Многопоточное тестирование программных интерфейсов

В. С. Мутилин
mutilin@ispras.ru

Аннотация. В статье описывается новый метод функционального тестирования параллельных программ, предоставляющих программный интерфейс, методы (процедуры) которого можно вызывать из нескольких потоков одновременно. Этот метод, названный Sapsan, позволяет проверять одно из распространенных требований к таким программам — требование сериализуемости интерфейса, заключающееся в том, что в любом состоянии программы результат параллельного выполнения методов интерфейса в нескольких потоках эквивалентен некоторому последовательному выполнению этих же методов. Это требование является формализацией широко используемого понятия thread-safety.

1. Введение

В последнее время многопоточное программирование получило широкое распространение. Для повышения производительности программ их выполнение все чаще организуют в виде нескольких потоков, которые работают параллельно. Но разработка многопоточных программ намного сложнее последовательных. Это связано с тем, что порядок, в котором будут выполнены инструкции разных потоков, заранее непредсказуем и разработчик должен предусмотреть корректную работу программы при всех возможных чередованиях инструкций.

В данной статье мы рассматриваем программы, предоставляющие программный интерфейс, через который с ними взаимодействуют другие программы. Интерфейс состоит из отдельных методов (процедур), описанных на языке программирования, которые можно вызывать с различными значениями параметров. Кроме того, методы интерфейса могут быть вызваны из различных потоков одновременно.

Одно из распространенных требований к многопоточным программам — требование *сериализуемости*. Оно, по сути, является формализацией широко используемого понятия *thread-safety*. Пусть в некотором состоянии программы выполняется несколько потоков, вызывающих по одному интерфейсному методу. Будем называть выполнение потоков *сериализуемым*, если результат

выполнения эквивалентен некоторому последовательному выполнению этих же потоков в том же исходном состоянии.

На практике требования сериализуемости широко распространены. Отметим два распространенных класса таких систем: интерфейсы промежуточного уровня клиент-серверных приложений и библиотеки, предназначенные для многопоточного использования. Обычно предполагается, что несколько клиентов могут одновременно работать в рамках клиент-серверного приложения, не замечая присутствия друг друга. Если два клиента совершают какие-то операции, то результат должен быть такой же, как если бы они были выполнены последовательно. Данный класс систем чрезвычайно широк, так как практически все современные системы, предоставляющие сервисы, могут обслуживать несколько клиентов одновременно.

В библиотеках, предназначенных для многопоточного использования, такое требование выдвигается по умолчанию. Функциональные требования описываются для каждого метода в отдельности, но кроме того требуется, чтобы библиотечные методы было безопасно вызывать из нескольких потоков (*thread-safe*). В наших терминах это и есть требование сериализуемости.

Одним из наиболее распространенных способов проверки свойств параллельных программ является метод проверки моделей (*model checking*), осуществляющий поиск чередований инструкций в параллельных потоках. В последнее время инструменты проверки моделей сделали существенный шаг вперед. Стало возможным проверять свойства для программ, написанных на широко распространенных языках программирования, а не на простых модельных языках. Так инструмент Java PathFinder [12], способен проверять свойства программ на языке программирования Java, а инструмент VeriSoft [8] предназначен для проверки программ на языке C. Однако попытки применения этих инструментов для решения задачи проверки сериализуемости программы, сталкиваются с рядом сложностей.

Во-первых, чтобы запустить эти инструменты требуется подготовить окружение, т.е. задать набор потоков, вызывающих методы интерфейса с некоторыми значениями параметров, и начальное состояние. Но так как требование сериализуемости формулируется для неограниченного числа потоков, то проверив сериализуемость на определенных конечных их наборах, мы не можем быть уверенны, что оно выполнено для программы в целом. Во-вторых, поиск всех возможных выполнений программы занимает большое время. В зависимости от количества потоков самым современным инструментам требуется для этого от нескольких минут до нескольких часов.

Метод Sapsan для проверки сериализуемости, описанный в данной работе, в принципе может быть применен для широкого класса программ на разных языках программирования. Однако в статье детали метода рассматриваются только для программ, написанных на языке программирования Java.

