И. Бурдонов, А. Косачев
Тестирование и верификация систем на основе формальных моделей. Лекция 3.
http://old.compsciclub.ru/courses/testingverification/
Уральский федельный университет, Computer Science клуб, Екатеринбург, 6,7,8 апреля 2012
54 слайда

Тестирование и верификация систем на основе формальных моделей.

6. Проблемы практического тестирования

Слайд 1. Сегодня у нас будет продолжение с переходом от теории к практике.

Слайд 2. Вот план доклада.

Сначала мы закончим оптимизацию тестов.

Слайд 3. 5. Оптимизация тестов.

Отношение следования ошибок (одиночное)

Теперь рассмотрим ещё один вид оптимизации тестов, связанный с зависимостью между ошибками.

Тестовая ошибка = трасса максимальной истории теста с вердиктом ***fail***. Будем говорить, что тест обнаруживает эту тестовую ошибку.

1▶Определим отношение *следования* ошибок: из ошибки *a* *следует* ошибка *b*, если в любой безопасно-тестируемой реализации, где есть трасса *a*, есть и трасса *b*. Это отношение, очевидно, является предпорядком, то есть рефлексивно и транзитивно.

2▶ Вот два примера следования ошибок.

Вычисляемые отказы: если трасса ошибки содержит такой отказ, то трасса без отказа тоже ошибка. На самом деле эти ошибки эквивалентны по следованию.

Вторичные ошибки: если есть вторичная ошибка, то по определению у неё есть неконформный строгий префикс, то есть ошибка 2-го рода. Очевидно, из этой вторичной ошибки следует эта ошибка 2-го рода.

3►Набор тестов полный, если он обнаруживает хотя бы одну тестовую ошибку из каждого минимального (по предпорядку) класса эквивалентности ошибок.

Оптимизация заключается в том, что из набора примитивных тестов удаляется такие тесты, что все оставшиеся тесты обнаруживают тестовые ошибки, эквивалентные всем тестовым ошибкам из минимальных классов эквивалентности, которые обнаруживают удаляемые тесты. Полнота тестирования при таком удалении сохраняется.

Слайд 4. 5. Оптимизация тестов.

Отношение следования ошибок (множественное)

В общем случае нужно рассматривать множественное следование ошибок, которое не сводится к одиночному следованию ошибок.

Будем говорить, что из множества ошибок A следует множество ошибок B, если в любой безопасно-тестируемой реализации, где есть какая-то ошибка из A, есть некоторая ошибка из B.

Множественное следование – это предпорядок на семействе множеств ошибок.

1►Набор тестов значимый, если из множества обнаруживаемых им тестовых ошибок следует множество всех ошибок. Иными словами, тесты из набора тестов не обнаруживают ложных ошибок: если какой-то тест выносит вердикт ***fail***, то реализация действительно неконформна.

Что можно понимать под множеством всех ошибок?

Когда мы определяли примитивные тесты, мы говорили только про ошибки 1-го рода. Потом мы ввели ошибки второго рода – неконформные безопасные трассы спецификации. Наконец, мы ввели первичные и вторичные ошибки: вторичная ошибка – это ошибка 1-го или 2-го рода, у которой есть строгий неконформный префикс, то

есть ошибка 2-го рода. Естественно, у вторичный ошибки всегда есть префикс, который является первичной ошибкой 2-го рода.

Поэтому есть, по крайней мере, 3 варианта, что понимать под множеством всех ошибок.

1-ый вариант: множество ошибок 1-го рода.

2-ой вариант: множество ошибок 1-го и 2-го родов.

3-ий вариант: множество первичных ошибок 1-го и 2-го родов.

В любом из этих трёх вариантов набор тестов, который обнаруживает хотя бы одну ошибку 1-го или 2-го рода, вторичную или первичную, будет значимым.

3. Также под множеством всех ошибок можно понимать множество только первичных ошибок. Если набор тестов их все обнаруживает

Набор тестов исчерпывающий, если множество обнаруживаемых им тестовых ошибок следует из множества всех ошибок. Иными словами, если в реализации есть ошибка, то какой-нибудь тест из набора вынесет вердикт ***fail***.

Набор тестов полный: если множество обнаруживаемых им тестовых ошибок эквивалентно множеству всех ошибок.

2▶Оптимизация заключается в поиске множества ошибок, которое эквивалентно множеству всех ошибок, и которое минимально по вложенности среди всех таких множеств ошибок.

С практической точки зрения нас интересует вопрос: есть ли у, быть может, бесконечного множества всех ошибок эквивалентные ему конечные подмножества?

И, если есть, как их найти?

В настоящее время этот раздел теории находится в стадии разработки. Уточняются необходимые и достаточные условия следования ошибок, то есть такие условия, которые можно было бы легко проверять по спецификации в процессе генерации тестов.

Что здесь самое важно для практического тестирования?

То, что полный набор тестов, построенный с учётом следования ошибок, может быть конечным, а без такого учёта – бесконечным.

Слайд 5. 5. Оптимизация тестов.

Пример множественного следования ошибок

Вот рассмотрим простой пример множественного следования ошибок.

Здесь семантика содержит две **R**-кнопки: пустая кнопка и кнопка ab. И две **Q**-кнопки, разрешающие каждая по одному действию: x и a.

Кнопка a объявлена безопасной по ***safe by*** после всех трасс, начинающихся с отказа ab и далее продолжающаяся, быть может, последовательностью из этого отказа и пустого отказа, затем хотя бы одним действием x. После всех остальных трасс она опасна.

Кнопка x объявлена безопасной по ***safe by*** после всех трасс, кроме трасс, заканчивающихся в состоянии 5. Иными словами, эта кнопка безопасна по ***safe in***.

Заметим, что такое отношение ***safe by*** не является ограниченным на этой LTS-спецификации, но оно регулярно.

Какие здесь есть ошибки?

Я буду говорить только про трассы, в которых нет вычисляемых отказов.

После отказа ab нет действий a и b. Это состояние 1. Но эти ошибки неактуальные, поскольку в реализации все трассы согласованы.

После любого отказа и действия x нет отказа ab. Это состояние 2.

Однако, если перед x стоит отказ ab, то такая ошибка неактуальна. Это объясняется тем, что тогда после такой трассы есть и отказ a. А это противоречит безопасности его в спецификации после соответствующей трассы.

Актуальные ошибки такие:

1. Отказ ab после пустого отказа и ненулевого количества действий x: состояние 2. Таких ошибок бесконечное число, поскольку действий x может быть сколько угодно.

2. Действие a или действие b после пустого отказа: состояние 1.

3. Действие a или действие b после пустой трассы: состояния 0 и 1.

Можно заметить, что в этой спецификации есть демонические трассы. Это трассы ненулевой длины, состоящие только из действия x. Каждая такая трасса заканчивается в трёх состояниях: 2, 3 и 4. И в совокупности в этих состояниях есть все наблюдения по кнопке ab: действия a и b в состояниях 2 и 3, и отказ ab в состоянии 4.

После действия a или b мы оказываемся в одном состоянии 5, где имеется дивергенция и поэтому все кнопки опасны. Поэтому продолжение действием a или b оставляет трассу демонической.

А после отказа ab мы оказываемся в одном состоянии 4, где имеется только отказ ab. Но после отказа ab, понятное дело, действия a и b в реализации не могут быть. Поэтому здесь имеется только одно актуальное наблюдение – отказ ab. Поэтому продолжение отказом ab оставляет трассу демонической.

Из ошибки 1-го вида следует хотя бы одна из ошибок 2-го вида. Это множественное следование. Почему?

1►Доказательство: Пусть в безопасно-тестируемой реализации есть ошибка 1-го вида, то есть трасса: пустой отказ, одно или несколько действий x и далее отказ ab.

Допустим, в реализации нет ошибки 2-го вида, то есть после пустого отказа нет действия a и нет действия b.

Но тогда после пустого отказа нет никаких действий из отказа ab.

Следовательно, по полноте трассовой модели реализации отказ ab можно вставить после пустого отказа перед каждым имеющимся в реализации продолжением.

Следовательно, в реализации должна быть трасса: пустой отказ, вставленный отказ ab, далее одно или несколько действий x и далее отказ ab.

Но тогда из согласованности трассовой модели реализации следует, что после такой трассы нет действия a. А тогда по конвергентности эта трасса должна продолжаться отказом a.

Далее мы можем удалить из этой трассы пустой отказ и второй отказ ab, и получим трассу, которая тоже должна быть в реализации

по замкнутости трассовой модели. Это трасса: отказ ab, далее одно или несколько действий x и далее отказ a.

Но отказ a – это **Q**-отказ, поэтому по ***safe in*** он опасен в реализации после трассы, состоящей из отказа ab и нескольких действий x.

Однако в спецификации отказ a безопасен после такой трассы. Тем самым, реализация оказалась не удовлетворяющей гипотезе о безопасности. Мы пришли к противоречию.

2►Следовательно, если в реализации есть ошибка 1-го вида, то в ней должна быть хотя бы одна из двух ошибок 2-го вида.

3►А из ошибки 2-го вида, очевидно, следует соответствующая ошибка 3-го вида. Это непосредственно следует из замкнутости реализации по удалению отказов.

Получается, что из бесконечного множества всех актуальных ошибок следует множество, состоящее всего из двух ошибок 3-го типа: действие a или действие b после пустой трассы.

4►Полный набор тестов состоит из одного теста: в самом начале нужно нажать кнопку ab и вынести вердикт ***fail***, если наблюдается действие a или действие b.

5►Спецификация конформна сама себе. И на слайде приведены две неконформные, но безопасно-тестируемые, реализации, в которых есть ошибки 1-го вида и, конечно, хотя бы одна ошибка 3-го вида.

Слайд 6. 5. Оптимизация тестов.

