Обзор существующих подходов

Introduction

Существующие техники можно разбить на 3 категории гарантий покрытия:

1. Никаких гарантий покрытия: Эвристические техники, основанные на философии использования эвристик для достижения чередований, которые наиболее часто содержат баги и тестирование их как можно больше (с ограничениями по времени и пространству). Особенно успешны в нахождении ошибок, но не могут предоставить какую-либо полезную информацию тестировщику о том, что было упущено в тестировании, то есть не могут предоставить никаких гарантий покрытия
2. Гарантии покрытия над пространством чередований: Техники поиска приоритетов получают более ориентированный на покрытие подход, чем техники категории 1 за счет использования идей вроде ограничения упреждения, ограничения контекста, ограничения глубины и ограничения задержек для поиска с приоритетами всех прерываний программы. Эти приоритеты позволяют нам дать количественную оценку какая часть пространства чередований протестирована, свойства, которое техники категории 1 не дают. Техники поиска приоритетов очень полезны в поиске багов. Например, Chess - отличный инструмент, основанный на ограничении контекста поиска приоритизации. Тем не менее, эти техники все основаны на фиксированых (предопределенных) множествах вводом и следовательно, если баг не может быть открыт на данном множестве вводов, он будет упущен. Более того гарантии покрытия даются только нам пространством программных чередований.
3. Гарантии покрытия над пространствами чередований и входных данных (Подозреваю, что это можно даже не смотреть, так как сейчас не учитываем входные данные)

Грубо говоря, все техники применяют отношение частичного порядка. То есть, в каждом алгоритме есть последовательность операций, которая должна произойти раньше другой. При этом, в разные потоки не имеют между собой никаких связей. Во всех случаях мы ищем паттерны вида "чтение и запись в одну переменную" в разных потоках и моделируем расписание таким образом, чтобы эти паттерны были в разных порядках. Если ничего плохого не произошло, то значит всё нормально.
Притом, 1 категория просто получает данные и алгоритм, который нужно к ним применить, находит баг - всё плохо, не находит, то считаем программу корректной. Очевидно, что данный подход не дает гарантии, что на других данных всё не сломается (вдруг есть к примеру, какой-либо if, который всё разрушит).
Методы 2 категории, же действуют иным образом. Они ищут все паттерны. И генерируют множество перестановок. Очевидно, что на каких-то данных расписание будет отличаться от запланированного, тогда даем этому расписанию другие данные чтобы всё было хорошо. При таком подходе, нужно учитывать, что при новых входных данных, нужно задать новое множество линеаризуемых результатов (например, так работает CHESS).

Представители 1 категории.

F. Chen, T. Serbanuta, and G. Roşu. JPredictor: A Predictive Runtime Analysis Tool for Java. In ICSE, pages 221–230,2008.

http://fsl.cs.illinois.edu/index.php/JPredictor

Выделяют 2 вида зависимостей

- Зависимость по управлению - событие Б зависит по управлению от А, если оно предшествует А
- Зависимость по данным - событие Б зависит по данным от А, если изменение состояния А может изменить состояние Б

Используется подход нарезанной трассы. Такой подход предполагает проверку какого-то определенного поля объекта, что существенно сокращает количество возможных перестановок.
На вход имеем некоторый алгоритм. Запускаем его и строит трассу.
Сперва записываем:

1. начало метода
2. true-переход условных операторов
3. доступ к переменным
4. Вызов методов (притом учитываем какая именно реализация вызывается)

Строим законченную трассу на уровне байткода, то есть добавляем в нынешнюю трассу:

1. Точки возврата из метода
2. Состояния до прыжков
3. Имена полей к которым обращаются объекты
4. Начала потоков
5. События снятия/условия блокировок

Режем трассу для каждого определенного свойства, с учетом зависимостей
Как режем трассу:

- Проходим по нашей трассе в обратном направлении, берем в нарезанную трассу события, которые непосредственно относятся к свойству P
- Берем в нарезанную трассу события, от которых зависит наше свойство P.