2. Формальная постановка задачи

Выполнение программы — это последовательность пар $(t_{i_1}, b_{j_1}),...,(t_{i_n}, b_{j_n})$, где t_{i_k} — выполняемый поток, b_{j_k} — инструкция потока. Будем называть $e_k = (t_{i_k}, b_{j_k})$ *единицей выполнения*.

Единица выполнения переводит систему из одного состояния в другое, что мы обозначаем как $s \xrightarrow{(t_i, b_i)} s'$. Тот факт, что выполнение $e_1,...,e_n$ переводит систему из состояния s в состояние s', обозначаем $s \xrightarrow{e_1,...,e_n} s'$.

Если в программе возможна последовательность инструкций представленная в выполнении, то такое выполнение называем *достижимым* в этой программе. Иначе, такое выполнение будем называть *недостижимым* в программе.

Путь в потоке — это последовательность инструкций $b_1,...,b_n$, выполняемых в одном потоке. Из любого выполнения $e_1,...,e_n$ можно выделить путь для потока t', выбрав из единиц выполнения $e_k = (t_k, b_k)$, в которых $t_k = t'$, инструкции b_k в том же порядке. И обратно, если у нас есть пути $p_1,..., p_k$ в потоках $t_1,...,t_k$, то мы можем составить выполнение, выбрав некоторую последовательность единиц выполнения $e_i = (t_i, b_i)$, где b_i из p_i так, что инструкции для каждого потока t_i расположены в том же порядке, что и для пути p_i. Отметим, что выполнение, составленное из путей в потоках, может быть недостижимым в программе.

Несколько последовательных инструкций в пути потока может принадлежать *взаимоисключающему интервалу* (например, блоки синхронизации и синхронизированные методы, реализованные как вход и выход из монитора). К интервалу приписан объект, по которому происходит взаимоисключение. Инструкции, принадлежащие двум взаимоисключающим интервалам с одинаковыми приписанными объектами, в любом достижимом выполнении не пересекаются, т.е. либо сначала выполняются все инструкции первого интервала, а потом второго, либо наоборот.

2.1. Сериализуемость выполнения

Определение 1. Пусть $s \xrightarrow{e_1,...,e_n} s'$ *и выполнение* $e_1,...,e_n$ *содержит потоки* $t_1,...,t_m$ *и пути* $b_i^1,...,b_i^{r_i}$ *в каждом потоке* t_i. *Выполнение* $e_1,...,e_n$ *сериализуемо, если существует выполнение* $\hat{f} = (t_{i_1}, b_{i_1}^1),...,(t_{i_1}, b_{i_1}^{r_{i_1}}),...,(t_{i_m}, b_{i_m}^1),...,(t_{i_m}, b_{i_m}^{r_{i_m}})$ *такое, что* $t_{i_1},...,t_{i_m}$

составляют все множество потоков и выполнение $\hat{f}$ *переводит систему в тоже состояние* s', *т.е.* $s \xrightarrow{\hat{f}} s'$.

В данной работе мы пытаемся решить следующую задачу: для заданной программы установить, сериализуемы ли все достижимые выполнения этой программы.

3. Достаточные условия сериализуемости выполнений

Понятие независимости инструкций широко используется при верификации программ. Одно из наиболее распространенных применений — сокращение пространства поиска достижимых состояний или выполнений [6,7,10].

Интуитивно, две инструкции, выполняющиеся в разных потоках, независимы в некотором состоянии, если любое их выполнение этом состоянии приводит к одному и тому же результату.

Определение 2. Единицы выполнения e_i, e_j *независимы в состоянии программы* s, *если:*

1. *В состоянии* s' *выполнения* e_i, e_j *и* e_j, e_i *достижимы или недостижимы одновременно;*
2. *Если* $s \xrightarrow{e_i, e_j} s_1$ *и* $s \xrightarrow{e_j, e_i} s_2$, *то* $s_1 = s_2$.

Иначе единицы выполнения называют зависимыми.

Если единицы выполнения зависимы (независимы), то говорят, что зависимы (независимы) инструкции в этих выполнениях.

Рассмотрим следующее условие.

Условие 1. Пусть $a_1,...,a_n$ и $b_1,...,b_m$ — пути в двух потоках. Условие выполнено, если существует единственная пара (a_{i_0}, b_{i_0}) зависимых инструкций. Все остальные пары (a_i, b_j), $(i, j) \neq (i_0, j_0)$ либо независимы во всех состояниях, либо они зависимы и выполнено:

1. a_{i_0}, a_i — принадлежат одному взаимоисключающему интервалу с obj_1;
2. b_{j_0}, b_j — принадлежат одному взаимоисключающему интервалу с obj_2;
3. $obj_1 = obj_2$.

