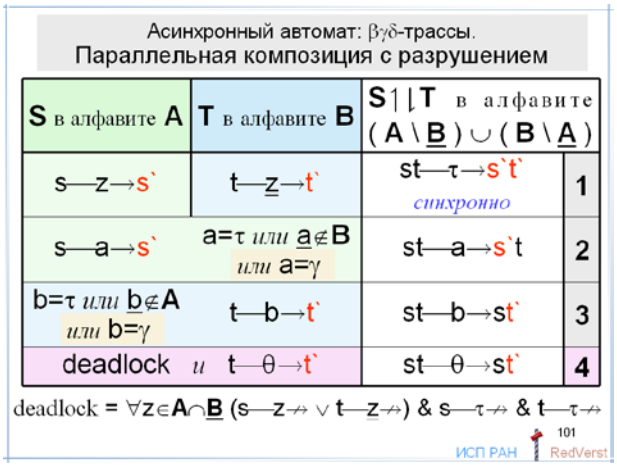
В трассе теста мы различаем переход по θ в двух типах состояний. В принимающем состоянии – это по-прежнему стационарность, а в посылающем состоянии – блокировка посылаемого стимула.

Чтобы получить трассу реализации, нужно заменить
- вопросительный знак восклицательным и наоборот,
- символ θ в принимающем состоянии – символом δ,
- а в посылающем состоянии – блокировкой посылаемого стимула.

Оператор параллельной композиции с разрушением.

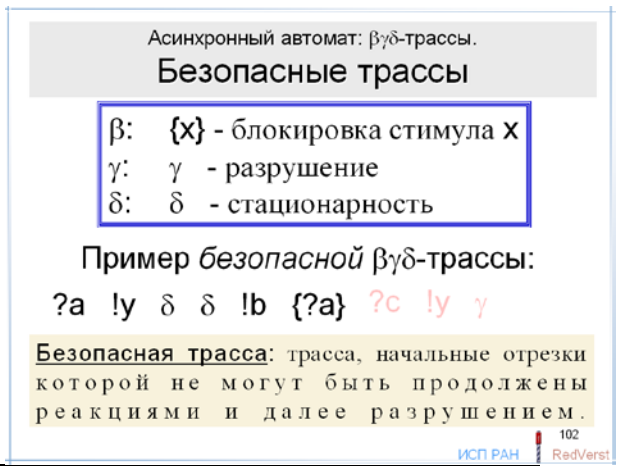
Посмотрим, что меняется в операторе параллельной композиции при добавлении нового типа перехода – перехода по разрушению?

Наиболее отвечает семантике разрушения асинхронное выполнение перехода по разрушению. В этом разрушение аналогично внутреннему переходу или переходу по внешнему стимулу или реакции, которые есть в алфавите одного автомата, а противоположные (по подчёркиванию) им символы отсутствуют в алфавите другого автомата.



Безопасные трассы.

Тестирование должно быть безопасным, то есть оно не должно приводить к разрушению реализации. Это означает, что при тестировании мы должны проходить только *безопасные* трассы – трассы, начальные отрезки которых которых не могут быть продолжены реакциями и далее разрушением.



βδγ-трасса: `?a !y δ δ !b {?a} ?c !y γ` не безопасна: приём стимула `c` приводит к разрушению. Её максимальный безопасный начальный отрезок: `?a !y δ δ !b {?a}`.

Гипотеза о безопасности и конвергентности.

Итак, любая трасса, которая может быть получена при тестировании, должна быть безопасна в реализации.

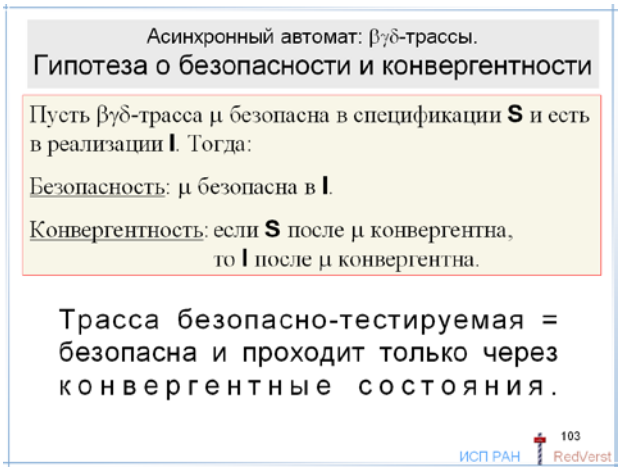
Кроме этого, мы должны быть уверены, что истечение тайм-аута действительно означает блокировку в посылающем состоянии теста или стационарность в принимающем состоянии. Для этого каждая тестируемая трасса не должна проходить через дивергентные состояния, а время перехода по стимулу или реакции так же, как цепочки τ-переходов, должно быть ограничено сверху. Иными словами, если в реализации есть дивергенция, то она должна быть недостижима по трассам, используемым в тестировании.

Трассу, которая безопасна и проходит только через конвергентные состояния, назовём *безопасно-тестируемой*.

О безопасно-тестируемости трасс реализации мы можем судить только на основании спецификации.

Для этого принимается реализационная гипотеза, которая называется *гипотезой о безопасности и конвергентности*:

Если βγδ-трасса безопасна в спецификации и имеется в реализации, то
безопасность: трасса безопасна в реализации,
конвергентность: реализация после этой трасса конвергентна, если спецификация после этой трассы конвергентна.



Поскольку продолжение безопасной трассы реакцией или стационарностью оставляет трассу безопасной, условие безопасности эквивалентно следующему: после общей трассы, безопасной в спецификации, любой стимул, безопасный в спецификации, безопасен в реализации.

Отношение *ioco*$_{βγδ}$.

Соответственно меняется определение соответствия для **ioco**$_{βγδ}$:

Если трасса безопасна в спецификации и есть в реализации, то после этой трассы реализация может ... тогда и только тогда, когда это возможно в спецификации после этой же трассы.
Под многоточием ... понимается теперь одно из четырёх:

1) принимать стимул ?x, безопасный после трассы в спецификации;
2) блокировать стимул {?x}, безопасный после трассы в спецификации;
3) выдавать реакцию !y;
4) не выдавать никаких реакций, то есть находиться в стационарном состоянии δ.

Спецификация.

Для этого нового соответствия мы можем использовать при пополнении спецификации любое допущение полноты: не только продолжение, но также блокировку и разрушение. При любом таком допущении проблема самоприменимости спецификации решается автоматически.

Для того чтобы тестирование по безопасным трассам спецификации было возможным, требуется

1) чтобы такие трассы были и
2) чтобы после таких трасс не было дивергенции.

Безопасность: Спецификация должна быть безопасной (не саморазрушающейся).

В противном случае множество тестов на соответствие этой спецификации пусто, а сама спецификация вырожденная – не предъявляет к реализации никаких функциональных требований. В частности, реализация может быть саморазрушающейся, и при любой попытке тестирования, ещё до начала собственно тестирования, реализация может разрушиться.

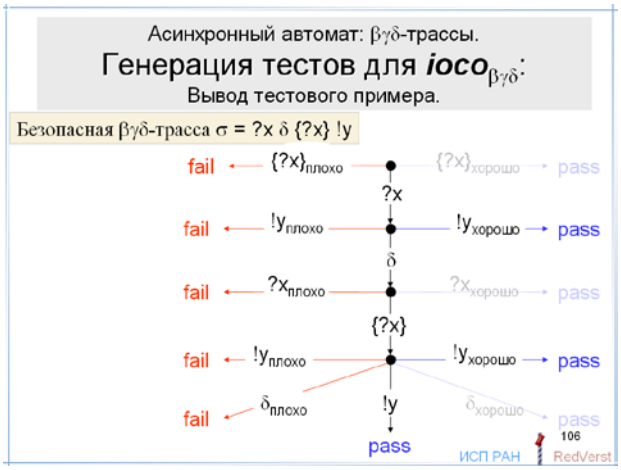
Безопасно-конвергентность: Все состояния спецификации, достижимые по безопасным трассам, должны быть конвергентны.

