Конфигурируемый метод поиска состояний гонок в операционных системах с использованием предикатных абстракций[1]

[1]*П.С. Андрианов <andrianov@ispras.ru>*
[1]*В.С. Мутилин <mutilin@ispras.ru>*
[1,2,3,4]*А.В. Хорошилов <khoroshilov@ispras.ru>*
[1]*Институт системного программирования РАН,*
109004, Россия, г. Москва, ул. А. Солженицына, дом 25
[2]*Московский государственный университет имени М.В. Ломоносова,*
119991, Россия, Москва, Ленинские горы, д. 1.
[3]*Московский физико-технический институт (государственный университет),*
141700, Россия, Московская область, г. Долгопрудный, Институтский пер., 9
[4]*Национальный исследовательский университет «Высшая школа экономики»*
101000, Россия, Москва, ул. Мясницкая, д. 20

Аннотация. В статье представлен конфигурируемый метод для поиска состояний гонок. Метод позволяет настраивать требуемую точность анализа, выбирая баланс между затрачиваемыми ресурсами и количеством ложных предупреждений подключением двух расширений: уточнением путей на основе предикатных абстракций и анализом потоков. Метод основан на алгоритме Lockset и использует упрощенную модель памяти для уменьшения количества ложных предупреждений. Предлагаемый подход был реализован в инструменте CPALockator, который был апробирован на модулях ядра операционной системы Linux, что позволило обнаружить несколько состояний гонок, которые были признаны и исправлены разработчиками.

Ключевые слова: статический анализ; состояние гонки; ядро операционной системы.

DOI: 10.15514/ISPRAS-2016-28(6)-5

Для цитирования: Андрианов П.С., Мутилин В.С., Хорошилов А.В. Конфигурируемый метод поиска состояний гонок в операционных системах с использованием предикатных

[1] Исследования проводились при финансовой поддержке Министерства образования РФ, уникальный идентификатор соглашения RFMEFI61614X0015.

абстракций. Труды ИСП РАН, том 28, вып. 6, 2016, стр. 65-86. DOI: 10.15514/ISPRAS-2016-28(6)-5

1. Введение

Ошибки, связанные с параллельным выполнением кода, традиционно считаются сложными для поиска и исправления. Наиболее многочисленным классом таких ошибок остается состояние гонки, при котором возможен доступ к одной области памяти одновременно из нескольких потоков [1]. Такая ситуация может привести к серьезным последствиям вплоть до падения всей системы [2, 3].

Существует множество инструментов для автоматизации процесса поиска состояний гонки. Мы не будем останавливаться на динамических инструментах, которые анализируют программу в процессе ее работы. Среди статических инструментов выделяют два класса: легковесные и тяжеловесные. Первые обходятся в своей работе незначительными ресурсами, но при этом демонстрируют невысокий процент истинных ошибок и/или частый их пропуск. Тяжеловесные инструменты позволяют доказывать отсутствие ошибок, но при этом потребляют значительный объем ресурсов, в первую очередь времени. Это приводит к тому, что тяжеловесные инструменты становится невозможно использовать на реальных программных системах.

Основной задачей, стоящей перед нами, была разработка легковесного статического анализатора, пригодного для применения к ядру операционной системы. Для минимизации ложных предупреждений ключевой задачей является повышение точности анализа в тех рамках, чтобы инструмент мог быть использован при верификации сложных программных комплексов, каким является ядро операционной системы ОС Linux. В работе описан легковесный метод, который был успешно расширен тяжеловесными подходами, которые обыкновенно применяются для проверки небольших программ тщательным образом.

Разработанный инструмент был успешно апробирован на драйверах ОС Linux.

2. Основные проблемы статического анализа при поиске состояний гонок

Сначала скажем несколько слов о статическом анализе. Программа моделируется с помощью графа потока управления, состоящего из некоторого множества состояний и переходов между ними, которые связаны с операторами программы. Под состоянием программы понимают состояние ее памяти, в которое включается и программный счетчик, который определяет конкретную точку программы. Задача статического анализа — найти множество достижимых состояний из некоторого заданного состояния с помощью переходов. Если среди всех достижимых состояний есть состояния определенного вида (ошибочные), то считается, что была найдена ошибка. Так

как состояний программы, как правило, слишком много, то в анализе используется некоторая абстракция состояний. От устройства абстракции состояния будет зависеть точность всего анализа. Например, если абстрактное состояние программы будет представлять собой значения всех переменных программы для всех ее потоков, то анализ будет очень точным, но крайне медленным. И наоборот, если не учитывать значения никаких переменных, то анализ будет очень быстрым, но неточным.

Теперь определим тот тип ошибки, который мы будем искать. В общем случае состоянием гонки принято называть ситуацию, при которой поведение некоторой системы зависит от последовательности внутренних неконтролируемых событий. Это означает, что в общем случае поведение такой системы не определено. В программных системах состояния гонки, которые происходят при одновременном доступе к одним и тем же областям памяти из разных потоков, при этом хотя бы один из доступов является записью, называются состояниями гонки на данных (англ. data race).

Состояния гонки не обязательно приводят к некорректному поведению программы. Например, одновременная модификация некоторого счетчика статистики может привести лишь к несущественным погрешностям. Такие ситуации называются безобидными (англ. benign) состояниями гонки. Однако известны случаи, в которых стоимость одной такой ошибки становилась катастрофической.

Поиск подобных ошибок является достаточно сложной задачей, в первую очередь потому, что ошибки, связанные с состояниями гонки, возникают недетерминированно. Поэтому любое сколь угодно качественное тестирование не сможет гарантировать, что многопоточная программа не упадет на тех же самых тестах в следующий раз.

Задача статического поиска гонок также является достаточно сложной, так как в общем случае для доказательства отсутствия ошибок необходимо рассмотреть все возможные варианты выполнения нескольких потоков — это задача экспоненциальной сложности. Методы статического анализа с использованием переключений (англ. interleavings) решают эту задачу для очень небольших программ [4, 5, 6, 7].

Многие инструменты нацелены на поиск как можно большего числа ошибок, не стремясь доказывать их отсутствие. В таких инструментах статического анализа также возникает ряд сложностей. Рассмотрим две наиболее важные проблемы: необходимость в модели памяти и присутствие определенного процента ложных срабатываний.

В отличие от динамического анализа, в котором всегда известно, к какой именно области памяти производится доступ, в статическом анализе неизвестно значение указателей, и точно определить, что доступ производится к одной и той же области памяти крайне сложно. Существуют различные методы для решения такой проблемы: точная модель памяти, анализ синонимов

и др. Результатом анализа синонимов является отображение указателей во множество областей памяти, на которые они указывают. Проблема заключается в том, что при консервативном подходе множество возможных областей памяти быстро разрастается при активном использовании указателей. В результате чего анализ тратит значительный объем ресурсов и используется только в тяжеловесных инструментах, для которых больше важна точность. Легковесные инструменты, для которых важна скорость, могут не использовать такие методы или использовать эвристики, пропускающие ошибки в некоторых случаях.

