ПРОГРАММИРОВАНИЕ, *2012*, *№ 3*, *с. 24–35*

ТЕСТИРОВАНИЕ, ВЕРИФИКАЦИЯ И ВАЛИДАЦИЯ ПРОГРАММ

УДК 681.3.06

ОПЫТ РАЗВИТИЯ ИНСТРУМЕНТА СТАТИЧЕСКОЙ ВЕРИФИКАЦИИ BLAST

© **2012 г.** П.Е. Швед, В.С. Мутилин, М.У. Мандрыкин
Институт Системного Программирования РАН
109004, Москва, ул. А. Солженицына, 25
E-mail: shved@ispras.ru, mutilin@ispras.ru, mandrykin@ispras.ru
Поступила в редакцию 18.11.2011 г.

Статический верификатор BLAST является одним лучших открытых верификаторов, работающих с программами на языке Си. В статье описываются принципы реализации BLAST, те ограничения, которые выявились при его практическом использовании для верификации драйвером ОС Linux и опыт развития BLAST, полученый в проекте LDV (Linux Driver verification) [3].

1. ВВЕДЕНИЕ

[1]

Название инструмента BLAST – это сокращение от “Berkeley Lazy Abstraction Software verification Tool”, то есть “Инструмент верификации ПО с применением ленивой абстракции, разработанный в Беркли”. Это инструмент статической верификации программ на языке C, который предназначен для решения задачи достижимости. Если дана программа на C, функция main (так называемая “точка входа”) и имя метки, инструмент проверяет, существует ли такое выполнение программы, которое начинается в точке входа и заканчивается в месте, отмеченном заданной меткой.

Инструмент анализирует программу с помощью подхода CEGAR (Counter-Example Guided Abstraction Refinement) – уточнения абстракции на основе контрпримеров. Подробнее об этом подходе можно прочитать в [1]. BLAST реализует CEGAR методом “ленивой абстракции”, который позволяет ему выполнять меньше работы, не уточняя ту часть абстракции, которая не должна измениться после анализа контрпримеров, вместо того, чтобы перестраивать её всю целиком. За использование этого подхода инструмент и получил своё имя BLAST, и был впервые представлен в секции “Экспериментальные результаты” соответствующей статьи [2]. Более того, эта статья не ссылается на какие-либо другие источники, в которых BLAST был бы также описан, или хотя бы упомянут.

Поставленная перед инструментом задача проверки достижимости является алгоритмически неразрешимой, поскольку к ней может быть сведена проблема останова. Инструмент, таким образом, не гарантирует, что его анализ когда бы то ни было завершится. BLAST также может завершиться аварийно, если обнаружит, что его средств недостаточно для корректного анализа программы. У инструмента есть и другие ограничения на проверяемые программы, подробнее о них написано в секции 2.14. В этих случаях BLAST может выдать некорректный результат: как найти “ложную ошибку” (то есть указать “ошибку” в программе, в которой её нет), так и “доказать” “ложную безошибочность” (утверждение, что в программе с достижимой меткой нет соответствующего выполнения). Однако, если инструмент считает, что в программе есть ошибка, он также печатает и “трассу ошибки”: путь от точки входа к ошибочной метке – а его уже может тщательно проверить пользователь на предмет того,

[1]Работа поддержана ФЦП “Исследования и разработки по приоритетным направлениям развития научно-технологического комплекса России на 2007–2013 годы” (контракт N 11.519.11.4006)

действительно ли такое выполнение возможно.

Чтобы построить открытую автоматизированную систему верификации драйверов Linux, описанную в статье [3], необходим инструмент статического анализа промышленного уровня. Но в этой же статье BLAST описывается, как инструмент, "intended for academic research in software verification" (то есть "предназначенный для академических исследований в области верификации ПО"). Наши эксперименты продемонстрировали, что инструмент может не только служить площадкой для экспериментов, но и на практике использоваться для интенсивной верификации драйверов. Однако много работы было проделано прежде, чем его это стало возможно.

Хорошее описание того, как работает BLAST, можно найти в [4]. Эта статья содержит пошаговое объяснение того, как BLAST верифицирует программу, на конкретном примере, но не фокусируется ни на деталях реализации, ни на объяснении корректности и концепций, на которых основан алгоритм работы этого инструмента.

В этой статье мы опишем, что сейчас собой представляет BLAST, и что мы улучшили в нём с момента релиза версии 2.5 его исходными разработчиками. Оставляя общие описания "чистых" алгоритмов другим статьям (см. [2], [4] и [5]), мы опишем те недокументированные или разбросанные по нескольким различным источникам модификации этих алгоритмов, которые были сделаны авторами инструмента. Мы будем рассматривать те алгоритмы и эвристики, которые в BLAST версии 2.5 включаются по рекомендуемым авторами опциям «-craig 2 -predH 7». Основное внимание, однако, будет уделено нашему вкладу, и, по возможности, мы будем представлять оценки эффекта от каждого отдельного улучшения в дополнение к той суммарной оценке общего эффекта от всех внесённых нами изменений, которая будет приведена в секции 3.

1.1. Как работает BLAST

Подход к статической верификации, который использует BLAST можно кратко описать как "CEGAR с ленивой декартовой предикатной абстракцией и интерполяцией Крейга для линейной арифметики и равенств с неинтерпретируемыми функциями в качестве процедуры для поиска предикатов".