Иными словами, если в спецификации есть дивергенция, то она недостижима по безопасным трассам, используемым в тестировании.

Генерация тестов.

Генерация тестов для ***ioco***$_{\beta\gamma\delta}$ отличается от генерации тестов для ***ioco*** возможностью блокировки стимулов.

Рассмотрим генерацию теста для βγδ-трассы, которая отличается от той, что у нас была для соответствия ***ioco*** заменой одного стимула на его блокировку.



Когда тест в посылающем состоянии выдаёт стимул в реализацию, он устанавливает тайм-аут, а по истечении тайм-аута считает, что реализация блокировала стимул.

βγδ-трассы

Пусть некоторый начальный отрезок трассы продолжается в трассе приёмом стимула. В спецификации этот начальный отрезок может иметь продолжение блокировкой стимула (в другом состоянии спецификации, естественно) или продолжаться только приёмом стимула.

В случае разрешения блокировки стимула в посылающем состоянии теста делается переход по блокировке стимула в состояние ***pass***, а в случае, когда стимул может только приниматься, – в состояние ***fail***.

Пусть теперь некоторый начальный отрезок трассы продолжается в трассе блокировкой стимула. В спецификации этот начальный отрезок может иметь продолжение приёмом стимула (в другом состоянии спецификации, естественно) или продолжаться только блокировкой стимула.

В случае разрешения приёма стимула в посылающем состоянии теста делается переход по приёму стимула в состояние ***pass***, а в случае, когда стимул может только блокироваться, – в состояние ***fail***.

В конце «инвертируем» все символы: приём стимула `?x` заменяется выдачей стимула `!x`, а выдача реакции `!y` заменяется приёмом реакции `?y`. Кроме того, символ δ и блокировки стимулов заменяются символом θ. Интерпретация символа θ зависит от состояния теста: в принимающем состоянии – это стационарность δ, а в состоянии, посылающем стимул, это блокировка этого стимула.

Конечность теста и перечислимость полного тестового набора.

Повторим, что для безопасного тестирования спецификация должна быть безопасной и безопасно-конвергентной.

Как мы уже говорили, с практической точки зрения нам требуются две вещи: 1) перечислимость полного тестового набора и 2) конечность теста.

Спецификация трасс.

Поскольку тест строится по каждой βγδ-трассе, которая есть и безопасна в спецификации, число таких трасс должно быть перечислимо. Для конечных трасс это эквивалентно перечислимости множества реакций и безопасных стимулов после каждой безопасной трассы σ. Заметим, что стимулов, которые опасны после трассы, может быть произвольное количество; эти стимулы (так же, как их блокировки) всё равно не используются при построении безопасных трасс.

1) Будем считать, что задан итератор `SI(σ)` – алгоритм перечисления реакций и безопасных стимулов после трассы σ.

Итератор, возвращая безопасный стимул, не сообщает, принимается этот стимул, блокируется или и то и другое (естественно, в разных состояниях).

Заметим, что такой итератор не перечисляет стимулы, которые являются разрушающими после трассы. Такой разрушающий стимул всегда принимается после трассы (с возможным последующим разрушением), но может также блокироваться после трассы.

βγδ-трассы

Для построения теста мы должны уметь после каждой βγδ-трассы за конечное время определить, продолжается ли эта трасса в спецификации данной реакцией, стационарностью, безопасным стимулом или его блокировкой. Поэтому множество этих символов должно быть разрешимо.

2) Будем считать, что задан алгоритм $P(\sigma,z)$, который за конечное время проверяет, имеет ли трасса σ продолжение реакций z=!y, безопасным стимулом z=?x или блокировкой безопасного стимула z={?x}.

3) Также будем считать, что задан алгоритм $P_\delta(\sigma)$, который за конечное время проверяет, имеет ли трасса σ продолжение стационарностью δ.

Итератор безопасных βγδ-трасс строится так. Для каждой трассы итератор реакций и безопасных стимулов после трассы задаёт нумерацию реакций, безопасных стимулов и их блокировок, которыми трасса может продолжаться. Заметим, что номер символа определяется не только самим символом, но и предыдущей трассой. Если трасса продолжается символом δ, то этому символу присвоим номер 1, а символы, возвращаемые итератором, будем нумеровать, начиная с 2. Если итератор возвращает реакцию, то присвоим ей очередной номер n. Если итератор возвращает безопасный стимул, то трасса продолжается этим стимулом и/или его блокировкой. Если трасса продолжается только стимулом, то присвоим ему очередной номер n; если трасса продолжается только блокировкой стимула, то этой блокировке присвоим номер n; если трасса продолжается и тем и другим, то присвоим стимулу номер n, а его блокировке номер n+1.

Индексом трассы назовём сумму номеров её символов. Очевидно, что число трасс с данным индексом конечно, и его можно перебрать алгоритмически. Тогда итерация трасс реализуется двумя вложенными циклами: во внешнем цикле перечисляем индекс, а во внутреннем – трассы с этим индексом.

При построении теста по трассе мы используем алгоритм $P(\sigma,z)$ для проверки:

1) каждой реакции z=!y, получаемой от реализации в принимающем состоянии теста,
2) каждого безопасного стимула z=?x, принимаемого реализацией в посылающем состоянии теста, когда следующий в трассе символ – это блокировка {?x},
3) блокировки каждого безопасного стимула z={?x} в посылающем состоянии теста, когда следующий в трассе символ – это стимул ?x.

Также используем алгоритм $P_\delta(\sigma)$ – для проверки стационарности, то есть истечение тайм-аута в принимающем состоянии теста, когда трасса продолжается реакцией.

Спецификация автомата. Безопасные стимулы (принимаемые и блокируемые).

Этих трёх требований нам было бы достаточно, если бы спецификация задавалась как множество βγδ-трасс. Однако наша модель – это асинхронный автомат. Поэтому мы должны переформулировать наши требования в терминах состояний и переходов.

Вместо перечислимости реакций и безопасных стимулов *после безопасной трассы*, мы должны говорить о перечислимости реакций и безопасных стимулов *в состоянии, достижимом по безопасной трассе*.

βγδ-трассы

1) Будем считать, что задан итератор `SI(s)`, перечисляющий реакции и стимулы, безопасные в состоянии s.

Итератор, возвращая безопасный стимул, не сообщает, принимается этот стимул или блокируется.

Заметим, что такой итератор не перечисляет стимулы, которые являются разрушающими в состоянии. Стимул, разрушающий в состоянии, естественно принимается в этом состоянии, поэтому блокировки этого стимула в этом состоянии быть не может.

Для перечислимости трасс нам было бы достаточно перечислимости переходов по каждой реакции и каждому безопасному стимулу в каждом состоянии.

Однако для проверки реакций, безопасных стимулов и их блокировок мы должны уметь алгоритмически проверять, имеется ли в состоянии s переход по данному безопасному символу. Теперь нам нужно за конечное время проверять каждый безопасный символ после трассы, опираясь на проверку безопасных символов во всех состояниях, достижимых по этой трассе. Это возможно только в том случае, когда трасса заканчивается в конечном числе состояний. Такое требование эквивалентно конечности числа переходов по каждому стимулу или реакции в каждом состоянии. Поэтому нам нужен соответствующий итератор переходов в состоянии.

Кроме того, нужно учитывать τ-переходы, поскольку они не меняют трассу. У нас должны быть конечными, во-первых, число τ-переходов из каждого состояния, достижимого по безопасной трассе, и, во-вторых, число состояний достижимых из такого состояния по τ-переходам. Второе условие перекрывается требованием безопасно-конвергентности.