Одним из способов сокращения потенциального множества переменных, для которых возможны состояния гонки, является введение дополнительного анализа разделяемых данных. Этот анализ выдает множество потенциальных разделяемых переменных, которые могут быть вовлечены в состояние гонки. Основным вопросом остается консервативность данного анализа. Анализ разделяемых данных не способен отследить сложные случаи, связанные, например, с адресной арифметикой. Это означает, что он способен упустить некоторые разделяемые переменные, а значит, способен привести к пропуску ошибок. С другой стороны, если анализ разделяемых данных будет излишне консервативен, тогда результирующее множество разделяемых переменных будет не сильно отличаться от множества всех исходных переменных.

Второй важной проблемой статического анализа является значительно больший процент ложных предупреждений по сравнению с динамическим анализом. Рассмотрим основные причины, по которым инструмент может выдавать ложные предупреждения об ошибках. Предположим, что найдено некоторое потенциальное состояние гонки, которое описывается одним путем выполнения программы, включающем операции в нескольких потоках. Пусть инструмент выдал трассу, которая приводит к двум доступам к одной и той же памяти. Возникает вопрос, возможно ли такое выполнение программы в реальности. Далее мы подробно опишем те причины, которые приводят к наибольшему числу ложных предупреждений.

Первой причиной того, что предупреждение оказывается ложным, может быть то, что любой путь, ведущий к некоторому доступу к памяти, не может быть получен при реальном выполнении из-за несовместности некоторых условий ветвления. Например, не было учтено значение некоторой переменной. В таком случае говорят, что путь недостижим. Зачастую недостижимость может быть связана только с операциями в одном из потоков. Если из большого пути выполнения программы, включающем пути выполнения для нескольких потоков, можно выделить один локальный путь, который является недостижимым, тогда говорят о локальной недостижимости пути. В ином случае имеет место глобальная недостижимость. Существуют несколько подходов к решению задачи недостижимости, но так или иначе они все связаны с построением логических формул и доказательства их выполнимости. Как известно, задача SAT является NP-полной, и для сложного пути с большим

количеством переменных и особенно указателей время на решение этой задачи может становиться запредельным.

Глобально недостижимый путь возможен, если, например, на поведение одного потока влияет значение разделяемой переменной, которое устанавливается в другом. Вторым вариантом такого поведения может стать создание в разных ветках некоторого условного оператора двух потоков, в которых производится доступ к разделяемой области памяти. В этом случае потоки не могут выполняться параллельно, так как обе ветки условного оператора не могут выполниться одновременно, хотя локальный путь внутри созданных потоков будет достижимым. Такие случаи определить и исключить еще сложнее, чем локально-недостижимые пути.

Второй причиной ложных предупреждений является определение возможных примитивов синхронизации, которые могут быть использованы для организации доступа к разделяемым данным. В первую очередь все примитивы синхронизации также являются некоторыми объектами в памяти, на которые могут указывать сразу несколько указателей. Вопрос тождественности указателей снова приводит нас к необходимости анализа синонимов. Следующей проблемой могут стать различные условные действия, например, mutex_trylock. Поведение программы зависит от возможности захватить блокировку, но блокировка будет захвачена, только если эта функция вернет код успешного завершения (обычно 0). Для того чтобы можно было проверить, верно ли анализ учел код возврата и пошел по нужному пути, необходимо связать каким-то образом возвращаемое значение с захваченной блокировкой.

Третьей важной причиной ложных срабатываний является необходимость в анализе потоков. Зачастую программа сначала выполняет некоторые подготовительные действия (инициализацию) из одного потока, а затем создается необходимое количество вспомогательных потоков. В этом случае выдавать предупреждения для тех переменных, один из доступов к которым был из блока инициализации, не нужно. Задачей анализа потоков как раз и является определение тех участков кода, которые могут выполняться параллельно.

3. Идея метода

Как уже было отмечено, состоянием программы является состояние ее памяти, включающее в себя программный счетчик. Если выполнение программы возможно в несколько потоков, тогда в состояние памяти включается множество программных счетчиков для каждого потока управления. Будем говорить, что для состояния многопоточной программы можно взять проекцию на поток. Проекция включает в себя состояние памяти, которое доступно только конкретному потоку: его локальные переменные и общие разделяемые данные. Для того чтобы не абстрагироваться от деталей взаимодействия потоков между собой, далее будем рассматривать анализ на проекциях. При

таком анализе каждый поток рассматривается отдельно от других, которые являются для него окружением. Определим две проекции, как совместные, если существует такое глобальное состояние, проекциями которого они являются.

Состояние гонки — это такое состояние программы, в котором возможны два доступа к одной области памяти из разных потоков, хотя бы один из доступов является записью. Для того чтобы определить состояние гонки на проекциях, необходимо уже два состояния-проекции. Так, состоянием гонки будем называть такую пару проекций, которые являются совместными и в соответствующем им конкретном состоянии программы возможна гонка.

При анализе обычно рассматривают некоторые абстрактные состояния, включающие в себя некоторое множество состояний программы. Так как мы рассматриваем анализ на проекциях, то в дальнейшем под абстрактным состоянием мы будем понимать абстрактное состояние-проекцию.

Предлагаемый метод основан на хорошо известном алгоритме Lockset [8]. Этот алгоритм изначально был реализован в инструменте динамического анализа, в состоянии которого хранилось множество захваченных блокировок для каждого потока. В процессе анализа полученное множество сохранялось для каждого доступа к разделяемой памяти. Если находилась такая пара обращений к памяти, множества блокировок для которых не пересекались, то считалось, что состояния совместны и выдавалось предупреждение о состоянии гонки.

Наш анализ устроен похожим образом: в абстрактном состоянии сохраняется информация о захваченных в данный момент блокировках. После построения всего графа достижимых состояний для каждого доступа к памяти имеется абстрактное состояние, содержащее множество захваченных блокировок. Аналогично алгоритму Lockset совместность состояний определяется по пересечению множества блокировок.

Вычисление множества блокировок в процессе анализа устроено очень просто: если встречается функция захвата блокировки, то в абстрактное состояние добавляется соответствующая блокировка, если встречается функция освобождения, то блокировка удаляется.