Утверждение 1. Пусть $p_1, p_2, ..., p_k$ *— пути в потоках. Если для любой из пар путей* (p_i, p_j) *выполнено условие 1, то любое достижимое выполнение, составленное из путей* $p_1, p_2, ..., p_k$ *, сериализуемо.*

Для доказательства данного утверждения достаточно заметить, что в достижимом выполнении независимые инструкции можно переставлять местами, при этом результирующее состояние остается неизменным. Дополнительно отметим, что кроме одной пары, все остальные зависимые инструкции принадлежат взаимоисключающим интервалам с одинаковыми объектами и, следовательно, в любом достижимом выполнении появляются либо сначала зависимые инструкции одного потока, затем второго, либо наоборот. Поэтому перестановкой остальных независимых инструкций получим выполнение, в котором потоки выполняются последовательно. ■

Будем говорить, что путь в потоке *достижим*, если его можно выделить из какого либо достижимого выполнения. Будем говорить, что множество путей в потоках $p_1, p_2, ..., p_k$ *исчерпывающее*, если оно включает все достижимые пути в потоках, т.е. является аппроксимацией сверху множества всех достижимых путей в потоках. Отметим, что наряду с достижимыми путями оно может включать и недостижимые пути.

Следствие. Заметим, что если в утверждении 1 $p_1, p_2, ..., p_k$ *— исчерпывающее множество путей в потоках, то любое достижимое выполнение программы сериализуемо.*

3.1. Дисциплина синхронизации доступа к разделяемым переменным

В данной работе мы считаем, что программа удовлетворят дисциплине синхронизации доступа к разделяемым переменным. Суть этой дисциплины заключается в том, что доступ к любой переменной из разных потоков должен происходить из взаимоисключающих интервалов с хотя бы одним совпадающим объектом. Проверка следования данной дисциплине может осуществляться одним из известных алгоритмов [5,11], например алгоритмом ERASER [11]. Отметим, что алгоритм ERASER может быть применен совместно с предложенным в данной работе методом проверки сериализуемости Sapsan.

Следование данной дисциплине позволяет устанавливать независимость инструкций на основе анализа принадлежности взаимоисключающим интервалам и сравнением объектов.

Утверждение 2. Две инструкции, которые не принадлежат взаимоисключающим интервалам с одинаковыми объектами, независимы в любых состояниях.

Утверждение следует из того, что инструкции осуществляют доступ к разным переменным, согласно предположениям о подчинении дисциплине синхронизации. ■

Вместо независимости инструкций мы можем рассматривать независимость блоков, состоящих из попарно независимых инструкций. Для блоков достаточные условия формулируются аналогично инструкциям.

4. Алгоритмы поиска достижимых выполнений

В данном разделе мы рассмотрим известные алгоритмы поиска достижимых выполнений. На входе у алгоритмов задана *конфигурация выполнения*, включающая начальное состояние и набор потоков с *воздействиями* на систему. В нашей задаче воздействиями являются вызовы интерфейсных методов с некоторыми значениями параметров.

Конфигурация является окружением для программы. Программу с окружением можно запустить в результате получим какое-то *достижимое выполнение*. В зависимости от чередования инструкций могут получаться разные достижимые выполнения. Все множество достижимых выполнений в заданной конфигурации — это выполнения, полученные при всех возможных чередованиях инструкций потоков.

Каждое выполнение имеет *результат*, представляющий собой результирующее состояние. Особо отметим, что в данной работе мы не выделяем возвращаемые значения методов как отдельную часть результата. Мы считаем, что они включаются в состояние, хотя в практических реализациях их удобнее выделять как отдельные сущности. Кроме того, мы не рассматриваем другие возможные реакции системы: обратные вызовы и сообщения.

Достижимых выполнений, как правило, очень много и не все из них требуются для установления корректности. Достаточно ограничиться полным множеством выполнений. Множество выполнений называется *полным*, если его множество результатов совпадает с множеством результатов всех достижимых выполнений. Если полное множество выполнений получается меньше чем множество всех достижимых выполнений, то говорят о сокращении множества выполнений.