2) Будем считать, что задан алгоритм перебора переходов по реакциям и безопасным стимулам в состоянии – итератор `TI(s,z)`.

Чтобы проверить реакцию или безопасный стимул z в состоянии s теперь достаточно вызвать итератор переходов по z в состоянии s и проверить, что он возвращает хотя бы один переход. Чтобы проверить блокировку безопасного стимула x в состоянии s теперь достаточно вызвать итератор переходов по x в состоянии s и проверить, что он не возвращает ни одного перехода.

Однако стационарность состояния означает, что в нём нет переходов ни по одной реакции из, быть может, бесконечного множества реакций. Поэтому нам требуется отдельный алгоритм проверки стационарности.

3) Будем считать, что задан алгоритм $P_\delta(s)$, проверяющий, что в состоянии s, достижимом по безопасной трассе, нет переходов по реакциям, то есть стационарность состояния.

4) Стимул, безопасный в одном состоянии после безопасной трассы, может оказаться опасным в другом состоянии после этой же трассы. Поэтому, кроме перечисления стимулов, безопасных в состоянии, нам нужна проверка безопасности стимула в другом состоянии. Должен быть задан алгоритм $P_\gamma(s,x)$, проверяющий безопасность стимула x в состоянии s, достижимом по безопасной трассе.

Итерация безопасных символов после безопасной трассы сводится к их итерации реакций и безопасных стимулов в конечном множестве состояний, в котором эта трасса заканчивается, и проверке стационарности в этих состояниях. Для этого с каждым состоянием этого множества связываем итератор символов в этом состоянии. Все эти итераторы связываем в цикл и

вызываем по циклу. Когда итератор символов в состоянии возвращает очередной символ, мы проверяем, а не было ли такого символа раньше, и безопасен ли он во всём множестве состояний. Стимул безопасен во множестве состояний, если он безопасен в каждом состоянии этого множества. Если символ, возвращаемый итератором, уже был, или оказался опасным, или итератор сообщил об окончании итерации, мы переходим к следующему по циклу итератору. Итерация символов во множестве состояний заканчивается, когда итераторы символов во всех состояниях множества закончили итерацию. Заметим, что, выбирая безопасный стимул, мы должны дополнительно проверить, принимается он в данном состоянии или блокируется. Это делается с помощью итератора переходов. Если стимул блокируется, трасса может продолжаться блокировкой этого стимула. После такого продолжения мы переходим к подмножеству состояний, в которых этот стимул блокируется. Для того чтобы продолжить трассу стационарностью, нужно проверить стационарность во всех состояниях множества. Если хотя бы одно из состояний стационарно, трасса может продолжаться символом δ. После такого продолжения мы переходим к подмножеству стационарных состояний.

Рассмотренный способ задания автомата весьма специфический: явно указываются (специфицируются) не только принимаемые безопасные стимулы, но и блокировки безопасных стимулов. В то же время для разрушающих стимулов указывается только тот факт, что они разрушающие. Куда ведут переходы по разрушающему стимулу, несущественно, важно лишь, что из постсостояния хотя бы одного перехода по данному стимулу, двигаясь по τ-переходам и переходам по реакциям, можно достигнуть разрушения. Тем не менее, такой способ задания спецификационного автомата полезен на практике.

Спецификация автомата. Принимаемые стимулы (безопасные и разрушающие).

Теперь мы рассмотрим альтернативный (и для нашей модели теоретически более естественный) способ, когда явно задаются (специфицируются) только переходы по стимулам и реакциям, определённые в состоянии, в том числе по разрушающим стимулам, а блокировки – это блокировки всех тех стимулов, переходы по которым не заданы, то есть стимулов, неспецифицированных в состоянии.

В этом случае все неспецифицированные стимулы блокируются в данном состоянии и, тем самым, безопасны в нём. Для специфицированного стимула мы должны проверить его безопасность. Таким образом, общее число стимулов, как безопасных в состоянии, так и разрушающих в состоянии, должно быть перечислимо.

Стимул опасен в состоянии, если из постсостояния хотя бы одного перехода по данному стимулу, двигаясь по τ-переходам и переходам по реакциям, можно достигнуть разрушения, то есть γ-перехода.

Гамма-состоянием будем называть состояние, в котором определён такой переход по разрушению.

Для того чтобы можно было проверить безопасность стимула в состоянии, требуется конечность числа состояний, которые можно достигнуть из каждого состояния, достижимого по безопасной трассе + стимул, двигаясь далее по маршруту из τ-переходов и переходов по реакциям. Для каждого такого маршрута мы проверяем, не заканчивается ли он в гамма-состоянии.

Поскольку нас не интересуют маршруты, которые проходят через гамма-состояния и двигаются дальше, это требование можно ослабить: нужно учитывать только те маршруты, которые не проходят через гамма-состояния, но могут заканчиваться в гамма-состояниях.

Это свойство можно назвать *регулярностью по реакциям* по аналогии с регулярными множествами последовательностей, порождаемыми конечными автоматами. В частности, число реакций, по которым определены переходы из данного состояния, должно быть конечным.

Таким образом, мы должны иметь два итератора:

итератор `XI(s)` перечисляет не более чем счётное множество стимулов, определённых в состоянии,

а итератор `YI(s)` перечисляет конечное множество реакций, определённых в состоянии.

Итератор переходов `TI(s,z)` по-прежнему требуется.

Поскольку число реакций в состоянии конечно, нам не требуется отдельный алгоритм проверки стационарности состояния. Для такой проверки достаточно итератором переходов проверить отсутствие τ-переходов, а итератором реакций – отсутствие реакций.

Для проверки безопасности стимула в конечном множестве состояний U нам теперь не нужен специальный алгоритм $P_\gamma(s,x)$. Вместо этого мы вычисляем с помощью итератора реакций и итератора переходов конечное множество состояний, достижимых из состояний множества U по τ-переходам и переходам по реакциям. Нам нужно лишь проверить наличие или отсутствие γ-перехода в каждом из этих состояний. Для этого достаточно, чтобы алгоритм `TI(s,γ)` мог перечислять также все переходы из состояния s по разрушению γ, хотя нам достаточно знать только, пусто или не пусто это множество переходов.

Конечно-безопасно-ветвящийся автомат.

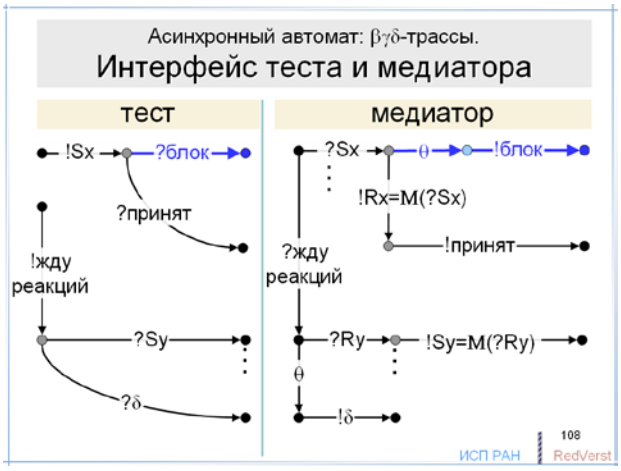
Как частный широко распространённый случай спецификаций, для которых выполнены последние условия, можно рассматривать спецификации, которые:

1. Конечно-безопасно-ветвящиеся, то есть такие, в каждом состоянии которых, достижимом по безопасной трассе, определено конечное число переходов. А в каждом состоянии, достижимом по безопасной трассе + стимул, определено конечное число τ-переходов и переходов по реакциям. И задан итератор `TI(s)` всех переходов в таких состояниях.
2. Регулярны по реакциям.
3. Имеют перечислимое множество стимулов и задан итератор стимулов `X`.