Уточним, что это означает.

Подход CEGAR, или уточнение абстракции на основе контрпримеров, описывает процесс решения задачи достижимости путём построения грубой абстракции всех возможных состояний программы при выполнении её, начиная с точки входа, и последующим итеративным уточнением этой абстракции посредством анализа возможных – с точки зрения абстракции – путей, ведущих к ошибочной метке. В уточнённой абстракции таких путей становится меньше, и этот процесс продолжается до тех пор, пока их не останется вообще, или пока один из таких путей не окажется выполнимым. См. [1] для более подробной информации.

Ленивая абстракция – модификация CEGAR, характеризующаяся тем, что уточнению подлежат только те части абстракции, которые оказываются непосредственно затронуты предикатами, полученными из проанализированной трассы контрпримера. Пересмотра других частей абстракции пытаются в таком подходе избежать. Смотрите [2] для дополнительной информации.

Декартова предикатная абстракция – представление абстрактного состояния точки программы в виде конъюнкции нуля или более предикатов над пространством переменных программы. В отличие от булевой абстракции, декартова ограничивает логику первого порядка до использования лишь конъюнкции предикатов абстракции; смотрите также [6].

Интерполяция Крейга – процедура построения так называемого интерполянта Крейга [18] по упорядоченной паре логических формул, конъюнкция которых невыполнима. Интерполянтом Крейга для такой пары называется логическая формула, которая следует из первой формулы, невыполнима в конъюнкции со второй и использует только общие для этих двух формул неинтерпретируемые символы. Хотя интерполяция Крейга изначально не являлась процедурой получения предикатов, BLAST версии 2.5 использует для этих целей именно её, руководствуясь идеями, изложенными в статье [19].

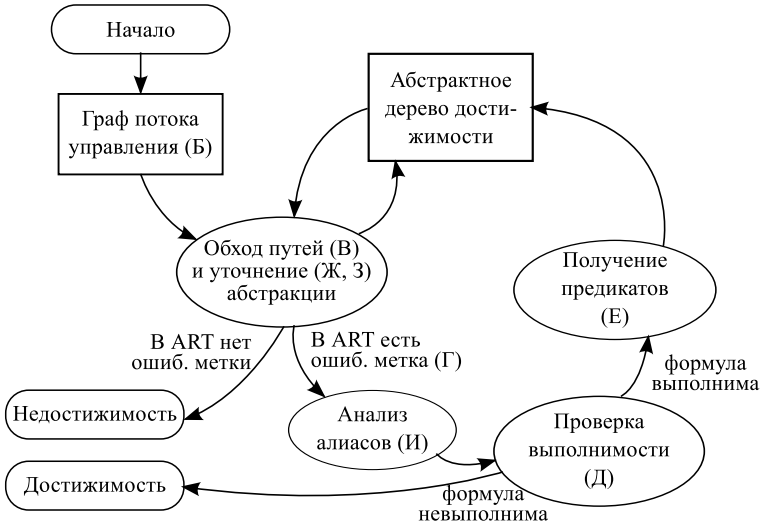

Рис. 1. Схема алгоритма верификации, используемого в BLAST. Цифрами в скобках отмечены секции, где соответствующая часть алгоритма описана более подробно.

В качестве способа представления программы используется граф потока управления (ГПУ) – конечный граф размеченной системы переходов для исходной программы. Абстракция представляется в виде абстрактного дерева достижимости (ART, Abstract Reachability Tree), префиксного дерева всех достижимых путей по вершинам ГПУ из точки входа, при этом каждая вершина дерева помечена абстрактным состоянием – регионом.

Общая схема работы инструмента изображена на рис. 1.

2. ДЕТАЛИ РЕАЛИЗАЦИИ

2.1. Общая информация

BLAST распространяется под лицензией Apache 2.0 как в виде исходного кода, так и собранным в бинарном виде под ОС Linux. Хотя BLAST включает в себя компоненты, написанные на различных языках программирования, его основная часть, реализующая алгоритмы верификации, написана на OCaml [20] и совместима с версиями OCaml 3.11–3.12. BLAST работает под Linux, но собрать его под Windows также возможно с помощью Cygwin.

OCaml – это язык с автоматическим управлением памятью, поэтому операции выделения памяти и сборки мусора при его использовании могут занимать значительное время. Эта проблема проявилась и в BLAST. Путём подстройки некоторых документированных опций виртуальной машины OCaml, мы уменьшили время, затрачиваемое на операции с памятью, и добились на некоторых программах 20% прироста в количестве посещённых точек проверяемой программы на затраченную единицу памяти.

2.2. Представление программы

Как было замечено ранее, подход, используемый в BLAST, требует, чтобы вся программа была представлена как конечный ГПУ (граф потока управления). Непосредственное использование такой формы представления означало бы, что все вызовы функций должны быть встроены (inlined), и что рекурсия запрещается. Однако BLAST подходит к этому вопросу с несколько другой стороны. Он использует CIL [7] чтобы построить ГПУ для каждой функции, а во время анализа программы, как только встречает оператор вызова функции, BLAST автоматически переходит на ГПУ, соответствующий вызываемой подпрограмме (если этот вызов был осуществлён напрямую, а не через указатель, косвенные вызовы функций BLAST не поддерживает). Этот

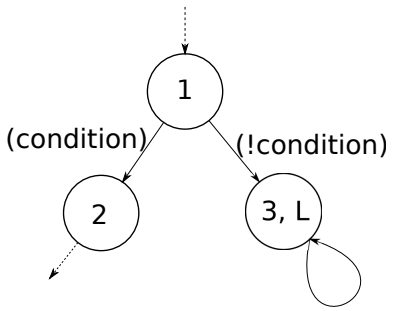

Рис. 2. Представление функции assume в виде графа потока управления.

