Трифанов Виталий Юрьевич

УДК 004.451.2

ОБНАРУЖЕНИЕ СОСТОЯНИЙ ГОНКИ В JAVA-ПРОГРАММАХ НА ОСНОВЕ СИНХРОНИЗАЦИОННЫХ КОНТРАКТОВ

Аннотация

Состояния гонки (data races) – это несинхронизированные обращения к одному и тому же участку памяти разных потоков параллельной программы. Состояния гонки являются одними из самых трудно обнаруживаемых ошибок многопоточного программирования. Автоматический поиск гонок является предметом активных исследований в последние двадцать лет, однако, например, для Java-приложений на настоящий момент не существует полноценного программного средства (детектора гонок), применимого для промышленных приложений (сотни и тысячи классов). В статье предлагается идея динамического обнаружения гонок на основе синхронизационных контрактов. Последние помогают корректно исключать из области анализа произвольные части приложения, по той или иной причине не интересные с точки зрения поиска гонок (например код стандартных библиотек), делая процесс поиска гонок гибко управляемым. Это, в свою очередь, позволяет существенно понизить накладные расходы при поиске гонок без потери точности. В статье также представлена реализация этой идеи и апробация созданного инструмента.

Ключевые слова: многопоточность, параллельное программирование, автоматическое обнаружение ошибок, состояние гонки.

1. ВВЕДЕНИЕ

В настоящее время один из основных ресурсов в увеличении производительности программно-аппаратных систем – это параллелизм. Становится все больше многоядерных и многопроцессорных вычислительных устройств, но, для того чтобы их эффективно использовать, нужны параллельные программы. Однако разработка таких программ является сложной задачей, при их создании допускаются специфические ошибки, связанные с синхронизацией параллельных потоков. *Состояния гонки (data race)* – одни из самых трудно обнаруживаемых ошибок параллельного программирования. Состояние гонки возникает, когда два потока несинхронизированно обращаются к одному и тому же участку памяти и одно из этих обращений является записью данных [27].

Гонки очень трудно обнаружить с помощью тестирования, так как их возникновение обычно не приводит к немедленному сбою в программе. Но повреждаются глобальные данные, и это проявляется лишь через некоторое время, в других модулях и подсистемах, отличных от той, где произошла гонка, в виде необъяснимых поведенческих эффектов. Более того, эти эффекты могут быть разными и иметь «плавающий» характер. Кроме того, разработка и тестирование программной системы обычно осуществляются на рабочих станциях с малым количеством ядер и процессоров, а использование – на больших серверах, которые могут содержать несколько сотен или даже тысяч ядер и процессоров. В последнем случае вероятность проявления многопоточных ошибок очень высока, а в первом случае они могут не проявиться из-за низкого уровня параллелизма.

Итак, задача автоматического обнаружения гонок является актуальной и востребованной на практике, исследования в этой области ведутся уже более двух десятков лет. Существуют два принципиально разных подхода к автоматическому обнаружению гонок – статический анализ исходного кода или скомпилированных файлов программы [18, 23, 26] и динамический анализ программы непосредственно во время её выполнения [16, 17, 19, 30].

Методы статического анализа не требуют запуска исследуемого приложения, поэтому их работа не зависит от входных данных, окружения и контекста исполнения приложения. Кроме того, статические методы осуществляют поиск гонок во всех модулях приложения, независимо от частоты их реального выполнения. К сожалению, задача проверки существования потенциальных состояний гонки в программе NP-полна для конечных графов выполнения и алгоритмически неразрешима в общем случае [27, 28]. Поэтому статическим детекторам приходится существенно ограничивать глубину анализа графа выполнения программы, что приводит к низкой точности поиска – то есть существующие в программе гонки не обнаруживаются (*false negatives*) – и множеству ложных срабатываний – находятся псевдогонки (*false positives*).

Динамические методы поиска гонок нацелены на преодоление этих проблем. Они осуществляют проверки непосредственно во время работы программы, когда доступна вся информация о ходе её выполнения. Однако существует ряд трудностей, препятствующих разработке эффективных динамических детекторов, главная из которых – большие накладные расходы, возникающие из-за того, что требуется обрабатывать очень большой объем данных: все операции синхронизации и все обращения к разделяемым данным.

Фактически, основными требованиями к промышленному динамическому детектору является сочетание приемлемой точности поиска и производительности. Проблема высокой точности является сугубо алгоритмической и была решена на заре исследований в области динамического поиска гонок посредством изобретения точного алгоритма happens-before, одна из эталонных реализаций которого представлена в работе [21]. В работе [11] представлена проведённая нами адаптация данного алгоритма к Java-программам. Однако обработку всех операций синхронизации и всех обращений к разделяемым данным в программе невозможно осуществить эффективно, и, по-видимому, оптимизации в этом направлении не дадут эффекта, поскольку количество перехватываемых операций все равно остаётся очень большим, равно как и накладные расходы на хранение векторных часов и т. д. В связи с этим сокращение объёма обрабатываемых данных видится более перспективным подходом – необходимо сузить область анализа программы для достижения приемлемой производительности. Этим путём идёт метод семплирования (*sampling*) [15, 24] (анализируются не все события, а лишь их часть). Такой подход существенно увеличивает производительность, однако ведёт к потере точности[1] . Необходимо отметить, что, несмотря на обилие исследований в области динамического поиска гонок, большинство работ ограничивается усовершенствованием существующих алгоритмов и реализацией прототипов.

В данной статье предлагается и обосновывается идея динамического обнаружения гонок на основе синхронизационных контрактов. Последние предназначены для того, чтобы позволять корректно исключать из области анализа произвольные части приложения, по той или иной причине не интересные с точки зрения поиска гонок (например код стандартных библиотек), и избежать потери точности, тем самым делая процесс поиска гонок гибко управляемым. Это, в свою очередь, позволяет существенно понизить накладные расходы динамического поиска гонок без потери точности. В статье также представлена реализация этой идеи и апробация созданного инструмента.

[1] Подробный обзор статических и динамических методов поиска гонок представлен автором в работах [12, 13].