βγδ-трассы: интерфейс теста и медиатора.

Теперь мы сняли запрет блокирующего deadlock`a, и поэтому меняется интерфейс теста и медиатора.

Здесь добавляется θ-переход в медиаторе при попытке передачи стимула в реализацию. Если происходит блокировка, медиатор сообщает об этом тесту.



Переопределение блокировок стимулов.

В самом общем виде пополнение спецификаций определяется как отображение множества спецификаций в себя:

Comp: Spec → Spec.

Соответственно отношение **ioco** связывает реализацию не с исходной, а с пополненной спецификацией:

Impl **ioco**$_{\beta\gamma\delta}$ Spec.

Мы говорили, что пополнение спецификаций следует понимать как переопределение всех или некоторых блокировок стимулов. Поэтому пополнение спецификаций как отображение должно удовлетворять следующим двум требованиям:

1. Старые βγδ-трассы без блокировок сохраняются.
2. Добавляются только такие βγδ-трассы, которые имеют вид μ·?x·λ, где βγδ-трасса μ была раньше и заканчивалась хотя бы в одном состоянии, в котором не был специфицирован стимул x (была трасса μ·{?x}).

Различают два вида пополнений: 1) пополнение состояний и 2) пополнение трасс.

При пополнении состояний переопределяются блокировки стимулов в каждом отдельно взятом состоянии.

При пополнении трасс переопределяются блокировки стимулов, которыми может непосредственно продолжаться каждая отдельно взятая трасса.

Можно показать, что если спецификационный автомат нужным образом преобразовать, то пополнение состояний преобразованного автомата эквивалентно пополнению трасс исходного автомата. Поэтому дальше мы будем рассматривать пополнения состояний.

Основные виды пополнений.

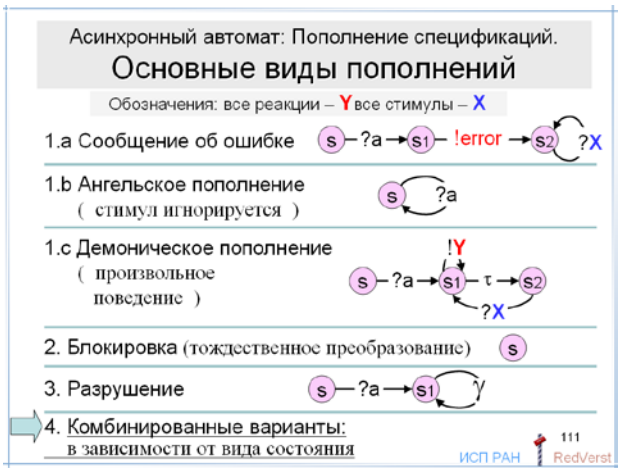
Рассмотрим несколько видов пополнения спецификации.

Продолжение. Для первого типа допущения полноты, когда стимул принимается без разрушения, пополнение определяется дальнейшим, после приёма стимула, поведением автомата.

Автомат может после приёма стимула выдать специальное *сообщение об ошибке*, то есть реакцию, и перейти в специальное ошибочное состояние, игнорируя в дальнейшем все стимулы.

Если автомат просто игнорирует принимаемый стимул, сохраняя своё состояние, то такое пополнение называют *ангельским*.

Демоническое пополнение определяет после приёма стимула произвольное поведение, то есть любую последовательность принимаемых стимулов, выдаваемых реакций и стационарности, но без блокировок стимулов и без разрушения.



Блокировка. Второй тип допущения полноты сохраняет блокировку стимула. Фактически, соответствующее пополнение спецификации – тождественное преобразование, ничего не меняющее.

Разрушение. Третий тип допущения полноты – разрушение после приёма стимула. Оно моделируется γ-переходом.

Кроме этих основных видов пополнений, рассматриваются также различные комбинированные варианты. Блокировка стимула переопределяется в зависимости от вида состояния.

Классическая семантика отношения *ioco*. Проблема самоприменимости спецификации.

Теперь мы посмотрим, какое пополнение спецификации сохраняет классическую семантику отношения **`ioco`** (не $\boldsymbol{ioco}_{\beta\gamma\delta}$, а просто ***ioco***). Прежде всего, заметим, что такое пополнение должно удовлетворять двум правилам.

Первое правило – это правило приоритета приёма стимула над его блокировкой. Пусть в спецификации трасса μ непосредственно продолжается как приёмом стимула x, так и его блокировкой (естественно, в разных состояниях). Тогда, по определению отношения ***ioco***, реализация, в которой такая трасса μ есть, должна после этой трассы μ принимать стимул x. Таким образом, все продолжения трассы μ блокировкой стимула x, которые есть в спецификации, должны быть удалены.

Второе правило – это правило сохранения поведения после стационарности. Пусть в спецификации трасса μ непосредственно продолжается как приёмом стимула x, так и стационарностью, но после стационарности не продолжается стимулом x. Очевидно, любое продолжение после трассы $\mu\cdot\delta$ должно быть и после трассы μ: переход по δ – это петля. Назовём это *законом петли*.

После пополнения трасса $\mu\cdot\delta$ не может продолжаться блокировкой $\{x\}$, так как в противном случае, по закону петли, трасса μ также продолжалась бы блокировкой $\{x\}$, что противоречит правилу приоритета. Значит, после пополнения трасса $\mu\cdot\delta$ продолжается стимулом x, то есть после трассы $\mu\cdot\delta$ добавляются продолжения вида $x\cdot\lambda$.

Любое добавленное продолжение $x\cdot\lambda$ после трассы $\mu\cdot\delta$, по закону петли, должно быть также после трассы μ. Но тогда это продолжение $x\cdot\lambda$ было после трассы μ в исходной спецификации, так как иначе пополнение ослабит соответствие: разрешит продолжение $x\cdot\lambda$ после трассы μ, которое раньше не разрешалось.

Наконец, если в исходной спецификации после трассы μ было продолжение $x\cdot\lambda$, то после пополнения оно должно быть после $\mu\cdot\delta$. В противном случае произошло бы усиление соответствия. Ведь до пополнения после трассы $\mu\cdot\delta$ в конформной реализации могло быть продолжение $x\cdot\lambda$, поскольку вообще не проверялось, может ли реализация после трассы $\mu\cdot\delta$ принимать стимул x, а, если может, то какое дальнейшее поведение допустимо, а какое нет.

Итак, правило сохранения поведения после стационарности звучит так: если трасса μ продолжается как стимулом x, так и стационарностью δ, а трасса $\mu\cdot\delta$ не продолжается стимулом x, то после пополнения трасса $\mu\cdot\delta$ не продолжается блокировкой $\{x\}$ и продолжается теми и только теми трассами вида $x\cdot\lambda$, которыми до пополнения продолжалась трасса μ.

Теперь мы можем пополнить спецификацию в соответствии с этими двумя правилами.

Это пополнение решает проблему самоприменимости спецификации. Теперь, если стимул принимается в одном состоянии после трассы, то он принимается в каждом состоянии после трассы, и, если стимул блокируется в одном состоянии после трассы, то он блокируется в каждом состоянии после трассы. Все состояния, в которых заканчивается трасса, имеют одно и то же множество блокируемых стимулов и одно и то же множество принимаемых стимулов.

Классическая семантика отношения *ioco*. Проблема несохранения соответствия.

Для решения проблемы сохранения соответствия мы должны продолжить пополнение спецификации. Дело в том, что если в исходной спецификации трасса во всех своих конечных состояниях не продолжалась стимулом, а только его блокировкой, то это так и останется. Поскольку классическая семантика **`ioco`** не проверяет блокировку, мы должны эту блокировку заменить приёмом стимула с соответствующим продолжением. Таким продолжением может быть либо произвольное продолжение без блокировок и разрушения, то есть демоническое пополнение, либо продолжение разрушением, то есть пополнение разрушения.

