Метод проверки линеаризуемости многопоточных Java программ

Мутилин В.С.
mutilin@ispras.ru

Аннотация. В статье описывается новый метод Sapsan. Он предназначен для функционального тестирования Java программ, предоставляющих программный интерфейс (API), процедуры (операции) которого можно вызывать из нескольких потоков одновременно. Метод Sapsan позволяет проверять одно из распространенных требований к таким программам – требование линеаризуемости, заключающееся в том, что параллельное выполнение операций эквивалентно некоторому последовательному выполнению этих же операций, удовлетворяющему спецификации.

1. Введение

В последнее время многопоточное программирование получило широкое распространение. Разрабатываемые программы с целью ускорения работы все чаще состоят из нескольких потоков, которые выполняют работу 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 работе мы рассматриваем Java программы, предоставляющие программный интерфейс, через который с ними взаимодействуют другие программы. Интерфейс состоит из *операций* (процедур), которые можно *выполнять* (вызывать) с различными значениями *параметров*. Выполнение операции завершается возвратом значения, называемого *результатом*. Кроме того, операции могут быть выполнены в различных потоках одновременно. Выполнение операций в нескольких потоках будем называть *параллельным выполнением*, а выполнение операций в одном потоке – *последовательным*.

Параллельное выполнение *линеаризуемо*, если оно эквивалентно некоторому последовательному выполнению, удовлетворяющему спецификации. Формально понятие линеаризуемости будет определено в разделе 2. Можно видеть, что задача проверки линеаризуемости – это частный случай задачи функционального тестирования, в которой проверяется, удовлетворяет ли программа функциональным требованиям к ней, заданным в виде

спецификации. Но в отличие от общего случая, спецификация требуется только для последовательных выполнений.

Свойство линеаризуемости во многом сходно с такими свойствами как сериализуемость [4,19], атомарность [11,23], последовательная согласованность (sequential consistency) [17]. В отличие от них, линеаризуемость предполагает наличие спецификации, тогда как эти свойства накладывают ограничения только на саму программу.

Впервые понятие линеаризуемости встречается в работе [16] 1990 года, в которой был предложен способ ручного доказательства этого свойства. Свойство линеаризуемости получило признание в научных работах, но до сих пор не было предложено метода его автоматизированной проверки. Работа ученых в основном концентрировалась на свойствах независимых от спецификации. Так, для программ на языке Java, были разработаны инструменты для проверки атомарности.

Выделим две группы инструментов. К первой группе относятся инструменты статического анализа программ [11,24]. Эти инструменты предполагают аннотирование кода программы или определенный метод программирования. Так, например инструмент [11], предполагает использование специальной системы типов. Основным недостатком этой группы инструментов является большое количество ложных срабатываний, возникающих из-за неверных предположений о динамическом поведении программы.

Ко второй группе относятся инструменты, исследующие динамическое поведение программы. Одними из наиболее распространенных являются инструменты на основе методов проверки моделей (*model checking*) [7,12,15], осуществляющие поиск чередований инструкций в параллельных потоках. В последнее время инструменты, реализующие эти методы, 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 [22], способен проверять свойства программ на языке программирования Java, а инструмент VeriSoft [14] предназначен для проверки программ на языке C. Однако инструменты на их основе, предназначенные для проверки атомарности, сталкиваются с рядом сложностей. Во-первых, чтобы запустить эти инструменты требуется подготовить окружение, т.е. задать набор потоков, вызывающих операции интерфейса с некоторыми значениями параметров. Но так как требование атомарности в общем случае формулируется для неограниченного числа потоков, то проверив атомарность на конечных наборах, мы не можем быть уверенны в атомарности программы в целом. Во-вторых, поиск занимает сравнительно большое время. В зависимости от количества потоков требуется от нескольких минут до нескольких часов. Поэтому, даже ограничившись конечными наборами потоков и, многократными запусками поиска, мы столкнемся со значительными временными затратами.

Отметим также инструменты мониторинга Java программ [6,10,23], которые пытаются проверить атомарность на основе выполнений, возникающих в процессе работы программы. Основной недостаток этих инструментов в том, что они не гарантируют атомарности на всех возможных выполнениях.

На практике линеаризуемые программы широко распространены. Отметим два распространенных класса: библиотеки, предназначенные для многопоточного использования и программы, предоставляющие интерфейс промежуточного уровня клиент-серверных приложений.

В библиотеках, предназначенных для многопоточного использования, такое требование выдвигается по умолчанию. Функциональные требования описываются для каждой операции в отдельности, и требуется, чтобы операции было безопасно вызывать из нескольких потоков (*thread-safe*). В наших терминах это и есть требование линеаризуемости.

Клиент-серверные приложения по своему назначению предоставляют клиентам сервисы. Клиенты могут использовать сервисы одновременно, при этом клиент не должен замечать присутствия других клиентов. Если два клиента совершают какие-то операции, то результат должен быть такой же, как если бы клиентов обслуживали последовательно. Данный класс систем чрезвычайно широк, так как практически все современные системы, предоставляющие сервисы, могут обслуживать несколько клиентов одновременно.

