DOI: 10.15514/ISPRAS-2019-31(5)-16

Анализ корректности синхронизации компонентов ядра операционных систем

П.С. Андрианов, ORCID: 0000-0002-6855-7919 <andrianov@ispras.ru>

Институт системного программирования им. В.П. Иванникова РАН, 109004, Россия, г. Москва, ул. А. Солженицына, д. 25

Аннотация. Большинство современных инструментов статической верификации плохо масштабируются на сложное программное обеспечение. Целью работы была разработка инструмента, который станет золотой серединой между точными и медленными инструментами статической верификации и быстрыми, но менее точными инструментами статического анализа. Основной идеей подхода является абстракция от точного взаимодействия потоков и анализ каждого потока отдельно от всех остальных, но в некотором окружении, которое моделирует влияние потоков друг на друга. Окружение содержит описание возможных действий над разделяемыми данными и примитивами синхронизации, а также условий их применения. Варьируя точность построения окружения, можно добиваться необходимого баланса между скоростью и точностью анализа в целом. Формальное описание предлагаемого подхода было сделано с использованием теории адаптивного статического анализа. Это позволило сформулировать условия и доказать корректность предлагаемого подхода в этих условиях. Для эффективного поиска состояний гонки используется специальная модель памяти, которая позволяет разделять области памяти на непересекающиеся регионы, соответствующие типам данных. Реализация предложенного подхода во фреймворке CPAchecker позволяет переиспользовать существующие техники анализа с минимальными изменениями. А реализация дополнительных техник анализа в рамках предложенной теории позволяет повысить точность анализа. Результаты проведенных экспериментов на двух наборах тестовых задач позволяют заключить о масштабируемости и практической применимости метода.

Ключевые слова: состояние гонки; раздельный анализ потоков; статическая верификация; операционная система Linux

Для цитирования: Андрианов П.С. Анализ корректности синхронизации компонентов ядра операционных систем. Труды ИСП РАН, том 31, вып. 5, 2019 г., стр. 203-232. DOI: 10.15514/ISPRAS-2019-31(5)-16

Analysis of correct synchronization of operating system components

P.S. Andrianov, ORCID: 0000-0002-6855-7919 <andrianov@ispras.ru>

Ivannikov Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia

Abstract. Most of the software model checker tools do not scale well on complicated software. Our goal was to develop a tool, which provides an adjustable balance between precise and slow software model checkers and fast and imprecise static analyzers. The key idea of the approach is an abstraction over the precise thread interaction and analysis for each thread in a separate way, but together with a specific environment, which models effects of other threads. The environment contains a description of potential actions over the shared data and synchronization primitives, and conditions for its application. Adjusting the precision of the environment, one can achieve a required balance between speed and precision of the complete analysis. A formal description of the suggested approach was performed within a Configurable Program Analysis theory. It allows formulating assumptions and proving the soundness of the approach under the assumptions. For efficient data race detection we use a specific memory model, which allows to distinguish memory domains into the disjoint set of regions, which correspond to a data types. An implementation of the suggested approach into the CPAchecker framework allows reusing an existed approaches with minimal changes. Implementation of additional techniques according to the extended theory allows to increase the precision of the analysis. Results of the evaluation allow confirming scalability and practical usability of the approach.

Keywords: Data race; Thread-Modular approach; Software verification; Linux kernel

For citation: Andrianov P.S. Analysis of correct synchronization of operating system components. Trudy ISP RAN/Proc. ISP RAS, vol. 31, issue 5, 2019, pp. 203-232 (in Russian). DOI: 10.15514/ISPRAS-2019-31(5)-16

1. Введение

Верификация многопоточных программ всегда являлась более сложной задачей, чем верификация последовательных программ. Точное вычисление всех возможных чередований (interleavings), приводит к комбинаторному взрыву числа состояний. Поэтому, большинство инструментов статической верификации применяют различные техники оптимизации: редукция частичных порядков (partial order reduction) [1,2], абстракция счетчика (counter abstraction) [3] и другие. Тем не менее, большинство современных инструментов статической верификации плохо масштабируются на промышленное программное обеспечение. Этот факт подтверждается результатами сравнения инструментов статической верификации на наборе задач SV-COMP [4]. Задачи из категории «многопоточность», основанные на драйверах операционной системы Linux, вызывают значительные сложности для всех инструментов статической верификации.

Одной из альтернатив методам проверки моделей являются методы статического анализа, которые нацелены на быстрый поиск ошибок без абсолютной уверенности в финальном вердикте. Такие инструменты применяют различные фильтры и эвристики для ускорения анализа и поэтому не могут гарантировать корректность, то есть отсутствие ошибок. В данной работе представлен подход к статической верификации многопоточного программного обеспечения, который позволяет выбирать баланс между скоростью и точностью проводимого анализа.

Суть предлагаемого подхода состоит в следующем. Поскольку объектом верификации будет большая многопоточная программа, мы заранее отказываемся от анализа всей программы с учетом всех возможных взаимодействий потоков и рассматриваем каждый поток по-отдельности. В этом случае состояния каждого потока становятся *частичными*, то есть они не содержат информацию о других потоках и, следовательно, не могут описать полное состояние всей программы. Возможное влияние потоков друг от друга аппроксимируется сверху множеством действий, которое потоки могут совершать над разделяемыми данными, в том числе примитивами синхронизации.

Таким образом, аппроксимация возможных действий, или *эффектов*, формируется одновременно для всех потоков и называется *окружением*. Несмотря на то, что окружение является единым для всех потоков, отсюда не следует, что все входящие в него эффекты будут применены к некоторому потоку, так как для каждого из эффектов определяются условия его применения, которые зависят от используемого анализа. Кроме того, в условия применения эффекта могут включаться требования на конкретные операторы и состояния потока.

Точность построения окружения определяет во многом точность и скорость работы всего инструмента. Точность анализа можно повысить, комбинируя различные техники анализа. Для реализации этой идеи была использована платформа CPAchecker [5,6],

которая предоставляет богатый набор техник верификации. Анализ с раздельным рассмотрением потоков (thread-modular approach) [7-10] также можно реализовать как одну из техник, которая встраивается в CPAchecker и пополняет традиционный набор техник верификации таких как, например, CEGAR [11] и предикатная абстракция [12].

Эффективное расширение платформы CPAchecker требует не просто добавления еще одного вида анализа. В идеале нужно, чтобы максимальное число видов анализа могли работать одновременно и обмениваться данными между собой. Необходимым условием такой тесной интеграции является либо следование уже определенной теории CPA, либо модификация имеющейся теории таким образом, что старая теория оказывалась частным случаем новой. Именно такая задача ставилась в данном исследовании.

Поиск состояний гонки обычно состоит из двух основных этапов:

1. построение множества достижимых состояний;
2. нахождение в построенном множестве специальных состояний, образующих состояние гонки.

Эти два шага могут выполняться последовательно или параллельно в зависимости от инструмента статической верификации. Например, некоторые инструменты статической верификации при добавлении каждого следующего состояния проверяют, не образует ли оно состояние гонки с некоторым уже достижимым ранее состоянием анализа. И в случае обнаружения ошибки, такой анализ останавливает свое выполнение. Однако, такой способ является слишком медленным и непрактичным для поиска состояний гонки, хотя он успешно применяется, например, для решения задач достижимости или поиска ошибок, связанных с некорректным использованием памяти в последовательных программах.

Инструменты статического анализа, которые ищут потенциальные состояния гонки, обычно используют Lockset алгоритм для поиска таких ошибок. В предложенном подходе используется более точный алгоритм, в котором потенциальное состояние гонки является парой *совместных* переходов, которые модифицируют одну и ту же память. Совместность здесь означает, что два частичных состояния двух потоков могут быть частью одного глобального состояния. Таким образом, если рассматривать только абстракцию над примитивами синхронизации, это сводится к алгоритму Lockset, но при использовании других вариантов анализа, является более точным. Предикатная абстракция вместе с моделью памяти BnB [14-16] позволяет значительно улучшить работу с доступами по указателю и позволяет сохранить корректность при разумных предположениях.

Оценка подхода производилась на множестве задач, основанных на драйверах операционной системы Linux. Они были подготовлены с помощью системы Klever, которая позволяет проводить верификацию больших программных систем [17, 18]. Klever разделяет большой объем кода на отдельные фрагменты – верификационные задачи – и подготавливает для них модель окружения.

Основным вкладом данной работы является:

1. развитие теории CPA, которая позволяет комбинировать технику thread-modular с другими подходами, такими как предикатная абстракция;
2. реализация предложенной теории в инструменте CPAchecker, который был успешно апробирован на множестве задач, основанных на драйверах операционной системы Linux.

Статья организована следующим образом. В разд. 2 представлены основные сложности современных инструментов статической верификации и основы предлагаемого подхода. В разд. 3 представлена основная идея подхода. Разд. 4 вводит основные определения и модель программы. Следующие 7 разделов посвящены описанию расширения теории CPA: разд. 6 описывает thread-modular подход в терминах CPA, разд. 7 – 12 содержат расширенное описание основных анализов (CPA). В разд. 13 описаны основные

особенности поиска состояний гонки в предлагаемом подходе. В разд. 14 представлены результаты работы инструмента на наборе SV-COMP и драйверах операционной системы Linux. В разд. 15 представлен краткий обзор родственных работ.

2. Пример запуска существующих инструментов верификации

Рассмотрим пример верификационной задачи[1] из SV-COMP'19 [4]. Эта верификационная задача основана на реальном состоянии гонки[2]. Файл с исходным кодом содержит более 7 000 строк кода и 4 создаваемых потока: один поток для базовых функций platform устройства, один – для обработки прерываний, один – для функций power management и один начальный поток, который выполняет операции инициализации-деинициализации модуля. Все примитивы синхронизации ядра (мьютексы и спинлоки) были заменены на pthread мьютексы. Известная ошибка была записана, как задача достижимости, следующим образом:

```
tmp = tspi->rst;
assert(tmp == tspi->rst);
```

Подробные результаты работы инструментов могут быть найдены на официальном сайте SV-COMP[3]. В основном, все современные инструменты верификации столкнулись с проблемами:

- CBMC: «pointer handling for concurrency is unsound – UNKNOWN»;
- CPAchecker: «Unsupported feature: BDD-analysis does not support arrays»;
- SMACK: «Exception thrown in lockpwn»;
- yogar-cbmc: «out of memory»;
- Ultimate: «Ultimate could not prove your program».

Основным вызовом для инструментов верификации в этом примере стало большое количество операций в потоках, большая часть из которых выполнялась над разделяемыми данными. Это означало, что потоки могли сильно влиять друг на друга. Это приводит к следующим подпроблемам, которые обычно игнорируются при анализе небольших «учебных» программ:

1. Анализ многопоточных программ должен быть достаточно точным, чтобы выдавать как можно меньшее количество ложных предупреждений, но быть достаточно эффективным, чтобы решать реальные задачи. Многие эффективные виды анализа не поддерживают сложные структуры данных (например, BDD анализ [19], анализ явных значений [20]). И наоборот, точные подходы вызывают проблемы при анализе длинных путей с переключениями между потоками (например, анализ предикатов [21], подход с ограничиваемой проверкой модели [22]).
2. Эффективное представление примитивов синхронизации. Многие подходы кодируют блокировки как переменные, которые атомарно проверяются и присваиваются (например, [8, 10]). Кодированные таким образом блокировки смешиваются с другими переменными и усложняют общий анализ.

Кроме того, задача поиска состояния гонки в верификационных задачах записывается как задача достижимости, что является подсказкой для инструмента верификации, какой доступ к памяти может содержать ошибку. На практике верификатор не знает точное местоположение потенциального состояния гонки, и поэтому он должен проверять все

[1] https://github.com/sosy-lab/sv-benchmarks.git, sv-benchmarks/c/ldv-linux-3.14-races/linux-3.14–drivers–spi–spi-tegra20-slink.ko.cil.i
[2] https://patchwork.kernel.org/patch/9915305
[3] https://sv-comp.sosy-lab.org/2019/results/results-verified/META ConcurrencySafety.table.html

возможные доступы к памяти. Это еще более усложняет задачу. В данной работе представлен разработанный метод, который позволяет решать такие задачи.

3. Схема предлагаемого метода

Как уже было сказано, в методах поиска состояний гонок условно можно выделить два этапа, точнее, две фазы анализа: построение множества достижимых состояний и проверка требований, в данном случае, поиск парных состояний, образующих гонку. Анализ программы может проводиться путем последовательного чередования этих фаз или путем их параллельного выполнения. Заметим, что проверка требований обычно требует не очень больших затрат, так как представляет собой условие на полученные состояния. Примерами таких проверок могут быть: отсутствие состояний специального вида (задача достижимости), отсутствие пары состояний специального вида (задача поиска гонок). Построение множества достижимых состояний, напротив, порождает ряд проблем, в первую очередь, связанных с эффективностью.