Задача алгоритма поиска — построить полное множество выполнений.

В данной работе мы используем алгоритмы, описанные в работе Годфруа [8]. Эти алгоритмы существенно используют понятие независимости для сокращения множества выполнений. Различают два класса алгоритмов поиска:

1. Алгоритмы с сохранением состояния;
2. Алгоритмы без сохранения состояния.

Для того чтобы осуществлять поиск выполнений, необходимо иметь возможность возвращаться в предыдущее состояние. В существующих алгоритмах это достигается двумя способами. В первом способе, возврат осуществляется восстановлением ранее сохраненного состояния. Он используется в алгоритмах с сохранением состояния. Второй способ, использующийся в алгоритмах без сохранения состояния, заключается в сбросе системы в начальное состояние и перевыполнении до нужного состояния. Для этого требуется наличие соответствующего механизма сброса.

Как правило, имея возможность хранить состояния, мы имеем возможность сравнивать их, что может быть использовано для сокращения перебора. Кроме того, сравнение состояний дает возможность обнаруживать циклы. Несмотря на эти преимущества, в данной работе выбран второй способ, потому как хранение состояний, которое потребовалось бы в первом способе обладает рядом недостатков:

1. Требует значительных объемов памяти для хранения состояний
2. Требует моделирования состояний недоступных для чтения.

Известные алгоритмы, использующие первый способ, реализованы в виде специальной виртуальной машины [4,9,12], которая хранит пройденные состояния и позволяет управлять последовательностью выполнения инструкций. Эти машины с легкостью справляются со всеми инструкциями Java кода. Однако программы на Java, кроме чистого кода содержат вызовы процедур, реализованных на других языках, называемых *внутренними* (native) методами. Для этих методов требуется написать модель, сохраняющую и восстанавливающую состояния. На практике программы с внутренними методами встречаются часто. Практически все стандартные библиотеки содержат внутренние методы. Например, библиотека работы с сетью. Кроме того, программы часто пользуются системами, написанными на других языках, например, базами данных.

5. Метод Sapsan

На вход методу Sapsan подается программа предоставляющая интерфейс, которая должна следовать дисциплине синхронизации доступа к разделяемым переменным, описанной в Разделе 3.1. Вместе с программой должен быть подготовлен тестовый набор, требования к которому мы рассмотрим ниже. Кроме того инструменту Sapsan могут быть предоставлены эвристики для оптимизации поиска. Схема метода показана на Рис. 1.

Метод Sapsan состоит из трех шагов. На первом шаге выполняется инструментация кода программы. До и после элементов синхронизации, необходимых для выделения взаимоисключающих интервалов, вставляются специальные блоки, сохраняющие путь. Так как мы предполагаем, что соблюдается дисциплина синхронизации доступа к разделяемым переменным, то инструментация всех остальных инструкций не требуется. В языке Java

элементами синхронизации являются инструкции входа в монитор, выхода из монитора, вызовы синхронизованных методов и методов wait, notify и notify All.

На втором шаге выполняются тесты. Так как программа инструментирована, то при выполнении автоматически сохраняется пройденный путь для каждого из потоков. К тестовому набору предъявляются следующие требования:

1. Вызовы интерфейсных методов должны быть разделены по потокам, так чтобы в каждом потоке был только один вызов;
2. Запуск тестового набора строит исчерпывающее множество путей в потоках;
3. В тестовом наборе сохраняется информация о состояниях и потоках для каждого достигнутого пути.

Смысл первого требования в том, что каждый путь в потоке должен быть путем для одного интерфейсного метода, и не должен включать инструкции других интерфейсных методов и вспомогательных методов тестовой системы. При выполнении этого условия сериализуемость путей в потоках будет означать сериализуемость интерфейсных методов.

Второе требование необходимо для того чтобы утверждать сериализуемость программы в целом на основе проверки достаточных условий сериализуемости для найденных путей (следствие из утверждения 1). На практике, если тестовый набор не обеспечивает построение исчерпывающего множества, метод Sapsan может быть применен, но гарантировать сериализуемость мы можем только для найденных путей в потоках. Для того чтобы проверить построено ли исчерпывающее множество можно использовать алгоритмы статического анализа (будем показано далее).