Далее в разделе 2 рассмотрено формальное определение понятия линеаризуемости. В разделе 3 описана модель Java программы, в терминах которой в разделе 4 сформулированы достаточные условия, являющиеся основой для проверки линеаризуемости. В разделах 5 и 6 краткое описание метода проверки линеаризуемости Sapsan сопровождено иллюстрацией на простом примере. А в разделе 7 приведены результаты его применения.

2. Понятие линеаризуемости

2.1. Понятие истории

Мы имеем набор операций *op<имя>*, каждая из которых имеет *начало* (вызов) *op<имя>_begin<параметры>* и *конец* (возврат) *op<имя>_end<результат>*.

История – это конечная последовательность из событий α : *op<имя>_begin<аргументы>* и α : *op<имя>_end<результат>*, где α - поток.

Конец *подходит* (пара) началу, если совпадают потоки и имена операций.

Определение 1. *История последовательная, если:*

1. *Первое событие – начало операции.*
2. *За каждым событием, кроме последнего, сразу же следует подходящий конец.*

История потока α (проекция, подистория) в истории *H* (обозначаем $H \mid \alpha$) – это последовательность всех событий в *H*, у которых имя процесса равно α.

Две истории *H, H'* *эквивалентны* (обозначаем *H~H'*), если для любого процесса α выполнено $H \mid \alpha = H' \mid \alpha$.

История *правильная*, если любая подистория $H \mid \alpha$ – последовательная. Все рассматриваемые в данной работе истории правильные.

Операция в истории является *незаконченной*, если за началом нигде далее в последовательности не следует подходящий конец.

complete(H) – максимальная подпоследовательность *H*, состоящая только из начал и подходящих концов (удалены незаконченные операции).

Множество *S замкнуто по префиксам* (prefix-closed), если для любой истории *H* из *S* верно, что любой префикс *H* тоже в *S*.

Последовательная спецификация программы – это замкнутое по префиксам множество последовательных трасс. История *H соответствует спецификации*, если $H \in S$.

2.2. Определение линеаризуемости

История *H* индуцирует на операциях иррефлексивный частичный порядок $<_H$ такой, что $e_0 <_H e_1$, если $end(e_0)$ предшествует $begin(e_1)$ в *H*.

Определение 2. *История H линеаризуема, если она может быть расширена (добавлением нуль и более возвратов) до некоторой истории H', для которой:*

1. *complete(H') эквивалентна некоторой последовательной истории S, соответствующей спецификации;*
2. $<_H \subseteq <_S$.

2.3. Самолинеаризуемость программы

Для установления свойства линеаризуемости нам будет полезно понятие самолинеаризуемости, независящее от спецификации. Под *достижимой историей программы* будем понимать историю, которая может реально возникнуть в программе. В дальнейшем мы определим понятие достижимой истории на основе трассы выполнения.

Определение 3. *Программа самолинеаризуема, если для любой достижимой истории H существует достижимая последовательная история H' такая, что H~H'.*

Если программа самолинеаризуема, то проверив, что все достижимые последовательные истории соответствуют спецификации, мы покажем линеаризуемость программы.

3. Модель программы

Программа (система, реализация) это тройка: $\langle s_0, S, P \rangle$, где $s_0 \in S$ начальное состояние, S – множество разделяемых состояний (может быть бесконечно), P – конечный набор подпрограмм операций.

Каждая подпрограмма P это четверка $\langle l_0, L, \nu, T \rangle$, где $l_0 \in L$ – начальное локальное состояние, L – локальные состояния (состояния управления), $\nu : T \to \Sigma$ – пометки, $T \subseteq L \times G \times C \times L'$ – переходы.

$\Sigma = \{\tau,$ *op<имя>_begin<аргументы>, op<имя>_end<результат>*$\}$. Все переходы из начального состояния помечены *op<имя>_begin<аргументы>,* промежуточные переходы τ, переход, помеченный *op<имя>_end<результат>*, завершает подпрограмму.

В переходах $C : S \to S$ – команда (command) изменения состояния (инструкция, последовательность инструкций), $G : S \to \{true, false\}$ – охранный предикат (guard). Множества локальных состояний разных подпрограмм не пересекаются. Подпрограмма операции, вообще говоря, может зацикливаться, но для дальнейших алгоритмов мы требуем ацикличность. Начальные и конечные переходы операций не меняют разделяемого состояния.

Для того чтобы *выполнить программу* необходимо задать потоки пользователя $\Psi = \varphi_1, \varphi_2, \ldots, \varphi_n$, которые будут выполняться. *Поток пользователя* φ_i задается как последовательность подпрограмм пользователя $p_0, p_1, \ldots, p_{n_i}$. Поток начинает выполнение в начальном состоянии подпрограммы p_0, при завершении текущей подпрограммы происходит переход из ее конечного состояния в начальное состояние следующей подпрограммы. Поток завершает выполнение после завершения подпрограммы p_{n_i}.