2. ОТНОШЕНИЕ HAPPENS-BEFORE

При динамическом подходе к поиску гонок нужно уметь отвечать на следующий вопрос: правда ли, что любые два обращения к разделяемому участку памяти из различных потоков, где одно из них является обращением на запись, упорядочены с помощью синхронизационных операций? Более формально, программа считается свободной от гонок, если в ней между любыми двумя обращениями разных потоков T1 (на запись) и T2 (на чтение или запись) к одной переменной происходит синхронизация (то есть устанавливается барьер памяти): поток T2 видит изменения, сделанные потоком T1 [31]. Это отношение называется отношением happens-before и формально задаётся следующим образом [20]:

– если событие A синхронизировано с событием B, то A happens-before B;

– действия в одном потоке, произошедшие одно после другого, находятся в отношении happens-before;

– завершение конструктора объекта предшествует запуску его финализатора (finalizer);

– отношение happens-before транзитивно замкнуто.

Два обращения A и B из различных потоков к одному участку памяти не образуют гонку, если A happens-before B или B happens-before A [20].

Для отслеживания отношения happens-before используется предложенный Лампортом в 1978 году механизм векторных часов [22]. *Векторные часы* представляют собой массив чисел, по длине равный количеству потоков в программе: каждому потоку соответствуют одни часы (одно число), увеличивающиеся по мере совершения потоком операций. Каждый поток хранит свою локальную копию векторных часов, синхронизируясь с копиями часов других потоков во время синхронизационных операций. Передача часов между потоками осуществляется с помощью вспомогательных часов, ассоциированных с синхронизационными объектами.

Векторные часы хранят информацию о каждом потоке в системе и потому очень «дороги»: если в программе n потоков, то каждая операция над векторными часами требует $O(n)$ времени, а для хранения часов нужно $O(n)$ памяти. Поэтому, хотя теоретически и возможен сбор информации о программе с любой степенью точности, на практике эта точность сильно ограничена необходимостью обеспечивать эффективность как по потреблению ресурсов, так и по скорости выполнения программы. Динамический анализ приводит к большим накладным расходам, поэтому его крайне сложно применять для высоконагруженных приложений.

В спецификации Java happens-before задается с помощью отношения «*synchronized-with*» (отношение *синхронизированности*), которое формально определяется следующим образом:

– освобождение монитора всегда синхронизировано с его последующим захватом;

– запись volatile-переменной всегда синхронизирована с её последующим чтением;

– операция запуска потока всегда синхронизирована с первым действием в этом потоке;

– последнее действие потока T1 всегда синхронизировано с любым действием потока T2, который «знает», что поток T1 остановился (с помощью Thread.join() или Thread.isAlive());

– если поток T1 прерывает поток T2, прерывание синхронизировано с любым действием в любом потоке, который «знает», что поток T1 был прерван (перехватил InterruptedException или вызвал Thread.isInterrupted());

– запись значения по умолчанию каждой переменной всегда синхронизирована с первым действием любого потока.

В Java каждой паре синхронизированных событий соответствует определённый синхронизационный объект – например монитор, volatile-переменная или сам поток. Первый элемент пары событий мы в дальнейшем будем называть захватом синхронизационного объекта, а второй – его освобождением (по аналогии с частным случаем – захватом и освобождением блокировки).

3. МОТИВАЦИЯ

Для популярных индустриальных языков программирования (Java, C# и т. д.) существует множество готовых библиотек и подсистем, реализующих стандартную функциональность – работу с базой данных, организацию клиент-серверного сетевого взаимодействия и т. д., которые имеют хорошо специфицированные и детально документированные интерфейсы (см. рис. 1). При разработке нового приложения в большинстве случаев используется значительное количество таких библиотек – это помогает сконцентрироваться на специфических для приложения задачах, делегировав типовую функциональность хорошо зарекомендовавшим себя надежным сторонним программным компонентам.

Безусловно, существует ряд систем с повышенными требованиями надёжности и отказоустойчивостью: как правило, это системы реального времени, например системы управления телевещанием, бортовые системы управления или регистраторы аварийных событий. В таких системах надежности используемых сторонних компонент уделяется не меньше внимания, чем самому приложению. Однако для широчайшего класса бизнес-систем надёжность используемых библиотек предполагается достаточной, то есть основным источником ошибок априори предполагается разрабатываемый код. Кроме того, сложность самих приложений достаточно велика, поэтому разработчики и тестеровщики фокусируются на корректности и качестве самого приложения, а не используемых им компонент.

Обнаружение и устранение гонок в приложении является частью разработки и тестирования системы, поэтому и в этом случае логично сфокусироваться на анализе самого приложения, а все сторонние библиотеки исключить из анализа. В некоторых разрабатываемых системах код самого приложения может составлять лишь малую часть (несколько процентов) от совокупного объема кода приложения и используемых библиотек. Например, именно таким будет соотношение для типичного клиент-серверного приложения с незначительной бизнес-логикой. Для других же систем количество собственного кода, напротив, может составлять 80% всего кода системы и более. Так обстоит дело в приложениях, реализующих сложные протоколы взаимодействия систем. В среднем можно утверждать, что объем программного кода используемых приложением библиотек *сопоставим* с объёмом кода самого приложения. В этом свете исключение библиотек из области поиска гонок позволит получить существенный прирост производительности.