Для чего строим трассу? Потому что каждый поток делаем линейным. То есть, нам плевать, что где-то есть if конструкция, мы просто верим, что если изначально получилось что-то вроде if then, то при како-то перестановке на каких-либо данных получим аналогичное (скорее моя догадка, в статьях ничего не сказано по этому поводу).

На основе векторных часов записываем всё это в множество, в котором учитываем зависимости и атомарные блоки.

Строим всевозможные трассы и смотрим есть ли где-то гонки или неделимость атомарных блоков.
На самом деле jPredictor не строит всевозможные трассы.

Гонки проверяются по условию, чтобы в двух потоках были операции чтения/запись не защищенные замком. Если такое есть - гонка.

Атомарность проверяются попарно, сопоставляя с 11 паттернами.

A. Farzan, P. Madhusudan, N. Razavi, and F. Sorrentino. Predicting Null-Pointer Dereferences in Concurrent Programs. In FSE, pages 47:1–47:11, 2012.

На вход получает программу и входной вектор. Запускает и ищет пары событий (w,r) - null WR пара:

- w - запись null в переменную x
- r - чтение непустой переменной x
- w,r независимы, то есть для них есть пересстановка

Для каждой такой пары находим множество всех выполнений и используют SMT solvers (Что это?) для поиска конкретного расписания, которое приводит к ошибке.

Запускаем найденное решение.

PENELOPE: Weaving Threads to Expose Atomicity Violations

https://github.com/sorfrancesco/Penelope

Считаем, что нам известны результаты последовательного выполнения. На вход получаем какой-то алгоритм и входные данные. Начинаем работу. Работа разбита на 3 этапа:

1. Мониторинг. Выполняем программу на входных данных, наблюдаем за ходом выполнения. Записываем операции чтения/записи в (потенциально) общие переменные, получение/снятие блокировок, создание потоков и барьеры.
2. Предсказание. Конструируем иные запуски, чтобы они содержали как минимум один из подозрительных шаблонов. Подозрительными считаем такие шаблоны:
    a. RWR
    b. WRW
    c. RWW
    d. WWR
    e. WWW

3. Изменение расписания. ВЫполняем программу на 1 процессоре, путем переплетения потоков на 1 процессоре. Если успешно выполняется - всё ок, если нет - рапортуем о баге. Если не можем составить расписание для запуска, то просто отбрасываем его.

CTrigger: Exposing Atomicity Violation Bugs from Their Hiding Places

Рассматривают все ошибки в виде предшествующих, настоящих и удаленных доступов. Предшествующий и настоящий - на одном потоке, а удаленный на другом, который потенциально может вклиниться между ними.
Вводят понятие окон:

1. Локальное - между предшествующим и настоящим доступом (с учетом границ синхронизации)
2. Удаленное - дистанция удаленного доступа до начала предшествующего или конца удаленного доступа

Рассматривают все такие потенциальные ошибки в многопоточной среде (ставят где-то задержки и прочее) с учетом метрики.

Представители 2 категории.

Microsoft Chess.

Для всех операций делает обертки, которые находятся с помощью профайлера. Эти обертки помогает планировщику планировать.

Сам планировщик работает в 3 этапа

1. воспроизведение - У планировщика есть файл расписания. На первом прогоне он пуст (видимо 1 итерацию начинаем с записи). Этот файл содержит частичный график, который был сгенерирован на предыдущих итерации (его генерирует этап запись).
2. запись - В каждой точке (интересных операциях) отмечает какие потоки параллельно могут работать. НА операциях переключения потоков выбираем поток основываясь на приоритетах.
3. поиск - Собственно генерирует однопроцесорное расписание на каждой точке.

Справедливый планировщик дает меньший приоритет потокам, которые передали управление.

Всего вставляют 2 прерывания на итерацию. Умеют сохранять состояния, что позволяет не воспроизводить заново всю итерацию, а продолжить с какого-то места.

Delay-Bounding

Просто формальное описание планировщиков (очень формальное). Приведены 2 примера, но на практике по моему применяется round-robin. Никакой конкретики и реализации. Рассматриваются только чужие реализации.

TBD