Рассмотрим простую программу, в которой всего два потока (рис. 1).

```
volatile int g = 0;
volatile int d = 0;
Thread1 {
1:  g = 1;
2:  d = 1;
3:  …
}
Thread2 {
4:  if (d == 1) {
5:    g = 2;
6:  }
}
```

Рис. 1. Пример небольшой программы
Fig. 1. An example of a small program

Это некоторый модельный пример, в котором используется конструкция неявной синхронизации между потоками: первый поток инициализирует некоторые данные (в данном случае, глобальную переменную g), а затем выставляет флаг, что данные готовы. Второй поток может использовать эти данные только после выставления флага, поэтому в этом примере нет состояния гонки для переменной g. Классические методы проверки моделей перебирают все возможные варианты чередования двух потоков (пример одного из возможных вариантов приведен на рис. 2).

Thread 1	Thread 2
g = 1;	
d = 1;	
	[d == 1]
	g = 2;

Рис. 2. Пример одного варианта выполнения потоков
Fig. 2. An example of an execution

С точки зрения инструмента статической верификации, необходимо рассмотреть полное множество состояний программы, которые возникают при всех возможных чередованиях (рис. 3).

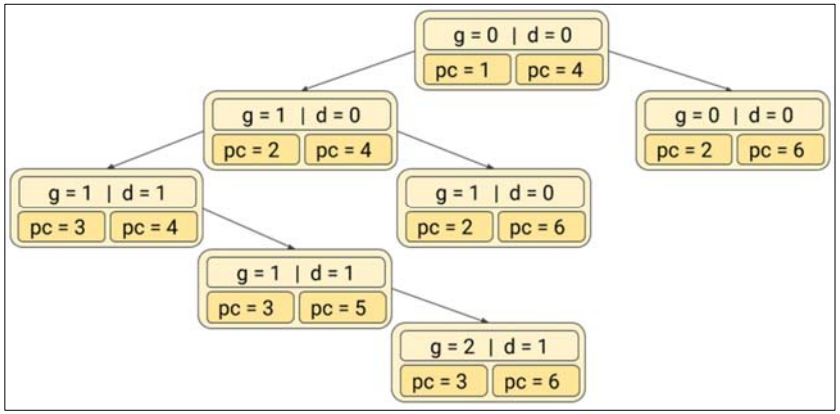

Рис. 3. Построение множества чередований
Fig. 3. Construction of interleaving set

Даже в простом примере и при различных оптимизациях общее число состояний растет с катастрофической скоростью. Происходит так называемый «комбинаторный взрыв» числа состояний, что приводит к исчерпанию ресурсов. Таким образом, классические методы проверки моделей не могут обеспечить доказательства корректности программы.

Простые методы статического анализа пытаются вычислить аппроксимацию сверху возможных действий одного потока на другой, так называемый эффект потока. Однако, они не способны прослеживать сложные зависимости между переменными. Например, зависимости между глобальными переменными, которые, в свою очередь, могут быть модифицированы из других потоков. В общем случае, это требует вычисления некоторой неподвижной точки, что является нежелательным при статическом анализе, так как значительно возрастают требования к ресурсам. В итоге, в таких сложных случаях считается, что глобальные переменные могут принимать любые значения. А это, в свою очередь, снижает точность анализа.

Предлагаемый подход базируется на известной идее раздельного анализа потоков (thread-modular approach). Потоки в этом случае анализируются по-отдельности, одновременно с этим строится общее для всех потоков окружение, которое аппроксимирует сверху влияние других потоков. Это окружение формируется на основе анализа всех потоков, так как каждый поток являются частью окружения для других потоков. Для каждого потока определяется, как и в каких условиях он может модифицировать разделяемые данные, использовать примитивы синхронизации и выполнять иные действия, влияющие на другие потоки. Точность анализа потока зависит от того, как точно будет сформировано окружение. Однако, остается вопрос как эффективно вычислять и представлять окружение.

При анализе последовательных программ успешной техникой, позволяющей уменьшить число рассматриваемых состояний программы, является абстракция. Она позволяет абстрагироваться от несущественных деталей программы и рассматривать обобщенные (абстрактные) состояния, которые могут соответствовать целому множеству реальных (конкретных) состояний программы. Это позволяет значительно сократить пространство состояний.

Ключевой идеей предлагаемого подхода является расширение абстракции не только на состояния программы, но и на операции, то есть, переходы потока. Настраивая уровень абстракции, можно получать варианты анализа, которые будут ближе к статическому анализу многопоточных программ, или к классической статической верификации.

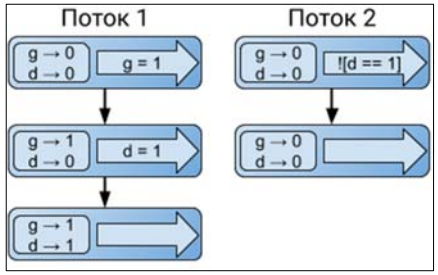

Рис. 4. Построение абстрактных переходов двух потоков
Fig. 4. Construction of abstract states of the first thread

Рис. 4 показывает часть абстрактного графа достижимости (Abstract Reachability Graph, ARG) для первого и второго потока без влияния друг на друга. Представленный анализ основан на простом анализе явных значений [20], который отслеживает только явные значения переменных. Переход содержит в себе абстрактное состояние и абстрактную операцию. Первое абстрактное состояние содержит информацию только о значении глобальной переменной x. Новая информация о значении переменной y появляется в дочерних элементах, после того как выполнен переход, соответствующей инициализации переменной.

Теперь необходимо учесть влияние потоков друг на друга, то есть сформировать окружение. Будем называть *проекцией* операции потока описание ее эффекта, видимого для других потоков. Например, любые модификации локальных переменных потока не влияют на другие потоки, то есть их проекция является пустой операцией. Модификация глобальной переменной является значимой для всех потоков, поэтому ее проекция должна совпадать с самой операцией, либо аппроксимировать ее сверху, например, теряя информацию о точном присваиваемом значении. При этом в проекции может быть не только информация о самом действии, но и об условии на его применение к другому потоку. Например, на рисунке 5 первый поток, присваивая “g = 1” меняет значение переменной g с нуля на единицу. Можно представить проекцию этой операции таким образом: если значение переменной x равно нулю, то оно может быть изменено на единицу. Иными словами, проекция состоит из двух частей: условия ее применения ([g == 0]) и непосредственно действия (g → 1).

При анализе некоторого потока одновременно строится его представление для остальных в качестве окружения. Оно состоит из набора проекций операций этого потока. Далее каждая из этих проекций должна быть применена ко всем возможным (с учетом условий внутри проекций) состояниям других потоков. Что, в свою очередь, может породить новые, еще не исследованные состояния, а значит, и проекции.

После построения переходов в потоках независимо друг от друга (рис. 4), для всех переходов вычисляются проекции. Для второго потока, например, это действие, которое меняет значение переменной x значение с нуля на тройку. Остальные действия второго потока не модифицируют глобальные переменные и не порождают значимых проекций. Затем эта проекция применяется к каждому состоянию первого потока. На рисунке 5 приведен результат применения к первому переходу. Также эту проекцию можно применить и ко второму, однако, никаких новых путей это не породит. К третьему переходу первого потока применить данную проекцию нельзя, так как значение переменной x не удовлетворяет условию проекции. Применение проекции к первому переходу порождает новый путь выполнения, который в свою очередь может породить новые проекции.

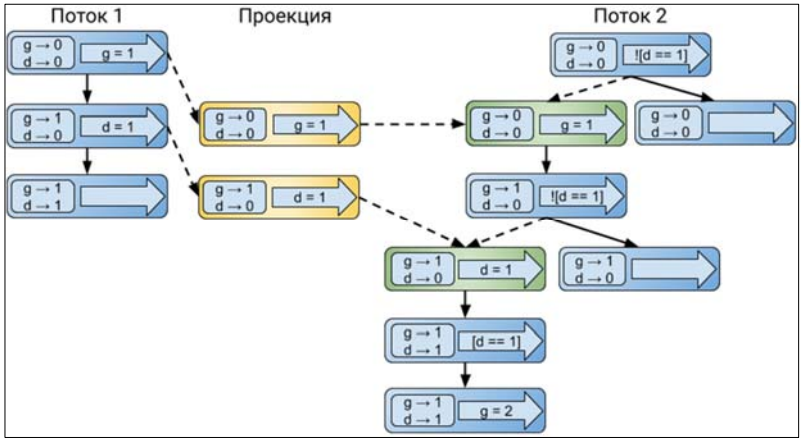

Рис. 5. Построение абстрактных переходов для двух потоков
Fig. 5. Construction of abstract transitions for both of the threads

На рис. 5 представлен вариант с точными проекциями, которые рассматривают переходы другого потока так, как они есть. На рисунке представлены не все возможные проекции и порожденные ими переходы. Например, отсутствует проекция перехода «g = 2» второго потока.

Для проверки возможности состояния гонки нам необходимо найти два перехода, которые модифицируют одну переменную: «g = 1» в первом потоке и «g = 2» во втором. Далее, необходимо проверить, являются ли два абстрактных состояния совместными, то есть, могут ли они быть частью одного глобального состояния. В данном случае, в частичных состояниях потока значения глобальных переменных имеют разные значения, а значит, они не могут быть частью одного глобального состояния, то есть, указанные два перехода не могут быть выполнены одновременно. Отсюда следует, что состояние гонки отсутствует.

Предлагаемый подход предоставляет гибкие варианты конфигурации для решения каждой конкретной задачи. Как было показано на примерах, проекции действий потока могут быть представлены более точной абстракцией или, наоборот, слишком общей. Проекции нескольких операций могут быть объединены в одну или быть рассмотрены по-отдельности. Это позволяет выбирать необходимый баланс между точностью и скоростью. Рассмотрим другой вариант построения абстрактных переходов на рис. 6.

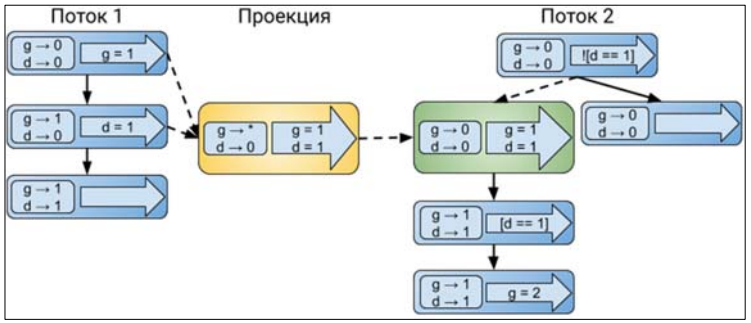

Рис. 6. Построение абстрактных переходов для двух потоков
Fig. 6. Construction of abstract transitions for both of the threads

Здесь используется более абстрактное представление проекции, при котором несколько воздействий потока объединяются в одну проекцию (эффект от окружения). При этом обычно теряется некоторая информация. В частности, в данном случае была потеряна информация о точном значении переменной g, и поэтому данный объединенный эффект может применяться при любых ее значениях. Это позволяет сократить число состояний для анализа.

Пример показывает, как анализ рассматривает эффекты влияния одного потока на другой, что гарантирует корректность подхода. Более того, он показывает гибкость подхода, который позволяет варьировать уровень абстракции, например, на стадии построения проекции, выбирая уровень абстракции каждого перехода в окружении.

4. Основные определения

В этом разделе представлены основные определения параллельной программы и достижимых конкретных состояний программы, необходимые для описания математической модели параллельной (многопоточной) программы и теории CPA.

Математической моделью параллельной программы будет программа на простом императивном языке, в котором имеются только такие операторы, которые оказывают эффект на другие потоки или сами зависят от эффектов, оказываемых другими потоками. Такими операторами служат операторы присваивания, проверки условия (ветвления), примитивы синхронизации, создания потоков. Формальный аппарат описания таких моделей – это теория, для построения которой воспользуемся простым императивным языком, достаточно выразительным, чтобы описывать модели, упомянутые выше. Обозначим множество операций в модели программе Ops.

Параллельная программа представляется автоматом потока управления (Control Flow Automaton, CFA [5]), который состоит из множества L точек программы (моделируются программным счетчиком, pc) и множеством G ⊆ L×Ops×L дуг (ребер) потока управления (моделируют операции, которые выполняются, когда управление переходит от одной точки в программе к другой). Операция создания потока создает новый поток с идентификатором из множества T и этот поток начинает свое выполнение из некоторой точки программы из L. Множество переменных программы, которые встречаются в операторах присваивания и условиях из Ops обозначаются X, а их значения ограничим множеством целых чисел $\mathbb{Z}$. Подмножества X, содержащие только локальные и глобальные переменные, обозначаются X^{local} и X^{global} соответственно. Операции захвата/освобождения примитивов синхронизации определяются на множестве переменных-блокировок S, которые имеют значения из $T \cup \{\perp_T\}$, где $t \in T$ означает, что соответствующая блокировка была захвачена потоком t, а $\perp_T$ означает, что соответствующая блокировка не была захвачена.

Конкретным состоянием программы называется четверка (c_{pc} , c_l , c_g , c_s), где

1. отображение c_{pc}: $T \to L$ является частичной функцией из идентификаторов потока в точку программы, в которой находится этот поток;
2. отображение c_l: $T \to C^{local}$ является частичной функцией из идентификаторов потока в присваивание локальным переменным их значений, то есть $C^{local} : X^{local} \to \mathbb{Z}$;
3. отображение c_g: $X^{global} \to \mathbb{Z}$ является присваиванием значений глобальным переменным;
4. отображение c_s: $S \to T \cup \{\perp_T\}$ является присваиванием значений переменным блокировки.

