Санкт-Петербургский государственный университет

БАТОЕВ Константин Аланович

Выпускная квалификационная работа

Направляемое свойством символьное исполнение программ на платформе .NET

Уровень образования: магистратура

Направление *09.04.04 «Программная инженерия»*

Основная образовательная программа *ВМ.5666.2019 «Программная инженерия»*

Научный руководитель:
д.т.н., профессор кафедры системного программирования Д.В. Кознов

Рецензент:
руководитель департамента анализа программ ООО «Техкомпания Хуавей» Д.А. Иванов

Санкт-Петербург
2021

Saint Petersburg State University

Konstantin Batoev

Master's Thesis

Property-directed symbolic execution of programs on .NET platform

Education level: master

Speciality *09.04.04 «Software Engineering»*

Programme *BM.5666.2019 «Software Engineering»*

Scientific supervisor:
Sc.D, prof. D.V. Koznov

Reviewer:
software analysis department team leader at "Huawei" D. Ivanov

Saint Petersburg
2021

Оглавление

Введение

Задача проверки качества и надежности кода является очень актуальной. При этом программное обеспечение может быть существенно разным с точки зрения требований качества: во-первых, критически важные медицинское и военное ПО, ошибки в которых могут влиять на жизни людей; во-вторых, прикладное и промышленное ПО, и цена ошибок в этих приложениях может стоить существенных денежных потерь, и, наконец, любительские приложения, ошибки которых влияют на настроение пользователей. Отметим, что количество критически важного ПО нарастает и, по факту, требования к нему увеличиваются. Одной из методик проверки качества и надежности кода является символьное исполнение [3].

К сожалению, символьное исполнение имеет свои ограничения. Для произвольной программы оно может свидетельствовать о наличии ошибок, но не способно доказать их отсутствие. Причиной этому является очень большое число путей исполнения программы и, следовательно, огромное число символьных состояний. Примерами не очень больших программ, порождающих, тем не менее, очень большое количество символьных состояний, является программа с циклом, количество итераций которого заранее неизвестно [6]. Однако даже если искусственно ограничивать количество посещений цикла, символьное исполнение будет столкнётся с комбинаторным взрывом множества состояний, что ограничивает эффективность этой технологии на практике.

Комбинаторный взрыв множества путей исполнения является также помехой для обратного символьного исполнения [7], которое начинается из заданной точки программы и двигается в сторону начальной точки входа в программу. Обратное символьное исполнение необходимо, например, для генерации полного тестового покрытия веток программы. Отчасти справится с проблемой роста множества состояний помогает использование различных стратегий исследования кода. Стратегия исследования — это функция, которая выбирает очередное состояние из очереди. А именно, стратегия позволяет процедуре символьного ис-

полнения быстрее найти ошибку или трассу исполнения, ведущую к заданной инструкции программы, но не может полностью преодолеть упомянутые выше ограничения.

В области формальной верификации аппаратного и программного обеспечения на текущий момент активно развивается подход к исполнению программы, направляемый свойством достижимость (Property Directed Reachability) [12]. В отличие от классического символьного исполнения, этот подход позволяет как доказывать корректность программы, так и находить в ней ошибки. Если адаптировать идеи этого подхода для символьного исполнения, то последнее будет способно доказывать недостижимость бесконечного числа путей исполнения, а также направлять символьное исполнение к заданным инструкциям кода, что повышает эффективность анализа программы.

Фактически, данный подход пытается инкрементально строить бесконечную цепочку лемм, приближающих поведение программы сверху относительно заданного свойства до тех пор, пока не будет построен индуктивный инвариант, доказывающий корректность программы относительно свойства или не будет предъявлен достижимый контрпример. Для применения этого подхода к символьному исполнению на практике было принято решение проводить разработку нового алгоритма символьного исполнения в рамках проекта V# — символьной виртуальной машины для анализа программ платформы .NET.

Постановка задачи

Целью данной выпускной квалификационной работы является создание алгоритма символьного исполнения, способного доказывать недостижимость путей исполнения программы и направлять символьное исполнение кода к заданным локациям кода. Для достижения цели были поставлены следующие задачи.

- Провести обзор символьного исполнения и подхода PDR (IC3).
- Разработать алгоритм направляемого свойством символьного исполнения.

- Реализовать алгоритм внутри символьной виртуальной машины V#.
- Провести эксперименты, доказывающие эффективность предложенного алгоритма.

1. Обзор

1.1. Символьное исполнение

Символьное исполнение — это техника анализа программ, заключается в исполнении программы в условиях символьных (то есть, максимально произвольных) значений входных аргументов программы, в отличие от обычного исполнения программы, когда входные аргументы программы заданы во время запуска. Это позволяет исполнять программу не по одному пути исполнения, а по нескольким путям сразу, причем для каждого пути вычисляется соответствующее ограничение на условие пути.

Символьное исполнение было предложено в 70-х годах прошлого века [11]. Чаще всего символьное исполнение применяется для порождения тестов, покрывающих все пути исполнения программы. Также оно используется для проверки достижимости ошибок, то есть для нахождения таких трасс исполнения, которые бы свидетельствовали ошибочное или непредвиденное разработчиком поведение. Например, ошибками могут быть разыменование символьного указателя, которому ничто не мешает оказаться нулевым во время обычного исполнения; достижение инструкции, которая вызывает исключение, никак не обрабатываемое в коде; нарушение условия у оператора «assert» и т.д. Часто эти два применения используются совместно.

Ключевым понятием символьного исполнения является *символьное состояние*, которое учитывает эффекты исполненных инструкций кода. Для конкретных промышленных языков программирования «начинка» символьного состояния может быть разной, основными полями символьного состояния являются:

- локация loc — информация для определения следующей неисполненной инструкции программы;
- ограничение условия пути PC — символьная формула, содержащая все условия перехода за время символьного исполнения;

- состояние памяти $store$ — отображение из множества локаций (локальных переменных, аргументов функции, глобальных переменных) в множество их символьных значений.

1.1.1. Ограничения символьного исполнения

Возможность исполнять программу по нескольким путям одновременно имеет свои минусы, ограничивающие применимость символьного исполнения на практике.

