Санкт-Петербургский Государственный Университет

Математико-механический факультет

Кафедра системного программирования

Применение статического анализа типов потоков для разграничения доступа в многопоточных приложениях на Java

Дипломная работа студента 544 группы

Егорова Ивана Владимировича

Научный руководитель	.....	асп. А.А. Бреслав
	/подпись/	
Рецензент	.....	ст. преп. В.С. Полозов
	/подпись/	
“Допустить к защите”	.....	д.ф.-м.н., проф. А.Н. Терехов
заведующий кафедрой	/подпись/	

Санкт-Петербург

2009

Содержание

1 Введение

Современная разработка практически немыслима без использования возможностей многопоточного программирования. В данной работе внимание сконцентрировано на многопоточных приложениях на языке Java. С самого начала своего развития этот язык позиционировался как язык, предназначенный для разработки переносимых многопоточных программ. Семантика языка явно описывает модель вычислений, в частности и модель памяти, что выгодно отличает язык от таких языков как C и C++, стандарты которых долгое время не включали таких существенных деталей.

Уже в первых версиях язык и стандартная библиотека предоставляли примитивы многопоточности: классы Thread, ThreadGroup, мониторы, ассоциированные с каждым объектом, блоки синхронизации на указанных мониторах, методы wait, notify и notifyAll. В момент появления языка такая поддержка на уровне языка и стандартной библиотеки была большим, чем могли предложить другие языки. Тем не менее в обзоре (16) рассматриваются проблемы, связанные с представленной языком моделью памяти. В результате стандарт языка Java 1.5 был пересмотрен в JSR-133 ((17)). К моменту создания спецификации Java 1.5 некоторые методы реализации взаимодействия потоков были признаны устаревшими в связи с их недостатками (10). Java 1.5 предлагает разработчикам библиотеку java.util.concurrent, в деталях рассмотренную в (10) За время развития языка и стандартной библиотеки Java были предложены надежные механизмы для синхронизации потоков, однако написание корректных многопоточных программ все равно требует большого опыта и тщательности, поэтому существует обширный набор публикаций, посвященных библиотекам, реализующим высокоуровневые примитивы взаимодействия, а также средствам дополнительного инструментирования и/или анализа Java-приложений.

2 Обзор

Ниже приведен обзор известных автору подходов к анализу многопоточных Java-приложений.

В (9) предлагается библиотека актеров (actors library) для языка Scala (15; 1). Scala – язык, объединяющий в себе объектно-ориентированную и функциональную парадигмы, имеющий более мощную в сравнении с Java систему типов, компилирующийся в Java-байткод. Программы на Scala могут использовать классы, написанные на Java и наоборот. Идея библиотеки актеров берет начало в языке Erlang (19). Актер определяется как самодостаточный активный объект, управляемый посредством асинхронных сообщений. Библиотека предоставляет примитивы передачи сообщений, не гарантирует корректности обращения к данным из различных потоков, однако при применении некоторых разумных соглашений по разработке позволяет эффективно разделять данные между актерами. В качестве такого соглашения может служить договоренность после передачи сообщения не хранить в передающем потоке ссылки на переданные в сообщении объекты. В работах (12; 20) отмечается основной недостаток актеров: плохое совмещение наследования и основного цикла актера, приводящее к невозможности гарантировать выполнение контракта объекта-актера при замещении объектом класса-наследника.

Средства анализа делятся на статические и динамические. Основная задача таких средств – обнаружение состояний состязаний (race conditions) и/или тупиков (deadlocks) (21). Преимущество статических средств анализа заключается в отсутствии дополнительных расходов в процессе исполнения. Некоторые статические средства анализа применимы не только к программам, но и к библиотекам. Динамические средства, такие как (14; 5) привлекают тем, что анализ производится на основе реальных потоков управления, за счет чего в первую очередь выявляются ошибки в “рабочем наборе” приложения. В (14) для обнаружения состояний состязания используется гибридный алгоритм на основе двух типов анализа: анализа установленной блокировки (lockset-based analysis) и анализа отношения “происходит-до” (happens-

before). Входными данными для анализа являются собранные инструментированным кодом данные. Результатом анализа установленной блокировки является множество пар операторов, имеющих доступ к разделямой памяти без общего захваченного ресурса. Недостатком такого алгоритма самого по себе является большое количество ложно-положительных пар. Для фильтрации вывода используется анализ отношения “происходит-до”, определяющий неупорядоченные в смысле “происходит-до” операторы, обращающиеся к общему ресурсу. Такой анализ не имеет ложноположительных сообщений: для любой потенцильной гонки, которую нашел анализ отношения “происходит-до” существует такое расписание действий потоков, при котором гонка происходит. На практике (14) удалось найти ошибки в таких крупных приложениях как Apache Tomcat (версии 4.0.4). Однако, в общем случае динамический анализ вычислительно сложен, кроме того от ветвлений программы зависят как объемы входных данных, так и количество обнаруженных ошибок. Еще одним недостатком динамического анализа является невозможность анализа кода библиотеки.

Среди статических решений есть несколько, гарантированно (или почти гарантированно) исключающие возможность возникновения состязания. В работе (3) разработан диалект языка Java – Guava,– исключающий возможность состязания без излишних блокировок. В Guava все типы подразделяются на три категории: значения, мониторы, объекты. Переменная типа-значения хранит значение, при присваивании происходит копирование значения из одной переменной в другую. Значения по определению безопасны в отношении потоков, поскольку разделение одного значения двумя потоками невозможно в принципе. К значениям относятся простые типы и структуры. Мониторы и объекты относятся к ссылочным типам. Мониторы могут быть доступны из нескольких потоков, но обращение к их полям и методам обязательно последовательно. Ссылки на объекты не распространяются за пределы потока. Для того, чтобы гарантировать локальность ссылки на объект внутри потока, вводится понятие владения объектом. Владельцем может выступать либо поле значения, либо поле монитора. Кроме того, объект не может менять владельца или ссылаться на объект другого владельца. За счет того, что операции над мониторами всегда синхронизова-

ны, операции над объектами не требуют синхронизации: для вызова метода объекта необходимо иметь на него ссылку, которая есть только у владельца, владельцем самого верхнего уровня является монитор, который таким образом защищает поля объекта от состязания. Для контроля за распространением ссылок на объекты вводится понятие региона: каждая переменная программы принадлежит некоторому региону, все переменные, находящиеся в одном регионе имеют либо одного и того же хозяина, либо не имеют хозяина. Кроме того все методы подразделяются на локальные – не имеющие доступа к мониторам, и глобальные – имеющие доступ к таковым. Созданная на основе этих примитивов система правил обладает свойством отсутствия состязаний. Для достижения лучшей производительности язык различает методы чтения и записи, а также изменяемые и неизменяемые параметры. Методы чтения не могут вносить изменений в состояние объекта. Такое разделение позволяет использовать более эффективную блокировку чтения-записи вместо общей блокировки состояния. К недостаткам предложенного диалекта стоит отнести сравнительно сложную для использования систему типов и модификаторов доступа.