Для заданных пользовательских потоков Ψ определим *состояние выполнения* $g = (s, l^1, l^2, \ldots, l^n)$, s – разделяемое состояние, l^i – локальное состояние потока φ_i. Начальное состояние выполнения $g_0 = (s_0, l_0^1, l_0^2, \ldots, l_0^n)$. Множество всех состояний выполнения обозначим G.

Введем следующие обозначения:

- $enabled(t, s) \equiv t.guard(s)$ – проверка охранного предиката перехода t в s.
- $pre(t)$ – начальное состояние перехода t, $post(t)$ – конечное состояние.

- $local(\alpha, g)$ – выдает локальное состояние потока α в состоянии g.
- *shared(g)* – выдает разделяемое состояние s.
- $t(\alpha)$ – обозначает, что t выполняется в потоке $\alpha \in \Psi$.
- $active(t(\alpha), g) \equiv pre(t) = local(\alpha, g)$
- $enabled(t(\alpha), g) \equiv active(t(\alpha), g) \wedge enabled(t, shared(g))$ – возможность выполнить переход $t(\alpha)$ в состоянии g.

Определим переходы $g \xrightarrow{t(\alpha)} g'$, $g = (s, l^1, \ldots, l^\alpha, \ldots, l^n)$. Переход существует, если $enabled(t(\alpha), g) = true$ и $g' = (\hat{s}, l^1, \ldots, \hat{l}^\alpha, \ldots, l^n)$, где $\hat{s} = t.command(s)$ и $\hat{l}^\alpha = post(t)$.

Трасса выполнения программы – это последовательность $t_1(\alpha_1), t_2(\alpha_2), \ldots, t_m(\alpha_m)$ такая, что $g_0 \xrightarrow{t_1(\alpha_1)} g_1 \xrightarrow{t_2(\alpha_2)} \cdots \xrightarrow{t_m(\alpha_m)} g_m$.

Трасса потока – проекция трассы выполнения программы на поток. *Трасса операции в потоке* – проекция трассы потока на операцию.

История выполнения для трассы $\sigma = t_1(\alpha_1), t_2(\alpha_2), \ldots, t_m(\alpha_m)$, обозначим $H(\sigma)$, это последовательность меток $\nu(t_i(\alpha_i))$, из которой удалены все метки τ. История *H достижима*, если существует трасса σ такая, что $H = H(\sigma)$.

4. Достаточные условия самолинеаризуемости

4.1. Понятие независимости

Мы будем использовать классическое определение понятия независимости [9,13,20] (Определение 4) и расширим его для произвольного набора пользовательских потоков Ψ (Определение 5).

Определение 4. *$D(\Psi)$ – рефлексивное, симметричное отношение зависимости для выполнения Ψ, в котором, если $(t_1(\alpha), t_2(\beta)) \notin D(\Psi)$ (независимы), то для любого достижимого состояния g выполнено:*

1. *Из $enabled(t_1(\alpha), g)$ и $g \xrightarrow{t_1(\alpha)} g'$ следует, что $enabled(t_2(\beta), g') = enabled(t_2(\beta), g)$;*
2. *Если $enabled(t_1(\alpha), g)$ и $enabled(t_2(\beta), g)$, то существует единственное состояние $\hat{g}$ такое, что $g \xrightarrow{t_1(\alpha), t_2(\beta)} \hat{g}$ и $g \xrightarrow{t_2(\beta), t_1(\alpha)} \hat{g}$.*

Определение 5. D – рефлексивное, симметричное отношение зависимости для программы, в котором, если $(t_1, t_2) \notin D$ (независимы), то $\forall \Psi; \forall \alpha, \beta; \exists D(\Psi) : (t_1(\alpha), t_2(\beta)) \notin D(\Psi)$.

Замечание. *Независимые переходы можно переставлять местами, история трассы и конечное состояние выполнения при этом не изменится [13].*

4.2. Понятие цикла по зависимостям

Пусть есть трасса программы $\sigma = t_1(\alpha_1), t_2(\alpha_2), \ldots, t_m(\alpha_m)$. Определим отношение следования.

Определение 6. *Отношение следования (без транзитивного замыкания)*

- $t_i(\alpha_i) < t_j(\alpha_j)$, *если* $t_i(\alpha_i)$ *в трассе лежит раньше* $t_j(\alpha_j)$ *и выполнено одно из условий:*
  - $t_i, t_j \neq \{op\_begin, op\_end\}$, $(t_1, t_2) \in D$ *и* $\alpha_i \neq \alpha_j$
  - $t_i = op\_end$, $t_j = op\_begin$ *и* t_i, t_j *не принадлежат одной операции.*
- $t_i(\alpha_i) = t_j(\alpha_j)$, *если* $\alpha_i = \alpha_j$ *и* t_i, t_j *принадлежат одной операции.*

Утверждение 1. *Пусть в трассе* σ *последовательно встречаются* $t_i(\alpha_i)$, $t_j(\alpha_j)$ *и* $t_i(\alpha_i) \not\leq t_j(\alpha_j)$. *Пусть* σ' *это трасса, полученная из* σ *перестановкой* $t_i(\alpha_i)$, $t_j(\alpha_j)$ *в обратном порядке. Тогда история трассы* σ' *эквивалентна истории трассы* σ ($H(\sigma') = H(\sigma)$).

