Введение в метод CEGAR — уточнение абстракции по контрпримерам

Мандрыкин М. У., Мутилин В. С., Хорошилов А. В.
mandrykin@ispras.ru, mutilin@ispras.ru, khoroshilov@ispras.ru

Аннотация. Точность, полнота и масштабируемость применяемых на практике инструментов статической верификации значительно возросла за последнее десятилетие. В частности, успешный автоматизированный анализ программных систем среднего размера с использованием проверки моделей, получаемых при помощи предикатной абстракции, стал возможен благодаря развитию метода уточнения абстракции по контрпримерам — CEGAR (Counter Example Guided Abstraction Refinement). Этот метод так или иначе используется в таких инструментах, как SLAM, BLAST, SATABS и CPAchecker. В рамках данной статьи мы рассмотрим метод CEGAR в том виде, как он реализован в инструментах статической верификации BLAST и CPAchecker.

Ключевые слова: статическая верификация, предикатная абстракция, проверка моделей, уточнение абстракции по контрпримерам, интерполяция Крейга, крупноблочное кодирование.

1 Введение

По мере того, как растет сложность программных систем и их роль в жизни общества, растет и потребность в эффективных методах проверки их корректности. Выделяют две основные группы методов проверки корректности программ: динамические методы (или методы тестирования) и статические.

Динамические методы выполняют проверку корректности программы на основе результатов наблюдения за её поведением в ходе работы в реальном или модельном окружении. Этот подход позволяет обнаружить только те ошибки, которые проявляются в ситуациях, воспроизводимых в процессе тестирования, поэтому качество проверки во многом и определяется тем, какие ситуации удается воспроизвести. Поскольку даже в умеренно сложной программе число различных ситуаций, потенциально приводящих к ошибкам, таково, что протестировать все их за разумное время не представляется возможным, то динамические методы нацелены скорее на обнаружение как можно большего числа ошибок, чем на доказательство их полного отсутствия. Опыт использования программных систем подтверждает этот факт примерами

печальных последствий вовремя не обнаруженных ошибок в программном обеспечении [1], в случае которых тщательное тестирование было либо трудноосуществимо, либо эти ошибки не выявило.

Статические методы выполняют анализ корректности исходного или бинарного кода программы без его выполнения и потенциально имеют возможность для обнаружения всех ошибок и доказательства корректности программы. Доказанная в 1936 году теорема об алгоритмической неразрешимости проблемы останова [2] показала невозможность существования полного и корректного алгоритма для автоматического доказательства корректности программ в рамках любой достаточно полной модели вычислителя. Поэтому исследования по развитию статических методов велись по двум направлениям: более или менее полное доказательство корректности программы с привлечением подсказок человека и полностью автоматический анализ с менее амбициозными целями, которые могут варьироваться от нахождения хоть каких-то типовых ошибок до доказательства выполнения определенных свойств программ при условии выполнения некоторых предположений.

1.1 Методы полуавтоматического анализа программ

Изначально преобладало направление разработки методов ручного доказательства, а также аксиоматической семантики и логических методов для работы с программами как с логическими объектами [3, 4, 5]. По мере роста размера и сложности программных систем ручное доказательство корректности их работы становилось слишком обременительным и порой давало повод усомниться в том, что получаемые в результате большие и сложные формальные доказательства заслуживают доверия [6].

Так появилась тенденция к автоматизации процесса верификации, при которой человек управляет процессом доказательства корректности, предоставляя инструменту необходимые утверждения (например, пред- и постусловия функций и инварианты циклов [3]). Это позволило расширить область применения методов автоматизации доказательства корректности программ, как в смысле их размеров, так и в смысле разнообразия проверяемых свойств.

Позднее область автоматизированной верификации программ испытала влияние трёх различных направлений исследований

- Развитие подходов к решению задачи выполнимости, сформулированной в рамках различных логических теорий, и непосредственно связанные с этим разработки решателей [7, 8, 9]. Эти инструменты предоставили собой практически применимое средство автоматического доказательства утверждений в бесконечных пространствах состояний.

- Разработка техник автоматической проверки моделей программ [10, 11, 12] на выполнимость в них свойств, сформулированных на языке темпоральных логик [13, 14], и соответствующих инструментов, основанных на обходе конечного пространства состояний модели.
- Формализация анализа программ в виде абстрактной интерпретации позволила использовать логические методы, оперирующие с бесконечными пространствами состояний при помощи алгоритмических конечных представлений.

В промежутке между 1980-ми и 1990-ми годами эти три направления развивались по большей части независимо с относительно редкими пересечениями. Однако к началу 1990-х произошло их сближение, в результате которого современные подходы к верификации программ основаны на объединении и одновременном использовании всех упомянутых техник. В частности, когда говорят “инструмент проверки моделей программ” – это, возможно, не отражает верного представления о современных инструментах автоматической статической верификации, поскольку большинство из них одновременно осуществляют различные виды анализа программ, такие как автоматическое доказательство теорем, проверка моделей и анализ потока данных [15].

В настоящее время направления исследований в области формальной верификации достаточно разнообразны. Для практического применения представляют интерес методы, которые достаточно хорошо автоматизируются и масштабируются для того, чтобы справляться с верификацией больших и сложных программных систем. Обзоры современных техник автоматизированной верификации программ можно найти, например, в [16], [17] и [18].

1.2 Методы автоматической верификации

Ключевыми характеристиками методов автоматической проверки программ являются

- полнота, то есть набор ограничений на используемые возможности языка программирования и виды обнаруживаемых ошибок;
- точность анализа, то есть количество выдаваемых ложных сообщений об ошибках;
- скорость их работы.

На основании этих характеристик существующие подходы можно условно разделить на два больших класса – легковесные и тяжеловесные.

1.2.1 Легковесные подходы

Легковесные подходы берут своё начало из области разработки компиляторов и программной оптимизации. С самого начала к ним предъявлялись высокие требования к скорости работы, потому что с их помощью должны были

достаточно быстро обрабатываться большие объёмы исходного кода. Для достижения скорости работы, сравнимой по порядку величины со временем компиляции анализируемого приложения, в данных подходах обычно применяется анализ потока данных [15] в сочетании с множеством различных эвристик, использование которых может привести к пропуску ошибок и снижению точности анализа. На сегодняшний день легковесные подходы развиты достаточно хорошо и существует большое количество реализующих их инструментов, как для индустриального, так и для академического использования [19, 20, 21, 22, 23].

Основной целью этих инструментов является нахождение наибольшего числа типовых ошибок, встречающихся при разработке программ на распространенных языках программирования, при приемлемом уровне ложных срабатываний. Примерами обнаруживаемых ошибок являются разыменование нулевого указателя, обращение за границами буфера, обращение к памяти из кучи после её освобождения и т. д. Для добавления поиска нового вида ошибок в легковесные инструменты, как правило, требуется разработка модуля на языке программирования, используемом для реализации инструмента.

1.2.2 Тяжеловесные подходы

Тяжеловесные подходы характеризуются тем, что они предъявляют существенно меньшие требования к скорости анализа. Это позволяет применять более полные и точные методы, которые при выполнении определенных ограничений на код программы позволяют доказать отсутствие в программе ошибок определенных видов. Большинство тяжеловесных подходов предназначено для работы с ошибками, выражаемыми в виде свойства достижимости определенной точки в программе. Примером таких ошибок является нарушение assertion'ов, расставленных в коде программы, или некорректное использование библиотечных функций.

Два наиболее развитых на сегодняшний день тяжеловесных метода анализа программ – это ограничиваемая проверка моделей [24] и уточнение абстракции по контрпримерам [25].

Ограничиваемая проверка моделей

Метод ограничиваемой проверки моделей (от англ. Bounded Model Checking, BMC) изначально рассматривался как одна из наиболее удачных техник верификации применительно к разработке полупроводниковых устройств. Схемы, используемые в таких устройствах, часто вначале моделируют программой на языке высокого уровня, например, Си, а затем, после тщательного тестирования или верификации, реализуют на языке описания аппаратуры (например, Verilog [26]).

Алгоритмы ограничиваемой проверки моделей основаны на разворачивании циклов программы на конечное фиксированное число шагов k и последующей проверки возможности нарушения проверяемого свойства на глубине, не большей, чем эти k шагов. Принудительное ограничение сверху числа k является основным ограничением на применимость этого метода для формального доказательства корректности.

Значительные достижения в области производительности современных SAT- и SMT-решателей (SAT от англ. Satisfiability – задача проверки выполнимости для формул пропозициональной логики, SMT от англ. Satisfiability Modulo Theories – задача проверки выполнимости для логических формул с учетом лежащих в их основе теорий, например, линейной арифметики) позволили анализировать поведение программ методом ограничиваемой проверки моделей с очень высокой точностью через побитовое кодирование значений переменных и операций над ними. Метод ограничиваемой проверки моделей позволяет проверять для языка программирования широкий набор конструкций, включая динамически выделяемые структуры в памяти, адресную арифметику, операции с массивами и побитовые операции. На сегодняшний день данный метод является наилучшим для поиска неглубоких – в смысле длины пути выполнения – ошибок в небольших программных системах.

Инструменты, реализующие метод ограничиваемой проверки моделей, такие как CBMC [27] или F-Soft [28] с успехом используются для поиска ошибок в системном программном обеспечении, например, в автомобильной промышленности [29]. Есть также примеры использования этого метода для доказательства отсутствия состояний гонки в низкоуровневом программном обеспечении [30].

Уточнение абстракции по контрпримерам

Метод уточнения абстракции по контрпримерам (от англ. Counter-Example Guided Abstraction Refinement, CEGAR) применяется для проверки достижимости определенной точки в программе. Основная идея метода заключается в следующем. Детальная модель анализируемой программы (то есть наиболее точная модель, семантически эквивалентная самой программе) слишком большая и сложная, чтобы за разумное время можно было проверить свойства всех возможных путей в ней, поэтому достижимость заданной точки проверяется на упрощенной модели. Но это может привести к некорректному результату: анализ упрощенной модели может показать, что точка недостижима, хотя в действительности она достижима, или наоборот.

Проблемы первого вида метод уточнения абстракции по контрпримерам решает за счет того, что выбирается такой способ построения упрощённой модели, при котором такой ситуации не может произойти «по построению». То есть всегда, когда анализ упрощённой модели говорит, что точка

недостижима, то она является недостижимой и в детальной модели программы.

Для борьбы с проблемами второго вида применяется «уточнение абстракции по контрпримерам». Если анализ упрощённой модели показывает, что заданная точка является достижимой, то это сопровождается предъявлением пути в исходной программе, по которому можно достичь целевую точку. Этот путь называется *контрпримером*. Поскольку задача проверки реализуемости одного конкретного пути в детальной модели программы уже не столь сложна, как анализ всех путей и, более того, текущий уровень развития инструментов позволяет её решать достаточно эффективно, то именно это и делается для всех контрпримеров, обнаруживаемых при помощи анализа упрощённой модели. Если проверка подтверждает реализуемость найденного пути, то на вопрос о достижимости заданной точки получен корректный ответ. В противном случае, контрпример является следствием абстрагирования от существенных деталей при построении упрощённой модели. Тогда необходимо провести анализ контрпримера, чтобы выявить те детали, из-за игнорирования которых появился некорректный контрпример, и уточнить упрощённую модель, чтобы учесть в ней эти детали. Работа с новой упрощённой моделью повторяется по тому же самому алгоритму.

В результате ряда итераций метод уточнения абстракции по контрпримерам либо найдет путь в заданную точку программы, либо докажет её недостижимость, либо упрется в ограничения по ресурсам. Таким образом, метод не гарантирует нахождения решения в общем случае. Тем не менее, на практике он показывает достаточно хорошие результаты.

Метод уточнения абстракции по контрпримерам лежит в основе инструмента статической верификации SLAM [58], который входит в состав системы Static Driver Verifier (SDV) [31] и в настоящее время активно используется разработчиками драйверов устройств для операционной системы Microsoft Windows. Другие инструменты тяжеловесного статического анализа, такие как BLAST [32], SATabs [33] и CPAchecker [34], также работающие по методу уточнения абстракции, имеют широкую известность среди исследователей в области автоматической верификации. В рамках проекта Linux Driver Verification (LDV) [35, 36] разрабатывается система статической верификации драйверов операционной системы Linux, в которой на практике активно используются инструменты BLAST и CPAchecker. Результаты использования этих инструментов говорят о возможности их успешного применения для покомпонентного анализа больших объёмов промышленного кода.

2 План статьи

Данная статья описывает основные принципы метода уточнения абстракции по контрпримерам (CEGAR) и затрагивает основные моменты реализации CEGAR в инструментах статической верификации BLAST и CPAchecker.

В следующем (третьем) разделе статьи вводятся основные понятия и определения, необходимые для изложения сущности рассматриваемого метода. Затем рассматривается сам метод в том его варианте, в котором он реализован в инструментах BLAST и CPAchecker. Поскольку инструменты, работающие на основе CEGAR, устроены достаточно сложно, рассмотрение метода проводится в несколько этапов. Сначала в разделах 4, 5, 6 и 7 рассматривается наиболее простой вариант предикатной абстракции и её уточнения по методу CEGAR. Рассмотрение проводится на примере верификации очень простой программы на языке C с указанным для неё свойством недостижимости. Затем в разделе 8 рассмотренный вариант CEGAR с предикатной абстракцией расширяется для применения его к более сложным программам, в этом же разделе рассказывается об используемых модификациях и оптимизациях метода, направленных на увеличение его точности, масштабируемости и производительности. В конце (в разделе 9) описываются внешние инструменты, которые используются при реализации метода CEGAR в BLAST и CPAchecker (такие как SAT-, SMT- и интерполирующие решатели).

3 Основные понятия и определения

3.1 Простейшие программы

В этом разделе мы будем рассматривать *простейшие программы* на C-подобном языке, каждая из которых удовлетворяет следующим ограничениям:

- Простейшая программа состоит из одной главной функции `main`, первый оператор которой считается единственной точкой входа в программу. Функция `main` имеет объявление

  ```
  void main(void);
  ```

  Таким образом, эта функция не принимает никаких параметров и не возвращает никакого значения.
- В простейшей программе могут быть объявлены только локальные переменные функции `main` целочисленного типа `int`. Объявления переменных допускаются только до первого оператора функции `main`. Все переменные должны быть явно инициализированы при объявлении недетерминированными значениями, то есть произвольными, не заданными наперёд значениями типа `int`. Для этого используется специальный инициализатор `nondet`.

- В качестве операторов допускаются только следующие:
  - присваивание переменной выражения без побочных эффектов;
  - ветвление с использованием оператора `if` (`if-else`) по условию, заданному выражением без побочных эффектов;
  - безусловный переход на метку (оператор `goto`).

  Данное ограничение, в частности, означает, что пустые операторы в простейших программах недопустимы.
- Выражения без побочных эффектов состоят только из целочисленных констант, переменных и операций над ними.
- В качестве операций в выражениях без побочных эффектов допускаются только линейные арифметические операции `+`, `-`, `*`(умножение на константу), операции сравнения `>`, `<`, `==`, `!=`, `>=`, `<=` и логическая операция `!`.

Будем считать, что в рассматриваемой простейшей программе все операторы, а также точка выхода из программы помечены своими уникальными метками.

Для рассматриваемых простейших программ будем проверять свойство недостижимости некоторого ошибочного оператора, помеченного заданной *ошибочной меткой*. При этом рассматривать свойство недостижимости будем вне зависимости от условий завершения программы. *Завершением программы* называется достижение в процессе выполнения программы выхода из этой программы при отсутствии ошибок фазы выполнения. В случае простейших программ ошибки фазы выполнения всегда исключены, потому что все допустимые операторы и операции в выражениях таких программ определены всегда, в частности, отсутствует операция деления. Поэтому простейшая программа не может завершиться аварийно. Она может либо достигнуть выхода, либо зациклиться. Если простейшая программа зацикливается, никогда не достигая в ходе выполнения ошибочного оператора, свойство недостижимости ошибочной метки для такого выполнения простейшей программы считается выполненным.

Пример простейшей программы с ошибочной меткой `ERR` представлен на рис. 1а. Приведенная программа вычисляет модуль (абсолютное значение) разности двух произвольно заданных чисел – x и y и сохраняет его в переменной z. Меткой `ERR` помечен ошибочный оператор, который не должен достигаться в ходе корректного выполнения данной программы.

```
void main()
{
      int x = nondet;
      int y = nondet;
      int z = nondet;
L1:   if (x > y) {
L2:         z = x - y;
      } else {
L3:         z = y - x;
      }
L4:   if (z < 0)
ERR:  goto ERR;
L5:}
```

Рис. 1а. Пример простейшей программы.

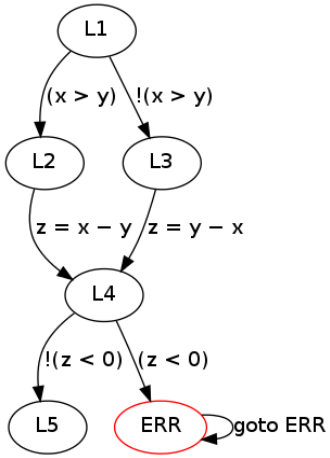

Рис. 1б. Граф потока управления для данной программы

3.2 Граф потока управления

Введем для простейшей программы понятие *графа потока управления* (*ГПУ*, от англ. Control flow graph, CFG). *Граф потока управления* – ориентированный граф, вершинам которого соответствуют уникальные метки, которыми помечен каждый оператор программы, а также точка выхода из неё, а направленным помеченным дугам соответствуют операторы программы. При этом каждому оператору присваивания или безусловного перехода однозначно соответствует единственная дуга в ГПУ, а оператору ветвления – две дуги: одна для указанного в операторе условия перехода и одна для отрицания этого условия. Вершина ГПУ, соответствующая метке первого оператора программы, называется его *начальной вершиной*. Для программы из

примера 1 на рис. 1а соответствующий граф потока управления представлен на рисунке 1б.

3.3 Состояния программы

Назовём *состоянием* простейшей программы пару (L, v), где L – уникальная метка следующего за ней (то есть помеченного этой меткой) выполняемого оператора программы или точки выхода, v – означивание всех переменных (то есть отображение имен всех переменных программы в их конкретные значения) непосредственно перед выполнением оператора, помеченного меткой L. Отметим, что состояние программы может как быть достижимо в ходе какого-либо её реального выполнения, так и не быть таковым.

Для программы в примере 1 состояниями считаются, например, такие пары:

$$(L1, \{x \mapsto 0, y \mapsto 0, z \mapsto 0\}),$$

$$(L2, \{x \mapsto 1, y \mapsto 0, z \mapsto 1\}),$$

$$(L5, \{x \mapsto 1, y \mapsto 0, z \mapsto 1\}),$$

$$(L5, \{x \mapsto 1, y \mapsto 0, z \mapsto -1\}).$$

Состояния программы (L, v), в которых L — имя метки первого оператора программы, назовём *начальными состояниями* этой программы. Примеры начальных состояний простейшей программы из примера 1:

$$(L1, \{x \mapsto 0, y \mapsto 0, z \mapsto 0\}),$$
$$(L1, \{x \mapsto 1, y \mapsto 0, z \mapsto 1\}),$$
$$(L1, \{x \mapsto 1, y \mapsto 2, z \mapsto -1\}).$$

Если ГПУ простейшей программы не содержит кратных дуг (формальное описание процесса построения ГПУ без кратных дуг приведено в секции 4.1), то любой упорядоченной паре состояний программы можно поставить в соответствие не более одной дуги ГПУ.

Например, в нашей программе упорядоченным парам состояний $((L1, \{x \mapsto 1, y \mapsto 0, z \mapsto 0\}), (L2, \{x \mapsto 1, y \mapsto 0, z \mapsto 0\}))$ и $((L1, \{x \mapsto 0, y \mapsto 0, z \mapsto 0\}), (L2, \{x \mapsto 1, y \mapsto 0, z \mapsto 1\}))$ в ГПУ соответствует одна и та же дуга $L1 \to L2$. А паре состояний $((L1, \{x \mapsto 0, y \mapsto 0, z \mapsto 0, L7, x \mapsto 1, y \mapsto 0, z \mapsto 1)$ не соответствует никакой дуги ГПУ.

Для упорядоченной пары состояний, которой соответствует некоторая дуга ГПУ, может быть выполнено отношение *непосредственной достижимости*. Смысл этого отношения в том, что второе состояние получается из первого в результате выполнения оператора, которым помечена соответствующая дуга ГПУ. В случае оператора ветвления по некоторому условию c договоримся считать, что одна из соответствующих ему дуг ГПУ помечена оператором перехода по условию c, а другая — оператором перехода по условию $\neg c$.

Будем обозначать непосредственную достижимость состояния (L_2, v_2) из состояния (L_1, v_1) по дуге ГПУ, помеченной оператором op, с помощью символа $\rightarrow$: $(L_1, v_1) \overset{op}{\rightarrow} (L_2, v_2)$. Формально отношение непосредственной достижимости для произвольной упорядоченной пары состояний $((L_1, v_1), (L_2, v_2))$ и произвольного оператора op можно определить следующим образом:

$$(L_1, v_1) \overset{op}{\rightarrow} (L_2, v_2) \Leftrightarrow \text{в ГПУ есть дуга } L_1 \overset{op}{\rightarrow} L_2 \text{ и}$$

$$\begin{cases} v_2 = v_1, & \text{если } op - \text{оператор } goto \\ v_2 = v_1\{x_k \mapsto e(v_1(x_1), \dots, v_1(x_n))\}, & \\ \text{если } op - \text{оператор присваивания } x_k = e(x_1, \dots, x_n) & \\ v_2 = v_1 \text{и } c(v_1(x_1), \dots, v_1(x_n)), & \\ \text{если } op - \text{оператор перехода по условию } c(x_1, \dots, x_n) & \end{cases}$$

Здесь $x_1, \dots, x_n$ — переменные простейшей программы; $v_1\{x_k \mapsto e(v_1(x_1), \dots, v_1(x_n))\}$ — означивание, которое получается из v_1 заменой значения переменной x_k на значение выражения $e(x_1, \dots, x_n)$ (из правой части оператора присваивания), вычисленное при значениях переменных, заданных означиванием v_1, $c(x_1, \dots, x_n)$ — условие, которым помечена одна из двух дуг ГПУ, соответствующих оператору ветвления.

Для нашей программы (на рис. 1) примеры состояний, связанных отношением непосредственной достижимости:

$$(L1, \{x \mapsto 1, y \mapsto 0, z \mapsto 0\}) \xrightarrow{(x>y)} (L2, \{x \mapsto 1, y \mapsto 0, z \mapsto 0\}),$$

$$(L2, \{x \mapsto 1, y \mapsto -1, z \mapsto -1\}) \xrightarrow{z = x - y} (L4, \{x \mapsto 1, y \mapsto -1, z \mapsto 2\}),$$

$$(ERR, \{x \mapsto 1, y \mapsto 0, z \mapsto -1\}) \xrightarrow{goto\ ERR} (ERR, \{x \mapsto 1, y \mapsto 0, z \mapsto -1\}).$$

3.4 Граф достижимости

Рассмотрим теперь *граф достижимости* — ориентированный граф, задающий отношение непосредственной достижимости на множестве всевозможных состояний простейшей программы. Вершинами этого графа являются состояния, а дуга от одной вершины к другой есть тогда и только тогда, когда соответствующая упорядоченная пара состояний связана отношением непосредственной достижимости. Дуги графа достижимости помечены соответствующими операторами.

Если в графе достижимости есть путь P из вершины (L_1, v_1) в вершину (L_n, v_n):

$(L_1, v_1) \overset{op_1}{\rightarrow} (L_2, v_2) \overset{op_2}{\rightarrow} \dots \overset{op_{n-1}}{\rightarrow} (L_n\ v_n)$, то говорят, что состояние (L_n, v_n) достижимо из состояния (L_1, v_1) по пути P. Будем также говорить о достижимости состояния (L_n, v_n), подразумевая его достижимость из какого-либо начального состояния.