подход предоставляет большую гибкость, позволяет сократить время на подготовку программы к анализу и даёт возможность реализовать эвристики, связанные с вызовами функций. Например, BLAST может встраивать и рекурсивные вызовы, но на какую-то заранее определённую пользователем глубину.

Мы также реализовали ограничение глубины стека вызова функций для всех вызовов, а не только рекурсивных, заметив, что в анализируемых нами программах (драйверах операционной системы) часто нет необходимости глубоко исследовать все функции, поскольку ошибки, которые находит BLAST, зачастую "лежат на поверхности". В проекте LDV (Linux Driver Verification) это позволило использовать инструменты генерации кода, которые разрешали все внешние вызовы функций в драйвере без критичной потери в скорости и качестве анализа.

При использовании представления программы в форме ГПУ, все циклы разворачиваются в условные переходы, поэтому BLAST, как инструмент, работающий с таким представлением, корректно поддерживает всевозможные операторы перехода, в том числе и оператор goto. Благодаря этому, пользователь или разработчик может подвергать программу серьёзным трансформациям, не влияя на возможность её верификации инструментом. Использование ГПУ даёт и другие преимущества. К примеру, однажды нам понадобился механизм указания условий assume, которые означают примерно следующее: "вне зависимости от предыдущего состояния в данной точке программы такое-то условие выполняется". Эти условия часто используется для указания предусловий для входных данных в случаях, когда их отсутствие может привести к ложным срабатываниям (нахождению реально невыполнимых путей). Вместо того чтобы пытаться встроить поддержку таких условий непосредственно в BLAST, мы смогли обойтись простой библиотечной функцией:

```
void assume(int condition)
{
 1: if (!condition)
 2:   L: goto L;
 3:
}
```

Эта функция не только описывает семантику assume, но и является корректным кодом на Си. Представление этой функции в виде ГПУ показано на рис. 2. Отметим, что такой способ записи assume не нов (см. например, [8]), но его поддержка обусловлена именно гибким представлением потока управления программы.

Поскольку исходный код ядра Linux использует все возможности языка C и многие его GNU-расширения, у инструментов статического анализа могут возникать проблемы с его синтаксическим разбором. Так, например, BLAST 2.5 не мог разобрать ни одного драйвера в ядре 2.6.31. Для увеличения доли драйверов, которые BLAST может корректно разобрать, мы интегрировали в BLAST последнюю версию CIL (вместо версии 1.3.1 мы встроили кандидата на версию 1.3.8). После этого изменения, BLAST смог разбирать до 98% драйверов ядра (измерено на версии 2.6.37), и лишь около 2% приводят к ошибкам разбора.

2.3. Обход абстрактного дерева достижимости

Построенная абстракция программы хранится в BLAST в виде дерева достижимости абстрактных состояний, в листьях которого и происходит вычисление абстрактных постусловий (множество всех листьев дерева также называется его "границей").

Порядок, в котором осуществляется рассмотрение в этом дереве существующих листьев и происходит добавление новых, которые

соответствуют состояниям, достижимым из первых за один переход по ребру в ГПУ, можно настраивать. Это может быть обход в ширину, в глубину, или в глубину с ограничением (т.е. обойти вершины ГПУ на определённую глубину и сформировать из вершин на границе очередь). По умолчанию используется обход в ширину, среди возможных причин – более быстрый поиск ошибочных меток и эксперименты (см. [9]), показывающие, что верификация осуществляется при обходе в ширину быстрее.

2.4. Анализ контрпримеров

Когда в абстракции находится контрпример – синтаксически достижимая точка в программе, которой, согласно текущей абстракции, соответствует непустое множество состояний, – начинается его анализ. Предусловие последовательности операций от точки входа до потенциальной ошибочной точки, называемое также "формулой пути", выписывается в форме SSA (Static Single Assignment). В этом представлении каждой переменной значение присваивается лишь единожды. Для этого переменные исходной программы разбиваются на версии, обычно с помощью добавления суффикса, и таким образом, каждое присваивание осуществляется уникальной версии переменной. Полученная формула хранится в памяти во внутреннем представлении BLAST (как структура данных OCaml), а затем конвертируется в нужный формат и передаётся внешнему инструменту проверки выполнимости логических формул – SMT-решателю (SMT-solver, см., например, [21]).

Поскольку применяемый в BLAST метод построения предусловий подразумевает их выписывание, начиная с конца (то есть с ошибочной точки), у двух формул пути не возникает общих частей, даже если им соответствует общий начальный подпуть. Возможно, здесь имеется возможность для оптимизации построения формулы путём обращения порядка выписывания предусловий и кэширования общих частей. Построение предусловий нетривиально и в случае, когда используется анализ возможных значений указателей в программе (т.е. анализ алиасов, см. секцию 2.9).