Следует из того, что $t_i(\alpha_i)$, $t_j(\alpha_j)$ независимы, так как $t_i(\alpha_i) \not\leq t_j(\alpha_j)$.

Последовательность по отношению следования – это последовательно переходов связанных отношениями по определению 6: $t_{i_1}(\alpha_{i_1}) \leq t_{i_2}(\alpha_{i_2}) \leq \cdots \leq t_{i_p}(\alpha_{i_p})$, где $t_{i_j}(\alpha_{i_j})$ – элементы трассы.

Цикл – это последовательность $t_{i_1}(\alpha_{i_1}) < t_{i_2}(\alpha_{i_2}) \leq \cdots < t_{i_p}(\alpha_{i_p})$, в которой $t_{i_1}(\alpha_{i_1}) = t_{i_p}(\alpha_{i_p})$ и $t_{i_1}(\alpha_{i_1})$ лежит раньше $t_{i_{p-1}}(\alpha_{i_{p-1}})$.

Утверждение 2 рассмотрим в данной статье без доказательства.

Утверждение 2. *Если в достижимых трассах нет циклов, то программа самолинеаризуема.*

5. Метод Sapsan

Метод многопоточного тетирования Sapsan состоит из предусловия применения и семи шагов. На вход методу подается программа, предоставляющая интерфейс из операций. Задача метода – проверить линеаризуемость программы.

Демонстрировать метод Sapsan будем на примере *Cell*, представленном на рис. 1. В этом примере интерфейс программы состоит из двух операций:

- *op<insert>: op<insert>_begin(int), op<insert>_end(boolean)*
- *op<delete>: op<delete>_begin(), op<delete>_end(boolean)*

```
 1  Integer x = null;
 2  boolean b = false;
 3
 4  boolean delete() {
 5    synchronized(this) {
 6      if(x!=null && b)
{//3_T
 7        x = null;
 8        b = false;
 9        return true;
10      } else {//3_F1, 3_F2
11        return false;
12      }
13    }
14  }
```

```
15  boolean insert(int i) {
16    boolean r = false;
17    synchronized(this) {
18      if(x==null) {//1_T
19        x = i;
20        r = true;
21      }//1_F
22    }
23    synchronized(this) {
24      if(r) {//2_T
25        b = true;
26        return true;
27      } else {//2_F
28        return false;
29      }
30    }
31  }
```

Рис. 1. Код программы Cell

Программа реализует ячейку, в которой хранится целочисленное значение. Операция *insert* записывает целочисленный элемент в ячейку и возвращает *true*, если ячейка свободна, иначе возвращает *false*. Операция *delete* удаляет элемент из ячейки и возвращает *true*, если ячейка не пуста, иначе возвращает *false*. В нашем простом примере операция *insert* специально немного усложнена, для того чтобы продемонстрировать трудности, встречающиеся в более сложных программах.

5.1. Предусловие применения

Для применения метода Sapsan требуется, чтобы программа удовлетворяла *дисциплине синхронизации доступа к разделяемым переменным*. Суть этой дисциплины заключается в том, что доступ к любой переменной из разных потоков должен быть защищен хотя бы одним общим объектом-монитором. Это условие гарантирует отсутствие *состояний гонок* (race conditions) в

программе. Проверка следования данной дисциплине может осуществляться одним из известных алгоритмов [8,21].

Следование данной дисциплине позволяет рассматривать в качестве переходов программы не отдельные инструкции, а целые последовательности инструкций, называемые *блоками*, ограниченные входами и выходами из объектов-мониторов.

В примере *Cell* используется две разделяемые переменные *x* и *b*. Каждое обращение к этим переменным защищено объектом-монитором *this*. Поэтому дисцилина синхронизации выполнена.

5.2. Шаг 1. Инструментация

Инструмент Sapsan инструментирует программу, т.е. вставляет в скомпилированный код программы дополнительные инструкции. Вместе с основным кодом программы должны быть инструментированы все используемые им библиотеки, в том числе поставляемые с виртуальной машиной Java, на которой будет происходить выполнение. Инструмент вставляет перехваты начала и конца операций, входа и выхода из объекта-монитора. Вставляет код для отслеживания результатов проверки условий в условных выражениях и записи в поля объектов. Инструментация необходима для сохранения трассы выполнения, анализа независимости блоков и управления переключением потоков.

5.3. Шаг 2. Прогон имеющихся тестов

Шаги 2-4 выполняются только при наличии готового поставляемого с программой тестового набора. Причем метод не накладывает никаких ограничений на этот тестовый набор. Цель этих шагов – извлечь максимальную пользу из имеющихся тестов.

В результате прогона тестов мы получаем:

1. Блоки инструкций;
2. Трассы операций;
3. Тестовое покрытие.