Более простые в освоении диалекты Java без состояния состязания развивались в (7; 8; 4). В теоретическом исследовании (7) предагается формальная система типов для объектно-ориентированных языков, исключающая состязания. Работа (8) является практической реализацией (7) для языка Java и предлагает диалект ConcurrentJava. Предлагается явно указывать в определении поля объекта каким монитором это поле защищено, а в определении методов указывать, какие мониторы должны быть захвачены перед вызовом. Поскольку монитор определяется статически во время компиляции, он может указывать лишь на объект, доступный от this. Это существенное ограничение, если объект не должен иметь собственной синхронизации, но должен быть частью сложной структуры, отвечающей за согласованность своего состояния. Примером такого рода объектов служит элемент словаря. Для обхода этого ограничения предлагается определять в классе так называемые мнимые поля. Мнимые поля являются параметрами типа, при определении переменной такого типа необходимо указать значения мнимых полей. Так же как и в (3) классы делятся на разделяе-

мые (thread_shared) и неразделяемые (thread_local). Обращение к полям и методам разделяемых классов возможно из любого потока. При этом все поля разделяемых классов либо защищены некоторым монитором, либо неизменяемы (final). В таких условиях легко исключить состязания требованием, чтобы доступ к защищенным полям происходил только при захваченном мониторе, равно как и легко проверить корректно ли обращение метода класса к полю, поскольку методы явно указывают какие мониторы должны быть захвачены перед их исполнением. К объектам неразделяемых классов доступ возможен только из текущего потока, гарантией этому служит то, что к объектам неразделяемых классов невозможно обратиться через поле разделяемого класса. Авторам работы (8) удалось применить свой подход не только для языка ConcurrentJava, но и для языка Java. В частности после аннотации класса java.util.Vector им удалось обнаружить одну гонку. В заключении (8) отмечается, что подход, основанный на дополнительной разметке программ является перспективным для анализа многопоточных приложений.

Еще один диалект Java - Parameterized Race-Free Java, изложенный в (4) предлагает параметризовать класс полем-владельцем. Конкретный владелец указывается при определении поля или переменной и является неизменяемым полем. В результате, в отличие от (7; 8; 3), механизм синхронизации является свойством не класса, но поля. Априорно существуют два типа владельцев: thisThread (текущий поток) и self (объект сам по себе). Для того, чтобы выполнить операцию над объектом O язык требует, чтобы самый верхний объект в иерархии владельцев объекта O имел захваченный монитор в текущем потоке. Монитор на объекте thisThread всегда неявно захвачен, что означает, что объекты, которыми владеет текущий поток не требуют дополнительной синхронизации, поскольку система типов не разрешит доступ к ним от другого владельца. В качестве расширений своей исходной системы типов авторы (4) предлагают такие интересные концепции, как уникальный объект. Для уникального объекта гарантируется, что всегда существует не более одной ссылки на него. Это достигается запретом присваивания ссылок на уникальные объекты другим объектам и введением дополнительной синтаксической конструкции, производящей одновремен-

но присваивание в левую часть и запись в правую часть значения null. Уникальные объекты удобно использовать для передачи данных между потоками. Существенными проблемами предложенного в (4) диалекта являются возможность испортить систему простым приведением типа и невозможность миграции объектов между владельцами.

ESC/Java (Extended Static Checker for Java) (6) использует специально форматированные комментарии для спецификации различных инвариантов программы, написанной на Java. Эти инварианты относятся не только к проверке гонок и тупиков, но и к другим возможным ошибкам (таким как потенциальные выходы за границы массивов или разыменование неинициализированного указателя). Для обнаружения ошибок ESC/Java на основе пользовательских аннотаций создает промежуточный код, который используется для автоматического доказательства корректности программ. Авторы не ставят своей целью ни полноту, ни точность, однако на практике утверждается эффективность метода даже в отношении зрелых проектов.

Существует так же класс средств статической проверки немодифицированного Java кода. Достаточно полный обзор таких исследований содержится в работе (18). Так в работах (2; 11) представлено средство Jlint. Jlint статически анализирует всю Java программу на наличие гонок и тупиков. При этом для поиска состояний состязания используется статический граф конкурентного доступа. О гонке сообщается при выполнении набора из пяти условий, часть из них допускает ложно-положительные результаты, а другая часть уменьшает количество как ложно-положительных, так и положительных результатов.

В работе (13), посвященной поиску состояний состязания, предложен алгоритм, комбинирующий результаты четырех алгоритмов статического анализа, каждый из которых проверяет одно из условий, при нарушении которого состязание становится невозможным. В работе использованы несколько известных, и предложено несколько новых методов статической проверки данных, которые по результатам тестов авторов за 132 минуты проверяли такой крупный проект как Apache Derby (700 тысяч строк кода).

3 Постановка задачи

Резюмируя предыдущюю часть можно утверждать, что различные средства анализа в многопоточных программах на Java становятся все более мощными. Разработка таких подходов была мотивирована необходимостью обнаружения ошибок в уже существующем коде.

В книге (10) отмечается, что зачастую в ООП имеют место неформальные и/или не задокументированные ограничения на доступ и изменение состояния объекта в многопоточном приложении. Так, во многих графических средах все задачи, связанные с выводом на экран и изменением состояния графических объектов должны осуществляться из одного потока. Однако проверить то, что метод вызывается из необходимого потока компилятор не может. Лучшее, что можно сделать, и что сделано в такой ситуации в библиотеке Swing – созданы методы invokeLater и invokeAndWait, позволяющие исполнить действие в графическом потоке. Этот метод позволит знакомому с библиотекой человеку избежать неприятностей, но никак не поможет человеку, впервые использующему Swing.

Объектно-ориентированное программирование основано на разделении сложной задачи на классы похожих друг на друга взаимодействующих объектов. При этом при проектировании для каждого поля и метода при помощи модификаторов доступа ограничивается множество методов, которые могут к ним обращаться. Мотивацией к написанию данной работы послужило то, что различные потоки исполнения программы предназначены для выполнения разных задач, причем один объект может по-разному использоваться в зависимости от того, какой поток получает к нему доступ, однако существующая система модификаторов доступа не предназначена для разграничения доступа для потоков в зависимости от исполняемых ими задач, с точки зрения компилятора Java все потоки исполнения идентичны.

В качестве примера, когда неэффективна существующая система разделения доступа, можно предложить объект-сервер (например, HTTP-сервер), который имеет методы как для осуществления пользовательских запросов (в примере с HTTP-

сервером это могут быть запросы на получение страниц), так и для осуществления диагностики (например, методы журналирования и наблюдения за нагрузкой). При этом потоку, обрабатывающему пользовательское соединение должны быть разрешены запросы к серверу, но не получение доступа к диагностической информации.

