Применение статического анализа для оптимизации динамического поиска гонок

Роскошный Яков
Университет ИТМО
watson9494@gmail.com

Цителов Дмитрий
ООО "Эксперт-Система"
tsitelov@acm.org

Трифанов Виталий
ООО "Эксперт-Система"
trifanov@devexperts.com

Елизаров Роман
ООО "Эксперт-Система"
elizarov@devexperts.com

***Аннотация*—Обнаружение гонок – актуальная задача верификации многопоточных программ. Существующие методы подразделяются на динамические, статические и смешанные. Одной из основных проблем динамических методов являются высокие накладные расходы, в частности, необходимость контролировать все обращения к разделяемым переменным, по которым возможна гонка. Для обеспечения корректной синхронизации при обращении к потенциально разделяемых переменным часто применяются устойчивые шаблонные решения. В статье рассматриваются некоторые наиболее типичные шаблоны обеспечения синхронизации доступа к разделяемым переменным и перспективы доказательства их корректности средствами статического анализа. Предложен алгоритм определения переменных защищённых во всех возможных ветвях исполнения и реализован соответствующий инструмент для языка Java. Инструмент апробирован на различных синтетических тестах и реальных программах, интегрирован с ранее разработанным авторами динамическим детектором гонок jDRD, который был представлен на TMPA-2013 [1].**

Ключевые слова—статический анализ; поиск гонок; многопоточность; Java;

I. Введение

Многие современные языки программирования, в том числе Java, используют многопоточную модель с разделяемой памятью. Для публикации изменений, сделанных потоком, и для получения изменений, сделанных другими потоками предусмотрены операции синхронизации. Контракты модели памяти описывают формальные гарантии, которые предоставляют различные операции синхронизации. Использование различных операций синхронизации обеспечивает частичную упорядоченность между операциями в различных потоках. Если между обращениями к разделяемой переменной нет упорядочивания по времени, и хотя бы одна операция является записью, то наступает *состояние гонки*. Обнаружить гонку на этапе тестирования очень сложно, поскольку исполнение многопоточных программ в общем случае недетерминировано и зависит от большого количества неконтролируемых факторов. Эффект гонки может проявляться только при очень специфическом порядке выполнения программы.. Если гонка произошла, то испорченные данные могут долго распространяться по программе, и несоответствие может обнаружиться совсем в другом месте. Повторный запуск программы на тех же данных может не привести к возникновению гонки. Существующие методы поиска гонок подразделяются на *динамические*, *статические* и *смешанные*. Задача нахождения гонок статическим анализом является NP-трудной[2, 3], поэтому при статическом подходе обычно снижается точность и полнота результата. Одной из главных проблем динамического подхода является снижение производительности анализируемой программы, поскольку в основе лежит регистрация обращений ко всем потенциально разделяемым переменным.

Для синхронизации разделяемых переменных часто применяются устойчивые шаблонные решения, которые гарантируют отсутствие гонок при доступе к этим переменным. В данной работе рассматриваются шаблоны обеспечения синхронизации доступа к данным, а также возможности доказательства их корректности средствами статического анализа.

Представлен алгоритм и инструмент для нахождения переменных, защищенных во всех ветвях исполнения. Инструмент интегрирован с динамическим детектором гонок для Java-программ jDRD.

II. Шаблоны обеспечения синхронизации

Для обеспечения синхронизации используются различные конструкции. Некоторые конструкции удовлетворяют повторяющимся шаблонам, для которых можно доказать корректность синхронизации объекта.