Как уже было сказано, на одном из шагов работы инструмента формулы пути конвертируются из внутреннего представления BLAST в один из форматов входных данных внешнего SMT-решателя (за конвертацию в каждый формат отвечает специальный компонент). Ввиду большого размера формулы и неэффективного, с точки зрения операции конкатенации, представления строк в языке OCaml, эта конвертация занимала много времени, особенно для SMLIB-решателей [22]. Мы улучшили конвертацию путём использования более эффективных структур данных для работы со строками, сделав накладные расходы на неё практически незаметными. Полученный результат также демонстрирует, что тесная интеграция с решателем (к примеру, составление формул сразу же в его формате или внутреннем представлении) необязательна для эффективно работающего инструмента, реализующего CEGAR.

Если решатель доказывает, что формула пути невыполнима (то есть, контрпример на самом деле нереализуем), начинается процедура поиска предикатов для построения уточнённой абстракции. Операции проверки выполнимости и поиска предикатов подробнее описаны в следующих двух секциях.

2.5. Проверка выполнимости пути

Проверкой выполнимости формулы пути занимаются внешние SMT-решатели. В качестве результата своей работы они сообщают, является ли логическая формула выполнимой или невыполнимой. Если формула пути выполнима, то этот путь является реальным, осуществимым, путём, и в программе, следовательно, есть ошибка. Иначе же формулу следует проанализировать и уточнить абстракцию по результатам анализа (см. раздел 2.6). Для формул в рамках теорий линейной арифметики и равенств с неинтерпретируемыми функциями (Linear Arithmetic and Equality with Uninterpreted Functions, LA+EUF) проверка выполнимости является алгоритмически разрешимой (алгоритмы для её решения с доказательством завершимости предлагаются, например, в [21]), но всё же вычислительно сложной задачей. Поэтому выбор качественного

и быстрого решателя критичен для построения эффективного инструмента верификации.

В BLAST версии 2.5 по умолчанию использовался решатель Simplify [23], который был "основан на стеке". В этот стек можно было добавлять (или удалять из него) конъюнкты и проверять выполнимость получившейся конъюнкции всех подформул, которые в текущий момент находятся в стеке. Такой стековый решатель являлся идеальным вариантом для инкрементального анализа формул. Однако Simplify – устаревший проприетарный решатель. После того, как мы решили проблемы со скоростью конвертации формул (см. секцию 2.4), мы смогли эффективно использовать решатели, получающие на вход формулы в формате SMTLIB – и включили в комплект поставки BLAST свободный решатель CVC3 [10], выпускаемый под лицензией LGPL. Возможность использовать другие решатели при этом была, разумеется, оставлена.

Используемый в паре с BLAST решатель помимо результатов "формула выполнима" ("sat") и "формула невыполнима" ("unsat") может также возвращать результат "неизвестно" ("unknown"), который BLAST интерпретирует как "формула выполнима" (для обеспечения корректности анализа, чтобы избежать выдачи "ложной безошибочности"). При этом доказать невыполнимость формул, которые генерирует BLAST, как правило, проще чем выполнимость. В качестве компенсации за возможную неточность (результат "unknown"), решатель может иногда работать быстрее, если у него есть такая возможность. Наши эксперименты с CVC3 показали, что в "честном" режиме доказательство выполнимости формул могло занимать у него порядка минуты и гигабайта оперативной памяти.

Оказалось, что основная причина такой низкой производительности заключалась в попытках доказательства выполнимости формул с кванторами (которые нужны для отбрасывания некоторых ложных срабатываний) с помощью встроенных эвристик CVC3, связанных с инстанцированием подкванторных выражений. Эти эвристики были активированы в CVC3 по умолчанию в режиме обработки формата SMTLIB. Одной из них была эвристика полного инстанцирования, которая заключалась в многократном инстанцировании каждой аксиомы с кванторами всеми возможными термами до достижения неподвижной точки. На это уходило много времени и памяти, поскольку в генерируемых BLAST формулах, как правило, большое количество переменных.

Мы использовали опцию для отключения эвристики полного инстанцирования подкванторных выражений, а также установили меньший лимит для повторных инстанцирований. Это сильно сократило потребляемую решателем память и время. В то же время, это существенно не повлияло на точность верификации, поскольку кванторные аксиомы достаточно редко использовались для доказательства недостижимости в проверяемых нами драйверах.

На сегодня мы удалили Simplify из BLAST, поскольку настроенный CVC3 работает намного быстрее, являясь при этом свободным (открытым) продуктом.

Мы также убрали предварительное упрощение формул, реализованное внутри BLAST, поскольку решатели, написанные на языках более низкого уровня и специально предназначенные для работы с формулами, должны справляться с этим быстрее – и это подтверждено нашими экспериментами.

2.6. Получение предикатов

Классический алгоритм поиска предикатов с помощью интерполяции Крейга, описанный в [5], выглядит следующим образом. Невыполнимая формула пути разделяется на конъюнкты, соответствующие некоторым базовым блокам исходной программы. В каждой такой точке разделения, которая также называется разрезом, выполняется поиск интерполянта Крейга между двумя частями формулы, соответствующими фрагментам трассы контрпримера до и после разреза. Для поиска интерполянтов используется специальный интерполирующий инструмент (интерполирующий решатель).