В примере 1 достижимо, например, состояние $(L5, \{x \mapsto 1, y \mapsto 0, z \mapsto 1\})$, из начального состояния $(L1, \{x \mapsto 1, y \mapsto 0, z \mapsto 0\})$ по пути

$$(L1, \{x \mapsto 1, y \mapsto 0, z \mapsto 0\}) \xrightarrow{(x>y)} (L2, \{x \mapsto 1, y \mapsto 0, z \mapsto 0\}) \xrightarrow{z = x - y}$$
$$(L4, \{x \mapsto 1, y \mapsto 0, z \mapsto 1\}) \xrightarrow{!(z<0)} (L5, \{x \mapsto 1, y \mapsto 0, z \mapsto 1\})$$

3.5 Задача инструмента верификации

Недостижимость, то есть отсутствие достижимости, всех состояний с ошибочной меткой (состояний (ERR, v), где ERR — ошибочная метка) соответствует для простейшей программы недостижимости самой ошибочной метки (в ходе любого выполнения программы). Состояния с ошибочной меткой будем также называть *ошибочными состояниями.* Формально можно сказать, что задача инструмента верификации – проверить для заданной простейшей программы достижимость какого-либо ошибочного состояния, то есть существование в графе достижимости пути из какого-либо начального состояния в какое-либо состояние с ошибочной меткой. В случае достижимости ошибочного состояния найденный инструментом путь из начального состояния в ошибочное будем называть *примером ошибочного пути*. Итак, задача инструмента верификации – либо доказать для данной на вход простейшей программы недостижимость ошибочных состояний, либо привести соответствующий пример ошибочного пути. В программе из примера 1 ошибочные состояния с меткой ERR (и соответственно сама ошибочная метка ERR) являются недостижимыми.

Оценим некоторым образом ту работу, которую необходимо было бы проделать инструменту статической верификации для проверки условия недостижимости, если бы эта проверка осуществлялась, к примеру, непосредственным поиском в графе достижимости. По крайней мере, инструменту потребовалось бы проверять достижимость из каждого возможного начального состояния программы. Если в нашем простейшем примере считать все переменные программы 32-битными, то инструменту верификации пришлось бы перебрать

$$(2^{32})^3 = 2^{3 \cdot 32} = 2^{96} > 7{,}9 \cdot 10^{28} > (2{,}5 \cdot 10^{12}) \cdot 365 \cdot 24 \cdot 60^2 \cdot 10^9$$

возможных начальных состояний. Такое количество состояний уже неприемлемо велико для современных вычислительных машин. Если бы в секунду удавалось проверять достижимость из миллиарда начальных состояний, то на верификацию простейшей программы из примера 1 ушло бы больше 2 триллионов лет.

Проблема резкого («взрывного») увеличения числа состояний программы с ростом числа её переменных и операторов получила название проблемы *комбинаторного взрыва*. Для её решения применяются различные подходы. Рассмотрим один из способов борьбы с проблемой комбинаторного взрыва, который называется предикатной абстракцией.

3.6 Предикатная абстракция и метод CEGAR

При использовании абстракции вместо состояний исходной программы рассматриваются *абстрактные состояния*, которые представляют собой множества состояний программы. Эти множества могут пересекаться или быть вложены одно в другое. На основе некоторых заданных вначале абстрактных состояний строится абстрактное дерево достижимости (АДД, определение АДД вводится в разделе 3.9), состоящее из абстрактных состояний и переходов между ними по операторам программы. Строится оно таким образом, что если существует переход между какими-либо двумя состояниями программы, то существует и переход между соответствующими им абстрактными состояниями, которые включают эти состояния. По построению получается, что АДД обязательно включает все достижимые в исходной программе состояния (как элементы соответствующих абстрактных состояний), а также, возможно, и некоторые другие. Поэтому пути, выполнение по которым возможно в исходной программе, обязательно оказываются представленными в АДД в виде последовательности переходов между абстрактными состояниями. Обратное, вообще говоря, неверно. Из-за использования абстрактных состояний вместо состояний исходной программы в АДД могут быть представлены и фиктивные пути, выполнение по которым на самом деле невозможно.

Если в АДД нет *ошибочных абстрактных состояний* (то есть абстрактных состояний, включающих ошибочные состояния программы), то ошибочные состояния недостижимы и в исходной программе. Если же в АДД находится ошибочное абстрактное состояние, то это не обязательно говорит о достижимости ошибочного состояния в исходной программе, так как выбранные абстрактные состояния могут быть слишком неточными, для того чтобы показать недостижимость ошибочного состояния.

Для описания абстрактных состояний будем использовать частный случай абстракций — *предикатные абстракции* — то есть абстракции, в которых абстрактные состояния представляются логическими выражениями (предикатами) над переменными программы. В таких абстракциях состояния программы объединяются во множества (абстрактные состояния) по признаку истинности в них некоторых логических выражений (например, состояния, где $x > 0$, состояния, где $x = y$ или состояния, где $x > 0 \wedge y = 0$).

Для построения переходов между вершинами АДД при использовании предикатных абстракций операторы программы кодируются в виде логических формул. Для проверки истинности того или иного предиката после

выполнения оператора также составляется логическая формула, задающая множество возможных состояний программы после выполнения оператора. Затем проверяется логическое следование (импликация) из этой формулы каждого из заданных логических выражений. Это соответствует проверке вложенности множества возможных состояний программы после выполнения оператора во множество состояний, в которых истинно заданное логическое выражение. Пересечение таких множеств, соответствующее конъюнкции логических выражений, выбирается в качестве следующего абстрактного состояния после выполнения оператора.

Если в построенном АДД существует путь, ведущий в одно из ошибочных абстрактных состояний, то этот путь называют контрпримером. Его наличие может означать как реальную ошибку, так и неточность выбранной предикатной абстракции.

Для установления истинности ошибки по контрпримеру составляется формула сильнейшего постусловия пути (определяется в разделе 6.3), описывающая существование аналогичного пути в графе достижимости по состояниям программы. Если формула выполнима, то программа содержит ошибку. Иначе данная формула используется для уточнения предикатной абстракции, то есть для пополнения набора выбранных предикатов. После этого АДД перестраивается заново. АДД, которое строится после добавления в набор новых предикатов, не должно более включать тот же самый ложный ошибочный путь.

Цикл повторных построений АДД повторяется, пока мы не докажем корректность программы (отсутствие ошибочных абстрактных состояний в АДД) или не найдем реальную ошибку.

Это общая схема работы метода CEGAR – уточнения абстракции (в данном случае – абстрактного дерева достижимости) по контрпримерам. Начнем рассмотрение его реализации в инструментах BLAST и CPAchecker с более подробного разбора понятия абстрактного состояния и предикатной абстракции (точнее, ее частного случая – декартовой предикатной абстракции) на примере.

3.7 Абстрактные состояния и декартова предикатная абстракция

3.7.1 Предикатная абстракция на примере

Предикатная абстракция основана на введении на множестве всевозможных означиваний всех переменных программы некоторой системы подмножеств, упрощенно представляющих основные и наиболее существенные свойства входящих в эти подмножества означиваний. Для примера возьмём простейшую программу с одной переменной (рис. 2).

```
void main() {
      int x = nondet;
L1:   if (x >= 5)
L2:         if (x <= 10) {
L3:               x = x + 1;
L4:               if (!(x >= 5))
ERROR:                  goto ERROR;
            }
L5:}
```

Рис. 2. Простейшая программа с одной переменной x.

На рис. 3 схематически изображено множество всех возможных значений (фактически, означиваний) единственной переменной x этой программы.

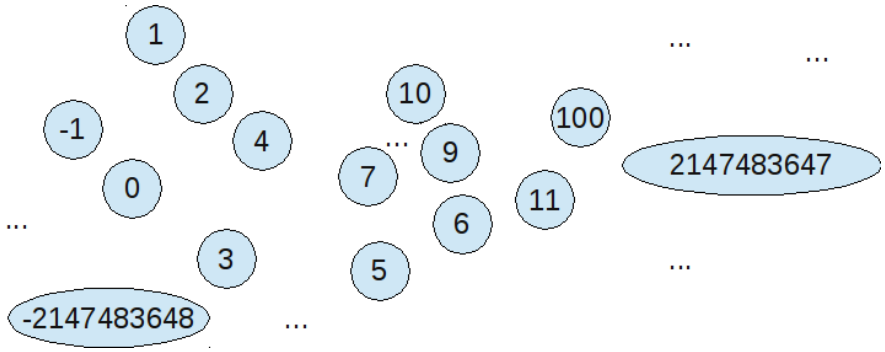

Рис. 3. Множество всех возможных значений переменной x.

Будем считать далее, что переменная x может принимать любые целочисленные значения, в том числе вне диапазона типа `int`. Таким образом, схематически изображенные на рисунке значения -2147483648 и 2147483647 являются просто одними из возможных значений этой переменной, а не наименьшим и наибольшим возможным её значением соответственно. В тексте программы встречаются условия $x \geq 5$ и $x \leq 10$, поэтому выберем их для примера. Рассмотрим систему подмножеств,

представляющую свойства $x \geq 5$ и $x \leq 10$ для входящих в эти подмножества означиваний.

На рис. 4 схематически изображена такая система подмножеств.

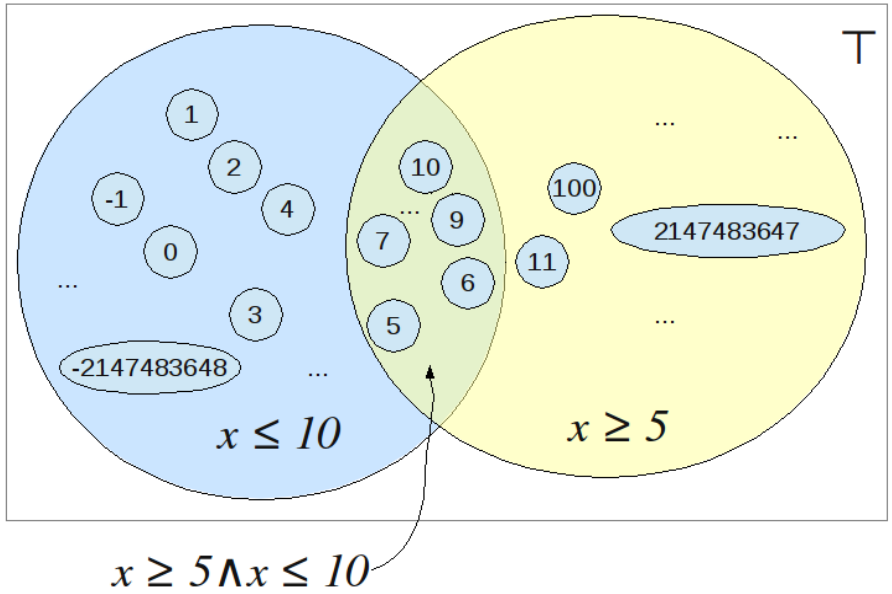

Рис. 4. Система подмножеств множества значений переменной x.

На рисунке выделены два подмножества таких означиваний переменной x, в которых выполнены условия $x \geq 5$ и $x \leq 10$ соответственно. Эти условия – *предикаты* над переменными (в данном случае – единственной переменной) простейшей программы. Показано также пересечение этих подмножеств, которому соответствует предикат $x \geq 5 \wedge x \leq 10$, и множество всех означиваний $\top$ («топ»), которому соответствует тождественная истина. Такие системы подмножеств называют *предикатными абстракциями* над множеством означиваний переменных программы.

В данной статье будем рассматривать только предикатные абстракции, в которые входят:

- подмножества означиваний, элементы которых удовлетворяют некоторому заранее заданному предикату (например, $x \geq 5$),
- пересечения этих подмножеств (соответствующие конъюнкциям предикатов, например, $x \geq 5 \wedge x \leq 10$),
- подмножество всех означиваний $\top$ (соответствующее тождественной истине) и

- пустое подмножество означиваний $\bot$ («ботом», «дно», соответствующее тождественной лжи).

В эти абстракции *не* входят объединения, разности, дополнения и результаты каких-либо других теоретико-множественных операций над подмножествами означиваний и соответственно не рассматриваются дизъюнкции и отрицания заданных предикатов. Такие предикатные абстракции называют *декартовыми*. Подмножества означиваний всех переменных программы, пересечения этих подмножеств, а также специальные подмножества $\top$ и $\bot$ называют *абстрактными состояниями данных*, а соответствующие им предикаты и их конъюнкции (в том числе тождественную истину и особый случай – тождественную ложь) – *регионами абстрактных состояний*. В абстрактное состояние программы также входит состояние её потока управления (имя уникальной метки следующего выполняемого оператора), поэтому *абстрактным состоянием программы* (или просто *абстрактным состоянием*) будем называть пару (L, r), где L – имя уникальной метки, задающее состояние потока управления программы, r – регион абстрактного состояния (конъюнкция предикатов или тождественная ложь, соответствующая абстрактному состоянию данных).

Рассмотрим теперь один из операторов нашей простейшей программы с одной переменной. Возьмём оператор $x = x + 1$, помеченный меткой $L3$. В данном разделе будем изображать на одном рисунке означивания переменной x в двух последовательных состояниях программы, например, на метке $L3$ (непосредственно перед выполнением оператора $x = x + 1$) и на метке $L4$ (перед проверкой условия $!(x \geq 5)$). Переходы между значениями переменной x при выполнении перехода из одного состояния в другое (например, из состояния с меткой $L3$ в состояние с меткой $L4$) будем изображать стрелками. На рис. 5 показаны переходы между означиваниями переменной x при выполнении выбранного нами оператора $x = x + 1$.

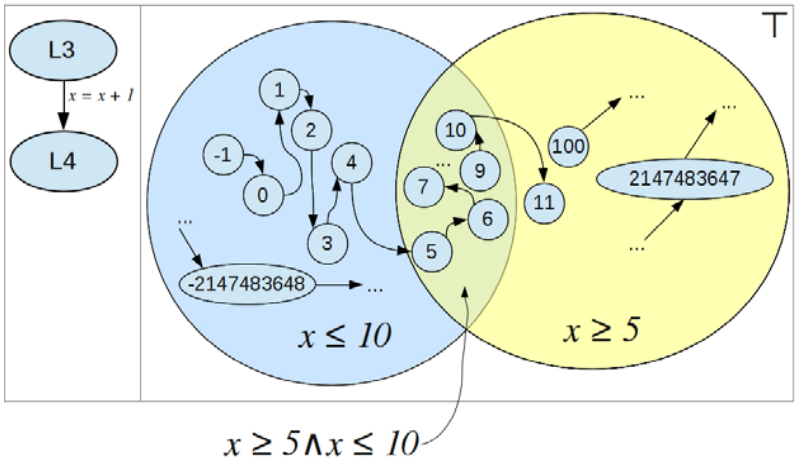

Рис. 5. Переходы между означиваниями переменной x при выполнении оператора $x = x + 1$.

На рисунке показано, что множество значений переменной рассматривается как неограниченное. Предположим, что при некотором состоянии потока управления (в нашем случае – перед выполнением оператора $x = x + 1$, помеченного меткой $L3$) все возможные в этом месте программы означивания переменной x входят в абстрактное состояние с регионом $x \geq 5 \wedge x \leq 10$. Рассмотрим теперь переходы по оператору $x = x + 1$ только из означиваний, входящих в это абстрактное состояние. Эти переходы показаны на рис. 6.

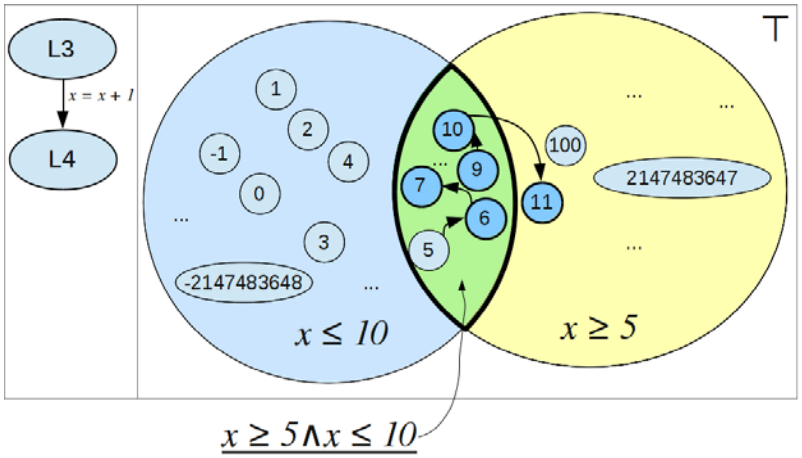

Рис. 6. Переходы по оператору $x = x + 1$ из абстрактного состояния с регионом $x \geq 5 \wedge x \leq 10$.

На рисунке выделены возможные значения переменной x после выполнения оператора $x = x + 1$ из абстрактного состояния с регионом $x \geq 5 \wedge x \leq 10$. Это подмножество выделенных возможных значений переменной x можно покрыть подмножеством $x \geq 5$, так как все выделенные значения принадлежат этому подмножеству. Таким образом, можно сказать, что после выполнения оператора $x = x + 1$ из абстрактного состояния $(L3,\ x \geq 5 \wedge x \leq 10)$, все возможные означивания переменной x можно ограничить (покрыть) абстрактным состоянием с регионом $x \geq 5$. В этом случае говорят, что переход из абстрактного состояния $(L3,\ x \geq 5 \wedge x \leq 10)$ по оператору $x = x + 1$ ведет в абстрактное состояние $(L4,\ x \geq 5)$. Вместо абстрактного состояния $x \geq 5$, вообще говоря, можно рассматривать абстрактное состояние $\top$, которое включает его. Но при использовании описанного далее (в следующем подразделе) метода построения переходов между абстрактными состояниями в данном случае будет получаться именно $x \geq 5$. Построенный переход изображен на рис. 7.

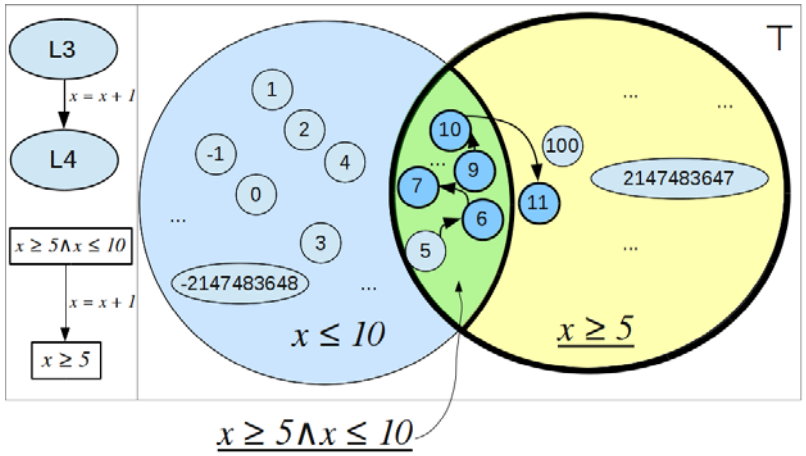

Рис. 7. Переход из абстрактного состояния с регионом $x \geq 5 \wedge x \leq 10$ по оператору $x = x + 1$ в абстрактное состояние с регионом $x \geq 5$.

3.7.2 Построение переходов между абстрактными состояниями с помощью решателей на примере

Для построения переходов между абстрактными состояниями на практике заметим, что отношение вложенности подмножеств означиваний соответствует отношению логического следования (импликации) между соответствующими логическими предикатами (в широком смысле). Множество означиваний A вложено в подмножество означиваний B тогда и

только тогда, когда для соответствующих предикатов a и b выполнено отношение $a \to b$ (рис. 8).

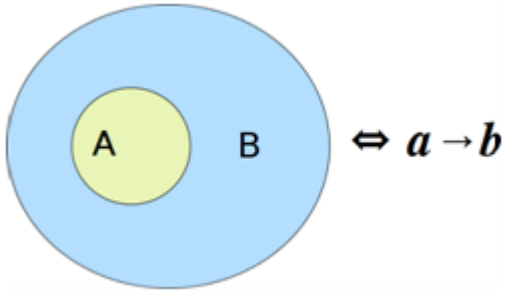

Рис. 8. Иллюстрация к соответствию отношений $A \subseteq B$ и $a \to b$.

Это означает, что абстрактное состояние данных A вложено в абстрактное состояние данных B тогда и только тогда, когда для соответствующих им регионов a и b выполнено логическое отношение $a \to b$.

Воспользуемся этим свойством для проверки вложенности подмножества всевозможных означиваний переменной x из предыдущего примера (после выполнения оператора $x = x + 1$ из абстрактного состояния $(L3,\ x \geq 5 \wedge x \leq 10)$) в различные абстрактные состояния данных ($x \geq 5$ и $x \leq 10$). В предположении неограниченности множества значений переменной x подмножеству всевозможных её означиваний можно поставить в соответствие предикат (в широком смысле) $x_1 \geq 5 \wedge x_1 \leq 10 \wedge x_2 = x_1 + 1$. Это предикат от переменных x_1 и x_2, соответствующих означиваниям переменной x до и после выполнения оператора $x = x + 1$ соответственно. Вложенность подмножества означиваний x после выполнения оператора $x = x + 1$ в подмножество $x \geq 5$ можно проверить, проверив импликацию $(x_1 \geq 5 \wedge x_1 \leq 10 \wedge x_2 = x_1 + 1) \to x_2 \geq 5$. Эта импликация является тождеством тогда и только тогда, когда подмножество всех возможных означиваний переменной x после выполнения перехода вложено в абстрактное состояние данных $x \geq 5$. Отношение является тождеством тогда и только тогда, когда невыполнимо его отрицание (в силу полноты и непротиворечивости логики первого порядка). Поэтому для проверки вложенности подмножества возможных означиваний x в абстрактное состояние данных $x \geq 5$ можно проверить выполнимость формулы

$$\neg((x_1 \geq 5 \wedge x_1 \leq 10 \wedge x_2 = x_1 + 1) \to x_2 \geq 5) =$$

$$\neg(\neg(x_1 \geq 5 \wedge x_1 \leq 10 \wedge x_2 = x_1 + 1) \vee x_2 \geq 5) =$$

$$(x_1 \geq 5 \wedge x1 \leq 10 \wedge x_2 = x_1 + 1) \wedge \neg(x_2 \geq 5)$$

Аналогично можно проверить вложенность в абстрактное состояние данных $x \leq 10$. Для проверки выполнимости таких логических формул – нулевого порядка, то есть без кванторов, с равенством и линейной целочисленной арифметикой – можно использовать специальные инструменты, называемые SMT-решателями.

Когда подмножество одновременно вложено в два или более абстрактных состояния данных, оно является вложенным в пересечение этих абстрактных состояний данных. Регион, соответствующий пересечению абстрактных состояний данных, является конъюнкцией регионов соответствующих абстрактных состояний.

При этом следует учитывать, что абстрактное состояние данных $\perp$ является вложенным в любое абстрактное состояние данных. В зависимости от заданных предикатов, абстрактное состояние данных $\perp$ может совпадать с пересечением двух или более (в том числе, всех) абстрактных состояний данных (например, $x \geq 5 \wedge x < 5 \equiv \perp$), а может быть и строго вложено в пересечение всех абстрактных состояний данных (как в данном примере). В то же время абстрактное состояние данных $\perp$ имеет большое значение, поскольку соответствует пустому множеству означиваний и, как следствие, пустому множеству состояний программы и неосуществимости перехода по некоторому оператору из какого-либо абстрактного состояния. В дальнейшем неосуществимость такого перехода будет использована для доказательства недостижимости ошибочной метки. Поэтому, чтобы отличать состояние $\perp$ от пересечения всех абстрактных состояний данных (в общем случае они не совпадают), вначале проверяют выполнимость предиката (в широком смысле), соответствующего множеству всевозможных означиваний переменных после выполнения перехода по оператору (в частности, по оператору проверки условия).