Однако недостаточно просто исключить библиотеки из анализа – это приведёт к тому, что детектор пропустит множество операций синхронизации и произведёт большое количество ложных срабатываний. Сложно точно оценить количество таких ложных срабатываний, однако результаты разработчи-

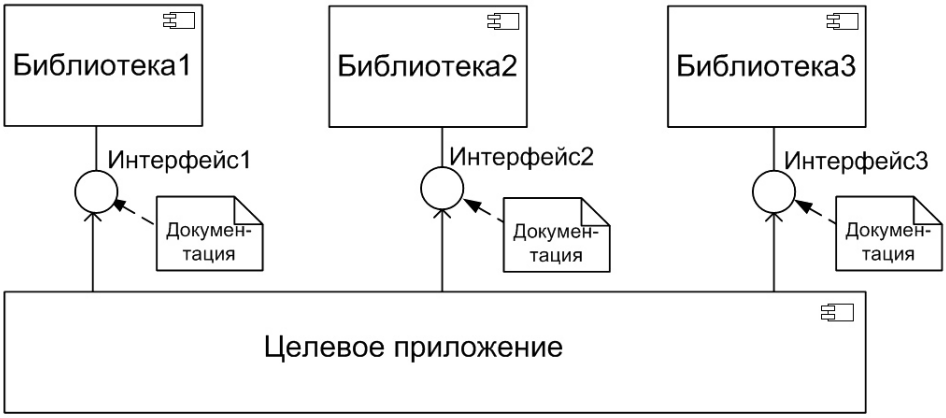

Рис. 1. Типовая организация взаимодействия приложения со сторонними библиотеками

ков метода семплирования позволяют утверждать, что в общем случае их количество будет достаточно большим, чтобы существенно затруднить вычленение реальных гонок из множества всех обнаруженных [15, 24]. Более того, в современном программировании очень часто непосредственное управление ходом выполнения программы, а также порождение и контроль выполнения потоков делегируется различным стандартным библиотекам. Например в типичном клиент-серверном Java-приложении ходом выполнения программы может управлять Spring[1] или Seam[2], а обработкой клиентских запросов в различных потоках – контейнер сервлетов Apache Tomcat[3]. В таких случаях фактически вся синхронизация между потоками осуществляется вне кода самого приложения, поэтому при исключении этих библиотек из анализа количество ложных срабатываний будет огромно.

Итак, если мы хотим исключить из анализа гонок код библиотек, то нам необходимо описать поведение этого кода в многопоточной среде. Например, если подсистема обмена сообщениями экспортирует интерфейс потокобезопасной очереди сообщений, то необходимо описать свойство потокобезопасности его методов и, возможно, дополнительные свойства, гарантирующие синхронизацию между потоками в случае корректного использования методов данной очереди.

Интерфейсы библиотек и модулей содержат только описания сигнатуры экспортируемых примитивов, но не их поведенческие свойства и, более того, не предоставляют средств для описания этих свойств. Можно было бы аннотировать исходный код библиотек – о применении аннотирования в статическом анализе и компиляции см. например работы [1, 9]. Однако исходный код сторонних библиотек, как правило, недоступен и фактически никогда не включается в сборку приложения.

В программировании существует широко известная концепция *контрактов* [25], воплощённая в языке Eiffel. Базовым понятием этой концепции является контракт программной сущности (как правило, метода), состоящий из предусловий, постусловий и инвариантов [25]. Кроме того, дополнительно можно задавать инварианты для классов и других комплексных сущностей. По сути, в данном подходе контракты являются описаниями поведенческих свойств, которые дополнительно экспортируются наружу наряду с сигнатурой. В общем случае язык описания контрактов имеет сложность, сопоставимую со сложностью обычных языков программирования[4].

Предлагаемый нами подход базируется на идее контрактов, однако не следует ей в строгом смысле, поскольку набор свойств, интересных с точки зрения обнаружения гонок, весьма узок и специфичен и может быть описан более простым способом, без использования полноценного языка описания контрактов.

4. СИНХРОНИЗАЦИОННЫЕ КОНТРАКТЫ

Принцип инкапсуляции в объектно-ориентированном подходе к программированию подразумевает, что использование объекта осуществляется посредством вызова его публичных операций. В Java это вызовы методов классов. Соответственно, достаточно уметь описать *возможность* использования методов исключённых классов в многопоточной среде. С этой точки зрения возможны следующие варианты:

– метод не является потокобезопасным, то есть его одновременное использование несколькими потоками не предусмотрено и требует внешней синхронизации;

[1] Spring framework: http://www.springsource.org/.

[2] Seam framework: http://www.seamframework.org/.

[3] Apache tomcat: http://tomcat.apache.org/.

[4] Например, см. такие расширения Java, предоставляющие возможность описания контрактов, как JavaTESK (http://www.unitesk.ru/content/category/5/25/60/), jContractor (http://jcontractor.sourceforge.net/), contract4j (http://sourceforge.net/projects/contract4j/), а также использование контрактов в модельно-ориентированном тестировании [3].

– метод потокобезопасен и может вызываться несколькими потоками одновременно без внешней синхронизации;

– метод не только потокобезопасен, но и является частью комплексного механизма синхронизации, который гарантирует синхронизацию потоков.

Эти варианты составляют различные примитивы синхронизационного контракта программной компоненты. Рассмотрим эти варианты подробней, а затем перейдём к определению синхронизационного контракта (рис. 2).

4.1. ПОТОКОНЕБЕЗОПАСНЫЕ МЕТОДЫ

Поскольку предполагается, что мы имеем дело с интерфейсом, не имеющим побочных эффектов, то вызов такого метода может модифицировать только тот объект, на котором он был вызван. Соответственно, необходимо отслеживать вызовы потоконебезопасных методов одного и того же объекта из различных потоков и сигнализировать об этом пользователю. Однако возможно и такое, что метод является немодифицирующим, то есть он лишь читает внутреннее состояние объекта, но не изменяет его. Согласно определению гонки, одновременные чтения общей разделяемой памяти из различных потоков не образуют гонки – необходимо, чтобы один из потоков осуществлял запись в эту память. Соответственно, необходимо разделять модифицирующие и немодифицирующие методы объектов. Однако для понимания природы метода зачастую необходимо существенно углубиться в его реализацию. С другой стороны, если все методы трактовать как модифицирующие, то возможно лишь обнаружение ложных гонок (*false positive*), что не так критично, как пропуск гонок (*false negative*). Поэтому по умолчанию нужно трактовать все методы как модифицирующие, но предоставлять возможность пометить метод как немодифицирующий.

4.2. ПОТОКОБЕЗОПАСНЫЕ МЕТОДЫ

Если метод предназначен для одновременного использования несколькими потоками, то его использование не может привести к гонке, поэтому динамическому детектору отслеживать вызовы данных методов не нужно.