Множество конкретных состояний программы обозначается как C. Отображения c_{pc} и c_l представляют собой локальную часть состояния потока, а c_g и c_s – глобальную. Обозначим dom – домен частичной функции, например, $dom(c_l) = \{t | \exists (t, \cdot) \in c_l\}$.

Для каждого состояния $c=(c_{pc},c_l,c_g,c_s) \in C$ должно выполняться условие $dom(c_{pc})=dom(c_l)$, означающее, что домены локальных частей состояния должны быть консистентны и содержать одинаковое число потоков. Это множество обозначим $dom(c)=dom(c_{pc})=dom(c_l)$.

Определим отношение переходов $\overset{g,t}{\to} \subseteq C \times G \times T \times C$, где дуга $g \in G$, а поток $t \in T$.

Определим множество конкретных переходов $\mathcal{T} = C \times G \times T$. Элемент $\tau \in \mathcal{T}$ – это тройка $\tau = (c, g, t)$. Будем писать $\tau_1 \to \tau_2$ если $\exists c_3 \in C$: $c_1 \xrightarrow{g_1,t_1} c_2 \xrightarrow{g_2,t_2} c_3$. Семантику операций определим позже. В любом случае корректный переход $g = (l,\cdot,l') \in G$ должен удовлетворять следующим условиям:

1. переход начинается в состоянии c: $t \in dom(c) \wedge c_{pc}(t) = l$.
2. программный счетчик потока t переходит к точке l': $t \in dom(c') \wedge c'_{pc}(t) = l'$.
3. каждый переход в потоке t может изменить локальные части только этого же потока, а локальные состояния других потоков должны остаться без изменений; заметим, что для программ с указателями это не верно, но пока мы не рассматриваем такие программы в теории.

Будем обозначать $Reach_{\to}(\tau) = \{\tau | \exists \tau_1, \ldots, \tau_n \in \mathcal{T}, .\tau \to \tau_1 \to \ldots \to \tau_n = \tau_0\}$. Будем обозначать *eval(c,t,expr)* значение выражения *expr* над переменными из $X^{local} \cup X^{global}$ со значениями из состояния $c \in C$ потока $t \in T$.

4.1 Операторы условия

Для дуги проверки условия $g = (l, assume(expr), l') \in G, t \in T, l, l' \in L$ переход $c \overset{g,t}{\to} c', c = (c_{pc}, c_l, c_g, c_s), c' = (c'_{pc}, c'_l, c'_g, c'_s) \in C$ существует, если

- $dom(c) = dom(c')$, то есть переход не меняет множества потоков;
- $c_l = c'_l, c_g = c'_g, c_s = c'_s$, то есть переход не меняет значений переменных;
- $c_{pc}(t) = l, c'_{pc}(t) = l'$, то есть переход соответствует общим условиям на начало и конец;
- $eval(c, t, expr) \neq 0$, то есть значения переменных потока удовлетворяют проверяемому условию.

4.2 Операторы присваивания

Для дуги присваивания $g = (l, assign(y, expr), l') \in G, t \in T, l, l' \in L$ переход $c \overset{g,t}{\to} c', c = (c_{pc}, c_l, c_g, c_s), c' = (c'_{pc}, c'_l, c'_g, c'_s) \in C$ существует, если:

- $dom(c) = dom(c')$, то есть переход не меняет множества потоков;
- $\forall x \in X^{local}, t' \in T. c'_l(t')(x) = \begin{cases} c_l(t')(x), \text{если } x \neq y \vee t \neq t' \\ eval(c, t, expr), \text{если } x = y \wedge t = t' \end{cases}$;
- $\forall x \in X^{global}. c'_g(x) = \begin{cases} c_g(x), \text{если } x \neq y \\ eval(c, t, expr), \text{если } x = y \end{cases}$;
- $c_s = c'_s$, множество блокировок не меняется при обычном присваивании.
- $c_{pc}(t) = l, c'_{pc}(t) = l'$, то есть переход соответствует общим условиям на начало и конец.

4.3 Операции над примитивами синхронизации

Определим операции над примитивами синхронизации *acquire/release*. Предполагаем, что операция *acquire(s)* в потоке $t \in T$, где $s \in S$ – это специальная переменная блокировки, имеет стандартную семантику: атомарная проверка значения переменной s и, в случае

если s=$\perp_T$, присвоение ей идентификатора текущего потока. Например, такая же семантика рассматривается в [8].

Для дуги захвата блокировки $g = (l, acquire(s), l') \in G, t \in T, l, l' \in L$ переход $c \xrightarrow{g,t} c', c = (c_{pc}, c_l, c_g, c_s), c' = (c'_{pc}, c'_l, c'_g, c'_s) \in C$ существует, если:

- $dom(c) = dom(c')$, то есть переход не меняет множества потоков;
- $c_l = c'_l, c_g = c'_g$, то есть переход не меняет значений переменных;
- $c_{pc}(t) = l, c'_{pc}(t) = l'$, то есть переход соответствует общим условиям на начало и конец;
- $c_s(s) = \perp_T \wedge c'_s(s) = t$, $\forall s' \in S: s' \neq s \Rightarrow c'_s(s') = c_s(s')$.

Операция освобождения блокировки *release(s)* является обратной к операции захвата блокировки и присваивает значение s = $\perp_T$, если эта блокировка была захвачена текущим потоком, то есть s = t.

Для дуги освобождения блокировки $g = (l, release(s), l') \in G, t \in T, l, l' \in L$ переход $c \xrightarrow{g,t} c', c = (c_{pc}, c_l, c_g, c_s), c' = (c'_{pc}, c'_l, c'_g, c'_s) \in C$ существует, если:

- $dom(c) = dom(c')$, то есть переход не меняет множества потоков;
- $c_l = c'_l, c_g = c'_g$, то есть переход не меняет значений переменных;
- $c_{pc}(t) = l, c'_{pc}(t) = l'$, то есть переход соответствует общим условиям на начало и конец;
- $c_s(s) = t \wedge c'_s(s) = \perp_T, \forall s' \in S: s' \neq s \Rightarrow c'_s(s') = c_s(s')$.

4.4 Операция создания потока

Определим семантику операции *thread_create(l_v)* таким образом, что текущий поток переходит в следующую точку программы, а новый поток создается с идентификатором v ∈ T и начинает свое выполнение из l_v ∈ L.

В общем случае возможно неограниченное создание потоков в программе, если, например, операция создания потока встречается в цикле.

Для дуги создания потока $g = (l, thread\_create(l_v), l') \in G, t \in T, l, l' \in L$ переход $c \xrightarrow{g,t} c', c = (c_{pc}, c_l, c_g, c_s), c' = (c'_{pc}, c'_l, c'_g, c'_s) \in C$ существует, если:

- $v \notin dom(c) \wedge dom(c') = dom(c) \cup \{v\}$, поток v добавляется во множество потоков;
- $c_l = c'_l, c_g = c'_g, c_s = c'_s$, то есть переход не меняет значений переменных текущего потока;
- $c_{pc}(t) = l, c'_{pc}(t) = l'$, то есть переход соответствует общим условиям на начало и конец;
- $c'_{pc}(v) = l_v$, то есть новый поток начинает свое выполнение из своего начального состояния

Пока мы не рассматриваем операции ожидания потока(*thread_join (*v)), так как они будут усложнять объяснение и доказательство основной теоремы корректности предложенного подхода. Тем не менее, их поддержка также может быть добавлена.

5. Адаптивный статический анализ с абстрактными переходами

В классической теории адаптивного статического анализа (Configurable Program Analysis, CPA) [5, 6], абстрактное состояние представляет множество конкретных состояний программы. В расширенной теории, абстрактное состояние является частичным и может не представлять никакое состояние программы. Вот почему функция конкретизации, которая сопоставляет абстрактные состояния с конкретными, в расширенной теории

отличается от классической. В частности, она определяется на множестве абстрактных состояний.

Как следствие, абстрактный переход также является частичным. Поэтому анализ не может гарантировать, что последующие конкретные переходы будут достижимы за один шаг оператора transfer. В общем случае для этого может понадобиться k шагов. Для подхода с раздельным анализом потоков k = 2: анализ выполняет обычный переход в потоке, а затем распространяет его на все остальные потоки в качестве перехода в окружении. Это требует двух итераций алгоритма.

Определим формально адаптивный статический анализ с абстрактными переходами $\mathbb{D}$ = (*D, Π, merge, stop, prec,* $\rightsquigarrow$). Он состоит из абстрактного домена *D*, множество точности *Π*, оператор слияния *merge*, оператор останова *stop*, оператор настройки точности *prec*, и отношение переходов $\rightsquigarrow$.

1. Абстрактный домен D = ($\mathcal{T}$, $\mathcal{E}$, [[·]]) определяется множеством $\mathcal{T}$конкретных переходов, $\mathcal{T} \subseteq$ C × G × T, полурешеткой $\mathcal{E}$ над абстрактными переходами и функцией конкретизации [[.]]. Полурешетка $\mathcal{E}$ = (E, $\perp$, $\top$, $\sqsubseteq$, $\sqcup$) состоит из (возможно бесконечного) множества E абстрактных элементов, верхнего элемента решетки $\top \in$E, нижнего элемента решетки $\perp \in$E, частичный порядок $\sqsubseteq \subseteq$ E × E и функцию объединения $\sqcup$: E × E → E. Функция объединения определяет минимальный элемент решетки, который больше заданных элементов.

Функция конкретизации [[·]] : $2^E \rightarrow 2^{\mathcal{T}}$ для каждого множества абстрактных переходов R ⊆ E определяет множество соответствующих конкретных переходов программы. Основным отличием от классической функции конкретизации – это определение на множестве абстрактных элементов. Таким образом, $\forall R \subseteq E : [[R]] \supseteq \bigcup_{e \in R} [[\{e\}]]$. Это означает, что суммарное знание от множества абстрактных переходов может быть больше, чем объединение знания от каждого частичного перехода.

2. Множество точности Π определяет возможную точность абстрактного домена. Анализ использует элементы точности, чтобы отслеживать различные абстрактные состояния с различной точностью. Пара (e, π) называется абстрактным элементом e с точностью π. Операторы на абстрактном домене параметризованы точностью.

Для R⊆E×Π будем обозначать $[[R]] = [[\bigcup_{(e,\pi) \in R} \{e\}]]$.

3. Отношение переходов $\rightsquigarrow$: E × Π × 2^E × E определяет для каждого частичного перехода ê с точностью π возможный следующий переход e'. При этом результат может зависеть от множества достижимых элементов R ⊆ E. Будем обозначать (ê, π) $\rightsquigarrow^R$e if (ê, π, R, e) ∈$\rightsquigarrow$.

Определим множество $Reach^k$ по индукции: $\forall R \subseteq E : Reach^0(R) = R$

$$\forall k \geq 1 : Reach^{k+1}(R) = \bigcup_{e \in Reach^k(R)} \{e' \mid e \rightsquigarrow e'\} \cup Reach^k(R)$$

Основное требование к оператору *transfer* является аппроксимация сверху множества конкретных переходов:

$$\exists k \geq 1 : \forall R \subseteq E : Reach^k(R) = \bigcup_{\tau \in [[R]]} \{\tau' \mid \tau \rightarrow \tau'\}$$

Это требование ослабляет требование на *transfer* классической теории. Оно означает, что оператор должен выдавать соответствующие конкретные переходы на после одного шага, как в классической версии, а после k шагов. Для подхода с раздельным анализом потоков k = 2, как мы увидим в дальнейшем.

4. Оператор слияния *merge*: E×E×Π→E ослабляет второй параметр, используя информацию от первого параметра и возвращает новый абстрактный элемент с точностью, соответствующей третьему параметру. Оператор *merge* должен удовлетворять следующему условию:

$$\forall e, e' \in E, \pi \in \Pi: e' \sqsubseteq merge(e, e', \pi)$$

5. Оператор останова *stop*: E×2^E×Π×Π→{true, false} проверяет, покрывается ли абстрактный элемент, данный в качестве первого параметра, множеством элементов, данных как второй параметр. Оператор останова может, например, искать среди множества элементов такой, который покрывает (⊑) данный элемент. Оператор останова должен удовлетворять следующим условиям:

$$\forall e, e' \in E, \pi \in \Pi: stop(e, R, \pi) \Rightarrow \forall \hat{R} \subseteq E: [[\{e\} \cup \hat{R}]] \subseteq [[R \cup \hat{R}]]$$

6. Функция настройки точности *prec*:E×Π×2^E×Π→E×Π вычисляет новый абстрактный элемент и новую точность для заданного элемента с точностью и множества абстрактных элементов. Функция настройки точности может выполнять ослабление (расширение) абстрактного элемента вместе с изменением точности. Функция настройки точности должна удовлетворять следующему требованию:

$$\forall e, e' \in E, \pi, \pi' \in \Pi, R \subseteq E \times \Pi: (e', \pi') = prec(e, \pi, R) \Rightarrow e \sqsubseteq e'$$

В целом, множество точности Π, оператор останова stop, оператор объединения merge, оператор настройки точности prec остаются такими же, как и в классической теории CPA. На рис. 7 представлен основной алгоритм, который вычисляет множество достижимых абстрактных переходов, также не изменяется за исключением расширения оператора *transfer*.

```
waitlist := {(e0, π0)};
reached := {(e0, π0)};
while waitlist ≠ ∅ do
    pop (e, π) from waitlist;
    for e' in (e, π) ⇝(reached) e') do
        (ê, π̂) = prec(e', π, reached);
        for each (e'', π'') ∈ reached do
            e_new = merge(ê, e'', π̂);
            if e_new ≠ e'' then
                waitlist := waitlist \ {(e'', π'')} ∪ {(e_new, π'')};
                reached := reached \ {(e'', π'')} ∪ {(e_new, π'')};
            end
        end
        if !stop(ê, reached, π̂) then
            waitlist := waitlist ∪ {(ê, π̂)};
            reached := reached ∪ {(ê, π̂)};
        end
    end
end
```

Рис. 7. Основной алгоритм CPA(D, e_0 , π_0)
Fig. 7. The main algorithm CPA(D, e_0 , π_0)

Для этого алгоритма основная теорема может быть доказана даже с ослабленными требованиями. Доказательство повторяет доказательство классической теоремы.