В примере *Cell* могут быть выделены блоки:

- *1_T* – блок от начала операции *insert* (стр. 15) до второго входа в монитор (стр. 23), выполнение оператора *if* (стр. 18) идет по ветке *true*.
- *1_F* – блок от начала операции *insert* (стр. 15) до второго входа в монитор (стр. 23), выполнение оператора *if* (стр. 18) идет по ветке *false*.

- *2_T* – блок от начала второго входа в монитор (стр. 23) до возврата из операции (стр. 26), выполнение оператора *if* (стр. 24) идет по ветке *true*.
- *2_F* – блок от начала второго входа в монитор (стр. 23) до возврата из операции (стр. 28), выполнение оператора *if* (стр. 24) идет по ветке *false*.
- *3_T* – блок от начала операции *delete* (стр. 4) до возврата из операции (стр. 9), выполнение оператора *if* (стр. 6) идет по ветке *true*.
- *3_F1* – блок от начала операции *delete* (стр. 4) до возврата из операции (стр. 11), выполнение оператора *if* (стр. 6) идет по ветке *false*, ложен первый операнд конъюнкции (*x!=null*).
- *3_F2* – блок от начала операции *delete* (стр. 4) до возврата из операции (стр. 11), выполнение оператора *if* (стр. 6) идет по ветке *false*, ложен второй операнд конъюнкции (*b*).

Заметим, что блок *3_F2* в последовательных тестах получить мы не можем, т.к. для его появления требуется, чтобы один из потоков выполнил запись в переменную *x* в операции *insert*, но не зафиксировал ее присваиванием переменной *b* значения *true*.

Примеры трасс операций.

Операция *insert*:

- *op<insert>_begin<1>, 1_T, 2_T, op<insert>_end<true>*
- *op<insert>_begin<2>, 1_F, 2_F, op<insert>_end<false>*

Операция *delete*:

- *op<delete>_begin, 3_T, op<delete>_end<true>*
- *op<delete>_begin, 3_F1, op<delete>_end<false>*

5.4. Шаг 3. Оценка покрытия

Для того чтобы гарантировать линеаризуемость на данном шаге, мы должны убедиться, что выполнены два требования:

1. Множество трасс операций, выделенное при прогоне тестов, содержит все трассы операций, встречающиеся в трассах выполнений программы.
2. Программа соответствует спецификации на последовательных выполнениях.

Данный шаг метода опирается на уже имеющиеся методы тестирования и оценки полноты тестов. Оценить, покрыли ли все трассы операций, помогает покрытие по путям в графе потока управления (path coverage) [1]. В примере

Cell оценка данного покрытия поможет выявить отсутствие трасс с блоком *3_F2*.

Для оценки соответствия спецификации на последовательных выполнениях разработано множество методов. Для примера *Cell* были разработаны тесты JUnit[26].

5.5. Шаг 4. Проверка достаточных условий

На данном шаге запускается инструмент Sapsan, который проверяет, выполнены ли достаточные условия.

Инструмент Sapsan выводит начальное отношение зависимости. Независимыми полагаются блоки, не изменяющие разделяемое состояние и блоки, обращающиеся к разделяемым переменным при непересекающихся множествах захваченных объектов мониторов.

В примере *Cell* блоки *1_F, 2_F, 3_F1, 3_F2* независимы между собой, т.к. не изменяют разделяемое состояние. Остальные пары блоков потенциально зависимы.

Инструмент ищет циклы. Если цикл найден, то достаточные условия не выполнены, иначе выполнены. В нашем примере циклы будут найдены, например $\alpha:1\_T < \beta:3\_T < \alpha:2\_T = \alpha:1\_T$.

5.6. Шаг 5. Разработка многопоточных тестов

Для использования продвинутых возможностей инструмента Sapsan необходимо разработать специальные тесты. К тестам предъявляются те же требования, что и на шаге 3. Кроме того, тесты обязаны предоставлять следующие возможности:

1. Создавать отдельный поток, в котором была достигнута заданная трасса операции.
2. Переходить в состояние, в котором начинался тест.
3. Вычислять конечное состояние.
4. Проверять соответствие трассы спецификации.

На данный момент разработаны библиотеки, которые позволяют разрабатывать тесты с данными возможностями на основе JUnit и UniTESK[2,3].

Для примера *Cell* в качестве тестового набора использовались тестовые варианты JUnit. Каждый тестовый вариант *testCase* состоит из одного вызова операции с некоторыми параметрами. Для тестового варианта задан метод *setUp* – инициализации состояния и метод *tearDown* – приведения системы в начальное состояние.

В процессе выполнения тестов сохраняется информация о том, каким тестовым вариантом была достигнута данная трасса. Возможность создания

отдельного потока обеспечивается библиотекой, которая по сохраненной информации находит тестовый вариант и создает выполняющий его поток.

Переход в состояние начала теста обеспечивается методами *setUp* и *tearDown*.

Для всех тестовых вариантов задан метод *getState*, вычисляющий текущее состояние системы. За счет этого обеспечивается возможность вычисления конечного состояния.

В отличие от обычных тестов JUnit, проверяющих соответствие спецификации прямо в тестовых вариантах (*assert*…), в многопоточных тестах проверки должны быть вынесены в отдельный метод *checkResult*. Таким образом, обычный запуск тестового варианта состоит из последовательности: *setUp, testCase, getState, checkResult, tearDown*.