Гипотеза о трёх оптимизациях

В настоящее время у нас есть следующая ГИПОТЕЗА: Если существует конечный полный набор тестов для класса всех безопасно-тестируемых реализаций, то его можно построить с помощью трех оптимизаций, основанных на

1) удалении неконформных трасс (пополнение),

2) учете демонических трасс,

3) множественном следовании ошибок: выделении конечного минимального по вложенности множества ошибок, эквивалентного множеству всех ошибок.

1►ДОПОЛНИТЕЛЬНАЯ ПРАКТИЧЕСКАЯ ГИПОТЕЗА: Для конечной спецификации в конечной семантике при регулярном отношении ***safe by*** это можно сделать алгоритмически за конечное время.

Пока что это только гипотезы. У нас есть некоторые идеи, как найти конструктивные условия множественного следования. Но формально это пока ещё не сделано. На этой основе можно создать алгоритм, позволяющий учитывать множественное следование и выделять минимальное по вложенности множество ошибок, эквивалентное множеству всех ошибок. Для конечной спецификации в конечной семантике при регулярном отношении ***safe by*** этот алгоритм должен работать конечное время и сообщать, получилось искомое множество ошибок конечное или нет.

Слайд 7. 6. Проблемы практического тестирования.

Метрики полноты тестирования = покрытия.

Мы занимаемся полным тестированием.

На практике бывают разные виды тестирования. Как правило, сформулировав разные допущения – иногда явно, а чаще неявно – для тестирования вводят разного рода метрики, называемые покрытиями. А затем оценивают набор тестов по этим метрикам. Часть таких метрик строится на основании того, что код тестируемой реализации нам известен и доступен «по ходу выполнения». Например, метрики покрытия кода, покрытия путей и т.п.

Нас же интересует проблема полного тестирования, т.е. такого тестирования, пройти которое может только конформная реализация.

Слайд 8. 6. Проблемы практического тестирования.

Проблема конечности

Естественным требованием практического тестирования является его конечность по времени: через конечное время мы должны вынести окончательный вердикт – реализация конформна или в ней есть ошибки. Для этого конечными по времени должны быть как генерация тестов, так и тестирование по этим тестам.

1▶Отсюда вытекает конечность полного набора тестов (или единого теста с рестартом). Для этого «почти» необходимы, хотя и недостаточны, конечность алфавита внешних действий ***L*** и конечность LTS-спецификации (числа ее переходов), что на практике вполне приемлемо. Условия конечности полного набора тестов для произвольной реализации в алфавите **L** были рассмотрены ранее.

2▶Конечность тестирования опирается на конечность полного набора тестов, конечность времени прогона каждого теста и конечность требуемого числа прогонов каждого теста.

Слайд 9. 6. Проблемы практического тестирования.

Проблема бесконечности полного набора тестов

Поскольку примитивные тесты генерируются по трассам спецификации, бесконечность полного набора тестов сводится к бесконечности числа таких трасс.

Если два источника бесконечности числа трасс.

1▶Бесконечное ветвление. Если трасса продолжается бесконечным числом действий или отказов, то мы имеем бесконечное число трасс, полученных добавлением в конец трассы этих её продолжений.

Но бесконечное ветвление возможно только при бесконечном алфавите внешних действий. Требование конечности алфавита во многих случаях на практике вполне примлемо.

2▶Бесконечные трассы. Если у нас есть бесконечная трасса, то у нас есть и бесконечное число её конечных префиксов. А тесты генерируются как раз по конечным трассам.

Нужно отметить, что бесконечная трасса может быть и в конечной спецификации: достаточно иметь цикл по переходам или даже один переход-петлю.

3►Конечно, для генерации тестов нужны не все трассы. Мы это уже видели на прошлых лекциях. Но может быть и бесконечное число трасс, существенных для генерации тестов.

Слайд 10.6. Проблемы практического тестирования.

Проблема конечности

Конечность времени прогона теста гарантируется для конечного теста «практическими предположениями» о семантике:

1) Любая конечная последовательность любых действий (как внешних, так и внутренних) совершается за конечное время, а бесконечная – за бесконечное время.

2) «Передача» тестового воздействия (нажатие кнопки) в реализацию и наблюдения от реализации выполняются за конечное время.

Эти предположения гарантируют наблюдение внешнего действия, выполняемого реализацией, через конечное время после нажатия кнопки, разрешающей это действие.

Эти предположения на практике всегда выполняются, так что мы не требуем ничего особенного.

1►Таким образом, остаются две основные проблемы: 1) конечность полного набора тестов и 2) конечность требуемого числа прогонов теста.

Эти проблемы не имеют решения в общем виде, поэтому такие решения приходится искать в частных случаях: либо ограничивая классы рассматриваемых спецификаций и/или реализаций, либо предполагая наличие дополнительных тестовых возможностей, либо сочетая одно с другим.

2►Одним из источников бесконечного набора тестов может быть то, что по одной трассе будет генерироваться бесконечное число тестов, отличающихся друг от друга кнопками перед наблюдениями трассы. Для того, чтобы этого избежать, достаточно конечности числа кнопок. Тогда по одной трассе будет генерироваться только конечное число примитивных тестов.

3►При конечном множестве кнопок для конечности набора тестов достаточно конечности набора трасс, по которым

генерируются тесты полного набора. Для конечной семантики это эквивалентно ограниченности длины нужных трасс.

Слайд 11.6. Проблемы практического тестирования.

Ограничения на недетерминизм реализации

Проблема конечности требуемого числа прогонов теста общая для любого вида тестирования. Гипотеза о глобальном тестировании дает только теоретическую возможность обнаружить любую ошибку в любой неконформной реализации.

На практике нам, прежде всего, требуется конечность различимых «погодных условий», то есть таких, которые определяют разное дальнейшее поведение реализации.

Кроме того, нам нужны также либо какие-то способы «управления погодой», либо гипотезы, ограничивающие возможные проявления недетерминизма реализации.

1▶ «Управление погодой» возможно в каких-то частных случаях. Примером может служить недетерминизм, который является следствием псевдопараллелизма, то есть псевдопараллельного выполнения нескольких детерминированных процессов на одном процессоре. Если мы можем вмешиваться в работу планировщика, мы тем самым можем «управлять погодой».

2▶Второй способ используется чаще всего в его экстремальном виде, когда ограничиваются только детерминированными реализациями даже при недетерминированной спецификации.

Слайд 12.6. Проблемы практического тестирования.

Три вида детерминизма реализации

Здесь мы должны уточнить понятие детерминизма для реализации.

До этого мы определяли LTS как детерминированную, если каждая ее трасса заканчивается ровно в одном состоянии. Это означает, что тау-переходов нет и в каждом состоянии определено не более одного перехода по каждому действию.

При безопасном тестировании нас, конечно, интересуют только те трассы, которые безопасны в спецификации.

Будем называть такой детерминизм *слабым детерминизмом*.

Для спецификации такого детерминихма достаточно.

Почему слабым?

Потому что для тестирования в ***R/Q***-семантике такого детерминизма реализации мало.

1▶Дело в том, что реализация может быть детерминирована в этом смысле, как здесь на слайде. Каждая её трасса заканчивается в одном состоянии. Но в начальном состоянии 0 определено два перехода по разным действиям a и b, разрешаемым одной кнопкой. Тем самым, начиная тестирование с этого состояния реализации и нажимая кнопку {a,b}, мы будем наблюдать действие a или b, которое выбирается в реализации недетерминированным образом.

2▶Для тестирования в ***R/Q***-семантике для детерминированности реализации требуется, чтобы после каждой трассы каждая кнопка разрешала ровно одно наблюдение, определённое в реализации. **Q**-кнопка определяет ровно одно действие, а **R**-кнопка определяет либо ровно одно действие, либо отказ.

При безопасном тестировании нас, конечно, интересуют только те трассы, которые безопасны в спецификации, и только те кнопки, которые после таких трасс безопасны в спецификации.

Такой детерминизм называется *наблюдаемым детерминизмом*.

В терминах тестов наблюдаемый детерминизм можно сформулировать так: безопасно-тестируемая LTS-реализация наблюдаемо детерминирована в данной ***R***/***Q***-семантике для заданной спецификации, если для любого безопасного теста при каждом его прогоне наблюдается одна и та же трасса. Поэтому каждый тест достаточно прогонять только один раз.

3▶Вот здесь на слайде показаны ещё две реализации: 2 и 3. Они наблюдаемо детерминированы, но не являются слабо детерминированными. В одной есть два перехода по действию a из одного состояния, а в другой есть тау-переход. Но после каждой трассы кнопка {ab} определяет ровно одно наблюдение: после пустой трассы – действие a, а после этого действия – отказ {ab}.

Реализация 1, наоборот, слабо детерминирована, но не является наблюдаемо детерминированной.

4▶Есть и ещё один вид детерминизма, который можно назвать *сильным детерминизмом*.

Это слабый детерминизм с дополнительным требованием: в каждом состоянии для каждой кнопки определено не более одного наблюдения, разрешаемого этой кнопкой. Это дополнительное требование означает, что в каждом состоянии определено не более одного перехода по *разным* действиям, разрешаемым одной кнопкой.

При безопасном тестировании нас, опять, интересуют только те состояния, которые достижимы по трассам, безопасным в спецификации, и только те кнопки, которые после таких трасс безопасны в спецификации.

В терминах тестов сильный детерминизм можно сформулировать так: безопасно-тестируемая LTS-реализация сильно детерминирована в ***R***/***Q***-семантике для заданной спецификации, если каждый тест при любом его прогоне всегда заканчивается в одном и том же состоянии реализации.

5▶Вот здесь на слайде показана сильно детерминированная реализация 4.

Понятно, что сильный детерминизм влечёт наблюдаемый детерминизм, но обратное вообще говоря не верно. Реализации 2 и 3 наблюдаемо детерминированы, но не сильно детерминированы.

Однако, если детерминизм наблюдаемый и слабый, то он сильный.