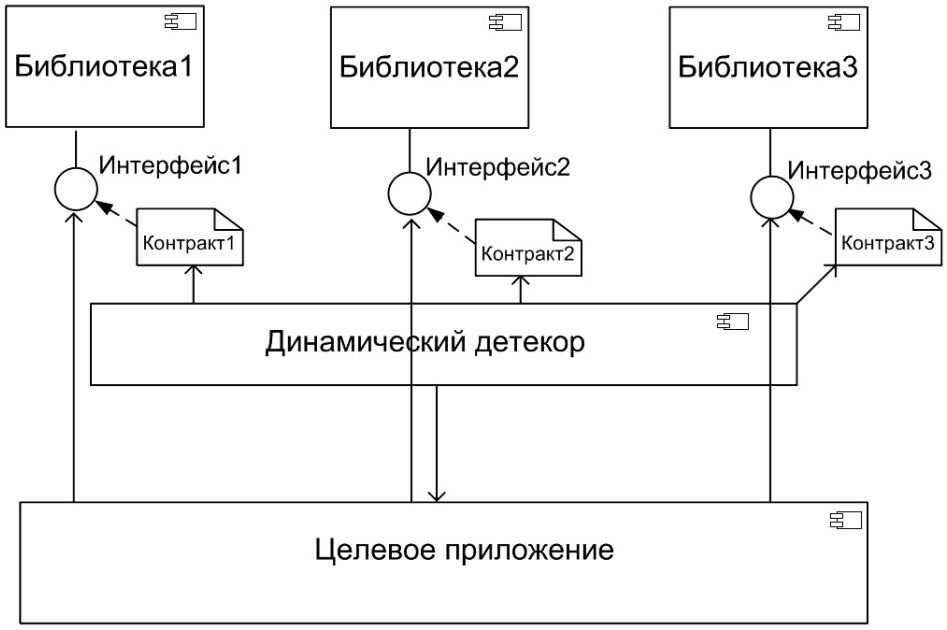

Рис. 2. Взаимодействие динамического детектора с целевым приложением и контрактами сторонних библиотек

4.3. МЕТОДЫ-ЧАСТИ КОМПЛЕСКНЫХ МЕХАНИЗМОВ СИНХРОНИЗАЦИИ

Методы экспортируемого интерфейса могут не только гарантировать корректность поведения в многопоточной среде, но и обеспечивать синхронизацию между потоками. Иными словами, такие методы могут быть частью высокоуровневого синхронизационного механизма, обеспечивающего передачу отношения happens-before, и это описано в документации. Данную информацию, безусловно, необходимо использовать при поиске гонок, поскольку это обеспечит отсутствие потери точности, которое возможно по причине сужения области анализа. Следовательно, необходим способ описания подобных механизмов.

Передача отношения happens-before, как говорилось выше, связана с синхронизационным объектом и состоит из двух частей – один поток освобождает синхронизационный объект, а второй впоследствии его захватывает, что и обеспечивает синхронизацию между этими потоками. Следовательно, передача happens-before обеспечивается *парой вызовов методов* – один поток вызывает первый метод, а потом второй поток вызывает второй метод (возможно, совпадающий с первым, возможно – нет). Таким образом, необходимо уметь описывать связи между вызовами пар методов. Это возможно, поскольку с этими вызовами связан синхронизационный объект. Поскольку мы предположили отсутствие побочных эффектов, то связь между вызовами методов посредством синхронизационного объекта должна быть *явной*, то есть опираться на объекты, являющиеся частями сигнатуры метода. Переберем все варианты *примитивной* связи между вызовами методов.

Пусть у нас есть два метода: метод $f(P_{11}, \ldots, P_{1n})$ объекта O_1 и метод $g(P_{21}, \ldots, P_{2m})$ объекта O_2, где $n, m \geq 0$. Объект O_1 будем называть *объектом-владельцем* метода f, а O_2 – объектом-владельцем[1] метода g. Примитивная связь может быть одной из следующих трёх типов, представленных ниже.

1. Связь «владелец-владелец»: $O_1 == O_2$, то есть методы принадлежат одному объекту.

2. Связь «владелец-параметр»: $\exists\, i \in [1..n]: O_2 == P_{1i}$ or $\exists\, j \in [1..m]: O_1 == P_{2j}$, то есть параметр одного метода является объектом-владельцем второго метода.

3. Связь «параметр-параметр»: $\exists\, i \in [1..n], j \in [1..m]: P_{1i} == P_{2j}$, то есть i-й параметр метода f и j-й параметр метода g являются одним и тем же объектом.

Фактически, любая реальная связь между вызовами методов является комбинацией конечного количества представленных выше примитивных связей[2]. Все такие связи, гарантирующие синхронизацию (отношение happens-before), необходимо явно описывать в синхронизационном контракте библиотеки.

Вернемся к потокобезопасным методам. Они предназначены для одновременного использования несколькими потоками и не требуют дополнительной синхронизации, то есть эти методы обеспечивают синхронизацию потоков внутри себя. Таким образом, если два потока вызывают по очереди один и тот же потокобезопасный метод, то между ними может возникнуть отношение happens-before. Но поскольку эта синхронизация не декларирована явно и носит, скорее, случайный характер, мы не будем описывать такую передачу отношения happens-before в наших контрактах. В качестве примера рассмотрим

[1] Строго говоря, в объектно-ориентированных языках программирования методы бывают двух типов – методы, принадлежащие непосредственно классу (например, в Java они помечаются ключевым словом static), и методы, принадлежащие конкретному объекту данного класса. Для упрощения изложения мы считаем, что для каждого класса существует единственный объект типа «class», которому принадлежат все методы из этого класса. Например, в Java такой объект действительно существует и может быть получен с помощью ключевого слова «class». Имея в виду данное допущение, мы можем трактовать любой метод как принадлежащий объекту, а не классу.

[2] Возможна также связь через возвращаемое значение метода, которую мы не рассматриваем в данной статье. По нашим наблюдениям, такая связь встречается крайне редко, однако безусловно требует изучения. Обсуждение этого вопроса см. в разделе 6 «Ограничения подхода».