Целью данной работы является разработка механизма разграничения доступа для потоков в зависимости от типа решаемых ими задач. Отдельно отмечу, что в отличие от рассмотренных в обзоре методов, не ставится задача автоматической проверки наличия конфликтов по доступу к данным. Решение должно стать средством явного выражения разделения программы на выявленные в процессе проектирования задачи.

В части 4 рассматривается формализованная модель механизма разделения по типу решаемых задач. В части 5 приводится псевдокод алгоритма проверки программы на соответствие модели, а также приведен обзор практической реализации алгоритма проверки в среде разработки Eclipse. В части 6 на примере указаны преимущества использования метода.

4 Теоретическая часть

В этой части работы вводится понятие корректности программы в терминах задач и изложена формальная модель прав доступа, обеспечивающая таковую корректность для любой программы, удовлетворяющей модели.

4.1 Классы задач и корректность в терминах классов задач

Основным понятием подхода является *класс задач*. По сути класс задач определяет функциональность, присущую потоку. В уже упомянутом в части 3 примере можно выделить два класса задач. Потоки из одного класса задач обрабатывают запросы пользователя. К другому классу задач относятся потоки, исполняющие диагностические функции. Такое различие в классах задач позволяет накладывать ограничения на методы, которые могут исполняться в каждом из потоков. На практике классами задач являются специальным образом помеченные интерфейсы языка Java. В частности каждый класс задач является типом языка Java.

Множество всех классов задач обозначим **T**. На этом множестве задается частичный порядок, для обозначения которого порядка будем использовать знак $<$. В множестве **T** содержится выделенный элемент $ATask$ такой, что $\forall t \neq ATask \;\; t < ATask$. Кроме отношения $<$ также будут использоваться отношения равенства (выполненное на диагонали) и $\leq$.

Типы java.lang.Thread, java.lang.Runnable, java. util .Callable используются стандартной библиотекой Java для определения и запуска новых потоков, поэтому мы выделяем их особо и в дальнейшем будем называть *конструкторами задач*. Класс R, унаследованный от конструктора задач C и класса задач T, будем называть *задачей*, метод s, первым получающий управление в потоке исполнения – *точкой входа* задачи. Класс задач T будем в таком случае называть ассоциированным с задачей R.

Инструкцией будем называть предписание на выполнение одного из следующих действий:

- чтение значения поля объекта,
- запись значения поля объекта,
- вызов метода объекта (в том числе и виртуального),
- вызов статического метода (в том числе конструктора объекта),
- возврат из метода,
- запуск нового потока исполнения.

Остальные инструкции в данной модели не рассматриваются, поскольку их исполнение не ведет к конкурентному доступу к полям и методам объектов. К таким инструкциям относятся чтение/запись значения на стеке (локальной переменной или аргумента), а также инструкции ветвления, арифметических операций, обработки исключений.

Реализацией метода будем называть множество инструкций, соответствующих телу этого метода в Java-коде. Отметим, что реализации существуют у статических методов классов и методов экземпляра с телом, то есть у всех методов, кроме абстрактных. *Объявлением метода* будем называть любой заголовок метода, а именно заголовки: конструкторов, абстрактных методов, методов интерфейсов, методов экземляров с телом, статических методов. На множестве объявлений методов вводится частичный порядок, обозначаемый знаком $<$. Для двух объявленияй методов $m_D^{(1)}$, $m_D^{(2)}$ выполнено $m_D^{(2)} < m_D^{(1)}$ в том случае, если $m_D^{(2)}$ – определяет или переопределяет (override) метод $m_D^{(1)}$. Объявлением реализации метода будем называть заголовок метода, содержащего тело, соответствующее реализации. Обозначать объявление m_D реализации m_I будем $D(m_I)$.

Для пояснения введенных обозначений рассмотрим пример, приведенный в листинге 1. В приведенном примере существуют объявления следующих методов: IPerson.getName, Employee.getName, Manager.getName, Employee.<init>,

Manager.<init>. Будем обозначать объявление метода m посредством m_D, например IPerson.getName$_D$. Приведенный код вводит отношения Manager.getName$_D$ $<$ Employee.getName$_D$ $<$ IPerson.getName$_D$. Отдельно обратим внимание на то, что конструкторы типов Employee и Manager не связаны отношением $<$. В примере следующие методы имеют реализацию: Employee.getName, Manager.getName, Employee.<init>, Manager.<init>. Реализацию метода m будем обозначать m_I. Так Manager.<init>$_I$ находится в строках 18-19 листинга, D(Manager.<init>$_I$) = Manager.<init>$_D$.

```
1  interface IPerson {
2      String getName();
3  }
4
5  class Employee implements IPerson {
6      private final String name;
7      Employee(String name) {
8          this.name = name;
9      }
10     String getName() {
11         return name;
12     }
13 }
14
15 class Manager extends Employee {
16     private final boolean male;
17     Manager(boolean isMale, String name) {
18         super(name);
19         this.male = isMale;
20     }
21
22     @Override
23     String getName() {
24         return (male ? "Mr." : "Mrs") + name;
25     }
26 }
```

Листинг 1: Объявление и реализация

В описываемой модели каждая реализация метода помечена некоторым набором классов задач, называемым *ограничением на исполнение реализации метода*, а каждое поле каждого класса помечено двумя наборами, называемыми *ограничением на чтение значения* и *ограничением на запись значения*. Суть ограничений заключается

в спецификации тех потоков, которым разрешена определенная операция: в случае ограничения на исполнение реализации метода – вызов метода, в случае ограничения на чтение/запись поля – чтение/запись значения определенного поля объектов некоторого класса. Более формально, на множестве $\mathbf{M}_I$ всех реализаций методов, встречающихся в программе, определена функция $E : \mathbf{M}_I \to 2^{\mathbf{T}}$. Отметим, что поскольку каждой реализации метода m_I соответствует объявление метода $D(m_I)$, то в дальнейшем считается, что функция определена на подмножестве множества $\mathbf{M}_D$ всех объявлений методов программы. Аналогично на множестве $\mathbf{F}$ определений всех полей в программе задаются отображения $R, W : \mathbf{F} \to 2^{\mathbf{T}}$, при этом каждое поле идентифицируется парой, состоящей из имени класса и имени поля.