Во-первых, символьное исполнение нельзя в общем случае применять для доказательства корректности программ, поскольку программа может обладать бесконечным числом путей исполнения. Это ограничение сильно усложняет задачу генерации тестового покрытия: например, исполнение будет продолжать пытаться достичь недостижимой локации кода, раскручивая цикл в программе.

Во-вторых, даже если в программе нет бесконечного числа путей исполнения, возможна ситуация, когда их больше, чем можно покрыть на практике (комбинаторный взрыв). Например, на листинге 1 представлена функция $TestArray$, которая принимает символьный массив целых чисел. Каждая итерация цикла будет порождать два новых символьных состояния, соответствующих каждой ветке оператора ветвления **if**. Поскольку таких итераций 100, то всего может быть 2^{100} путей исполнения, каждому из которых соответствует своё символьное состояние.

В-третьих, классическое символьное исполнение (т.н. прямое символьное исполнение) не направлено на решение задачи покрытия каждой инструкции в коде.

1.1.2. Стратегия исследования

Существует подход, который позволяет обойти упомянутые выше ограничения. Поскольку алгоритм символьного исполнения исследует программу сразу по нескольким путям исполнения, то на каждой итерации она должна выбирать очередное состояние для исполнения. Будем называть функцию выбора очередного состояния *стратегией исследо-*

```
void TestArray(int[] array) {
    int errors = 0;
    for (int i = 0; i < 100; i++) {
        if (array[i] == 0) {
            errors++;
        }
    }
    if (errors >= 50) throw new Exception();
}
```

Листинг 1: Пример программы, для который будет комбинаторный взрыв состояний для символьного исполнения

вания.

Эффективность на практике зависит от стратегии исследования. Например, для программы 1 удачная стратегия исполнения позволит проанализировать граф потока управления, распознать цикл и использовать графовый алгоритм поиска в глубину. Тогда значение переменной $errors$ было бы максимально большим с вероятностью 50% (в зависимости от того, какую ветку оператора ветвления выберет обход dfs), что бы могло засвидетельствовать оператор генерации исключения.

В общем случае стратегия исследования не может преодолеть ограничения символьного исполнения. Она способна лишь дать хорошую эвристику, которая улучшит эффективность алгоритма на программах конкретного вида. Например, стратегия, являющаяся комбинацией нескольких других стратегий, могла бы увеличить множество видов успешно анализированных программ программ посредством чередования последних.

1.2. Композициональное символьное исполнение

Есть ещё один способ сделать символьное исполнение более эффективным. Сделать так, чтобы исполнение по возможности не исполняло заново инструкции одних и тех же путей программы. Например, если в коде происходит последовательный вызов метода M три раза, то можно исполнить тело метода M один раз от входной точки до конца и полу-

чить символьное состояние s_M. А затем переиспользовать состояние s_M, чтобы получить символьное состояние, соответствующее вызову метода M три раза.

Композициональное символьное исполнение — это расширение символьного исполнения, которое позволяет порождать новые символьные состояния не только с помощью исполнения инструкций программы, но и с помощью операции композиции состояний [1]. Композиция состояний s_1 и s_2 — это состояние $Compose(s_1, s_2)$, которое соответствует пути исполнения, который сначала прошёл путь состояния s_1, а затем прошёл путь состояния s_2.

Благодаря операции композиции можно не исполнять заново одни и те же блоки инструкций, что позволяет существенно повысить эффективность символьного исполнения. Например, можно один раз символьно исполнить метод M от входной точки до выхода, тем самым получить состояние s_M, отражающее символьный эффект всех инструкций этого метода. И для всех символьных состояний, путь которых достигает вызова метода M, получить состояние, соответствующее результату исполнения метода M на состоянии s с помощью композиции $Compose(s, s_M)$. Если таких вызовов много, это может существенно увеличить эффективность символьного исполнения.

1.3. Обратное символьное исполнение

Существует еще одна разновидность символьного исполнения — *обратное символьное исполнение*, которое пытается предъявить трассу программы, ведущую в заданную локацию кода [7, 10]. Обратное символьное исполнение может быть полезно для генерации полного тестового покрытия программы, например, когда какая-то локация не была еще посещена.

У обратного исполнения есть свои ограничения. Во-первых, комбинаторный взрыв множества состояний программы. Он случается, когда в текущую локацию кода можно попасть из нескольких других. Во-вторых, пока *локация loc* символьного состояния не равна входной

точки EP программы, состояние может соответствовать недостижимому пути исполнения программы, в то время как прямое символьное исполнение взаимодействует с достижимыми путями исполнения.

1.4. Формальная верификация программ

Одним из ограничений символьного исполнения является неспособность *доказывать* недостижимость путей исполнения программы. Чтобы попытаться преодолеть это ограничение, можно обратиться за идеями в область формальной верификации.

На сегодняшний день в области верификации аппаратного и программного обеспечения доминирующим подходом является исполнение программы, направляемое свойством достижимость (Property Directed Reachability, PDR) [4]. Целью PDR заключается в доказательстве того, что свойство P выполняется для всех состояний программы, когда отношение отношение перехода программы T может быть исполнено произвольное число раз, либо найти состояние, либо в предъявлении состояния, нарушающего свойство P.

В контексте верификации аппаратного обеспечения подход пытается инкрементально конструировать цепочку *лемм* $F_0, F_1, \ldots, F_k$, отталкиваясь от свойства P. Леммы F_i — это множество формул, приближающих сверху состояния программы, которые исполнили отношение перехода программы не более, чем i раз (i называют уровнем лемм). Подход пытается *стабилизировать* цепочку, то есть, найти такой k, что леммы на уровне k в точности совпадают с леммами на уровне $k{+}1$ ($F_k = F_{k+1}$). В таком случае, будет показана *взаимная индуктивность* лемм F_k, то есть, леммы F_k приближают сверху состояния программы, исполнившие отношение перехода *любое* конечное число раз. С помощью таких взаимно-индуктивных лемм подход PDR *доказывает*, что свойство P выполняется *всегда* для всех состояний программы.

В работе [12] было предложено обобщение подхода PDR для области формальной верификации программного обеспечения. Обобщенный подход нацелен на доказательство безопасности рекурсивных про-

грамм. В отличие от оригинального алгоритма, который пользуется монолитным отношением перехода программы, новый подход взаимодействует с отношениями переходов отдельных функций, что придает ему модульность.