В нашем примере предикат $x_1 \geq 5 \wedge x_1 \leq 10 \wedge x_2 = x_1 + 1$ как логическая формула является выполнимым. Новое абстрактное состояние данных после перехода не равно $\perp$. В качестве нового абстрактного состояния данных берем конъюнкцию всех абстрактных состояний данных, в которые вложено подмножество всевозможных означиваний переменной x после перехода. Формула $(x_1 \geq 5 \wedge x_1 \leq 10 \wedge x_2 = x_1 + 1) \wedge \neg(x_2 \geq 5)$ невыполнима, а формула $(x_1 \geq 5 \wedge x_1 \leq 10 \wedge x_2 = x_1 + 1) \wedge \neg(x_2 \leq 10)$ выполнима. В качестве результата берем конъюнкцию из одного абстрактного состояния данных $x \geq 5$.

Так как в примере оператор $x = x + 1$ помечен меткой $L3$, а следующий за ним оператор – меткой $L4$, то графически построенный переход можно изобразить так, как показано на рис. 9а.

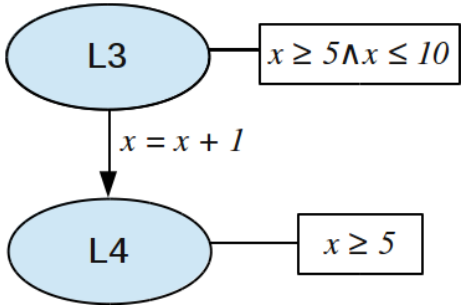

Рис. 9а. Переход из абстрактного состояния $(L3, x \leq 5 \wedge x \geq 10)$ *по оператору* $x = x + 1$ *в абстрактное состояние* $(L4, x \geq 5)$.

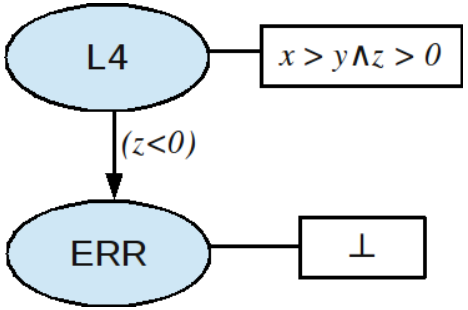

Рис. 9б. Переход из абстрактного состояния $(L4, x > y \wedge z > 0)$ *в абстрактное состояние* $(ERR, \perp)$.

Приведем еще один пример. Для программы из примера 1 построим переход из абстрактного состояния $(L4, x > y \wedge z > 0)$ по оператору ветвления с условием $(z < 0)$. Предикат на возможные означивания переменных после перехода: $x > y \wedge z > 0 \wedge z < 0$ – не является выполнимым как логическая формула. Следовательно, новое абстрактное состояние равно $(ERR, \perp)$. Построенный переход показан на рисунке 9б.

3.7.3 Общий случай декартовой предикатной абстракции

Итак, инструменты статической верификации, использующие абстракцию, рассматривают вместо каждого отдельного состояния сразу подмножества состояний исходной программы. Каждое такое подмножество может включать

в себя очень большое число состояний, например, 2^{96} или даже намного больше. В подмножество объединяют некоторые состояния, имеющие одинаковые метки. Мы рассматриваем случай, когда в подмножество объединяются состояния, в которых переменные программы удовлетворяют некоторому набору логических предикатов нулевого порядка (без кванторов).

Пусть задан некоторый предикат $p(x_1, \dots, x_n)$ над переменными программы $x_1, \dots, x_n$ (предикат не обязательно существенно зависит от всех переменных программы). В каждом состоянии программы (L, v) означивание v задаёт значения всех переменных $x_1, \dots, x_n$. Подставим эти значения вместо соответствующих переменных в предикат $p(x_1, \dots, x_n)$. Получим $p\big(v(x_1), \dots, v(x_n)\big)$ —*значение предиката p в состоянии* (L, v). Это значение всегда можно вычислить, оно будет являться логической (булевой) константой и будет равно либо истине, которую мы обозначим символом $\top$, либо лжи, которую мы обозначим символом $\bot$. В случае $p\big(v(x_1), \dots, v(x_n)\big) = \top$ будем писать $v \vdash p(x_1, \dots, x_n)$ или просто $v \vdash p$.

Пусть теперь дана некоторая фиксированная уникальная метка L какого-либо оператора исходной простейшей программы. Для данного предиката p можно рассмотреть подмножество всех возможных состояний программы с заданной меткой L, в которых значение предиката p равно тождественной истине. Обозначим это подмножество через $[\![p]\!]_L$: $[\![p]\!]_L \stackrel{\text{def}}{=} \{(L, v) : v \vdash p\}, L$ фиксировано.

Одним из возможных способов задания и представления набора множеств состояний исходной программы (для последующей работы с полученным набором представлений этих множеств) является *декартова предикатная абстракция*. Пусть задан некоторый конечный набор предикатов $\{p_1, \dots, p_n\}$. Рассмотрим некоторое подмножество предикатов из этого набора: $\{p_{i_1}, \dots, p_{i_m}\} \subseteq \{p_1, \dots, p_n\}$. В декартовой предикатной абстракции подмножеству предикатов $\{p_{i_1}, \dots, p_{i_m}\} \subseteq \{p_1, \dots, p_n\}$ ставят в соответствие конъюнкцию этих предикатов

$$p_{i_1} \wedge \dots \wedge p_{i_m} = \bigwedge_{k=1}^{m} p_{i_k},$$

которую, в свою очередь, рассматривают как новый предикат, и ставят в соответствие этому предикату подмножество состояний исходной программы с фиксированной меткой L:

$$\left[\!\!\left[\bigwedge_{k=1}^{m} p_{i_k} \right]\!\!\right]_L = [\![p_{i_1} \wedge \dots \wedge p_{i_m}]\!]_L$$

$$= \{(L, v) : v \vdash p_{i_1} \wedge \dots \wedge p_{i_m}\}, L \text{ фиксировано.}$$

Формально в декартовой предикатной абстракции *абстрактным состоянием* называют пару (L, r), где L — уникальная метка, а r — либо конъюнкция подмножества предикатов из заранее заданного набора, $r \equiv p_{i_1} \wedge \dots \wedge p_{i_m}, \{p_{i_1}, \dots, p_{i_m}\} \subseteq \{p_1, \dots, p_n\}, 0 \le m \le n$, либо тождественная ложь, $r \equiv \bot$. В частности, если $m = 0$, то $r \equiv \top$ и $v \vdash r$ для любого v; это означает что абстрактное состояние $(L, \top)$ задаёт все возможные состояния программы с заданной меткой L. Если же $r \equiv \bot$, то $v \vdash r$ не верно ни для какого означивания v, то есть абстрактное состояние $(L, \bot)$ задаёт пустое множество состояний программы, которому формально приписана некоторая уникальная метка L. Обозначим множество состояний программы, заданных абстрактным состоянием (L, r) через $[\![(L, r)]\!]$. Тогда формально можно записать:

$$[\![(L, r)]\!] \stackrel{\text{def}}{=} [\![r]\!]_L = \{(L, v) : v \vdash r\}, L \text{ фиксировано.}$$

В абстрактном состоянии (L, r) r является предикатом, который называют *регионом абстрактного состояния*. Для состояния программы (L, v) такого, что $v \vdash r$или, что то же самое, $(L, v) \in [\![(L, r)]\!]$ будем говорить, что состояние программы (L, v) входит в абстрактное состояние (L, r).

В качестве пояснения приведём примеры возможных регионов для заданного набора из двух предикатов:

$$p_1 \equiv x > 0$$

$$p_2 \equiv x = 0$$

Возможные регионы:

$$r_1 \equiv \top$$

$$r_2 \equiv p_1 \equiv (x > 0)$$

$$r_3 \equiv p_2 \equiv (x = 0)$$

$$r_4 \equiv p_1 \wedge p_2 \equiv (x > 0) \wedge (x = 0) \equiv \bot$$

$$r_5 \equiv \bot \equiv r_4$$

Здесь, к примеру, абстрактное состояние $(L, r_2) = (L, p_1) = (L, x > 0)$ (x — целое) задаёт множество конкретных состояний с меткой L, в которых

переменная x принимает строго положительные значения ($x = 1, 2, ...$), а остальные переменные принимают произвольные значения.

3.8 Сильнейшие постусловия и вычисление регионов

В инструментах статической верификации на основе предикатной абстракции, в том числе в BLAST и CPAchecker, логические формулы являются основным средством представления абстрактных состояний и работы с ними. Теперь для дальнейшего изложения нам требуется определить некоторую аналогию непосредственной достижимости для абстрактных состояний.

Пусть дано некоторое абстрактное состояние (L, r) и дуга в ГПУ с меткой op. Будем формально говорить для заданной метки L и предиката от переменных программы $p(x_1, \dots, x_n)$, что все состояния программы из множества $[\![p]\!]_L = \{(L, v): v \vdash p\}$ заданы меткой L и предикатом p. *Сильнейшим постусловием оператора* op при данном *предусловии* p называется предикат от переменных программы, задающий вместе с некоторой меткой L' те и только те состояния программы, которые непосредственно достижимы из состояний, заданных предусловием (предикатом) p и некоторой меткой L, по дуге, помеченной оператором op. Если задать дугу $L \xrightarrow{op} L'$ в ГПУ, и некоторое предусловие $p(x_1, \dots, x_n)$, то можно сказать, что сильнейшее постусловие p' оператора op при заданном предусловии p задаёт вместе с меткой L' множество $[\![p']\!]_{L'} = \{(L', v'): \exists (L, v),$ такое, что $v \vdash p$ и $(L, v) \xrightarrow{op} (L', v')\}$. Предикат сильнейшего постусловия можно задавать в виде, зависящем только от предусловия p и оператора op. Поэтому сильнейшее постусловие оператора op при данном предусловии p мы будем обозначать через $SP(p, op)$ (то есть $p' = SP(p, op)$).

Регион является предикатом от переменных программы и может быть использован в качестве предусловия. Рассмотрим сильнейшее постусловие $SP(r, op)$ для региона r абстрактного состояния (L, r) и оператора op на дуге $L \xrightarrow{op} L'$ ГПУ. Использование декартовой предикатной абстракции даёт возможность достаточно просто подобрать регион, являющийся следствием сильнейшего постусловия $SP(r, op)$ и таким образом задать новое абстрактное состояние (L', r'), в которое входят все конкретные состояния, достижимые из данного абстрактного состояния (L, r) по дуге с пометкой op. В это абстрактное состояние могут также войти и другие конкретные состояния, в которые не входит дуга с пометкой op, выходящая из (L, r). Поэтому (L', r') называют *верхним приближением (*от англ. *over-approximation)* множества конкретных состояний, достижимых из (L, r) по дуге с меткой op. По сути, верхнее приближение для абстрактных состояний означает то же, что и надмножество.

Введём обозначение выполнимости логической формулы: будем писать $\varphi - SAT$, если формула φ выполнима[1] и $\varphi - UNSAT$, если φ — невыполнима.

Для того чтобы подобрать регион, являющийся следствием сильнейшего постусловия $SP(r, op)$, воспользуемся свойством для произвольных предикатов $P, p_1, \dots, p_m$:
$P \to p_1 \wedge \dots \wedge p_m \Leftrightarrow \neg P \vee (p_1 \wedge \dots \wedge p_m) \Leftrightarrow (\neg P \vee p_1) \wedge \dots \wedge (\neg P \vee p_m)$.
Последняя конъюнкция выполнена тогда и только тогда, когда $P \wedge \neg p_1 - UNSAT, \dots, P \wedge \neg p_m - UNSAT$. Поэтому если выбрать из заданного набора предикатов $\{p_1, \dots, p_n\}$ те и только те предикаты $p_{i_k}, 1 \le k \le m$, для которых $P \wedge \neg p_{i_k} - UNSAT$, то для региона $\bigwedge_{k=1}^{m} p_{i_k}$ будет выполнено $P \to \bigwedge_{k=1}^{m} p_{i_k}$.

Исходя из этих соображений будем пользоваться для вычисления региона r' абстрактного состояния (L', r') следующим правилом:

- В декартовой предикатной абстракции для множества предикатов $\{p_1, \dots, p_n\}$ регион r' абстрактного состояния (L', r'), которое является верхним приближением множества конкретных состояний, заданных сильнейшим постусловием $SP(r, op)$ оператора op для предусловия r, равен:
  - $\bot$, если формула $SP(r, op)$ невыполнима, т.е. $SP(r, op) - UNSAT$;
  - конъюнкции предикатов $p_{i_1}, \dots, p_{i_m}$, для каждого из которых формула $SP(r, op) \wedge \neg p_{i_k}$ невыполнима, т.е.

$$\bigwedge_{\substack{p_{i_k} \in \{p_1, \dots, p_n\}: \\ SP(r, op) \wedge \neg p_{i_k} - UNSAT}} p_{i_k},$$

если $SP(r, op) - SAT$.

Пустая дизъюнкция считается соответствующей тождественной истине ($\top$).

Формальное доказательство того, что использование этого правила всегда даёт верхнее приближение множества конкретных состояний, достижимых из (L, r) по оператору op, можно найти в статье [37], в которой оно и было впервые

[1] Логическая формула (в логике первого порядка) на основе теорий (SMT-формула) называется выполнимой, если можно назначить всем входящим в неё неинтерпретируемым символам, то есть символам (константам, функциям, предикатам), не являющимся пропозиционными связками (такими как $\top, \wedge, \neg$) или символами теорий (такими как $2, +, \le$), значения из соответствующих доменов (областей определения) так, чтобы формула стала истинной.

предложено. Регион, соответствующий абстрактному состоянию (L', r'), вычисленный по приведённому правилу для множества предикатов $\pi = \{p_1, \dots, p_n\}$ и предусловия r, будем в дальнейшем обозначать через $\mathrm{post}_\pi(r, op)$.

3.9 Абстрактное дерево достижимости

Дадим теперь, наконец, определение основной структуры данных, используемой инструментами BLAST и CPAchecker для представления верхнего приближения множества всех достижимых состояний исходной программы, то есть для приближенного решения поставленной перед ними задачи проверки свойства недостижимости. *Абстрактное дерево достижимости* или *АДД* (от англ. *Abstract Reachability Tree, ART)* – это ориентированное дерево с помеченными дугами и вершинами, представляющее верхнее приближение некоторой части состояний исходной программы, достижимых из её начального состояния. Каждая вершина АДД помечена абстрактным состоянием (L, r), представленным в виде пары (метка, регион). Будем обозначать помеченные вершины АДД так: $N\colon (L, r)$ или просто N, где N — уникальный идентификатор вершины. Договоримся использовать в качестве уникальных идентификаторов вершин АДД имена меток соответствующих абстрактных состояний, приписывая к ним разделитель # и дополнительный индекс для обеспечения уникальности. Например, для вершины с меткой $(L6, x > y \wedge z \geq 0)$ будем использовать идентификатор $L6\#1$, а для вершины с меткой $(L6, \top)$ — идентификатор $L6\#2$. Дуги АДД помечаются теми же операторами, что и соответствующие дуги ГПУ. Соответствие это устанавливается во время построения АДД уже рассмотренным ранее способом по именам меток в вершинах АДД. Путь от корня до произвольной вершины в АДД может соответствовать одному из представленных в графе достижимости путей P возможного выполнения программы до достижения соответствующей метки. Регион же в вершине АДД представляет (с точностью до метки) верхнее приближение множества состояний программы, достижимых при условии её выполнения по этому пути P.

Будем называть АДД *полным*, если:

1. Его корень помечен $(L_1, \top)$, где L_1 — метка первого оператора программы, или, иначе, метка начального состояния программы.
2. АДД *замкнуто относительно постусловий*, то есть для любой его внутренней вершины $N\colon (L, r)$, такой, что $r - SAT$ $(r \not\equiv \bot)$, и для любой дуги ГПУ, исходящей из его вершины с меткой L и помеченной оператором op — $L \xrightarrow{op} L'$ — из вершины N АДД исходит дуга с меткой op в его вершину $N'\colon (L', r')$, такую что $\mathrm{post}_\pi(r, op) \to r'$, где π — выбранное множество предикатов.
3. Для любого листа АДД $N\colon (L, r)$ верно что:
   a. либо у вершины с меткой L в ГПУ нет исходящих дуг,
   b. либо $\varphi \equiv \bot$ (то же, что и $\varphi - UNSAT$),
   c. либо в АДД существует внутренняя вершина $N'\colon (L, r')$, такая, что $r \to r'$.

В случае 3c говорят, что вершина $N\colon (L, r)$ *покрыта* вершиной $N'\colon (L, r')$, потому что выполнена вложенность соответствующих множеств состояний программы
($[\![(L, r)]\!] \subseteq [\![(L, r')]\!]$) и, как следствие, любое выполнение программы из вершины N с точностью до выбранной абстракции возможно также из вершины N'.

Полное АДД приближает сверху множество достижимых состояний программы. Интуитивно АДД представляет собой конечную развёртку ГПУ, вершины которого помечены регионами.

Полное АДД называют *безопасным по отношению к конфигурации* (E, p), где E — ошибочная метка, p — предикат над переменными программы, если для любой его вершины $N\colon (E, r)$ конъюнкция $r \wedge p$ невыполнима $(r \wedge p - UNSAT)$.

Полное АДД, безопасное по отношению к конфигурации $(E, \top)$, является доказательством недостижимости в исходной программе ошибочного оператора с меткой E. Это частный случай применения следующей теоремы:

Теорема 1. Пусть C — ГПУ, T — полное АДД для C, а p — предикат. Для любой вершины L из C если T безопасно по отношению к конфигурации (L, p), то ни одно конкретное состояние с меткой L и значениями переменых, обращающими p в истину, не является достижимым в программе, соответствующей C.

Доказательство этой теоремы можно найти в [38].

4 Построение абстрактных деревьев достижимости

4.1 Построение ГПУ

Формально построение ГПУ по данной простейшей программе с уникальными метками можно описать следующим образом:

- оператору присваивания op_a, помеченному меткой L_1, за которым следует оператор (или точка выхода) с меткой L_2 в ГПУ ставится в

соответствие дуга из вершины L_1 в вершину L_2, помеченная оператором op_a;

- оператору ветвления по условию c_b, помеченному меткой L_1, который передаёт управление на метку L_2 в случае выполнения условия c_b или на метку L_3 в противном случае, в ГПУ ставится в соответствие две дуги – одна из вершины L_1 в вершину L_2 с пометкой (c_b), а другая – из вершины L_1 в вершину L_3 с пометкой $!\,(c_b)$;
- оператору безусловного перехода на метку L_2, помеченному L_1, в ГПУ ставится в соответствие дуга с меткой $goto\ L_2$, направленная из вершины L_1 в вершину L_2.

Таким образом, более одной исходящей дуги могут иметь только вершины ГПУ, соответствующие оператору ветвления. Но так как в простейших программах недопустимы пустые операторы, дуги, исходящие из этих вершин, будут всегда входить в различные вершины (соответствующие меткам операторов в ветвях оператора if). Поэтому в построенном ГПУ не будет кратных дуг.

4.2 Представление предикатов

При построении АДД будем представлять предикаты с помощью логических формул. В логических формулах будем использовать функции сложения (+), вычитания (−), умножения на константу (·), предикаты $>, <, =, \neq, \geq, \leq$, логические связки $\wedge$ и $\vee$, а также логическое отрицание $\neg$. Таким образом, каждому допустимому выражению простейшей программы можно будет однозначно поставить в соответствие логическую формулу, заменив используемые в этом выражении операции соответствующими функциями и предикатами .

При установлении такого соответствия делается упрощающее предположение о неограниченности диапазона возможных значений типа `int` и о соответствии конкретных аппаратных реализаций операций сложения, вычитания, умножения, сравнения и т.п. (с учётом возможных переполнений) их математическим аналогам. Поскольку на практике такие упрощающие предположения не верны, использование логических формул с функциями +, −, · и предикатами $>, <, =, \neq, \geq, \leq$ снижает точность верификации и может приводить как к выдаче инструментом верификации невыполнимых на практике примеров ошибочного пути, так и к фиктивному (основанному на ложных предположениях) доказательству недостижимости ошибочной метки.

Строго говоря, к примеру, при выполнении на компьютере с архитектурой, в которой размер типа `int` равен 32 битам, программа на языке C, аналогичная простейшей программе из примера 1, является некорректной в соответствии со свойством недостижимости ошибочной метки ERR. Вот выполнимый пример ошибочного пути:

$$(L1, \{x \mapsto 2147483647, y \mapsto -2147483648, z \mapsto 0\}) \xrightarrow{(x>y)}$$
$$(L2, \{x \mapsto 2147483647, y \mapsto -2147483648, z \mapsto 0\}) \xrightarrow{z=x-y}$$
$$(L4, \{x \mapsto 2147483647, y \mapsto -2147483648, z \mapsto -1\}) \xrightarrow{(z<0)}$$
$$(ERR, \{x \mapsto 2147483647, y \mapsto -2147483648, z \mapsto -1\})$$

На практике все существующие на сегодняшний день инструменты статической верификации, использующие предикатную абстракцию (в частности, декартову) с указанными функциями и предикатами, не гарантируют не только абсолютную, но даже самую высокую точность верификации среди вообще всех существующих инструментов. К примеру, инструмент CBMC, использующий подход BMC (ограничиваемую проверку моделей), найдёт для программы из примера 1 выполнимый пример ошибочного пути. Инструменты BLAST и CPAchecker, использующие CEGAR с предикатной абстракцией, считают приведённую в примере 1 программу корректной. Существуют, впрочем, и гибридные подходы к верификации с использованием CEGAR и предикатной абстракции, но без использования математических функций +, −, · и т.п., которые также могут найти для данной программы пример ошибочного пути (пример такого инструмента – SATabs). Основным преимуществом инструментов BLAST и CPAchecker является их применимость к реальным промышленным программам среднего размера (порядка 20 тыс. строк кода на C), которые практически не поддаются верификации с помощью более точных инструментов.

Выполнимость логических формул с указанными нами функциями и предикатами могут проверять инструменты проверки выполнимости (SMT-решатели). Подробнее о таких инструментах рассказано в 8 разделе данной статьи.

4.3 Задание сильнейших постусловий с помощью представления SSA

Сильнейшие постусловия можно задавать непосредственно для дуг ГПУ в виде логических формул, например, с использованием так называемого представления SSA [39]. В представлении SSA (от англ. Single State

Assignment) каждой переменной значение присваивается лишь единожды. Для этого к каждой переменной программы приписывается индекс, который увеличивается при каждом присваивании.