метод print, отвечающий за печать в файл, и предположим, что он потокобезопасен и внутри себя защищает непосредственное обращение к файлу критической секцией, поскольку с файлом одновременно и безопасно может работать только один поток. Два потока, которые по очереди вызвали этот метод, в действительности синхронизируются – выход первого потока из критической секции будет предшествовать (happens-before) входу второго потока в критическую секцию. Однако данная передача отношения happens-before является частью *реализации* метода print, не описана в его документации и не должна использоваться программистом для обеспечения синхронизации потоков.

4.4. СИНХРОНИЗАЦИОННЫЕ КОНТРАКТЫ

Будем называть *happens-before контрактом* описание пары вызовов методов, которые, будучи вызванными в определённом порядке (связь между вызовами этих методов определяется суперпозицией примитивных связей, определённых выше), гарантируют корректную синхронизацию потоков.

Рассмотрим пример. В листинге 1 приведён фрагмент happens-before контракта для пары методов put() и get() известного Java-интерфейса ConcurrentMap. В этом контракте указано, что вызов метода put() синхронизирован с последующим вызовом метода get() *того же объекта* по *тому же ключу* (в обоих методах ключ является первым параметром; в Java нумерация параметров начинается с нуля). То есть описываемая в данном случае связь является суперпозицией примитивных связей первого и третьего рода.

Будем называть *синхронизационным контрактом* программной компоненты объединение всех happens-before контрактов для вызовов методов её публичных интерфейсов и перечисление всех ее потокобезопасных методов.

Можно описать следующим образом схему анализа кода, который готовится к исключению и для которого мы хотим описать синхронизационный контракт. Нужно найти все вызовы методов исключаемого кода в основном коде и обработать их по алгоритму, представленому на рис. 3.

Каждый вызов метода нужно проанализировать на предмет его потокобезопасности. Если метод потокобезопасен и не является частью механизма синхронизации, то необходимо пометить его как потокобезопасный в контракте данной программной компоненты; если является – создать соответствующий happens-before контракт. Если же метод потоконебезопасен, то по нему следует искать гонки. Как говорилось выше, любой такой метод будет по умолчанию трактоваться детектором, как модифицирующий (write). Однако, если анализ показал, что данный метод точно является немодифицирующим, можно пометить его соответству-

Листинг 1. Пример happens-before контракта для Java

```
<Sync>
    <Links>
        <Link send="owner" receive="owner"/>
        <Link send="param" send-number="0" receive="param" receive-number="0"/>
    </Links>
    <Send>
        <MethodCall owner="java.util.concurrent.ConcurrentMap" name="put"
                              descriptor="(Ljava/lang/Object;Ljava/lang/Object;)
                                                          Ljava/lang/Object;"/>
    </Send>
    <Receive>
        <MethodCall owner="java.util.concurrent.ConcurrentMap" name="get"
                             descriptor="(Ljava/lang/Object;)Ljava/lang/Object;"/>
    </Receive>
</Sync>
```

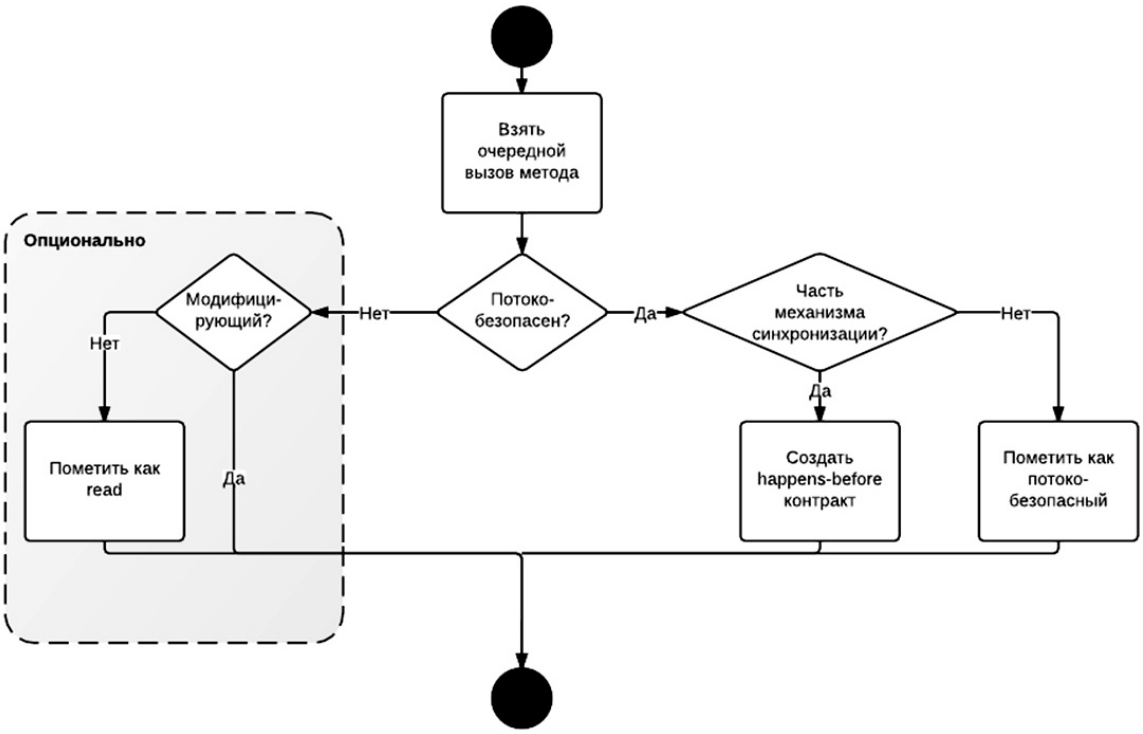

Рис. 3. Алгоритм анализа вызовов методов исключенного кода из анализируемого кода

ющим образом для повышения точности детектора.

Разумеется, все вызовы методов исключаемого кода должны быть проанализированы и синхронизационный контракт исключаемой программной компоненты должен быть описан полностью. В этом смысле исключаемая компонента должна обладать замкнутой функциональностью, то есть не должна содержать побочных эффектов – нужно, чтобы обе части её happens-before конрактов принадлежали этой компоненте.