Если говорить более конкретно, обобщенный подход строит цепочки лемм отдельно для каждой функции программы. Кроме того благодаря модульности подхода нет необходимости встраивать тело вызываемой функции, поэтому для каждой функции подход строит также цепочки формул. Эти цепочки приближают снизу состояния программы, которые исполнили данную функцию. Цепочки лемм, приближающих снизу состояния программы, нужно для того, чтобы предъявлять состояние программы, нарушившее свойство P.

Можно было бы применить данные идеи к символьному исполнению: строить цепочки лемм для вершин графа потока управления, без необходимости строить отношение перехода каждой функции. Это помогло бы доказывать недостижимость как межпроцедурных путей исполнения программы, так и путей внутри тела самой функции.

2. Алгоритм символьного исполнения, направляемого свойством

Цель классического алгоритма символьного исполнения — исполнить как можно больше путей в программе, начинающихся из входной точки, и приводящих программу в ошибочное состояние (например, необработанное исключение останавливает исполнение программы). В отличие от этого, целью предлагаемого в данной работе алгоритма, является генерация *тестового покрытия* для заданных локаций кода. Таким образом, для каждой локации алгоритм должен либо найти входные аргументы для программы, приводящие символьное исполнение в данную локацию (ответить положительно), либо доказать, что данная локация недостижима (ответить отрицательно).

Перед тем, как детально описать предложенный алгоритм, введем ряд важных понятий.

2.1. Основные понятия

Ключевым понятием алгоритма является *запрос*, представимый в виде тройки (loc, φ, lvl). Рассмотрим подробно элементы этой тройки:

- loc — это информация, уникально определяющая *локацию кода*; в качестве такой информации в алгоритме использовано имя метода и номер инструкции внутри этого метода;
- φ — это *свойство*, требующее проверки и являющееся формулой первого порядка;
- lvl — *уровень*, т.е. максимальное число шагов алгоритма, которое может содержаться в пути исполнения для ответа на данный запрос (т.е. выполняется ли данное свойство для выбранной локации в программе).

Запросы подаются на вход алгоритму для получения *ответов*. Положительный ответ на запрос означает, что нашлось некоторое символь-

ное состояние s, выполняющее свойство φ и соответствующий путь исполнения алгоритма, стартовавший из входной точки EP программы (Entry Point) и дошедший до локации loc, причем этот путь не длинее, чем lvl шагов. Будем называть такое состояние *свидетелем* запроса.

Следует отметить, что задачу проверки достижимости заданных локаций кода ($initialLocs$) можно свести к задаче ответа на запросы следующего вида: $(loc, \top, \infty)$, где $loc \in initialLocs$, $\top$ — всегда выполнимая формула (тавтология), а ∞ означает бесконечный уровень. Запросы такого вида являются *главными*. Ответ на главный запрос (loc, $\top$, ∞) эквивалентен проверке достижимости локации loc и отвечает на вопрос о том, может ли какое-то символьное состояние s, соответствующее пути исполнения, начинающегося из входной точки программы EP, достичь локации loc, пройдя *любое* конечное число шагов, так что оно выполняет всегда выполнимую формулу $\top$.

Итак, если ответ алгоритма на запрос является положительным, то это означает, что он нашел свидетеля запроса. Если же алгоритм ответил на запрос отрицательно, то это означает, что он доказал, что не существует свидетеля для данного запроса.

2.2. Идея алгоритма

Ответить на нетривиальные главные запросы о достаточно большой программе с помощью символьного исполнения, так сказать, «в лоб», крайне затруднительно. Ведь алгоритму нужно рассмотреть различные пути исполнения, которые могут состоять из произвольного (ничем не ограниченного) числа шагов, и таких путей может быть бесконечно много. Вместо этого алгоритм может отвечать запросы инкрементально: он может создать аналоги главных запросов (уровень 0) и пытаться ответить на них. Если не на все запросы будут найдены ответы, то алгоритм заводит вспомогательные запросы (уровень 1), и попытается ответить на них. Этот процесс будет продолжаться до тех пор, пока главный запрос остаётся без ответа.

Очевидно, что в конечном счете алгоритм ответит на все главные

запросы, на которые можно ответить положительно. Но что делать с запросами, на которые в принципе нельзя ответить положительно? В этом случае алгоритм должен уметь доказывать недостижимость таких запросов.

Предлагается доказывать недостижимость запросов с конечным уровнем посредством вывода *лемм* с конечным уровнем. Лемма для локации loc с уровнем lvl — это формула первого порядка, приближающая сверху состояния, достигающие локацию loc не более, чем за lvl шагов.

Однако лемм с конечным уровнем недостаточно для ответа на главные запросы, у которых бесконечный уровень. Лемма с *бесконечным* уровнем для локации loc — это формула первого порядка, приближающая сверху состояния, которые соответствуют путям исполнения, достигающим локацию loc за любое конечное число шагов.

Чтобы выводить леммы с бесконечным уровнем, алгоритм стремится находить множества *взаимно индуктивных* лемм. Взаимная индуктивность множества лемм означает, что из выполнимости всех лемм на уровне k следует выполнимость этих же лемм на уровне $k + 1$. Все взаимно индуктивные леммы являются леммами с бесконечным уровнем.

Используя леммы с бесконечным уровнем алгоритм получает возможность при необходимости эффективно отрицательно отвечать на главные запросы.

Однако можно представить такую ситуацию. Пусть в достаточно большой программе имеется далеко расположенный (относительно входной точки программы EP) метод M, у которого в некоторой локации кода имеется главный запрос. Предположим, что данная локация находится в недостижимом месте в коде («dead code»). Тогда будет неэффективно начинать процесс символьного исполнения из входной точки программы, поскольку это потребует больших вычислительных ресурсов. Для того, чтобы уметь эффективно отвечать на такой главный запрос, алгоритм мог бы стартовать символьное исполнение из входной точки метода M, сумев прийти к заключению, что данный запрос недостижим для всех путей исполнения, посетивших входную точку (локацию) метода M. Из этого знания можно было сделать вывод о том, что

ни один путь исполнения, стартующий из входной точки программы EP, не сможет положительно ответить на главный запрос потому, что для этого нужно оказаться внутри локации метода M, а это невозможно сделать, обойдя входную локацию метода M.