Мы используем упрощенную модель памяти и считаем, что указатель указывает на область памяти, однозначно связанную с его именем: разные указатели указывают на разную память, а один и тот же указатель указывает на одинаковую память независимо от точки в программе. Кроме того, при анализе указателей, записанных в поля структур, учитывается только имя поля, но не имя указателя. Это означает, что анализ не сможет разделить память, на которую указывают два указателя A→a и B→a, так как имя поля у них одинаково, и будет считать, что они указывают на одну память. Стоит заметить, что поля не различаются только для структур одного типа, если же структуры A и B имеют разный тип, то будет считаться, что их поля, даже если они носят одно имя, указывают на различные области памяти.

Рассмотрим пример работы анализа, который отвечает за сбор блокировок (см. рис 1). Заметим, что на рисунке представлены только состояния программы, в

которых хранится информация о захваченных блокировках. В таком примере будет обнаружено состояние гонки: существует переход, осуществляющий доступ к глобальной переменной global из состояния, в котором отсутствуют любые блокировки, а также из состояния, в котором захвачена блокировка lock. Пересечение этих множеств пусто, значит, имеет место потенциальное состояние гонки.

```
1  int global;
2  int func(int var) {
3    if (var) {
4      lock();
5    }
6    global++;
7    if (var) {
8      unlock();
9    }
10 }
```

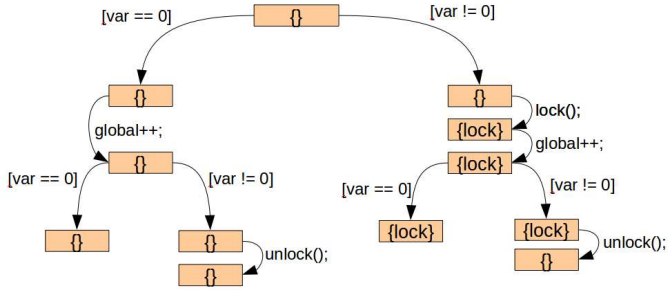

Рис. 1. Пример анализа примитивов синхронизации

Fig. 1. Example of analysis of synchronization primitives

Необходимо отметить, что в нашем инструменте используется эвристика для снижения количества ложных предупреждений об ошибках. Она основана на предположении, что случай, при котором работа с некоторыми разделяемыми данными не защищается вообще никогда, маловероятен. Обычно для большинства доступов применяется необходимая защита, и только в некоторых случаях она бывает пропущена. Таким образом, в соответствии с нашей эвристикой предупреждения не будут выдаваться для тех переменных, работа с которыми производится вообще без примитивов синхронизации, хотя, с точки зрения классического алгоритма Lockset, эта ситуация рассматривается как потенциальное состояние гонки.

Похожим образом устроен анализ разделяемых данных, задача которого состоит в отслеживании множества указателей, которые указывают на локальные данные в данной точке программы. Например, после выделения памяти с помощью функции malloc указатель не может указывать на разделяемые данные. В абстрактном состоянии для этого типа анализа хранится информация о том, какие доступные в данной точке программы области памяти являются локальными, а какие разделяемыми.

4. Схема реализации метода

Описываемый метод был реализован с использованием концепции CPA (англ. Configurable Program Analysis) [9]. Эта концепция конфигурируемого статического анализа позволяет легко запускать различные виды анализа

совместно: последовательно или параллельно. При этом каждый анализ способен самостоятельно выбирать баланс между точностью и затрачиваемыми ресурсами.

Рис. 2. Схема реализации метода

Fig. 2. Implementation of the method

Рассмотрим схему анализа, представленную на рис. 2. Сначала запускается анализ разделяемых данных. Результатом его работы является множество неразделяемых областей памяти. Дальнейший анализ для переменных, указывающих на память из этого множества, производиться не будет. После анализа разделяемых данных запускается параллельная композиция трех анализов: анализа примитивов синхронизации, анализа потоков и анализа предикатных абстракций. На этом шаге отслеживается и сохраняется информация об используемых примитивах синхронизации, а также определяются точки создания потоков. После завершения анализа программы, вычисляется множество предупреждений о потенциальных ошибках. Для всех обнаруженных потенциальных ошибок запускается следующий этап анализа — уточнение. На этой стадии производится более детальный анализ с целью отбросить ложные предупреждения. В первую очередь производится проверка логической выполнимости всех условий на пути. В случае, если один из путей, ведущих к ошибке, оказался ложным, например, не было учтено значение условия, то путь считается ложным и исключается из абстракции, а переменная, значение которой повлияло на недостижимость пути, добавляется в абстракцию, и дальнейший анализ будет учитывать ее значение. Эта операция называется уточнением абстракции. После этого анализ повторяется для более точной абстракции, то есть с учетом значений добавленных переменных. Новые найденные предупреждения также подвергаются проверке при уточнении. Так продолжается до тех пор, пока процесс не сойдется. Стоит заметить, что

операция уточнения абстракции позволяет в конечном итоге учитывать только те условия, которые влияют на достижимость потенциальных состояний гонки.

Теоретически можно придумать пример, на котором число необходимых уточнений будет бесконечным, однако на практике обычно встречаются программы, для анализа которых требуется некоторое конечное число уточнений. Однако для большого объема исходного кода одна итерация уточнения может занимать несколько десятков секунд. Поэтому обычно используется некоторое ограничение по времени на анализ, по истечении которого все найденные предупреждения, если только не было доказано, что они ложные, выдаются пользователю. Проведенные эксперименты показали, что во многих случаях ложные предупреждения отсеивались за небольшое количество итераций уточнения, а остальное время тратилось на то, чтобы подтвердить, что остальные предупреждения являются истинными с точки зрения достижимости пути.

Анализ разделяемых данных и анализ примитивов синхронизации был представлен ранее в статье [10]. Анализ предикатов является стандартным анализом, уже реализованным в концепции CPA [11]. Детальное описание процесса уточнения и анализа потоков будет представлено в соответствующих разделах.

5. Описание метода уточнения

Задачей этой части анализа является исключение ложных предупреждений, связанных с наличием недостижимых путей выполнения. Важно отметить, что здесь решается только задача локальной достижимости, и взаимодействие потоков друг с другом не учитывается.

Рассмотрим пример на рис. 1. В случае, если анализ не учитывает возможные значение переменных, то будут рассмотрены 4 пути выполнения. Один из них, в котором сначала захватывается блокировка (строка 4), а затем не освобождается, влияет на весь дальнейший анализ. Однако он является недостижимым, так как условие var является одинаковым в обоих ветвях условного оператора (строки 3 и 7). Соответственно, при реальном выполнении возможно только два пути выполнения, на каждом из которых в конце будет отсутствовать захваченная блокировка.