6▶Сильный недетерминизм – это отсутствие детерминизма любого вида. Вот здесь на слайде показаны две сильно недетерминированные реализации: 5 и 6.

В 5-ой реализации есть два перехода по действию a из одного состояния 0 – поэтому она не слабо детерминирована. Кроме того, в ней после трассы a по кнопке ab может быть как наблюдение действия b, так и отказ ab. Поэтому она не является наблюдаемо детерминированной.

В 6-ой реализации есть тау-переход – поэтому она не слабо детерминирована. Кроме того, в ней после пустой трассы по кнопке

ab может быть как наблюдение действия a, так и отказа ab. Поэтому она не является наблюдаемо детерминированной.

Слайд 13.6. Проблемы практического тестирования.

Три вида детерминизма реализации

В целом соотношение этих трёх видов детерминизма такое.

Больше всего требований к реализации предъявляет сильный детерминизм.

1▶ Слабый детерминизм ослабляет требования в одну сторону: разрешается иметь из одного состояния несколько переходов по разным действиям из одной кнопки.

2▶Наблюдаемый детерминизм ослабляет требования в другую сторону: то, что разрешает слабый детерминизм, здесь запрещается. Но зато разрешаются тау-переходы и несколько переходов из одного состояния по одному действию. Лишь бы после трассы нажатие кнопки давало ровно одно наблюдение.

3▶Всё остальное – это сильный недетерминизм.

Слайд 14.6. Проблемы практического тестирования.

Ограничения на недетерминизм реализации

В этом месте может возникнуть вопрос: почему спецификация может оказаться недетерминирована, если все исследуемые реализации считаются детерминированными, даже сильно детерминированными?

На это есть две причины.

1▶Первая причина такая: спецификация описывает не одну реализацию, а класс всех реализаций, в данном случае класс всех детерминированных реализаций, удовлетворяющих сформулированным требованиям.

Вот простой пример с квадратным корнем.

Как бы ни была задана спецификация функции вычисления квадратного корня – имплицитно или эксплицитно – она разрешает при аргументе 4 возвращать как +2, так и -2. Это наблюдаемый недетерминизм. Реализация всегда может возвращать что-то одно, но

только не 3 и не 5. Так функция вычисления квадратного корня обычно и устроена, то есть обычно она сильно детерминирована.

Слайд 15.6. Проблемы практического тестирования.

Ограничения на недетерминизм реализации

Вторая причина: недетерминизм спецификации может быть следствием повышения уровня абстракции при задании спецификационных требований. При таком повышении уровня абстракции мы абстрагируемся от некоторых деталей. Среди этих деталей могут оказаться и «погодные условия», определяющие детерминированное поведение реализации.

Например, программа распределения памяти блоками переменной длины работает сильно детерминированно. У неё есть структура данных, которая детально описывает, сколько блоков памяти занято, какой длины каждый блок и какой у него начальный адрес. Соответственно, имеется или может быть построен список свободных блоков с их размерами и начальными адресами.

При запросе памяти указывается размер требуемого блока памяти. Чтобы запрос мог быть удовлетворён, этот размер, конечно, не должен превышать суммарный объём свободной памяти A. Но этого недостаточно для того, чтобы определить, будет запрос удовлетворён или нет. Нужно ещё, чтобы был свободным блок подходящего размера. Если свободная память сильно фрагментирована, запрос может быть неудовлетворён даже в том случае, когда запрашиваемый размер памяти много меньше A (суммарного объёма свободной памяти).

Однако при составлении некоей грубой спецификации мы можем отвлечься от этих деталей и свести всю структуру данных только к суммарному объёму свободной памяти – A и числу занятых блоков – n. При этом мы всё же можем предъявить к реализации некоторые спецификационные требования, хотя и очень простые:

1) Если запрашиваемый размер превышает A – суммарный объём свободной памяти, запрос не должен быть удовлетворён.

2) Если запрашиваемый размер не превышает A/(n+1) – суммарного объёма свободной памяти, делённого на число занятых блоков плюс 1, то запрос должен быть удовлетворён. Это объясняется

тем, что n+1 – это максимальное число свободных блоков, поэтому хотя бы один из них должен быть не меньше, чем A/(n+1).

Однако такая спецификация оказывается наблюдаемо недетерминированной: если размер запрашиваемого блока не больше A, но превышает A/(n+1), то запрос может быть как удовлетворён, так и нет.

Слайд 16.6. Проблемы практического тестирования.

Ограниченный недетерминизм (t-недетерминизм)

Мы используем следующую гипотезу о *t*-недетерминизме:

если в состоянии реализации *i* кнопку *P* нажимать *t* раз, то реализация продемонстрирует все возможные варианты поведения, то есть будут получены все возможные пары (наблюдение, постсостояние).

При этом нажатия кнопки P в данном состоянии могут чередоваться с рестартами, нажатием кнопок в других состояниях и других кнопок в этом состоянии.

При t>1 реализация может быть даже сильно недетерминированной.

При *t*=1 мы имеем сильно детерминированную реализацию.

1► Соотношение между видами детерминизма видно на слайде для t>1.

Слайд 17.6. Проблемы практического тестирования.

Ограниченный недетерминизм (t-недетерминизм)

Примером ограниченного недетерминизма может служить недетерминизм как следствие псевдопараллелизма, если у нас есть возможность вмешиваться в работу системы.

Здесь предполагается, что каждый процесс работает вполне детерминированно, а недетерминизм возникает в результате параллельной композиции процессов.

Если один процесс при нажатии кнопки ab выполняет только действие a, а другой – только действие b, и если эти действия асинхронные, то есть остаются в алфавите композиции, то в

композиционном состоянии будет два перехода по кнопке ab: по действию a и по действию b.

Например, действия a и b – это посылка сообщений. Оба процесса готовы выдать свои сообщения. Какое сообщение будет выдано первым, а какое вторым, зависит от системы.

Также, если возникают синхронные переходы, то в композиции будут тау-переходы. Такие переходы соответствуют взаимодействию процессов между собой.

Однако, вмешиваясь в работу системы, мы можем на каждом шаге перебирать все возможные варианты выбора того или иного активного процесса на выполнение. Иными словами, мы можем управлять выбором одного из нескольких возможных переходов композиции при нажатии той или иной кнопки, включая тау-переходы.

Если у нас есть t процессов, то на каждом шаге мы выбираем на выполнение один из активных процессов, а их не больше, чем t. Если мы будем делать выбор детерминированно, просто перебирая процессы в текущей ситуации, то получится t-недетерминизм.

Слайд 18. 6. Проблемы практического тестирования.

Ограниченный недетерминизм (t-недетерминизм)

Другим примером ограниченного недетерминизма может служить тестирование с преобразованием семантик.

Дело в том, что семантики реализации и спецификации могут различаться. Спецификация, как правило, абстрагируется от некоторых деталей взаимодействия. Её семантика обобщённая.

Связь семантик при тестировании осуществляет специальная программа в составе тестовой системы. Она называется медиатором. Его задача: преобразовывать кнопки спецификационной семантики в кнопки реализационной семантики. А также делать обратное преобразование реализационных наблюдений в спецификационные.

Из-за различия семантик детерминированная реализация может казаться недетерминированной при тестировании её на спецификационной машине тестирования. Мы нажимаем в одном и том же состоянии два раза одну и ту же спецификационную кнопку,

но медиатор преобразует их в две разные реализационные кнопки. Соответственно, реализация может выполнять два разные реализационных действия. И эти действия медиатор может преобразовать тоже в разные спецификационные действия. Получится, что при нажатии одной и той же спецификационной кнопки два раза в одном и том же состоянии реализации мы получаем два разных спецификационных наблюдения. А это и значит, что реализация недетерминирована.

Пусть каждая спецификационная кнопка преобразуется медиатором не более, чем в t реализационных кнопок. Пусть также медиатор работает детерминированно, просто перебирая в текущей ситуации реализационные кнопки, соответствующие данной спецификационной кнопке. Тогда у нас получается t-недетерминизм.

1▶ Например, функция вычисления квадратного корня из 4 может возвращать +2 или -2 в зависимости от дополнительного булевского аргумента, который как раз и говорит, какой корень нужно вернуть: положительный или отрицательный. Медиатор может поочерёдно сообщать в качестве значения этого дополнительного аргумента то ***true***, то ***false***, тем самым гарантируя получение то +2, то -2 при чётных и нечётных обращениях. Здесь t=2.

Слайд 19. 6. Проблемы практического тестирования.

Тестирование с открытым и закрытым состоянием

Одной из дополнительных тестовых возможностей является опрос текущего состояния реализации – status message. Предполагается, что эта операция всегда выполняется правильно и ее не нужно тестировать.

Если можно «подсматривать» состояния реализации, говорят о тестировании с открытым состоянием, в противном случае – о тестировании с закрытым состоянием. Дальше мы рассмотрим эти два вида тестирования по отдельности и будем вводить соответствующие ограничения на классы реализаций и/или спецификаций.

1▶Опрос текущего состояния – это очень сильная тестовая возможность. Дальше мы продемонстрируем как эта возможность обеспечивает полное тестирование за конечное время, хотя при

отсутствии этой возможности - тестировании с закрытым состоянием – нам понадобился бы бесконечный набор тестов.

7. Тестирование с закрытым состоянием

Слайд 20. 6. Проблемы практического тестирования.

LTS- и RTS-спецификация

Мы уже говорили о процедуре детерминизации LTS, которая позволяет по LTS построить детерминированную RTS с тем же множеством трасс.

Я напомню процедуру детерминизации: состояниями RTS становятся множества состояний исходной LTS. Переход по символу ведёт из множества A во множество B, если B – это множество состояний, в которых заканчиваются трассы из одного этого символа, начинающиеся в состояниях из A. При этом B не пусто. Переход по дивергенции из множества A проводится в терминальное состояние, если в A есть дивергентные состояния.