Эти рассуждения приводят к идее направлять символьное исполнение к запросам, т.е. стартовать символьное исполнение от произвольной (не обязательно стартовой) локации кода, чтобы оно быстрее достигло локации с запросом.

Теперь зададим следующий вопрос: а что, если мы стартовали символьное исполнение из произвольной локации кода (например, из входной точки метода), дошли до запроса, не превысив уровень запроса, и оказалось, что финальное состояние s выполняет свойство запроса? Эта ситуация говорит о том, что состояние s является *абстрактным* свидетелем данного запроса. В таком случае правомочным является следующий вопрос: а можем ли мы достигнуть точки старта состояния s (например, входной точки метода), а только потом пройти по пути, соответствующему состоянию s, до локации нашего основного запроса так, чтобы ответить на него.

Чтобы ответить на данный вопрос, нужно создать новый запрос, который будет обладать такими характеристиками (локация, свойство, уровень), что положительный ответ на него повлечет положительный ответ на исходный запрос.

2.3. Стратегия исследования

Чтобы сделать алгоритм максимально общим, было принято решение возложить ответственность выбор выполнения описанных выше действий на *стратегию исследования*, которая уже затрагивалась в главе «Обзор». Таким образом, множество действий стратегии исследования пополняется. А именно, на очередном шаге алгоритма стратегия может, кроме выбора состояния из очереди и, собственно, самого символьного исполнения следующей инструкции, также выбирать запрос и состояние, а затем пытаться распространить запрос к другой локации

кода; либо может создать пустое символьное состояние, соответствующее пути, начинающемуся из новой локации, и поместить его в очередь состояний.

Казалось бы очевидно, что эффективность предложенного алгоритма существенно зависит от эффективности реализации стратегии исследования, которая будет решать, в каком *порядке* алгоритм выполняет основные действия. Тем не менее, алгоритм спроектирован так, чтобы его корректность работы быть независима от этого порядка.

Здесь уместна следующая аналогия: алгоритм — это блоки, а стратегия исследования — это строитель, и если строитель умеет строить, то рано или поздно он сможет построить здание, т.е. ему всегда хватит этих блоков. Таким образом, стратегия исследования будет влиять на скорость сходимости алгоритма, но не на его корректность.

2.4. Типы данных, глобальные переменные

В процессе работы алгоритм накапливает информацию в глобальных переменных. Прежде чем описывать их, нужно описать их типы (см. листинг 1). Во-первых, необходим тип для запросов *pob* (Proof Obligation), который содержит три поля: локацию loc, формулу φ и уровень lvl (см. подглаву 2.1). Во-вторых, тип символьного состояния для классического алгоритма (см. раздел 1.1) необходимо было дополнить новыми полями:

- loc_0 — стартовая локация, из которой начинается символьное исполнение;
- lvl — количество уже пройденных шагов;
- *pobs* — множество запросов, которые потенциально могут засвидетельствовать состояние.

```
type pob = { loc : loc; φ : formula; lvl : level }
type state = {
    loc : loc; PC : formula; store : store;
    loc_0 : formula; lvl : level; pobs : Set<pob> }
```

Листинг 1: Типы глобальных переменных: тип запроса (pob) и типа символьного состояния (state)

В процессе своей работы алгоритм использует и обновляет следующие глобальные переменные. Уровень $curLvl$ означает текущую итерацию алгоритма. Множество $mainPobs$ содержит *главные* запросы, на которые алгоритм еще не ответил. Множество $pobs$ содержит аналоги главных запросов с конечным уровнем $curLvl$. Множество Q_f содержит символьные состояния, готовые для исполнения. Множество Q_b содержит пары запросов и состояний (p, s), готовые для распространения «назад».

```
curLvl : level
mainPobs : Set<pob>
pobs : Set<pob>
Q_f : Set<state>
Q_b : Set<pob × state>
witnesses : pob → Set<state>
blockedLocs : pob → Set<loc>
pobsLocs : Set<loc>
T : loc → Set<state>
L : loc × level → Set<formula>
```

Листинг 2: Объявление глобальных переменных

Глобальные переменные $witnesses$ и $blockedLocs$ были введены для того, чтобы алгоритм был способен отвечать на запросы отрицательно. Функция $witnesses$ сопоставляет запросу p множество состояний, которые могут его *засвидетельствовать*, причем само состояние тоже хранит запрос $p \in s.pobs$.

Функция *blockedLocs* сопоставляет запросу множество локаций кода, которые его блокируют, т.е. любое состояние, посетившее заблокированную локацию, *точно не засвидетельствует* запрос p.

Поскольку алгоритм будет начинать символьное исполнение из произвольной локации кода loc_0, то на каждую такую локацию может быть распространен «назад» запрос-ребенок. Чтобы впоследствии корректно ответить на этот запрос, нужно помнить все символьные состояния, пути которых проходят через локацию loc_0. Для решения этой задачи были заведено множество *pobsLocs* и отображение T. Обе глобальные переменные обновляются динамически, во время работы алгоритма. Множество *pobsLocs* хранит все локации, к которым впоследствии могут быть перенаправлены запросы-потомки. Отображение T сопоставляет локации кода множество символьных состояний, соответствующий путь исполнения которых посетил данную локацию. Она будет использована для распространения запросов «назад», а также для вывода лемм, которые хранятся в отображении $L(loc, lvl)$. Это отображение сопоставляет локации кода и уровню некоторое множество лемм, которые приближают сверху символьные состояния, соответствующий путь символьного исполнения которых остановился в локации loc и который содержит не более lvl шагов.

2.5. Алгоритм

Детально опишем предложенный алгоритм, который, в свою очередь, состоит из отдельных процедур. Основная процедура *PropDirSymExec* дает старт алгоритму и пытается последовательно отвечать на запросы. Процедуры *addWitness* и *blockWitness* поддерживают актуальную информацию о состояниях символьного исполнения, которые могут засвидетельствовать запросы. Процедура *start* порождает символьное состояние с произвольной стартовой локацией. Процедура *forward* получает символьное состояние, исполняет на нем инструкцию и порождает новые символьные состояния. Процедура *backward* пытается отвечать на запросы. Процедура *checkInductive* проверяет взаимную индуктив-

ность лемм и отвечает на главные запросы.