Для выделения шаблонов рассматривались поля, по которым были зарегистрированы обращения и не было найдено гонок динамическим детектором. Анализировались поля, обращения к которым встречались наиболее часто. Были выделены следующие шаблоны :

- *Безопасная публикация объекта без последующих изменений (Safe published read-only)*. Поля класса инициализируются на начальном этапе конфигурации, а затем только читаются (речь не идёт о так называемых *неизменяемых (immutable)* объектах с исключительно final полями). Типичный представитель – конфигурация приложения, доступная через статический контекст. Гонки по таким полям могут возникнуть только при отсутствии синхронизации между начальной инициализацией объектов данного класса и их дальнейшим использованием. Для доказательства отсутствия гонок при обращении к таким полям потребуется проанализировать последовательность операций инициализации и передачи объектов между потоками, при этом, в общем случае, отсутствие гонок может быть гарантировано только для некоторых объектов класса. Доказательство теории об отсутствии гонок по полю будет при этом *локальным*, то есть относиться к некоторой ограниченной области кода или зависеть от точки порождения объекта.
- Объект, *принадлежащий потоку (Thread confined)*. Поля класса, экземпляры объектов которого не покидают контекст одного потока. Это свойство также в общем случае зависит от контекста использования или точки порождения объекта.
- *Безопасная публикация объекта (Safe publishing)*. Поля класса, экземпляры которого передаются потоку, и затем не покидают его пределов. Типичный случай – конструирование потомка класса Thread или Runnable для инициализации нового потока. Это свойство также в общем случае зависит от экземпляра класса. Для верификации таких полей понадобится различать экземпляры объектов
- *Внутренне синхронизированное поле (Internally synchronized)*. Поле корректно защищено внутренней блокировкой. То есть, существует такая блокировка, что любая операция с полем проводится с этой блокировкой. Данное свойство является *глобальным*, то есть не зависит от экземпляра объекта и контекста использования.
- *Неизменяемые (Immutable)* объекты – все их поля final и ссылка на объект (this) не «утекает» при конструировании. Данное свойство глобально и тривиально доказывается.[1]

Если теория об отсутствии гонок по полю глобальна, то можно исключить из динамического анализа все обращения к этому полю. Проверка локальных теорий в случае динамического детектора потребует встраивания дополнительных проверок на соответствие контексту при инструментировании байт-кода или генерации альтернативных реализаций класса, как предложено в [11]. Оба подхода потребовали бы существенно более сложных технических решений, поэтому было принято решение начать с исследования оптимизаций на основе глобальных свойств. В данной работе рассматривается поиск внутренне синхронизированных полей.

[1] Объекты такого типа не попали в область часто используемых полей, поскольку jDRD исключает их из анализа автоматически.

III. Алгоритм для поиска корректно-синхронизированных полей.

Будем называть поле корректно-синхронизированным, если существует такая блокировка, что любая операция с полем проводится под этой блокировкой.

Формально, поле f является корректно-синхронизированным, если существует такой синхронизационный объект l, что любая операция доступа к полю *access(f)* в потоке T_i выполняется между взятием и освобождением объекта *lock* в том же потоке при любом сценарии исполнения ($a \rightarrow b$ означает *a happens-before b*):

T_i: *acquire(l)* → *access(f)* → *release(l)*

Поскольку по определению синхронизационного объекта для произвольных обращений либо T_i:*release(l)* → T_j:*acquire(l)* либо T_j:*release(l)* → T_i:*acquire(l)*, то для любой пары операций T_i:*access(f)* и T_j:*access(f)* найдётся такая пара обращений (T_i:*release(l)* → T_j:*acquire(l)*) или (T_j:*release(l)* → T_i:*acquire(l)*), что либо будет верно T_i:*access(f)* → T_i:*release(l)* → T_j:*acquire(l)* → T_j:*access(f)*, либо T_j:*access(f)* → T_j:*release(l)* → T_i:*acquire(l)* → T_i:*access(f)* соответственно. Таким образом, между любыми двумя обращениями к полю f есть отношение *happens-before* и гонка при обращении к этому полю невозможна.

Алгоритм ищет для полей блокировки, удовлетворяющие условиям. Алгоритм можно разделить на 3 части.

- Получение промежуточного представления и графа потока исполнения.
- Получение множества возможных блокировок.
- Обход графа потока исполнения и выделение корректно-синхронизированных полей.