Если LTS имеет k состояний, то RTS будет иметь число K состояний, которое не более, чем 2^k-1.

Будем считать, что отношение ***safe by*** ограниченное: после всех трасс, заканчивающихся в RTS в одном состоянии, безопасность кнопок одинаковая.

Слайд 21. 7. Тестирование с закрытым состоянием.

Простые тесты

Сначала рассмотрим тестирование без тестовой возможности опроса состояний, то есть тестирование с закрытым состоянием.

Путь задана RTS-спецификация ***S*** с ограниченным отношением ***safe by*** и рассмотрим некоторую произвольную безопасно-тестируемую LTS-реализацию ***I***.

Пусть σ безопасная трасса спецификации, а *P* - кнопка, безопасная в спецификации после трассы σ.

Обозначим через s_σ (единственное) состояние спецификации, в котором заканчивается трасса σ, а через i_σ – одно из состояний реализации ***I***, в которых заканчивается трасса σ.

1►Если для каждой тройки (s_σ, i_σ, P) в наборе тестов будет примитивный тест, сгенерированный по какой-нибудь трассе σ` такой, что $s_\sigma = s_{\sigma`}$, в котором после трассы σ` нажимается кнопка *P*, то такой набор тестов будет полным.

Почему?

Да потому, что неважно, после какой трассы мы нажимаем кнопку P, если после этой трассы мы оказались в спецификации в состоянии s_σ, а в реализации – в состоянии i_σ.

Возможные в реализации наблюдения зависят только от состояния i_σ и кнопки P.

Сама кнопка P, если безопасна после одной безопасной трассы спецификации, заканчивающейся в состоянии s_σ, то безопасна и после других безопасных трасс, заканчивающихся в этом состоянии.

А допустимые наблюдения, определяются спецификацией только в зависимости от состояния s_σ и кнопки P.

2►При однократном прогоне теста, сгенерированного по трассе σ, на реализации ***I*** мы получаем последовательность пар состояний (s_μ, i_μ), где μ - префикс трассы σ.

3►Будем говорить, что тест *простой* для данной реализации ***I***, если хотя бы при одном его прогоне в полученной последовательности пар состояний нет одинаковых пар.

Очевидно, что для полноты тестирования данной реализации ***I*** достаточно тестов, которые простые для этой реализации.

Почему?

Потому что, если тест не простой, то в последовательности пар состояний (s_μ, i_μ) имеются два вхождения одной пары. И мы можем просто выбросить кусок теста между этими двумя вхождения одной пары.

Для полноты тестирования всех реализаций достаточно, чтобы в набор тестов входили все тесты, которые просты для той или иной реализации.

Слайд 22. 7. Тестирование с закрытым состоянием.

Оценка длины простого теста

Оценим сверху длину *N* большое простого теста.

Очевидно, $N \le nK \le n(2^k-1)$, где

n – число состояний реализации,

K – число состояний RTS-спецификации,

k – число состояний исходной LTS-спецификации.

Если набор тестов состоит из всех тестов длины не больше *N*, то такой набор тестов содержит все простые тесты для всех реализаций с числом состояний не больше *n*. Следовательно, он полный на классе таких реализаций ограниченного размера.

1►Эта верхняя оценка является точной по порядку: для любых *k* и *n* существует семантика с O(*k*) действиями, LTS-спецификация с O(*k*) состояниями и неконформная реализация с O(*n*) состояниями, ошибка в которой не может быть обнаружена тестом длины меньше $O(n2^k)$.

Слайд 23. 7. Тестирование с закрытым состоянием.

Оценка длины простого теста

Можно показать, что даже для семантики с ограниченным числом действий оценка остается суперполиномиальной.

В частности, существует семантика с двумя действиями, для которой в точной верхней оценке показатель степени *k* заменяется на $c(\boldsymbol{ln}\ k)^2$.

1►LTS-спецификации, на которых достигаются эти оценки, существенно недетерминированы.

В фиксированной семантике для детерминированной LTS-спецификации K=**O**(k), а не $\mathbf{O}(2^k)$.

2►Однако даже минимальный недетерминизм LTS-спецификации, когда нет τ-переходов и только в одном состоянии только по одному действию имеются два перехода в разные состояния, оставляет оценки суперполиномиальными.

Слайд 24. 7. Тестирование с закрытым состоянием.

Оценка числа прогонов простого теста

Число прогонов примитивного теста с N кнопками при t-недетерминизме равно $O(t^N)$.

Почему?

Если мы t раз прогоняем тест, то реализация пройдёт t маршрутов, причём первые пары (наблюдение, постсостояние) в этих маршрутах будут всеми возможными парами, которые в реализации разрешает в начальном состоянии первая кнопка.

Если мы t^2 раз прогоняем тест, то реализация пройдёт t^2 маршрутов, причём каждая первая пара (наблюдение, постсостояние) в этих маршрутах будет получена не менее t раз. Поэтому в этих маршрутах будут все возможные последовательности из двух пар (наблюдение, постсостояние), соответствующих первым двум кнопкам.

И так далее.

В итоге получается требуемая оценка.

Эта оценка достижима для некоторых реализаций, но, конечно, для каких-то реализаций она может оказаться гораздо меньше.

1►В частности, для детерминированных реализаций тест достаточно прогонять один раз.

Разрыв между число 1 и экспоненциальной оценкой, конечно, слишком велик, чтобы его не замечать. Это говорит о том, что дальнейшие исследования в этой области были бы крайне полезны. А именно: нужно искать такие ограничения на недетерминизм реализации, которые были бы, с одной стороны, практически значимыми, то есть определяли бы практически значимые классы реализаций, а, с другой стороны, давали бы, например, полиномиальные оценки.

8. Тестирование с открытым состоянием

Слайд 25. 8. Тестирование с открытым состоянием.

Один адаптивный тест с рестартом

Теперь мы переходим к тестированию с дополнительной тестовой возможностью опроса состояний реализации. Это тестирование с открытым состоянием.

При таком тестировании мы будем строить не набор тестов, а один адаптивный тест с возможными рестартами в середине теста.

1▶Для полноты тестирования нужно проверить все переходы реализации, лежащие на маршрутах с трассами, безопасными в спецификации.

2▶Для этого, поскольку у нас только один тест, LTS-реализация должна быть сильно-связной: из каждого состояния достижимо по переходам каждое другое.

Рестарт понимается как одно из внешних действий, отличающихся только тем, что гарантированно переводит реализацию в начальное состояние.

Переход по рестарту делает трассу пустой.

Переходы по рестарту дополнительно учитываются при определении сильно-связности.

Слайд 26. 8. Тестирование с открытым состоянием.

Один адаптивный тест с рестартом

Также требуется, чтобы начальное состояние реализации было стабильным или хотя бы в одном состоянии, достижимом по безопасной трассе спецификации, был определен рестарт.

Действительно, в примере реализация в начале тестирования может перейти в состояние 1 по тау-переходу.

После этого реализация может попасть в начальное состояние 0 либо по рестарту, либо по трассе <*x*>.

Но после этой трассы в спецификации кнопка {*y*} опасна, поскольку вызывает разрушение после действия y. Её нельзя нажимать при тестировании.

В реализации есть ошибка: в начальном состоянии есть переход по y. Получается, что эту ошибку нам не удастся обнаружить, если нельзя сделать рестарт реализации в состоянии 1.

Слайд 27.8. Тестирование с открытым состоянием.

*Построение модели реализации **I** в процессе тестирования*

В процессе тестирования мы будем получать наблюдения и опрашивать состояния реализации. Это даст нам возможность поэтапно строить LTS-реализацию ***I*** с одновременной проверкой тестируемого условия.

Более точно: строится LTS-модель, имеющая такое же как в реализации множество трасс, безопасных в спецификации, и такое же множество состояний, достижимых по этим трассам.

При этом переходы-петли по отказам мы будет рисовать явно.

Слайд 28.8. Тестирование с открытым состоянием.

*Построение модели реализации **I** в процессе тестирования*

В начале тестирования и после каждого перехода опрашиваем состояние реализации.

Переход $i—z\rightarrow i`$ по внешнему действию $z$ добавляется, когда после опроса состояния $i$ нажимается кнопка $P$, после чего наблюдается действие $z\in P$, а затем опрашивается постсостояние $i`$.

Если наблюдается отказ с *тем же самым* постсостоянием $i`=i$, то добавляется виртуальный переход-петля по отказу $i—P\rightarrow i$.

Если отказ P наблюдается с *другим* постсостоянием $i`\neq i$, то добавляются переходы $i—\tau\rightarrow i`—P\rightarrow i`$.

Вместе с каждым переходом по внешнему действию $i—z\rightarrow i`$ будем хранить кнопку, нажатие которой вызвало этот переход. Это *управляющая* кнопка перехода.

Слайд 29.8. Тестирование с открытым состоянием.

Соответствие состояний

В процессе тестирования будем строить соответствие между состояниями реализации и спецификации: i $\leftrightarrow$ s.

Состояния соответствуют друг другу, если они достижимы по одной и той же трассе, которая безопасна в спецификации.

В частности, начальное состояние спецификации и каждое состояние спецификации после пустой трассы соответствуют каждому состоянию реализации, которое опрашивается в начале тестирования и после каждого рестарта.

Слайд 30.8. Тестирование с открытым состоянием.

Допустимые в состоянии кнопки и их счётчики

Будем говорить, что кнопка *P допустима в* реализационном состоянии *i*, если она безопасна хотя бы в одном соответствующем состоянии спецификации.

Только допустимые кнопки будут нажиматься в состоянии *i*.

Для каждой допустимой кнопки *P* определим счётчик числа ее нажатий в состоянии *i*.

Кнопка *P полна в состоянии* i, если

1) счетчик=*t* или

2) счетчик=1 и в состоянии *i* есть переход-петля по отказу *P*.

В обоих случаях уже получены все возможные переходы из состояния *i* при нажатии кнопки *P*.