Операцией называется результат исполненния инструкции. При этом для инструкции вызова виртуального или абстрактного метода операцией является вызов конкретной (с учетом типа времени исполнения) реализации метода. Так в листинге 2 в строке 3 присутствует инструкция вызова IPerson.getName$_D$ из листинга 1, результатом исполнения этой инструкции является вызов реализации Employee.getName$_I$.

```
1 public static void main(String[] args) {
2     IPerson person = new Employee(args[0]);
3     System.out.println(person.getName());
4 }
```

Листинг 2: Инструкция и операция

Для определения корректности программы в терминах классов задач введем отношение $\ll$ и рассмотрим множество всевозможных последовательностей операций каждой из задач, определенных в программе, в отдельности.

$$\ll \subset \mathbf{T} \times 2^{\mathbf{T}}, \quad c \ll Q \Leftrightarrow \exists q \in Q : c < q$$

Перебор последовательностей ведется по всем возможным ветвлениям в процессе исполнения задачи. Источниками ветвлений являются условия циклов, условные переходы и инструкции вызова виртуальных методов. Конкретная последовательность $\mathbf{O}$ операций (возможно бесконечная), допустимая при некоторой последовательности выборов в точках ветвления в процессе исполнения задачи R из класса задач T, на-

зывается *корректной в терминах классов задач*, если выполнены все условия:

1. Для каждой операции вызова метода m_I из $\mathbf{O}$ выполнено отношение $T \ll E(D(m_I))$
2. Для каждой операции чтения поля f класса C из $\mathbf{O}$ выполнено отношение $T \ll R(C, f)$
3. Для каждой операции записи поля f класса C из $\mathbf{O}$ выполнено отношение $T \ll W(C, f)$

Задача называется *корректной в терминах классов задач*, если корректны все возможные последовательности операций. Программа называется *корректной в терминах классов задач*, если все задачи корректны в терминах классов задач. В дальнейшем тексте под корректностью будет пониматься именно корректность в терминах классов задач, если иное не указано явно.

4.2 Дополнительные модификаторы доступа и ограничения, накладываемые на них, достаточные для корректности

Далее будет введена модель модификаторов доступа, достаточная для корректности в том смысле, что корректность программы в терминах, введенных данной моделью модификаторов доступа, влечет корректность программы в терминах классов задач.

В рассматриваемой модели явно специфицируются значения R, W для полей классов. Область определения функции E расширяется таким образом, чтобы включать все определениия методов программы. При этом значение функции E на некотором элементе множества $\mathbf{M}_I$ определяет ограничения на исполнение метода.

Кроме того модель вводит несколько дополнительных понятий. *Частичной задачей* называется тип, являющийся конструктором задач или унаследованный от некоторого конструктора задач. Про переменные, имеющие типы частичной задачи, также будем говорить, что они являются частичными задачами. Среди всех полей и

формальных параметров программы, являющихся частичными задачами, выделяется некоторое подмножество *M полей и формальных параметров с ограниченным доступом*. В листинге 3 переменные $t1$, $t2$ являются частичными задачами, поскольку первая имеет тип, являющийся конструктором задач (java.lang.Runnable), а тип второй унаследован от конструктора задач. Поле $t3$ кроме того является полем с ограниченным доступом, поскольку помечено аннотацией @ThreadStarter (подробнее это изложено в части 5).

```
1 interface MyRunnable extends Runnable {}
2 class TasksExample {
3     Runnable t1;
4     MyRunnable t2;
5
6     @ThreadStarter
7     Runnable t3;
8 }
```

Листинг 3: Частичные задачи и ограниченный доступ

Модель накладывает ограничения на возможные действия программы.

Ограничение 4.1 (Ограничение на точку входа задачи)**.** Для точки входа s задачи R из класса задач T всегда выполнено $E(s) = \{T\}$.

Для определения следующих ограничений будет введено отношение $\preceq \subset 2^{\mathbf{T}} \times 2^{\mathbf{T}}$:

$$P \preceq Q \Leftrightarrow \forall p \in P \;\; \exists q \in Q : p \leq q$$

Ограничение 4.2 (Ограничение на вызов метода)**.** Ограничение на вызов метода распространяется на все инструкции вызова. Пусть такая инструкция находится в теле метода m_I, и вызывает метод с объявлением c_D. Тогда модель требует, чтобы было выполнено соотношение $E(D(m_I)) \preceq E(c_D)$.

Ограничение 4.3 (Ограничение на реализации методов)**.** Ограничение на реализации методов связано с тем, что инструкция вызова метода m_D во время исполнения может повлечь вызов любого из методов из множества $\{m_I^* : D(m_I^*) \leq m_D\}$, и определяется следующим образом:

$$\forall s_D : s_D \leq m_D \;\; E(m_D) \preceq E(s_D)$$

Этому ограничению не удовлетворяют точки входа задач.

Здесь стоит сделать ремарку относительно смысла накладываемых на методы ограничений. Помечая метод некоторым набором классов задач, мы вводим ограничение на те потоки, в которых может исполняться метод. Ограничения 4.2, 4.3 гарантируют, что если в потоке разрешено исполнение реализации метода $m_I^{(1)}$, то также разрешено исполнение и реализации любого метода $m_I^{(2)}$, который может быть вызван посредством исполнения инструкции вызова объявления m_D^*. В примере с листингом 2 это означает, что потоку, в котором разрешено исполнение реализации метода *main* разрешено также исполнение любой реализации метода, соответствующей одному из объявлений IPerson.getName$_D$, PrintStream.println$_D$, Employee.<init>$_D$. Соответствующими считаются те реализации методов, которые могут быть исполнены в результате исполнения инструкции вызова объявления.

Ограничение 4.4 (Ограничение на доступ к данным)**.** Ограничение на доступ к данным призвано разграничить данные между логически различными задачами, исполняемыми программой. В модели различаются доступ на чтение и на запись. Реализация метода m_I может иметь доступ к полю (C, f) на чтение, если имеет место отношение $E(D(m_I)) \preceq R(C, f)$. Реализация m_I может иметь доступ к полю (C, f) на запись, если имеет место отношение $E(D(m_I)) \preceq W(C, f)$.

Ограничение 4.5 (Ограничение на приведение типов)**.** Ограничение на приведение типов и разыменование частичных задач состоит из трех частей. Во-первых, поля и формальные параметры методов с ограниченным доступом, не могут быть разыменованы, к их полям и методам невозможно обращение. Существует две операции, допустимые в отношении помеченной частичной задачи: присвоение другому полю или формальному параметру с ограниченным доступом и передача частичной задачи в качестве входной точки нового потока. Во-вторых, приведение типа задачи с потерей информации о классе задач возможно только в том случае, когда это значение присваивается полю или формальному параметру с ограниченным доступом. В-третьих, приведение типа с приобретением информации о классе задач, недопустимо.