Метод уточнения, применяемый при поиске состояний гонки, основан на классическом алгоритме CEGAR — Counterexample Guided Abstraction Refinement [12]. Сначала опишем исходный алгоритм. Основная идея этого подхода заключается в том, что некоторое выбранное свойство доказывается не на исходной системе, которая обыкновенно устроена достаточно сложно, а на некоторой упрощенной ее модели, которая называется абстракцией. Такая абстракция может быть очень грубой, то есть упускать значительное число деталей, но она должна оставаться корректной. Другими словами все

состояния, достижимые в исходной системе, должны иметь соответствующие достижимые состояния в абстракции.

Число состояний в упрощенной модели — абстракции — обыкновенно значительно меньше, что упрощает анализ, но из-за грубости модели могут возникнуть ложные предупреждения об ошибках. В таком случае строится контрпример — пример пути, который может привести к ошибке. После этого выполняется его проверка на исходной системе. Если он выполним на исходной системе, значит, найденная ошибка истинная. В противном случае это означает, что контрпример был получен из-за неточности абстракции. Тогда абстракция уточняется на основе этого контрпримера, то есть в нее добавляются те детали, из-за отсутствия которых был получен данный контрпример. После этого анализ возобновляется на уточненной абстракции. Цикл повторяется до тех пор, пока либо не будет доказана корректность системы относительно выбранного свойства, либо не будет найдена ошибка.

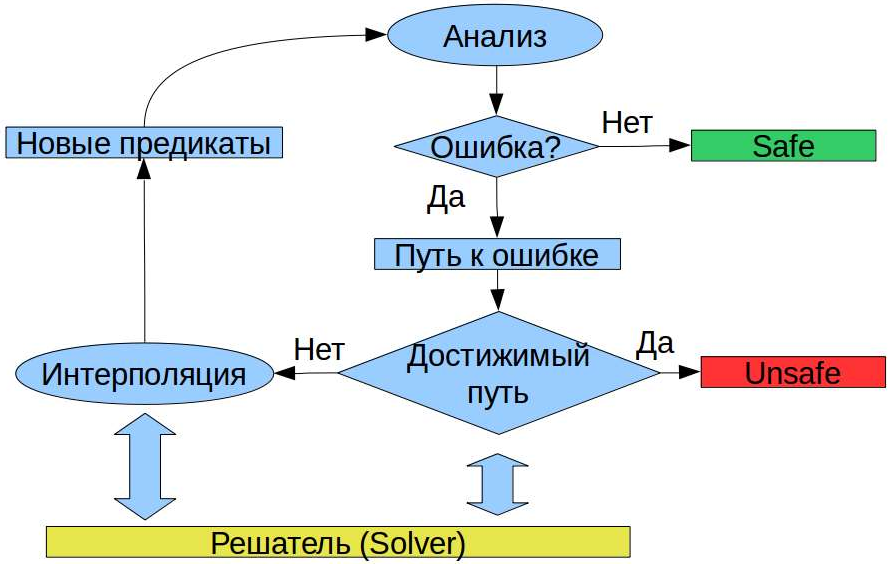

Рис. 3. Схема традиционного метода CEGAR

Fig. 3. CEGAR (Counterexample Guided Abstraction Refinement)

Абстракцию программы можно делать различными способами, но наиболее часто встречается предикатная абстракция, основанная на разбиении множества состояний программы (значений ее переменных) на подмножества с одинаковым значением выбранных предикатов. Можно выделить декартову абстракцию, в которой предикаты комбинируются только с помощью конъюнкции, и булеву абстракцию, в которой возможна комбинация произвольных логических формул.

Для проверки контрпримера и пересчета абстракции необходимо представить последовательность инструкций исходной программы в виде некоторых логических формул. Для этого используются различные техники, такие как пред- и постусловия или формулы пути на основе представления SSA. Для проверки этих формул на выполнимость используются различные SMT-решатели (англ. Satisfiability Modulo Theories). В случае, если формула пути является невыполнимой, необходимо извлечь из нее получить те условия, которые в последствии будут добавлены к анализу. Эти условия выражаются предикатами. Существуют различные способы для выделения необходимых предикатов из формулы пути, например, синтаксические методы [13] и интерполяция Крейга [14, 15].

Традиционный метод CEGAR успешно реализован во многих инструментах статической верификации. Там он успешно применяется для решения задачи достижимости, при которой производится проверка достижимости некоторой ошибочной метки, которая присутствует в исходном коде, например, вызова специальной функции. Процесс уточнения запускается, если был найден путь, ведущий от некоторой точки входа до этой ошибочной метки.

Теперь перейдем к описанию модифицированного алгоритма CEGAR, который применяется для уточнения найденных потенциальных состояний гонки. Для наличия состояния гонки требуется два совместных состояния. Необходимым условием для этого является наличие двух локально-достижимых состояний. Соответственно, для каждого потенциального состояния гонки существуют как минимум два локальных пути, которые необходимо проверить на достижимость. Заметим, что каждый путь по отдельности не может считаться ошибочным, так как доступ к разделяемым данным без блокировок сам по себе может не быть ошибкой, необходимо, чтобы нашелся второй доступ к этим же данным.

При уточнении примера ложного пути, рассмотренного выше, будет построена формула пути, в которой встретится такая часть $[var == 1] \wedge [var == 0]$. Полученная формула будет передана специальному компоненту — решателю (англ. Solver), который выдаст вердикт, что формула невыполнима. Вместе с вердиктом решатель выдаст интерполянты, которые по сути являются противоречивой частью формулы. Для нашего примера интерполянтом является, например, $[var == 0]$. При дальнейшем анализе значение этой переменной будет учитываться, и подобная недостижимая трасса больше не появится. На рис 4 представлен граф абстрактных состояний для анализа предикатов и анализа примитивов синхронизации для программы, изображенной на рис. 1. В фигурных скобках будем заключать абстрактное состояние анализа примитивов синхронизации, а в квадратных скобках – абстрактное состояние анализа предикатов.