5.7. Шаг 6. Эвристический анализ программы

Эвристический анализ программы включает действия шагов 2 и 4, т.е. тесты выполняются обычным образом, собираются блоки, трассы, покрытие, дальше происходит проверка достаточных условий линеаризуемости. Если они не выполнены, то начинается основная часть эвристического анализа.

Как видно из достаточных условий, на их выполнимость оказывает влияние независимость блоков и появление трасс операций в трассах выполнений программы. Чем больше у нас знаний о независимости блоков и непоявлении трасс, тем больше шансов, что достаточные условия будут выполнены. На шаге 4 мы считали независимыми только те блоки, которые были гарантированно независимы. Считали, что любая комбинация из трасс операций составляет трассу выполнения. На этом же шаге инструмент Sapsan уточняет эту информацию эвристическими методами. Более подробно анализ описан в разделе 6.

В результате эвристического анализа для примера *Cell* мы имеем:

1. Установлена независимость блоков *(1_T, 1_T), (1_T, 2_T), (2_T, 2_T), (3_T, 3_T), (3_T, 3_F2), (1_T, 2_F), (2_T, 1_F), (2_T, 2_F), (2_T, 3_F1), (3_T, 2_F)*.
2. Установлена зависимость *(1_T, 1_F), (1_T, 3_T), (1_T, 3_F1), (3_F2, 1_T), (3_T, 2_T), (2_T, 3_F2), (3_T, 1_F), (3_F1, 3_T)*.
3. Установлено непоявление упорядоченных пар *(1_F, 1_T), (1_T, 1_T), (1_T, 2_T), (2_T, 1_T), (1_T, 3_T), (1_T, 3_F1), (3_F2, 1_T), (2_T, 2_T), (3_T, 2_T), (2_T, 3_F1), (3_F1, 2_T), (2_T, 3_F2), (3_T, 1_F), (3_T, 3_T), (3_F1, 3_T), (3_T, 3_F2), (3_F2, 3_T)*.
4. Выведено, что блоки *1_T, 2_T, 3_T, 3_F1* не встречаются между блоками *1_T, 2_T*.

Циклы вида $\alpha:1\_T < \beta:3\_T < \alpha:2\_T = \alpha:1\_T$, $\alpha:1\_T < \beta:3\_F2 < \alpha:2\_T = \alpha:1\_T$ становятся недостижимыми, остальные невозможны по установленной независимости. Поэтому программа *Cell* линеаризуема.

5.8. Шаг 7. Анализ результатов

Возможно три варианта ответов инструмента «Да», «Нет», «Не удалось установить». Если инструмент выдает ответ «Да», то программа линеаризуема. Если ответ «Нет», то программа не линеаризуема и инструмент выдает трасу выполнения, набор потоков и состояние, на которых нарушается свойство линеаризуемости. Возможен также и третий вариант, когда инструменту не удалось установить линеаризуема ли программа. На выходе в этом случае трассы операций, для которых не выполнены достаточные условия. Эти подозрительные трассы могут быть проанализированы вручную.

6. Эвристический анализ программы

Схема работы анализа показана на рис. 2. Для найденных трасс операций проверяются достаточные условия самолинеаризуемости. Если достаточные условия выполнены, то, следовательно, мы показали, что программа линеаризуема. Иначе возможно три варианта:

1. В программе присутствует трасса, на которой нарушается линеаризуемость;
2. Наша информация о независимости блоков неполна;
3. Невозможно установить линеаризуемость, используя достаточные условия.

В первом случае нам необходимо попытаться найти подходящую трассу выполнения. Во втором попытаться установить независимость блоков.

Эвристический анализ в инструменте Sapsan основан на классических алгоритмах поиска. На входе у алгоритмов программа и набор пользовательских потоков Ψ (см. разд. 3). Имея Ψ, программу можно выполнить и получить какую-то трассу выполнения. В зависимости от чередования инструкций могут получаться разные трассы. Все множество трасс для заданного Ψ – это трассы, полученные при всех возможных чередованиях инструкций потоков.

Трасс, как правило, очень много, но не все из них требуются для проверки требуемых свойств. В зависимости от проверяемого свойства выбирается условие эквивалентности трасс. Например, для установления линеаризуемости можно считать эквивалентными трассы с одинаковыми историями. Задача алгоритма поиска – построить полное множество трасс, т.е. множество, содержащее все неэквивалентные трассы.