В BLAST его оригинальными авторами заложено два улучшения этого алгоритма. Во-первых, базовые блоки не разделяются на индивидуальные присваивания (см. [2]).

Во-вторых, вместо запуска процедуры интерполяции в каждой точке разреза, BLAST предварительно выделяет “полезные блоки” – подмножество операций в трассе, которые и являются своеобразной “причиной” невыполнимости формулы. То есть, вначале определяется минимальный набор базовых блоков, такой, что соответствующая ему конъюнкция подформул невыполнима, а остальная часть формулы пути – выполнима. Возможно выделение несколько таких наборов. Особенностями этого приёма (отличающие его от похожей концепции “ядра недостижимости”, “ядра невыполнимости” или “unsatisfiable core”), являются:

- гранулярность выбора подформул во множество полезных блоков. Разделение на подформулы происходит на уровне базовых блоков, которые примерно соответствуют операторам исходной программы. При этом весь базовый блок (или условие) либо целиком входит в набор полезных блоков, либо целиком в этот набор не входит. В ядро недостижимости же могут входить и подформулы с меньшей гранулярностью;
- в “полезных блоках” могут участвовать и регионы. Напомним, что регион – это абстрактное состояние. По построению это некоторая аппроксимация формулы пути до точки, которой этот регион соответствует. Поэтому вместо анализа части трассы, предшествующей региону, можно в “полезные блоки” добавлять сам этот регион.

После выделения полезных блоков, BLAST запускает интерполирующий инструмент для каждого разреза в каждом наборе “полезных блоков”, считая каждый такой набор как бы маленькой трассой ошибки. Таким образом, снижается нагрузка на интерполирующий инструмент, который по сравнению с SMT-решателем более подвержен ошибкам ввиду своей меньшей точности и большей сложности.

Отметим, что соответствующие “полезным блокам” части формулы могут и сами быть использованы как предикаты (например, как это делается на одном из шагов работы инструмента SLAM [11]).

Чтобы выделить полезные блоки, BLAST 2.5 по очереди соединяет соответствующие предикаты из формулы пути, пока их конъюнкция ни станет невыполнимой. Тогда последний добавленный блок считается полезным. Затем эта процедура повторяется сначала, только теперь выделенный полезный блок добавляется к каждой проверяемой конъюнкции, и таким образом определяется следующий полезный блок. Так происходит до тех пор, пока конъюнкция из всех найденных полезных блоков сама не станет невыполнимой.

Эта процедура естественным образом соответствует работе решателя, основанного на стеке. SMTLIB-решатели, однако, обрабатывают каждую из множества промежуточных формул отдельно. Мы заметили, что выполнимость конъюнкции всех блоков от последнего до i-го – это монотонная функция от i (чем больше блоков присоединяется, тем скорее их конъюнкция окажется невыполнимой). Поэтому для получения очередного полезного блока мы добавили процедуру двоичного поиска соответствующего ему индекса i. Это существенного уменьшило нагрузки на SMT-решатели. Также мы реализовали кэширование частей преобразованной в SMTLIB-формат формулы. Как показали эксперименты на примере драйвера cxausb, количество вызовов решателя сократилось с 32630 до 831, давая семикратный прирост производительности.

В качестве интерполирующего инструмента BLAST использует CSIsat [12], распространяемый под лицензией GPL. Он использует входной формат, совместимый с другим интерполятором – FOCI [24]. CSIsat предназначен для интерполирования формул в рамках теории линейной вещественной арифметики и неинтерпретируемых функций (LA+EUF). Это, в частности, значит, что целочисленную линейную арифметику он изначально не поддерживает, и, несмотря на то, что в нём реализованы некоторые эвристики для получения интерполянтов с целыми коэффициентами, всё же иногда он может выдавать неравенства вроде «$x < 0.1y$» (вместо эквивалентного «$(10x < y)$». В этом случае BLAST игнорирует такие части полученного интерполянта и пытается

доказать недостижимость ошибочной метки без соответствующих предикатов.

В качестве способа представления регионов в BLAST используется BDD над атомарными предикатами – частями интерполянта, содержащими только линейную арифметику и дизъюнкции.

2.7. Уточнение абстракции

Уточнение абстракции происходит, начиная с тех вершин в абстрактном дереве достижимости, которые были отмечены как "полезные блоки" в предшествующем анализе контрпримеров. Для вычисления того, какие предикаты описывают регион в вершине, по региону в предшествующей вершине и операции на соответствующем ребре ГПУ используется процедура вычисления "абстрактного постусловия", описанная в [2]. Для проверки выполнимости соответствующих формул используется тот же решатель, что и для обработки трассы контрпримера.

Поскольку вычисление производится на основании локальных данных, процесс пересмотра абстракции может быть распараллелен; в BLAST отсутствует такая возможность, но исследования показывают перспективность такого улучшения [13].

2.8. Конфигурируемая верификация

В BLAST встроена возможность комбинировать анализ потока данных, основанный на решётках, с CEGAR. Эта возможность реализована на основе подхода, описанного в статье [25], где вводится формализм адаптивного программного анализа (CPA, Configurable Program Analysis) и даются соответствующие определения абстрактного домена (abstract domain), решётки (lattice), отношения перехода (transfer relation), оператора слияния (merge operator) и проверки покрытия (termination check, также именуется stop operator). Здесь мы воспользуемся этими определениями при описании некоторых деталей реализации возможностей CPA в BLAST.