Кроме того, алгоритм пользуется следующими *внешними* функциями:

- *isSAT* (*isUNSAT*) проверяет выполнимость (невыполнимость) формулы первого порядка при помощи SMT-решателя;
- *answerYes* закрывает запрос с положительным вердиктом, а *answerNo* — с отрицательным;
- *strategy.ChooseAction* выбирает, какое действие будет исполнять *стратегия исследования* на текущей итерации;
- *nextLevel* увеличивает значение уровня;
- *canReach*($loc_1, loc_2, locs$) проверяет наличие межпроцедурного пути в графе потока управления от локации loc_1 в локацию loc_2, не посещающего ни одну из локаций *locs*;
- *generalize*(φ, ψ) выводит лемму для невыполнимой конъюнкции формул, например, с помощью интерполяции Крейга [8], либо с помощью вычисления ядра невыполнимости [9];
- *executeInstruction* символьно исполняет инструкцию, соответствующую очередной локации состояния;
- *mkPrime*(φ) заменяет все переменные в формуле на штрихованные аналоги, обозначающие значения переменных в «следующем» состоянии.

2.5.1. Основные процедуры

Главная процедура *PropDirSymExec* принимает на вход множество локаций кода и программу. В самом начале она подготавливает для своей работы глобальные переменные. Процедура работает, пока все главные запросы не отвечены. Она работает итеративно, и на каждой итерации для еще не отвеченных главных запросов создаются конечные аналоги — запросы с уровнем *curLvl*. После чего алгоритм пытается ответить

на все конечные запросы. Назовем данный процесс минорным. На каждой минорной итерации *стратегия исследования* выбирает действие, которое будет выполнять алгоритм: *start*, *forward* и *backward*. После ответа на все конечные запросы минорная итерация алгоритма заканчивается, алгоритм увеличивает уровень $curLvl$. Если все главные запросы были отвечены, то алгоритм печатает «ответы» на главные запросы и завершает свою работу.

Процедура 1: Алгоритм направляемого свойством символьного исполнения

```
1  Procedure PropDirSymExec(locs : Set<loc>, P : Program )
2      ИНИЦИАЛИЗИРОВАТЬ ВСЕ ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ;
3      while mainPobs ≠ ∅ do
4          pobs := ∅;
5          forall p ∈ mainPobs do pobs := pobs ∪ {(p.loc, curLvl, ⊤)} ;
6          while pobs ≠ ∅ do
7              switch strategy.ChooseAction(P, Q_f, Q_b) do
8                  case START(loc) do start(loc) ;
9                  case GOFRONT(s : state) do forward(s) ;
10                 case GOBACK(p' : pob, s' : state) do backward(p',
                       s') ;
11         curLvl := nextLevel(curLvl);
12     // Reporting status of all basic queries;
```

Процедура *start* создает пустое символьное состояние, стартующее из произвольной локации кода loc. Для нового состояния перебираются *все* запросы программы, для которых вызывается процедура $addWitness$, поскольку новое состояние может потенциально *засвидетельствовать* любой запрос. Критически важно то, чтобы стратегия исследования *никогда не* выбирала локацию loc, которая когда-либо была посещена символьным исполнением (пусть состоянием s). В противном случае состояние s может быть не добавлено в множество состояний, достигающих локацию loc, что может повлечь неправильные ответы на запросы для данной локации.

Процедура 2: Создание нового символьного состояния, стартующего из заданной локации кода

```
1  Procedure start(loc : loc)
2      startState := (loc, ⊤, ∅, loc, 0, ∅);
3      Q_f := Q_f ∪ {startState};
4      T(loc) := ∅;
5      forall p ∈ pobs do addWitness(s, p) ;
```

Процедура *forward* принимает состояние s и символьно исполняет инструкцию на текущей локации $s.loc$, после чего обновляет глобальные переменные, чтобы сохранить инварианты алгоритма. Во-первых, каждое порожденное состояние s' добавляется в множество Q_f. Во-вторых, нужно обновить отображение T: если мы достигли локации кода, от которой стартовало пустое символьное состояние, то к данной локации могут распространиться «назад» запросы, на которые состояние s' может ответить. В самом конце остается убрать нерелевантные для состояния s запросы.

Процедура 3: Порождение новых символьных состояний

```
1  Procedure forward(s : state)
2      Q_f := Q_f \ {s};
3      forall s' ∈ executeInstruction(s, s.loc) do
4          assert(isSAT(s'.PC));
5          Q_f := Q_f ∪ {s'};
6          if s'.loc ∈ pobsLocs then T(s'.loc) := T(s'.loc) ∪ {s'} ;
7          forall p ∈ s.pobs do
8              addWitness(s', p);
9              if p.loc = s'.loc then Q_b := Q_b ∪ {(s', p)} ;
10     forall p ∈ s.pobs do
11         if p.loc ≠ s.loc then blockWitness(s, p) ;
```

Ключевой процедурой алгоритма является *backward*, которая отвечает на запросы. Для этого вычисляется формула $WLP(s', p')$ — слабейшая формула, которой должно удовлетворять состояние, чтобы при

исполнении такого же пути, что и состояние s', выполнить свойство запроса $p'.\varphi$. Если конъюнкция φ такой формулы и всех лемм на соответствующем уровне не выполнима, то запрос не может быть засвидетельствован для состояния (вызывается $blockWitness(s', p')$). Если после этого оказалось, что заблокированы все локации, через которые можно попасть в локацию запроса, то запрос отвечен отрицательно, а для локации $p'.loc$ вычисляется лемма, являющаяся обобщением (*generalize*) аппроксимации $apxm$ поведения программы на уровне $p'.lvl$ и свойства запроса $p'.\varphi$. После вывода леммы вызывается проверки лемм на взаимную индуктивность.