Теорема (корректность). Для заданного адаптивного статического анализа с абстрактными переходами $\mathbb{D}$, начального состояния e_0 с точностью π_0 алгоритм CPA(D, e_0 , π_0) вычисляет множество абстрактных переходов, которое аппроксимирует сверху множество достижимых конкретных переходов:

$$[[CPA(\mathbb{D}, e_0, \pi_0)]] \supseteq Reach_{\rightarrow}([[\{e_0\}]])$$

Следует отметить, что на практике используется одновременно несколько различных CPA для анализа исходного кода. При этом структура множества CPA напоминает древовидную, где корень этого дерева предоставляет операторы для основного алгоритма, представленного на рис. 7. Различные CPA взаимодействовать друг с другом для повышения точности анализа. Так, в разд. 6 будет представлен верхнеуровневый CPA,

который требует наличие вложенного CPA. Примеры вложенных CPA будут представлены в разделах 7 – 11. CPA, представленный в разд. 12, реализует параллельную композицию нескольких вложенных анализов. Разд. 13 описывает, как реализуется поиск состояний гонки при помощи этих инструментов.

Некоторые служебные CPA (CallstackCPA, AutomatonCPA), не реализующие никакие техники анализа, могут быть применены без изменений по сравнению с классической версией теории, и поэтому не будут описываться далее. CPA, которые реализуют различные техники анализа (анализ явных значений, анализ предикатов и др.) необходимо доработать для того, чтобы они могли поддерживать переходы потоков в окружении, в том числе реализовать оператор проекции. Эти CPA будут описаны в соответствующих разделах.

6. ThreadModularCPA

В этом разделе представлен CPA, который реализует логику подхода с раздельным анализом потоков. Основная функциональность ThreadModularCPA заключается в вычислении для каждого перехода в потоке потенциального эффекта для окружения, то есть проекцию этого перехода, а также в применении полученных эффектов окружения на соответствующий поток.

ThreadModularCPA включает в себя некоторый внутренний анализ с расширенным множеством операторов, которые необходимы для работы с окружением: оператор проекции, оператор совместности, и оператор композиции. Сначала формально определим расширение внутреннего CPA.

6.1 Расширение внутреннего CPA

Определение CPA для подхода с раздельным анализом потоков расширяется тремя новыми операторами: $\mathbb{I} = (D_I, \Pi_I, I, merge_I, stop_I, prec_I, compatible_I, \cdot|_p, compose_I)$.

Абстрактный домен $D_I = (\mathcal{T}_I, \mathcal{E}_I, \oplus_I)$ состоит из множества конкретных переходов $\mathcal{T}_I$, полурешетки $\mathcal{E}_I$, и оператора композиции частичных состояний $\oplus_I$. Таким образом, внутренний анализ должен определить не функцию конкретизации $[[\cdot]]$, а оператор композиции $\oplus$, так как подход с раздельным анализом потоков требует одинаковой схемы вычисления конкретных состояний.

Как было уже сказано, состояния и переходы являются частичными, поэтому они могут не соответствовать напрямую конкретным состояниям и переходам. Чтобы получить полный переход, нужно взять композицию множества частичных переходов, которые соответствуют всем доступным потокам. Совместные частичные переходы могут быть объединены в полный конкретный переход с помощью оператора композиции $\oplus: E \times T \times 2^{E \times T} \rightarrow 2^{\mathcal{T}}$. Он возвращает множество конкретных переходов, которое соответствует данным частичным переходам.

$\oplus$ должен соответствовать полурешетке. Так, если один из абстрактных переходов меньше, чем другой, то композиция с тем же множеством не должна получить большее множество конкретных переходов.

Оператор проверки совместности $compatible_I$: E × E → {true, false} проверяет, могут ли два частичных перехода начинаться из общего полного родительского состояния.

Оператор проекции $\cdot|_p$: E → E проецирует переход в потоке на другой поток. Например, проекция может содержать модификации глобальных переменных, но опускать изменения локальных данных для потока.

$compose_I$: E × E → E объединяет два абстрактных перехода в один. Он применяет абстрактную дугу из одного перехода к абстрактному состоянию другого перехода.

В дальнейшем мы будем использовать оператор apply_I, как комбинацию трех операторов: $\cdot|_\text{p}$, compose_I и $\text{compatible}_\text{I}$:

$$\forall \boldsymbol{e}, \boldsymbol{e'} \in \boldsymbol{E}\colon \boldsymbol{apply}(\boldsymbol{e}, \boldsymbol{e'}) = \begin{cases} \boldsymbol{compose_I}\left(\boldsymbol{e}, \boldsymbol{e'}|_\boldsymbol{p}\right), \text{если } \boldsymbol{compatible_I}\left(\boldsymbol{e}, \boldsymbol{e'}|_\boldsymbol{p}\right) \\ \bot, \text{иначе} \end{cases}$$

Таким образом, оператор apply_I означает, что переходы могут быть объединены, только если они совместны. Результатом применения оператора является новый переход, который будем называть переходом в окружении, так как он представляет собой эффект окружения.

6.2 CPA для раздельного анализа потоков

Определим специальный CPA, который реализует логику раздельного анализа потоков: $\mathbb{TM} = (\text{D}_\text{TM}, \Pi_\text{TM}, \rightsquigarrow_\text{TM}, \text{merge}_\text{TM}, \text{stop}_\text{TM}, \text{prec}_\text{TM})$, который основан на внутреннем CPA $\mathbb{I} = (\text{D}_\text{I}, \Pi_\text{I}, \text{I}, \text{merge}_\text{I}, \text{stop}_\text{I}, \text{prec}_\text{I}, \text{compatible}_\text{I}, \cdot|_\text{p}, \text{compose}_\text{I})$.

Абстрактный домен $\text{D}_\text{TM} = (\mathcal{T}, \mathcal{E}, [[\cdot]])$, множество конкретных переходов $\mathcal{T} = \mathcal{T}_\text{I}$, а решетка $\mathcal{E} = \mathcal{E}_\text{I}$. Функция конкретизации $[[\cdot]]$ выражается через оператор композиции $\oplus_\text{I}$:

$$\forall R \subseteq E : [[R]] = \bigcup_{k=1}^{\infty} \bigcup_{\substack{e_0, e_1, \dots, e_k \in R \\ t_0, t_1, \dots, t_k \in T}} \oplus_I \left(\begin{matrix} e_0 \\ t_0 \end{matrix}, \begin{Bmatrix} e_1 \\ t_1 \end{Bmatrix}, \dots, \begin{matrix} e_k \\ t_k \end{matrix}\right)$$

Отношение переходов определяет следующие переходы, после чего применяются все достигнутые переходы, как переходы в окружении, к новым переходам, а новые переходы, как переходы в окружении, – к уже достижимым (рис. 8).

```
result := ∅ ;
for each ê : e0 ⇝_I^R ê do
    result := result ∪ {ê} ;
    for each e' ∈ reached do
        result := result ∪ {apply(e', ê)} ;
        result := result ∪ {apply(ê, e')} ;
    end
end
return result
```

Puc. 8. $\text{transfer}_\text{TM}(e_0, \pi_0, \text{reached})$
Fig. 8. $\text{transfer}_\text{TM}(e_0, \pi_0, \text{reached})$

Множество точности Π, операторы *merge*, *stop*, *prec* соответствуют операторам внутреннего CPA.

7. LocationCPA

В этом разделе представлен простой анализ точек программы (Location Analysis) $\mathbb{L} = (\text{D}_\text{L}, \Pi_\text{L}, \rightsquigarrow_\text{L}, \text{merge}_\text{L}, \text{stop}_\text{L}, \text{prec}_\text{L}, \text{compatible}_\text{L}, \cdot|_\text{p}, \text{compose}_\text{L})$, который отслеживает абстрактные точки программы. Анализ расширяет классический LocationCPA для возможности применения его вместе с ThreadModularCPA.

1. Абстрактный домен $\text{D}_\text{L} = (\mathcal{T}_\text{L}, \mathcal{E}_\text{L}, \oplus_\text{L})$. Абстрактный переход состоит из абстрактного состояния $\text{s} \in E_\text{L}^S$, и абстрактной дуги $\text{q} \in E_\text{L}^\text{T}$. E_L^S – это множество абстрактных точек программы (англ. program location), которые отображаются на конкретные точки программы в CFA с помощью функции $loc\colon E_\text{L}^S \to 2^L$. $\top_L^S$ означает, что анализ не знает, в какой именно точке программы находится анализ. Более формально, $loc\left(\top_L^S\right) = L$. В общем случае анализ может использовать абстрактными точками программы, которые соответствуют нескольким конкретным точкам программы, но в дальнейшем будет

описан простой вариант анализа, который рассматривает только одиночные точки программы:

$$\forall \text{s} \in E_\text{L}^S\colon \text{s} = \top_L^S \vee \text{s} = \bot_\text{L}^\text{S} \vee loc(\text{s}) = \{\text{l}\} \in \text{L}$$

В этом случае используемая решетка $\mathcal{E}_\text{L}^S$ является плоской, то есть любые два невырожденных состояния (неравные $\top_L^S$ или $\bot_\text{L}^\text{S}$) несравнимы.

Абстрактная дуга основана на обычной CFA дуге и содержит только начальную точку программы и конечную: $E_\text{L}^T \subseteq E_\text{L}^S \times E_\text{L}^S$.

2. Множество точности является вырожденным и содержит только один элемент: $\Pi_S = \{\emptyset\}$.

3. Переход $e \rightsquigarrow_\text{L} e', e = (s, g), e' = (s', g'), q = (pred, suc), q' = (pred', suc')$ существует, если изменение точки программы соответствует абстрактной дуге. Более формально, $e \rightsquigarrow_P e' \Leftrightarrow loc(s) \cap loc(pred) \neq \emptyset, \exists g' \in G\colon g' = (l'_1, op, l'_2) \wedge l'_1 \in loc(pred') \cap loc(s') \wedge l'_2 \in loc(suc') \wedge s' = suc$.

4. Оператор слияния *merge* не объединяет элементы: $merge_\text{L}(e, e', \pi) = e'$.

5. Оператор останова *stop* рассматривает уникальные состояния: $stop_\text{L}(e, R, \pi) = (e \in R)$.

6. Точность не регулируется: $prec_\text{L}(e, \pi, R) = (e, \pi)$.

7. Переход в одном потоке никак не влияет на переход в другом: $\forall e \in E_\text{L}, e = (s, g)\colon e|_P = (s, \varepsilon)$.

8. $\forall e, e' \in E_\text{L}, e = (s, q)\colon compose_\text{L}(e, e') = \tilde{\text{e}} = (s, \hat{q})$, где $\hat{q} = (s, s)$, так как переход в одном потоке никак не может повлиять на положение другого потока.

9. $\forall e_1, e_2 \in E_\text{L}\colon compatible_\text{L}(e_1, e_2) \equiv true$, так как два потока могут быть совместны в любых точках программы.

8. ThreadCPA

Определим простой анализ потоков $\mathbb{T} = (\text{D}_\text{T}, \Pi_\text{T}, \rightsquigarrow_\text{T}, \text{merge}_\text{T}, \text{stop}_\text{T}, \text{prec}_\text{T}, \text{compatible}_\text{T}, \cdot|_\text{p}, \text{compose}_\text{T})$, который отслеживает идентификаторы потока.

Анализ потоков наследует ограничения из [7] и ограничен программами с конечным числом созданий потоков. Пусть в программе есть конечное число потоков, которые идентифицируются точками программы, в которых они начинают свое выполнение, то есть $T \subseteq L$ и для каждого оператора создания потока *thread_create*(pc_v) всегда создается поток с идентификатором pc_v. Заметим, что остальные типы анализов не ограничены числом созданий потоков, более того, возможно применение более сложного анализа потоков, который будет поддерживать неограниченное число созданий. Таким образом, общая теория поддерживает неограниченное число созданий потоков.