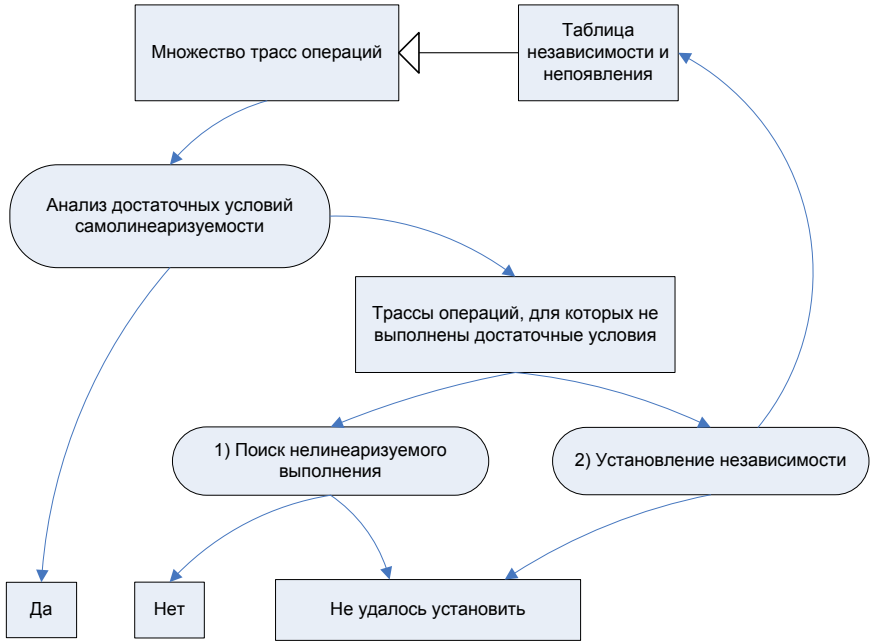

Рис. 2. Установление линеаризуемости

6.1. Алгоритмы поиска

В данной работе мы используем алгоритмы, описанные в работе Годфруа [14]. Эти алгоритмы существенно используют понятие независимости для оптимизации поиска. Различают два класса алгоритмов поиска:

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

Для того чтобы осуществлять поиск, 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 программы 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 в графе

переходов. 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 [5,18,22], 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 обращаются к системам, написанным на других языках, например, к базам данных.

6.2. Установление независимости

В инструменте Sapsan в качестве основы используется *алгоритм поиска* достижимых выполнений без сохранения состояний [14], реализованный в инструменте VeriSoft. Этот алгоритм был оснащен возможностью *поиска по шаблону*, что позволило сократить пространство поиска. Шаблон задает порядок, в котором должны встретиться переходы в искомой трассе. Шаблоны строятся на основе пар путей, для которых не выполнено достаточное условие самолинеаризуемости (Утверждение 2). Поиск по шаблону нацеливается только на трассы, удовлетворяющие шаблону, т.е. трассы выполнения, не подходящие под шаблон, отбрасываются.

В примере Cell с помощью поиска по шаблонам было установлено, что следующие пары блоков не появляются ни в одной из трасс: (1_F, 1_T), (1_T, 1_T), (1_T, 2_T), (2_T, 1_T), (1_T, 3_T), (1_T, 3_F1), (3_F2, 1_T), (2_T, 2_T), (3_T, 2_T), (2_T, 3_F1), (3_F1, 2_T), (2_T, 3_F2), (3_T, 1_F), (3_T, 3_T), (3_F1, 3_T), (3_T, 3_F2), (3_F2, 3_T). Из непоявления пар можно сделать вывод о зависимости блоков. Если пара (X,Y) появляется, а в том же состоянии пара (Y,X) – нет, то данная пара является гарантированно зависимой. Отсюда мы получили, что пары (1_T, 1_T), (1_T, 2_T), (2_T, 2_T), (3_T, 3_T), (3_T, 3_F2), (1_T, 2_F), (2_T, 1_F), (2_T, 2_F), (2_T, 3_F1), (3_T, 2_F) – зависимы. Если не появляются обе пары (X,Y) и (Y,X), то пара независима. Так мы получили независимость (1_T, 1_F), (1_T, 3_T), (1_T, 3_F1), (3_F2, 1_T), (3_T, 2_T). Остальные пары (2_T, 3_F2), (3_T, 1_F), (3_F1, 3_T) были признаны независимыми, так как перестановка их местами не меняет конечное состояние ни в одном из выполнений.

Метод установки независимости является эвристическим, потому что в общем случае ответить на вопрос, достижимо данное выполнение, невозможно. Поэтому для поиска выбирается ограниченный набор пользовательских потоков Ψ. Ограничения на количество потоков задает пользователь.

7. Результаты применения метода

На данный момент метод был успешно применен на нескольких простых примерах, встречающихся в литературе. Были найдены известные ошибки в StringBuffer[11,15] и Vector[23]. Для примера MultiSet из [6] инструмент не смог установить линеаризуемость (MultiSet не самолинеаризуем), но было показано, что параллельное выполнение всех троек операций удовлетворяет спецификации.

Кроме того, были написаны многопоточные тесты для реализации кэша Ehcache [25], который оптимизирует доступ к хранящимся в нем элементам, размещая часто используемые элементы в памяти и сохраняя остальные на диске. Реализация этого кэша составляет примерно 40 тысяч строк кода на Java. Было выявлено нарушение достаточных условий и найдено выполнение не соответствующее спецификации.

8. Заключение

В работе описан новый метод Sapsan, поддержанный одноименным инструментом, который позволяет автоматизированно проверять свойство линеаризуемости программ.