Для всюду определённых по стимулам реализаций эти два пополнения эквивалентны с точки зрения соответствия. Реализация конформна демонически пополненной спецификации тогда и только тогда, когда она конформна спецификации, пополненной разрушением. При тестировании это проявляется в том, что в первом случае давать стимул бессмысленно, а во втором случае стимул давать нельзя.

Однако пополнение разрушения имеет ряд преимуществ по сравнению с демоническим пополнением.

1) Пополнение разрушения не усиливает соответствие для не всюду определённых реализаций, поскольку стимул, который нельзя давать в реализацию, реализация может блокировать. Демоническое пополнение запрещает такую блокировку.

   Конечно, здесь мы уже используем модифицированное соответствие: не ***ioco***, а $\boldsymbol{ioco}_{\beta\gamma\delta}$. Поскольку блокировок у нас нет, это соответствие можно назвать $\boldsymbol{ioco}_{\gamma\delta}$.

2) Демоническое пополнение решает проблему несохранения соответствия при переходе к асинхронному тестированию, но при этом теряется информация о том, что стимул бессмысленно давать в реализацию. При пополнении разрушения стимул остаётся разрушающим, мы не будем его давать в реализацию и при асинхронном тестировании тоже.

   Для решения проблемы несохранения соответствия нужно, чтобы синхронное и асинхронное тестирование были одинаковыми: одинаково строгими, то есть проверяющими все стимулы, или одинаково слабыми, то есть не проверяющими одни и те же неспецифицированные стимулы. Демоническое пополнение предлагает в обоих случаях строгое тестирование, а пополнение разрушения – слабое тестирование. Что, конечно, лучше.

3) Семантика разрушения вообще шире семантики произвольного поведения, поскольку допускает в реализации не только приём стимула с любой последующей последовательностью приёма стимулов, выдачи реакций и стационарности, но также блокировку стимула и разрушение реализации после приёма стимула. Более того, после приёма разрушающего стимула допускается даже дивергенция.

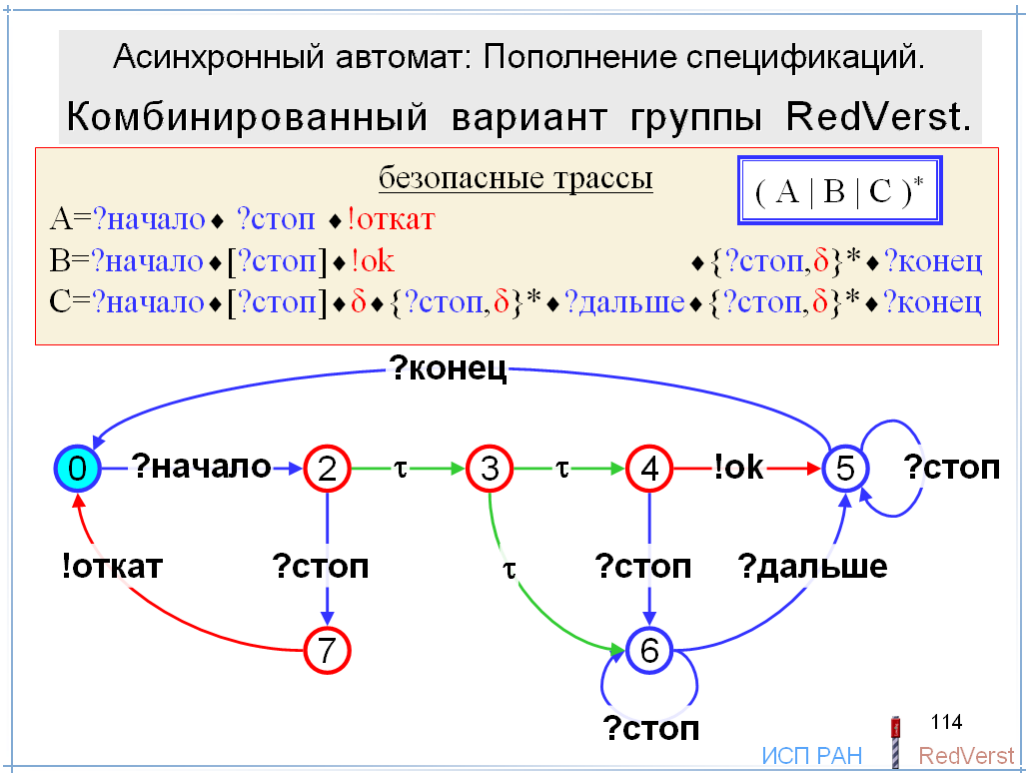
Комбинированный вариант группы RedVerst.

В группе RedVerst предлагается комбинированный вариант пополнения:

в нестационарных состояниях, в которых реализация может иметь внутреннюю активность или готова выдать реакции, неспецифицированный стимул не принимается, а в стационарных состояниях – разрушает реализацию.

При асинхронном тестировании с входной очередью стимулов и выходной очередью реакций блокировка стимула в нестационарном состоянии оказывается временной: приём стимула из входной очереди откладывается до перехода системы в стационарное состояние.

Сейчас мы рассмотрим пример системы и её асинхронного тестирования, имея в виду, что среда передачи – это входная и выходная очереди. Система выполняет в цикле различные виды работ. Каждая работа выполняется в два этапа. Первый этап инициируется в начальном состоянии 1 командой, то есть стимулом, *начало*. Эта команда может иметь различные параметры, определяющие вид работы. Это означает, что, фактически, может быть несколько стимулов, которые мы называем командой *начало*. Для нашего примера мы эти реализационные стимулы не различаем. Если всё хорошо, система проходит ряд нестационарных состояний 2,3,4, они отмечены красным цветом, выдаёт реакцию *ok* и попадает в стационарное состояние 5, после чего можно инициировать второй этап работы командой *конец*.



В процессе выполнения первого этапа у системы могут возникнуть проблемы, требующие дальнейших указаний. В этом случае она не выдаёт реакцию *ok* и переходит в стационарное состояние 6. Мы намеренно не сделали в системе выдачу реакции, чтобы показать необходимость при тестировании проверять не только наличие реакции, но и отсутствие реакций, то есть стационарность.

После этой стационарности можно командой *дальше* продолжить выполнение работы. Эта команда также может иметь различные параметры, которыми мы управляем дальнейшим выполнением работы. Это означает, что, фактически, может быть несколько стимулов, которые мы называем командой *дальше*. После этой команды уже можно не ждать реакцию *ok* и сразу давать команду *конец*.

Кроме этого, предоставляется возможность в любой момент времени после того, как первый этап инициирован, но до перехода ко второму этапу, приостановить выполнение работы командой *стоп*. Этого нельзя делать до начала работы или после команды *конец*, то есть в состоянии 1 команда *стоп* разрушающая.

Из трёх нестационарных состояний 2, 3 и 4 команда *стоп* может приниматься только в состояниях 2 и 4. Отметим, что в рассматриваемой модели асинхронного автомата система не обязана принимать стимул в нестабильном состоянии 2, поскольку ей разрешается асинхронно выполнять внутренний переход. При асинхронном тестировании система также не обязана принимать стимул в стабильном, но нестационарном, состоянии 4, поскольку выходная очередь всегда готова принимать реакции.

Если команда *стоп* принимается в состоянии 2, система может отменить выполнение работы, очищая все её следы, и становится готовой к выполнению следующей работы. Об этом система извещает реакцией *откат*. Теперь у нас появляется нестационарное состояние 7, в котором, однако, команду *стоп* давать нельзя.

Если команда *стоп* приходит в систему позже, в состоянии 4, то система может перейти в стационарное состояние 6, ожидая дальнейших указаний, то есть команду *дальше*.