A. *Получение промежуточного представления и графа потока исполнения.*

Граф потока исполнения (англ. control flow graph, CFG) — это все возможные пути исполнения части программы, представленные в виде графа. Вершинами данного графа являются последовательности операций, не содержащие в себе ни операций передачи управления, ни точек, на которые управление передается из других частей программы. Ребра показывают возможные переходы между операциями [4]. Построение данных графов зависит от выбранной формы представления программы. Для Java-программ изначально доступно представление в виде байт-кода. Если доступен исходный код, то можно проводить анализ самой Java-программы. Данные представления неудобны для последующего анализа. Байт-код имеет большое количество инструкций. Java-код имеет большое

количество синтаксических конструкций, которые трудно проанализировать.

Промежуточное представление (англ. Intermediate representation, IR) — язык абстрактной машины, упрощающий проведение анализа.

Для данной работы было выбрано промежуточное представление приведенное в *SSA*-форму [5]. SSA-форма технически упрощает получение множества блокировок. В промежуточном представлении, используемом в данной работе, вся работа со стеком заменена на локальные переменные. Как и в байт-коде имеется две примитивные операции синхронизации: *monitorEnter* и *monitorExit*. Эти операции отвечают за взятие и освобождение монитора объекта. Каждая переменная имеет единственное место присваивания, так как представление удовлетворяет *SSA*-форме. Для присваивания переменной может использоваться *φ-функция* Если локальная переменная может принимать несколько значений, то данная переменная принимает значение φ-функции от всех возможных ее значений. В данной работе промежуточное представление и CFG получены с помощью библиотеки Soot [6]. Присваивания в данном представлении удовлетворяют следующей грамматике (local — локальная переменная, field — поле класса, constant – константа).

Таблица I. ГРАММАТИКА ИСПОЛЬЗУЕМОГО IR.

Продукции грамматики	
imm	local constant
expr	imm_1 binop imm_2 (type) imm imm instanceof type invokeExpr new refType newarray (type) [imm] neg imm
assignStmt	local = φ(imm_1, imm_2, ...) local = local.field | field | imm | expr field = imm local.field = imm

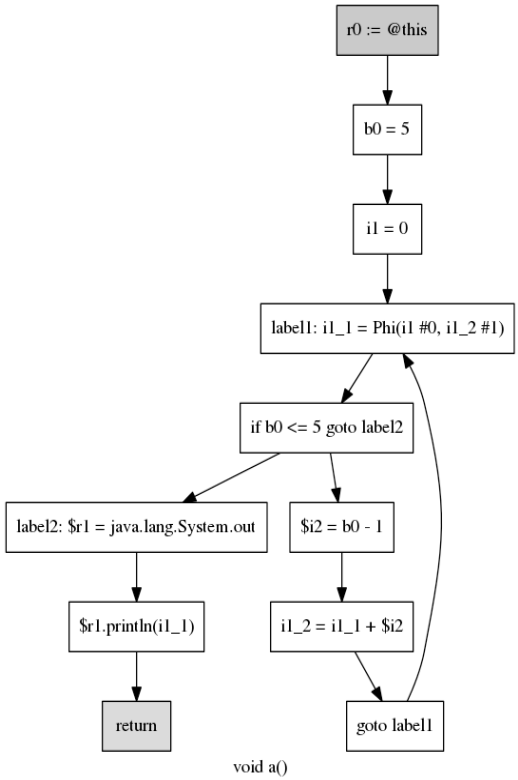
Листинг I. ПРИМЕР JAVA-ПРОГРАММЫ

```
public void a() {
        int x = 5;
        int y = 0;
        while (x > 5) {
                y += x − 1;
        }
        System.out.println(y) ;
}
```

Листинг II. ОПИСАННОЕ ПРОМЕЖУТОЧНОЕ ПРЕДСТАВЛЕНИЕ

```
public void a() {
        r0 := @this ;
        b0 = 5;
        i1 = 0;
      label1 :
        i1_1 = Phi(i1 #0, i1_2 #1);
        if b0 <= 5 goto label2 ;
        $i2 = b0 − 1;
        i1_2 = i1_1 + $i2 ;
        goto label1 ;
      label2 :
        $r1 = java.lang.System.out;
        $r1.println(i1_1) ;
        return ;
}
```