Зачастую анализ с помощью решёток не обладает достаточной точностью для описания состояний программы, но может помощь в отсечении заведомо недостижимых путей. Оставшиеся пути будет проанализированы с помощью описанных выше алгоритмов. В BLAST включены несколько соответствующих решёток; одна из них, symbolic store, представляет собой анализ общего назначения, подходящий для специально не размеченных программ на C. Эта решётка хранит информацию о конкретных значениях целочисленных переменных, а также некоторую информацию о состоянии кучи, см. [14].

Чтобы использовать информацию, поступающую от основанного на решётках анализа потока данных, BLAST расширяет понятие о регионе вершины абстрактного дерева достижимости, добавляя к BDD над атомарными предикатами ещё и элемент соответствующей решётки. Если этот элемент равен $\bot$ (так обозначается элемент решётки, соответствующий пустому или недостижимому абстрактному состоянию), то дальнейшее рассмотрение путей из этой вершины бессмысленно, так как эта вершина недостижима.

Что касается анализа покрытия, то есть остановки процесса рассмотрения путей из некоторой вершины в случае, если в другой части дерева абстракции имеется набор вершин, покрывающих данную в смысле представляемых множеств состояний программы, BLAST содержал здесь серьёзную ошибку. Было жёстко запрограммировано использование оператора покрытия stop-join, то есть регион, соответствующий вершине, мог быть покрыт объединением других регионов, а не каждым из них в отдельности. Поскольку элементы решётки symbolic store, в отличие от предикатов, не образуют степенной домен (powerset domain), что, проще говоря, означает, что в таком домене не всегда находится элемент для представления объединения нескольких других его элементов, этот оператор покрытия отсекал некоторые осуществимые пути выполнения, на которых располагались, в том числе, и реальные ошибки. Число таких случаев возрастало при использовании определённых моделей окружения – автоматически сгенерированных функций main для анализа драйверов ядра (см. [3]). Поэтому мы реализовали оператор покрытия stop-sep, проверяющий на предмет покрытия элементы решётки регионов в каждой

вершине отдельно. Мы также реализовали несколько операторов слияния элементов решётки: слияние всего всегда (оператор merge-join), слияние элементов решётки в вершинах с одинаковыми предикатами (merge-pred-join) и отсутствие слияния (merge-sep).

После проведения экспериментов в качестве конфигурации по умолчанию были выбраны операторы stop-sep и merge-pred-join. Хотя в результате этого время работы BLAST увеличилось на 50%, точность анализа тоже была значительно увеличена: количество найденных реальных ошибок возросло на 50%.

Концепция простого совмещения различных операторов при обходе пространства состояний одной программы разработана в другом инструменте, реализующем схожий с BLAST подход, CPAchecker [9]. Наш опыт показывает, что BLAST достаточно хорошо структурирован, чтобы добавление таких операторов было возможным, хотя для этого в коде не предусмотрено "синтаксического сахара" и специально выделенных интерфейсов, таких как в CPAChecker.

2.9. Анализ алиасов

Алиасом переменной обычно называется одно из возможных обозначений этой переменной в данной точке программы. Например, пусть в программе имеются объявления int a; int $*p$;. Тогда в точке программы, где выполнено условие $(p == \&a)$, разыменование $*p$ будет являться алиасом (иначе говоря, синонимом) переменной a. Аналогично можно говорить об алиасах некоторого выражения, например, $*(q + 1)$ для $b[1]$, если $(q == b)$. В BLAST используется не зависящий от потока данных анализ возможных алиасов переменных для уточнения верификации в присутствии присваиваний с участием указателей. В [2] описано, как информация об алиасах используется для построения предусловий, использующихся в формуле пути и при пересмотре абстракции, а в [5], в секции о "плоских" программах, этот алгоритм расширен для поддержки структурных типов. Вкратце, анализ алиасов позволяет для данного присваивания определить, какие ещё выражения (в каждый момент это подмножество всех возможных алиасов данного выражения) меняют своё значение в результате выполнения данного оператора.

Анализ алиасов осуществляется в терминах анализа Андерсена [15]. Для хранения и запросов к информации об алиасах используется анализ BDD и алгоритм, похожий (но не совпадающий) с алгоритмом в [16]. Два символьных выражения считаются возможными алиасами друг друга, если существует значение на стеке (или точка вызова функции выделения памяти), которое является возможным алиасом для каждого из них.

Как указано в [2], программу для корректного анализа алиасов необходимо преобразовать в форму, где в левой части каждого присваивания используется не более одного разыменования. Это преобразование, называемое "упрощением памяти" поддерживается в CIL и включается в BLAST по умолчанию.

Мы реализовали несколько эвристик, позволяющих сократить время, требующееся для анализа алиасов. Во-первых, некоторые возможные алиасы (may-alias) также являются и обязательными (must-alias). Это, например, дополнительные переменные, добавляемые при упрощении памяти, которые, по построению, не используются в других частях программы. Для обязательных алиасов (must-aliases) какого-то выражения не обязательно проверять условия равенства адресов алиаса и выражения, поскольку они всегда истинны. Отдельная обработка двух множеств возможных и обязательных алиасов принесла заметное ускорение работы алгоритма их анализа. Мы также реализовали возможность включения предположения о том, что все константные указатели являются обязательными алиасами.