Команда *стоп* может поступить и тогда, когда система находится в состоянии 3. Однако приём команды откладывается до перехода в стационарное состояние, в котором команда всегда принимается, или нестационарное состояние, в котором она может приниматься. В данном случае это нестационарное состояние 4, которое мы уже рассматривали.

Если команда *стоп* откладывается до стационарного состояния 6, то в нём она принимается, но состояние не меняется. После этого можно продолжить командой *дальше*.

Если команда *стоп* откладывается до стационарного состояния 5, с предварительной выдачей реакции *ok*, то в нём она также принимается без изменения состояния. После этого можно продолжить командой *дальше* или перейти ко второму этапу командой *конец*.

Теперь посмотрим, какие трассы мы можем получить, учитывая безопасность тестирования.

Множество безопасных трасс описывается следующим регулярным выражением: (A|B|C)*.

С самого начала мы можем давать только команду *начало*, поскольку начальное состояние стационарное и, следовательно, все остальные команды, которые не определены в этом состоянии, могут привести к разрушению реализации.

Вариант A: Сразу после команды *начало*, мы можем безопасно давать только команду *стоп*, поскольку она определена во всех стационарных состояниях, достижимых по τ-переходом и переходам по реакциям после команды *начало*. Любая другая команда этим свойством не обладает.

После команды *стоп* мы можем получить реакцию *откат*, и на этом цикл завершается.

Вариант B: Реакцию *ok* мы можем получить независимо от того, давали мы команду *стоп* или нет. После реакции ok мы можем сколько угодно раз давать команду *стоп* и,

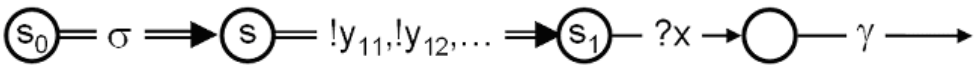
принимая реакции, обнаруживать стационарность. Затем, давая команду конец, завершаем цикл.

Вариант C: Отсутствие реакций, то есть стационарность, мы также можем получить после начала работы независимо от того, давали мы команду стоп или нет. После этого мы также можем сколько угодно раз давать команду стоп и, принимая реакции, обнаруживать стационарность. Но, в отличие от реакции ok, после этого мы можем давать не команду конец, которая запрещена в стационарном состоянии 6, а команду дальше. После этого мы опять можем сколько угодно раз давать команду стоп и, принимая реакции, обнаруживать стационарность, а затем, давая команду конец, завершаем цикл.

Этот пример демонстрирует следующие важные случаи:
1) Некоторые стимулы можно безопасно давать в стационарном состоянии, а некоторые нет. В нашем примере в начальном состоянии можно давать только команду начало.
2) Некоторые стимулы, которые нельзя давать вначале, можно давать после других стимулов. Команду стоп можно давать после команды начало.
3) Некоторые стимулы можно давать только после некоторых реакций или их отсутствия. Команду дальше вслед за командой начало и, быть может, последующей командой стоп можно давать после стационарности. После команды дальше можно давать команду конец, но её можно давать и после реакции ok.

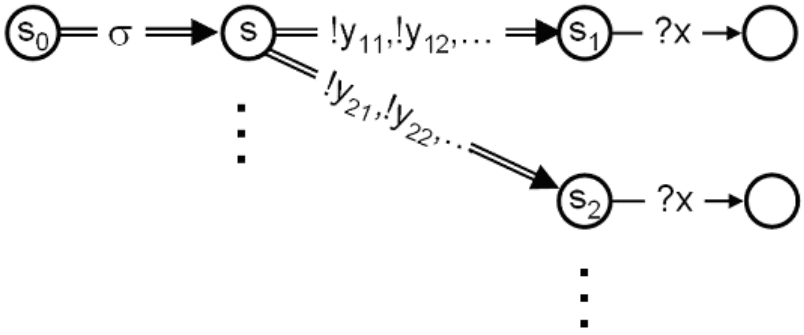
В целом безопасность стимула x определяется предшествующей трассой σ.

Пусть такая трасса σ, начинающаяся с начального состояния асинхронного автомата, может быть продолжена такой последовательностью реакций (в частности, пустой), после которой мы можем оказаться в стационарном состоянии, где стимул x не определён.



допущение разрушения

Тогда, в соответствии с нашим допущением полноты, в таком стационарном состоянии стимул x понимается как разрушающий. Такой стимул x опасен, и его нельзя давать при тестировании после трассы σ.



В противном случае, если при любом, возможном в автомате, продолжении трассы σ последовательностью реакций мы можем оказаться только в тех стационарных состояниях, в которых стимул x определён, то он безопасен и его можно при тестировании давать после трассы σ.

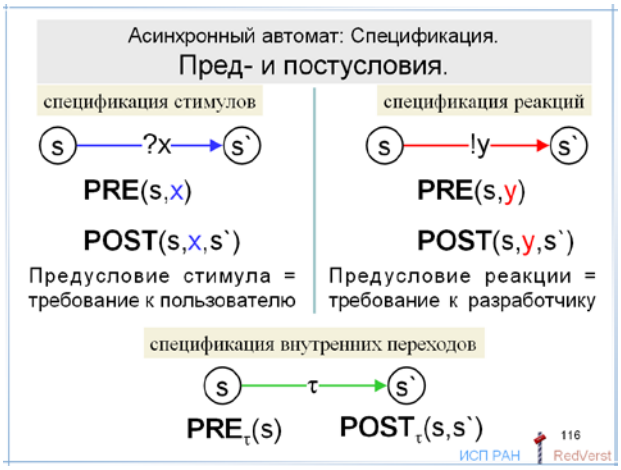
Пред- и постусловия.

Мы рассматривали спецификацию автоматов Мили в пред- и постусловиях. Теперь посмотрим такую спецификацию для асинхронных автоматов общего вида.

В отличие от автоматов Мили реакция асинхронного автомата, вообще говоря, не связана жёстко со стимулом: после стимула может не быть реакций, или быть несколько реакций. В общем случае мы уже не можем рассматривать реакцию как ответную на тот или иной стимул. В целом допустимость стимула или реакции определяется состоянием автомата. Поэтому стимулы и реакции специфицируются раздельно.

Предусловие стимула – это предикат от пресостояния и стимула: `PRE(s,x)`.
Постусловие стимула – это предикат от пресостояния, стимула и постсостояния: `POST(s,x,s`)`.

Соответственно, предусловие реакции – это предикат от пресостояния и реакции: `PRE(s,y)`.
Постусловие реакции – это предикат от пресостояния, реакции и постсостояния: `POST(s,y,s`)`.



Мы видим, что пред- и постусловия стимула и реакции определяются полностью аналогично. Однако, их интерпретация различна.

Посмотрим, что означает предусловие стимула и реакции для разработчика реализации и для пользователя, создающего программу, которая будет обращаться к реализации. Поскольку при тестировании тест подменяет собою окружение (или его часть), точка зрения тестировщика, фактически, совпадает с точкой зрения пользователя.

Для разработчика предусловие стимула означает: только для тех стимулов, которые удовлетворяют предусловию стимулов, нужно реализовать поведение, требуемое постусловием стимула. Если стимул не удовлетворяет предусловию, можно считать, что такой стимул в реализацию не поступает; во всяком случае, разработчик может не беспокоиться о том, что случиться, если такой стимул всё-таки поступит. Для пользователя предусловие стимула определяет, когда он может обращаться к реализации, а когда нет. Иными словами, предусловие стимула – это требование к пользователю.

Предусловие реакции, наоборот, для реализатора определяет, когда он может выдавать такую реакцию, а когда нет. Пользователь может рассчитывать, что он получит только такую реакцию, которая удовлетворяет предусловию реакции. Во всяком случае, он не должен беспокоиться, что произойдёт, если такая реакция всё-таки поступит от реализации. Иными словами, предусловие реакции – это требование к реализатору.