Рис 1. СХЕМА CFG МЕТОДА



B. Получение множества возможных блокировок.

Алгоритм должен быть точным, то есть сообщать только о тех полях, которые действительно корректно-синхронизированы. В полученном представлении операция взятия блокировки осуществляется с локальной переменной. Под блокировкой подразумевается ссылка на объект, по которому осуществляется синхронизация(носитель монитора при встроенной синхронизации или объект блокировки для java.util.concurrent). При обходе графа, который будет рассмотрен далее, нужно поддерживать текущее множество взятых блокировок. Если нельзя статически доказать, что локальная переменная всегда ссылается на одно поле, или переменная ссылается на поле, которое может измениться, то операции взятия блокировки по

Листинг III. БЛОКИРОВКА С НЕСКОЛЬКИМИ ВОЗМОЖНЫМИ ЗНАЧЕНИЯМИ

```
Class A {
        final Object lock1 = new Object() ;
        final Object lock2 = new Object() ;
        void a() {
                Object lock ;
                if (System.currentTimeMilis() % 10000 == 0) {
                        lock = lock1 ;
                } else {
                        lock = lock2 ;
                }
                syncronized(lock) {
                        // some code
                }
        }
}
```

Листинг IV. БЛОКИРОВКА ПО ПОЛЮ ДРУГОГО КЛАССА.

```
Class A {
        final B b = new B() ;
        public void a() {
                synchronized (b. lock) {
                        //some code
                }
        }
}
```

таким переменным не должны добавлять информацию относительно взятых блокировок. Так как полученное представление является SSA, то каждая переменная имеет единственное место инициализации. Если переменная инициализируется как *φ*-функция то эта переменная может иметь не единственное значение. Также если локальная переменная *l1* принимает значение другой локальной переменной *l2* или поля локальной переменной *l2.field*, и *l2* может иметь не единственное значение, то *l1* тоже может иметь не единственное значение и не может являться блокировкой.

Сначала выделим множество переменных, которые инициализируются φ-функциями. Затем построим замыкание данного множества относительно операций присваивания. Полученное множество, очевидно, будет искомым.

Поле, которое может являться потенциальной блокировкой должно быть неизменяемым, то есть должно иметь модификатор final. В общем случае каждое поле в пути блокировки должно иметь модификатор final. В случае со static полями достаточно проверить, что поле имеет модификатор final[1].

C. *Обход графа потока исполнения.*

Обход является рекурсивным, напоминает обход в глубину, но с некоторыми отличиями. При обходе поддерживается множество текущих взятых блокировок *curLocks*. Также для каждой вершины *CFG* хранится множество блокировок, с которыми обход уже посещал данную вершину *v.locks*. При входе в вершину v нужно сравнить *curLocks* и *v.locks*. Если $v.locks \subseteq curLocks$, то можно не продолжать обход вершины v. Иначе, в *v.locks* и *curLocks* запишем $v.locks \cap curLocks$ и продолжим обход. Записывать нужно пересечение, так как если существует ветка обхода, в которой вершина *v* посещена без блокировки *l*, то нельзя гарантировать, что операция вершины *v* защищена блокировкой *l*. Изменять *curLocks* нужно при операциях взятия и освобождения блокировки. Для каждого поля сохраним множество блокировок, с которыми обращались к данному полю *f.locks*. При обращении к полю *f* нужно пересечь *f.locks* и *curLocks*.

Рассмотрим подробно обработку операций синхронизации и обращений к полям.