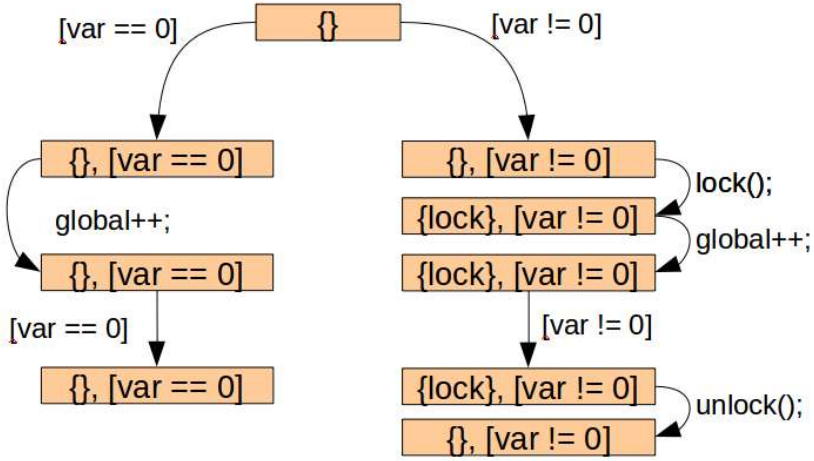

Рис 4. Пример анализа предикатов совместно с анализом примитивов синхронизации

Fig. 4. Example of predicate analysis together with analysis of synchronization primitives

Потенциально в программе может быть не одно состояние гонки. Тогда существует два варианта поведения при обнаружении потенциальной ошибки: переход к уточнению сразу или продолжение анализа для поиска следующих ошибок. Рассмотрим эти две стратегии подробнее.

1. Уточнение после обнаружения потенциального состояния гонки. При таком способе анализ продолжается до тех пор, пока не будет найдена пара совместных состояний с доступами к одним разделяемым данным. После обнаружения такой пары запускается уточнение, которое проверяет, достижим ли каждый из путей по-отдельности. Такой способ, в отличие от предыдущего, производит только необходимые проверки, и не получит излишне точную абстракцию. Однако, большая проблема заключается в том, что нужно определить, какие еще доступы к переменным следует удалить, если путь оказался недостижим. В общем случае доступ к одним данным в одной и той же строке исходного кода возможен по разным путям. Допустим, путь к некоторому доступу к переменной оказался ложным. Тогда, некоторое множество доступов к другим данным на этом же пути тоже могут оказаться недостижимыми, значит, их следует удалить. Но возможно, что к некоторым существуют другие истинные пути. В итоге для определения множества недостижимых доступов к данным, необходимо заново произвести анализ программы целиком, что сводит на нет все преимущества данного подхода.

2. Уточнение всех полученных предупреждений после полного анализа программы. В этом варианте весь анализ проводится целиком, после

чего уточняются все полученные пути, приводящие к состояниям гонки. При необходимости анализ повторяется, при этом вся информация о доступах к памяти собирается заново, таким образом, этот способ не имеет проблем с удалением зависимых доступов к памяти. Минусом данного подхода является большой объем повторяющейся работы. Так, для некоторого класса путей, например, проходящих через одну функцию, результат уточнения может быть одинаков, и нет необходимости проводить уточнение для каждого. Кроме того, могут быть участки кода, в которых не было найдено ни одного предупреждения, тогда их не нужно анализировать заново.

Были проведены некоторые эксперименты и с одной стратегией уточнения и с другой. И тот, и другой метод показал неплохие результаты, однако второй вариант обладал большей гибкостью: можно было остановить весь анализ при достижении некоторого уровня точности абстракции. Тогда как первый вариант требовал построения детальной абстракции для получения результатов.

Описанный метод уточнения позволяет исключать из выдаваемых предупреждений такие, которые образованы локально-недостижимыми путями. Однако он требует значительного количества времени, так как для исключения всех ложных путей требуется достаточно точная абстракция, что означает наличие большого количества предикатов. Например, если на пути встретился цикл, то при уточнении придется рассматривать все его итерации. По результатам экспериментов можно сказать, что многие ложные предупреждения отбрасываются достаточно быстро, а большую часть времени уточнения (~99%) тратится на то, чтобы доказать истинность остальных предупреждений.

6. Анализ потоков

Зачастую возникают ситуации, при которых доступ к некоторым разделяемым данным допускается без использования явных примитивов синхронизации. Основная причина заключается в том, что в этот момент возможно выполнение только единственного потока. Например, при инициализации данных одним потоком другие, как правило, даже не созданы. Однако, наш анализ не учитывал зависимость между точками создания потоков и считал, что все они созданы одновременно. Это порождало большое число ложных срабатываний. Рассмотрим такой пример: один поток порождает другой. Очевидно, что оба эти потока могут работать параллельно, вместе с этим часть кода первого потока до создания второго потока никак не может быть выполнена параллельно. Задачей разработанного анализа потоков является вычисление множества точек программы, которые потенциально могут работать параллельно друг с другом.

Рассмотрим пример искусственного ложного срабатывания.


```
1 int global;
2
3 int worker(void* arg) {
4   …
5   global++;
6   …
7 }
```

```
8  int start() {
9    …
10   global = 0;
11   …
12   pthread_create(&thread, .., worker, ..);
13   …
14   pthread_join(&thread);
15   result = global;
16   …
17 }
```

В этом примере стартовая функция start выполняет некоторую инициализацию, а затем создает поток, в котором происходит работа с глобальной переменной. Так как поток не может выполниться до того, как он будет создан, одновременный доступ к глобальной переменной невозможен. Этот пример схематично можно изобразить на следующем рисунке (рис. 5).

На рисунке вертикальными линиями показано выполнение потоков, а горизонтальный стрелки означают создание нового потока и его завершение.

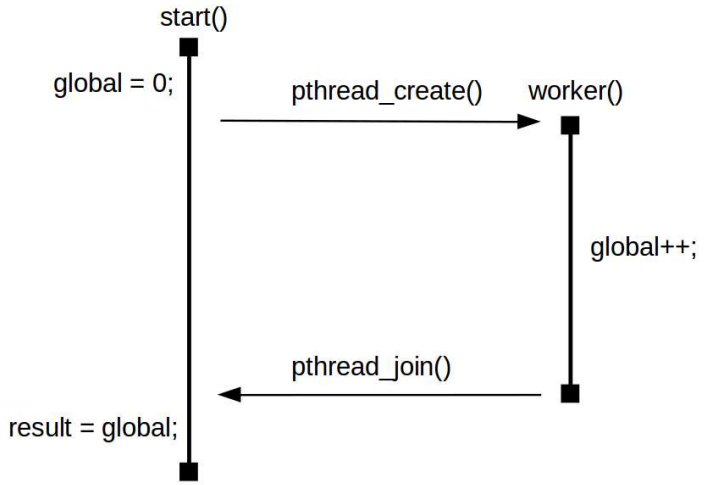

Рис. 5. Схема параллельной работы функций

Fig. 5. Example of parallel execution of functions

Абстрактное состояние содержит в себе множество меток, которое описывает множество активных в данный момент потоков. Совместность состояний определяется как наличие совместных меток, что означает, что состояния соответствуют коду, работающему параллельно. Множество меток модифицируется в точках создания или завершения потоков. Метка

идентифицируется уникальным именем, привязанным к потоку, и имеет две части, отличающиеся друг от друга бинарным флагом. При создании потока в множество меток родительского потока добавляется флаг метки 0, а в множество меток дочернего потока — флаг метки 1. Рассмотрим пример: пусть создается некоторый поток. Ему присваивается уникальный идентификатор 2. Тогда во множество меток родительского потока заносится метка 2.0. А во множество меток созданного потока заносится метка 2.1. Два оператора могут быть выполнены параллельно друг с другом, если во множестве меток, им соответствующим, найдутся различные флаги одной метки. Для рассмотренного примера это означает, что состояния, в которых есть метка 2.0 несовместны, а если в одном из состояний есть метка 2.0, а в другом 2.1, то такие состояния являются совместными. По сути это означает, что в какой-то момент был создан некоторый поток, после этого родительский поток пришел к одному использованию, а дочерний — к другому.