Запрос может быть засвидетельствован по состоянию s', если формула φ выполнима. Теперь нужно определить каким именно свидетелем является состояние s'. Если стартовая локация $s'.loc_0$ равна входной точке программы $P.EP$, то s' — настоящий свидетель запроса p', и поэтому можно ответить положительно на запрос p' и на всех его предков (процедура *answerYes*). В противном случае, s' — абстрактный свидетель свойства p', для которого создается запрос-ребенок p с формулой φ, который привязывается к стартовой локации $s'.loc_0$ с уровнем равным разности уровней запроса $p'.lvl$ и уровня состояния $s'.lvl$.

Процедура 4: Распространение запросов «назад»

```
1  Procedure backward(p', s', P)
2      Q_b := Q_b \ {(p', s')};
3      assert(s'.loc= p'.loc);
4      lvl := p'.lvl − s'.lvl;
5      Ψ := WLP(s', p'.φ) ∧ overApxm(s'.loc_0, lvl);
6      if Ψ is SAT then
7          if p'.loc = P.EP then answerYes(p') ;
8          else
9              p := (s'.loc_0, lvl, Ψ);
10             pobs := pobs ∪ {p};
11             child(p') := child(p') ∪ {p};
12             forall s ∈ witnesses(p') do addWitness(s, p) ;
13             forall s ∈ T(p.loc) do Q_b := Q_b ∪ {(p, s)} ;
14     else
15         blockWitness(s', p');
16         if canReach(P.EP, p'.loc, blockedLocs(p')) then return;
17         answerNo(p');
18         apxm := ⊥;
19         forall s ∈ T(p'.loc) | s.loc_0 ∈ blockedLocs(p') do
20             apxm := apxm ∨ overApxm(s.loc_0, p'.lvl − s.lvl) ∧
               encode(s)
21         L(p'.loc, p'.lvl) := L(p'.loc, p'.lvl) ∪ {generalize(apxm, p'.φ)};
22         checkInductive();
```

Процедура *checkInductive* распространяет леммы на следующие уровни. Это делается инкрементально от самого низкого уровня 0 до текущего уровня $curLvl$. Если так случилось, что все леммы одного уровня были распространены, то эти леммы являются взаимно индуктивными, т.е. они все влекут друг друга. А значит, они выполняются *всегда* и их можно перевести на уровень ∞. С помощью таких лемм можно ответить отрицательно на *главные* запросы. Чтобы проверить может

ли лемма $l \in L(loc, lvl)$ быть переведена на следующий уровень, нужно проверить, следует ли она из всех формул, приближающих сверху состояния программы, достигшие локации loc с уровнем, не меньшем, чем её текущий уровень lvl. Из всех таких уровней выбирается минимальный. Он и будет новым уровнем $newLvl$ для леммы.

Процедура 5: Проверка *взаимной индуктивности* лемм

```
1  Function checkInductive()
2      for lvl = 0 to curLvl do
3          ind := true;
4          propagatedLemmas := ∅;
5          forall l ∈ L(loc, lvl) | loc ∈ pobsLocs do
6              blocked := false;
7              newLvl := ∞;
8              forall s ∈ T(loc) do
9                  propagatedForState := false;
10                 for lvl' = curLvl downto lvl − s.lvl do
11                     if
                        isUNSAT( WLP(s, ¬l) ∧ overApxm(s.loc_0, lvl'))
                        then
12                         propagatedForState := true;
13                         newLvl := min(newLvl, lvl' + s.lvl);
14                         break;
15                 if not propagatedForState then
16                     blocked := true;
17                     ind := false;
18             if not blocked then
19                 L(loc, lvl) := L(loc, lvl) \ {l};
20                 L(loc, newlvl) := L(loc, newlvl) ∪ {l};
21                 propagatedLemmas := propagatedLemmas ∪
                    {(loc, l, newLvl)};
22         if ind then
23             forall (loc, newLvl, l) ∈ propagatedLemmas do
24                 L(loc, newLvl) := L(loc, newLvl) \ {l};
25                 L(loc, ∞) := L(loc, ∞) ∪ {l};
26                 // try to block main pobs
```

2.5.2. Вспомогательные процедуры и функции

Опишем смысл «вспомогательных» функций алгоритма, которыми пользуются «основные».

Чтобы поддерживать в актуальном состоянии информацию о запросах и состояниях, которые могут их засвидетельствовать, алгоритм использует вспомогательные процедуры **addWitness** и **blockWitness**.

Процедура *addWitness* принимает состояние s и запрос p и пытается определить, может ли состояние быть свидетелем запроса. Для этого проверяются два необходимых условия. Во-первых, проверяется уровень, а во-вторых, может ли состояние s достичь вершины $p.loc$ по межпроцедурному графу потока управления программы, не посещая заблокированных локаций из множества $blockedLocs(p)$. Если оба условия выполнены, то обновляются множества $witnesses(p)$ и $s.pobs$ соответствующим образом.

Процедура *blockWitness* убирает состояние s из множества свидетелей запроса p. Если больше нет свидетелей запроса, стартующих из локации $s.loc_0$, то локация $s.loc_0$ блокируется. После этого фильтруется множество свидетелей запроса, поскольку некоторые состояния могли достигать локации запроса $p.loc$, только через заблокированную локацию $s.loc_0$. Для таких состояний процедура *blockWitness* вызывается рекурсивно.

Процедуры 6: Обновление потенциальных свидетелей для запросов

```
1  Procedure addWitness(s, p)
2      if s.lvl ≤ p.lvl and canReach(s.loc, p.loc, blockedLocs(p)) then
3          witnesses(p) := witnesses(p) ∪ {s};
4          s.pobs := s.pobs ∪ {p};
5  Procedure blockWitness(s', p')
6      s'.pobs := s'.pobs \ {p'};
7      witnesses(p') := witnesses(p') \ {s'};
8      forall s ∈ witnesses(p') do
9          if s.loc_0 = s'.loc_0 then return;
10     blockedLocs(p') := blockedLocs(p') ∪ {s'.loc_0};
11     forall s ∈ witnesses(p') do
12         if not canReach(s.loc, p'.loc, blockedLocs(p')) then
13             blockWitness(s, p');
```

Функция *overApxm* вычисляет формулу, приближающую сверху достижимые состояния для заданной локации loc и уровня lvl. Она рассматривает все уровни $lvl' \geq lvl$ и вычисляет конъюнкцию лемм каждого уровня $L(loc, lvl')$. Условие $lvl' \geq lvl$ выбрано намеренно, чтобы не копировать, а перемещать леммы от уровня к более высокому уровню.