С этой точки зрения эффективнее всего описывать синхронизационные контракты библиотек, используемых системой. Однако, возможно и иное применение данного подхода – например, можно описать синхронизационный контракт протестированного ранее модуля системы и исключить его из анализа.

Рассмотрим пример кода, представленный в листинге 2 в предположении, что объекты write_lock (блокировка) и list (непотокобезопасный список) не принадлежат области анализа, а приведённый фрагмент кода – принадлежит.

Поскольку метод lock() объекта write_lock служит для входа в критическую секцию, он, во-первых, потокобезопасен, а во-вторых, совместно с методом unlock() образует механизм синхронизации. Следовательно, необходимо создать соответствующий синхронизационный контракт, указав, что вызовы методов lock() и unlock() связаны через объект-владелец (примитивная связь первого типа).

Метод add() объекта list служит для добавления объектов в список и является модифицирующим. Поскольку список непотокобезопасен, а по умолчанию все методы, не

Листинг 2. Пример фрагмента псевдокода для анализа

```
write_lock.lock(); /*вход в критическую секцию*/
list.add(composite_object); /*Добавление объекта в список*/
write_lock.unlock(); /*выход из критической секции*/
```

помеченные явно как потокобезопасные, трактуются детектором как модифицирующие, нет необходимости помечать его как модифицирующий.

5. СИНХРОНИЗАЦИОННЫЕ КОНТРАКТЫ В JAVA

Вышеуказанная концепция может быть реализована для широкого класса объектно-ориентированных языков и платформ. Однако существуют два основных требования, которым целевой язык должен удовлетворять.

Во-первых, язык должен обладать архитектурно-независимой моделью исполнения, которая позволяет описывать поведение многопоточных программ достаточно детерминировано. Семантика многопоточных программ во многом определяется моделью памяти (memory model) – набором правил, по которым потоки могут взаимодействовать друг с другом посредством общей памяти. К сожалению, формальная семантика некоторых традиционно применяемых языков программирования[1] не обладает свойством последовательной консистентности (sequential consistency) в условиях многопоточного исполнения. Это означает, что в общем случае путь выполнения программы не может быть описан как чередование операций в различных потоках, поскольку «видимость» изменений, производимых в общей памяти, остаётся недетерминированной [14].

Во-вторых, в культуре программирования на данном языке должна отчетливо проявляться ориентированность на тщательную проработку документации экспортируемых интерфейсов и особенно их поведения в многопоточной среде.

Для реализации нашей концепции мы выбрали язык и платформу Java как одну из самых популярных платформ для разработки индустриальных систем. На Java написано огромное количество бизнес-приложений, поэтому задача поиска гонок в ней актуальна, а обзор автора [13] показал, что промышленного динамического детектора гонок для Java-приложений не существует[2].

Кроме того, в Java имеется очень строгая модель памяти. В частности, в ней формально определено взаимодействие потоков и отношение happens-before на множестве событий программы, на отслеживании которого базируется наиболее популярный точный алгоритм happens-before [11, 17, 19]. Поэтому синхронизационные контракты можно описывать в терминах этого отношения.

6. ОГРАНИЧЕНИЯ ПОДХОДА

Представленный подход обладает рядом ограничений.

1. С помощью предложенного подхода могут быть описаны лишь явные happens-before контракты, то есть подразумевается наличие явной связи между вызываемыми методами через параметр или через объект-владелец метода. Можно представить себе ситуацию с более сложной, неявной связью. Описывать данные контракты очень затруднительно, кроме того, их наличие свидетельствует о плохой организации кода. Это ограничение подхода является компромиссом между точностью обнаружения гонок и сложностью реализации.

2. В описанном подходе вызовы методов, являющихся частью happens-before контрактов, трактуются как атомарные операции, хотя таковыми не являются – точка синхронизации, в которой происходит непосредственная синхронизация между потоками, обычно находится где-то внутри метода и отделена по времени как от точки входа в метод, так и от точки выхода из него. В эти временные промежутки потоки могу приостанавливаться, а управление передаваться другим потокам, что может повлиять на состояние системы. Следовательно, информация о синхронизации потоков на момент выхода из метода может оказаться устарев-

[1] Например для языка C++ формальная модель памяти появилась лишь в стандарте 2011 года ISO/IEC 14882:2011.

[2] Поскольку платформы Java и .NET очень схожи, то возможен перенос нашей реализации на .NET.

шей или вовсе некорректной. Это является принципиальным недостатком подхода и может привести к ложным срабатываниям. Однако, по нашим предварительным оценкам, связанная с этим ограничением вероятность ложных срабатываний является крайне незначительной, так как размер методов, являющихся частью механизма синхронизации, как правило, достаточно мал. В данный момент мы разрабатываем систему тестов, которая позволит проверить наши оценки экспериментальным способом.

3. В атомарных операциях, предложенных в подходе, не учитываются возвращаемые значения методов, являющихся частью синхронизационного контракта, что, вообще говоря, не совсем корректно.

- Для передачи отношения happens-before может требоваться выполнение определённых условий на возвращаемые значения методов. Например может требоваться совпадение возвращаемых значений. Реализация выполнения проверки таких условий нам видится скорее техническим вопросом, над которым мы работаем в настоящий момент.
- Во-вторых, наряду с входными параметрами метода и его объектом-владельцем, возвращаемое значение является частью сигнатуры метода и может участвовать в передаче отношения happens-before. Следовательно, возвращаемые значения методов могут быть вовлечены в примитивную связь, описанную выше. Как показывает наш опыт, такие связи встречаются достаточно редко, но, безусловно, эта задача заслуживает глубокого рассмотрения и является одной из основных теоретических задач, над которыми работа будет продолжена впоследствии.

7. АПРОБАЦИЯ JDRD

Предложенный подход был реализован в Java-детекторе jDRD. Этот детектор был апробирован на ряде промышленных проектов. На тех же проектах мы запускали детектор IBM MSDK [29] – единственный (кроме jDRD) имеющийся в наличии динамический детектор гонок для Java-приложений. Ниже представлены результаты апробации.