1) *Операции синхронизации.* Если при обходе встретилась операция синхронизации, то нужно изменить *curLocks*. Но сначала нужно проверить, что переменная, над которой осуществляется операция синхронизации, может являться блокировкой. Данная проверка описана в предыдущем разделе. Далее, если текущая операция — операция взятия блокировки, то добавляем блокировку в *curLocks*; а если операция освобождения, то удаляем блокировку из *curLocks*. Помимо стандартных операций *monitorEnter* и *monitorExit* в данной работе рассмотрены блокировки пакета java.util.concurrent и их парные операции *lock*() и *unlock*().

2) *Обращения к полям.* При обращении к полю может возникнуть состояние гонки. Но, если существует блокировка *l*, такая что любая операция чтения и записи с полем *v* производится со взятой *l*, то поле *v* корректно-синхронизировано. Таким образом, для каждого поля *f* надо поддерживать множество блокировок *f.locks*, с которыми гарантировано обращались к данному полю. А при обращении к полю *f* сужать *f.locks* до пересечения *f.locks* и *curLocks*.

D. *Обработка методов, защищенных блокировкой.*

Пока в алгоритме рассматривался обход *CFG* каждого метода независимо. Но существуют методы, вызовы которых всегда защищены определенной блокировкой. Соответственно, любая операция в данном методе защищена этой блокировкой. Анализ методов может добавить информации о множестве текущих блокировок, что приведет к увеличению найденных корректно-синхронизированных полей. Предварительно выделив для метода множество блокировок, с которыми он гарантировано вызывается, можно улучшить анализ.

Для поиска блокировок, которыми защищен метод, можно использовать алгоритм аналогичный базовому алгоритму поиска корректно-синхронизированных полей. Для каждого метода *m* надо поддерживать множество блокировок *m.locks*, с которыми гарантировано вызывался данный метод. При операции вызова метода m записывать

Листинг V. ПСЕВДОКОД ОБХОДА ГРАФА

```
function visit( CFGVertex v, Set<Lock> currentLocks)
        if v.locks ⊆ currentLocks then
                break
        else
                v.locks ← v.locks ∩ currentLocks
                currentLocks ← v.locks ∩ currentLocks
        op ← v.getOperation()
        if op.isMonitorEnterOperation() then
                currentLocks.add(v.getOperations.getLock())
        if op.isMonitorExitOperation() then
                currentLocks.remove(v.getOperations.getLock())
        if op.isFieldAssignmentOperation() then
                field ← op.getField()
                field.locks ← field.locks ∩ currentLocks
        for all CFGVertex c : v.childs
                do visit(c, currentLocks)
end function
```

[1] Строго говоря, поле хранящее ссылку на объект блокировки может и не быть final, если гарантируется его неизменность и инициализация предшествует любому использованию. Однако это является плохим стилем и на практике редко встречается.

в *m.locks* пересечение *curLocks* и *m.locks*.

После одного обхода для каждого метода *m* будет сформировано множество *m.locks*. Далее можно повторить обход графа с появившимися новыми блокировками. Второй и последующие обходы нужны, так как после каждого обхода множество блокировок, которые защищают метод может увеличиться. Если после очередного обхода в множество блокировок, защищающих метод *m*, добавлена блокировка *l*, то все операции метода *m* защищены *l*. Это означает, что при следующем обходе, любой метод *k*, вызываемый из m может стать защищенным блокировкой *l*. Оценим количество таких обходов. Если после очередного обхода для каждого метода *m* не изменилось *m.locks*, то можно завершать алгоритм. Теоретически может понадобиться M×L обходов (M — количество анализируемых методов, L — количество блокировок, которыми защищен хотя бы один метод). На практике нескольких (трех или четырех) таких обходов достаточно, так как операции синхронизации редко используются для того, чтобы синхронизировать операции через пять вызовов метода.

Также при обработке методов необходимо корректно обрабатывать виртуальные методы. Класс, у которого будет вызван метод, можно определить только во время выполнения. При статическом анализе можно утверждать, что при вызове метода *m* у объекта типа *t* будет выполнен метод одного из наследников класса *t*. Таким образом, все внутренние вызовы метода *m* класса *t* гарантировано защищены блокировкой *l*, если все вызовы метода *m* предков класса *t* защищены *l*.