Однако эти эвристики не принесли ожидаемого увеличения размера программ, для которых возможен анализ указателей (несмотря на ускорение работы подсистем, анализирующих алиасы, более чем в сотню раз). Мы считаем, что для того, чтобы сделать анализ алиасов масштабируемым, необходим качественный скачок в исследованиях на эту тему.

По этой причине использование информации об алиасах в BLAST является опциональным и отключено по умолчанию.

2.10. Взаимодействие с пользователем

Пользовательский интерфейс инструмента консольный. Опции предаются ему через аргументы командной строки, а вывод происходит непосредственно в стандартный поток. Мы практически не изменили этот интерфейс, лишь добавили в выводимый BLAST результат специальные маркеры, позволяющие автоматически выделять в нём вердикт (результат анализа) и контрпример. Мы также добавили в получаемые трассы контрпримеров специальные маркеры, сигнализирующие о причинах, по которым некоторые функции в них были пропущены (ограничение ли это глубины стека или отсутствие тела функции во входных файлах).

2.11. Инфраструктура

Мы добавили автоматизированные регрессионные тесты, которые основаны на ситуациях, встреченных нами в реальном коде драйверов. Эти тесты содержат как истинный, так и ожидаемый результат, чтобы можно было отслеживать улучшения и деградации.

Внешние решатели и интерполяторы подсоединяются к BLAST с помощью специального медиатора, который занимается уменьшением задержек в ситуациях, когда инструмент уже выдал вердикт, но ещё не завершил работу (возможно, вследствие нетривиальной процедуры освобождения ресурсов).

BLAST содержит большое количество неиспользуемого кода; в основном, это экспериментальные возможности, которые не оправдали себя и перестали поддерживаться. Мы не пытались от него избавиться, так как он часто несёт в себе полезную информацию о вариантах развития инструмента. Однако в документации к инструменту мы отделили поддерживаемые опции от тех, которые являются экспериментальными или не поддерживаются вовсе.

2.12. Описание внешних компонентов

После наших изменений BLAST, сконфигурированный “по умолчанию”, включает в себя следующие компоненты:

- Пакет CUDD для работы с BDD. Реализован на языке C и распространяется под лицензией, похожей на MIT. Встроен с помощью возможностей языка OCaml по взаимодействию с программами на C.
- Решатель CVC3 [10] – используется для проверки выполнимости формул. Реализован на C++, лицензирован под GPL. Встроен как внешнее приложение, и может быть заменён на другой решатель, поддерживающий формат SMTLIB.
- Инструмент для интерполяции CSIsat [12] – вычисляет интерполянты Крэйга для формул в рамках LA+EUF. Распространяется под лицензией GPL. Взаимодействует с BLAST как внешнее приложение, и может быть заменён на другой инструмент, поддерживающий формат, совместимый с FOCI.
- CIL – конвертирует программу на языке C в синтаксическое дерево, которое хранится как структура данных OCaml. Распространяется под лицензией Apache. Статически скомпонован с BLAST.

2.13. Известные ограничения

BLAST не поддерживает присваивание структур, вызовы функций по указателю, игнорирует ассемблерные вставки, не осуществляет автоматического определения свойств сложных структур данных (таких как хэши и связные списки), не поддерживает массивы, считая каждый доступ к массиву отдельной переменной (например, a[i] и a[j] считаются всегда разными переменными), не может использовать неравенство адресов структур на стеке, не имеет быстрого алгоритма анализа алиасов, не всегда правильно обрабатывает короткую логику в выражениях, не разбит на легко заменяемые модули и всегда анализирует исходную программу целиком.

Эти ограничения часто являются причиной ложных срабатываний инструмента. В таких случаях выданный им контрпример приходится анализировать вручную. Однако наш опыт применения BLAST для верификации кода драйверов Linux показал, что несмотря на

Таблица 1. Сравнение версий BLAST 2.5 и 2.6

Версия	Запусков	Метка недостижима	Найдено ошибок	Неизв.	Время	Лимит времени	Лимит памяти	Прочие ошибки
2.5	389	274	5	110	11,5 часов	36	16	58
2.6	389	317	15	57	2,1 часа	2	28	27

перечисленные ограничения, этот инструмент вполне способен находить в том числе и реальные ошибки (см. [26]).

3. ОЦЕНКА РЕЗУЛЬТАТОВ

Чтобы оценить эффект от наших улучшений, мы сравнили, как последняя версия BLAST, названная нами 2.6 [17], и оригинальная версия BLAST 2.5 справляются с верификацией драйверов ядра Linux версии 2.6.37 из папки drivers/media/. Мы проверили выполнение спецификации, заданной правилом под названием “повторная блокировка мьютекса (одноместного семафора) или его разблокировка без предшествующей блокировки”. Каждый запуск BLAST был ограничен 15 минутами и 1 Гб памяти. Чтобы оригинальный BLAST мог справляться с синтаксическим разбором кода ядра, мы встроили в него последнюю версию CIL (без этого оригинальная версия BLAST не была способна разобрать ни один драйвер). Результаты сравнения приведены в табл. 1.