В листинге 4 приведены примеры действия последнего ограничения. Здесь в строках 28, 31, 34 происходит приведение типа с потерей информации о классе задач – во всех случаях PrinterManager приводится к Runnable. В строках 28 и 34 результат приведения передается формальному параметру с ограниченным доступом, поэтому код корректен. В строке 31 результат передается в локальную переменную с неограниченным доступом, поэтому такое присваивание некорректно. В строке 32 происходит приведение типа с приобретением класса задач TPrinterManager, что недопустимо. Присваивание в строке 6 корректно, равно как и передачи параметров в строках 28, 35, поскольку значения из полей и параметров с ограниченным доступом присваиваются в поля и параметры с ограниченным доступом. Обращение к методу getModel() в строке 29 также корректно, поскольку переменная manager не является переменной с ограниченным доступом. Строки 22-23 корректны, поскольку в них происходит передача параметра с ограниченным доступом в качестве входной точки нового потока.

```
1  class Wrapper {
2      @ThreadStarter
3      public final Runnable task;
4
5      public Wrapper(@ThreadStarter Runnable task) {
6          this.task = task;
7      }
8  }
9
10 @ThreadMarker
11 interface TPrinterManager {}
12
13 class PrinterManager implements Runnable, TPrinterManager {
14     ...
15     public String getModel() {
16         ...
17     }
18 }
19
20 class Example {
21     public static startPrinterManager(@ThreadStarter Runnable
           manager) {
22         Thread t = new Thread(manager);
23         t.start();
24     }
25
```

```
26      public static void main(String[] args) {
27          PrinterManager manager = new PrinterManager();
28          startPrinterManager(manager);
29          System.out.println("Printer model: " +
                manager.getModel());
30
31          Runnable r = manager;
32          System.out.println("Printer model: " +
                ((PrinterManager)r).getModel());
33
34          Wrapper w = new Wrapper(manager);
35          startPrinterManager(w.task);
36      }
37  }
```

Листинг 4: Ограничение на приведение типов

Утверждение 4.1 (Корректность модели)**.** *Если программа соответствует правилам 4.1-4.5 модели, то программа корректна.*

Для доказательства этого утверждения понадобится следующая лемма.

Лемма 4.1 (О свойствах отношений $\preceq$, $\ll$)**.** *Имеют место следующие утверждения:*

1. $P_1 \preceq P_2, P_2 \preceq P_3 \Rightarrow P_1 \preceq P_3$
2. $t \ll P_1, P_1 \preceq P_2 \Rightarrow t \ll P_2$

Доказательство леммы. 1. Для доказательства этой части утверждения мы покажем, как по некоторому $p_1 \in P_1$ выбрать $p_3 \in P_3$, такой что $p_1 \leq p_3$. В самом деле из $P_1 \preceq P_2$ следует, что существует элемент $p_2^* \in P_2$, такой что $p_1 \leq p_2^*$, а из $P_2 \preceq P_3$ следует, что для $p_2^* \in P_2$ существует $p_3^* \in P_3$ такой, что $p_2^* \leq p_3^*$. Элемент p_3^* и является искомым элементом p_3, поскольку по свойствам частичного порядка из $p_1 \leq p_2^*$, $p_2^* \leq p_3^*$ следует $p_1 \leq p_3^*$.

2. Необходимо доказать, что $\exists p_2 \in P_2 : t < p_2$. Из $t \ll P_1$ следует, что существует $p_1^* \in P_1$ такой, что $t < p_1^*$, из $P_1 \preceq P_2$ следует $\exists p_2^* \in P_2 : p_1^* \leq p_2^*$. p_2^* и есть искомый элемент p_2.

□

Доказательство утверждения. В условиях утверждения необходимо доказать, что для каждой задачи R из класса задач T выполнены условия коррекности.

Рассмотрим некоторую последовательность операций задачи R. С каждым префиксом длины l этой последовательности однозначно ассоциируется стек вызванных методов $\mathfrak{S} = [m_I^{(1)}, \ldots, m_I^{(depth(l))}]$ следующим образом: префикс читается слева направо и

- каждая операция вызова реализации метода m_I добавляет m_I на вершину стека
- каждая операция возврата снимает верхний элемент со стека

Докажем, что отношение

$$T \ll E(D(top(\mathfrak{S}))) \ , \quad \text{где } top(\mathfrak{S}) \text{ – вершина стека}$$

всегда верно, используя индукцию по длине префикса.

База Во входной точке потока s стек имеет вид $\mathfrak{S} = [s]$. Согласно модели $E(s) = \{T\}$, что немедленно влечет верность базы.

Переход Пусть известно, что для стеков $\mathfrak{S}_0, \ldots, \mathfrak{S}_l$, соответствующих префиксам длины до l, утверждение верно. Рассмотрим операцию *op* на $(l+1)$-й позиции. Если эта операция отлична от операций вызова и возврата, то стек не изменяется, и утверждение верно для $\mathfrak{S}_{l+1}$. Если *op* является операцией возврата, то либо в стеке более одного элемента, тогда $\exists m < l : \mathfrak{S}_m = \mathfrak{S}_{l+1}$ и переход доказан, либо *op* – последняя операция в потоке, поскольку снимает со стека метод s. Если *op* – вызов реализации метода m_I, то реализация $c_I = top(\mathfrak{S}_l)$ имеет в своем теле на месте *op* инструкцию вызова i_D такой, что $D(m_I) \leq i_D$, что в силу ограничения 4.2 означает верность $E(c_D) \preceq E(i_D)$, в силу ограничений 4.3 и 4.5 выполнено отношение $E(i_D) \preceq E(D(m_I))$. Тогда в силу леммы 4.1 выполнено $E(c_D) \preceq E(D(m_I))$, а поскольку в силу индукционного предположения выполнено $T \ll E(c_D)$, в силу той же леммы имеет место $T \ll E(D(m_I))$.

Докажем теперь, что для любой операции чтения поля в последовательности выполнения выполнено отношение $T \ll R(C, f))$. Рассмотрим реализацию метода

$c_I = top(\mathfrak{S}_r pos)$, где $rpos$ – позиция операции. Именно реализация метода c_I производит чтение поля (C, f), что в силу ограничения 4.4 влечет верность отношения $E(D(c_I)) \preceq R(C, f)$. Из предыдущей части доказательства известно, что выполнено $T \ll E(D(c_I))$, что в силу леммы 4.1 влечет верность $T \ll R(C, f)$.

Доказательство для операции записи почти дословно повторяет приведенное рассуждение для операции чтения.

Таким образом доказано, что для любой возможной последовательности действий задачи R из класса задач T выполнено условие её корректности, что означает корректность самой задачи. Поскольку нигде не были использованы какие-либо свойства R, рассуждение применимо для всех задач, а значит программа, удовлетворяющая условию, является корректной. □

В этой части введено понятие корректности программы в терминах классов задач, изложена модель модификаторов доступа Java-программы, и доказано, что корректная с точки зрения введенной модели программа корректна в терминах классов задач. В следующем разделе приведен алгоритм, который позволяет проверять соответствие программы правилам дополнительной разметки, не требуя при этом полного анализа кода.

5 Реализация

Для практической реализации рассмотренной модели для языка Java были использованы возможности системы типов и определения пользовательских аннотаций. Все аннотации, используемые для разметки программ, располагаются в пакете com.google.code.annatasha.annotations, в дальнейшем имя пакета в именах этих аннотаций будем опускать.