Кроме переходов по стимулам и реакциям, в асинхронном автомате могут быть τ-переходы. Для них предусловие – это предикат от пресостояния `PRE(s)`, а постусловие – предикат от пресостояния и постсостояния `PRE(s,s`)`.

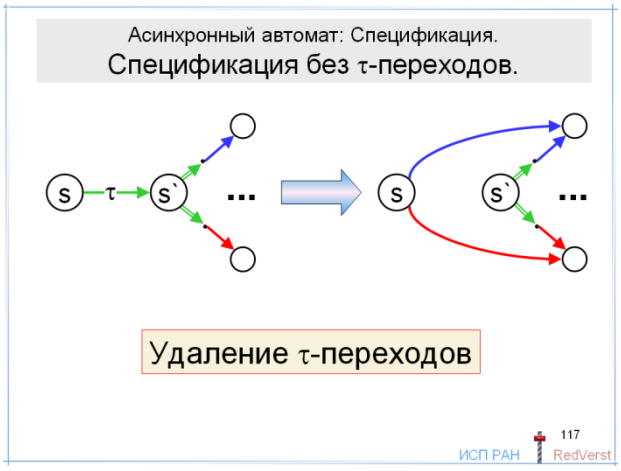
Спецификация без τ-переходов.

Может показаться странным, что в спецификациях UniTesK нет τ-переходов.

На самом деле, во многих случаях спецификация с τ-переходами может быть заменена спецификацией без τ-переходов, эквивалентной ей, по крайней мере, в смысле отношения ***ioco***$_{\beta\gamma\delta}$.

Действительно, τ-переход из s в s` не наблюдаем при тестировании. После стимула или реакции, наблюдаемых при переходе в состояние s, может следовать любой стимул или реакция, наблюдаемые при переходе из состояния s`, или любого другого состояния, достижимого из состояния s` по τ-переходам.

Поэтому вместо τ-перехода из состояния s в состояние s` мы можем все такие наблюдаемые переходы сдублировать из состояния `s`.



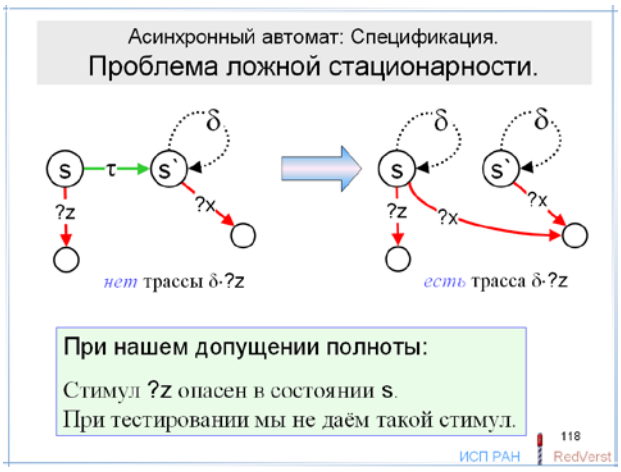
Проблема ложной стационарности.

Проблема может возникнуть только со стационарностью.

Если из состояния s и состояний, достижимых из s по τ-переходам, вели только переходы по стимулам, то после удаления τ-переходов состояние s становится стационарным.

Пусть при этом некоторый стимул ?z был определён в s, но не был определён ни в одном из стационарных состояний, достижимых из s.

Тогда до преобразования в состоянии s не было трассы δ?z, а после преобразования такая трасса появляется.



Таким образом, удаление из спецификации τ-переходов изменяет соответствие ***ioco***. Точнее, строит автомат, который не эквивалентен по отношению ***ioco*** исходной спецификации.

Другое дело, что при нашем допущении полноты этой проблемы не возникает. Напомню, что мы интерпретируем неспецифицированный стимул в зависимости от состояния: в стационарном состоянии он разрушающий, а в нестационарном – блокируемый. В этом случае при безопасном тестировании мы никогда не будем давать такой стимул z, который не принимается в некотором стационарном состоянии, достижимом из текущего состояния по τ-переходам. Поэтому, при удалении τ-переходов перед безопасным стимулом не может возникнуть стационарность, если её не было до преобразования.

Здесь ошибка!

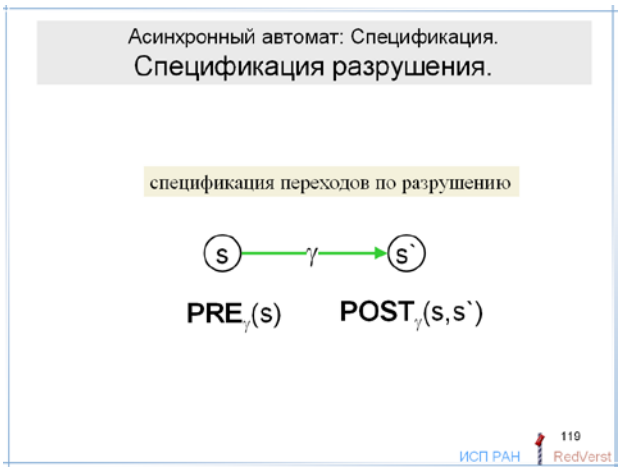
Если s—?z→—!a→ и s`—?z→—!b→, то после удаления внутренних переходов появится новая трасса δ·?z·!a. Это имеет место и для нашего допущения полноты.

Спецификация разрушения.

При нашем допущении полноты разрушающий стимул – это стимул, нарушающий предусловие в стационарном состоянии. Тем самым разрушение специфицировано неявно.

Однако в некоторых случаях полезно уметь явно специфицировать разрушение.

Спецификация перехода по разрушению аналогична спецификации τ-перехода. Предусловие – это предикат от пресостояния, а постусловие – это предикат от пресостояния и постсостояния.



Блокировка в стационарных состояниях.

Проблема ложной стационарности остаётся, если использовать иные допущения полноты. Например, наше допущение полноты можно обобщить. А именно: разрешить некоторые стимулы, неспецифицированные в стационарном состоянии, понимать не как разрушающие, а как блокируемые. Тогда может оказаться, что стимул принимается без последующего разрушения в нестационарном состоянии, но блокируется во всех стационарных состояниях, достижимых из данного состояния по τ-переходам. При удалении τ-переходов возникнет ложная стационарность перед этим стимулом.

Это обобщение нашего допущения полноты практически полезно. Оно не обязательно вызывает глухой deadlock при асинхронном тестировании. Например, если среда передачи содержит не одну, а две параллельные входные очереди стимулов, автомат может в стационарном состоянии блокировать все или некоторые стимулы из одной очереди и принимать все стимулы из другой очереди. Такое асинхронное тестирование мы подробнее рассмотрим позже (в разделе о гипертестировании). А сейчас посмотрим, как может выглядеть спецификация при таком обобщённом допущении полноты.

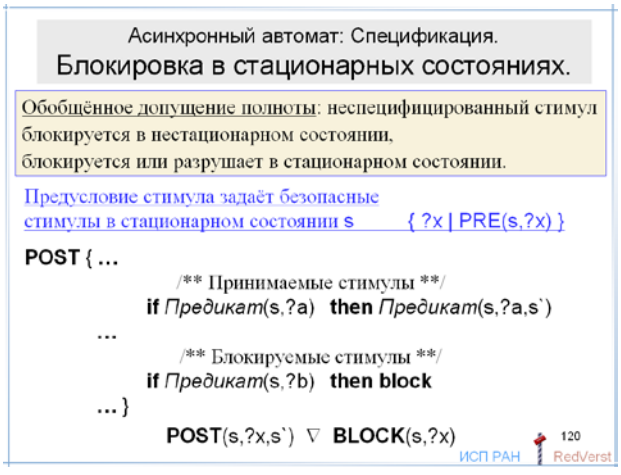
Проблема специфицирования заключается в том, что в данном состоянии (при нашем обобщённом допущении полноты – в данном стационарном состоянии) мы должны некоторые стимулы интерпретировать как блокируемые, а некоторые – как разрушающие. Как это описать?