Хранение информации для каждого пути необходимо для того чтобы, во-первых, можно было повторно выполнить поток, который привел к данному пути и, во-вторых, вернуться в состояние, в котором выполнялся данный поток. Информация о том, какой поток и в каком состоянии привели к данному пути, составляет основу для выбора состояния и набора потоков в инструменте Sapsan.

На третьем шаге запускается инструмент Sapsan. На входе у него имеется исчерпывающее множество путей в потоках, вместе с сохраненной информацией о том, как они были достигнуты, и эвристики по выбору набора потоков. Инструмент пытается установить сериализуемость для всех найденных пар путей в потоках на основе достаточных условий сериализуемости. Если инструмент выдает ответ «Да», то программа сериализуема. Если ответ «Нет», то программа не сериализуема и инструмент выдает набор потоков, состояние и информацию о выполнении на котором нарушается свойство сериализуемости. Возможен также и третий вариант, когда инструменту не удалось установить сериализуема ли программа. На

выходе в этом случае пары путей, для которых не выполнены достаточные условия. Эти подозрительные пары могут быть проанализированы вручную.

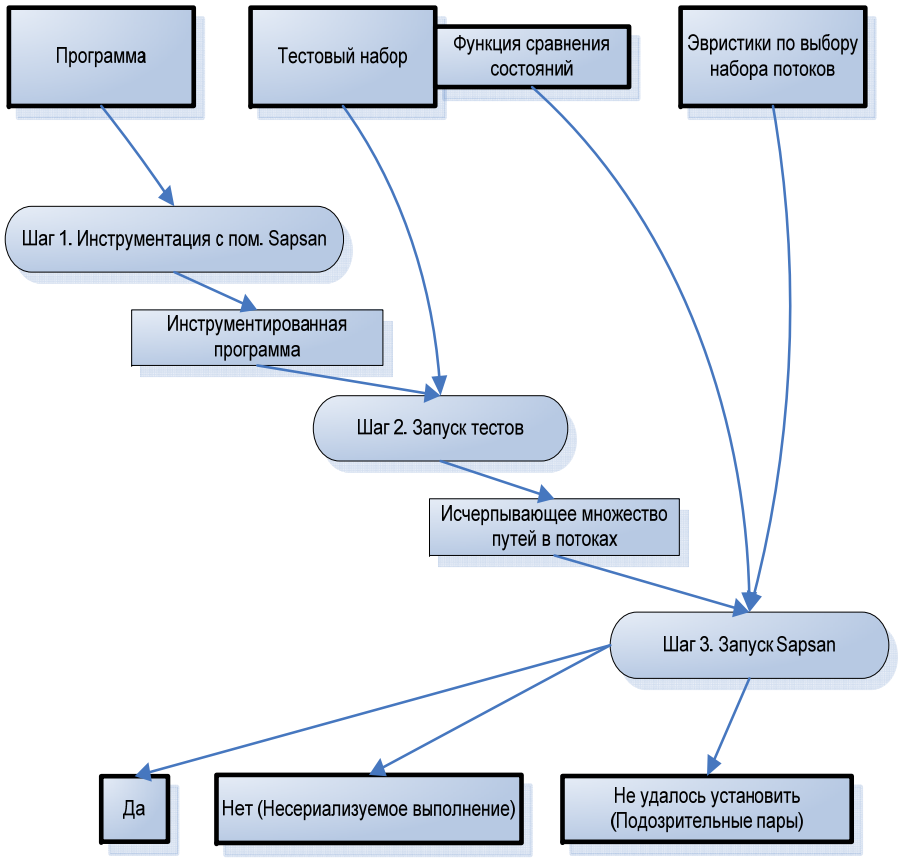

Рис. 1. Схема метода Sapsan.

Теперь рассмотрим основные некоторые моменты метода более детально.

5.1. Построение исчерпывающего множества путей в потоках

Цель первых двух шагов метода — построить исчерпывающее множество путей в потоках. В методе Sapsan основным средством его построение является запуск тестов. Возможны и другие способы вычисления путей в потоках, например, один из наиболее распространенных — *статический анализ*. Статический анализ — это построение модели программы и анализ ее свойств без запуска программы. Он хорошо зарекомендовал себя при анализе

кода одного из методов, но *межпроцедурный анализ* и *ссылочный анализ* обладают существенным недостатком — модели программы содержат большое количество ложных (недостижимых) путей. В объектно-ориентированных программах анализ усложняется наличием полиморфизма и наследования.