- Для определения пользовательских классов задач используются интерфейсы, помеченные аннотацией @ThreadMarker. Класс задач $ATask$ не имеет соответствующего ей типа в Java. Определенный пользователем класс задач не должен иметь методов. Все супер-типы интерфейса I, определяющего класс задач, обязаны быть классами задач. Если класс задач I не имеет супер-типов, то класс задач считается прямым наследником класса задач $ATask$ ($I < ATask$), в противном случае, если указаны супер-типы $S_1, \ldots, S_n$, то вводится система отношений

$$\left.\begin{array}{l} I < S_1, \\ \ldots \\ I < S_n. \end{array}\right\} \tag{1}$$

  Отсутствие циклов наследования в данном случае гарантирует система типов языка Java.

- Для определения задачи необходимо определить класс, унаследованный ровно от одного класса задач и одного конструктора задач. Этот класс должен реализовывать точку входа задачи.

- Все объявления методов, кроме точек входа задачи, могут явно специфицировать значение функции E при помощи аннотирования заголовка метода посредством @ExecPermissions($\{t_1, \ldots, t_n\}$). Если такая аннотация присутствует при объявлении метода m_D, то $E(m_D) = \{t_1, \ldots, t_n\}$, иначе $E(m_D) = \{ATask\}$

- Значения функций R, W на поле f класса C задаются при помощи аннотаций @ReadPermissions($\{t_1^{(r)}, \ldots, t_{n_r}^{(r)}\}$), @WritePermissions($\{t_1^{(w)}, \ldots, t_{n_w}^{(w)}\}$). Множества $\{t_1^{(r)}, \ldots, t_{n_r}^{(r)}\}$, $\{t_1^{(w)}, \ldots, t_{n_w}^{(w)}\}$ являются значениями $R(C, f)$ и $W(C, f)$ соответственно. Если какая-то из аннотаций не указана, значение соответствующей функции принимается равным $\{ATask\}$.

- Для того, чтобы включить поле или формальный параметр в множество M полей и формальных параметров с ограниченным доступом, необходимо пометить его при помощи @ThreadStarter.

5.1 Алгоритма проверки корректности модификаторов доступа

Отметим, что для проверки каждого оператора языка Java на соответствие ограничениям 4.1-4.5 достаточно знать заголовок текущего метода и заголовки всех типов, участвующих в выражениях этого метода. Поскольку при компиляции программ на языке Java аннотации могут сохраняться в бинарном файле, вся заголовочная информация типов и методов при компиляции сохраняется. Отсюда, во-первых, следует, что единицей проверки кода является отдельный метод класса. Во-вторых, если пользовательская программа использует некоторую библиотеку, ранее проверенную на соответствие модели, то для проверки всей программы достаточно иметь заголовочную информацию из библиотеки и проверить на корректноть разметки методы, не входящие в библиотеку.

В листинге процедуры 1 приведен псевдокод алгоритма проверки метода.

Сначала проверяется заголовок метода на соответствие ограничению 4.3. Для этого метод Check-Method-Header (процедура 2) перебирает все объявления r_D, такие что $D(m_I) < r_D$ и проверяет выполнение отношения $E(r_D) \preceq E(D(m_I))$.

После проверки заголовка вызывается функция Build-Thread-Starter-Pool (процедура 3), которая возвращает множество выражений с ограниченным доступом, к которым относятся:

Процедура 1 Check-Method

Check-Method(m)

▷ Параметры: $m \in \mathbf{M}_I$

▷ проверка на соответствие ограничению 4.3

1 Check-Method-Header(m)

▷ создаем пул выражений, имеющих ограничение на доступ

2 $marked \leftarrow$ Build-Thread-Starter-Pool(m)

▷ проверяем корректность доступа к полям

3 Check-Read-Write-Access($m, marked$)

▷ проверяем корректность вызовов методов

4 Check-Method-Calls($m, marked$)

- формальные параметры с ограниченным доступом,
- обращения к полям, помеченным аннотацией @ThreadStarter,
- приведения типов с потерей класса задач.

Для обозначения выражения используется тип *Expression*. Его внутренняя структура не отражена в листингах процедур, она должна хранить информацию, однозначно идентифицирующую выражение, например, смещение в файле и длину в символах. Кроме того, в выражениях обращения к полю требуется информация о том, на чтение или на запись происходит доступ. Предикаты Is-Read-Access, Is-Write-Access, определенные на объектах типа *Expression*, предназначены для получения этой информации.

После построения множества выражений с ограниченным доступом происходит проверка корректности всех выражений. Check-Read-Write-Access (процедура 4)

Процедура 2 CHECK-METHOD-HEADER

CHECK-METHOD-HEADER(m)

▷ Параметры: $m \in \mathbf{M}_I$

1 $p \leftarrow E(m)$
2 **for** $r_D \in \mathbf{M}_D$
3 **do**
4 **if** $m < r_D$ **and not** $E(r_D) \preceq E(m)$
▷ m – (пере)определение метода r_D
▷ не удовлетворяет отношению $\preceq$
5 **then error** "нарушено отношение $\preceq$"

проверяет выражения на соответствие ограничению 4.4. CHECK-METHOD-CALLS (процедура 5) проверяет выражения на соответствие ограничению 4.2.

Вызов CHECK-CASTS (процедура 6) необходим для проверки соответствия выражений ограничению 4.5 в части, касающейся приведения с приобретением класса задачи.

5.2 Практическая реализация

Для реализации прототипа системы проверки создан проект Annatasha (http://code.google.com/p/annatasha), созданный с использованием библиотеки JDT, входящей в состав Eclipse IDE. Проект состоит из двух подключаемых модулей Eclipse IDE. Первый из них (com.google.code.annatasha.validator.core, в дальнейшем будем называть его "модуль core") реализует алгоритм проверки корректности программы, второй (com.google.code.annatasha.validator.nui, для краткости будем обозначать его "модуль nui") предоставляет расширения интерфейса пользователя среды Eclipse, интегрирующие программу проверки в среду разработки Java.

При использовании Eclipse IDE пользователь работает в некотором *окруже-*

нии (workspace). Это окружение содержит разрабатываемые пользователем *проекты* (project). Каждый проект имеет некоторое множество *проявлений* (nature, в единственном числе мы также будем употреблять термин *природа*). Природа проекта определяет правила работы с ним. Так проекты на языке Java имеют природу org.eclipse.jdt.core.javanature. Для тех проектов, которые используют Annatasha для проверки корректности дополнительной разметки, описанной в данной работе, определяется природа com.google.code.annatasha.validator.core.nature. Модуль core постоянно отвечает за следующее:

- хранение информации о дополнительной разметке символов (полей, методов, типов) проектов, обладающих природой com.google.code.annatasha.validator.core.nature,
- поддержание актуальности информации о дополнительной разметке символов,
- хранение и поддержание в актуальном состоянии информации об операциях чтения, записи, вызова и привдениях типов в телах методов,
- проверку корректности дополнительной разметки символов и операций методов.

Для поддержания актуальности информации и реализации проверок корректности модуль регистрирует в среде Eclipse дополнительный сборщик (builder) com.google.code.annatasha.validator.core.AnnatashaBuilder. Объект этого класса отвечает за инкрементную сборку актуальной информации о разметке.