Гораздо сложнее ситуация с операцией ожидания завершения потоков (оператора join). В общем случае алгоритм распространения меток становится достаточно сложным, поэтому мы рассмотрим упрощенный вариант, при котором делается предположение, что поток завершается в том же родительском потоке, который его и создал. В таком случае при завершении потока соответствующая метка удаляется из множества меток родительского потока. При рассматриваемых ограничениях (операция join выполняется из родительского потока) в состоянии родительского потока должна быть метка завершающегося потока с флагом 0, а в дочернем потоке — метка с флагом 1. Никаких других меток быть не должно, кроме тех, которые уже есть в родительском состоянии. Например, возможна такая ситуация. Родительский поток с идентификатором 1, имея множество таких меток {1.1, 2.0}, выполняет операцию join для дочернего потока с идентификатором 2, который имеет состояние {1.1, 2.1}. Результатом этой операции будет множество меток {1.1}.

Теперь более формально опишем алгоритм вычисления множества меток. В состоянии для этого типа анализа хранится два множества: Tset и Rset. Первое — это множество активных меток, а второе — это множество удаленных меток. Нам потребуется расширенная операция пересечения множеств меток, которую будем обозначать так:

$$Tset_1 * Tset_2 = \{Lid : \exists(Lid, flag_1) \in Tset_1 \wedge \exists(Lid, flag_2) \in Tset_2 \wedge (flag_1 \neq flag_2)\}$$

Эта операция возвращает множество тех меток, у которых различаются флаги в пересекаемых множествах. При создании потока (операция pthread_create) в новое множество Tset родительского потока добавляется метка с флагом 0, а во множество Tset дочернего потока — метка с флагом 1. Множество Rset при этой операции не изменяется. При операции ожидания завершения потока (pthread_join) новое значение множеств $\hat{Tset}$ и $\hat{Rset}$ вычисляются следующим образом:

$$\hat{Rset} = Rset_1 \cup Rset_2 \cup (Tset_1 * Tset_2),$$
$$\hat{Tset} = \{(Lid, flag) : Lid \in (Tset_1 \cup Tset_2) / \hat{Rset}\},$$

где $Tset_1, Tset_2$ — это множества из двух состояний родительского потока и потока, который завершается.

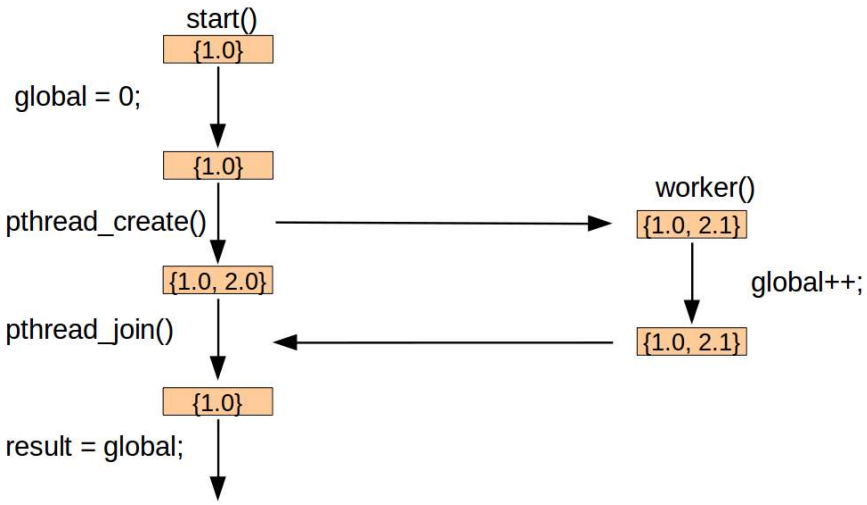

Рис. 6. Пример состояний анализа потоков

Fig. 6. Example of abstract states in thread analysis

Для рассмотренного примера ложного предупреждения анализ получит, что для первого доступа к этому полю будет вычислено следующее множество меток {1.0}, означающее, что в этот момент работает только один поток. Для доступа к этому полю из функции worker() будет получено множество {1.0, 2.1}, означающее, что в этот момент работает два потока. Однако эти два состояния несовместны, так как в них нет метки с различными флагами (метка для первого потока встречается в обоих состояниях с одним и тем же флагом). Отсюда следует, что предупреждение ложное.

Нужно отметить, что описанный метод может быть применен только в простых случаях, в которых поток завершается в том же потоке, в котором был создан. Более хитрые ситуации, в которых, например, родительский поток завершается в дочернем не рассматриваются. Теоретически метод расстановки меток позволяет проводить анализ таких программ, однако для этого требуется значительная переработка реализации инструмента. Однако, более сложным ситуациям довольно сложно найти применение в реальном программном обеспечении, поэтому такое ограничение реализованного метода выглядит достаточно естественным.

7. Эксперименты

Эксперименты были проведены на наборе задач, основанных на модулях ядра ОС Linux 4.5-rc1 подсистемы drivers/net/wireless. Для формирования задач из исходного кода использовалась инфраструктура инструмента LDV [16, 17], с помощью которой для драйвера формировалась модель окружения [18]. Были проведены запуски в четырех вариантах работы инструмента: с одним из анализов (анализ с уточнениями и анализ потоков), с обоими и без них (табл. 1).

В указанной подсистеме было проанализировано 113 модулей. Было несколько причин, которые приводили к незавершенному анализу, например, превышение ограничений по времени, но большая часть из них относилась к этапу подготовки модуля к верификации, то есть падению некоторых компонентов системы LDV.

По результатам видно, что анализ потоков не сильно увеличивает время на анализ, но значительно повышает расход памяти. А уточнение ведет себя с точностью до наоборот. Это связано с тем, что анализ потоков требует для себя отдельного пространства состояний, а уточнение — лишь проверяет, правильно ли вычислены достижимые состояния.