Предполагая, что индексы всех переменных в SSA представлении изначально равны 1, сильнейшие постусловия для операторов простейшей программы можно записать так:

$$SP\big(p(x_1,\dots,x_k,\dots,x_n), x_k = e(x_1,\dots,x_k,\dots,x_n)\big) =$$
$$p({x_1}^1,\dots,{x_k}^1,\dots,{x_n}^1)$$
$$\wedge\big({x_k}^2 = e({x_1}^1,\dots,{x_k}^1,\dots,{x_n}^1)\big), \text{индекс } x_k \text{ теперь равен 2}$$

$$SP(p(x_1,\dots,x_n),(\,c(x_1,\dots,x_n))) = p({x_1}^1,\dots,{x_n}^1)\wedge c({x_1}^1,\dots,{x_n}^1)$$

$$SP(p(x_1,\dots,x_n), goto\ L) = p({x_1}^1,\dots,{x_n}^1)$$

Здесь верхний индекс переменной указывает её индекс в представлении SSA. Аналогично сильнейшие постусловия можно выписать при других значениях индексов переменных. Если обозначить индекс произвольной переменной x_k через $\sigma(x_k)$, то:

$$SP\big(p(x_1,\dots,x_n), x_k = e(x_1,\dots,x_n)\big)$$
$$= p\big({x_1}^{\sigma(x_1)},\dots,{x_n}^{\sigma(x_n)}\big)$$
$$\wedge\Big({x_k}^{\sigma(x_k)+1} = e\big({x_1}^{\sigma(x_1)},\dots,{x_n}^{\sigma(x_n)}\big)\Big); \sigma(x_k)\leftarrow\sigma(x_k)+1$$

$$SP(p(x_1,\dots,x_n), c(x_1,\dots,x_n)) = p\big({x_1}^{\sigma(x_1)},\dots,{x_n}^{\sigma(x_n)}\big)\wedge c({x_1}^{\sigma(x_1)},\dots,{x_n}^{\sigma(x_n)})$$

$$SP(p(x_1,\dots,x_n), goto\ L) = p({x_1}^{\sigma(x_1)},\dots,{x_n}^{\sigma(x_n)})$$

Здесь $\sigma(x_k)\leftarrow\sigma(x_k)+1$ означает увеличение на 1 индекса переменной x_k.

4.4 Вычисление регионов через сильнейшие постусловия

При вычислении регионов по приведённому ранее (в разделе 3.7) правилу будем использовать уточнённую SSA-индексами запись формулы конъюнкции предикатов для случая $SP(r,op)-SAT$:

$$\mathrm{post}_{\{p_1,\dots,p_n\}}(r,op)$$
$$= \bigwedge_{\substack{p_{i_k}\in\{p_1,\dots,p_n\}:\\ SP(r,op)\wedge\neg p_{i_k}({x_1}^{\sigma(x_1)},\dots,{x_n}^{\sigma(x_n)})-UNSAT}} p_{i_k}({x_1}^{\sigma(x_1)},\dots,{x_n}^{\sigma(x_n)}),$$

Здесь $p_{i_k}({x_1}^{\sigma(x_1)},\dots,{x_n}^{\sigma(x_n)})$ получается из $p_{i_k}(x_1,\dots,x_n)$ приписыванием самых последних индексов SSA представления всем его переменным.

4.5 Построение АДД на основе ГПУ

Будем строить соответствующее полное АДД, последовательно перебирая пути в ГПУ с помощью обхода в глубину. Взяв начальный регион ⊤, мы построим начальную вершину АДД, соответствующую начальной вершине ГПУ. Затем для последней построенной вершины АДД $N\colon(L,r)$ будем строить переход по одной из соответствующих исходящих дуг ГПУ (из L в L' с меткой op — $L\xrightarrow{op}L'$) в новую вершину $N'\colon(L',r')$, где $r'=\mathrm{post}_\pi(r,op)$, π — выбранное множество предикатов. Для каждой вновь построенной вершины $N'\colon(L',\varphi')$ будем проверять наличие соответствующих исходящих дуг в ГПУ, выполнимость соответствующего региона φ', а также покрытие какой-либо уже построенной вершиной $N''\colon(L',\varphi'')$, где $\varphi'\to\varphi''$. При выполнении хотя бы одного из этих условий будем прекращать построение дуг из последней вершины и возвращаться к предыдущей построенной вершине. Построение АДД будем заканчивать тогда, когда для каждой его непокрытой вершины с выполнимым регионом будут построены переходы по всем соответствующим ей исходящим дугам в ГПУ. Полученное АДД будет являться полным по построению, поэтому для проверки свойства достижимости ошибочной метки E достаточно будет проверить его безопасность по отношению к конфигурации $(E,\top)$, то есть, по сути, проверить выполнимость регионов в его вершинах с меткой E (что на самом деле уже будет сделано во время построения).

Конечность АДД обеспечивается конечностью числа дуг ГПУ и описанным способом построения, обеспечивающим, в частности, конечность числа дуг АДД, соответствующих одной дуге ГПУ, за счёт проверки покрытия вершин. Дуге ГПУ, исходящей из вершины L, может соответствовать лишь конечное число дуг АДД, не большее, чем число различных возможных абстрактных состояний с меткой L, то есть 2^n+1, где n — число предикатов в выбранной абстракции. Это объясняется тем, что если из какой-либо уже построенной вершины АДД $N\colon(L,r)$ исходит дуга, то другая его вершина с тем же абстрактным состоянием $N'\colon(L,r)$ окажется покрыта ранее построенной вершиной N (т.к. $\forall\varphi.\,\varphi\to\varphi$) и, как следствие, не будет иметь исходящих дуг.

5 Пример построения АДД

Чтобы лучше осознать основные проблемы, которые позволяет решить метод CEGAR, рассмотрим в начале процесс построения АДД по заранее заданному набору предикатов на примере уже рассмотренной ранее простейшей программы (пример 1). Для этой программы мы попытаемся с помощью полного АДД доказать недостижимость ошибочного оператора, помеченного меткой ERR. Данный простой пример наглядно демонстрирует, что подбор предикатов, подходящих для доказательства недостижимости ошибочного состояния даже в очень простых программах, является непростой задачей.

Итак. ГПУ для примера 1 представлен на рис. 1б.

Выберем множество из пяти предикатов $\pi = \{p_1, \dots, p_5\}$:

$$p_1 \equiv x > 0$$
$$p_2 \equiv x = 0$$
$$p_3 \equiv y > 0$$
$$p_4 \equiv y = 0$$
$$p_5 \equiv z \geq 0$$

Построим АДД описанным выше способом:

1. Строим вершину $L1\#1\colon (L1, \top)$.
2. Вершина $L1\#1$имеет соответствующую исходящую дугу $L1 \xrightarrow{x>y} L2$ в ГПУ, её регион $\top - SAT$ и она, очевидно, не покрыта никакой ранее построенной вершиной.
3. Строим дугу $L1\#1\colon (L1, \top) \xrightarrow{x>y} L2\#1\colon (L2, \mathrm{post}_\pi(\top, (x > y)))$.
4. Вычисляем $\mathrm{post}_\pi(\top, (x > y))$, пользуясь соответствующим правилом:

   предполагаем все индексы SSA равными 1 и пользуемся представлением SSA для записи сильнейшего постусловия:

   $$SP\big(\top, (x > y)\big) \equiv x_1 > y_1 - SAT\ (x_1 = 1, y_1 = 0)$$

   (в скобках указана модель для выполнимой формулы),

   значит $\mathrm{post}_\pi(\top, (x > y)) \not\equiv \bot$

$$p_1\colon SP\big(\top, (x > y)\big) \wedge \neg p_1$$
$$\equiv \top \wedge (x_1 > y_1) \wedge \neg(x_1 > 0)$$
$$- SAT(x_1 = 0, y_1 = -1)$$

значит p_1не входит в искомую конъюнкцию

$$p_2\colon SP\big(\top, (x > y)\big) \wedge \neg p_2$$
$$\equiv \top \wedge (x_1 > y_1) \wedge \neg(x_1 = 0)$$
$$- SAT(x_1 = 1, y_1 = 0)$$
$$p_3\colon SP\big(\top, (x > y)\big) \wedge \neg p_3$$
$$\equiv \top \wedge (x_1 > y_1) \wedge \neg(y_1 > 0)$$
$$- SAT(x_1 = 1, y_1 = 0)$$
$$p_4\colon SP\big(\top, (x > y)\big) \wedge \neg p_4$$
$$\equiv \top \wedge (x_1 > y_1) \wedge \neg(y_1 = 0)$$
$$- SAT(x_1 = 2, y_1 = 1)$$
$$p_5\colon SP\big(\top, (x > y)\big) \wedge \neg p_5$$
$$\equiv \top \wedge (x_1 > y_1) \wedge \neg(z_1 \geq 0)$$
$$- SAT(x_1 = 1, y_1 = 0, z_1 = -1)$$

Искомая конъюнкция пуста. Значит, $\mathrm{post}_\pi\big(\top, (x > y)\big) \equiv \top$.

5. Переходим к вершине $L2\#1\colon (L2, \top)$. Она имеет соответствующую исходящую дугу $L2 \xrightarrow{z=x-y} L4$ в ГПУ, её регион $\top - SAT$и она не покрыта никакой ранее построенной вершиной.
6. Строим дугу $L2\#1\colon (L2, \top) \xrightarrow{z=x-y} L4\#1\colon (L4, \mathrm{post}_\pi(\top, z = x - y))$.
7. Вычисляем $\mathrm{post}_\pi(\top, z = x - y)$, пользуясь правилом:

   полагаем индексы переменных равными 1 и пользуемся представлением SSA для записи сильнейшего постусловия:

   $$SP(\top, z = x - y) \equiv \top \wedge z_2$$
   $$= x_1 - y_1 - SAT(x_1 = 1, y_1 = 2, z_2 = -1),$$

   значит $\mathrm{post}_\pi(\top, z = x - y) \not\equiv \bot$

$p_1: SP(\top, z = x - y) \wedge \neg p_1 \equiv \top \wedge z_2$
$= x_1 - y_1 \wedge \neg(x_1 > 0) - SAT(x_1 = 0, y_1 = 0, z_2$
$= 0)$

...

$p_5: SP(\top, z = x - y) \wedge \neg p_5 \equiv \top \wedge z_2$
$= x_1 - y_1 \wedge \neg(z_2 \geq 0) - SAT(x_1 = 0, y_1 = 1, z_2$
$= -1)$

Значит, $\text{post}_\pi(\top, z = x - y) \equiv \top$.

8. Переходим к вершине $L4\#1: (L4, \top)$. Дуга $L4 \xrightarrow{!(z<0)} L5$в ГПУ, регион $\top - SAT$, вершина не покрыта.
9. Строим дугу $L4\#1: (L4, \top) \xrightarrow{!(z<0)} L5\#1: (L5, \text{post}_\pi(\top, !(z < 0)))$.
10. Вычисляем $\text{post}_\pi(\top, !(z < 0))$ по правилу:
    полагаем индексы переменных равными 1 и пользуемся представлением SSA:

$$SP(\top, !(z < 0)) \equiv \top \wedge \neg(z_1 < 0) - SAT(z_1 = 0)$$
$$p_1: SP(\top, !(z < 0)) \wedge \neg p_1 \equiv \top \wedge \neg(z_1 < 0) \wedge \neg(x_1 > 0) - SAT(z_1 = x_1 = 0)$$

...

$$p_5: SP(\top, !(z < 0)) \wedge \neg p_5 \equiv \top \wedge \neg(z_1 < 0) \wedge \neg(z_1 \geq 0) \equiv \neg(z_1 < 0) \wedge (z_1 < 0) - UNSAT$$

    Значит, $\text{post}_\pi(\top, !(z < 0)) \equiv z \geq 0$
11. Переходим к вершине $L5\#1: (L5, z \geq 0)$. Эта вершина не имеет соответствующих исходящих дуг в ГПУ (соответствующая ей вершина $L5$ не имеет исходящих дуг).
12. Возвращаемся к предыдущей построенной вершине $L4\#1$. Рассматриваем следующую соответствующую ей исходящую дугу $L4 \xrightarrow{(z<0)} ERR$ в ГПУ.

13. Строим дугу $L4\#1: (L64\top) \xrightarrow{(z<0)} ERR\#1: \left(ERR, \text{post}_\pi(\top, (z < 0))\right)$.
14. Вычисляем $\text{post}_\pi(\top, (z < 0))$. Получаем $\text{post}_\pi(\top, (z < 0)) = \top$.
15. Переходим к вершине $ERR\#1: (ERR, \top)$. Рассматриваем дугу $ERR \xrightarrow{goto\ ERR} ERR$в ГПУ. Регион $\top - SAT$, вершина не покрыта.
16. Строим дугу $ERR\#1: (ERR, \top) \xrightarrow{goto\ ERR} ERR\#2: (ERR, \text{post}_\pi(\top, goto\ ERR))$.
17. Вычисляем результат $\text{post}_\pi(\top, goto\ ERR) = \top$.
18. Переходим к вершине $ERR\#2(ERR, \top)$. Она имеет исходящую дугу
    $ERR \xrightarrow{goto\ ERR} ERR$в ГПУ. Регион $\top - SAT$. Но в АДД есть уже построенная вершина $ERR\#1: (ERR, \top)$, такая, что $\top \to \top$. Значит, вершина $ERR\#2$ покрыта вершиной $ERR\#1$.
19. Возвращаемся к предыдущей построенной вершине $ERR\#1$. Соответствующая ей единственная исходящая дуга ГПУ ($ERR \xrightarrow{goto\ ERR} ERR$) уже рассмотрена.
20. Возвращаемся к предыдущей построенной вершине $L4\#1$. Все соответствующие ей исходящие дуги ГПУ $L4 \xrightarrow{!(z<0)} L5$ и $L4 \xrightarrow{(z<0)} ERR$ уже рассмотрены.
21. Возвращаемся к предыдущей построенной вершине $L2\#1$. Соответствующая ей исходящая дуга ГПУ $L2 \xrightarrow{z = x - y} L4$ уже рассмотрена.
22. Возвращаемся к предыдущей построенной вершине $L1\#1$. Рассматриваем следующую ещё не рассмотренную соответствующую ей исходящую дугу в ГПУ $L1 \xrightarrow{!(x>y)} L3$.
23. Строим дугу $L1\#1: (L3, \top) \xrightarrow{!(x>y)} L3\#1: (L3, \text{post}_\pi(\top, !(x > y)))$.
24. Вычисляем регион $\text{post}_\pi(\top, !(x > y)) = \top$.

25. Переходим к вершине $L3\#1\colon (L3, \top)$. Рассматриваем дугу $L3 \xrightarrow{z=y-x} L4$в ГПУ. Регион $\top - SAT$, вершина $L3\#1$ не покрыта.
26. Строим дугу $L3\#1\colon (L5, \top) \xrightarrow{z=y-x} L4\#2\colon (L4, \mathrm{post}_\pi(\top, z = y - x))$.
27. Вычисляем регион $\mathrm{post}_\pi(\top, z = y - x) = \top$.
28. Переходим к вершине $L4\#2\colon (L4, \top)$. Она покрыта вершиной $L4\#1\colon (L4, \top)$.
29. Убеждаемся, что для каждой непокрытой вершины АДД с выполнимым регионом построены переходы по всем соответствующим ей исходящим дугам в ГПУ. Таким образом, построение АДД закончено.

Полученное АДД показано на рис. 10. Регионы, соответствующие вершинам дерева, показаны в прямоугольниках рядом с вершинами. Покрытие обозначено пунктирной стрелкой, помеченной словами “covered by”, в направлении от покрываемой вершины к покрывающей.

Очевидно, что построенное АДД *не* является безопасным по отношению к конфигурации $(ERR, \top)$ и таким образом не позволяет доказать недостижимость ошибочной метки ERR. В то же время легко видно, что (в рамках наших упрощающих предположений) в исходной программе эта метка не является достижимой.

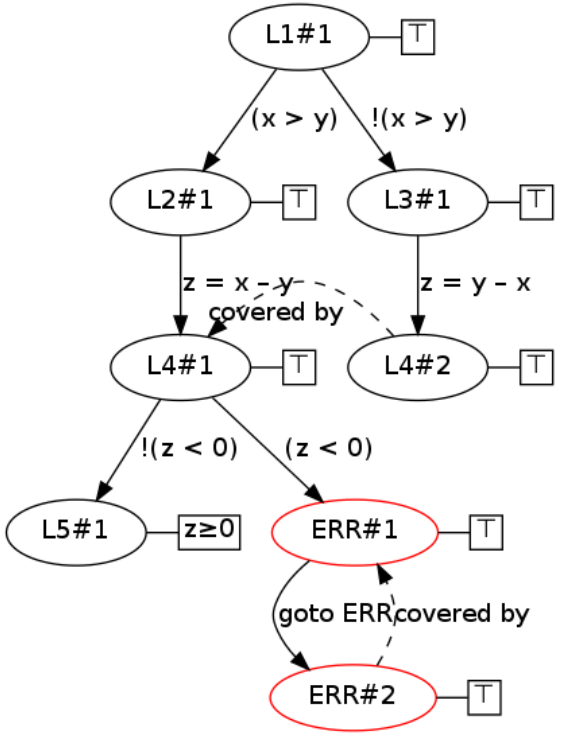

Рис. 10. Абстрактное дерево достижимости для заранее заданного набора предикатов $b_1 - b_5$.

На рассмотренном примере хорошо видно, что подбор набора предикатов, подходящего для доказательства недостижимости ошибочного состояния, является непростой задачей. Подобрать нужные предикаты заранее трудно, так как часто они сильно зависят от конкретной программы. Поэтому исследования в области методов поиска набора предикатов для абстракции ведутся в основном в направлении получения новых предикатов на основе исходной программы и какой-либо уже построенной для неё абстракции (например, АДД), которая не позволила доказать корректность программы относительно свойства недостижимости.

6 Метод CEGAR

6.1 Контрпример

В этом разделе *контрпримером* будем называть некоторый путь в ГПУ из его начальной вершины в вершину, помеченную ошибочной меткой. Основное отличие этого понятия от примера ошибочного пути в графе достижимости состоит в том, что контрпример не включает состояния программы. Это, в частности, означает, что контрпример может не соответствовать никакому пути в графе достижимости. В таком случае контрпример называют *ложным*. Для рассмотренной простейшей программы (пример 1) ложным контрпримером является, например, такой путь:

$$L1 \xrightarrow{x = nondet} L2 \xrightarrow{y = nondet} L3 \xrightarrow{(x>y)} L4 \xrightarrow{z = x - y} L6 \xrightarrow{(z < 0)} ERR$$

В этом разделе мы рассмотрим метод CEGAR – Counter Example Guided Abstraction Refinement, то есть метод уточнения абстракции по контрпримеру, суть которого, как следует из названия, заключается в получении новых предикатов на основе ложного контрпримера. Ложный контрпример получают в результате построения АДД, которое не позволят доказать корректность исходной программы относительно свойства недостижимости.

6.2 Уточнение абстракции на примере

Рассмотрим сначала простейшую программу на рис. 11:

```
void main()

{

      int x = nondet;

      L1: x = 0;

      L2: if (x < 0)

      ERROR: goto ERROR;

}
```

Рис 11. Простейшая программа

Будем считать, что для неё построено АДД для пустого набора предикатов, так что во всех его вершинах регионы абстрактных состояний равны $\top$.

Соответствующий простейший ложный контрпример: $L1 \xrightarrow{x=0} L2 \xrightarrow{(x<0)} ERROR$. Запишем для первого перехода в этом пути сильнейшее постусловие: $SP(\top, x = 0) \equiv (x^2 = 0)$ (верхние индексы, как и ранее, обозначают индексы в представлении SSA). Запишем теперь постусловие для второго перехода, подставив в него вместо региона $\top$ первое сильнейшее постусловие: $SP(SP(\top, x = 0), x < 0) \equiv (x^2 = 0) \wedge (x^2 < 0)$. Полученная конъюнкция состоит из двух частей. Первая задаёт состояние переменной x перед выполнением оператора условного перехода, а вторая – дополнительные ограничения, наложенные на переменную x в результате этого перехода. Эта конъюнкция как логическая формула является невыполнимой.

Вспомнив правила построения абстрактных состояний с помощью оператора post_π, можно попытаться подобрать новый подходящий предикат для исключения найденного ложного контрпримера из АДД, исходя из следующих соображений:

При вычислении региона абстрактного состояния $(L2, \text{post}_\pi(\top, x = 0))$ новый предикат должен войти в результирующую конъюнкцию, чтобы изменить (уточнить) абстрактное состояние. Для этого должно выполняться условие $SP(\top, x = 0) \to p(x^2)$, или в более общем виде, $SP(\top, op_1) \to p(x^{\sigma(x)})$, где $p(x)$ – новый предикат, $\sigma(x)$ — значение SSA-индекса переменной x в состоянии $(L2, \text{post}_\pi(\top, op_1))$.

Исходя из предыдущего соображения, $p(x)$ войдёт в регион абстрактного состояния $(L2,\ \text{post}_\pi(\top, x = 0))$. Будем считать $\text{post}_\pi(\top, x = 0) \equiv p(x)$. При вычислении абстрактного состояния $(L3,\ \text{post}_\pi(p(x),\ x < 0))$ нужно получить результат $\bot$. Это покажет недостижимость метки $ERROR$. Для этого должно выполняться условие $SP(p(x), x < 0) - UNSAT$, то есть $p(x^1) \wedge (x^1 < 0) - UNSAT$. В более общем виде: $SP(p(x), op_2) - UNSAT$.

- В условиях $SP(\top, x = 0) \to p(x^2)$ и $SP(p(x), x < 0) - UNSAT$, то есть $(x^2 = 0) \to p(x^2)$ и $p(x^1) \wedge (x^1 < 0) - UNSAT$ в предикат $p(x)$ подставляются, вообще говоря, различные переменные (в данном случае x^1 и x^2). Однако эти условия можно рассмотреть и совместно, если воспользоваться конъюнкцией $SP(SP(\top, x = 0), x < 0) \equiv (x^2 = 0) \wedge (x^2 < 0)$ и взять предикат $p(x)$ только над общими переменными обеих частей этой конъюнкции. Тогда, взяв условия $(x^2 = 0) \to p(x^2)$ и $p(x^2) \wedge (x^2 < 0) - UNSAT$, мы можем найти удовлетворяющий им предикат в виде $p(x^2)$, то есть в общем случае $p(x^{\sigma(x)})$. При этом простое переименование переменных $x_2 \to x_1$ во

втором условии даст в точности нужное исходное условие $p(x^1) \wedge (x^1 < 0) - UNSAT$.

В более общем случае для конъюнкции $SP(SP(\top, op_1), op_2) \equiv SP(\varphi, op_2) \equiv \varphi \wedge \psi$ следует искать предикат ρ над общими переменными φ и ψ такой, что $\varphi \to \rho$ и $\rho \wedge \psi - UNSAT$. Для такого ρ будут выполняться условия

$SP(\top, op_1) \to \rho(x_{i_1}^{\sigma(x_{i_1})}, \dots, x_{i_m}^{\sigma(x_{i_m})})$ и $SP\big(\rho\big(x_{i_1}, \dots, x_{i_m}\big), op_2\big) - UNSAT$, где $x_{i_1}, \dots, x_{i_m}$ — общие переменные, используемые как в операторах op_1 и op_2.

Итого для постусловия вида $SP(SP(\top, op_1), op_2) \equiv SP(\varphi,\ op_2) \equiv \varphi \wedge \psi$ и искомого предиката ρ имеем следующие три условия:

- $\varphi \to \rho$
- $\rho \wedge \psi - UNSAT$
- ρ — формула над общими переменными, входящими в формулы φ и ψ.

Задача поиска такого предиката ρ является известной [8, 19, 53, 54, 55, 56, 57, 58] задачей построения интерполянта Крейга для конъюнкции двух логических формул (в данном случае, двух SMT-формул без кванторов). Такие предикаты при определенных ограничениях можно искать с помощью специальных инструментов – интерполирующих решателей. Для дальнейшего использования с целью уточнения абстракции предикат ρ при этом должен быть получен в рамках логики нулевого порядка, то есть не содержать кванторов. В этом случае новый предикат для абстракции (то есть без индексов) можно получить из ρ, опустив индексы всех входящих в него переменных.

6.3 Сильнейшее постусловие пути

Итак, вернувшись к рассмотренному в разделе 5 примеру, заметим, что построенное АДД не только не доказывает недостижимость ошибочной метки ERR, но также позволяет легко получить ложный контрпример, рассмотрев какой-либо путь из своей начальной вершины (соответствующей начальной вершине ГПУ) до одной из вершин с выполнимым регионом, помеченных ошибочной меткой ERR. По построению этот путь будет соответствовать

некоторому пути в ГПУ из его начальной вершины в вершину с ошибочной меткой.