Состояние *полно*, если каждая допустимая в нем кнопка полна.

Слайд 31.8. Тестирование с открытым состоянием.

Начало тестирования

В начале тестирования после опроса состояния это состояние реализации соответствует каждому состоянию спецификации в конце пустой трассы.

Других состояний реализации пока нет. Переходов тоже нет.

Счётчик каждой кнопки в этом состоянии равен 0.

Слайд 32.8. Тестирование с открытым состоянием.

Общая схема работы алгоритма тестирования

На рисунке изображена общая схема работы алгоритма.

Сначала проверяем полноту текущего состояния.

1►Если текущее состояние полное, то для продолжения тестирования нужно перейти в неполное состояние.

2►Если таких состояний нет, алгоритм заканчивается с вердиктом ***pass***.

3►Рассмотрим переход в неполное состояние.

Слайд 33.8. Тестирование с открытым состоянием.

Общая схема работы алгоритма тестирования

В построенном графе LTS реализации всегда можно выбрать множество деревьев, покрывающих все опрошенные состояния так, чтобы из каждого состояния выходило не более одного перехода, принадлежащего деревьям, и все эти деревья ориентированы к своим корням, которыми являются все неполные состояния.

Будем двигаться по переходам этих деревьев, нажимая управляющие кнопки этих переходов. Из-за недетерминизма при попытке пройти по некоторому переходу мы можем оказаться не в постсостоянии этого перехода, а в другом состоянии, то есть пройти другой переход. Если мы попали в неполное состояние, мы снова будем нажимать управляющую кнопку. За конечное число шагов мы гарантированно попадем в неполное состояние.

Слайд 34.8. Тестирование с открытым состоянием.

Общая схема работы алгоритма тестирования

1►Если текущее состояние неполное, то выбираем неполную кнопку, нажимаем ее и получаем один переход или два перехода.

В любом случае можно считать, что мы получаем новую трассу, состоящую из одного наблюдения u, начинающуюся в состоянии i и заканчивающуюся в состоянии i`.

Это наблюдение u равно действию z из кнопки P или отказу P.

Постсостояние i` становится новым текущим состоянием.

Корректируем счетчик нажимавшейся кнопки в состоянии i.

2▶Переход, который мы получили, может быть новым или построенным ранее.

3▶Если получен старый переход, возвращаемся к началу работы алгоритма.

4▶Если получен новый переход, то добавляем его в строящуюся реализацию.

После этого выполняется блок «Распространение с верификацией».

Слайд 35.8. Тестирование с открытым состоянием.

Общая схема работы алгоритма тестирования

Проверяем тестируемое условие для полученной новой трассы $i=u\Rightarrow i`$ и каждого состояния s спецификации, которое $i \leftrightarrow s$:

u безопасно в s $\Rightarrow \exists s`\ s=u\Rightarrow s`$.

Если не было соответствия $i` \leftrightarrow s`$, то делаем это соответствие $i` \leftrightarrow s`$ и проверяем тестируемое условие для каждой уже имеющейся трассы $i`=v\Rightarrow i``$ и состояния s`. И т.д.

1▶Если ошибка – ***fail***.

2▶Иначе – на начало.

Слайд 36.8. Тестирование с открытым состоянием.

Пример работы алгоритма тестирования

Посмотрим, как работает этот алгоритм на примере.

На слайде изображены семантика, спецификация и реализация.

Это ioco-семантика. Имеются две реакции a и b, и один стимул x, который в спецификации определен только в начальном состоянии. Это значит, что стимул x мы можем безопасно посылать в реализацию только после рестарта, но не после получения какой-нибудь реакции.

Спецификация детерминирована в обычном смысле детерминизма LTS: каждая её трасса кончается в одном состоянии. Поэтому процедура детерминизации такой LTS-спецификации ничего по сути не меняет. Мы можем считать это RTS-спецификацией.

Реализация недетерминирована в обычном смысле, поскольку в начальном её состоянии есть два перехода по одному символу x. Но даже если бы перехода-петли по x не было, она всё равно не была бы наблюдаемо детерминированной: в ней есть состояние 1, в котором определено два перехода по реакциям – a и b.

Но для нашего алгоритма это не так важно. Нам важно, чтобы выполнялась гипотеза о t-недетерминизме. Для примера будем предполагать, что t=2. Это значит, что в каждом состоянии нажатие кнопки приёма реакций два раза вызовет наблюдение всех реакций, которые есть в этом состоянии со всеми постсостояниями. В состоянии 1 это две реакции a и b, а в остальных состояниях – только одна реакция a.

В процессе тестирования мы будем заполнять вот такую таблицу.

В левом столбце будем записывать наблюдаемые переходы.

Во втором столбце будем записывать соответствие состояний спецификации и реализации.

Для каждой кнопки в каждом наблюдаемом состоянии реализации мы будем поддерживать счётчик нажатий кнопки.

В последнем столбце комментариев будет ссылка на те или иные блоки алгоритма, которые используются.

Как только мы пройдём какой-нибудь переход в реализации, сразу будет заполняться соответствующая ему строчка таблицы.

1▶Итак, мы начинаем тестирование. Реализация нам неизвестна.

2▶Всё начинается с рестарта, чтобы реализация гарантированно сбросилась в начальное состояние. Опрашиваем состояние реализации и записываем соответствие начальных состояний спецификации и реализации. Для состояния реализации 0 счётчики кнопок равны нулю – мы ещё не нажимали ни одной управляющей

кнопки, но каждая из них безопасна в начальном состоянии спецификации.

Нам нужно нажать кнопку, которая неполна в текущем состоянии 0. Таких кнопок две. Выбираем произвольным образом кнопку приёма реакций.

3▶Нажимаем выбранную кнопку и наблюдаем реакцию a, опрашиваем состояние – это будет состояние 1. Записываем переход из 0 по a в 1. Увеличиваем счётчик кнопки приёма реакций в состоянии 0. Это мы выполнили блок «Воздействие + Наблюдение».

Далее верифицируем полученный переход. Нам нужно проверить, что в состоянии спецификации, которое соответствует началу перехода, тоже есть переход по реакции a. Проверяем – всё правильно. Конец этого перехода в спецификации – состояние 1. Записываем соответствие состояний 1 и 1.

Для нового состояния 1 счётчик приёма реакций устанавливаем в 0. А вот вместо значения счётчика для кнопки посылки стимула в таблице записан прочерк. Это потому, что в соответствующем состоянии спецификации – 1 – кнопка посылки стимула x опасна: в этом состоянии нет перехода по x.

Снова нужно нажать кнопку, которая неполна в текущем состоянии 1. Такая кнопка одна – приём реакций.

4▶Нажимаем её, наблюдаем реакцию a и опрашиваем постсостояние 2. Записываем полученный переход. Увеличиваем счётчик кнопки приёма реакций в состоянии 1. Мы снова выполнили блок «Воздействие + Наблюдение».

Опять верифицируем полученный переход. Нам нужно проверить, что в состоянии спецификации 1 тоже есть переход по реакции a. Проверяем – всё правильно. Конец этого перехода в спецификации – состояние 2. Записываем соответствие состояний 2 и 2.

Для нового состояния 2 счётчик приёма реакций устанавливаем в 0. А вместо значения счётчика для кнопки посылки стимула в таблице опять прочерк. В соответствующем состоянии спецификации – 2 – кнопка посылки стимула x опасна: в этом состоянии нет перехода по x.

5▶Всё повторяется аналогично и теперь. Нажимаем кнопку приёма реакций, получаем реакцию a. Но теперь, опросив постсостояние, мы видим, что оно не новое, мы там уже были. Тем не менее, нам нужно проверить, что в соответствующем спецификационном состоянии 2 тоже есть переход по реакции a. Проверяем: он там есть и ведёт в состояние 1. Это состояние уже соответствует реализационному состоянию 1. Поэтому на этом проверка заканчивается, новых соответствий состояний не появляется.

Кнопку приёма реакций в текущем состоянии 1 мы уже нажимали, но только один раз, а t=2.

6▶ Поэтому нажимаем эту кнопку 2-ой раз. И действительно, наблюдаем другую реакцию b. Аналогично проводится верификация. У нас новый переход и новое соответствие состояний 6 и 6.

7▶В состоянии 6 аналогично принимаем реакции. Получаем реакцию a и постсостояние 2, в котором мы уже были. В спецификации переход из состояния 6 по a тоже есть и тоже ведет в состояние 2, уже соответствующее реализационному состоянию 2.

Кнопку приёма реакций в текущем состоянии 2 мы уже нажимали, но только один раз, а t=2.

8▶ Поэтому нажимаем эту кнопку 2-ой раз. Но мы получаем старый переход по реакции a, ведущий в состояние в 1. Поэтому верификация не нужна.

В состоянии 1 кнопку приёма реакций мы уже нажимали 2 раза. Эта кнопка полна в состоянии 1. А кнопка посылки стимула недопустима. Следовательно, наше состояние 1 полное. Нам нужно перейти в неполное состояние. Среди пройденных состояний есть 2 таких состояния: 0 и 6. Но в 0 пути по пройденным переходам нет, туда можно попасть только через рестарт.

Наше дальнейшее поведение зависит от двух вещей:

1) Можно ли попасть из текущего состояния в неполное состояние без рестарта.

2) Как мы относимся к рестарту: считаем его дорогой операцией или дешёвой. Для многих систем рестарт операция дорогая, требует много времени. Поэтому обычно стараются делать рестарт только в

случае крайней необходимости – когда в неполное состояние нельзя попасть без рестарта.

Здесь можно отметить, что если граф реализации сильно-связен без учёта рестартов, то всегда можно попасть в неполное состояние, если такое есть. Поэтому для сильно-связных реализаций можно вообще обойтись без рестарта. Но если реализация не сильно-связна, рестарт понадобится.