Сборщик заново сканирует изменившиеся файлы проекта и в зависимости от этого изменяет хранящуюся информацию о разметке символов и осуществляемых операциях. Для сканирования исходного кода используется класс org.eclipse.jdt.core.dom.ASTParser из библиотеки JDT. Этот класс строит AST файла, содержащего Java-код, и отдает это дерево через интерфейс обратного вызова. В качестве реализации интерфейса обратного вызова используется класс com.google.code.annatasha.validator.internal.build.SourceFileRequestor, который иници-

ирует проход по дереву синтаксического разбора. В процессе разбора происходит обновление информации о разметке символов.

Кроме изменившихся файлов исходного Java кода производится частичный анализ скомпилированных файлов: заново сканируются только class-файлы, содержащие определения, на которые есть ссылки в исходных файлах, и только в том случае, когда такой файл изменился с момента последней сборки. В скомпилированных файлах не осуществляется проверка кода методов на соответствие модели. Считается, что поставляемые библиотеки были на стадии разработки проверены на соответствие. В части, касающейся стандартных библиотек, не осуществляющих обратные вызовы клиентского кода, это предположение подтверждается, поскольку все методы и данные можно считать помеченными множеством классов задач { ATask }, что удовлетворяет модели.

Собранная информация представляется в виде объектов, содержащих информацию о разметке символов, а также объектов, содержащих сведения об операциях, осуществляемых в методах проверяемого кода. После стадии сборки осуществляется полная проверка всех этих сведений на соответствие требованиям 4.1-4.5, а также требованиям реализации, изложенным в начали части 5.

Процедура 3 BUILD-THREAD-STARTER-POOL

BUILD-THREAD-STARTER-POOL(m)

▷ Параметры: $m \in \mathbf{M}_I$

▷ Возвращаемый тип: *Expression*[]

▷ перебираем формальные параметры с ограничением доступа

1 **for** $f \in$ THREAD-STARTER-FORMAL-PARAMETERS(m)

2 **do**

3 $result \leftarrow result \cup \{f\}$

▷ перебираем все обращения к полям с ограниченным доступом

4 **for** $(e.f) \in$ EXPRESSIONS(m)

5 **do**

6 **if** M(TYPE-OF(e), f)

7 **then**

8 $result \leftarrow result \cup \{(e.f)\}$

▷ перебираем все явные и неявные приведения типов

9 **for** $(from, to, e) \in$ CASTS(m)

10 **do**

11 **if** IS-PARTIAL-TASK(to) **and** IS-TASK($from$)

12 **then**

▷ Запоминаем выражение, содержащее

▷ приведение типов с потерей класса задач

13 $result \leftarrow result \cup \{e\}$

14 **return** $result$

Процедура 4 Check-Read-Write-Access

Check-Read-Write-Access(m, marked)

▷ Параметры: $m \in \mathbf{M}_I, marked \in Expression[]$

```
1  p ← E(m)
   ▷ перебираем все обращения к полям
2  for (e.f) ∈ Expressions(m)
3        do
4           if e ∈ marked ▷ выражение e имеет ограниченный доступ
5              then error "обращение к полю невозможно"
6           t ← Type-Of(e)
7           if Is-Read-Access(e.f) and not R(t, f)⪯p
8              then error "некорректное чтение"
9           t ← Type-Of(e)
10          if Is-Write-Access(e.f) and not W(t, f)⪯p
11             then error "некорректная запись"
```

Процедура 5 Check-Method-Calls

Check-Method-Calls($m, marked$)

▷ Параметры: $m \in \mathbf{M}_I, marked \in Expression[]$

1 $p \leftarrow E(m)$

▷ перебираем все вызовы методов

2 **for** $(e.c(a_1, \ldots, a_{n(c)})) \in$ Expressions(m)

3 **do**

4 **if** $e \in marked$ ▷ выражение e имеет ограниченный доступ

5 **then error** "вызов метода невозможен"

6 $t \leftarrow$ Type-Of(e)

7 **if not** $E(t, c) \preceq p$

8 **then error** "некорректный вызов метода"

Процедура 6 Check-Casts

Check-Casts(m)

▷ Параметры: $m \in \mathbf{M}_I$

▷ перебираем все явные и неявные приведения типов

1 **for** $(from, to, e) \in$ Casts(m)

2 **do**

3 **if** Is-Task(to)

4 **then**

5 **error** "некорректное приведение типов"

---

6 Пример использования

В предыдущих частях была рассмотрена формальная система ограничения доступа на основе информации о классах задач. В этой части приведен пример приложения, разработанного с учетом модели, и продемонстрированы те аспекты реализации, которые не выразимы формально в рамках существующего языка Java.

В качестве примера используется клиент-серверное приложение, осуществляющее операции над матрицами по запросу пользователя. Сервер должен принимать запросы и обрабатывать их, по возможности разбивая задачу на несколько потоков исполнения и иметь механизм управления посредством графической консоли. Клиент представляет собой приложение, отправляющее на сервер запросы на сохранение, загрузку и перемножение ранее сохраненных матриц с сохранением результата в некоторой матрице.

Основная логика работы как клиента, так и сервера содержится в пакетах с префиком ru.spbu.math.m04eiv.maths.common. Протокол (Protocol) инкапсулирует всю работу по передаче данных, создает слушателя и имеет метод для записи команды в сокет.

```
1  public class Protocol {
2      ...
3      @ExecPermissions(TProtocolRunner.class)
4      public void start() {
5          ...
6      }
7      @ExecPermissions(TCommandWriter.class)
8      public void writeCommand(Command command) {
9          ...
10     }
11 }
```

Листинг 5: Protocol.java

Метод start() запускает слушателя сетевого соединения, а метод writeCommand(Command command) предназначен для отправки команды удаленной стороне. Оба метода имеют ограничения на потоки, которые могут их исполнять. Так, writeCommand(Command command) может исполняться *только*

в потоках, имеющих класс задач TCommandWriter. Для того, чтобы отследить в процессе исполнения каких задач может быть вызван метод writeCommand достаточно определить классы, унаследованные от интерфейса TCommandWriter. Такая информация даёт разработчику дополнительные (в сравнении с обычным Java-кодом) сведения об архитектуре системы, поскольку определяет не только интерфейс метода, но и способ его использования в данной программе. В приведенном примере задача TaskRunner реализует интерфейс TCommandWriter. Поскольку TaskRunner служит для обработки поступающих команд, для разработчика дополнительная разметка означает, что ответ посылается в том же потоке, в котором обрабатывается команда, альтернативой чему могла бы служить, скажем, общая очередь отправляемых команд с отдельным потоком, отвечающим за непосредственно передачу.