1. Абстрактный домен $\text{D}_\text{T} = (\mathcal{T}_\text{T}, \mathcal{E}_\text{T}, \oplus_\text{T})$ основан на плоской решетке над множеством потоков T, где $\mathcal{E}_\text{T} = \mathcal{E}_\text{T}^S \times \mathcal{E}_\text{T}^\text{T}$. Множество абстрактных состояний $E_T^S = T \cup \{\bot_T^\text{S}, \top_T^S\}$, в котором любые два невырожденных состояния (неравные $\top_T^S$ или $\bot_T^\text{S}$) несравнимы. Множество абстрактных дуг содержит множество обычных CFA дуг и пустой переход в окружении: $E_\text{T}^\text{T} = \{\bot_\text{T}^\text{T}, \varepsilon, \top_\text{T}^\text{T}\} \cup G$.

2. Множество точности является вырожденным и содержит только один элемент: $\Pi_\text{T} = \{\emptyset\}$.

3. Переход $e \rightsquigarrow_\text{T} e', e = (s, g), e' = (s', g'), \text{g} = (\cdot, \text{op}, \cdot)$ существует, если

- $\text{op} \neq$ *thread_create*(pc_v), $s' = s, g' \in \text{G}$, то есть при любой операции, кроме создания потока, состояние не меняется;
- $\text{op} =$ *thread_create*(pc_v), $s' = pc_v \vee s' = s, g' \in \text{G}$. При создании потока создаются два дочерних состояния: одно соответствует созданному потоку, а другое – родительскому.

4. Оператор слияния *merge* не объединяет элементы: $merge_{\mathrm{T}}(e, e', \pi) = e'$.

5. Оператор останова *stop* рассматривает уникальные состояния: $stop_{\mathrm{T}}(e, R, \pi) = (e \in R)$.

6. Точность не регулируется: $prec_{\mathrm{T}}(e, \pi, R) = (e, \pi)$.

7. Переход в одном потоке никак не влияет на переход в другом: $\forall e \in E_{\mathrm{T}}, e = (s, g)\colon e|_P = (s, \varepsilon)$. Переходы в окружении ($\varepsilon$-переходы) не могут изменить идентификатор другого потока. Тем не менее, проекция влияет на совместность состояний, чтобы переходы в потоке не применялись, как эффекты окружения, к этому же самому потоку.

8. $\forall e, e' \in E_{\mathrm{T}}, e = (s, q), e' = (s', q')\colon compose_{\mathrm{T}}(e, e') = \tilde{e} = (s, q')$.

9. $\forall e_1, e_2 \in E_{\mathrm{T}}, e_1 = (s_1, q_1), e_2 = (s_2, q_2)\colon compatible_{\mathrm{T}}(e_1, e_2) = s_1 \neq s_2$, так как два состояния могут быть совместны, если они относятся к разным потокам.

9. ValueCPA

Определим анализ явных значений $\mathbb{V}$ = ($\mathrm{D_V}$, Π_{V}, $\rightsquigarrow_{\mathrm{V}}$, $\mathrm{merge_V}$, $\mathrm{stop_V}$, $\mathrm{prec_V}$, $\mathrm{compatible_V}$, $\cdot|_{\mathrm{p}}$, $\mathrm{compose_V}$), который отслеживает явные значения переменных. Он состоит из следующих элементов.

1. Абстрактный домен $\mathrm{D_P} = (\mathcal{T}_{\mathrm{V}}, \mathcal{E}_{\mathrm{V}}, \bigoplus_{\mathrm{V}})$. $\mathcal{E}_{\mathrm{V}} = (\mathrm{E_V}, \bot_{\mathrm{V}}, \top_{\mathrm{V}}, \sqsubseteq_{\mathrm{V}}, \bigsqcup_{\mathrm{V}})$. Абстрактный переход состоит из абстрактного состояния $s \in E_V^S$, а абстрактная дуга $q \in E_V^{\mathrm{T}}$, таким образом, $\mathrm{E_V} = E_{\mathrm{V}}^S \times E_{\mathrm{V}}^{\mathrm{T}}$, а $\mathcal{E}_{\mathrm{V}} = \mathcal{E}_{\mathrm{V}}^S \times \mathcal{E}_{\mathrm{V}}^{\mathrm{T}}$.

Абстрактное состояние этого анализа является отображением из имен переменных в их значение: $\forall s \in E_V^S, s\colon X \mapsto \mathcal{Z}$, где $\mathcal{Z} = \mathbb{Z} \cup \{\bot_{\mathcal{Z}}, \top_{\mathcal{Z}}\}$. Таким образом, множество абстрактных состояний является плоской решеткой над целыми числами. Верхний элемент решетки $\top_{\mathrm{V}}^S = \{v \mid \forall x \in X\colon v(x) = \top_{\mathcal{Z}}\}$. является отображением, в котором любая переменная имеет любое значение. А нижний элемент решетки $\bot_{\mathrm{V}}^S = \{v \mid \exists x \in X\colon v(x) = \bot_{\mathcal{Z}}\}$ является отображением, в котором никакая переменная не может иметь никакого явного значение. Такое состояние является недостижимым при реальном выполнении программы. Порядок является тривиальным: любые два невырожденных состояния (неравные $\top_{\mathrm{V}}^S$ или $\bot_{\mathrm{V}}^S$) несравнимы.

Множество абстрактных дуг содержит множество обычных CFA дуг и переходы в окружении, которые определяются изменением глобальных переменных: $E_{\mathrm{V}}^{\mathrm{T}} = 2^{X \mapsto \mathcal{Z}} \cup G$.

2. Точность анализа явных значений определяется отслеживаемыми переменными, таким образом множество точности содержит подмножества из всех переменных программы: $\Pi_{\mathrm{V}} = 2^X$.

3. Отношение перехода $e \rightsquigarrow_{\mathrm{V}} e', e = (s, g), e' = (s', g')$.

I. $g \in G, g = (\cdot, op, \cdot)$.

a. $g = (\cdot, assume(expr), \cdot)$:

$$\forall x \in X\colon s'(x) == \begin{cases} \bot_{\mathcal{Z}}, \text{если } \nexists c.\,(x \to c)\colon (\mathrm{expr} \neq 0)_{/s} \\ c, \text{если } \exists!\, c.\,(x \to c)\colon (\mathrm{expr} \neq 0)_{/s} \text{ или } s(x) = c \\ \top_{\mathcal{Z}}, \text{иначе} \end{cases}$$

Здесь $\mathrm{expr}_{/v}$ означает интерпретацию выражения expr над переменными из X для абстрактного присваивания v. А выражение $(x \to c)\colon (\mathrm{expr} \neq 0)_{/s}$ означает, что значение c у переменной x удовлетворяет интерпретации.

b. $g = (\cdot, assign(w, expr), \cdot)$:

$$\forall x \in X\colon s'(x) = \begin{cases} \mathrm{expr}_{/s}, \text{если x} = \mathrm{w} \\ s(x), \text{иначе} \end{cases}$$

c. В остальных случаях состояние не меняется $s = s'$.

II. $g\colon X \mapsto \mathcal{Z}$, это означает, что мы имеем переход, который меняет определенные переменные. В этом случае, следующее состояние

$$\forall x \in X\colon s'(x) = \begin{cases} g(x), \text{если x} \in \mathrm{dom}(g) \\ s(x), \text{иначе} \end{cases}$$

4. Оператор слияния *merge* не объединяет элементы: $merge_{\mathrm{V}}(e, e', \pi) = e'$.

5. Оператор останова *stop* рассматривает уникальные состояния: $stop_{\mathrm{V}}(e, R, \pi) = (e \in R)$.

6. Функция настройки точности вычисляет новое абстрактное состояние и точность, ограничивая присваивания только теми переменными, которые содержатся в точности: $prec_{\mathrm{V}}(e, \pi, R) = (e_{|\pi}, \pi)$.

7. Переход в окружении может затрагивать только глобальные переменные: $\forall e \in E_{\mathrm{V}}, e = (s, g)\colon e|_P = (s^{global}, g^{global})$. Здесь отображение s^{global} означает только ту часть, которая относится к глобальным переменным.

8. $\forall e, e' \in E_{\mathrm{V}}, e = (s, q), e' = (s', q')\colon compose_{\mathrm{V}}(e, e') = \tilde{e} = (s, q')$.

9. $\forall e_1, e_2 \in E_{\mathrm{V}}, e_1 = (s_1, q_1), e_2 = (s_2, q_2)\colon compatible_{\mathrm{V}}(e_1, e_2) = \forall x \in X^{global}\colon (x \in dom(s_1) \wedge x \in dom(s_2)) \Rightarrow (s_1(x) \sqsubseteq s_2(x) \vee s_2(x) \sqsubseteq s_1(x))$, что означает консистентность значений глобальных переменных.

10. PredicateCPA

В этом разделе описан известный анализ предикатов (Predicate Analysis) [21] с абстрактными переходами. Переходы анализа предикатов состоят из двух частей: абстрактного состояния и абстрактной дуги, которая может быть выражена либо обычной CFA дугой, либо логической формулой, кодирующей выполняемую операцию. Кроме того, локальные переменные в этих формулах должны быть переименованы для избежания коллизии имен для разных потоков.

Пусть $\mathcal{P}$– это множество формул над переменными программы в теории без кванторов. Пусть $\mathrm{P} \subseteq \mathcal{P}$– это множество предикатов, а v: $\mathrm{X} \to \mathbb{Z}$ – это отображение из переменных в их значения. Определим $v \vDash \varphi$, где v называется моделью формулы φ.

Определим переименование переменных θ:X → X' из пространства имен X в X', которое применимо к формулам $\theta(\varphi)$. Обозначим

$\theta_{X,i} = \begin{cases} \mathrm{x} \mapsto \mathrm{x} \quad \mathrm{i}, \text{если x} \in \mathrm{X} \\ \mathrm{x} \mapsto \mathrm{x}, \text{иначе} \end{cases}$, и $\theta_{\mathrm{X,i}}^{-1} = \begin{cases} \mathrm{x} \quad \mathrm{i} \mapsto \mathrm{x}, \text{если x} \in \mathrm{X} \\ \mathrm{x} \mapsto \mathrm{x}, \text{иначе} \end{cases}$

Обозначим $(\varphi)^{\pi}$ декартову предикатную абстракцию формулы φ.

Обозначим $\mathrm{SP_{op}}(\varphi)$ – сильнейшее постусловие формулы φ и операции op.

Определим анализ предикатов $\mathbb{P}$ = ($\mathrm{D_P}$, Π_{P}, $\rightsquigarrow_{\mathrm{P}}$, $\mathrm{merge_P}$, $\mathrm{stop_P}$, $\mathrm{prec_P}$, $\mathrm{compatible_P}$, $\cdot|_{\mathrm{p}}$, $\mathrm{compose_P}$), который отслеживает выполнимость предикатов над переменными программы. Он состоит из следующих компонентов:

1. Абстрактный домен $\mathrm{D_P} = (\mathcal{T}_{\mathrm{P}}, \mathcal{E}_{\mathrm{P}}, \bigoplus_{\mathrm{P}})$. $\mathcal{E}_{\mathrm{P}} = (\mathrm{E_P}, \bot_{\mathrm{P}}, \top_{\mathrm{P}}, \sqsubseteq_{\mathrm{P}}, \bigsqcup_{\mathrm{P}})$. Абстрактный переход состоит из абстрактного состояния $s \in E_P^S$, а абстрактная дуга $q \in E_P^{\mathrm{T}}$, таким образом, $\mathrm{E_P} = E_P^S \times E_P^{\mathrm{T}}$, а $\mathcal{E}_{\mathrm{P}} = \mathcal{E}_P^S \times \mathcal{E}_P^{\mathrm{T}}$.

$E_P^S = \mathcal{P}$, таким образом, состояние является формулой первого порядка. Верхний элемент решетки является тождественно истинной формулой $\top_{\mathrm{P}}$ = true, а нижний элемент – тождественно ложной: $\bot_{\mathrm{P}}$ = false. Частичный порядок $\sqsubseteq_P^S \subseteq E_P^S \times E_P^S$ определяется как $e \sqsubseteq_P^S e' \Leftrightarrow e \Rightarrow e'$. Оператор слияния $\bigsqcup_P^S\colon E_P^S \times E_P^S \to E_P^S$ определяет ближайший элемент в соответствии с частичным порядком.

Абстрактная дуга $q \in E_P^{\mathrm{T}}$ – это действие, которое может быть выражено или формулой, или обычной CFA дугой: $E_P^{\mathrm{T}} = E_P^S \cup G$.

Не будем приводить формальное определение $\oplus_P$, так как оно требует достаточно много места. Основная идея состоит в том, что он возвращает множество конкретных переходов, которые соответствуют формуле (сильнейшему постусловию). Наиболее сложная часть определения это проверка, может ли множество частичных переходов соответствовать единственному глобальному переходу. Для перехода в потоке e_0 с обычной CFA дугой $q_0 \in G$ и переходов в окружении $e_1, \ldots, e_n$ проверка состоит из двух частей:

a) для абстрактного состояния s_0 из e_0 и всех состояний из переходов в окружении $s_1, \ldots, s_n$ существует общая модель v;

b) дуга q_p из проекции $e_0|_p$ покрывается абстрактными дугами q_j переходов в окружении $q_p \sqsubseteq q_j$.