В нашем примере реализация не сильно-связна: состояние 0 достижимо только из самого себя. Но мы находимся в состоянии 1, а это состояние принадлежит той же компоненте сильно-связности, что и неполное состояние 6. Поэтому будем стараться попасть в состояние 6. Тем более, что в нашем примере из 1 в 6 ведёт всего-навсего один переход.

9▶Поэтому нажимаем управляющую кнопку этого перехода. Но тут нам не повезло: в этом месте реализация недетерминирована: вместо желаемой реакции b мы получаем реакцию a и попадаем не в состояние 6, а в состояние 2, которое уже полное.

Но не нужно отчаиваться. Из состояния 2 тоже можно попасть в состояние 6 по двум переходам: из 2 в 1 и из 1 в 6. В состоянии 2 у нас есть тоже управляющая кнопка перехода из 2 в 1.

10▶Нажимаем эту кнопку. Тут уже гарантированно попадаем в состояние 1, поскольку других переходов из состояния 2 в реализации нет: состояние полное и все переходы из него мы уже получили.

11▶Опять нажимаем в состоянии 1 управляющую кнопку перехода из 1 в 6. Это уже её второе нажатие. Поскольку t=2 нам гарантируется, что мы получим на этот раз нужный нам переход.

Итак, мы попали в неполное состояние 6. В нём неполна кнопка приёма реакций, кнопка посылки стимула недопустима.

12▶Нажимаем эту кнопку и получаем повторный переход из 6 по a в 2. Верификацию делать не надо.

Вот теперь мы видим, что из состояния 2 нельзя попасть в неполное состояние без рестарта. Все состояния, куда можно попасть без рестарта, это полные состояния 1, 2 и 6.

13▶Делаем рестарт.

В начальном состоянии 0, по-прежнему, кнопка приёма реакций неполна.

14▶Нажимаем её второй раз. Но мы получаем опять старый переход. Верификацию делать не надо. И опять мы попадаем в состояние 1, откуда без рестарта недостижимо всё ещё неполное состояние 0. Заметим, что состояние 0 неполно теперь уже только из-за кнопки посылки стимула, которая в нём допустима, но мы её ни разу не нажимали.

15▶Делаем рестарт.

16▶В состоянии 0 посылаем стимул x и попадаем в новое состояние 4. Корректируем счётчики, верифицируем новый переход, устанавливаем соответствие между состояния 4 и 4.

17▶Опять принимаем реакции в состоянии 4, получаем реакцию a и новое постсостояние 5. Корректируем счётчики, верифицируем новый переход, устанавливаем соответствие между состояния 5 и 5.

18▶В состоянии 5 получаем реакцию a и старое постсостояние 6. Из спецификационного состояния 5 тоже есть переход по a в состояние 6. Поскольку состояния 6 и 6 уже соответствуют друг другу, дальнейшая верификация не требуется.

Мы опять оказались в полном состоянии из которого по пройденным переходам недоступны неполные состояния 0, 4 и 5 без рестарта.

19▶Делаем рестарт.

Теперь в состояни 0 нам осталось только 2-ой раз нажать кнопку посылки стимула x.

20▶Нажимаем эту кнопку и получаем переход-петлю. Начинаем распространение с верификацией. Здесь уже распространение будет не такое короткое, как раньше.

В спецификации из состояния 0 переход по x ведёт только в состояние 4. Из этого следует, что мы должны попытаться установить соответствие между реализационным состоянием 0 и спецификационным состоянием 4.

21▶Проверяем, что из реализационного состояния 0 ведут переходы по x в 0 и 4 и по a в 1. Но кнопка посылки стимула опасна в

спецификационном состоянии 4, а переход по a тоже есть и ведёт в состояние 5.

Теперь нужно добавить соответствие состояний 5 и 1. Из реализационного состояния 1 ведут переходы по a в 2 и по b в 6. Из спецификационного состояния 5 тоже есть переход по a в 6 и петля по b. Нам нужно установить соответствие состояний 6 и 2, а также 5 и 6.

И так далее. Я не буду дальше рассказывать распространение с верификацией в этом примере. Ход верификации показан на слайде. Скажу только, чем она закончится через несколько шагов.

22▶А закончится она тем, что мы увидим в реализации трассу xaaa, начинающуюся в начальном состоянии и заканчивающуюся в состоянии 1 и состоянии 2. В спецификации тоже есть такая трасса, она заканчивается в состоянии 2. Состояния 2 и 2 соответствуют друг другу. А вот состояния 2 и 1 не соответствуют. Почему? Потому что в реализационном состоянии 1 есть переход по b, а в спецификационном состоянии 2 нет перехода по b, хотя кнопка приёма реакций безопасна. Это значит, что трасса xaaab имеется в реализации, но является ошибкой в спецификации.

Хочу обратить внимание на то, что эту трассу мы не проходили при тестировании. Мы проходили только её префикс – одно действие x и убедились, что это переход-петля. Зато мы проходили постфикс трассы aaab. Иными словами, мы эту трассу вычислили на основании тех переходов, которые проходили.

Можно также отметить, что если бы в реализации не было петли по x в состоянии 0, то она была бы конформна спецификации. И это выяснилось бы после того, как мы повторно нажали кнопки приёма реакций в состояниях 4 и 5, где она нажималась только 1 раз.

Слайд 37.8. Тестирование с открытым состоянием.

Сравнение тестирования с открытым и закрытым состоянием

Я хочу ещё раз обратить внимание на особенность тестирования с открытым состоянием, которые мы подметили на примере.

При тестировании с открытым состоянием верифицируются не только наблюдения, полученные после *реальных* трасс, пройденных при тестировании, но и возможные наблюдения после *потенциальных*

трасс, то есть наблюдения и трассы, про которые установлено, что они есть в реализации. Это даёт существенную экономию числа тестовых воздействий, необходимых для проверки конформности: мы выполняем множество проверок без реального тестирования, основываясь на полученном знании о поведении реализации.

Например, если при тестировании получены две трассы $\mu_1 \cdot \lambda$ и μ_2, где трассы μ_1 и μ_2 заканчиваются в реализации в одном состоянии i, то мы можем проверить обе трассы: как $\mu_1 \cdot \lambda$, которую реально прошли при тестировании, так и потенциальную трассу $\mu_2 \cdot \lambda$.

Это преимущество дает дополнительная тестовая возможность опроса состояния реализации.

Слайд 38. 8. Тестирование с открытым состоянием.

Оценки сложности алгоритма. Число тестовых воздействий

При тестировании обычно наиболее важным считается число тестовых воздействий.

Оценка экспоненциальная в общем случае и полиномиальная для детерминированных реализаций.

1►Такая полиномиальная оценка достигается не только для детерминированной реализации, но и во всех случаях, когда переход в неполное состояние можно гарантированно выполнить, проходя *путь* (маршрут без самопересечений), длина которого ограничена n.

Слайд 39. 8. Тестирование с открытым состоянием.

Оценки сложности алгоритма. Число тестовых воздействий

Рассмотрим три таких случая:

1. Сильно-Δ-связные LTS-реализации. Это такие LTS, в которых для любой пары состояний a и b можно в каждом состоянии i найти такую кнопку $P(i)$, что, нажимая только такие кнопки, мы гарантированно окажемся в состоянии b, хотя путь из a в b, который мы проходим, зависит от недетерминированного поведения LTS. Детерминированные LTS — это частный случай сильно-Δ-связных LTS.

1►2. Упомянутый выше случай недетерминизма как следствие псевдопараллелизма, когда мы можем вмешиваться в работу планировщика процессов, тем самым «управляя погодой».

Мы можем запоминать в каком порядке выполнялись процессы при каком-то прохождении трассы, то есть при прохождении некоторого маршрута с этой трассой, и воссоздавать этот порядок, когда нужно повторно пройти тот же маршрут.

2►3. Недетерминизм как следствие повышения уровня абстракции при тестировании с преобразованием семантик.

При тестировании связь реализационной и спецификационной семантик осуществляется программой-медиатором. В этом случае при нажатии в состоянии *i* кнопки *P* медиатор может сообщить тесту некий дополнительный параметр, от которого абстрагируется модель. Этот параметр, по сути, является реализационной кнопкой, в которую преобразуется медиатором спецификаицонная кнопка. Если есть возможность при повторном нажатии в состоянии *i* кнопки *P* сообщить медиатору этот параметр, то гарантированно выполнится тот же реализационный переход. Конечно, если реализация детерминирована в реализационной семантике.

Слайд 40.8. Тестирование с открытым состоянием.

Оценки сложности алгоритма. Объем вычислений

Оценка объема вычислений, кроме тестовых воздействий, содержит три слагаемых:

1) вычисления, необходимые для поиска опрошенного состояния среди пройденных при каждом тестовом воздействии, имеют экспоненциальную оценку в общем случае и полиномиальную для детерминированных реализаций;

2) построение множества деревьев имеет полиномиальную оценку;

3) вычисления в блоке «Распространение с верификацией» имеют O от m K большое.

9. Развитие теории конформности

Слайд 41.9. Развитие теории конформности.

Проблема монотонности

Теперь я хочу сказать несколько слов о дальнейшем развитии теории безопасной конформности. Какие-то проблемы здесь уже решены, а какие-то ещё предстоит решить.

Первой проблемой является проблема верификации сложных составных систем. Такие системы компонуются инкрементально и иерархически из компонентов разных уровней.Тестированию композиционных систем, последние годы уделяется большое внимание. Основная проблема композиции – это проблема соотношения спецификации системы и спецификаций её компонентов.

Неформально проблема звучит так: если компоненты работают правильно, то почему система в целом работает неправильно?

Вот пусть у нас есть реализации компонентов, которые правильные, то есть конформны своим спецификациям.

1▶Из этих реализаций компонентов собирается реализация системы.

Если компоненты моделируются LTS, то композиция моделируется оператором параллельной композиции LTS. Мы о нём уже говорили.