Один важный, хотя и частный, случай использования заключается в ограничении доступа к данным и методам, предназначенным для исполнения в задаче, являющейся гарантировано единственной в рамках приложения. К таким задачам часто относится диспетчеризация уникального ресурса (например дисплея), поэтому в дальнейшем будем называть её потоком-диспетчером. Предложенная модель не имеет возможности и не ставит целью доказательство единственности задачи, однако позволяет безопасно избежать излишней синхронизации тогда, когда единственность задачи гарантирована. Безопасность в данном случае заключается в невозможности случайно получить доступ к данным, которыми управляют методы, исполняющиеся в потоке-диспетчере.

В рассматриваемом примере сервер Server имеет метод getManager(), который возвращает объект, предназначенный для управления обрабатываемыми запросами (см. листинг 6).

```
1 public final class Server implements Runnable, TServer {
2     ...
3
4     @ExecPermissions(TServerController.class)
5     public WorkersManager getManager() {
6         return manager;
7     }
8 }
```

Листинг 6: Server.java

Если гарантировать, что существует ровно одна задача из класса TServerController, то отпадает необходимость в дополнительной синхронизации метода getManager(). В случае часто вызываемых методов отсутствие дополнительной синхронизации может оказаться существенным с точки зрения производительности.

Резюмируя содержание раздела, можно подчеркнуть, что предложенный подход, в отличие от классической модели разграничения доступа, используемой при разработке с использованием объектно-ориентированного подхода, определяет не инкапсуляцию данных приложения, а доступные правила взаимодействия различных объектов, формально не выразимые при помощи стандартных средств языка Java. В рамках рассматриваемой работы разработано приложение, демонстрирующее сильные стороны предложенного подхода.

7 Заключение

В работе предложен подход к проектированию и реализации многопоточных программ на языке Java. В основе подхода лежит концепция классов задач, предназначенная для явного выражения различий потоков в зависимости от их предназначения. Информация о типе потока используется для ограничения доступа потока к полям и методам объектов. В отличие от известных работ предложенный подход позволяет проектировать в терминах ограничения возможностей потоков по обращению к методам и даёт гарантию того, что метод вызывается только в тех потоках, в которых это целесообразно с точки зрения проектировщика.

Разработана формальная модель классов задач и введено понятие корректности программы в терминах классов задач. Предложена модель разметки Java-программы, и доказано, что корректно размеченная программа корректна в терминах типов задач. Предложен алгоритм проверки корректности разметки программ, единицей проверки которого является отдельный метод класса. Разработана реализация этого алгоритма с интеграцией диагностики в среду разработки Eclipse.

В качестве примера применения подхода разработано клиент-серверное приложение. Произведенный анализ выявил несколько общих ситуаций, когда подход позволяет формально выражать ограничения, которые в противном случае остались бы неформальным соглашением, невыразимым на языке Java.

Одно из направлений дальнейших исследований заключается в модификации введенной модели разметки программ с тем, чтобы гарантировать более сильные свойства, такие как отсутствие состояний состязания. Другое направление может заключаться в исследовании возможности декларативной записи модели исполнения задач с тем, чтобы задача стала столь же явным и по возможности компактно описанным примитивом языка как класс.

Список литературы

[1] An overview of the Scala programming language / M. Odersky, P. Altherr, V. Cremet et al. // *LAMP-EPFL.* — 2004.

[2] *Artho, C.* Applying static analysis to large-scale, multi-threaded Javaprograms / C. Artho, A. Biere // Software Engineering Conference, 2001. Proceedings. 2001 Australian. — 2001. — Pp. 68–75.

[3] *Bacon, D.* Guava: A dialect of Java without data races / D. Bacon, R. Strom, A. Tarafdar // Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications / ACM New York, NY, USA. — 2000. — Pp. 382–400.

[4] *Boyapati, C.* A parameterized type system for race-free Java programs / C. Boyapati, M. Rinard // *ACM SIGPLAN Notices.* — 2001. — Vol. 36, no. 11. — Pp. 56–69.

[5] *Christiaens, M.* TRaDe, a topological approach to on-the-fly race detection in java programs / M. Christiaens // Proceedings of the 2001 Symposium on Java TM Virtual Machine Research and Technology Symposium-Volume 1 table of contents / USENIX Association Berkeley, CA, USA. — 2001. — Pp. 15–15.

[6] Extended static checking for Java / C. Flanagan, K. Leino, M. Lillibridge et al. `http://www.eecs.umich.edu/~bchandra/courses/papers/Flanagan_ESC.pdf`.

[7] *Flanagan, C.* Object types against races / C. Flanagan, M. Abadi // *Lecture notes in computer science.* — 1999. — Pp. 288–303.

[8] *Flanagan, C.* Type-based race detection for Java / C. Flanagan, S. Freund. `http://www.cs.cornell.edu/courses/cs711/2005fa/papers/ff-pldi00.pdf`.

[9] *Haller, P.* Event-based programming without inversion of control / P. Haller, M. Odersky // *Lecture Notes in Computer Science.* — 2006. — Vol. 4228. — P. 4.

[10] Java concurrency in practice / B. Goetz, T. Peierls, J. Bloch et al. — Addison-Wesley Reading, MA, 2006.

[11] *Knizhnik, K.* Jlint manual. — 2002.

[12] *Matsuoka, S.* Analysis of inheritance anomaly in object-oriented concurrent programming languages / S. Matsuoka, A. Yonezawa. — 1993.

[13] *Naik, M.* Effective static race detection for Java / M. Naik, A. Aiken, J. Whaley // Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation / ACM New York, NY, USA. — 2006. — Pp. 308–319.

[14] *O'Callahan, R.* Hybrid dynamic data race detection / R. O'Callahan, J. Choi // Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming / ACM New York, NY, USA. — 2003. — Pp. 167–178.

[15] *Odersky, M.* The Scala experiment: can we provide better language support for component systems? / M. Odersky // Annual Symposium on Principles of Programming Languages / ACM New York, NY, USA. — 2006. — Pp. 166–167.

[16] *Pugh, W.* Fixing the Java memory model / W. Pugh // Proceedings of the ACM 1999 conference on Java Grande / ACM New York, NY, USA. — 1999. — Pp. 89–98.

[17] *Pugh, W.* Java Specification Request 133 / W. Pugh. — 2004. `http://jcp.org/en/jsr/detail?id=133`.

[18] *Rutar, N.* A comparison of bug finding tools for Java / N. Rutar, C. Almazan, J. Foster // Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on. — 2004. — Pp. 245–256.

[19] *Virding, R.* Concurrent programming in ERLANG (2nd ed.) / R. Virding, C. Wikström, M. Williams; Ed. by J. Armstrong. — Hertfordshire, UK, UK: Prentice Hall International (UK) Ltd., 1996.

[20] *Von Behren, R.* Why events are a bad idea / R. Von Behren, J. Condit, E. Brewer // *Lihue, Hawaii USA*. — 2003.

[21] *Таненбаум, Э.* Современные операционные системы. 2-е изд. / Э. Таненбаум. — СПб: Питер, 2005.