Предположим вначале, что полученный контрпример представляет собой одно из реально возможных выполнений исходной программы, приводящих к достижению ею ошибочного состояния. Это означает, что существует соответствующий пример ошибочного пути в графе достижимости. Чтобы указать путь в графе достижимости, нужно найти состояния программы, соответствующие вершинам этого пути. Эти состояния содержат означивания переменных программы непосредственно до выполнения каждого оператора найденного контрпримера.

Найденный контрпример, то есть путь P в ГПУ, с другой стороны, состоит из последовательности меток и операторов исходной программы, выполнение которых по нашему предположению приводит к достижению метки ERR. Вспомним, что данное нами определение сильнейшего постусловия оператора op при данном предусловии p позволяет получить предикат, задающий (вместе с некоторой меткой) ограничения на состояния программы, непосредственно достижимые из состояний, заданных предусловием p (и некоторой меткой), в результате выполнения оператора op. Пусть мы имеем два последовательно выполняемых оператора op_1 и op_2 $(op_1; op_2)$. Если в качестве предусловия p_2 для оператора op_2 взять сильнейшее постусловие оператора op_1 при некотором предусловии p $(p_2 = SP(p, op_1))$, то сильнейшее постусловие оператора op_2 при предусловии p_2 $(SP(p_2, op_2) = SP(SP(p, op_1), op_2))$ будет задавать состояния программы, достижимые из состояний, заданных предусловием p, в результате последовательного выполнения операторов op_1 и op_2 $(op_1; op_2)$. Аналогично можно построить сильнейшее постусловие для трех и более последовательно выполняемых операторов $op_1; \dots; op_n (n \geq 3)$. Имея теперь последовательность операторов найденного пути $P: L_1: op_1; L_2: op_2; \dots; L_n: op_n$, мы можем по индукции определить сильнейшее постусловие для всей этой последовательности операторов при некотором предусловии r_0, задающем начальные состояния. Данное сильнейшее постусловие будет представлять собой предикат, задающий состояния, достижимые из какого-либо начального состояния по этому пути P. Так как P — путь в вершину с ошибочной меткой, то полученный предикат будет задавать ошибочные состояния, достижимые из какого-либо начального состояния по пути P. Возьмём в качестве r_0 тождественную истину $r_0 \equiv \top$. Получим:

$$SP(P) \equiv SP(r_0, P) \equiv SP(\top, P) \equiv SP(\top, op_1; op_2; \dots; op_{n-1}; op_n)$$
$$\stackrel{\text{def}}{=} SP(SP(\top, op_1; op_2; \dots; op_{n-1}), op_n) = \cdots$$
$$= SP(SP(\dots SP(SP(\top, op_1), op_2) \dots, op_{n-1}), op_n)$$

Здесь через $SP(P)$ мы обозначили сильнейшее постусловие пути P.

Если представить предикат $SP(P)$ в виде логической формулы, можно говорить о выполнимости, либо невыполнимости этой формулы, а также в случае её выполнимости — о модели, обращающей эту формулу в тождественную истину.

Если формула $SP(P)$ выполнима, то это означает, что она задаёт хотя бы одно достижимое из начального по пути P ошибочное состояние. Таким образом, исходная программа не является корректной относительно свойства недостижимости, а найденный контрпример P не является ложным. Кроме этого, при записи постусловий с помощью представления SSA, модель этой формулы, обращающая её в тождественную истину, будет задавать значения некоторых переменных программы непосредственно до выполнения каждого оператора найденного контрпримера. Эти значения можно использовать для указания соответствующего примера ошибочного пути в графе достижимости. Инструменты верификации (в частности, BLAST и CPAchecker) в таком случае, как правило, просто выдают найденный контрпример P в качестве результата своей работы.

Если формула $SP(P)$ невыполнима, можно говорить о том, что путь P до ошибочной метки, полученный в результате построения АДД, не соответствует никакому возможному выполнению исходной программы. В этом случае инструменты верификации обычно пытаются получить на основе невыполнимой формулы $SP(P)$ новые предикаты, а затем построить на основе объединения множеств старых и новых предикатов новое АДД, в котором ошибочная метка будет недостижима, по крайней мере, по пути P. В случае BLAST и CPAchecker получение множества новых предикатов основано на использовании интерполяции Крейга для невыполнимой конъюнкции двух или более логических формул.

6.4 Интерполяция Крейга

В математической логике интерполяционная теорема Крейга утверждает, что если для двух логических формул φ и ψ общезначима (тождественно истинна на любой модели) импликация $\varphi \to \psi$, то существует логическая формула ρ, называемая *интерполянтом Крейга*, которая удовлетворяет трём условиям:

1. $\varphi \to \rho$;
2. $\rho \to \psi$;
3. каждый неинтерпретируемый (не заданный какой-либо логической теорией) символ в формуле ρ является общим для формул φ и ψ.

Теорема была впервые доказана для логики первого порядка У. Крейгом в 1957 году [40]. Третье условие для формулы ρ в этой теореме, в частности, означает, что ρ может содержать только те переменные и

неинтерпретируемые функции, которые являются общими для формул φ и ψ, то есть входят в обе эти формулы.

Подставим вместо формулы ψ в условие интерполяционной теоремы формулу вида $\neg\psi_1$. Тогда общезначимость $\varphi \to \psi \equiv \neg\varphi \vee \psi$, $(\neg\varphi \vee \psi)|_{\psi=\neg\psi_1} \equiv \neg\varphi \vee \neg\psi_1$ будет эквивалентна невыполнимости $\neg(\neg\varphi \vee \neg\psi_1) \equiv \varphi \wedge \psi_1$. Значит можно сказать, что соответствующий интерполянт ρ существует для невыполнимой конъюнкции двух логических формул $\varphi \wedge \psi_1$ и удовлетворяет трём условиям:

1. $\varphi \to \rho$
2. $\rho \to \neg\psi_1 \Leftrightarrow \neg\rho \vee \neg\psi_1 \Leftrightarrow (\rho \wedge \psi_1 - UNSAT)$
3. каждый неинтерпретируемый (не заданный какой-либо логической теорией) символ в формуле ρ является общим для формул φ и ψ_1.

6.5 Уточнение абстракции

Рассмотрим теперь невыполнимую формулу $SP(P)$. Она получена в результате индуктивного выписывания постусловий операторов в пути $P\colon L_1\colon op_1; L_2\colon op_2; \dots; L_n\colon op_n$, причем при использовании представления SSA каждое следующее постусловие получается из предыдущего добавлением к нему через конъюнкцию новой логической подформулы[2] (тождественной истины $\top$ в случае оператора `goto`) с возможным увеличением индекса одной из переменных программы в представлении SSA (см. соответствующие формулы для $SP(p, op)$ в разделе 4.3). Это означает, что полученную формулу $SP(P)$ можно представить в виде $SP(P) = \top \wedge \psi_1 \wedge \psi_2 \wedge \dots \wedge \psi_n = \psi_1 \wedge (\psi_2 \wedge \dots \wedge \psi_n)$, где подформулы $\psi_1, \psi_2, \dots, \psi_n$ соответствуют операторам $op_1; op_2; \dots; op_n$ пути P.

Рассмотрим интерполянт Крейга ρ_1 для невыполнимой конъюнкции формул ψ_1 и $\psi_2 \wedge \dots \wedge \psi_n$. Так как по определению интерполянта в формулу ρ_1 могут входить только переменные, общие для формул ψ_1 и $\psi_2 \wedge \dots \wedge \psi_n$, то учитывая формулы, используемые для построения сильнейших постусловий, каждая переменная в формулу ρ_1 может входить только с одним индексом представления SSA. Это означает, что соответствующим переименованием переменных формуле ρ_1 можно поставить в соответствие новый предикат p_1' над переменными программы, опустив индексы представления SSA у входящих в эту формулу переменных. Это можно записать так:

$$p_1'\left(x_1{}^{\sigma(x_1)}, \dots, x_n{}^{\sigma(x_n)}\right) = \rho_1$$

[2] Подформулой называется часть формулы, сама являющаяся формулой.

То есть после приписывания переменным программы в новом предикате p_1' соответствующих индексов представления SSA (имеющихся после вычисления $SP(\top, op_1)$), из этого предиката получится интерполянт ρ_1.

Предположим теперь, что мы строим АДД, имея множество предикатов π_1', включающее в себя p_1'. Тогда при вычислении региона

$$\mathrm{post}_{\pi_1'}(\top, op_1) = \bigwedge_{\substack{p_{j_k} \in \pi_1': \\ SP(\top, op_1) \wedge \neg p_{j_k}\left({x_1}^{\sigma(x_1)}, \ldots, {x_n}^{\sigma(x_n)}\right) - UNSAT}} p_{j_k}$$

из условия (1) для интерполянта Крейга ρ_1 $(\psi_1 \to \rho_1)$, учитывая, что по построению $\psi_1 = SP(\top, op_1)$, мы получим

$$SP(\top, op_1) \to \rho_1 \Leftrightarrow$$

$$\neg SP(\top, op_1) \vee \rho_1 \Leftrightarrow$$

$$\neg\neg(\neg SP(\top, op_1) \vee \rho_1) \Leftrightarrow$$

$$\neg(SP(\top, op_1) \wedge \neg\rho_1) \Leftrightarrow$$

$$SP(\top, op_1) \wedge \neg p_1'\left({x_1}^{\sigma(x_1)}, \ldots, {x_n}^{\sigma(x_n)}\right) - UNSAT)$$

Это означает, что

$$\mathrm{post}_{\pi_1'}(\top, op_1) = p_1' \wedge r_1, \qquad r_1 - \text{некоторая конъюнкция}$$

То есть новый предикат p_1' из множества π_1' обязательно входит в регион для второй вершины L_2 на пути P.

Теперь рассмотрим оставшийся путь P': L_2: op_2; …; L_n: op_n.

$$SP\left(\mathrm{post}_{\pi_1'}(\top, op_1), P'\right)$$

$$= p_1'({x_1}^1, \ldots, {x_n}^1) \wedge r_1({x_1}^1, \ldots, {x_n}^1) \wedge (\psi_2' \wedge \ldots \wedge \psi_n')$$

$$= p_1'({x_1}^1, \ldots, {x_n}^1) \wedge (\psi_2' \wedge \ldots \wedge \psi_n') \wedge r_1({x_1}^1, \ldots, {x_n}^1)$$

Переименуем переменные в этой формуле, начав индексирование с последних индексов SSA, которые были получены для $SP(\top, op_1)$. После замены переменных вида ${x_1}^1 \to {x_1}^{\sigma(x_1)}, {x_2}^1 \to {x_2}^{\sigma(x_2)}, \ldots, {x_n}^1 \to {x_n}^{\sigma(x_n)}, {x_1}^2 \to {x_1}^{\sigma(x_1)+1}, {x_2}^2 \to {x_2}^{\sigma(x_2)+1}, \ldots$ получим:

$$p_1'\left({x_1}^{\sigma(x_1)}, \ldots, {x_n}^{\sigma(x_n)}\right) \wedge (\psi_2 \wedge \ldots \wedge \psi_n) \wedge r_1\left({x_1}^{\sigma(x_1)}, \ldots, {x_n}^{\sigma(x_n)}\right)$$

$$= \rho_1 \wedge (\psi_2 \wedge \ldots \wedge \psi_n) \wedge r_1$$

Эта формула выполнима тогда и только тогда, когда выполнима формула $SP\left(\mathrm{post}_{\pi_1'}(\top, op_1), P'\right)$. Поэтому из условия (2) для интерполянта Крейга ρ_1

$(\rho_1 \wedge (\psi_2 \wedge \ldots \wedge \psi_n) - UNSAT)$, получаем, что $SP\left(\mathrm{post}_{\pi_1'}(\top, op_1), P'\right) - UNSAT$. Таким образом, невыполнимость сильнейшего постусловия $SP(\top, P)$ для пути P сохраняется для оставшейся части пути P', взятой начиная со второго оператора op_2. Отметим, что для невыполнимости $SP\left(\mathrm{post}_{\pi_1'}(\top, op_1), P'\right)$ существенно лишь присутствие в π_1' нового предиката p_1'. Помимо него π_1' может включать и другие произвольные предикаты, в частности, предикаты, использованные при построении предыдущего АДД ($\pi \subset \pi'$). Это позволяет в ходе уточнения абстракции всё время лишь включать в соответствующие множества новые предикаты, не исключая при этом уже найденных новых предикатов. Поэтому будем предполагать $\pi \subset \pi_1' = \pi \cup \{p_1'\} \subset \pi_2' = \pi \cup \{p_1', p_2'\} \subset \cdots$.

Далее аналогично тому, как мы рассмотрели невыполнимую формулу $SP(\top, P)$, рассмотрим последовательно невыполнимые формулы $SP\left(\mathrm{post}_{\pi_1'}(\top, op_1), P'\right), SP\left(\mathrm{post}_{\pi_2'}\left(\mathrm{post}_{\pi_2'}(\varphi_1, op_1), op_2\right), P''\right), \ldots$ Через $\pi_1', \pi_2', \ldots$ обозначены соответствующие новые множества предикатов, а через $P', P'', \ldots$ — соответствующие оставшиеся части пути P. Обозначим $\mathrm{post}_{\pi_1'}(\varphi_1, op_1) = r_1'$, $\mathrm{post}_{\pi_2'}(\mathrm{post}_{\pi_2'}(\varphi_1, op_1), op_2) = r_2', \ldots$ Рано или поздно, то есть, по крайней мере, при рассмотрении $SP\left(\mathrm{post}_{\pi_{n-1}'}(\ldots, op_{n-1}), op_n\right) = SP(r_{n-1}', op_n) - UNSAT$ мы по правилу вычисления регионов (см. секцию 4) получим $\mathrm{post}_{\pi_{n-1}'}(r_{n-1}', op_n) = \bot$ (независимо от множества предикатов π_{n-1}'). Условие $SP(r_{k-1}', op_k) - UNSAT$ может быть выполнено и при меньших значениях $k < n$, но т.к. путь P состоит из конечного числа помеченных операторов, оно обязательно будет выполнено, по крайней мере, для самого последнего перехода $L_n \xrightarrow{op_n} ERR$ при $k = n$. Это означает, что в новом АДД, которое затем будет построено для набора предикатов $\pi_{n-1}' = \pi \cup \{p_1', \ldots, p_{k-1}'\}$ (здесь $k \leq n$: $SP(r_{k-1}', op_k) - UNSAT$) либо не будет присутствовать соответствующая вершина с ошибочной меткой ERR, либо регион в этой вершине будет невыполним.

При построении АДД описанным ранее способом одному и тому же пути в ГПУ не может соответствовать более одного пути в АДД, так как все пути в АДД по построению различаются хотя бы одной дугой. Поэтому если множество используемых предикатов будет изменяться только за счёт включения в него новых предикатов, то в результате работы инструмента верификации больше не будет получен контрпример, содержащий уже рассмотренный невыполнимый путь P до ошибочной метки. Таким образом, однократное успешное уточнение абстракции позволяет исключить из АДД, по крайней мере, один путь, не соответствующий какому-либо возможному выполнению исходной программы.

На практике в инструментах BLAST и CPAchecker реализация процесса уточнения абстракции отличается от рассмотренного индуктивного варианта.

После получения формулы $SP(\top, P)$ для сильнейшего постусловия пути P в виде $\top \wedge \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_n$ инструмент верификации всего за одно обращение к интерполирующему решателю получает все интерполянты Крейга, необходимые для уточнения набора предикатов.

Для n логических формул $\psi_1, \ldots, \psi_n$, конъюнкция которых невыполнима $(\psi_1 \wedge \ldots \wedge \psi_n - UNSAT)$, *набором интерполянтов Крейга* называется набор из $n-1$ формулы $\rho_1, \ldots, \rho_{n-1}$, удовлетворяющих условиям:

1. $\psi_1 \rightarrow \rho_1, \psi_1 \wedge \psi_2 \rightarrow \rho_2, \ldots, \psi_1 \wedge \ldots \wedge \psi_{n-1} \rightarrow \rho_{n-1}$
2. $\rho_1 \wedge (\psi_2 \wedge \ldots \wedge \psi_n) - UNSAT, \ldots, \rho_k \wedge (\psi_{k+1} \wedge \ldots \wedge \psi_n) - UNSAT, \ldots, \rho_{n-1} \wedge \psi_n - UNSAT$
3. для любого $k = \overline{1, n-1}$ каждый неинтерпретируемый символ в формуле ρ_k является общим для формул $\psi_1 \wedge \ldots \wedge \psi_k$ и $\psi_{k+1} \wedge \ldots \wedge \psi_n$.

При $n \geq 2$ для невыполнимой конъюнкции формул $(\psi_1 \wedge \ldots \wedge \psi_n - UNSAT)$ можно также определить набор индуктивных интерполянтов Крейга $\rho_1, \ldots, \rho_{n-1}$, для которых вместо условий (1) выполняются условия: $\psi_1 \rightarrow \rho_1, \rho_1 \wedge \psi_2 \rightarrow \rho_2, \ldots, \rho_{k-1} \wedge \psi_k \rightarrow \rho_k, \ldots, \rho_{n-2} \wedge \psi_{n-1} \rightarrow \rho_{n-1}$. Набор индуктивных интерполянтов является частным случаем набора интерполянтов Крейга. Действительно, предположим, что не выполнено одно из условий (1) для некоторого $1 \leq k \leq n-1$, то есть не выполнено $\psi_1 \wedge \ldots \wedge \psi_k \rightarrow \rho_k$. По определению импликации это означает, что $\psi_1, \ldots, \psi_k$ истины, в то время как ρ_k — ложь. Но тогда по условию для индуктивных интерполянтов учитывая, что $\psi_1, \ldots, \psi_k$ истины, получим $\psi_1 \rightarrow \rho_1, \rho_1 \rightarrow \rho_2, \ldots, \rho_{k-1} \rightarrow \rho_k$, то есть ρ_k — истина, что противоречит предположению. Легко видеть, что для индуктивных интерполянтов Крейга одновременное получение всех интерполянтов с помощью одного вызова интерполирующего решателя по определению эквивалентно уже рассмотренному индуктивному способу их получения. Для общего случая интерполянтов Крейга при некоторых ограничениях на формулы $\psi_1, \ldots, \psi_n$ и с существенным использованием условия (3) можно доказать исключение соответствующего невыполнимого ошибочного пути из нового АДД, как это делается, к примеру, в статье [41].

Таким образом, на практике успешное уточнение абстракции также означает исключение из АДД, по крайней мере, одного пути, не соответствующего какому-либо возможному выполнению исходной программы.

6.6 Цикл CEGAR

Верификация исходной программы методом CEGAR в инструментах BLAST и CPAchecker представляет собой цикл, на каждой итерации которого выполняются следующие последовательные шаги:

1. Построение АДД по текущему набору предикатов.

2. Проверка безопасности построенного АДД по отношению к конфигурации $(E, \top)$, где E — ошибочная метка.
   а) Если построенное АДД оказывается безопасным по отношению к конфигурации $(E, \top)$, то оно считается доказательством недостижимости ошибочной инструкции в исходной программе. Таким образом, инструмент верификации успешно завершает работу, доказав корректность программы относительно свойства недостижимости.
   б) Иначе происходит переход к шагу 3.
3. В построенном АДД находится вершина $N: (E, \varphi)$, $\varphi \neq \perp$. Строится соответствующий путь P из начальной вершины АДД в вершину N. Путь P затем рассматривается в качестве пути контрпримера. Вычисляется сильнейшее постусловие этого пути — $SP(P) \equiv SP(\top, P)$.
4. Проверка выполнимости $SP(P)$.
   а) Если постусловие $SP(P)$ как логическая формула оказывается выполнимой, то найденный контрпример P выдаётся в качестве результата работы инструмента, как пример выполнения исходной программы, при котором достигается заданная ошибочная метка.
   б) Иначе происходит переход к шагу 5.
5. Выполняется уточнение текущего набора предикатов с использованием невыполнимого ошибочного пути P с помощью интерполяции Крейга. При успешном уточнении набора предикатов происходит переход к шагу 1.

Таким образом, цикл CEGAR либо завершается на шаге 2 (с доказательством недостижимости), либо на шаге 4 (с контрпримером пути до ошибочной метки), либо выполняется вплоть до исчерпания выделенных инструменту верификации ресурсов (памяти и процессорного времени), то есть теоретически бесконечно. Бесконечное выполнение возможно, например, при

существовании подходящей бесконечной последовательности предикатов, выводимых на шаге 5.

7 Пример применения метода CEGAR

Рассмотрим применение метода CEGAR для доказательства недостижимости ошибочной метки в программе из примера 1.

1. Выберем пустой начальный набор предикатов $\pi = \emptyset$. Построим АДД, соответствующее пустому набору предикатов. Получим результат, показанный на рис. 12.

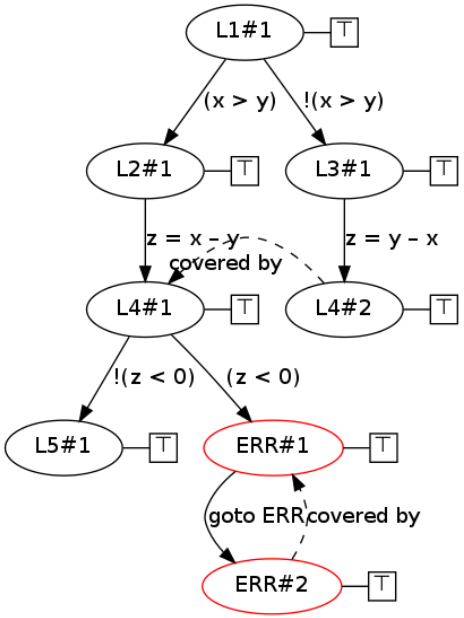

Рис.12. Абстрактное дерево достижимости для пустого набора предикатов $\pi = \emptyset$.

2. Построенное АДД не является безопасным по отношению к конфигурации $(ERR, \top)$, так как в нём присутствует вершина $ERR\#1: (ERR, \top)$.

3. Рассмотрим путь P от начальной вершины АДД $L1\#1$ до вершины $ERR\#1$:

```
L1: (x > y);
```



```
L2: z = x - y;
L4: (z < 0);
```

Рассмотрим этот путь в качестве контрпримера. Вычислим сильнейшее постусловие $SP(P)$:

$$SP(\varphi_1, P) \equiv \top \wedge \psi_1 \wedge \psi_2 \wedge \psi_3 \equiv$$

$$\top \wedge \underbrace{x_1 > y_1}_{\psi_1} \wedge \underbrace{z_2 = x_1 - y_1}_{\psi_2} \wedge \underbrace{z_2 < 0}_{\psi_3} - UNSAT$$

4. Постусловие невыполнимо. Выполним уточнение набора предикатов π на основе интерполянтов Крейга:

$$\rho_1 \equiv x_1 > y_1$$

$$\rho_2 \equiv z_2 > 0$$

Новое множество предикатов $\pi_2' = \{p_1', p_2'\}, p_1' \equiv x > y, p_2' \equiv z > 0$.

5. Строим АДД, соответствующее новому набору предикатов $\pi' = \{x > y, z > 0\}$.

5.1. Обрабатываем переход

$$L1\#1: (L1, \top) \xrightarrow{x>y} L2\#1: (L2, \text{post}_{\pi_2'}(\top, x > y))$$

$$SP(\top, x > y) \equiv x_1 > y_1 - SAT(x_1 = 1, y_1 = 0)$$

$$p_1': \top \wedge x_1 > y_1 \wedge \neg(x_1 > y_1) - UNSAT$$

$$p_2': \top \wedge x_1 > y_1 \wedge \neg(z_1 > 0) - SAT\ (x_1 = 2, y_1 = 1, z_1 = -1)$$

Результат: $L2\#1: (L2, x > y)$.