2▶Будет ли такая реализация системы правильной, то есть будет ли она конформна имеющейся спецификации системы?

3▶В общем случае, конечно, нет.

Почему?

4▶Потому что, прежде всего, у нас должно быть корректное соотношение между спецификациями компонентов и спецификацией составной системы. Это соотношение корректно, если композиция компонентов, конформных своим спецификациям, всегда конформна спецификации системы.

К сожалению, это не всегда так. Проверку правильности такого соотношения называют верификацией декомпозиции системных требований.

Конечно, если спецификация системы никак не связана со спецификациями компонентов, то трудно ожидать их правильного соотношения.

5▶Казалось бы, спецификацию системы можно получить как композицию спецификаций компонентов, выполняя её по тем же правилам, по которым сама система компонуется из реализаций компонентов. Для LTS-спецификаций опять используем оператор параллельной композиции. Получаем композицию спецификаций компонентов.

6▶Если конформность сохраняется при композиции, то это называют *монотонностью*.

А именно: композиция реализаций компонентов, конформных спецификациям этих компонентов, должна быть конформна композиции этих спецификаций.

7▶К сожалению, это далеко не так. Это и есть проблема монотонности.

Эта проблема вызвана разными уровнями абстракции, используемыми в определениях конформности и прямой композиции. Конформность основана на трассах наблюдений над поведением реализационной модели, а композиция, кроме того, на ненаблюдаемых напрямую состояниях и ненаблюдаемых действиях (τ-действиях).

Отсутствие решения проблемы монотонности препятствует верификации системных требований.

Наоборот, если найти такое решение, то можно не только верифицировать на корректность имеющуюся спецификацию системы, но и автоматически получить корректную спецификацию по заданным спецификациям компонентов., просто скомпоновав их.

Сгенерированную спецификацию системы можно использовать как описание системы для её пользователей, как «техническое задание» разработчику системы при дальнейших модификациях системы и её компонентов, когда предполагается сохранение внешней функциональности системы, и, наконец, для генерации системных тестов.

8▶Частным случаем проблемы композиции является *асинхронное тестирование* или тестирование в контексте.

Такое тестирование можно рассматривать как тестирование системы из двух компонентов, один из которых – реализация, а другой – фиксированная среда передачи. Этот частный случай очень важен, поскольку практическое тестирование часто является асинхронным, особенно, если тестируется система в процессе её реальной работы.

Слайд 42.9. Развитие теории конформности.

Асинхронное тестирование = тестирование в контексте

Рассмотрим более подробно проблему асинхронного тестирования.

При синхронном тестировании тест непосредственно взаимодействует с реализацией. Это моделируется оператором параллельной композиции теста и реализации.

1▶При асинхронном тестировании реализация оказывается погруженной в, так называемый, тестовый контекст. Стандарт ISO определяет тестовый контекст в самом общем виде как отображение, преобразующее одну реализацию в другую. Тест компонуется с результатом такого отображения. Поэтому получается, что непосредственно мы тестируем не реализацию, а реализацию в контексте, то есть другую реализацию.

Понятно, что не любое такое преобразование имеет практический смысл.

2▶Для систем ввода-вывода чаще всего контекст – это две очереди: входная в реализацию очередь стимулов и выходная из реализации очередь реакций.

Очереди бывают разными Они могут быть ограниченного размера или неограниченного, могут допускать или не допускать нарушение порядка поступающих в них элементов, допускать или не допускать потерю элементов или генерацию ложных элементов и т.п. В любом случае очередь можно моделировать соответствующей LTS.

.Система из двух очередей – входной и выходной – это тоже LTS, которая получается как результат композиции двух LTS-очередей. Также часто рассматриваются несколько входных или выходных очередей. Реализация в контексте – это результат композиции LTS-реализации и LTS, моделирующей эти очереди.

3▶В общем случае мы можем говорить не о двух или нескольких очередях, а о некоторой среде передачи общего вида между реализацией и окружением.

Такая среда также моделируется LTS, которая компонуется с реализацией. Результат такой композиции это и есть реализация в контексте. Тест, в свою очередь, компонуется с этой парой.

С асинхронным тестированием связаны две проблемы.

4▶Первая проблема называется *вседозволенность*, по-английски, *permissiveness*. Она заключается в том, что некоторые ошибки, которые обнаруживаются при синхронном тестировании, не могут быть обнаружены при асинхронном тестировании.

В общем-то, вседозволенность – это естественное следствие асинхронности тестирования. Тест не имеет такого непосредственного контакта с реализацией, как при синхронном тестировании, поэтому у него меньше возможностей обнаружить ошибку.

5▶ Другая проблема асинхронного тестирования не столь очевидна и не столь терпима. Это проблема *несохранения соответствия*, по-английски, *non preservation of conformance*. Асинхронное тестирование может обнаружить ошибку, которая не обнаруживается при синхронном тестировании.

Такого рода ошибки следует считать ложными, поскольку именно синхронное тестирование, как наиболее приближенное к реализации, является основным. Наличие ложных ошибок свидетельствует о том, что что-то не так с нашим моделированием конформности на основе композиции LTS.

6▶Предлагаемое решение проблемы несохранения конформности заключается в специальном преобразовании спецификации, которое сохраняло бы класс конформных реализаций и не сужало класс безопасно-тестируемых реализаций. При этом, если реализация конформна исходной спецификации, то её

композиция со средой была бы конформна композиции преобразованной спецификации с той же средой.

Слайд 43.9. Развитие теории конформности.

Асинхронное тестирование = тестирование в контексте

В качестве примера и наиболее широко распространённого частного случая можно рассмотреть среду, состоящую из двух неограниченных очередей: очередь стимулов и очередь реакций.

Мы будем рассматривать ***ioco***-семантику.

1▶Вот у нас есть такая спецификация.

Я напомню, что в нотации CCS стимулы обозначаются префиксом «?», а реакции – префиксом «!».

Реакция `a` может быть выдана с самого начала, а реакция `b` – после приёма стимула `x`.

2▶ А вот реализация I1. В ней есть ошибка, легко обнаруживаемая при синхронном тестировании: после приёма стимула `x` выдаётся не реакция `b`, а реакция `a`.

Посмотрим, что будет происходить при асинхронном тестировании. Если тест начинает с ожидания реакций, то он получит реакцию `!a`, никакой ошибки мы не обнаружим.

Если тест начинается с передачи стимула `?x`, то этот стимул попадает в очередь стимулов. И реализация, и спецификация имеют право сначала выдать реакцию `!a` в очередь реакций. Поэтому, если тест после передачи стимула начнёт ждать реакций, то он в обоих случаях может получить реакцию `!a`. Для реализации с очередью есть только один вариант: трасса `?x!a`. Но спецификация с очередью предлагает на выбор два варианта: `?x!b` или `?x!a`: всё зависит от того, что сначала делается; выдача реакции a или приём стимула x и после него выдача реакции b.

3▶ Поэтому никакой ошибки обнаружено не будет.

Я уже говорил, что вседозволенность – это естественное следствие асинхронности тестирования. Тест не имеет такого непосредственного контакта с реализацией, как при синхронном

тестировании, поэтому у него меньше возможностей обнаружить ошибку.

4▶А вот другая реализация I2.

Здесь определён дополнительный приём стимула z и далее выдача реакции c.

Рассмотрим тест, который выдаёт два стимула x и z, а потом ждёт реакций. С учётом буферизации стимулов и реакций в очередях среды передачи, спецификация может выдать реакцию a или b.

Реализация может выдать реакцию a, b или c.

Если реализация выдаст реакцию c, будет обнаружена ошибка.

5▶Но с точки зрения отношения ***ioco*** это вовсе не ошибка. Ведь после стимула x в состоянии s2 спецификации стимул z не определён, поэтому спецификация не регламентирует поведение реализации после приёма этого стимула.

Эта ложная ошибка, конечно, не обнаруживается при синхронном тестировании. В этом случае тест не выдаёт стимул z сразу после выдачи стимулы x, в только после приёма реакции b.

Слайд 44.9. Развитие теории конформности.

Монотонное преобразование

Несохранение конформности при асинхронном тестировании – особый случай общей проблемы несохранения конформности при композиции.

1▶ Решение заключается в специальном монотонном преобразовании спецификации.

Это преобразование, во-первых, ничего не портит:

Сохраняется класс конформных реализаций,

Сохраняется класс безопасно-тестируемых реализаций.

Для такой, монотонно преобразованной, спецификации конформность сохраняется при композиции. В том числе для особого случая асинхронного тестирования, когда среда не меняется.

2▶Но здесь возникает одно «НО». Это годится только для семантик без **Q**-кнопок.

Как быть?

Слайд 45.9. Развитие теории конформности.

Пополнение как первый шаг к монотонному преобразованию

И тут нам на помощь приходит пополнение, о котором мы уже говорили. Оно так преобразует спецификацию, что её можно рассматривать как в R/Q-семантике, так и в R объединённое с Q семантике. Само пополнение не меняет класс конформных реализаций и не сужает класс безопасно-тестируемых реализаций.

Слайд 46.9. Развитие теории конформности.

Пополнение + монотонное преобразование

Итак, мы сначала делаем пополнение, а потом монотонное преобразование. Класс конформных реализаций не меняется, а класс безопасно-тестируемых реализаций не сужается. И получается общее решение проблемы монотонности.

Слайд 47.9. Развитие теории конформности.

Проблема монотонности

Здесь возникает вопрос: в какой семантике следует рассматривать композицию монотонно преобразованных спецификаций?

Дело в том, что разные компоненты могут быть определены в разных алфавитах и разных семантиках.

По счастью, этот вопрос снимается тем, что композиция монотонно преобразованных спецификаций оказывается самой сильной согласованной спецификацией *в любой* семантике.

Естественно, для одного и того же композиционного алфавита, однозначно определяемого алфавитами компонентов и не зависящего от семантик компонентов при тех же алфавитах компонентов.