Функция *encode* кодирует состояние в формулу. Во-первых, она учитывает условие пути ($s.PC$). Во-вторых, кодирует память ($s.store$) в конъюнкцию формул вида $mkPrime(x) = expr$, где $mkPrime(x)$ — новая переменная x' для обозначения значения переменной x после исполнения символьного состояния s; $expr$ — значение новой переменной x' в терминах исходных значений переменных.

Функция $WLP(s, \varphi)$ — слабейшая формула, которой должно удовлетворять состояние, чтобы при исполнении такого же пути, что и состояние s, выполнить формулу φ.

Функции 7: overApxm вычисляет формулу, приближающую сверху состояния для заданной локации; **encode** кодирует состояние в формулу первого порядка; **WLP** вычисляет слабейшее предусловие для формулы и состояния

```
1  Function overApxm(loc, lvl)
2      apxm := ⊥;
3      for lvl' ≥ lvl do
4          apxm := apxm ∧ ⋀ L(loc, lvl');
5      return apxm;
6  Function encode(s)
7      ψ := s.PC;
8      forall (x, expr) ∈ s.store do
9          ψ := ψ ∧ mkPrime(x) = expr;
10     return ψ;
11 Function WLP(s, φ)
12     return encode(s) ∧ mkPrime(φ);
```

2.6. Пример

Продемонстрируем работу алгоритма для доказательства недостижимости главной локации кода. Для этого выберем показательный и известный [5] пример, для доказательства которого нужно вывести взаимно индуктивные леммы. Пример представлен в следующем листинге:

Алгоритм 8: Пример программы, для доказательства корректности которой необходимо вывести взаимно индуктивные леммы

```
1 Function F()
2     x := 1; y := 1;
3     while nondet() do
4         x := x + y;
5         y := x;
6     if x ≤ 0 then
7         fail();
```

На листинге внутри функции F есть цикл, который исполняется **недетерминированное** число раз. После исполнения цикла может случиться падение программы, если значение переменной x окажется меньше либо равно 0. Из кода программы человеку очевидно, что падение программы никогда не случится (сложение происходит без переполнения). Хочется, чтобы *алгоритм* смог автоматически *доказать*, что падения программы *никогда* не случится.

2.6.1. Подготовительная работа

В рамках примера будем считать, что локация — это номер инструкции внутри метода F. В программе будет всего один главный запрос $p_M = (7, \top, \infty)$, спрашивающий, можно ли достичь локации 7 (инструкции **fail**). Чтобы ускорить процесс трассировки алгоритма, будем считать, что стратегия исследования самая благоприятная. А именно, представим, что стратегия поиска решила стартовать (**start**) символьное исполнение из локаций $2, 3, 5$ и получила (**forward**) следующие символьные

состояния:

$$
\begin{aligned}
\boldsymbol{s_1} = \quad & (\boldsymbol{loc_0} = 2, && \mathbf{loc} = 3, && \mathbf{PC} = \top, \\
& \ \mathbf{store} = \{x' \leftarrow 1; y' \leftarrow 1\}, && \mathbf{lvl} = 0, && \mathbf{pobs} = \{p_M\}), \\
\boldsymbol{s_2} = \quad & (\boldsymbol{loc_0} = 3, && \mathbf{loc} = 5, && \mathbf{PC} = \top, \\
& \ \mathbf{store} = \{x' \leftarrow x + y; y' \leftarrow y\}, && \mathbf{lvl} = 0, && \mathbf{pobs} = \{p_M\}), \\
\boldsymbol{s_3} = \quad & (\boldsymbol{loc_0} = 5, && \mathbf{loc} = 3, && \mathbf{PC} = \top, \\
& \ \mathbf{store} = \{y' \leftarrow x; x' \leftarrow x\}, && \mathbf{lvl} = 1, && \mathbf{pobs} = \{p_M\}), \\
\boldsymbol{s_4} = \quad & (\boldsymbol{loc_0} = 3, && \mathbf{loc} = 7, && \mathbf{PC} = x <= 0, \\
& \ \mathbf{store} = \{y' \leftarrow y; x' \leftarrow x\}, && \mathbf{lvl} = 0, && \mathbf{pobs} = \{p_M\}).
\end{aligned}
$$

В процессе работы алгоритма нужно будет кодировать состояния в формулы, поэтому сразу напишем, чему равны результаты кодирования. Кодирование состояния — это формула, отражающая изменение переменных и условие пути до локации. Новые значения переменных помечаются штрихами.

$$
\begin{aligned}
\boldsymbol{encode(s_1)} &\equiv && x' = 1 \wedge y' = 1 \\
\boldsymbol{encode(s_2)} &\equiv && x' = x + y \wedge y' = y \\
\boldsymbol{encode(s_3)} &\equiv && x' = x \wedge y' = x \\
\boldsymbol{encode(s_4)} &\equiv && x \leq 0 \wedge x' = x \wedge y' = y
\end{aligned}
$$

Поскольку условие выхода из цикла не детерминировано, для упрощения ситуации будем считать, что исполнение всегда может как продолжить, так и закончить цикл, поэтому условия пути PC у состояний s_2, s_4 не учитывают изменения состояния, которые производит функция **nondet()**(). Важно то, что это не умаляет общности. Следует обратить внимание, что уровень у третьего состояния s_3 равен 1. Это сделано намеренно, чтобы отразить то, что путь представляет собой обратное ребро в графе потока управления. В общем случае уровень **всегда** должен учитывать циклы и рекурсию, иначе просто алгоритм не будет

сходиться для доказательства недостижимости главных запросов.

2.6.2. Трассировка

После того, как эти символьные состояния были получены и добавлены в глобальную переменную T, которая хранит состояния, могущие засвидетельствовать запросы, алгоритм будет пытаться ответить на запрос p_M отрицательно. Для этого стратегия исследования в основном алгоритм всегда выбирает действие GoBack и вызывает функцию **backward** (см. *Алгоритм* 4).