IV. Практические результаты

Реализован инструмент для поиска корректно-синхронизированных полей, использующий алгоритм из предыдущего раздела. Инструмент написан на языке программирования Java и интегрирован с динамическим детектором гонок jDRD.

jDRD получает список корректно-синхронизированных полей через конфигурационный файл, генерируемый разработанным инструментом. В jDRD был также включен сбор статистики, который отслеживает количество обращений к корректно-синхронизированным полям, выделенным статическим анализом.

Предусмотрено два режима работы: локальный и глобальный. *Локальный режим* подразумевает, что анализируемый код может использоваться сторонними приложениями, не входящими в область анализа. Таким образом, если поле корректно-синхронизировано в рамках анализируемой программы, но оно доступно для изменения по правилам видимости, то в локальном режиме оно не считается корректно-синхронизированным. Аналогичные рассуждения касаются методов. Такой режим нужен для анализа различных библиотек. *Глобальный режим* подразумевает, что в приложении используется только код из области анализа. Если поле корректно-синхронизировано в рамках анализируемой программы, то даже если оно доступно для изменения, глобальный режим отметит его как корректно-синхронизированное. Данный режим предназначен для тестирования законченных приложений. Множество переменных, выделенное в глобальном режиме включает в себя множество, выделенное при работе в локальном режиме.

В качестве операций синхронизации поддерживаются *synchronized* методы и блоки, а также парные операции захвата и освобождения блокировки java.util.concurrent.locks.ReentrantLock.

Было проведено тестирование разработанной программы на различных тестах. Создан набор синтетических тестов, покрывающий большинство конструкций Java-программ. В качестве тестов были использованы программы использующие различные операции синхронизации, обработку исключений(exceptions), статические и нестатические блокировки и поля, внутренние классы и т.д. Также были проведены запуски на реальных библиотеках и приложениях с последующей проверкой результатов.

Инструмент был запущен на различных приложениях, которые активно используют конкурентный доступ к данным. Далее представлено краткое описание тестируемых программ.

- *MARS*(Monitoring and reporting system) — система мониторинга реального времени. Используется для отображения различных данных приложения.
- *dxFeed* — система, отвечающая за быструю доставку больших объёмов данных(котировок).
- *Tomcat* — контейнер сервлетов. Позволяет запускать веб-приложения.
- *jtt* — система менеджмента времени, использующая в качестве сервера JIRA

Таблица II. Полученные результаты

Приложение	Общее количество полей	Корректно-синхронизированных полей	Процент
dxFeed	5212	982	19
MARS	2311	477	21
Tomcat	6793	1546	23
Jtt	1165	176	15

Таблица III. Полученные результаты, без учета final полей

Приложение	Общее количество полей	Корректно-синхронизированных полей	Процент
dxFeed	1555	89	6
MARS	545	20	4
Tomcat	2924	132	5
Jtt	405	28	7

V. Аналогичные исследования

В настоящее время статический анализ рассмотрен довольно широко. Существует большое количество инструментов, которые проводят статический анализ Java-программ для различных целей.

Существует несколько реализаций статического подхода для обнаружения гонок[7, 8]. Также существуют инструменты, такие как FindBugs [9] и ThreadSafe [10], которые проводят анализ конкурентного доступа к данным. Но все эти реализации ориентированы на поиск ошибок в программах и для оптимизации динамического детектора не подходят.

В работе [11] приведен способ для нахождения объектов, которые не покидают контекст потока или метода. Результаты данной работы используются для выделения объектов на стеке вместо кучи, а также для ликвидации ненужных операций синхронизации. Целью работы являлась оптимизация компилятора, однако метод анализа, рассмотренный в данной работе теоретически можно использовать для оптимизации динамического детектора за счет исключения из области динамического анализа отдельных экземпляров объектов. Однако применение результатов такого анализа может потребовать нетривиальных изменений самого детектора и его характеристик. Это направление является возможной областью развития данной работы.