1►Монотонное преобразование позволяет решить две основные задачи:

1) верификация декомпозиции системных требований, то есть верификация согласованности имеющейся спецификации системы со спецификациями компонентов, и

2) при отсутствии спецификации системы её генерация по спецификациям компонентов.

Это позволяет генерировать системные тесты даже при отсутствии явно заданной спецификации системы.

Слайд 48.9. Развитие теории конформности.

Фи-модель как средний уровень абстракции

Я не буду здесь рассказывать алгоритм монотонного преобразования. Он довольно сложный, а главное опирается на специальную теорию, для которой у нас здесь нет времени.

Обо всём этом можно прочитать вот здесь:

Теория конформности (функциональное тестирование программных систем на основе формальных моделей). LAP LAMBERT Academic Publishing, Saarbrucken, Germany, 2011, ISBN 978-3-8454-1747-9, 428 стр. (содержание книги доступно по адресу:

http://www.ispras.ru/~RedVerst/RedVerst/Publications/TR-01-2007.pdf)

1▶Хочу только сказать, что эта теория основана на новом виде трасс. Мы назвали их фи-трассами. Соответствующая фи-модель занимает по уровню абстракции промежуточное место между моделью трасс наблюдений и LTS-моделью.

2▶Во-первых трассовая модель однозначно определятся фи-моделью.

Во-вторых, для фи-трасс, как и для LTS и в отличие от трасс наблюдений можно определить оператор параллельной композиции.

При этом фи-модель композиции LTS равна композиции фи-моделей этих LTS. Это самое важное свойство этой фи-модели.

Слайд 49.9. Развитие теории конформности.

Общая структура теории

В этом месте имеет смысл обратить ваше внимание на общую структуру теории.

Всё начинается с формализации тестового эксперимента.

Её основные характеристики:

– всё определяет семантикой взаимодействия, которая описывается формально,

– эта семантика обобщает широкий класс используемых в теории и практике семантик,

– эта семантика имеет универсальный вид и параметризуется наборами R- и Q-кнопок для того, чтобы представлять ту или иную конкретную семантику.

1▶ В первую очередь для такой обобщённой семантики определяются:

– Безопасное тестирование, которое избегает трёх опасностей: ненаблюдаемые отказы, попытки выйти из дивергенции и запрещённое поведение, которое названо разрушением. Эти три вещи препятствуют получению наблюдений в ответ на тестовое воздействие.

– Далее определяется гипотеза о безопасности, описывающая класс реализаций, которые можно безопасно-тестировать на проверку конформности заданной спецификации.

– Наконец, определяется сама безопасная конформность.

2▶Проблема сохранения конформности при композиции решается с помощью монотонного преобразования. Это преобразование сохраняет класс конформных и класс безопасно-тестируемых реализаций. Теперь композиция колнформных реализация конформна композиции монотонно преобразованных спецификаций.

Также следует отметить, что композиция монотонно преобразованных спецификаций является самой сильной спецификацией системы. Она предъявляет к реализации системы самые сильные требования среди всех корректных спецификаций системы. Иначе говоря, она сама конформна любой корректной спецификации системы.

3▶Здесь возникает проблема, о которой я только что говорил: безопасная конформность определена для класса A всех R/Q-

семантик, а монотонное преобразование только для строгого подкласса B семантик, в которых все отказы наблюдаемы.

4▶Эта проблема решается с помощью пополнения. Оно тоже сохраняет класс конформных реализаций и, хотя и не сохраняет, но не сужает класс безопасно-тестируемых реализаций.

Пополненная спецификация в этом смысле эквивалентна исходной спецификации в R/Q-семантике и эквивалентна сама себе в R∪Q/∅-семантике.

Напомню, что, кроме этого, пополнение решает проблему удаления из спецификации неконформных трасс и проблему нерефлексивности конформности.

5▶Важно также обратить внимание, что все эти проблемы решаются\ алгоритмически.

По спецификации автоматически генерируется полный набор тестов для проверки безопасной конформности.

Пополнение и монотонное преобразование также выполняются алгоритмически.

Правда, для пополнения мы уже разработали улучшенный алгоритм, который для конечной спецификации в конечной семантике с ограниченным отношением safe by строит конечное пополнение. А вот для монотонного преобразования такоё алгоритм имеется только в самом общем виде, и требуется некоторое дополнительное исследование, чтобы получить из него алгоритм, более пригодный на практике. Хотя уже ясно. Что это можно сделать.

Слайд 50. 9. Развитие теории конформности.

Приоритеты

Теория, которую мы здесь рассказываем, да и вообще все существующие теории конформности работают с системами без приоритетов, когда любое действие, разрешённое оператором и определённое в реализации, может быть недетерминированным образом выбрано на выполнение. В то же время на практике часто

встречаются системы с приоритетами. Вот только несколько примеров, показывающих важность и полезность приоритетов.

Выход из дивергенции. Если нет приоритетов, то при возникновении дивергенции дальнейшее взаимодействие с системой проблематично, поскольку она может бесконечно долго выполнять только внутренние действия. В то же время дивергенция, как бесконечная внутренняя работа, встречается во многих реальных системах, но при возникновении внешнего воздействия система сразу же на него реагирует. Это происходит из-за того, что внешние действия имеют больший приоритет, чем внутренние τ-действия.

Выход из осцилляции. Под осцилляцией понимается бесконечная цепочка выдачи сообщений системой. Для того, чтобы такую цепочку можно было прервать, заставив систему обрабатывать поступающий извне запрос, последний должен иметь больший приоритет, чем выдача сообщений.

Прерывание цепочки действий. Команда «отменить» (cancel) должна останавливать выполнение последовательности действий, инициированной предыдущим запросом, и вызывать цепочку завершающих действий. Если команда «отменить» не имеет наивысшего приоритета, она может быть выполнена уже после того, как вся обработка закончится, то есть, фактически, ничего «не отменяет».

Приоритетная обработка входных воздействий. Если в систему поступает одновременно несколько запросов, то часто требуется их обработка в соответствии с некоторыми приоритетами между ними. Это реализуется в виде очереди запросов с приоритетами или в виде нескольких очередей запросов с приоритетами между очередями. К этому типу приоритетов относится и обработка аппаратных прерываний в операционной системе.

Нам удалось ввести в теорию приоритеты. Конкретно мы ввели приоритеты в LTS-модель и соответствующим образом модифицировали конформность, генерацию тестов. Также удалось определить параллельную композицию систем с приоритетами. Нерешенными остаются проблемы пополнения и монотонного преобразования для систем с приоритетами.

Слайд 51. 9. Развитие теории конформности.

Симуляции

Кроме конформности, основанной только на трассах наблюдений, в литературе рассматриваются разные виды конформностей, основанных на соответствии состояний реализации и спецификации. Такие конформности называются симуляциями. Симуляция требует, чтобы правильным было не только наблюдаемое внешнее поведение реализации, но и изменение ее состояний. Все рассматриваемые в литературе симуляции либо не учитывают безопасности тестирования, предполагая отсутствие дивергенции и ненаблюдаемых отказов, либо предполагают возможность прямого наблюдения дивергенции и всех отказов. Также они не учитывают возможность разрушения.

Разумеется, тестирование симуляции возможно только в том случае, когда мы можем опрашивать текущее состояние реализации, то есть это всегда тестирование с открытым состоянием.

Нам удалось определить слабую симуляцию с учетом отказов (как наблюдаемых, так и ненаблюдаемых), дивергенции и разрушения. Мы назвали ее безопасной симуляцией. Установлена связь безопасной симуляции с трассовой конформностью ***saco*.**

Теоретически исследовано полное тестирование безопасной симуляции. В отличие от трассовой конформности полное тестирование не всегда возможно. Найдено достаточное условие полноты тестирования и предложен общий алгоритм тестирования безопасной симуляции.

Для практического использования предложена модификация общего алгоритма, которая аналогична алгоритму тестирования с открытым состоянием, о котором я рассказывал, но только верификация выполняется не по ходу исследования реализация, а после завершения такого исследования.

Нерешенными остаются проблемы пополнения и монотонного преобразования для безопасной симуляции.

Слайд 52. 9. Развитие теории конформности.

Тестирование с преобразованием семантик

В теории конформности обычно предполагается, что реализация и спецификация заданы в одной семантике. Однако на практике часто

требуется некоторое преобразование спецификационных тестовых воздействий в реализационные тестовые воздействия и обратное преобразование реализационных наблюдений в спецификационные наблюдения. Программа, осуществляющая эти преобразования, называется медиатором. По сути, это означает, что реализация может быть задана в другой семантике взаимодействия, и медиатор осуществляет преобразование семантик.

В простейшем случае различие семантик только в разных способах представления одних и тех же действий в реализации и спецификации. Достаточно взаимно-однозначного преобразования алфавитов реализации и спецификации. В общем случае такого преобразования недостаточно, поскольку спецификация, как правило, определяется на более высоком уровне абстракции. Медиатор может оказаться довольно сложной программой, осуществляющей медиативные функции между реализацией в одной семантике и тестом, генерируемым по спецификации в другой семантике.

Кроме тестовых воздействий и наблюдений, медиатор может преобразовывать также и состояния: из реализационного в спецификационное. Это означает, что на множестве состояний реализации предполагается заданным отношение эквивалентности, и состояние спецификации соответствует классу эквивалентных состояний реализации. Это соответствие является не предусловием тестирования, а частью проверяемого условия.

Предложена формализация устройства теста и медиатора при тестировании с преобразованием семантик. Соответствующим образом модифицированы гипотеза о безопасности и конформность ***saco***. Дополнительная модификация предложена для медиатора, осуществляющего преобразование состояний.

Для практического использования модифицирован алгоритм конечного полного тестирования с открытым состоянием, в том числе и для случая преобразования состояний в медиаторе.

Не до конца исследованы проблемы пополнения и монотонного преобразования для этого случая.