Во введении были рассмотрены различные подходы к проверке линеаризуемости. Было отмечено, что существующие инструменты нацелены на проверку независимых от спецификации свойств. В методе Sapsan, напротив, спецификация является важной составляющей. В эвристическом анализе спецификация позволяет утверждать, что ошибка реально существует, а не просто предупреждать о возможной ошибке. Кроме того, спецификация используется при установлении независимости блоков.

По сравнению с инструментами проверки моделей (model checking) метод Sapsan позволяет делать заключение о линеаризуемости программы для произвольного набора пользовательских потоков. Кроме того, алгоритмы эвристического анализа в силу их узкоспециализированной направленности на установление независимости оказываются быстрее классических алгоритмов поиска.

Экспериментальные результаты показывают, что метод применим для практически значимых приложений.

Литература

[1] Борис Бейзер. *Тестирование черного ящика*. Питер, 2004.

[2] Виктор В. Кулямин, Александр К. Петренко, Александр С. Косачев, Игорь Б. Бурдонов. *Подход UniTESK к разработке тестов*. Программирование, том 29, стр. 25-43, 2003.
[3] Алексей В. Хорошилов. *Спецификация и тестирование компонентов с асинхронным интерфейсом*. Диссертация на соискание ученой степени кандидата физико-математических наук, 2006.
[4] Rajeev Alur, Ken Mcmillan, Doron Peled. *Model-checking of correctness conditions for concurrent objects*. Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science, стр. 219-228, 1996.
[5] Derek Bruening. *Systematic testing of multithreaded Java programs*. Master's thesis, MIT, 1999.
[6] Tayfun Elmas, Serdar Tasiran, Shaz Qadeer. *VYRD: verifYing concurrent programs by runtime refinement-violation detection*. Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, стр. 27-37, 2005.
[7] Tayfun Elmas, Serdar Tasiran. *VyrdMC: Driving Runtime Refinement Checking with Model Checkers*. Proceedings of the Fifth Workshop on Runtime Verification, том 144(4), стр. 41-56, 2005.
[8] Tayfun Elmas, Shaz Qadeer, Serdar Tasiran. *Goldilocks: a race and transaction-aware Java runtime*. Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, стр. 245-255, 2007.
[9] Cormac Flanagan, Patrice Godefroid. *Dynamic partial-order reduction for model checking software*. Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on the Principles of Programming Languages, стр. 110-121, 2005.
[10] Cormac Flanagan, Stephen N. Freund. *Atomizer: A dynamic atomicity checker for multithreaded programs*. Proceedings of the ACM Symposium on the Principles of Programming Languages, стр. 256-267, 2004.
[11] Cormac Flanagan, Shaz Qadeer. *A type and effect system for atomicity*. Proceedings of the ACM Conference on Programming Language Design and Implementation, том 38(5), стр. 338-349, 2003.
[12] Cormac Flanagan. *Verifying commit-atomicity using model-checking*. Proceedings of 11th International SPIN Workshop, том 2989, стр. 252-266, 2004.
[13] Patrice Godefroid. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*. Springer-Verlag, 1996.
[14] Patrice Godefroid. *Model checking for programming languages using Verisoft*. Symposium on Principles of Programming Languages, стр. 174-186, 1997.
[15] John Hatcliff, Robby, Matthew B. Dwyer. *Verifying atomicity specifications for concurrent object-oriented software using model checking*. Proceedings of the Fifth International Conference on Verification, Model Checking and Abstract Interpretation, 2004.
[16] Maurice P. Herlihy, Jeannette M. Wing. *Linearizability: a correctness condition for concurrent objects*. ACM Transactions on Programming Languages and Systems, стр. 463-492, 1990.
[17] Leslie Lamport. *Specifying concurrent program modules*. ACM Transactions on Programming Languages and Systems, том 5(2), стр. 190-222, 1983.
[18] Vadim S. Mutilin. *Concurrent testing of Java components using Java PathFinder*. Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation, том 2, стр. 53-59, 2006.
[19] Christos H. Papadimitriou. *The serializability of concurrent database updates*. Journal of the ACM, том 26(4), стр. 631-653, 1979.

[20] Doron Peled. *Combining partial order reductions with on-the-fly model checking*. Formal Methods in System Design, том 8, стр. 39-64, 1996.
[21] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, Thomas Anderson. *Eraser: A dynamic data race detector for multithreaded programs*. ACM Transactions on Computer Systems, том 15(4), стр. 391-411, 1997.
[22] Willem Visser, Klaus Havelund, Guillaume Brat, Seungjoon Park, Flavio Lerda. *Model checking programs*. Automated Software Engineering, том 10(2), стр. 203-232, 2003.
[23] Liqiang Wang, Scott D. Stoller. *Runtime analysis of atomicity for multithreaded programs*. IEEE Transactions on Software Engineering, том 32(2), стр. 93-110, 2006.
[24] Liqiang Wang, Scott D. Stoller. Static analysis of atomicity for programs with non-blocking synchronization. Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, стр. 61-71, 2005.
[25] http://ehcache.sourceforge.net
[26] http://www.junit.org