Существуют два варианта такого описания. Они отличаются тем, как понимаются стимулы, удовлетворяющие предусловию для стационарного состояния: как принимаемые или как безопасные.

В первом варианте (предусловие описывает стимулы принимаемые в стационарном состоянии) для указания разрушающего стимула мы могли бы в постусловии использовать ключевое слово **swap**. Однако более общим является явная спецификация не разрушающего стимула, а перехода по разрушению, которую мы рассмотрели на предыдущем слайде.

Во втором варианте (предусловие описывает стимулы безопасные в стационарном состоянии) предлагается предусловие стимула понимать как предикат, выделяющий все безопасные в данном состоянии стимулы: как принимаемые в этом состоянии, так и блокируемые в нём.

Тогда в постусловии для блокируемого стимула должно быть явно указано, что он блокируется или принимается.



Для нестационарного состояния стимул, нарушающий предусловие, – всегда блокируемый.

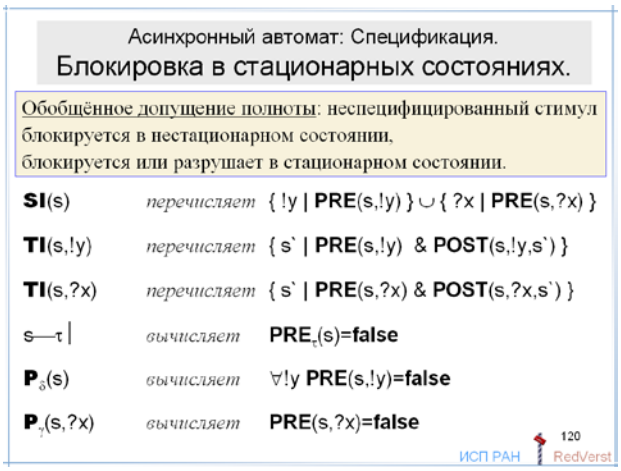
Если стимул принимается, то после ключевого слова ***then*** (или аналогичной конструкции в языке спецификации) описывается предикат от пресостояния, постсостояния и стимула. Этот предикат описывает приём стимула, фактически, неявно указывая возможные постсостояния, поскольку к этому моменту возможные стимул и пресостояние известны (они выделены предусловием и условие branch`a).

Для указания не приёма, а блокировки стимула предлагается использовать вместо такого предиката специальное ключевое слово ***block***.

В целом выполнение требований постусловия записывается как исключающая дизъюнкция предиката от пресостояния, стимула и постсостояния для принимаемых безопасных стимулов и предиката от пресостояния и стимула для блокируемых безопасных стимулов.

Предусловие реакции задаёт множество реакций, которые должны перечисляться для данного состояния итератором реакций и безопасных стимулов. Предусловие стимула теперь задаёт множество безопасных стимулов, которые должны перечисляться этим итератором, причём какие-то из этих стимулов принимаются в данном состоянии, а какие-то блокируются. Именно это множество стимулов должно быть перечислимым.

Таким образом, итератор символов SI(s) перечисляет реакции и стимулы, удовлетворяющие предусловиям реакций и стимулов, соответственно, для данного состояния s.



Итератор переходов TI(s,z) перечисляет постсостояния, которые возможны для пресостояния s и реакции !y, если они удовлетворяют предусловию реакций, или для пресостояния s и стимула ?x, если они удовлетворяют предусловию стимулов, а в постусловии для этой пары не написано ***block***.

Стабильность состояния s проверяется по спецификации τ-переходов: состояние s не удовлетворяет предусловию этой спецификации.

Алгоритм $P_\delta(s)$ проверяет, что ни одна реакция не удовлетворяет предусловию реакций для данного стабильного состояния s.

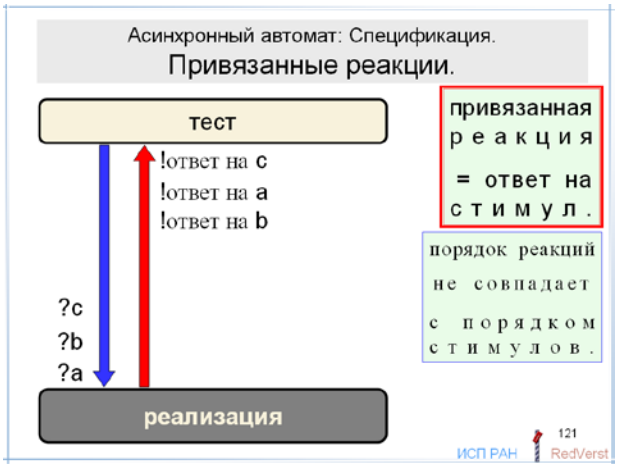
Дополнение множества безопасных стимулов – это множество стимулов, разрушающих в данном состоянии. Поскольку нам требуется разрешимость множества безопасных стимулов, множество разрушающих стимулов также должно быть перечислимо.

Алгоритм $P_\gamma(s,x)$ проверяет, что стимул x разрушающий для состояния s, то есть не удовлетворяет предусловию стимула для состояния s.

Привязанные реакции.

Частный, но широко распространённый, случай реакций в асинхронном автомате – это, так называемые, *привязанные* реакции. Такая реакция всегда является ответом на некоторый стимул, но она может выдаваться реализацией не обязательно сразу после приёма стимула.

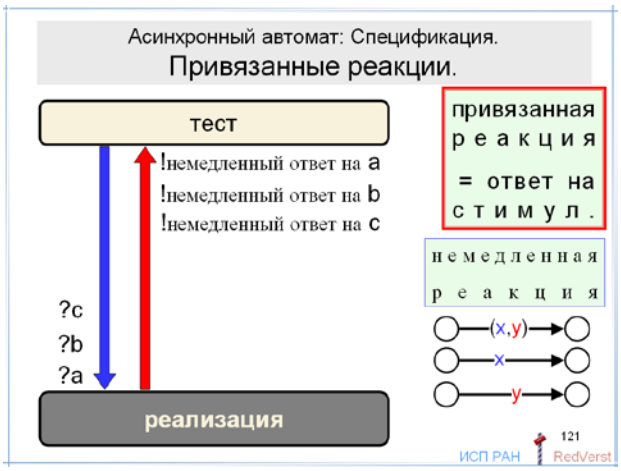
Реализация может принять несколько стимулов, а потом уже выдавать реакции на них в том же порядке или в порядке, отличном от порядка стимулов.



Концептуально привязанные реакции – это способ сокращённой записи. Привязку реакции к стимулу всегда можно отобразить через состояния автомата.

Привязанную реакцию естественно специфицировать вместе со спецификацией стимула. Такое локальное описание имеет ряд преимуществ. Во-первых, реализуется понятие привязанности: мы понимаем, что после стимула мы должны получить одну из привязанных реакций, хотя, в отличие от автомата Мили, не обязательно немедленно. Во-вторых, имена привязанных реакций могут быть локальными, они также привязываются к имени стимула. При взаимодействии реализации с окружением это реализуется параметром реакции, указывающим тот стимул, на который выдаётся эта реакция.

Можно также считать, что стимул, кроме привязанных реакций, может иметь немедленную реакцию, которая должна поступить сразу после стимула. Это своеобразная переходная форма от автомата Мили к общему случаю асинхронного автомата.



Кроме того, можно разрешить выдачу нескольких привязанных реакций в ответ на один стимул. Правда, если мы хотим уметь определять окончание передачи реакций в ответ на стимул, требуется указание, какая из реакций является последней. Это похоже на вход отстройки в кластерной технологии программирования.