Формально,

$$\forall \boldsymbol{e_0, e_1, \ldots, e_n} \in \boldsymbol{E_P}, \boldsymbol{e_i} = (\boldsymbol{s_i}, \boldsymbol{q_i}), \boldsymbol{s_i} \in \boldsymbol{E_P^S}, \boldsymbol{q_i} \in \boldsymbol{E_P^T}$$
$$\boldsymbol{C_{check}}(\boldsymbol{e_0}, \{\boldsymbol{e_1}, \ldots, \boldsymbol{e_n}\}) = \exists \boldsymbol{v}: \boldsymbol{v} \vDash \boldsymbol{s_0} \wedge \boldsymbol{\theta_{X^{local},1}}(\boldsymbol{s_1}) \wedge \cdots \wedge \boldsymbol{\theta_{X^{local},n}}(\boldsymbol{s_n}) \wedge$$
$$(\boldsymbol{q_0} \in \boldsymbol{G} \wedge \forall \boldsymbol{0} < \boldsymbol{j} \leq \boldsymbol{n}: \boldsymbol{q_j} \notin \boldsymbol{G} \wedge \boldsymbol{e_0}|_{\boldsymbol{p}} = (\boldsymbol{s_p}, \boldsymbol{q_p}): \boldsymbol{q_p} \sqsubseteq \boldsymbol{q_j}$$

2. Множество точности $\Pi_P = 2^P$ определяет точность абстрактного состояния как множество предикатов.

3. Оператор объединения *merge* может иметь несколько модификаций, например,

$\text{merge}_{\text{Join}}$ объединяет обе части перехода:

$$\forall e, e' \in E_P, \pi \in \Pi_p, e = (s, g):$$
$$merge_p(e, e', \pi) = \begin{cases} (s \vee s', g), \text{если } g = g', g \in G \\ e', \text{если } g \in G \wedge g' \in \mathcal{P} \vee g \in \mathcal{P} \wedge g' \in G \\ (s \vee s', g \vee g'), \text{если } g, g' \in \mathcal{P} \end{cases}$$

merge_{Eq} объединяет только абстрактные дуги при равных (или покрытых) состояниях.

$\text{merge}_{\text{Sep}}$ не объединяет элементы.

4. Оператор останова *stop* проверяет, покрывается ли переход e некоторым другим переходом в множестве достижимости: $stop_P(e, R, \pi) = \exists e' \in R : (e \sqsubseteq e')$.

5. Функция настройки точности *prec* вычисляет предикатную абстракцию над предикатами в точности π: $prec_P(e, \pi, R) = e^{\pi} = (s^{\pi}, q)$.

6. Отношение переходов $e \rightsquigarrow_P e', e = (s, g), e' = (s, \cdot)$. Так как анализ предикатов не отслеживает релевантные дуги, он возвращает все возможные.

Для g ∈ G существует переход $e \rightsquigarrow_P e'$ с g = (·, op, ·), если $s' = (SP_{op}(e))^{\pi}$.

Для $g = \varphi \in \mathcal{P}$ следующее абстрактное состояние имеет вид $s' = \hat{\varphi} \wedge e$.

7. Определение проекции:

$$\forall e \in E_P, e = (s, g): e|_P = \begin{cases} e, \text{если } g \notin G \\ \left(\theta_{X^{local},env}(s), \theta_{X^{local},env}\left(SP_{op}(s)\right)\right), \text{иначе} \end{cases}$$

Проекция определяет, как переход выглядит в качестве окружения потока. Локальные переменные переименовываются для избежания коллизии имен. Поэтому только предикаты над глобальными переменными остаются значимыми. Состояние (первая часть перехода) представляет абстракцию над глобальной частью состояния потока. А дуга (вторая часть перехода) соответствует конкретной операции над глобальными переменными.

8. $\forall e, e' \in E_P, e = (s, q), e' = (s', q'): compose_P(e, e') = \tilde{e} = (s, q')$

9. $\forall e_1, e_2 \in E_P, e_i = (s_i, q_i), s_i \in E_P^S: compatible_P(e_1, e_2) = \exists v: v \vDash \theta_{X^{local},1}(s_1) \wedge \theta_{X^{local},2}(s_2)$

Проверка совместности означает, что переходы могут быть объединены в один глобальный. А это невозможно, если глобальные предикаты неконсистентны, то есть не существует общей модели для двух частичных формул.

11. LockCPA

Определим анализ примитивов синхронизации $\mathbb{S} = (D_S, \Pi_S, \rightsquigarrow_S, \text{merge}_S, \text{stop}_S, \text{prec}_S, \text{compatible}_S, \cdot|_S, \text{compose}_S)$, который отслеживает множество захваченных блокировок (переменных синхронизации) для каждого потока. Он состоит из следующих компонентов:

1. Абстрактный домен $D_S = (\mathcal{I}_S, \mathcal{E}_S, \oplus_S)$. $\mathcal{E}_S = \mathcal{E}_S^S \times \mathcal{E}_S^T$. $\mathcal{E}_S^S = 2^S \cup \{\perp_S^S, \top_S^S\}$ – это множество всех подмножеств из переменных синхронизации, $\perp_S^S \sqsubseteq_S^S ls \sqsubseteq_S^S \top_S^S$ и $ls \supseteq ls' \Longrightarrow ls \sqsubseteq_S^S ls'$ для всех элементов $ls, ls' \subseteq S$. $E_S^T = \{\perp_S^T, \varepsilon, \top_S^T\} \cup G$.

2. Для этого анализа множество точности содержит только один элемент: $\Pi_S = \{\emptyset\}$.

3. Оператор перехода добавляет захватываемую блокировку во множество ls по время оператора *acquire* и удаляет ее из него во время оператора *release*. Формально, переход $e \rightsquigarrow_P e', e = (ls, g), e' = (ls', g'), g = (\cdot, op, \cdot)$ существует, если

- op = acquire(s) и $s \notin ls \wedge ls' = ls \cup \{s\}, g' \in G$.
- op = release(s) и $ls' = ls \backslash \{s\}, g' \in G$.
- op = thread create(l_v) и $ls' = \emptyset, g' \in G$
- иначе, $ls' = ls, g' \in G$.

4. Оператор слияния *merge* не объединяет элементы: $merge_S(e, e', \pi) = e'$.

5. Оператор останова *stop* проверяет, существует ли состояния, которое содержит меньше захваченных блокировок: $stop_S(e, R, \pi) = \exists e' \in R : (e \sqsubseteq e')$.

6. Точность не регулируется: $prec_S(e, \pi, R) = (e, \pi)$.

7. Определение проекции: $\forall e \in E_S, e = (s, g): e|_P = (s, \varepsilon)$.

Переходы в окружении (ε-переходы) не могут изменить множество захваченных блокировок, так как один поток не может повлиять на захваченные блокировки другого потока. Тем не менее, проекция сильно влияет на совместность состояний, так как потоки не могут захватить одну и ту же блокировку одновременно.

8. $\forall e, e' \in E_S, e = (ls, q), e' = (ls', q'): compose_S(e, e') = \tilde{e} = (ls, q')$.

9. $\forall e_1, e_2 \in E_S, e_i = (ls_i, q_i), ls_i \in E_S^S: compatible_P(e_1, e_2) = (ls \cap ls' = \emptyset)$. Проверка совместности сильно похожа на классический алгоритм Lockset. Если существует одна и та же блокировка в обоих состояниях, операции не могут быть объединены, так как два потока не могут дважды захватить одну и ту же блокировку.

12. CompositeCPA

Анализ используется для комбинации различных техник анализа вместе. Примеры таких анализов были описаны выше. $\mathbb{C} = (D_C, \Pi_C, \rightsquigarrow_C, \text{merge}_C, \text{stop}_C, \text{prec}_C, \text{compatible}_C, \cdot|_C, \text{compose}_C)$ включает в себя n вложенных анализов $\Delta_i = (D_i, \Pi_i, \rightsquigarrow_i, \text{merge}_i, \text{stop}_i, \text{prec}_i, \text{compatible}_i, \cdot|_i, \text{compose}_i)$.

1. Абстрактный домен $D_C = D_1 \times \cdots \times D_n$ реализует декартово произведение всех вложенных абстрактных доменов.

2. Множество точности также реализует декартово произведение вложенных множеств точности: $\Pi_C = \Pi_1 \times \cdots \times \Pi_n$.

3. Оператор перехода применяет вложенные операторы перехода к соответствующим частям состояния. Таким образом, переход возможен, если для всех вложенных анализов возможен переход между соответствующими вложенными состояниями:

$$\forall e_1, e_2 \in E_C: e_1 \rightsquigarrow_C e_2 \Leftrightarrow \forall 1 \le i \le n: e_1^i \rightsquigarrow_i e_2^i$$

4. Оператор слияния *merge* вызывает вложенные операторы: $merge_C(e_1, e_2, \pi) = (merge_1(e_1^1, e_2^1, \pi^1), \ldots, merge_n(e_1^n, e_2^n, \pi^n))$.

5. Оператор останова *stop* вызывает вложенные операторы: $\forall e \in E, R \subseteq E, \pi \in \Pi: stop_C(e, R, \pi) = \exists \hat{e} \in R, \forall 1 \le i \le n : stop_i(e^i, \{\hat{e}^i\}, \pi^i)$.

6. Аналогично оператор настройки точности вызывает вложенные операторы: $prec_C(e, \pi, R) = (prec_1(e^1, \pi^1, R), \ldots, prec_n(e^n, \pi^n, R))$.

7. Аналогично оператор вычисления проекции вызывает вложенные операторы: $\forall e \in E_C, : e|_P = (e^1|_P, \ldots, e^n|_P)$.

8. $\forall e_1, e_2 \in E_C : compose_C(e_1, e_2) = (compose_1(e_1^1, e_2^1), \ldots, compose_n(e_1^n, e_2^n))$.

9. $\forall e_1, e_2 \in E_C : compatible_C(e_1, e_2) = (compatible_1(e_1^1, e_2^1), \ldots, compatible_n(e_1^n, e_2^n))$.

13. Поиск состояний гонки

Как уже было описано, поиск состояний гонки разбивается на два этапа: построение множества достижимых состояний и поиск пар, которые образуют состояние гонки. Для решения первой подзадачи применяется набор из тех CPA, которые были описаны в предыдущих разделах.

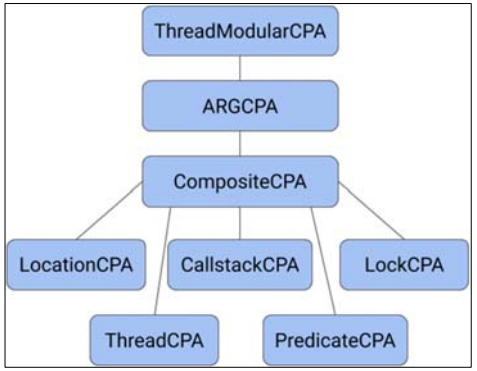

Рис. 9. Пример конфигурации CPA
Fig. 9. An example of CPA configuration

На рис. 9 представлен пример конфигурации инструмента. Так, набор CPA содержит ThreadModularCPA (раздел 6), в качестве верхнеуровневого. В него вложен ARGCPA, который обеспечивает связь между различными абстрактными состояниями, в том числе связи «родительский переход – дочерний переход», «проецируемый переход – проекция» и др. CompositeCPA (разд. 12) обеспечивает параллельную работу вложенных в него CPA: LockationCPA (разд. 7) моделирует счетчик команд потока, CallstackCPA обеспечивает межпроцедурность анализа, LockCPA (разд. 11) отслеживает захваты блокировок, ThreadCPA(разд. 8) отслеживает точки создания потоков, PredicateCPA(разд. 10) реализует анализ предикатов. Важно отметить, что это не единственная возможная конфигурация инструмента. Более того, для решения различных задач могут потребоваться различные конфигурации, то есть наборы, используемых техник анализа.

Итак, с помощью некоторой конфигурации инструмента производится построение абстрактного графа достижимости (англ. Abstract Reachability Graph, ARG) из абстрактных переходов. То есть, по завершению этого этапа имеется граф из переходов программы, которые являются достижимыми при заданном уровне абстракции, то есть, совсем не обязательно они будут достижимыми при реальном выполнении программы.

Следующим этапом необходимо найти среди этих переходов те, которые образуют состояние гонки.

Обычно под состоянием гонки подразумевается ситуация, при которой имеет место одновременный доступ к некоторой памяти из разных потоков, причем один из доступов должен быть записью. Два основных вопроса, которые возникают при статическом поиске гонок: как определить, что доступ производится к одной и той же памяти, и как определить одновременность доступов. Дальше будут подробно представлены эти две особенности в предлагаемом подходе.