Мы выбрали тестирование, потому что оно не вносит ложных путей, строя только те пути, которые есть в достижимых выполнениях.

Но как проверить исчерпываемость построенного с помощью тестирования набора путей, требующуюся в методе Sapsan?

В методе Sapsan предлагается для частичного ответа на данный вопрос использовать статический анализ в рамках одного метода. Если метод тестирования достигает все пути по вызовам в потоке, и статический анализ в рамках одного метода показывает, что все пути достигнуты (были пройдены все инструкции синхронизации), то полученный набор путей исчерпывающий.

5.2. Установление сериализуемости

Цель третьего шага, на котором мы запускаем инструмент Sapsan — установить сериализуемость для всех пар путей в потоках. Схема работы инструмента показана на Рис. 2.

Для найденных путей проверяются достаточные условия сериализуемости. При проверке достаточных условий используется информация о независимости участков кода. Для Java программ по умолчанию зависимыми считаются все взаимоисключающие интервалы с одинаковыми объектами. Все остальные блоки кода считаются независимыми, это свойство следует из условия применения дисциплины синхронизации разделяемых переменных.

Если достаточные условия выполнены, следовательно, мы показали, что методы интерфейса сериализуемы. Иначе возможно три варианта:

1. В системе присутствует путь, на котором нарушается сериализуемость;
2. Наша информация о независимости блоков кода неполна;
3. Невозможно установить сериализуемость, используя достаточные условия.

В первом случае нам необходимо попытаться найти достижимое выполнение не являющееся сериализуемым. Во втором попытаться установить независимость блоков кода.

В инструменте Sapsan в обоих случаях в качестве основы используется *алгоритм поиска* достижимых выполнений без сохранения состояний (см. раздел 4) [8], реализованный в инструменте VeriSoft. Этот алгоритм был оснащен возможностью *поиска по шаблону*, что позволило сократить пространство поиска. Шаблон задает порядок, в котором должны встретиться единицы выполнения в искомом выполнении. Шаблоны строятся на основе пар путей, для которых не выполнено Условие 1 раздела 3. Поиск по шаблону

нацеливается только на выполнения удовлетворяющие шаблону, т.е. выполнения, не подходящие под шаблон, отбрасываются.

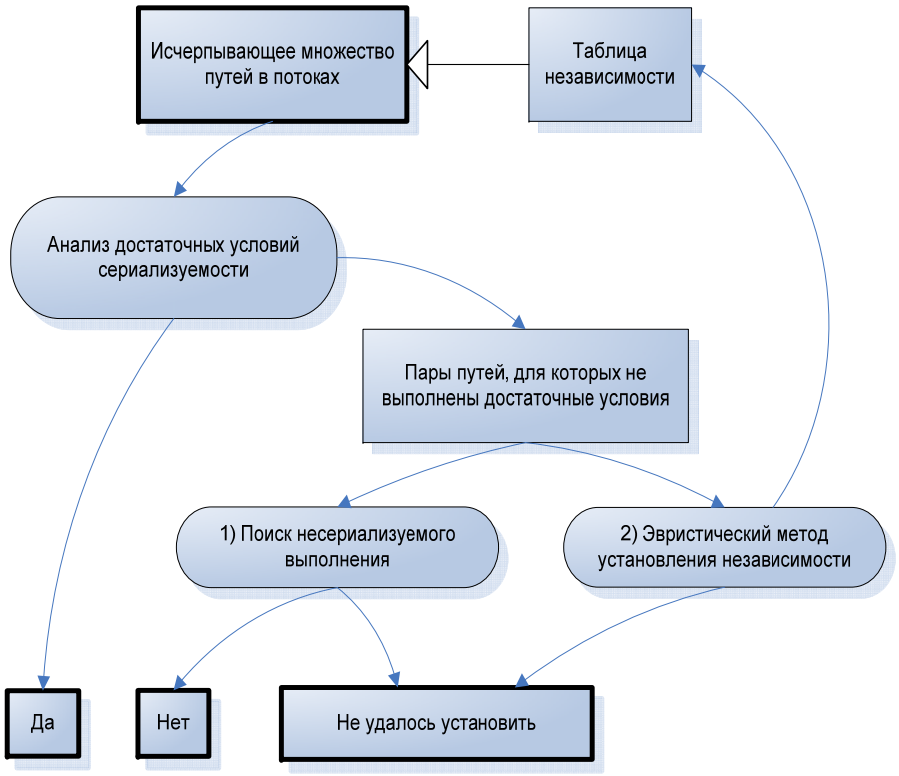

Рис. 2. Установление сериализуемости.