Для запуска с двумя включенными дополнительными анализами был проведен анализ результатов. Все 5 найденных предупреждений связаны с тем, что некоторые данные достаются из разделяемого списка под блокировками, при этом они удаляются из списка, а после этого работа с ними осуществляется без блокировок. Таким образом, данные случаи не являются состояниями гонки.

Табл. 1. Результаты запуска инструмента на drivers/net/wireless/

Table 1. Results of launch of the tool on drivers/net/wireless/

	Предупреждения	Незавершенный анализ	Корректные модули	Время, ч	Память, Гб
+ Потоки, + уточнение	5	61	51	3.2	8.1
- Потоки, + уточнение	6	67	44	4.1	4.0
+ Потоки, - уточнение	27	57	49	2.3	8.2
- Потоки, - уточнение	186	54	43	2.1	3.5

Был проведен запуск инструмента на наборе задач, подготовленных на основе всей папке drivers/ ядра ОС Linux, в конфигурации с анализом потоков и уточнением. Было получено 2219 предупреждений. Наиболее важной причиной

ложных срабатываний стало несовершенство модели окружений, из-за которой было получено более 50 % ложных предупреждений. Например, некоторые функции-обработчики драйвера при реальном выполнении вызываются под блокировками, в то время как в модели этот факт не учитывается. Кроме того зачастую возникают неявные зависимости между обработчиками разных структур, которые не позволяют им выполняться одновременно и которые также не учитываются в модели окружения. Следует еще раз отметить, что модель окружения относится к этапу подготовки задачи, а не к методу анализу непосредственно.

Возникали ситуации, в которых структуры одного типа использовались для разного назначения, которое предполагало наличие различных блокировок. Таким образом, упрощенная модель памяти, которую используется в методе, принесла около 10 % ложных предупреждений.

Еще 10% ложных предупреждений связаны с ситуациями, в которых из разделяемого множества достается объект под блокировками, а работа с ним осуществляется без них. Около 10% связаны с неточностями анализа: это и несовершенство анализа функциональных указателей, и проблемы в анализе примитивов синхронизации (в т. ч. отсутствие некоторых блокировок в конфигурации), и неточности в анализе разделяемых данных.

Около 15% предупреждений оказались истинными. Следует отметить, что зачастую для одного драйвера, содержащего состояние гонки, может выдаваться более 10 предупреждений на разные переменные. То есть, предупреждение означает лишь проявление ошибки. Так, найденные 290 предупреждений соответствовали лишь 32 ошибкам. Эти ошибки были сообщены разработчикам. Часть из них были признаны, некоторые даже исправлены, и на некоторые сообщения об ошибках ответ не был получен. Список исправленных ошибок приведен на странице http://linuxtesting.org/results/ldv, категория data race.

Таким образом, представленный метод поиска состояний гонок в операционных системах продемонстрировал практически значимые результаты с приемлемым уровнем ложных срабатываний. Являясь, по сути, легковесным методом, он допускает более гибкую настройку баланса между ресурсами и точностью анализа.

Результаты анализа причин ложных срабатываний позволяют сделать вывод, что основные усилия при применении инструмента к модулям ядра операционной системы Linux необходимо направить на повышение точности используемых моделей окружения. Развитие собственно предложенного метода может продолжаться в нескольких направлениях: повышение точности входящих в него анализов и применяемой модели памяти, поддержка новых примитивов синхронизации, например, RCU, применение новых подходов с целью ускорения анализа. Кроме того, отдельной задачей является исследование возможности практического применения разработанного инструмента к другим классам целевых программ.

Список литературы

[1]. Мутилин В.С., Новиков Е.М., Хорошилов А.В. Анализ типовых ошибок в драйверах операционной системы Linux. Труды ИСП РАН, том 22, 2012 г., стр. 349-374. DOI: 10.15514/ISPRAS-2012-22-19

[2]. Nancy Levenson, Safeware: System Safety and Computers, 1995

[3]. Nachum Dershowitz, Software horror stories, http://www.cs.tau.ac.il/~nachumd/pub.html

[4]. Cordeiro, L., Fischer, B.: Verifying Multi-Threaded Software using SMT-based Context-Bounded Model Checking. In: ICSE, pp. 331–340 (2011)

[5]. E. Clarke, D. Kroening, N. Sharygina, K. Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05), pp. 570-574, 2005.

[6]. A. Gupta, C. Popeea, A. Rybalchenko. Predicate abstraction and refinement for verifying multi-threaded programs In Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2011), pp. 331-344, 2011.

[7]. A. Gupta, C. Popeea, A. Rybalchenko. Threader: a constraint-based verifier for multi-threaded programs In Proceedings of the 23rd International Conference on Computer Aided Verification (CAV 2011), LNCS, vol. 6806, pp. 412-417, 2011.

[8]. Stefan Savage, Michael Burrows, Greg Nelson, Patric Sobalvarro, Thomas Anderson Eraser: A Dynamic Data Race Detector for Multithreaded Programs ACM Transactions on Computer Systems, Vol. 15, No. 4, November 1997, Pages 391–411.

[9]. Dirk Beyer, Thomas A. Henzinger, and Gregory Theoduloz, Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis, ACM Transactions on Computer Systems, Vol. 15, No. 4, November 1997, Pages 391–411.

[10]. Андрианов П.С., Мутилин В.С., Хорошилов А.В. Метод легковесного статического анализа для поиска состояний гонок. Труды ИСП РАН, том 27, вып. 5, 2015 г., стр. 87-116. DOI: 10.15514/ISPRAS-2015-27(5)-6

[11]. Dirk Beyer , M. Erkan Keremoglu , Philipp Wendler, Predicate abstraction with adjustable-block encoding, Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design, October 20-23, 2010, Lugano, Switzerland

[12]. Мандрыкин М.У., Мутилин В.С., Хорошилов А.В. Введение в метод CEGAR — уточнение абстракции по контрпримерам. Труды ИСП РАН, том 24, 2013, стр. 219-292. DOI: 10.15514/ISPRAS-2013-24-12

[13]. Shostak R., Deciding Combinations of Theories, Journal of the ACM, 1-12, 1984

[14]. Beyer Dirk, Zuerey Damien, Majumdar Rupak., Csisat: Interpolation for la+euf, CAV, 304-308, 2008

[15]. Bruttomesso Roberto, Cimatti Alessandro, Franzen Anders, Griggio Alberto, Sebastiani Roberto., The mathsat 4 smt solver, CAV, 299-303, 2008

[16]. Alexey Khoroshilov, Mikhail Mandrykin, Vadim Mutilin, Eugene Novikov, Alexander Petrenko, Ilya Zakharov. Configurable toolset for static verification of operating systems kernel modules. Programming and Computer Software, vol. 41, № 1, 2015, pp. 49-64. DOI: 10.1134/S0361768815010065