7.1. НАГРУЗОЧНЫЙ ТЕСТ ДЛЯ ПРОТОКОЛА ДОСТАВКИ КОТИРОВОК

Это приложение является консольным, содержит несколько десятков потоков, несколько тысяч классов и требует для своей работы около 200 Мб оперативной памяти. Общий размер дистрибутива приложения – 2.8 МБ. Приложение не использует сторонние библиотеки.

Вскоре после запуска IBM MSDK на данном приложении объем потребления оперативной памяти достиг предела, приложение «зависло» и после некоторого времени завершило работу с ошибкой переполнения памяти.

jDRD показал увеличение потребления памяти в полтора раза и увеличение потребления CPU в два раза. Время выполнения базового приложения увеличилось примерно в 10 раз, но несмотря на это, эксперимент дошёл до конца, и в исходном приложении было обнаружено более ста гонок. После анализа и консультации с разработчиками выяснилось, что в приложении используется собственный механизм синхронизации потоков. Для него были описаны соответствующие синхронизационные контракты, и после этого количество обнаруженных гонок сократилось до шести, из которых пять были безопасными (так называемые *benign races*) – их наличие было известно разработчикам и не влияло на логику работы программы. Последняя же гонка оказалось ошибкой, разработчики подтвердили это и исправили её.

7.2. ПОЛЬЗОВАТЕЛЬСКИЙ КЛИЕНТ К БАГ-ТРЕКЕРУ JIRA[10]

Это приложение содержит несколько десятков потоков, несколько тысяч классов, требует порядка 500 Мб оперативной памяти. Общий размер дистрибутива (приложение вместе со всеми используемыми библио-

[1] Atlassian JIRA: http://www.atlassian.com/software/jira/overview/.

теками) – 4 МБ. Отношение кода приложения к коду библиотек составляет 1/3.

IBM MSDK существенно замедлил работу приложения – потребление памяти выросло в 4 раза, потребление CPU – в два, замедление скорости реакции пользовательского интерфейса было видно невооружённым глазом и достигало нескольких сотен миллисекунд, – но успешно закончил работу, обнаружив 8 гонок, из которых 6 оказались ошибками, а ещё 2 – ложными срабатываниями, возникшими по причине неполноты поддержки средств пакета java.util.concurrent детектором MSDK.

jDRD, фактически, не увеличил потребление CPU, потребление памяти увеличил на 50%, задержки в работе пользовательского интерфейса заметны не были. Все гонки, обнаруженные детектором IBM MSDK, были также обнаружены и jDRD, но, кроме них, было найдено также 5 новых гонок. Из пяти гонок, обнаруженных только jDRD, три возникали в одном и том же участке функциональности, отвечающей за мониторинг активности пользователя. Устранение этих гонок помогло исправить ошибку, о которой часто сообщали пользователи и которую не удалось отследить на этапе тестирования по причине ее невоспроизводимости.

7.3. КЛИЕНТСКАЯ ЧАСТЬ МОНИТОРИНГОВОЙ СИСТЕМЫ

Это приложение содержит несколько десятков потоков, несколько тысяч классов, требует порядка 500 Мб оперативной памяти. Общий размер дистрибутива (приложение + все используемые библиотеки) – 13 МБ. Отношение кода приложения к используемым библиотекам – 22/78.

IBM MSDK, запущенный на данном приложении, «завис» и завершил работу с ошибкой переполнения памяти через несколько минут.

jDRD работал стабильно и успешно завершился. Потребление памяти приложения выросло в 2 раза и более не увеличивалось. Потребление ресурсов процессора выросло на 10%. Задержки в работе интерфейса, заметные невооруженному глазу, проявлялись лишь на сложных операциях (добавление нового подключения к серверу, запрос исторических данных за несколько суток). В приложении было обнаружено более 10 гонок, все из которых оказались реальными (то есть ложных срабатываний не было). Из этого списка было выделено и передано команде разработки 5 наиболее критических гонок, возникших в функциональности, отвечающей за создание соединения и взаимодействие с сервером.

8. ЗАКЛЮЧЕНИЕ

В статье предложен способ реализации идеи частичного анализа программы, который основывается на идее гибкого конфигурирования контрактов «пограничного кода» – классов и компонент, которые не попадают в используемую область программы, но используются в ней. С учетом того, что в последнее время все больше внимания уделяется тщательному документированию кода с точки зрения его многопоточной корректности, такой механизм видится нам перспективным. Мы представили сам подход и описали апробацию его реализации – детектора jDRD. Данная реализация не является прототипом и представляет собой промышленный детектор, однако она ещё не завершена, поэтому ближайшей задачей является завершение реализации jDRD – исправление ошибок, связанных с инструментированием байт-кода, обеспечение покрытия функциональности jDRD тестами и т. д.

После завершения разработки планируется проведение масштабных испытаний – тестирование на крупных промышленных проектах. Апробация, результаты которой приведены в данной работе, свидетельствует об успешности и перспективности предложенного подхода, однако не позволяет сформулировать утверждения о ресурсопотреблении подхода. Также перспективной является интеграция подхода со средствами статического анализа [2] и model-checking [6], модельно-ориентированного тестирования [3, 5] и модельно-ориентированных средств визуализации программ [4, 7, 10].

Литература

1. *Булычев Д.Ю.* Компонентизация языковых процессоров на основе расширяемых типов данных и управляемых ими преобразователей // Системное программирование, 2012. Т. 7. 1.

2. *Глухих М.И., Ицыксон В.М., Цесько В.А.* Использование зависимостей для повышения точности статического анализа программ // Моделирование и анализ информационных систем, 2011. Т. 18. № 4. С. 68–79.

3. *Иванников В.П., Камкин А.С., Косачев А.С., Кулямин В.В., Петренко А.К.* Использование контрактных спецификаций для представления требований и функционального тестирования моделей аппаратуры // Программирование, 2007. Т. 33. № 5. С. 47–62.

4. *Иванов Б.Н., Кознов Д.В., Мурашова Т.С.* Поведенческая модель RTST / Записки семинара кафедры системного программирования «Case-средства RTST++», 1998. С. 37.