5.2.1. Поиск несериализуемых выполнений

Если для двух путей p_1, p_2 не выполнено Условие 1, то существует четверка блоков u_1,u_2,v_1,v_2 такая, что блоки u_1,u_2 взяты из пути потока p_1, v_1,v_2 взяты из пути потока p_2, u_1,v_1 — зависимы, u_2,v_2 — зависимы, и:

1. либо u_1,u_2 не принадлежат одному взаимоисключающему интервалу,
2. либо v_1,v_2 не принадлежат одному взаимоисключающему интервалу,
3. либо u_1,u_2 и v_1,v_2 принадлежат одним взаимоисключающим интервалам одновременно, но с разными объектами.

Отсюда получаем, что возможно четыре шаблона: u_1,v_1,u_2,v_2, u_1,v_1,v_2,u_2, v_1,u_1,u_2,v_2, v_1,u_1,v_2,u_2.

Поиск несериализуемых путей требует наличия механизма проверки сериализуемости для пути выполнения. В методе Sapsan проверка происходит сравнением результата выполнения с результатами последовательного прогона потоков. Отметим, что возможны и другие способы проверки. Например, если в тестовом наборе уже описана модель требований, обладающая сериализуемостью, то можно ее переиспользовать. Так, например, в технологии UniTESK [1,3], зачатую ее бывает выгоднее переиспользовать.

5.2.2. Эвристический метод установления независимости

Эвристический метод установления независимости требует наличия механизма сравнения результирующих состояний выполнений.

Для того чтобы установить независимость блоков b_1,b_2 в потоках t_1,t_2 соответственно, мы делаем следующее. Осуществляем поиск по шаблону, находим выполнения, в которых встречаются последовательно единицы выполнения $(t_1,b_1),(t_2,b_2)$ и $(t_2,b_2),(t_1,b_1)$ и запоминаем результирующие состояния.

Дальше для всех пар выполнений, которые отличаются перестановкой единиц выполнения, сравниваем результирующие состояния. Если на всех таких выполнениях результаты совпадают, то считаем, что b_1,b_2 независимы, иначе зависимы.

То есть если для любых двух выполнений:

1. $e_1,...,e_k,(t_1,b_1),(t_2,b_2),e_{k+1},...,e_n$ с результатом r_1 и
2. $e_1,...,e_k,(t_2,b_2),(t_1,b_1),e_{k+1},...,e_n$ с результатом r_2

выполнено $r_1=r_2$, то b_1,b_2 независимы.

5.2.3. Эвристики по выбору набора потоков

Везде, где мы осуществляем поиск, требуется задать конфигурацию (см. Раздел 4), т.е. нужно задать потоки и состояние, в котором будет осуществлен поиск. Мы имеем два пути p_1, p_2 для которых не выполнены достаточные условия, и шаблон по которому производиться поиск.

От тестового набора, запускаемого на шаге 2, мы потребовали сохранения для каждого пути информации о состоянии s_i и потоке t_i. Используя эту информацию, мы можем запускать соответствующие потоки и переходить в соответствующие состояния.

В общем случае ответить на вопрос, достижимо данное выполнение, невозможно. Поэтому задача выбора набора потоков и состояния для достижения заданного выполнения неразрешима. Однако, исходя из

сохраненной информации для путей p_1, p_2, мы можем предложить следующие эвристики, перечисленные в порядке приоритетности их применения.

1. Рассматриваем пары потоков, сохраненные для путей p_1, p_2, и пересечение состояний для путей p_1, p_2.
2. То же самое, только берем объединение состояний для путей p_1, p_2.
3. Добавляем другие потоки.
4. Добавляем другие состояния.