При описании формальной модели программы использовался простой императивный язык программирования, который поддерживает разделяемые данные, представленные только глобальными переменными. В реальном программном обеспечении используется большое количество операций с указателями, структурами и более сложными типами данных. Для работы с ними используется модель памяти BnB, которая разделяет всю память на непересекающиеся множества регионов. Каждый регион относится к одному типу данных или полю структуры, в случае если у него не брался адрес. Такая модель памяти имеет ряд ограничений. В первую очередь, она не полностью поддерживает адресную арифметику и кастирование, что накладывает некоторые дополнительные условия на применимость подхода. Кроме того, она может приводить к ложным предупреждениям об ошибках, в случае если два указателя одного типа никогда не указывают на одну и ту же память.

В случае, если обнаружилась пара доступов к одной области памяти, в данном случае, к одному региону, необходимо проверить возможность одновременного доступа к ней. Для этого используется понятие совместности переходов. Совместность означает, что два частичных абстрактных состояния могут быть частью одного глобального состояния. Или, другими словами, один переход может быть применен, как переход в окружении, к другому переходу и наоборот, откуда следует, что эти два перехода могут быть выполнены параллельно. Таким образом, предложенный подход является обобщением подхода Lockset[13], который определяет состояние гонки, как два доступа к некоторой памяти с непересекающимся множеством блокировок. Одним из ограничений подхода Lockset является отсутствие поддержки других типов синхронизации. В расширении подхода для этого используется оператор compatible. Так как проверка совместности использует различные типы анализа, включая анализ примитивов синхронизации, анализ предикатов и другие, такой способ является более точным, чем алгоритм Lockset.

Итак, алгоритм поиска состояний гонки состоит из следующих шагов:

1. построить множество достижимых абстрактных переходов (алгоритм 1);
2. для каждого перехода определить регионы памяти, к которым может производиться доступ;
3. для каждого региона памяти попытаться подобрать пару совместных переходов, которые образуют состояние гонки;
4. проверить каждую пару путей, приводящих к состоянию гонки, на достижимость и уточнить предикатную абстракцию в случае необходимости [21].

Следует добавить, что алгоритм уточнения абстракции по контрпримерам (англ. Counterexample Guided Abstraction Refinement, CEGAR [11]) был использован без существенных модификаций. Однако, в таком виде он позволяет проводить уточнение только локальных путей в потоке, то есть не позволяет обнаружить противоречие между различными потоками. Однако, это не является принципиальным ограничением подхода, и при соответствующем расширении метода CEGAR на случай подхода с раздельным рассмотрением потоков, возможно получение более точных результатов.

14. Результаты

Подход с раздельным помодульным анализом потоков и абстракными переходами был реализован в инструменте CPAchecker, как композиция следующих типов анализа:

1. Анализ точек программы (англ. Location analysis) L;
2. Анализ стека вызовов функци (англ. Callstack analysis) CS;
3. Анализ потоков (англ. Thread Analysis) T;
4. Анализ примитивов синхронизации (англ. Lock analysis) S (раздел 8);
5. Анализ предикатов с опциями (англ. Predicate analysis) P (раздел 7):
   I. Join – объединение абстрактных состояний и абстрактных дуг. В данном варианте все эффекты окружения предикатного анализа объединяются в один, что, естественно, снижает точность, но повышает скорость.
   II. Eq – объединение абстрактных дуг, если равны абстрактные состояния;
   III. Sep – переходы никогда не объединяются.
6. Анализ явных значений (Value analysis) V [20].

14.1 Результаты на наборе задач SV-COMP

Набор задач SV-COMP состоит из 1082 задач, большая часть из которых являются небольшими примерами около 100 строк кода. Однако, они содержат в себе нетривиальные механизмы синхронизации, в том числе алгоритмы Деккера, Петерсона и др. 7 задач были подготовлены на основе драйверов ОС Linux. Все задачи доступны в официальном репозитории SV-COMP[4]. Эксперименты проводились с использованием кластера из 191 машины VerifierCloud[5]. Были использованы ограничения по памяти в 8 Гб и по времени 15 минут.

Оценка результатов реализованного подхода в инструменте CPAchecker была проведена для следующих вариантов анализа:

1. Подходы с раздельным анализом потоков с абстракцией переходов:
   a. Варианты объединения для предикатного анализа
      i. *MergeJoin*. (L,CS,T,S,P) Join.
      ii. *MergeEq*. (L,CS,T,S,P) Eq.
      iii. *MergeSep*. (L,CS,T,S,P) Sep.
   b. *Value*. Анализ явных значений (L, CS, T, S, V).
2. *Threading*. Анализ чередований, описанный в [17] использует классическую теорию CPA и рассматривает все возможные чередования потоков.

Табл. 1. Результаты экспериментов
Table 1. Experiment Results

Подход	MergeJoin	MergeEq	MergeSep	Value	Threading
Найденные ошибки	1026	1027	763	963	720
Истинные	805	806	548	752	720
Ложные	221	221	215	211	0
Доказательства корректности	27	28	28	30	165
Истинные	27	28	28	30	165
Ложные	0	0	0	0	0

[4] https://github.com/sosy-lab/sv-benchmarks
[5] https://vcloud.sosy-lab.org/cpachecker/webclient/master/info

Незавершенный анализ	29	27	297	89	197
Процессорное время, с	16 800	15 900	278 000	75 300	93 700
Астрономическое время, с	10 200	9 630	258 000	64 900	67 500

Результаты (табл. 1) подтверждают, что подход с раздельным анализом потоков не пропускает ошибок при некоторых заранее известных ограничениях.

MergeJoin показывает результаты лучше, чем конфигурация *MergeSep*. Это происходит, в основном, из-за большого количества переходов в окружении, которые *MergeSep* рассматривает по одному. *MergeJoin* объединяет их в один и применяет за один раз. Это позволяет сохранить огромное количество времени. В то же время *MergeSep* позволяет избежать некоторых неточностей из-за анализа переходов по-отдельности и выдает меньшее количество ложных сообщений об ошибках.

Конфигурация *Value* является достаточно простым и быстрым анализом, но иногда она вынуждена рассматривать все возможные варианты значений переменных, что приводит к исчерпанию ресурсов по времени.

Классический анализ *Threading* является корректным и точным и не выдает ложных вердиктов. Однако, он требует значительного числа ресурсов, это является главным недостатком подхода. Эта конфигурация решила только один из семи сложных задач, основанных на драйверах операционной системы Linux. Подходы с анализом потоков по-отдельности (*MergeJoin*, *MergeEq*, *MergeSep*) решают пять из семи таких задач.

Большая часть новых доказательств корректности, полученных новыми подходами, (26 из 27 для *MergeJoin*) не находились классическими методами (*Threading*). Это также является одним из важных вкладов данного метода.

14.2 Поиск состояний гонки в драйверах устройств

Верификационные задачи, основанные на драйверах операционной системы Linux, были подготовлены системой Klever, которая предназначена для верификации различного программного обеспечения [17, 18]. Она разделяет большой объем целевого исходного кода на отдельные небольшие верификационные задачи. Для ядра операционной системы Linux верификационная задача соответствует одному модулю. Система Klever автоматически готовит модель окружения модуля, которая включает в себя модель потоков, модель сердцевины ядра и операций над модулем. После подготовки верификационной задачи Klever запускает верификацию через общий интерфейс – BenchExec [23].

Сравнение проводилось на подсистеме drivers/net/ ядра операционной системы Linux 4.2.6, для которой Klever подготовил 425 верификационных задач. Эксперименты также проводились с использованием кластера из 191 машины VerifierCloud. Были использованы ограничения по памяти в 8 Гб и по времени 15 минут.

Алгоритм поиска состояний гонки с помощью инструмента был описан в разд. 12. При этом подготовленные задачи не содержали кодирование состояния гонки в виде проверки задачи достижимости, поэтому было невозможно снова включить в сравнение участников соревнования SV-COMP, так как они не поддерживают поиск состояний гонки.

В сравнении (табл. 2) участвовали следующие конфигурации:

1. *Base*. Конфигурация *MergeJoin* из предыдущего пункта.

2. *Havoc*. Конфигурация *MergeJoin* из предыдущего пункта без возможности извлечения предикатов над глобальными переменными. Это означает, что анализ не учитывает значения глобальных переменных и считает, что они могут иметь случайное значение.

Табл. 2. Сравнение конфигураций Base и Havoc
Table 2. Comparing Base and Havoc Configurations

Подход	Base	Havoc
Модули с ошибкой	6	26
Корректные модули	262	254
Незавершенный анализ	157	145
Процессорное время, с	137 000	125 000

Модули с ошибкой означают, что в них было найдено хотя бы одно потенциальное состояние гонки. Конфигурация *Havoc* чуть более быстрая, но менее точная, поэтому она не может доказать корректность 8 корректных модулей, однако способна обнаружить больше ошибок. При этому 6 из этих ошибок были найдены в пропущенных корректных модулях, что означает, что найденные ошибки – ложные из-за неточности анализа. Но 13 ошибок были соответствуют незавершенному анализу у *Base* конфигурации, что означает, что *Havoc* может находить новые ошибки.

Если изучить эти 8 модулей, которые были доказаны, как корректные, то окажется, что в них структура данных устройства (дескриптора устройства) была некорректно инициализирована при подготовке верификационной задачи, и поэтому в ней действительно нет состояния гонки. На самом деле проблема связана с подготовкой модели окружения модуля (часть инфраструктуры Klever), так как они не учитывает семантику данных.

Часть полученных ошибок была проанализирована. Обычно для каждой верификационной задачи могут быть получены несколько потенциальных состояний гонки. Будем называть их предупреждениями. Соотношение истинных предупреждений к общему количеству составляет около 42% (34 корректных предупреждения и 47 некорректных). Основной проблемой ложных предупреждений об ошибках являются проблемы с моделью памяти (более 90%). Остальные предупреждения связаны со спецификой ядра (например, обработкой прерываний), анализом функциональных указателей, другими примитивами синхронизации и др. Интересно, что таких проблем, связанных с неточностью подхода, как были в задачах SV-COMP, в модулях операционной системы Linux не наблюдается. Это подтверждает предположение, что задачи SV-COMP являются слишком искусственными, по крайней мере в категории многопоточных задач (Concurrency).

Из 24 сообщений о возможном состоянии гонки 14 были подтверждены разработчиками, для 8 не было получено обратного ответа, и только в двух случаях найденная гонка была признана недостижимой из-за аппаратной синхронизации устройства. Однако большинство истинных ошибок были найдены в «древних» драйверах, а их исправление не является актуальной задачей. Только 4 исправления из 14 были включены в основную ветку разработки ядра.

Результаты экспериментов позволяют утверждать, что удалось успешно интегрироваться с другими техниками анализа, которые доступны во фреймворке CPAchecker. Такие подходы, как CEGAR, могут быть применены с минимальным количеством изменений технического характера. Некоторые реализации конкретных видов анализа могут быть использованы без модификаций (AutomatonCPA) или с минимальными изменениями в коде (CallstackCPA). Для некоторых существующих техник анализа (PredicateCPA, ValueCPA, CompositeCPA) была реализована поддержка операций над эффектами окружения, при этом основная функциональность анализа осталась без изменений. Изменения параметров работы инструмента (вариантов анализа) достигается лишь опцией запуска, то есть пересобирать инструмент не требуется.

Из результатов видно, что для разных целей может быть использованы различные варианты работы инструмента: для небольших задач, типа SV-COMP, необходимы более точные подходы. Для прикладных задач, типа драйверов, могут быть полезны менее точные варианты анализа, чтобы иметь возможность получить некоторую оценку корректности. Вместе с тем, более точные подходы, могут применяться для глубокой верификации, которая требует определенных ресурсов.

15. Обзор похожих работ

Существующие подходы к анализу многопоточного программного обеспечения имеют различные свойства и производительность. С одной стороны, существуют точные подходы, которые могут доказать корректность программ при некоторых определенных предположениях. Начиная с ограничиваемой проверки моделей, большинство подходов прилагают значительные усилия для оптимизации обхода пространства состояний программы. Примерами таких оптимизаций могут стать редукция частичных порядков [1], ограничение контекста [24, 25], и др.

Попыткой абстрагироваться от нерелевантного окружения является подход с раздельным анализом потоков (thread-modular approach), который впервые был предложен в [7]. Эта версия еще не использовала никакую абстракцию. Подход с раздельным анализом потоков для формальной верификации был представлен в [26]. Основная идея была в поиске инвариантов для каждого процесса, из которых потом должно следовать проверяемое свойство. Оценка подхода проводилась для двух протоколов взаимного исключения. Предикатная абстракция была добавлена к этому подходу в [27]. Основным отличием было то, что рассматривался только один поток из нескольких копий. Поэтому окружение формировалось этим же потоком. При этом никакие примитивы синхронизации не рассматривались.

Расширение подхода с раздельным анализом потоков, которое использует абстракцию, был представлен в [28] и затем реализован в инструменте TAR [8]. Описанный в данной статье подход имеет ряд отличий:

TAR рассматривает примитивы синхронизации, как обычные переменные. Предложенный метод использует специальный анализ блокировок, который может применяться совместно с другими вариантами анализа. Это позволяет избегать лишних итераций уточнения абстракции.