5.2. Обрабатываем переход

$$L2\#1: (L2, x > y) \xrightarrow{z=x-y} L4\#1: (L4, \text{post}_{\pi_2'}(x > y, z = x - y))$$

$$SP(x > y, z = x - y) \equiv x_1 > y_1 \wedge z_2 = x_1 - y_1 - SAT(x_1 = 1, y_1 = 0, z_2 = 1)$$

$$p_1': x_1 > y_1 \wedge z_2 = x_1 - y_1 \wedge \neg(x_1 > y_1) - UNSAT$$

$p_2'\colon x_1 > y_1 \wedge z_2 = x_1 - y_1 \wedge \neg(z_2 > 0) - UNSAT$

Результат:$L4\#1\colon (L4, x > y \wedge z > 0)$.

5.3. Обрабатываем переход

$$L4\#1\colon (L4, x > y \wedge z > 0) \xrightarrow{!(z<0)} L5\#1\colon (L5, \text{post}_{\pi_2'}(x > y \wedge z > 0, !(z < 0)))$$

$$SP(x > y \wedge z > 0, !(z < 0)) \equiv x_1 > y_1 \wedge z_1 > 0 \wedge \neg(z_1 < 0) - SAT(x_1 = 1, y_1 = 0, z_2 = 1)$$

$$p_1'\colon x_1 > y_1 \wedge z_1 > 0 \wedge \neg(z_1 < 0) \wedge \neg(x_1 > y_1) - UNSAT$$

$$p_2'\colon x_1 > y_1 \wedge z_1 > 0 \wedge \neg(z_1 < 0) \wedge \neg(z_1 > 0) - UNSAT$$

Результат:$L5\#1\colon (L5, x > y \wedge z > 0)$.

5.4. Обрабатываем переход

$$L4\#1\colon (L4, x > y \wedge z > 0) \xrightarrow{z<0} ERR\#1\colon (ERR, \text{post}_{\pi_2'}(x > y \wedge z > 0, (z < 0)))$$

$$SP\big(x > y \wedge z > 0, (z < 0)\big) \equiv x_1 > y_1 \wedge z_1 > 0 \wedge z_1 < 0 - UNSAT$$

Ошибочное состояние недостижимо, результат: $ERR\#1\colon (ERR, \bot)$.

В результате успешного уточнения абстракции путь, на котором мы в первый раз нашли ошибку, больше не приводит к ошибочному состоянию в АДД.

5.5. Вычисляем переходы

$$L1\#1\colon (L1, \top) \xrightarrow{!(x>y)} L3\#1(L3, \top),$$

$$L3\#1(L3, \top) \xrightarrow{z=y-x} L4\#2\colon (L4, \top)$$

$$L4\#2\colon (L4, \top) \xrightarrow{!(z<0)} L5\#2\colon (L5, \top)$$

$$L4\#2\colon (L4, \top) \xrightarrow{(z<0)} ERR\#2\colon (ERR, \top)$$

$$ERR\#2\colon (ERR, \top) \xrightarrow{goto\ ERR} ERR\#3\colon (ERR, \top)$$

Вершина $ERR\#3$ покрыта вершиной $ERR\#2$. Завершаем построение АДД. Результирующее дерево показано на рис. 13.

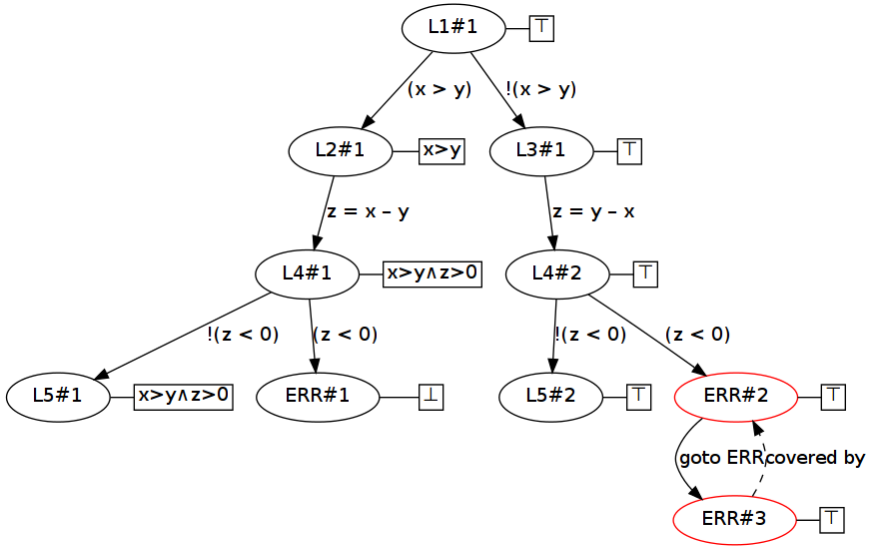

Рис. 13. Абстрактное дерево достижимости для набора предикатов π_2'.

6. Построенное АДД вновь не является безопасным по отношению к конфигурации $(ERR, \top)$, т.к. в нём присутствует вершина $ERR\#2(ERR, \top)$.

7. Соответствующий путь от начальной вершины АДД до вершины $ERR\#2$:

```
L1: !(x > y);
L3: z = y - x;
L4: (z < 0);
```

Построим сильнейшее постусловие пути:

$$SP(P) \equiv \top \wedge \underbrace{\neg(x_1 > y_1)}_{\psi_1} \wedge \underbrace{z_2 = y_1 - x_1}_{\psi_2} \wedge \underbrace{z_2 < 0}_{\psi_3} - UNSAT$$

8. Постусловие невыполнимо. Выполним уточнение набора предикатов π' на основе интерполянтов Крейга:

$$\rho_1 \equiv x_2 \leq y_2$$

$$\rho_2 \equiv z_2 \geq 0$$

Новое множество предикатов $\pi_2'' = \pi_2' \cup \{p_1'', p_2''\} = \{p_1', p_2', p_1'', p_2''\}$

$$p_1' \equiv x > y$$

$$p_2' \equiv z > 0$$

$$p_1'' \equiv x \leq y$$

$$p_2'' \equiv z \geq 0$$

9. Покажем, как будет построено поддерево АДД, существенно отличающееся от соответствующего поддерева в предыдущем АДД.

9.1. Переход $L1\colon (L1, \top) \xrightarrow{!(x>y)} L3\#1\colon (L3, \text{post}_{\pi_2''}(\top, !(x > y)))$

$SP(\top, !(x > y)) \equiv \top \wedge \neg(x_1 > y_1) - SAT(x_1 = 0, y_1 = 1)$

$p_1''\colon \top \wedge \neg(x_1 > y_1) \wedge \neg(x_1 \leq y_1) - UNSAT$

$p_2''\colon \top \wedge \neg(x_1 > y_1) \wedge \neg(z_1 \geq 0) - SAT\ (x_1 = 1, y_1 = 2, z_1 = -1)$

Результат: $L3\#1\colon (L3, x \leq y)$.

9.2. Переход $L3\#1\colon (L3, x \leq y) \xrightarrow{z=y-x} L4\#2\colon (L4, \text{post}_{\pi_2''}(x \leq y, z = y-x)$

$SP(x \leq y, z = y - x) \equiv x_1 \leq y_1 \wedge z_2 = y_1 - x_1 - SAT(x_1 = 1, y_1 = 0, z_2 = 1)$

$p_1''\colon x_1 \leq y_1 \wedge z_2 = y_1 - x_1 \wedge \neg(x_1 \leq y_1) - UNSAT$

$p_2''\colon x_1 \leq y_1 \wedge z_2 = y_1 - x_1 \wedge \neg(z_2 \geq 0) - UNSAT$

Результат: $L6\#2\colon (L6, x \leq y \wedge z \geq 0)$.

9.3. Переход

$L4\#2\colon (L4, x \leq y \wedge z \geq 0) \xrightarrow{!(z<0)} L5\#2\colon (L5, \text{post}_{\pi_2''}(x \leq y \wedge z \geq 0, !(z < 0)))$

$SP(x \leq y \wedge z \geq 0, !(z < 0)) \equiv x_1 \leq y_1 \wedge z_1 \geq 0 \wedge \neg(z_1 < 0) - SAT(x_1 = z_1 = 1, y_1 = 0)$

$p_1''\colon x_1 \leq y_1 \wedge z_1 \geq 0 \wedge \neg(z_1 < 0) \wedge \neg(x_1 \leq y_1) - UNSAT$

$p_2''\colon x_1 \leq y_1 \wedge z_1 \leq 0 \wedge \neg(z_1 < 0) \wedge \neg(z_1 \geq 0) - UNSAT$

Результат: $L5\#2\colon (L5, x \leq y \wedge z \geq 0)$.

9.4. Переход

$L4\#2\colon (x \leq y \wedge z \geq 0) \xrightarrow{z<0} ERR\#2\colon (ERR, \text{post}_{\pi_2''}(x \leq y \wedge z \geq 0, (z < 0)))$

$SP(x \leq y \wedge z \geq 0, (z < 0)) = x_1 \leq y_1 \wedge z_1 \geq 0 \wedge z_1 < 0 - UNSAT$

Результат: $ERR\#2\colon (ERR, \bot)$.

Результирующее АДД показано на Рис. 14.

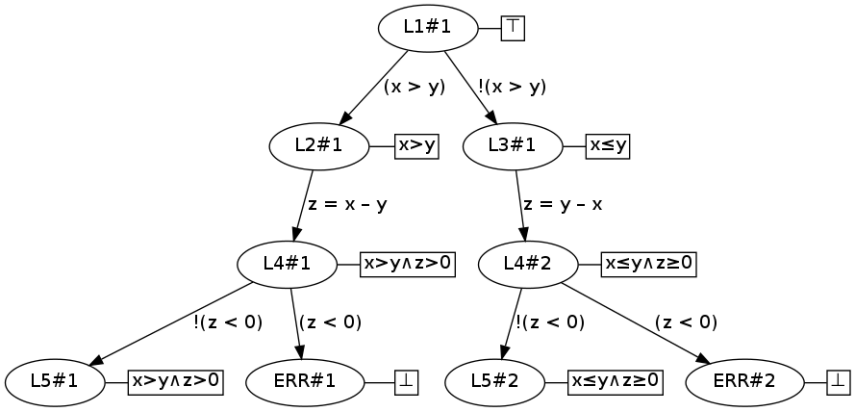

Рис.14. АДД для набора предикатов π_2''.

10. Полученное АДД безопасно по отношению к конфигурации $(ERR, \top)$, т.к. обе вершины $ERR\#1$ и $ERR\#2$, соответствующие ошибочной метке ERR в этом дереве, имеют невыполнимые регионы $\bot$.

Таким образом метод CEGAR в рассмотренном варианте позволил доказать недостижимость ошибочной метки в программе из примера 1 за 3 итерации рассмотренного цикла верификации (шаги 1-4, 5-8, 9-10).

8 Оптимизации и расширения метода CEGAR

Мы рассмотрели применение метода CEGAR для построения и уточнения предикатной абстракции простейшей программы на языке C. В этой программе определена всего одна функция (`main`), используются только глобальные переменные типа `int`. Кроме этого в рассмотренном примере отсутствуют циклы. Рассмотрим возможности ослабления данных ограничений и применяемые при этом оптимизации построения абстракции (АДД) программы.

8.1 Ленивая абстракция

В рассмотренном примере 1 (из предыдущего раздела) можно заметить, что процесс построения предикатной абстракции можно несколько упростить, если

- не перестраивать заново те части абстрактного дерева достижимости, где уточнение абстракции не повлияет на достижимость ошибочной инструкции (недостижимость уже доказана существующими предикатами);
- вычислять значения вновь вводимых при уточнении предикатов только в тех вершинах, для которых эти предикаты были получены, то есть только там, где их значения будут существенно использоваться в доказательстве исключения из АДД невыполнимого ошибочного пути.

В нашем примере это будет означать следующее:

- не пересчитывать значение предикатов $b_1' \equiv x \leq y$ и $b_2' \equiv z \geq 0$ в вершинах $L1\#1, L2\#1, L4\#1, L5\#1$ и $ERR\#1$, т.к. недостижимость ошибочной инструкции из этих вершин перед добавлением предикатов b_1' и b_2' уже доказана предикатами $b_1 \equiv x > y$ и $b_2 \equiv z > 0$. По сути это означает, что при уточнении абстракции можно полностью оставить без изменения всё АДД, кроме поддерева с корнем в вершине $L3\#1$.
- не пересчитывать значение предикатов b_1и b_2в вершинах $L3\#1, L4\#2, L5\#1, L5\#2, ERR\#1$ и $ERR\#2$, потому что они были получены для вершин $L2\#1$ и $L4\#1$, и для доказательства недостижимости ошибочной метки их значения существенно требуются только в этих вершинах.

Абстракция, которая строится с учетом предложенных оптимизаций, имеет таким образом не один фиксированный набор предикатов для всех вершин АДД, а свой набор предикатов для каждой вершины. Предикаты в таких наборах называют *локальными*, а соответствующий метод построения АДД – *ленивой абстракцией (*от англ. *lazy abstraction)*. Именно этот метод построения АДД реализован в инструментах CPAchecker и BLAST (который поэтому называется Berkeley *Lazy Abstraction* Software verification Tool).

8.2 Программы с циклами и крупноблочное кодирование

Для программ, содержащих циклы, метод CEGAR используется аналогично рассмотренному примеру (в силу того, что программа с циклами является простейшей, а метод CEGAR был рассмотрен для любой простейшей программы). Как было отмечено, конечность абстрактного дерева достижимости обеспечивается конечностью множества абстрактных состояний программы и применением операции покрытия вложенных состояний.

В примере 1 при построении АДД по заранее заданному набору предикатов (в разделе 5) выполнялось покрытие вершины $L4\#2\colon (L4, \top)$ другой ранее построенной вершиной $L4\#1\colon (L4, \top)$ с таким же абстрактным состоянием. В этом случае вложенность состояний была установлена тривиально. В общем случае для предикатной абстракции покрытие проверяется как импликация, т.е. состояние φ_1 покрывает состояние φ_2, если $\varphi_2 \rightarrow \varphi_1$. В случае с циклом, как правило, какая-либо вершина, соответствующая предыдущей итерации цикла будет покрывать вершину для того же места в программе на следующей его итерации, если в обеих вершинах будет выполнен один и тот же набор предикатов. Этот набор предикатов может, в частности, содержать инвариант цикла[3].

Рассмотрим пример простейшей программы, представленной на рис. 15. На рис. 16 для этой программы показан абстрактный граф достижимости, построенный для набора предикатов $\pi = \{b_1 \equiv z \geq 0, b_2 \equiv x > y, b_3 \equiv x \leq y\}$.

[3]Инвариантом в программировании называется логическое выражение, истинное после каждого прохода тела цикла и перед началом выполнения цикла и зависящее от переменных, изменяющихся в теле цикла.

```
void main() {
     int x = nondet;
     int y = nondet;
     int z = nondet;
L1:  z = 0;
LOOP:if(x > 0) {
L2:       if (y > 0) {
L3:            if (x > y){
L4:                 z = x – y;
L5:                 x = z;
               } else {
L6:                 z = y – x;
L7:                 y = z;
               }
L8:            goto LOOP;
          }
     }
L9:  if (z < 0)
ERR:      goto ERR;
L10:}
```

Рис. 15. Пример простейшей программы 2

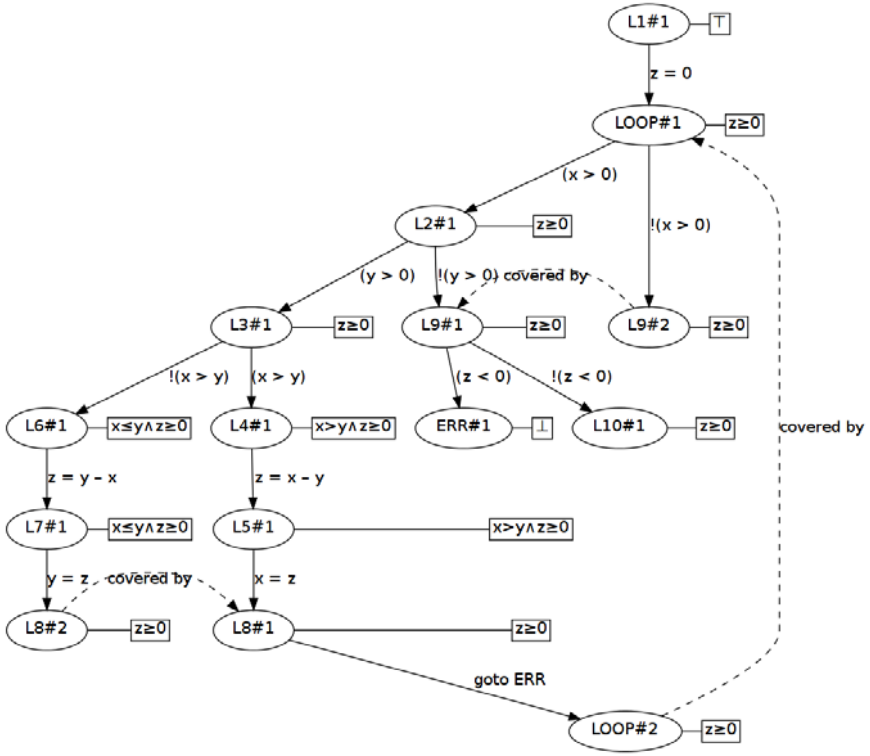

Рис. 16. Абстрактный граф достижимости для простейшей программы из примера 2

На этом примере видно, что по существу для обеспечения конечности абстрактного дерева достижимости и доказательства недостижимости ошибочной метки важны лишь состояния абстракции в вершинах, помеченных абстрактными состояниями$(LOOP, z \geq 0)$ и $(ERR, \bot)$. Также видно, что предикат $b_1 \equiv z \geq 0$ наиболее важен для доказательства недостижимости. Остальные вершины представляют по сути некоторые промежуточные состояния абстракции, а предикаты b_2 и b_3служат для представления некоторой промежуточной информации. Эти рассуждения наталкивают на мысль об оптимизации процесса построения предикатной абстракции за счет склеивания промежуточных вершин.

В инструменте CPAchecker при построении предикатной абстракции применяется крупноблочное кодирование, или LBE (кодирование большими блоками, Large Block Encoding) [42]. Его суть заключается в склеивании вершин на линейных участках графа достижимости и кодировании сильнейшего постусловия сразу для нескольких соответствующих последовательных переходов. В примере 2 можно склеить, к примеру,

вершины $L4\#1\colon (L4, x > y \wedge z \geq 0)$, $L5\#1\colon (L5, x > y \wedge z \geq 0)$ и $L8\#1 (L8, z \geq 0)$. В таком случае сильнейшее постусловие $SP(z \geq 0, (x > y); z = x - y;\ x = z) = z_1 \geq 0 \wedge x_1 > y_1 \wedge z_2 = x_1 - y_1 \wedge x_2 = z_2$.

$$b_1\colon z_1 \geq 0 \wedge x_1 > y_1 \wedge z_2 = x_1 - y_1 \wedge x_2 = z_2 \wedge \neg(z_2 \geq 0) - UNSAT$$

В результате получим дугу $L3\#1\colon (L3, z \geq 0) \xrightarrow{x>y, z=x-y, x=z} L4 - L8\#1\colon (L8, z \geq 0)$. Метка $L4 - L8\#1$ дана новой вершине, полученной в результате склеивания. Аналогично можно склеить вершины $L6\#1\colon (L6, x \leq y \wedge z \geq 0)$, $L7\#1\colon (L7, x \leq y \wedge z \geq 0)$ и $L8\#2\colon (L8, z \geq 0)$. Пометим новую вершину, полученную в результате этой склейки меткой $L6 - L8\#1$. После этого предикаты $b_2 \equiv x \leq y$ и $b_3 \equiv x > y$ окажутся ненужными для доказательства недостижимости в данном примере. Последовательности инструкций $x > y,\ z = x - y,\ x = z$ и $!(x > y),\ z = y - x,\ y = z$ называют блоками кодирования. Полученный граф достижимости показан на рис. 17.

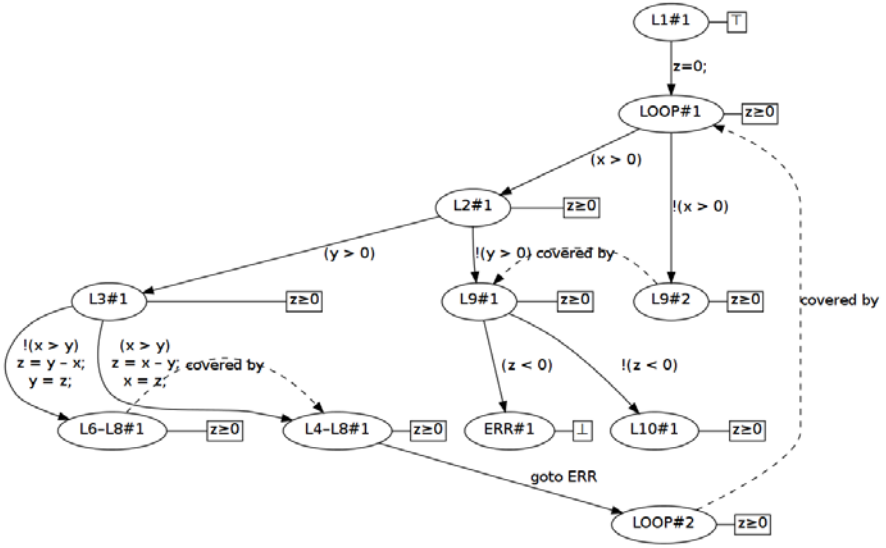

Рис. 17. Граф достижимости после склейки вершин на линейных участках

Склеивать можно не только вершины на линейных участках графа достижимости, но и вершины, соответствующие одному и тому же месту в программе, в котором происходит слияние потока управления. В нашем примере такими вершинами являются вершины $L6 - L8\#1$ и $L4 - L8\#1$. Склеивание таких вершин называется *слиянием* (от англ. *merge*). При слиянии необходимо также осуществить объединение двух альтернативных блоков кодирования, в нашем случае между вершинами $L3\#1$ и $L4 - L8\#1$, а также между $L3\#1$ и $L6 - L8\#1$. Для этого можно использовать дизъюнкцию. Пометим новую вершину $L4 - L8||L6 - L8\#1$ и посчитаем абстракцию в ней через сильнейшее постусловие:

$$SP(z \geq 0, x > y, z = x - y, x = z\ ||!(x > y), z = y - x, y = z) =$$

$$(z_1 \geq 0 \wedge x_1 > y_1 \wedge z_2 = x_1 - y_1 \wedge x_2 = z_2) \vee (z_1 \geq 0 \wedge \neg(x_1 > y_1) \wedge z_2 = y_1 - x_1 \wedge y_2 = z_2)$$

$$b_1\colon ((z_1 \geq 0 \wedge x_1 > y_1 \wedge z_2 = x_1 - y_1 \wedge x_2 = z_2) \vee (z_1 \geq 0 \wedge \neg(x_1 > y_1) \wedge z_2 = y_1 - x_1 \wedge y_2 = z_2)) \wedge$$

$$\wedge \neg(z_2 \geq 0) - UNSAT$$

В результате получаем дугу $L3\#1\colon (L3, z \geq 0) \xrightarrow{x>y, z=x-y, x=z\ ||!(x>y), z=y-x, y=z} L4 - L8||L6 - L8\#1 (L8, z \geq 0)$.

Применяя последовательно аналогичные операции склейки и слияния вершин, получим абстрактный граф достижимости для крупноблочного кодирования с пересчётом состояния абстракции только в заголовке цикла и на ошибочной метке, показанный на рисунке 10 (для получения этого графа необходимо также предварительно дублировать вершину $L2\#1$).