Результаты показывают, что разработанная нами новая версия BLAST способна найти втрое больше ошибок, при этом не пропуская ошибок, найденных предыдущей версией. Число драйверов, во время проверки которых инструмент вышел за пределы выделенных ему ресурсов, уменьшилось с 52 до 30. Это что означает что новая версия BLAST смогла верифицировать 42% тех самых сложных драйверов, с которыми не справилась его оригинальная версия, за те же ограничения по ресурсам.

Мы также провели дополнительные эксперименты для всех драйверов ядра 2.6.37 (не отражены в таблице). По сравнению с оригинальной версией BLAST, количество найденных ошибок увеличилось в 5 раз, среднее время работы уменьшилось в 8 раз, а ускорение для 15 самых сложных из проверенных оригинальным BLAST драйверов достигло 27 раз.

4. ВЫВОДЫ

Начав с версии BLAST 2.5, выпущенной в середине 2008 года, мы внесли множество исправлений в инструмент, что улучшило его производительность при верификации промышленного кода (драйверов ядра Linux) в среднем в 8 раз, увеличивая с ростом сложности верифицируемого драйвера этот показатель до 27. При этом анализ стал более точным и способным найти большее количество ошибок (до 5 раз больше). Улучшение точности было не просто обнаружено постфактум, после оптимизации использования ресурсов, а вызвано в том числе и устранением ошибок, негативно влиявших на точность.

Мы также успешно внедрили SMT-решатель общего назначения и продемонстрировали, что конвертация формул из внутреннего представления в формат SMTLIB при проверке контрпримеров на промышленных программах, соразмерных с драйверами ядра Linux, не вносит никакого заметного замедления.

Мы также обнаружили, что BLAST является расширяемым, и на его основе можно реализовывать и оценивать новые алгоритмы верификации, хотя это и не всегда простая задача для разработчика. Таким образом, несмотря на то, что развитие этого инструмента оригинальными разработчиками прекращено, его дальнейшая разработка всё ещё может приносить новые полезные результаты как в исследовательском, так и в практическом плане.

СПИСОК ЛИТЕРАТУРЫ

1. *Clarke E.M., Grumberg O., Jha S., Lu Y., Veith H.* Counterexample-guided abstraction refinement, CAV, 1855:154–169, 2000.
2. *Henzinger T.A., Jhala R., Majmudar R., Abstraction L.* Symposiym on Principles of Programming Languages. P. 58–70, 2002.
3. *Khoroshilov A., Mutilin V., Shcherbina V., Strikov O., Vinogradov S., Zakharov V.* How to Cook an Automated System for Linux Driver Verification, 2nd Spring Young Researchers' Colloquium on Software Engineering. P. 11–14, 2008.
4. *Beyer D., Henzinger T., Jhala R., Majumdar R.* The software model checker BLAST: Applications to software engineering, Int. J. Softw. Tools Technol. Transf., 2007.
5. *Henzinger T., Jhala R., Majumdar R., McMillan K.* Abstractions from proofs, Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages. P. 232–244, 2004.
6. *Ball T., Podelski A., Rajamani S.* Boolean and Cartesian abstractions for model checking of C programs, Proceedings of TACAS, 2001.
7. *Necula G., Mcpeak S., Rahul S., Weimer W.* CIL: Intermediate language and tools for analysis and transformation of C programs, International Conference on Compiler Construction. P. 213–228, 2002.
8. *Yunho M., Hotae K.* A Comparative Study of Software Model Checkers as Unit Testing Tools: An Industrial Case Study, 2011.
9. *Beyer D., Cimatti A., Griggio A., Keremoglu M., Sebastiani R.* Software model checking via large-block encoding., FMCAD. P. 25–32, 2009.
10. *Barrett C., Tinelli C.* CVC3, CAV. P. 298–302, 2007.
11. *Ball T., Bounimova E., Kumar R., Levin V.* Static driver verification with under 4 false alarms, FMCAD, 2010.
12. *Beyer D., Zufferey D., Majumdar R.* CSIsat: Interpolation for LA+EUF, CAV. P. 304–308, 2008.
13. *Lopes N., Rybalchenko A.* Distributed and predictable software model checking, VMCAI. P. 340–355, 2011.
14. *Beyer D., Henzinger T.A., Theoduloz G.* Lazy shape analysis, CAV. P. 532–546, 2006.
15. *Andersen L.O.* Program Analysis and Specialization for the C Programming Language, 1994.
16. *Berndl M., Lhotak O., Qian F., Hendren L., Umanee N.* Points-to analysis using BDDs, PLDI, 2003.
17. http://forge.ispras.ru/projects/blast
18. *Craig W., Reasoning L.* A New Form of the Herbrand-Gentzen Theorem // J. Symbolic Logic. V. 22. I. 3 (1957). P. 250–268.
19. *McMillan K.L.* An Interpolating Theorem Prover, TCS 2005.
20. http://caml.inria.fr/ocaml/
21. *Kroening D., Strichman O.* Decision Procedures: An Algorithmic Point of View. Springer, July 2008.
22. http://www.smtlib.org/
23. *Detlefs D., Nelson G., Saxe J.B.* Simplify: A Theorem Prover for Program Checking // J. of the ACM. V. 52. № 3. May 2005. P. 365–473.
24. http://www.kenmcmil.com/foci.html
25. *Beyer D., Henzinger T.A., Theoduloz G.* Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis, CAV 2007, LNCS 4590. P. 504-518, 2007.
26. http://linuxtesting.org/results/ldv