[17]. Alexey Khoroshilov, Mikhail Mandrykin, Vadim Mutilin, Eugene Novikov, Pavel Shved. Using Linux Device Drivers for Static Verification Tools Benchmarking. Programming and Computer Software, vol. 38, № 5, 2012, pages 245-256. DOI: 10.1134/S0361768812050039

[18]. Alexey Khoroshilov,Vadim Mutilin, Ilya Zakharov. Pattern-based environment modeling for static verification of Linux kernel modules. Programming and Computer Software, vol. 41, № 3, 2015, pages 183-195. DOI: 10.1134/S036176881503007X

Adjustable method with predicate abstraction for detection of race conditions in operating systems[2]

[1]*P.S. Andrianov <andrianov@ispras.ru>,*
[1]*V.S. Mutilin <mutilin@ispras.ru>,*
[1,2,3,4]*A.V. Khoroshilov <khoroshilov@ispras.ru>,*
[1]*Institute for System Programming of the RAS,*
25 Alexander Solzhenitsyn Str., Moscow, 109004, Russian Federation
[2]*Lomonosov Moscow State University,*
GSP-1, Leninskie Gory, Moscow, 119991, Russia
[3]*Moscow Institute of Physics and Technology (State University)*
9 Institutskiy per., Dolgoprudny, Moscow Region, 141700, Russia
[4]*National Research University Higher School of Economics (HSE)*
20 Myasnitskaya Ulitsa, Moscow, 101000, Russia

Abstract. The paper presents a configurable method of static data race detection that is trying to keep a balance between resource consumption and a number of false alarms. The method is based on well known Lockset approach. It uses simplified memory model to be fast enough. At the same time it includes advanced techniques aimed to achieve acceptable false alarms rate. The key techniques are thread analysis and predicate abstraction based refinement. The method was implemented in CPALockator tool built on top of CPAchecker framework. The tool was evaluated on Linux kernel modules and it has detected several actual data races, which were approved by developers and were fixed in upstream Linux kernel.

Key words: static analysis; race condition; kernel of operating system.

DOI: 10.15514/ISPRAS-2016-28(6)-5

For citation: Andrianov P.S., Mutilin V.S., Khoroshilov A.V. Adjustable method with predicate abstraction for detection of race conditions in operating systems. Trudy ISP RAN/Proc. ISP RAS, vol. 28, issue 6, 2016, pp. 65-86 (in Russian). DOI: 10.15514/ISPRAS-2016-28(6)-5

References

[1]. Mutilin V.S., Novikov E.M., Khoroshilov A.V. Analysis of typical faults in Linux operating system drivers. Trudy ISP RAN / Proc. ISP RAS, vol. 22, 2012, pp. 349–374 (in Russian). DOI: 10.15514/ISPRAS-2012-22-19
[2]. Nancy Levenson, Safeware: System Safety and Computers, 1995

[2] The research was carried out with funding from the Ministry of Education and Science of Russia (the project unique identifier is RFMEFI61614X0015)

[3]. Nachum Dershowitz, Software horror stories, http://www.cs.tau.ac.il/~nachumd/pub.html
[4]. Cordeiro, L., Fischer, B.: Verifying Multi-Threaded Software using SMT-based Context-Bounded Model Checking. In: ICSE, pp. 331–340 (2011)
[5]. E. Clarke, D. Kroening, N. Sharygina, K. Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05), pp. 570-574, 2005.
[6]. A. Gupta, C. Popeea, A. Rybalchenko. Predicate abstraction and refinement for verifying multi-threaded programs In Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2011), pp. 331-344, 2011.
[7]. A. Gupta, C. Popeea, A. Rybalchenko. Threader: a constraint-based verifier for multi-threaded programs In Proceedings of the 23rd International Conference on Computer Aided Verification (CAV 2011), LNCS, vol. 6806, pp. 412-417, 2011.
[8]. Stefan Savage, Michael Burrows, Greg Nelson, Patric Sobalvarro, Thomas Anderson Eraser: A Dynamic Data Race Detector for Multithreaded Programs ACM Transactions on Computer Systems, Vol. 15, No. 4, November 1997, Pages 391–411.
[9]. Dirk Beyer, Thomas A. Henzinger, and Gregory Theoduloz, Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis, ACM Transactions on Computer Systems, Vol. 15, No. 4, November 1997, Pages 391–411.
[10]. Andrianov P.S., Mutilin V.S., Khoroshilov A.V. Lightweight Static Analysis for Data Race Detection in Operating System Kernels. Trudy ISP RAN / Proc. ISP RAS, vol. 27, 2015, pp. 87-116 (in Russian). DOI: 10.15514/ISPRAS-2015-27(5)-6
[11]. Dirk Beyer , M. Erkan Keremoglu , Philipp Wendler, Predicate abstraction with adjustable-block encoding, Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design, October 20-23, 2010, Lugano, Switzerland
[12]. Mandrykin M.U., Mutilin V.S., Khoroshilov A.V. Introduction to CEGAR — Counter-Example Guided Abstraction Refinement. Trudy ISP RAN / Proc. ISP RAS, vol. 24, 2013, pp. 219-292 (in Russian). DOI: 10.15514/ISPRAS-2013-24-12
[13]. Shostak R., Deciding Combinations of Theories, Journal of the ACM, 1-12, 1984
[14]. Beyer Dirk, Zuerey Damien, Majumdar Rupak., Csisat: Interpolation for la+euf, CAV, 304-308, 2008
[15]. Bruttomesso Roberto, Cimatti Alessandro, Franzen Anders, Griggio Alberto, Sebastiani Roberto., The mathsat 4 smt solver, CAV, 299-303, 2008
[16]. Alexey Khoroshilov, Mikhail Mandrykin, Vadim Mutilin, Eugene Novikov, Alexander Petrenko, Ilya Zakharov. Configurable toolset for static verification of operating systems kernel modules. Programming and Computer Software, vol. 41, № 1, 2015, pp. 49-64. DOI: 10.1134/S0361768815010065
[17]. Alexey Khoroshilov, Mikhail Mandrykin, Vadim Mutilin, Eugene Novikov, Pavel Shved. Using Linux Device Drivers for Static Verification Tools Benchmarking. Programming and Computer Software, vol. 38, № 5, 2012, pages 245-256. DOI: 10.1134/S0361768812050039
[18]. Alexey Khoroshilov,Vadim Mutilin, Ilya Zakharov. Pattern-based environment modeling for static verification of Linux kernel modules. Programming and Computer Software, vol. 41, № 3, 2015, pages 183-195. DOI: 10.1134/S036176881503007X