TAR применяет эффекты потока, которые напрямую получены из операторов программы. CPALockator предоставляет возможность абстракции (оператор $e|_P$) и объединения (оператор $merge$) различных переходов. Это может позволяет выбирать баланс между скоростью анализа и его точностью.

TAR поддерживает фиксированное количество потоков, в то время как предложенный метод теоретически поддерживает неограниченное количество потоков.

Для построения окружения TAR использует аппроксимацию снизу, а предложенный метод – аппроксимацию сверху.

Похожий подход также был реализован в инструменте Threader [29]. Этот инструмент использует аппроксимацию сверху для построения окружения, основанную на дизъюнктах Хорна. Также как и в инструменте CPALockator, Threader может искать модульные доказательства, но при этом он может искать и немодульные, в которых учитывается полной взаимодействие между потоками.

Существуют различные техники трансформации параллельной программы в последовательную (англ. sequentialization) для последующей проверки ее обычными инструментами статической верификации [30–32]. Один из примеров является WHOOP [33], который использует эту технику трансформации и не учитывает взаимодействие потоков. Более того, он сильно использует алгоритм Lockset и не может быть расширен

какими-нибудь другими анализами. Он применяется в качестве преданализа для более точного инструмента CORALL [34].

С другой стороны, существуют множество легковесных подходов, которые могут применяться к огромному объему кода. Такие техники определяются слабыми требованиями к ресурсам и низкой точностью. Примерами являются RELAY [35] и Locksmith [36], которые применялись для анализа кода операционной системы Linux. RELAY нашел несколько тысяч предупреждений, что потребовало применения неточных фильтров для сокращения этого количества. Этот инструмент вообще не учитывает взаимодействие потоков.

Инструмент Locksmith, напротив, учитывает точки создания потоков, но не может точно определить разделяемые данные и способы взаимодействия потоков. Он вычисляет общий эффект от потока. В терминах представленной теории используется оператор $merge$, объединяющий все переходы в окружении в один. При этом экспериментальная оценка Locksmith показала, что он имеет проблемы с масштабируемостью.

В [37] авторы представили расширение анализа алиасов Андерсена на случай многопоточных программ. Идея был похожа на подход с раздельным анализом потоков, так как вычислялось множество операторов, которое могло выполняться параллельно, а затем эти операторы применялись к другим потокам.

Также существуют различные подходы для поиска состояний гонки в специфичном программном обеспечении. Например, в низкоуровневом программном обеспечении со вложенными прерываниями [38], поиск гонок во FreeRTOS [39, 40]. А также состояний гонки специального вида – использования памяти после ее освобождения (англ. use-after-free) в драйверах устройств операционной системы Linux. Такие подходы показывают достаточно хорошие результаты, но значительно используют специфику исходного кода или проверяемого свойства. Представленный в данной статье теоретический подход претендует на то, чтобы быть более общим, однако это не отменяет того факта, что он может быть также улучшен при использовании дополнительной информации об исходном коде или проверяемом свойстве.

16. Заключение

В работе представлен подход к практическому поиску состояний гонки в сложном программном обеспечении. Была расширена теория CPA и реализована в новом инструменте. Эксперименты показали преимущества подхода на больших верификационных задачах перед существующими техниками статической верификации. Небольшие задачи со сложным взаимодействием потоков лучше решаются другими инструментами, так как предложенный подход абстрагируется от такого взаимодействия. Однако наш подход также не пропускает ошибок (при некоторых предположениях) и может быть развит в будущем.

Таким образом, можно заключить, что основные требования к новому инструменту были выполнены, так как он успешно применяется к различным программным системам, в том числе, к драйверам ОС Linux. Так же, как и в классических методах статической верификации, может быть получена гарантия отсутствия ошибок при выполнении указанных предположений: требований на операторы CPA и условий корректного разбиения на непересекающиеся регионы BnB модели.

Предложенная теория позволяет описывать достаточно сложные варианты анализа, в том числе и те, которые не включаются в понятие анализа с раздельным рассмотрением потоков. Тем не менее, эта теория не исчерпывает всех возможных подходов, и для эффективного описания масштабируемого подхода с частичным вычислением чередований потребуется ее расширение. Однако, эта тема выходит за рамки данной статьи.

Одним из возможных направлений развития подхода является добавление взаимодействия потоков, возможно, не в полном объеме, чтобы сохранить масштабируемость. Это может быть отдельный анализ CPA, который будет динамически настраивать свою точность с помощью алгоритма CEGAR, обеспечивая некоторый промежуточный вариант между подходом с раздельным анализом потоков и перебором чередований.

Другим возможным улучшением инструмента может стать интеграция его с другим подходом. Например, комбинация быстрого подхода с раздельным анализом потоков в качестве первого этапа, а затем классический тяжеловесный анализ – на втором этапе. Это может быть реализовано в соответствии с идеей кооперативной верификации [42].

Еще одним возможным направлением развития подхода является построение реального пути с чередованием потоков на основе пути с примененными эффектами окружения. Этот путь был бы полезен для исследования и уточнения абстракции.

Список литературы / References

[1]. Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Optimal dynamic partial order reduction. SIGPLAN Notices, vol. 49, issue 1, 2014, pp. 373–384.

[2]. Patrice Godefroid. Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem. Springer-Verlag, Berlin, Heidelberg, 1996, 143 p.

[3]. Gérard Basler, Michele Mazzucchi, Thomas Wahl, and Daniel Kroening. Symbolic counter abstraction for concurrent software. Lecture Notes in Computer Science, vol. 5643, 2009, pp. 64–78.

[4]. Dirk Beyer. Automatic verification of C and Java Programs: SV-COMP. Lecture Notes in Computer Science, vol. 11429, 2019, pp. 133–155.

[5]. Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. Configurable software verification: concretizing the convergence of model checking and program analysis. Lecture Notes in Computer Science, vol. 4590, 2007, pp. 504–518

[6]. D. Beyer, T.A. Henzinger, and G. Theoduloz. Program analysis with dynamic precision adjustment. In Proc. of the 23rd IEEE/ACM International Conference on Automated Software Engineering, 2008, pp. 29–38.

[7]. Cormac Flanagan and Shaz Qadeer. Thread-modular model checking. Lecture Notes in Computer Science, vol. 2648, 2003, pp. 213–224.

[8]. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Shaz Qadeer. Thread-Modular Abstraction Lecture Notes in Computer Science, vol. 2725, 2003, pp, 262–274.

[9]. Byron Cook, Daniel Kroening, and Natasha Sharygina. Verification of Boolean programs with unbounded thread creation. Theoretical Computer Science, vol. 388, issue 1-3, 2007, pp. 227–242.

[10]. Ashutosh Gupta, Corneliu Popeea, and Andrey Rybalchenko. Threader: A constraint-based verifier for multi-threaded programs. Lecture Notes in Computer Science, vol. 6806, 2011, pp. 412–417.

[11]. E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. Lecture Notes in Computer Science, vol. 1855, 2000, pp. 154–169.

[12]. Susanne Graf and Hassen Saidi. Construction of abstract state graphs with PVS. Lecture Notes in Computer Science, vol. 1254, 1997, pp. 72–83.

[13]. Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multi-threaded programs. ACM SIGOPS Operating Systems Review, vol. 31, issue 5, 1997, pp. 27–37.

[14]. Richard Bornat. Proving pointer programs in Hoare logic. Lecture Notes in Computer Science, vol. 1837, 2000, pp. 102–126.

[15]. R M Burstall. Some techniques for proving correctness of programs which alter data structures. Machine Intelligence, vol. 7, 1972, pp. 23–50.

[16]. Pavel Andrianov, Karlheinz Friedberger, Mikhail Mandrykin, Vadim Mutilin, and Anton Volkov. CPA-BAM-BnB: Block-abstraction memoization and region-based memory models for predicate abstractions. Lecture Notes in Computer Science, vol. 10206, 2017, pp. 355–359.

[17]. Evgeny Novikov and Ilja Zakharov. Towards automated static verification of GNU C programs. Lecture Notes in Computer Science, vol. 10742, 2018, pp. 402–416.

[18]. Evgeny Novikov and Ilja Zakharov. Verification of operating system monolithic kernels without extensions. Lecture Notes in Computer Science, vol. 11247, 2018, pp. 230–248.
[19]. Dirk Beyer and Karlheinz Friedberger. In Proc. of the 11th Doctoral Workshop on Mathematical and Engineering Methods in Computer Science, 2016, pp. 61–71.
[20]. Dirk Beyer and Stefan Löwe. Explicit-state software model checking based on CEGAR and interpolation. Lecture Notes in Computer Science, vol. 7793, 2013, pp. 146–162.
[21]. D. Beyer, M.E. Keremoglu, and P. Wendler. Predicate abstraction with adjustable-block encoding. In Proc. of 10th International Conference on Formal Methods in Computer-Aided Design, 2010, pp. 189–197.
[22]. Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. Lecture Notes in Computer Science, vol. 1579, 1999, pp. 193–207
[23]. Dirk Beyer, Stefan Löwe, and Philipp Wendler. Reliable benchmarking: requirements and solutions. International Journal on Software Tools for Technology Transfer, vol. 21, issue 1, 2019, pp. 1–29.
[24]. Shaz Qadeer and Jakob Rehof. Context-bounded model checking of concurrent software. Lecture Notes in Computer Science, vol. 3440, 2005, pp. 93–107.
[25]. Lucas Cordeiro, Jeremy Morse, Denis Nicole, and Bernd Fischer. Context-bounded model checking with esbmc 1.17. Lecture Notes in Computer Science, vol. 7214, 2012, pp. 534–537.
[26]. Ariel Cohen and Kedar S. Namjoshi. Local proofs for global safety properties. Formal Methods in System Design, vol. 34, issue 2, 2009, pp. 104–125.
[27]. Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Race checking by context inference. In Proc. of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation, 2004, pp. 1–13.
[28]. Alexander Malkis, Andreas Podelski, and Andrey Rybalchenko. Thread-modular verification is cartesian abstract interpretation. Lecture Notes in Computer Science, vol. 4281, 2006, pp. 183–197.
[29]. Ashutosh Gupta, Corneliu Popeea, and Andrey Rybalchenko. Predicate abstraction and refinement for verifying multi-threaded programs. ACM SIGPLAN Notices, vol. 46, issue 1m 2011, pp. 331–344.
[30]. Akash Lal and Thomas Reps. Reducing concurrent analysis under a context bound tosequential analysis. Formal Methods in System Design, vol. 35, issue 1, 2009, pp. 73–97.
[31]. Salvatore La Torre, P. Madhusudan, and Gennaro Parlato. Reducing context-bounded concurrent reachability to sequential reachability. Lecture Notes in Computer Science, vol. 5643, 2009, pp. 477–492.
[32]. Ermenegildo Tomasco, Omar Inverso, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. MU-CSeq: Sequentialization of C programs by shared memory unwindings. Lecture Notes in Computer Science, vol. 8413, 2014, pp. 402–404.
[33]. Pantazis Deligiannis, Alastair F. Donaldson, and Zvonimir Rakamaric. Fast and precise symbolic analysis of concurrency bugs in device drivers (t). In Proc. of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2015, pp. 166–177.
[34]. Akash Lal, Shaz Qadeer, and Shuvendu K. Lahiri. A solver for reachability modulo theories. Lecture Notes in Computer Science, vol. 7358, 2012, pp. 427–443.
[35]. Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. RELAY: Static race detection on millions of lines of code. In Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, 2007, pp. 205–214.
[36]. Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. LOCKSMITH: Context-sensitive correlation analysis for race detection. In Proc. of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2006, pp. 320–331.
[37]. Peng Di and Yulei Sui. Accelerating dynamic data race detection using static thread interference analysis. In Proc. of the 7th International Workshop on Programming Models and Applications for Multicores and Manycores, 2016, pp. 30–39.
[38]. Daniel Kroening, Lihao Liang, Tom Melham, Peter Schrammel, and Michael Tautschnig. Effective verification of low-level software with nested interrupts. In the Europe Conference & Exhibition on Design, Automation & Test, 2015, pp. 229–234.
[39]. Suvam Mukherjee, Arun Kumar, and Deepak D'Souza. Detecting all high-level dataraces in an RTOS kernel. Lecture Notes in Computer Science, vol. 10145, 2017, pp. 405–423.
[40]. Nikita Chopra, Rekha Pai, and Deepak D'Souza. Data races and static analysis for interrupt-driven kernels. Lecture Notes in Computer Science, vol. 11423, 2019, pp. 697–723.

[41]. Jia-Ju Bai, Julia Lawall, Qiu-Liang Chen, and Shi-Min Hu. Effective static analysis of concurrency use-after-free bugs in Linux device drivers. In the USENIX Annual Technical Conference, 2019, pp. 255–268.
[42]. Dirk Beyer, Marie-Christine Jakobs, Thomas Lemberger, and Heike Wehrheim. Reducer-based construction of conditional verifiers. In Proceedings of the 40th International Conference on Software Engineering, 2018, pp. 1182–1193.

Информация об авторах / Information about authors

Павел Сергеевич АНДРИАНОВ – младший научный сотрудник. Сфера научных интересов: статическая верификация, анализ многопоточных программ.

Pavel Sergeevich ANDRIANOV – junior researcher. Research interests: software model checking, analysis of multithreaded software.