На практике мы осуществляем поиск лишь для небольшого числа потоков. В результате экспериментов мы установим, насколько часто удается установить независимость или найти ошибку. Мы надеемся, что ситуация, при которой мы не можем достоверно сказать есть ли ошибка или нет, на практике будет встречаться редко. Отметим, что даже когда этого нельзя достоверно утверждать для всех выполнений, в процессе поиска мы проверяем сериализуемость вплоть до некоторого количества потоков.

6. Результаты применения метода

Для апробации инструмента Sapsan в качестве тестового набора использовались тестовые варианты JUnit. Каждый тестовый вариант *testCase* состоит из одного вызова интерфейсного метода с некоторыми параметрами. Для тестового варианта задан метод *setUp* — инициализации состояния и метод *tearDown* — приведения системы в начальное состояние. Для всех тестовых вариантов задан метод *getState*, вычисляющий текущее состояние системы. Таким образом, запуск тестового варианта состоит из последовательности: *setUp, начало сбора пути в потоке, testCase, конец сбора пути в потоке, сохранить информацию, getState, tearDown*.

На шаге 3 использовались эвристики 1 и 2 (см. 5.2.3). Проверка сериализуемости выполнения происходит сравнением результата выполнения с результатами последовательных выполнений.

Были написаны тесты для реализаций кэша Ehcache [2], который оптимизирует доступ к хранящимся в нем элементам, размещая часто используемых из них в памяти, а остальные сохраняя на диске. Реализация этого кэша составляет примерно 40 тысяч строк кода на Java.

Кэш предоставляется интерфейс, состоящий из трех основных методов:

1. put — положить элемент в кэш;
2. get — считать элемент из кэша;
3. remove — удалить элемент из кэша.

Для каждого из методов было написано по нескольку тестовых вариантов. Проведенные запуски инструмента Sapsan показали следующее:

1. Пути в потоках не зависят от набора потоков и слабо зависят от состояний. Получается, что исчерпывающее множество путей, можно получить при последовательном выполнении потоков в небольшом наборе состояний.
2. Было выявлено два нарушения достаточных условий. Для первого из них было найдено несериализуемое выполнение, а для второго была установлена независимость блоков эвристическим методом. Причем в конфигурациях использовались только пары потоков и пересечение состояний (эвристика 1).

Кроме того, эксперименты показали, что исчерпывающий набор путей может быть получен уже при последовательном выполнении потоков.

7. Заключение

В работе описан метод Sapsan, который позволяет проверять сериализуемость программы. Если результатом применения метода является ответ «Да», то программа гарантированно является сериализуемой. Это отличает данный метод от методов проверки моделей, которые могут гарантируют сериализуемость лишь для конечного числа потоков. Кроме того за счет применения алгоритмов поиска лишь для некоторых пар путей и поиска по шаблону метод значительно выигрывает у них в скорости.

Метод был успешно опробован на достаточно большой реализации кэша. В дальнейшем планируется применить метод для ряда других реализаций.

Литература

[1] В. В. Кулямин, А. К. Петренко, А. С. Косачев, И. Б. Бурдонов. Подход UniTESK к разработке тестов. Программирование, 29:25–43, 2003.

[2] http://ehcache.sourceforge.net.

[3] А. В. Хорошилов. Спецификация и тестирование компонентов с асинхронным интерфейсом. Диссертация на соискание ученой степени кандидата физико-математических наук. 2006.

[4] D. Bruening. Systematic testing of multithreaded Java programs, 1999.

[5] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: a race and transaction-aware Java runtime. In Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation, pp. 245–255, NY, USA, 2007.

[6] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In POPL '05: Proceedings of the 32-nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 110–121, NY, USA, 2005.

[7] P. Godefroid. Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996. Foreword By-Pierre Wolper.

[8] P. Godefroid. Model Checking for Programming Languages using Verisoft. In Symposium on Principles of Programming Languages, pp. 174–186, 1997.

[9] V. S. Mutilin. Concurrent testing of Java components using Java PathFinder. Proceedings of ISoLA 2, 2006.
[10] D. Peled. Combining partial order reductions with on-the-fly model checking. In Formal Methods in System Design, vol. 8, pp. 39–64, 1996.
[11] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. ACM Transactions on Computer Systems, 15(4):391–411, 1997.
[12] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. Automated Software Engg., 10(2):203–232, 2003.