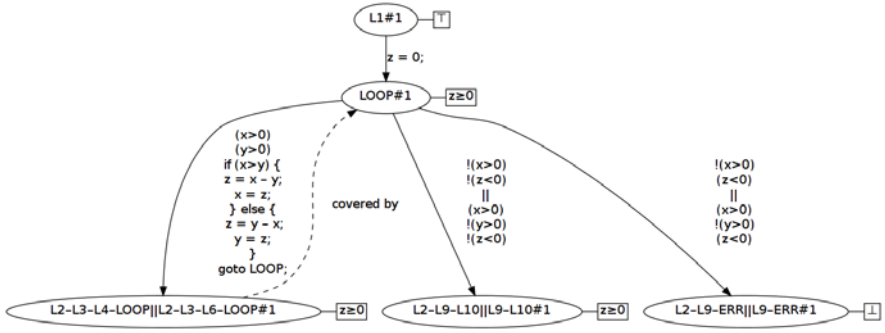

Рис.10. Граф достижимости для крупноблочного кодирования со слияниями

В инструменте CPAchecker размер блока кодирования является настраиваемым, как вручную, так и динамически во время построения абстракции. Подробнее об этом написано в статье [43]. Построение абстракции при крупноблочном кодировании в CPAchecker осуществляется аналогично рассмотренному ранее, при помощи вычисления абстрактных постусловий и решения задач о выполнимости формул. CPAchecker сразу строит абстракцию с заданным размером блока, не делая никаких последовательных преобразований АДД, то есть осуществляет склейку и слияние вершин «на лету». Размер блока кодирования в этом инструменте можно ограничивать максимальным количеством объединяемых в один переход инструкций или условием пересчёта абстракции (например, только на заголовке цикла или на заголовке цикла и операторах ветвления).

8.3 Поддержка указателей

В инструменте BLAST помимо типа данных `int` реализована поддержка некоторых выражений с указателями на основе анализа алиасов. В программировании словом *алиасинг* (от англ. *alias* – имя, прозвище) описывается ситуация, при которой какая-либо ячейка с данными в памяти программы оказывается доступна в исходном тексте (коде) этой программы под различными обозначениями (именами). Таким образом, изменение данных с использованием одного из таких обозначений неявно ведет к изменению значений, доступных по всем остальным обозначениям той же ячейки памяти. В результате алиасинг значительно осложняет как анализ и оптимизацию, так и верификацию соответствующих программ. Анализ алиасов предназначен для извлечения из кода программы полезной информации о присутствующем в ней алиасинге.

Например, пусть в программе имеются объявления:

```
int a;
int *p;
```

Тогда в точке программы, где выполнено условие $p == \&a$, разыменование $*p$ будет являться алиасом переменной a.

Другой пример: $*(q+1)$ может являться алиасом для $b[1]$, если $q == b$.

Если в результате анализа алиасов оказалось, что для некоторого выражения в программе найденное множество возможных значений указателя включает лишь один вариант (иными словами данное выражение всегда адресует один и тот же объект), то говорят, что такое выражение является *обязательным алиасом* (от англ. *must-alias*) этого объекта. В других случаях говорят о *возможных алиасах* (от англ. *may-aliases*).

Для методов статической верификации, которые сами по себе анализируют различные пути выполнения программы, бывает достаточно не чувствительного к потоку управления метода анализа возможных алиасов. В инструменте BLAST информация об алиасах используется при генерации ограничений, которые на состояние памяти программы накладывает операция присваивания. При анализе контрпримера или вычислении абстрактного постусловия для каждого возможного алиаса цели присваивания генерируется проверка, является ли его адрес равным адресу цели. В зависимости от результата этой проверки с помощью логических формул выражается либо обновление соответствующего выражения-алиаса, либо сохранение прежнего значения этого выражения. Выборка возможных алиасов для каждой цели присваивания, таким образом, производится на основе результатов работы нечувствительного к потоку анализа возможных алиасов. В качестве нечувствительного к потоку управления алгоритма анализа алиасов BLAST используется алгоритм на основе BDD, описанный в [44].

Пусть, например, в некоторой программе переменные a, pa и pb имеют тип `int`, `int *` и `int *` соответственно. Анализ алиасов показал, что выражение $*pb$ является возможным алиасом выражения $*pa$ и в момент генерации формулы перехода для операции присваивания выражениям $pa,\ pb,\ a,\ *pa,\ *pb$ соответствовали следующие индексы в SSA-представлении: $\{pa \to i, pb \to j, a \to k, *pa \to l, *pb \to m\}$. Тогда для операции присваивания `*pa = a` BLAST сгенерирует формулу перехода:

$$*pa_{l+1} = a_k \wedge \left(\left(pb_j = pa_i \wedge *pb_{m+1} = a_k\right) \vee \left(\neg\left(pb_j = pa_i\right) \wedge *pb_{m+1} = *pb_m\right)\right)$$

Такой подход к использованию анализа возможных алиасов для генерации ограничений в формулах пути был расширен в [41] на анализ алиасов с участием структурных типов. В статье приводится алгоритм генерации ограничений, соответствующих присваиваниям и условиям в ветвлениях рассматриваемой программы, с учетом наличия структурных типов. В статье вводятся ограничения на точность анализа в случае рекурсивных структур (к примеру, связных списков) и сильно вложенных указателей на структуры (переменные, генерируемые при анализе, не могут отличаться от встречаемых в программе более, чем на фиксированное количество разыменований). Для целей присваивания обновляются не только непосредственно выражение в левой части и его алиасы (в случае равенства их адресов), но и их поля, если это структурные типы, и поля тех структур, на которые они указывают, если это указатели на структуру. Обновляемые таким образом значения генерируются рекурсивно не более чем на заданную глубину, которая измеряется количеством разыменований.

8.4 Анализ реальных программ на C

Реальные программы, в отличие от простейших

- состоят из большого числа функций, принимающих параметры и возвращающих значения;
- используют глобальные и локальные переменные различных типов,
- размещают составные объекты в статической и динамической памяти, в том числе структуры, массивы и объединения;
- могут содержать все конструкции (операторы и выражения) языка C, в том числе нелинейные и побитовые операции.

Кроме этого реальные инструменты статической верификации (в том числе BLAST и CPAchecker) позволяют использовать недетерминированные значения в любом месте программы, где допустим вызов функции без параметров.

Поэтому при верификации реальных программ инструменты используют ряд приёмов и упрощающих предположений.

BLAST и CPAchecker перед верификацией применяют к данной на вход программе некоторые упрощающие преобразования, которые, в частности, позволяют свести все виды циклов, короткую логику и тернарный оператор к использованию только операторов `if` и `goto`. В BLAST эти преобразования выполняются с использованием инструмента CIL [45].

Поддержка вызовов функций в BLAST и CPAchecker реализована с помощью встраивания тела функции в месте её вызова. При этом каждому формальному параметру функции ставится в соответствие переменная, которой присваивается значение соответствующего фактического параметра в месте вызова функции. Возвращаемое значение функции присваивается временной переменной непосредственно перед выходом из функции. При такой реализации BLAST и CPAchecker в случае рекурсивных функций поддерживают только небольшую конечную глубину рекурсивных вызовов.

Некоторые инструменты, использующие метод CEGAR, например, Yogi [46] реализуют возможности так называемого *обобщения* (от англ. *summarization*) вызовов функций. Такие инструменты в месте вызова функции сначала пытаются переиспользовать некоторую информацию о ней, собранную во время анализа тела функции. Например, Yogi анализирует вызовы функций в три этапа.

- Сначала он пытается использовать некоторое приближение множества переменных, значения которых могут изменяться в теле вызываемой функции. Информацию об этих переменных Yogi получает, в частности, из результатов анализа возможных алиасов. Yogi использует полученное приближение в предположении, что все переменные, возможно изменяемые вызываемой функцией, принимают произвольные значения (`nondet`).

- Если первое обобщение оказывается неудачным, что происходит в результате построения ложного контрпримера (проверку контрпримера Yogi выполняет с помощью интерпретации программы), инструмент пытается использовать следующее обобщение. Оно состоит из двух пар регионов – (φ_1, ψ_1) и (φ_2, ψ_2). Первая пара представляет собой *обобщение достижимости* (от англ. *must-summary*). Его смысл в том, что для каждого состояния, заданного φ_1 (для любого места вызова функции) существует выполнение вызываемой функции, приводящее в какое-либо состояние, заданное ψ_1. Вторая пара – это *обобщение недостижимости* (от англ. *not-may summary*), оно означает, что не существует выполнений вызываемой функции, приводящих из какого-либо состояния, заданного φ_2 в какое-либо состояние, заданное ψ_2. Предположим, что регион абстрактного состояния программы непосредственно перед вызовом функции равен α, а непосредственно после возврата из неё – β. Тогда если выполнены условия $\varphi_1 \to \alpha$ и $\psi_1 \cap \beta \neq \emptyset$, то это означает, что можно попытаться построить уточненный контрпример, включающий вызов функции, в два этапа. Сначала построить такой контрпример, чтобы перед вызовом функции был выполнен предикат φ_1, а после её вызова – предикат ψ_1. Затем построить контрпример выполнения тела функции, который существует по определению пары регионов (φ_1, ψ_1). Если же выполнены условия $\alpha \to \varphi_2$ и $\beta \to \psi_2$, то можно исключить построенный ложный контрпример, уточнив регион α соответствующего абстрактного состояния с помощью конъюнкции с регионом φ_2 (конъюнкция соответствует пересечению абстрактных состояний данных). По определению пары (φ_2, ψ_2) при условии $\beta \to \psi_2$ все состояния, заданные β окажутся недостижимыми из уточненного абстрактного состояния. Использование обобщений, таким образом, позволяет переиспользовать информацию о достижимости, полученную ранее при анализе тела функции, оптимизируя построение контрпримера и позволяя уточнять абстракцию без вызова интерполирующего SMT-решателя.
- Если использовать второе обобщение не удается, Yogi подставляет и анализирует тело вызываемой функции в месте вызова. При этом могут уточняться имеющиеся для этой функции пары регионов (φ_1, ψ_1) и (φ_2, ψ_2).

Для поддержки переменных различных типов в инструментах верификации используются два основных подхода – приближение значений всех переменных математическими целыми и вещественными числами, независимо от их типа, и точное кодирование значений переменных и операций над ними с помощью битовых векторов различного фиксированного размера. BLAST использует первый подход. В CPAchecker были реализованы оба подхода.

Решение и интерполяция логических формул в теории битовых векторов поддерживается некоторыми SMT-решателями, например, MathSAT 5 [47].

Использование битовых векторов позволяет также точно кодировать большинство нелинейных и побитовых операций над переменными различных типов. При использовании целых и вещественных чисел в BLAST и CPAchecker нелинейные и побитовые операции кодируются неинтерпретируемыми функциями, что снижает точность анализа.

Поддержка структур и указателей в BLAST реализована на основе анализа алиасов. CPAchecker поддерживает представление структур в виде битовых векторов с возможностью обращения к полям структуры, как к выделенным битовым подвекторам. Поддержка массивов в BLAST и CPAchecker на момент написания статьи не была реализована (используется неточная эвристика на основе неинтерпретируемых функций).

9 Использование решателей

Как было отмечено в части, описывающей процесс построения абстракции программы, для автоматического решения задачи о выполнимости логической формулы на практике используются специальные инструменты, называемые решателями.

В широком смысле *решающие процедуры* (от англ. *decision procedures*) или решатели, (от англ. *solvers*) – это алгоритмы, которые приняв на вход проблему разрешимости, то есть вопрос, сформулированный в рамках какой-либо формальной системы (аксиоматической теории), и требующий ответа «да» или «нет»; и выдают на выходе соответствующий корректный результат. В узком смысле в рамках данной статьи нас интересуют решатели для алгоритмически разрешимых задач, сформулированных в рамках теорий первого порядка, и используемых на практике в областях верификации, доказательства корректности, оптимизации и др. Возможности многих технологий в этих областях ограничены возможностями используемых решателей. Поэтому последние остаются объектом многих активных исследований во всём мире, как в академической среде, так и в промышленности.

Решающие процедуры можно разделять по поддерживаемым ими теориям. Решатели для логики высказываний, т.е. классической логики нулевого порядка, которые по сути являются инструментами для решения задачи выполнимости булевых формул, называются SAT-решателями. Решатели для формул классической логики первого порядка с равенством, заданных в рамках комбинации некоторых аксиоматических теорий, называются SMT-решателями. Такие решатели могут поддерживать только формулы нулевого порядка (без кванторов). Среди наиболее часто используемых на практике теорий – вещественная и целочисленная линейная арифметика, неинтерпретируемые функции, массивы и битовые векторы.

Практически все современные решатели для пропозициональной (логической) части формулы используют либо схему DPLL (Davis-Putnam-Logemann-Loveland), основанную на поиске с возвратом, для решения задачи в виде КНФ, либо основаны на *суперпозиционном исчислении* (от англ. *superposition calculus*), расширении резолютивного вывода [48]. Алгоритм DPLL описан, например, в книге [49]. Для взаимодействия с решающими процедурами и для подформул, заданных в рамках какой-либо теории, используются в основном техники комбинирования Нельсона-Оппена (Nelson-Oppen Combination Procedure), пропозиционального кодирования (от англ. propositional encodings) и отложенного комбинирования теорий (DTC, Delayed Theory Combination). Техники комбинирования решателей Нельсона-Оппена и пропозиционального кодирования приводится, например, в книге [49]. Отложенное комбинирование теорий предложено в статье [50] и используется в решателе MathSAT.

Также некоторые решатели могут в результате своей работы помимо основного ответа «выполнимо»/«невыполнимо», давать также некоторую дополнительную информацию, например, для выполнимой формулы – её модель, для невыполнимой – опровержение (например, в виде дерева вывода тождественной лжи), набор дизъюнктов, использованных для доказательства невыполнимости (англ. unsatisfiable core), для невыполнимой конъюнкции пары формул – интерполянт Крейга (Craig interpolant) [40].

Решатели, позволяющие находить интерполянты Крейга, называются интерполирующими процедурами, или интерполирующими решателями, или просто интерполяторами. В существующих инструментах интерполянты строятся из дерева вывода для доказательства невыполнимости. Некоторые методы построения интерполянтов из дерева вывода для комбинации определённого класса теорий описаны в статье [51]. Они требуют предварительного построения частичных интерполянтов для используемых теорий. Методы их построения для вещественной линейной арифметики предлагаются в статье [52], а для теории неинтерпретируемых функций – в статье [53]. На данных методах основывается интерполирующий решатель CSIsat [54] для логических формул без кванторов в рамках теорий вещественной линейной арифметики и неинтерпретируемых функций. Этот интерполирующий решатель используется инструментом верификации BLAST[4]. Другой интерполирующий решатель для формул без кванторов, MathSAT, поддерживающий также целочисленную линейную арифметику, использует свой метод построения и комбинирования частичных интерполянтов, описанный в статье [55] и диссертации [56]. MathSAT используется как основной интерполирующий решатель в инструменте CPAchecker [34]. SLAM2 использует для проверки выполнимости пути и извлечения предикатов SMT-решатель Z3 [57], о чём говорится, например, в статье [58].

Таким образом, практическая эффективность работы современных инструментов верификации (SLAM2, BLAST, CPAchecker) во многом

определяется эффективностью существующих на данный момент решающих процедур, особенно SMT-решателей с поддержкой теорий вещественной и целочисленной линейной арифметики и неинтерпретируемых функций, а также соответствующих интерполяторов.

10 Заключение

Мы подробно рассмотрели метод CEGAR для декартовой предикатной абстракции, применив его для доказательства недостижимости ошибочной метки в коде примера простейшей программы на языке C. На рассмотренном примере мы показали такие этапы работы инструментов статической верификации BLAST и CPAchecker, основанных на CEGAR, как построение графа потока управления программы, обход этого графа, построение абстрактного дерева достижимости, в том числе вычисление сильнейших постусловий и регионов абстрактных состояний, а также процесс уточнения абстракции по ложному контрпримеру с использованием интерполяции Крейга. В конце мы также рассказали о некоторых наиболее распространённых модификациях и оптимизациях, используемых при реализации подхода CEGAR в современных инструментах.

Инструменты, основанные на методе CEGAR, занимают первые места в международных соревнованиях по верификации Си-программ [59], [60]. Их текущее состояние позволяет успешно решать задачи верификации моделей аппаратных систем [61], драйверов устройств в операционных системах Windows [31, 58] и Linux [18, 35, 36, 62, 63, 64], а также систем, образующих так называемые product lines [65, 66].

Тем не менее, в реальном коде встречается немало сложностей, которые вызывают проблемы даже у самых современных инструментов, реализующих подход CEGAR. Открытые направления для развития включают в себя:

- поддержку нелинейной арифметики, возможно, с помощью комбинирования предикатной абстракции с другими видами анализа [67, 68];
- поддержку битовой арифметики, побитовых операций, учета возможных арифметических переполнений и преобразования типов;
- поддержку преобразования типов и объединений (union);
- анализ указателей, в том числе поддержку алиасинга, массивов, адресной арифметики, конструкций вида `container_of` [69] и др.;
- анализ сложных типов данных, таких как ссылочные структуры, хэш-таблицы и др.;
- анализ многопоточных программ.

Литература

[1]. Dershowitz N. Software horror stories. URL: http://www.cs.tau.ac.il/~nachumd/horror.html

[2]. Turing A. M. On Computable numbers, with an application to the Entscheidungsproblem // Proceedings of the London Mathematical Society. 1936. pp. 230—265.

[3]. Floyd R. Assigning Meanings to Programs // Mathematical Aspects of Computer Science. 1967. pp. 19—32.

[4]. Hoare C. An Axiomatic Basis for Computer Programming // Communications of the ACM. 1969. vol. 12. pp. 576—580.

[5]. Dijkstra E. A Discipline of Programming // Prentice-Hall, 1976.

[6]. Millo R. D., Lipton R., Perlis A. Social Processes and Proofs of Theorems and Programs // Communications of the ACM. 1979. vol. 22. pp.271—280.

[7]. Nelson G. Techniques for Program Verification // Tech. Rep. CSL81-10: Xerox Palo Alto Research Center, 1981.

[8]. Nelson G., Oppen D. Fast Decision Procedures Based on Congruence Closure // Journal of the ACM. 1980. vol. 27. pp. 356—364.

[9]. Shostak R. Deciding Combinations of Theories // Journal of the ACM. 1984. vol. 31. pp. 1—12.

[10]. Clarke E. M., Emerson E. A. Synthesis of Synchronization Skeletons for Branching Time Temporal Logic // Logic of Programs. 1981. vol. 131. pp. 52—71.

[11]. Queille J., Sifakis J. Specification and Verification of Concurrent Systems in Cesar // Fifth International Symposium on Programming / Ed. by M. Dezani-Ciancaglini, U. Montanari. Lecture Notes in Computer Science. Springer-Verlag, 1981. pp. 337—351.

[12]. Vardi M., Wolper P. Reasoning about Infinite Computations // Information and Computation. 1994. vol. 115. pp. 1—37.

[13]. Pnueli A. The Temporal Logic of Programs // Proceedings of the 18th Annual Symposium on Foundations of Computer Science. IEEE Computer Society Press, 1977. pp. 46—57.

[14]. Emerson E. Temporal and Modal Logic // Handbook of Theoretical Computer Science / Ed. by J. van Leeuwen. Elsevier Science Publishers, 1990. vol. B. pp. 995—1072.

[15]. Khedker Uday P., Sanyal Amitabha, Karkare Bageshri. Data Flow Analysis: Theory and Practice // CRC Press (Taylor and Francis Group), 2009.

[16]. D'Silva V., Kroening D., Weissenbacher G. A Survey of Automated Techniques for Formal Software Verification // Computer-Aided Design of Integrated Circuits and Systems. 2008. vol. 27, no. 7. pp. 1165—1178. On IEEE Transactions.

[17]. Jhala R., Majumdar R. Software model cheking // ACM Computing Surveys. 2009.

[18]. Д. Бейер, А.К. Петренко. Верификация драйверов операционной системы Linux // Труды Института системного программирования РАН, том 23, ISSN 2220-6426 (Online), ISSN 2079-8156 (Print), 2012 г. стр. 405-412. DOI: 10.15514/ISPRAS-2012-23-23.

[19]. Vladimir Nesov. Automatically Finding Bugs in Open Source Programs // Third International Workshop on Foundations and Techniques for Open Source Software Certification. OpenCert 2009 vol. 20 2009. pp.19—29.