5. *Ицыксон В.М., Захаров А.В., Ахин М.Х., Мяснов А.В.* Автоматическое обнаружение дефектов программных систем на основе метода проверки моделей // Научно-технические ведомости Санкт-Петербургского государственного политехнического университета, 2008. № 65. С. 127–133.

6. *Карпов Ю.Г., Борщев А.В., Рудаков В.В.* Верификация дискретных систем реального времени // Информационный бюллетень РФФИ, 1994. Т. 2. № 1. С. 312.

7. *Кознов Д.В., Ольхович Л.Б.* Визуальные языки проектов // Системное программирование, 2011. Т. 1. С. 148–167.

8. *Петренко А.К.* Методы тестирования программного обеспечения на основе формальных спецификаций // Информационный бюллетень РФФИ, 1998. Т. 6. № 1. С. 416.

9. *Сергей И.Д.* Реализация гибридных типов владения в Java посредством атрибутных грамматик // Системное программирование, 2011. Т. 6. № 1. С. 47–76.

10. *Сорокин А.В., Кознов Д.В.* Обзор Eclipse Modeling Project // Системное программирование, 2010. Т. 5. № 1. С. 6–32.

11. *Трифанов В.Ю.* Динамическое обнаружение гонок в Java-программах с помощью векторных часов // Системное программирование. Вып. 5: Сб. статей / Под ред. А.Н. Терехова, Д.Ю. Булычева. С. 95–116.

12. *Трифанов В.Ю., Цителов Д.И.* Статические и post-mortem средства обнаружения гонок в параллельных программах // Компьютерные инструменты в образовании, 2011. № 5. С. 3–13.

13. *Трифанов В.Ю., Цителов Д.И.* Динамические средства обнаружения гонок в параллельных программах // Компьютерные инструменты в образовании, 2011. № 6. С. 3–15.

14. *Boehm H.* Threads cannot be implemented as a library. In Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, June 2005. Vol. 40, Is. 6. P. 261–268.

15. *Bond M., Coons K., McKinley K.* Pacer: Proportional Detection of Data Races. In Proceedings of 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2010), Toronto, June 2010. P. 255–268.

16. *Choi J., Lee K., Loginov A., O'Callahan R., Sarkar V., Sridharan M.* Efficient and precise datarace detection for multithreaded object-oriented programs. In Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation, 2002. P. 258–269.

17. *Christiaens M., Brosschere K.* TRaDe: A topological approach to on-the-fly race detection in Java programs. In Proceedings of the 2001 Symposium on Java Virtual Machine Research and Technology Symposium, 2001. Vol. 1. P. 105–116.

18. *Engler D., Ashcraft K.* RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In Proceedings of The Nineteenth ACM Symposium on Operating Systems Principles, 2003. P. 237–252.

19. *Flanagan C., Freund S.* FastTrack: Efficient and Precise Dynamic Race Detection. In ACM Conference on Programming Language Design and Implementation, 2009. P. 121–133.

20. Java Language Specification, Third Edition. Threads and Locks. Happens-before Order / http://docs.oracle.com/javase/specs/jls/se7/html/jls-17.html#jls-17.4.5 (дата обращения 29.08.2012).

21. *Itzkovitz A., Schuster A., Zeev-Ben-Mordechai O.* Towards integration of data race detection in DSM systems. Journal of Parallel and Distributed Computing (JPDC), 59(2), 1999. P. 180–203.

22. *Lamport L.* Time, Clocks and the Ordering of Events in a Distributed System. Communications of the ACM, 1978. Vol. 21, Is. 7. P. 558–565.

23. *Leino K., Nelson G., Saxe J.* ESC/Java user's manual. SRC Technical Note 2000–002, 2001.

24. *Marino D., Musuvathi M., Narayanasamy S.* LiteRace: Effective Sampling for Lightweight Data-Race Detection. PLDI '09 Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation, 2009. Vol. 44, Is. 6. P. 134–143.

25. *Meyer B.* Object-Oriented Software Construction. Prentice Hall; 2nd edition (March 21, 2000).

26. *Naik M., Aiken A., Whaley J.* Effective Static Race Detection for Java. In Proceedings of The 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation, 2006. P. 308–319.

27. *Netzer R., Miller B.* What Are Race Conditions? Some Issues and Formalizations. In ACM Letters On Programming Languages and Systems, 1(1), 1992. P. 74–88.

28. *Netzer R.* Race Condition Detection for Debugging Shared-Memory Parallel Programs. PhD Thesis, Madison, 1991.

29. *Qi Y., Das R., Luo Z., Trotter M.* MulticoreSDK: a practical and efficient data race detector for real-world applications. In Proceedings Software Testing, Verification and Validation (ICST), IEEE, 21–25 March 2011. P. 309–318.

30. *Savage S., Burrows M., Nelson G., Sobalvarro P., Anderson T.* Eraser: A Dynamic Data Race Detector for Multithreaded Programs. In ACM Transactions on Computer Systems, 1997. Vol. 15, Is. 4. P. 391–411.

31. *Yu Y., Rodeheffer T., Chen W.* RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking. In SOSP, 2005. P. 221–234.

Abstract

Data races occur in parallel programs when several threads perform concurrent accesses to the same location of shared memory without consistent synchronization. Data races are one of the most hardly detectable multithreading errors. A lot of research in the area of automatic data race detection has been held during last twenty years, but, for example, no full-fledged dynamic data race detector for Java-applications, applicable to industrial systems (hundreds and thousands classes), still exists. In this article an idea of dynamic data race detection based on synchronization contracts is proposed. Synchronization contracts assist to exclude certain, uninteresting from the view of data race detection (for example, code of standard libraries), parts of application form the analysis scope, making data race detection process flexibly manageable. By-turn, it makes possible to reduce overhead significantly without loss of precision. An implementation of this idea and evaluation of resulting tool are also introduced.

Keywords: concurrency, data race, automatic bugs detection.

Трифанов Виталий Юрьевич,
аспирант кафедры системного программирования математико-механического факультета СПбГУ,
инженер-программист компании «Эксперт-Система»,
vitaly.trifanov@gmail.com