VI. Заключение

Поиск гонок – актуальная задача верификации многопоточных программ и динамические методы являются одним из основных подходов в этой области. Одной из главных проблем динамических методов является снижение производительности анализируемой программы. В данной работе был рассмотрен вопрос оптимизации динамических детекторов, путем проведения предварительного статического анализа.

Рассмотрены некоторые наиболее типичные шаблоны обеспечения синхронизации доступа к разделяемым переменным и перспективы доказательства их корректности средствами статического анализа.

В рамках работы был разработан алгоритм для поиска корректно-синхронизированных полей. Данный алгоритм позволяет обнаруживать поля, при обращении к которым гарантированно не может возникнуть гонки. Применение результатов предложенного анализа позволяет сократить время выполнения и объем потребления памяти динамического детектора без внесения существенных изменений. Полученный алгоритм позволяет отслеживать различные операции синхронизации, используемые для защиты обращений к разделяемым переменным. Алгоритм основан на обходе графов потока исполнения методов

Разработан инструмент, реализующий предложенный алгоритм. Корректность работы программы анализа была протестирована на наборе синтетических тестов, покрывающем различные варианты условий, предусмотренных алгоритмом. Полученный инструмент предоставляет интерфейс для интеграции с динамическими детекторами. В рамках данной работы он был интегрирован с существующим динамическим детектором гонок для Java-программ.

Также результаты анализа могут быть использованы для поиска некоторых ошибок синхронизации, поскольку в результате работы инструмента выдаётся информация о критических обращениях, препятствующих признанию поля корректно синхронизированным.

Данная работа имеет хорошие перспективы для развития. Используя результаты работы и методы *escape*-анализа, планируется дальнейшая оптимизация динамического детектора за счет исключения из области динамического анализа отдельных экземпляров объектов.

[1] Трифанов В.Ю, Цителов Д.И. Динамический поиск гонок в Java-программах на основе синхронизационных контрактов. Материалы конференции «Инструменты и методы анализа программ (TMPA-2013”), Кострома, 2013. сс. 273-285

[2] Netzer R. H. B. Race Condition Detection for Debugging Shared-memory Parallel Programs. Ph.D. thesis. Madison, WI, USA: University of Wisconsin at Madison, 1991. UMI Order No. GAX91-34338.

[3] Netzer R. H. B., Miller B. P. What Are Race Conditions?: Some Issues and Formalizations // ACM Lett. Program. Lang. Syst. New York, NY, USA, 1992. Vol. 1, no. 1. P. 74–88. URL: http://doi.acm.org/10.1145/130616.130623.

[4] Soot - a Java Bytecode Optimization Framework / R. Vall´ee-Rai, P. Co, E. Gagnon et al. 2000. URL: http://dl.acm.org/citation.cfm?id=781995.782008.

[5] Bilardi G., Pingali K. Algorithms for Computing the Static Single Assignment Form // J. ACM. New York, NY, USA, 2003. Vol. 50, no. 3. P. 375–425. URL: http://doi.acm.org/10.1145/765568.765573.

[6] Soot. A framework for analyzing and transforming Java and Android Applications. URL: http://sable.github.io/soot/.

[7] Naik M., Aiken A., Whaley J. Effective Static Race Detection for Java // SIGPLAN Not. New York, NY, USA, 2006. Vol. 41, no. 6. P. 308–319. URL: http://doi.acm.org/10.1145/1133255.1134018.

[8] Chord: A Program Analysis Platform for Java. URL: http://pag.gatech.edu/chord.

[9] FindBugs. URL: http://www.ibm.com/developerworks/java/library/j-findbug1/.

[10] ThreadSafe. URL: http://www.contemplateltd.com/threadsafe.

[11] Whaley J., Rinard M. Compositional Pointer and Escape Analysis for Java Programs // SIGPLAN Not. New York, NY, USA, 1999. Vol. 34, no. 10. P. 187–206. URL: http://doi.acm.org/10.1145/320385.320400.