[20]. Dawson Engler, Benjamin Chelf, Andy Chou. Checking system rules using system-specific, programmer-written compiler extensions // Proceedings of the 4th conference

on Symposium on Operating System Design & Implementation vol. 4 OSDI'00. 2000. pp. 1—16.
[21]. Julia L. Lawall, Julien Brunel, Nicolas Palix, Rene Rydhof Hansen, Henrik Stuart, Gilles Muller. WYSIWIB: A declarative approach to finding API protocols and bugs in Linux code // DSN'09 – The 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. 2009. pp. 43—52.
[22]. Арутюн Аветисян, Алексей Бородин. Механизмы расширения системы статического анализа Svace детекторами новых видов уязвимостей и критических ошибок // Труды Института системного программирования РАН, том 21, ISSN 2220-6426 (Online), ISSN 2079-8156 (Print). 2011. стр. 39-54.
[23]. В.Н. Игнатьев. Использование легковесного статического анализа для проверки настраиваемых семантических ограничений языка программирования // Труды Института системного программирования РАН, том 22, ISSN 2220-6426 (Online), ISSN 2079-8156 (Print). 2012. стр. 169-188. DOI: 10.15514/ISPRAS-2012-22-11.
[24]. Biere A., Cimatti A, Clarke E., Strichman O., Zhu Y. Bounded model checking // Advances in Computers. 2003.
[25]. Clarke E., Grumberg O., Jha S., Lu Y., Veith H. Counterexample-Guided Abstraction Refinement for Symbolic Model Checking // Journal of the ACM. 2003.
[26]. Thomas Donald, Moorby Phillip. The Verilog Hardware Description Language // Norwell, MA.: Kluwer Academic Publishers.
[27]. Clarke E., Kroening D., Lerda F. A tool for checking ANSI-C programs // Tools and Algorithms for the Construction and Analysis of Systems. 2004.
[28]. Ivancic F., Yang Z., Ganai M.K., Gupta A., I. Shlyakhter, Ashar P. F-soft: Software verification platform // Computer Aided Verification. vol. 3576 of Lecture Notes in Computer Science. Springer, 2005. pp. 301—306.
[29]. Post H., Sinz C., Merz F., Gorges T., Kropf T. Linking functional requirements and software verification // 17th IEEE International Requirements Engineering Conferene. 2009. pp. 295—302.
[30]. Donaldson A. F., Kroening D., Ruemmer P. Automatic analysis of DMA races using model checking and k-induction // Formal Methods in System Design. 2011. vol. 39. pp. 83—113.
[31]. Ball T., Bounimova E., Levin V., Kumar R., Lichtenberg J. The Static Driver Verifier Research Platform // Computer Aided Verification. 2010.
[32]. D. Beyer, T. A. Henzinger, R. Jhala, R. Majumdar. The software model checker BLAST: Applications to software engineering // Int. J. Softw. Tools Technol. Transf., 2007. vol. 9, № 5, ISSN 1433-2779. Springer-Verlag, Berlin, Heidelberg. pp. 505—525.
[33]. E. Clarke, D. Kroening, N. Sharygina, K. Yorav. SATABS: SAT-based Predicate Abstraction for ANSI-C // Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005), 2005. Lecture Notes in Computer Science, Springer Verlag. vol. 3440. ISBN 3-540-25333-5. pp.570—574.
[34]. D. Beyer, M. E. Keremoglu. CPAchecker: a tool for configurable software verification // Proceedings of the 23rd international conference on Computer aided verification. 2011. ISBN 978-3-642-22109-5. Springer-Verlag. pp. 184—190.
[35]. В.С. Мутилин, Е.М. Новиков, А.В. Страх, А.В. Хорошилов, П.Е. Швед. Архитектура Linux Driver Verification. // Труды Института системного программирования РАН, том 20, стр. 163—187, 2011.
[36]. A. Khoroshilov, V. Mutilin, E. Novikov, P. Shved, A. Strakh. Towards an Open Framework for C Verification Tools Benchmarking // Proceedings of the Eighth International Andrei Ershov Memorial Conference “Perspectives of Systems Informatics” (PSI 2011), pp. 82—91, 2011.
[37]. S. Graf, H. Saïdi. Construction of Abstract State Graphs with PVS // Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel. pp. 72—83.
[38]. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G. Lazy abstraction // Proc. POPL. ACM, New York. pp. 58—70, 2002.
[39]. Cytron, R., Ferrante, J., Rosen, B.K.,Wegman,M.N., Zadek, F.K. Efficiently computing static single-assignment form and the program dependence graph // ACM Trans. Program. Languages Systems 13(4), pp. 451—490. 1991.
[40]. W. Craig. Linear reasoning // J. Symbolic Logic. 1957. vol. 22, pp. 250—268.
[41]. T. A. Henzinger, K. L. McMillan, R. Jhala, R. Majumdar. Abstractions from Proofs. // POPL 2004.
[42]. D. Beyer, A. Cimatti, A. Griggio, M.E. Keremoglu, R. Sebastiani. Software Model Checking via Large-Block Encoding // Proc. FMCAD, pp. 25—32. IEEE, 2009.
[43]. D. Beyer, M. E. Keremoglu, P. Wendler. Predicate abstraction with adjustable-block encoding // Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design. Lugano, Switzerland. 2010. pp. 189—198.
[44]. M. Berndl, O. Lhotak, F. Qian, L. Hendren, N. Umanee. Points-to analysis using BDDs // Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation (PLDI '03). ISBN 1-58113-662-5. San Diego, California, USA. pp.103—114.
[45]. George C. Necula, Scott McPeak, S. P. Rahul, Westley Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. // in Proc. of Conference on Compiler Construction, 2002, pp. 213—228.
[46]. Aditya V. Nori and Sriram K. Rajamani. An Empirical Study of Optimizations in Yogi // ICSE '10: International Conference on Software Engineering, May 2010.
[47]. Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, Roberto Sebastiani. The MathSAT5 SMT Solver. // Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, volume 7795, 2013, pp. 93—107.
[48]. Robert Nieuwenhuis, Alberto Rubio. Paramodulation-Based Theorem Proving // Handbook of Automated Reasoning I (7), Elsevier Science and MIT Press, 2001.
[49]. D. Kroening, O. Strichman. Decision Procedures. An Algorithmic Point of View // Springer, 2008.
[50]. M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, S. Ranise, P. van Rossum, R.Sebastiani. Efficient Satisfiability Modulo Theories via Delayed Theory Combination // CAV'05.
[51]. G. Yorsh, M. Musuvathi. A Combination Method for Generating Interpolants // CADE 2005.
[52]. A.Rybalchenko, V.Sofronie-Stokkermans. Constraint Solving for Interpolation // AVACS TR No.56, 2009.
[53]. K. L. McMillan. An Interpolating Theorem Prover // TCS'05.
[54]. D.Beyer, D.Zufferey, R.Majumdar. CSIsat: Interpolation for LA+EUF. Tool paper. 2008.
[55]. A.Cimatti, A.Griggio, R.Sebastiani. Efficient Generation of Craig Interpolants in Satisfiability Modulo Theories // ACM Transactions on Computational Logic. vol. 12, issue 1, October 2010.
[56]. Thi Thieu Hoa Le. A Novel Technique for Computing Craig Interpolants in Satisfiabilility modulo the Theory of Integer Linear Arithmetic. PhD thesis.

[57]. Leonardo De Moura, Nikolaj Bjørner. Z3: an efficient SMT solver // TACAS'08/ETAPS'08. Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems. pp. 337—340.
[58]. T. Ball, E. Bounimova, R. Kumar, V. Levin. SLAM2: Static driver verification with under 4% false alarms // Formal Methods in Computer-Aided Design (FMCAD), oct. 2010. pp. 35—42.
[59]. D. Beyer. Competition on Software Verification // C. Flanagan, B. König eds. Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, vol. 7214. ISBN 978-3-642-28755-8. Springer Berlin Heidelberg, 2012. pp. 504—524.
[60]. D. Beyer. Second Competition on Software Verification // N. Piterman, S. A. Smolka eds. Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, vol. 7795. ISBN 978-3-642-36741-0. Springer Berlin Heidelberg, 2013. pp. 594-609.
[61]. A. Cimatti, A. Micheli, I. Narasamdya, M. Roveri. Verifying SystemC: A Software Model Checking Approach. // in Proc. FMCAD, FMCAD Inc. 2010. pp. 51–59.
[62]. A. Galloway, G. Lüttgen, J. T. Mühlberg, R. I. Siminiceanu. Model-Checking the Linux Virtual File System // N. D. Jones, M. Müller-Olm (eds.). VMCAI 2009. LNCS, vol. 5403, pp. 74—88. Springer, Heidelberg. 2009.
[63]. J. T. Mühlberg, G. Lüttgen. Blasting Linux Code // L. Brim, B. R. Haverkort, M. Leucker, J. van de Pol (eds.). FMICS 2006 and PDMC 2006. LNCS, vol. 4346, pp. 211—226. Springer, Heidelberg (2007)
[64]. W. Penninckx, J. T. Mühlberg, J. Smans, B. Jacobs, F. Piessens. Sound Formal Verification of Linux's USB BP Keyboard Driver // A. E. Goodloe, S. Person (eds.). NFM 2012. LNCS, vol. 7226, pp. 210—215. Springer, Heidelberg. 2012.
[65]. S. Apel, H. Speidel, P. Wendler, A. von Rhein, D. Beyer. Detection of feature interactions using feature-aware verification // 26th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2011, pp.372—375. 6-10 Nov. 2011.
[66]. A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay. Symbolic model checking of software product lines // in Proceedings of the International Conference on Software Engineering (ICSE). ACM, 2011, pp. 321—330.
[67]. D. Beyer, T. A. Henzinger, G. Theoduloz. Program Analysis with Dynamic Precision Adjustment // Proc. ASE, pp. 29—38. IEEE (2008)
[68]. P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, X. Rival. Combination of Abstractions in the ASTREE Static Analyzer // M. Okada, I. Satoh (eds.). ASIAN 2006. LNCS, vol. 4435, pp. 272—300. Springer, Heidelberg (2008)
[69]. J. Corbet, A. Rubini, G. Kroah-Hartman. Linux Device Drivers, 3rd Edition, Chapter 3, section "The open Method", pp. 58—59 // O'Reilly Media. 2005.

Introduction to CEGAR —Counter-Example Guided Abstraction Refinement

Khoroshilov A.V., Mandrykin M. U., MutilinV. S.
khoroshilov@ispras.ru, mandrykin@ispras.ru, mutilin@ispras.ru
ISP RAS, Moscow, Russia

Abstract. Precision, completeness and scalability of static verification tools have dramatically improved over the last decade. In particular, automatic checking of moderate-sized software systems has been made possible due to development of CEGAR — Counter-Example Guided Abstraction Refinement. This approach is used in such tools as SLAM, BLAST, SATABS, and CPAchecker. The paper presents an extended review of predicate abstraction-based CEGAR. It provides an introduction to the general principles of CEGAR and describes some implementation details of CEGAR in BLAST and CPAchecker. In particular, the paper concerns deciding the reachability problem for C programs by means of symbolic predicate abstraction. The set of predicates for the abstraction is obtained by Craig interpolation of the logical formulas representing the counterexample traces being discovered during the analysis. This technique is explained by two examples analyzed step-by-step both in intuitive and in formal manner. The explanation proceeds from a number of greatly simplified programs employing a very restricted subset of C language features (e.g. using only a finite set of integer variables) to more complicated programs of arbitrary size with pointers, heap allocations and bit operations. In terms of considered abstract domains the paper describes the simplest fine-grained Cartesian abstraction and a coarse-grained Boolean abstraction with adjustable block encoding. The paper also includes small discussions on common issues arising from verification of real industrial C codebase and current capabilities of existing decision procedure implementations.

Keywords: static verification, predicate abstraction, model checking, counter-example guided abstraction refinement, Craig interpolation, large-block encoding.

References

[1]. Dershowitz N. Software horror stories. URL: http://www.cs.tau.ac.il/~nachumd/horror.html
[2]. Turing A. M. On Computable numbers, with an application to the Entscheidungsproblem. In Proc. London Mathematical Society, pp. 230—265, 1936.
[3]. Floyd R. Assigning Meanings to Programs. Mathematical Aspects of Computer Science, pp. 19—32, 1967.
[4]. Hoare C. An Axiomatic Basis for Computer Programming. Communications of the ACM (CACM), vol. 12, issue 10, pp. 576-580, 1969. doi: 10.1145/363235.363259
[5]. Dijkstra E. A Discipline of Programming. Prentice-Hall, 1976.
[6]. [6]. Millo R. D., Lipton R., Perlis A. Social Processes and Proofs of Theorems and Programs. Communications of the ACM (CACM),vol. 22, issue 5, pp. 271-280, 1979. doi: 10.1145/359104.359106
[7]. Nelson G. Techniques for Program Verification. Technical Report CSL81-10: Xerox Palo Alto Research Center, 1981.

[8]. Nelson G., Oppen D. Fast Decision Procedures Based on Congruence Closure. Journal of the ACM (JACM), vol. 27. pp. 356-364, 1980.
[9]. Shostak R. Deciding Combinations of Theories. LNCS, vol. 138, pp. 209-222, 1982. doi: 10.1007/BFb0000061
[10]. Clarke E. M., Emerson E. A. Synthesis of Synchronization Skeletons for Branching Time Temporal. LNCS, vol. 131, pp. 52-71, 1982. doi: 10.1007/BFb0025774
[11]. Queille J., Sifakis J. Specification and Verification of Concurrent Systems in Cesar. In Proc. of 5th International Symposium on Programming, LNCS, vol. 137, pp. 337-351, 1981. doi: 10.1007/3-540-11494-7_22
[12]. Vardi M., Wolper P. Reasoning about Infinite Computations. Information and Computation, vol. 115, issue 1, pp. 1–37, 1994. doi: 10.1006/inco.1994.1092
[13]. Pnueli A. The Temporal Logic of Programs. In Proc. 18th Annual Symposium on Foundations of Computer Science, pp. 46-57, 1977. doi: 10.1109/SFCS.1977.32
[14]. Emerson E. Temporal and Modal Logic. Handbook of Theoretical Computer Science. Elsevier Science Publishers, vol. B, pp. 995-1072, 1990.
[15]. Khedker U. P., Sanyal A., Karkare B. Data Flow Analysis: Theory and Practice. CRC Press (Taylor and Francis Group), 2009.
[16]. D'Silva V., Kroening D., Weissenbacher G. A Survey of Automated Techniques for Formal Software Verification. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), vol. 27, issue 7, pp. 1165-1178, 2009. doi: 10.1109/TCAD.2008.923410
[17]. Jhala R., Majumdar R. Software model checking. ACM Computing Surveys (CSUR), vol. 41, issue 4, article 21, 2009. doi: 10.1145/1592434.1592438
[18]. Beyer D. Petrenko A. Linux Driver Verification. In Proc. Leveraging Applications of Formal Methods, Verification and Validation. Applications and Case Studies, LNCS, vol. 7610, pp. 1-6, 2012. doi: 10.1007/s10009-007-0044-z.
[19]. Nesov V. Automatically Finding Bugs in Open Source Programs. Third International Workshop on Foundations and Techniques for Open Source Software Certification, OpenCert 2009, vol. 20, pp.19-29, 2009.
[20]. Engler D., Chelf B., Chou A., Hallem S. Checking system rules using system-specific, programmer-written compiler extensions. In Proc. 4th conference on Symposium on Operating System Design & Implementation (OSDI), vol. 4, pp. 1-16, 2000.
[21]. Lawall J. L., Brunel J., Palix N., Rydhof H. R., Stuart H., Muller G. WYSIWIB: A declarative approach to finding API protocols and bugs in Linux code. In Proc. 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 43—52, 2009.
[22]. Avetisyan A., Borodin A. Mekhanizmy rasshireniya sistemy staticheskogo analiza Svace detektorami novykh vidov uyazvimostej i kriticheskikh oshibok [Mechanisms for extending the system of static analysis Svace by new types of detectors of vulnerabilities and critical errors]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 21, pp. 39-54, 2011 (in Russian).
[23]. Ignatyev V.N. Ispol'zovanie legkovesnogo staticheskogo analiza dlya proverki nastraivaemykh semanticheskikh ogranichenij yazyka programmirovaniya [Using static analysis for checking configurable semantic restrictions on a programming language]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 22, pp. 169-188, 2012. DOI: 10.15514/ISPRAS-2012-22-11. (in Russian).
[24]. Biere A., Cimatti A, Clarke E., Strichman O., Zhu Y. Bounded model checking. Advances in Computers, vol. 58, 2003.

[25]. Clarke E., Grumberg O., Jha S., Lu Y., Veith H. Counterexample-Guided Abstraction Refinement for Symbolic Model Checking. Journal of the ACM (JACM), vol. 50, issue 5, pp. 752-794, 2003. doi: 10.1145/876638.876643
[26]. Thomas D., Moorby P. The Verilog Hardware Description Language. Kluwer Academic Publishers, 1998.
[27]. Clarke E., Kroening D., Lerda F. A tool for checking ANSI-C programs. In Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 2988, pp. 168-176, 2004. doi: 10.1007/978-3-540-24730-2_15
[28]. Ivancic F., Yang Z., Ganai M.K., Gupta A., Shlyakhter I., Ashar P. F-soft: Software verification platform. In Proc. Computer Aided Verification (CAV), LNCS, vol. 3576, pp. 301-306, 2005. doi: 10.1007/11513988_31
[29]. Post H., Sinz C., Merz F., Gorges T., Kropf T. Linking functional requirements and software verification. In Proc. 17th IEEE International Requirements Engineering Conference, pp. 295-302, 2009. doi: 10.1109/RE.2009.43
[30]. Donaldson A. F., Kroening D., Ruemmer P. Automatic analysis of DMA races using model checking and k-induction. Formal Methods in System Design, vol. 39, pp. 83-113, 2011. doi: 10.1007/s10703-011-0124-2
[31]. Ball T., Bounimova E., Levin V., Kumar R., Lichtenberg J. The Static Driver Verifier Research Platform. Computer Aided Verification (CAV), LNCS, vol. 6174, pp. 119-122, 2010. doi: 10.1007/978-3-642-14295-6_11
[32]. Beyer D., Henzinger T., Jhala R., Majumdar R. The Software Model Checker Blast: Applications to Software Engineering. Int. Journal on Software Tools for Technology Transfer (STTT), vol. 5, pp. 505-525, 2007. doi: 10.1007/s10009-007-0044-z
[33]. Clarke E., Kroening D., Sharygina N., Yorav K. SATABS: SAT-based Predicate Abstraction for ANSI-C. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 3440, pp. 570-574, 2005. doi: 10.1007/978-3-540-31980-1_40
[34]. Beyer D., Keremoglu M.E. CPAchecker: A Tool for Configurable Software Verification. In Proc. Computer Aided Verification (CAV), LNCS, vol. 6806, pp. 184–190, 2011. 10.1007/978-3-642-22110-1_16
[35]. Mutilin V.S., Novikov E.M., Strakh A.V., Khoroshilov A.V., Shved P.E. Arkhitektura Linux Driver Verification [Linux Driver Verification Architecture]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 20, pp. 163-187, 2011 (in Russian).
[36]. Khoroshilov A., Mutilin V., Novikov E., Shved P., Strakh A. Towards an Open Framework for C Verification Tools Benchmarking. In Proc. Perspectives of Systems Informatics (PSI), LNCS, vol. 7162, pp. 82-91, 2012. doi: 10.1007/978-3-642-29709-0_17
[37]. Graf S., Saidi H. Construction of Abstract State Graphs with PVS. In Proc. Computer Aided Verification (CAV), LNCS, vol. 1254, pp. 72-83, 1997. doi: 10.1007/3-540-63166-6_10
[38]. Henzinger T.A., Jhala, R., Majumdar, R., Sutre, G. Lazy abstraction. In Proc. 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL), Pages 58-70, 2002. doi: 10.1145/503272
[39]. Cytron R., Ferrante J., Rosen B.K.,Wegman M.N., Zadek F.K. Efficiently computing static single-assignment form and the program dependence graph. ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 13, issue 4, pp. 451-490, 1991. doi: 10.1145/115372.115320
[40]. Craig W. Linear reasoning. Symbolic Logic, vol. 22, pp. 250–268, 1957.

[41]. Henzinger T.A., Jhala R., Majumdar R., McMillan K.L. Abstractions from proofs. In Proc. 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL), pp. 232-244, 2004. doi: 10.1145/964001.964021
[42]. Beyer D., Cimatti A., Griggio A., Keremoglu M.E., Sebastiani R. Software Model Checking via Large-Block Encoding. In Proc. Formal Methods in Computer-Aided Design (FMCAD), pp. 25–32, 2009. doi: 10.1109/FMCAD.2009.5351147
[43]. Beyer D., Keremoglu M. E., Wendler P. Predicate abstraction with adjustable-block encoding. In Proc. Conference on Formal Methods in Computer-Aided Design (FMCAD), pp. 189-198, 2010.
[44]. Berndl M., Lhotak O., Qian F., Hendren L., Umanee N. Points-to analysis using BDDs. Proc. ACM SIGPLAN conference on Programming language design and implementation (PLDI), pp. 103 – 114, 2003. doi: 10.1145/781131.781144
[45]. George C. Necula, Scott McPeak, S. P. Rahul, Westley Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In Proc. Conference on Compiler Construction, LNCS, vol. 2304, pp. 213-228, 2002. doi: 10.1007/3-540-45937-5_16
[46]. Nori A.V., Rajamani S.K. An Empirical Study of Optimizations in Yogi. In Proc. 32nd ACM/IEEE International Conference on Software Engineering (ICSE), vol. 1, pp. 355-364, 2010. doi: 10.1145/1806799.1806852
[47]. Cimatti A., Griggio A., Joost S. B., Sebastiani R. In Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 7795, pp. 93—107, 2013.doi: 10.1007/978-3-642-36742-7_7
[48]. Nieuwenhuis R., Rubio A. Paramodulation-Based Theorem Proving. Handbook of Automated Reasoning, Elsevier Science and MIT Press, 2001.
[49]. Kroening D., Strichman O. Decision Procedures: An Algorithmic Point of View. Journal of Automated Reasoning, vol. 51, issue 4, pp. 453-456, 2008. doi: 10.1007/s10817-013-9295-4
[50]. Bozzano M., Bruttomesso R., Cimatti A., Junttila T., Ranise S., Van Rossum P., Sebastiani R. Efficient Satisfiability Modulo Theories via Delayed Theory Combination. In Proc. Computer Aided Verification (CAV), LNCS, vol. 3576, pp. 335-349, 2005. doi: 10.1007/11513988_34
[51]. Yorsh G., Musuvathi M. A Combination Method for Generating Interpolants. In Proc. Conference on Automated Deduction (CADE), LNCS, vol. 3632, pp. 353-368, 2005. doi: 10.1007/11532231_26
[52]. Rybalchenko A., Sofronie-Stokkermans V. Constraint Solving for Interpolation. In Proc. Verification, Model Checking, and Abstract Interpretation (VMCAI), LNCS, vol. 4349, pp. 346-362, 2007. doi: 10.1007/978-3-540-69738-1_25
[53]. McMillan K. L. An Interpolating Theorem Prover. In Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), vol. 2988, pp 16-30, 2004. doi : 10.1007/978-3-540-24730-2_2
[54]. Beyer D., Zufferey D., Majumdar R. CSIsat: Interpolation for LA+EUF. In Proc. Computer Aided Verification (CAV), LNCS, vol. 5123, pp. 304-308, 2008. doi: 10.1007/978-3-540-70545-1_29
[55]. Cimatti A., Griggio A., Sebastiani R. Efficient Generation of Craig Interpolants in Satisfiability Modulo Theories. ACM Transactions on Computational Logic (TOCL), vol. 12, issue 1, 2010. doi: 10.1145/1838552.1838559
[56]. Thi Thieu Hoa Le. A Novel Technique for Computing Craig Interpolants in Satisfiabilility modulo the Theory of Integer Linear Arithmetic. PhD thesis, 2010.

[57]. De Moura L., Bjorner N. Z3: an efficient SMT solver. In Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 4963, pp. 337-340, 2008. doi: 10.1007/978-3-540-78800-3_24
[58]. Ball T., Bounimova E., Kumar R., Levin V. SLAM2: Static Driver Verification with Under 4% False Alarms. In Proc. Conference on Formal Methods in Computer-Aided Design (FMCAD), pp. 35-42, 2010.
[59]. Beyer D. Competition on Software Verification. In Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 7214, pp. 504-524, 2012. doi: 10.1007/978-3-642-28756-5_38
[60]. Beyer D. Second Competition on Software Verification. In Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 7214, pp 504-524, 2012. doi: 10.1007/978-3-642-28756-5_38
[61]. Cimatti A., Micheli A., Narasamdya I., Roveri M. Verifying SystemC: A Software Model Checking Approach. In Proc. Conference on Formal Methods in Computer-Aided Design (FMCAD), pp. 51–59, 2010.
[62]. Galloway A., Lüttgen G., Mühlberg J.T., Siminiceanu R.I. Model-Checking the Linux Virtual File System. In Proc. Verification, Model Checking, and Abstract Interpretation (VMCAI). LNCS, vol. 5403, pp. 74—88, 2009. doi: 10.1007/978-3-540-93900-9_10
[63]. Mühlberg J.T., Lüttgen G. Blasting Linux Code. In Proc. Formal Methods: Applications and Technology, LNCS, vol. 4346, pp. 211—226, 2007. doi: 10.1007/978-3-540-70952-7_14
[64]. Penninckx W., Mühlberg J. T., Smans J., Jacobs B., Piessens F. Sound Formal Verification of Linux's USB BP Keyboard Driver. NASA Formal Methods, LNCS, vol. 7226, pp. 210-215, 2012. doi: 10.1007/978-3-642-28891-3_21
[65]. Apel S., Speidel H., Wendler P., von Rhein A., Beyer D. Detection of feature interactions using feature-aware verification. In Proc. IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 372-375, 2011. doi: 10.1109/ASE.2011.6100075
[66]. Classen A., Heymans P., Schobbens P.-Y., Legay A. Symbolic model checking of software product lines. In Proc. International Conference on Software Engineering (ICSE), pp. 321-330, 2011. doi: 10.1145/1985793.1985838
[67]. Beyer D., Henzinger T. A., Theoduloz G. Program Analysis with Dynamic Precision Adjustment. In Proc. IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 29-38, 2008. doi: 10.1109/ASE.2008.13
[68]. Cousot P., Cousot R., Feret J., Mauborgne L., Miné A., Monniaux D., Rival X. Combination of Abstractions in the ASTREE Static Analyzer. In Proc. Advances in Computer Science (ASIAN), LNCS, vol. 4435, pp. 272-300, 2007. doi: 10.1007/978-3-540-77505-8_23
[69]. Corbet J., Rubini A., Kroah-Hartman G. Linux Device Drivers. O'Reilly Media, pp. 58—59, 2005.