Уровень 0. В начале работы алгоритма, до достижения состояний s_1, s_2, s_3, s_4, был заведен конечный аналог главного запроса $p_0 = (7, \top, 0)$, $witnesses(p_0) = \{s_4, s_1\}$, $blockedLocs(p_0) = \{5\}$.

PropDirSymExec вызывает backward(p_0, s_4, P).

Строка 5. Формула $\varphi \equiv WLP(s_4, p_0.\varphi) \wedge L(3,0) \equiv encode(s_4) \wedge mkPrime(p_0.\varphi) \wedge \top \equiv x <= 0 \wedge x' = x \wedge y' = y \wedge \top$ является выполнимой.

Строки 9 – 13. Создается запрос $p_1 = (3, x <= 0, 0)$ и обновляются глобальные переменные: $witnesses(p_1) = \{s_1\}$, $blockedLocs(p_1) = \{5\}$, поскольку у $s_2.lvl = 1$.

PropDirSymExec вызывает backward(p_1, s_1, P).

Строка 5. Формула $\varphi \equiv WLP(s_1, p_1.\varphi) \wedge L(2,0) \equiv encode(s_1) \wedge mkPrime(p_1.\varphi) \wedge \top \equiv \top \wedge (x' = 1 \wedge y' = 1) \wedge x' <= 0 \wedge \top$ является невыполнимой.

Строка 15. $witnesses(p_1) = \varnothing$. $blockedLocs(p_1) = \{2, 5\}$.

Строка 17. На запрос p_1 получен отрицательный ответ.

Строка 18. $apxm \equiv L(2,0) \wedge encode(s_1) \vee L(5,-1) \wedge encode(s_3) \equiv \top \wedge x' = 1 \wedge y' = 1 \vee \bot \wedge x' = x \wedge y' = x \equiv x' = 1 \wedge y' = 1$.

Строка 19. $generalize(x' = 1 \wedge y' = 1, x' <= 0)$ возвращает лемму $x = 1$. $L(3,0) = \{x = 1\}$

Строка 20. Вызов **checkInductive**, который не протолкнет лемму вперед.

Вызов PropDirSymExec вызывает backward(p_0, s_4, P).

Строка 5. Формула $\varphi \equiv WLP(s_4, p_0.\varphi) \wedge L(3,0) \equiv x <= 0 \wedge x' = x \wedge y' = y \wedge x = 1$ является невыполнимой.

Строка 15. $witnesses(p_0) = \varnothing$. $blockedLocs(p_0) = \{2, 3, 5\}$.

Строка 17. На запрос p_0 получен отрицательный ответ.

Строка 18. $apxm \equiv L(3,0) \wedge encode(s_4) \equiv x = 1 \wedge x <= 0 \wedge x' = x \wedge y' = y \equiv \bot$.

Строка 19. *generalize*$(\bot, \top)$ вернет лемму $\bot$. $L(7,0) = \{\bot\}$ Смысл такой леммы в том, чтобы показать недостижимость локации 7, если у состояния уровень 0.

Строка 20. Вызов **checkInductive**, который не протолкнет лемму вперед.

Уровень 0 завершился. Поскольку главный запрос остался без ответа, начнется уровень 1.

Уровень 1. $p_0 = (7, \top, 1)$, $witnesses(p_0) = \{s_1, s_2, s_3, s_4\}$, $blockedLocs(p_0) = \varnothing$.

PropDirSymExec вызывает backward(p_0, s_4, P).

Строка 5. Формула $\varphi \equiv WLP(s_4, p_0.\varphi) \wedge L(3,1) \equiv encode(s_4) \wedge mkPrime(p_0.\varphi) \wedge \top \equiv x <= 0 \wedge (x' = x \wedge y' = y) \wedge \top \wedge \top$ является выполнимой.

Строки 9 – 13. Будет создан запрос $p_1 = (3, x <= 0, 1)$ и будут обновлены глобальные переменные: $witnesses(p_1) = \{s_1, s_2, s_3\}$, $blockedLocs(p_1) = \varnothing$.

PropDirSymExec вызывает backward(p_1, s_1, P).

Строка 5. Формула $\varphi \equiv WLP(s_1, p_1.\varphi) \wedge L(2,1) \equiv encode(s_1) \wedge mkPrime(p_1.\varphi) \wedge \top \equiv \top \wedge (x' = 1 \wedge y' = 1) \wedge x' <= 0 \wedge \top \equiv x' <= 0 \wedge x' = 1 \wedge y' = 1$ является не выполнимой.

Строка 15. $witnesses(p_1) = \{s_2, s_3\}$. $blockedLocs(p_0) = \{2\}$.

PropDirSymExec вызывает backward(p_1, s_3, P).

Строка 5. Формула $\varphi \equiv WLP(s_3, p_1.\varphi) \wedge L(5,0) \equiv encode(s_3) \wedge mkPrime(p_1.\varphi) \wedge \top \equiv \top \wedge (x' = x \wedge y' = x) \wedge x' <= 0 \wedge \top$ является выполнимой.

Строки 9 – 13. Будет создан запрос $p_2 = (5, y <= 0 \wedge x <= 0, 0)$ и будет обновлены глобальные переменные: $witnesses(p_2) = \{s_1, s_2, s_3\}$, $blockedLocs(p_2) = \varnothing$.

PropDirSymExec вызывает backward(p_2, s_2, P).

Строка 5. Формула $\varphi \equiv WLP(s_2, p_2.\varphi) \wedge L(3,0) \equiv encode(s_2) \wedge mkPrime(p_2.\varphi) \wedge x = 1 \equiv \top \wedge (x' = x + y \wedge y' = y) \wedge (y' <= 0 \wedge x' <= 0) \wedge x = 1 \equiv x + y <= 0 \wedge y <= 0 \wedge x = 1 \equiv 1 + y <= 0 \wedge y <= 0$ является выполнимой.

Строки 9 – 13. Будет создан запрос $p_3 = (3, 1 + y <= 0 \wedge y <= 0, 0)$, так что $witnesses(p_3) = \{s_1\}, blockedLocs(p_3) = \{5\}$.

Вызов PropDirSymExec вызывает backward(p_3, s_1, P).

Строка 5. Формула $\varphi \equiv WLP(s_1, p_3.\varphi) \wedge L(2,0) \equiv encode(s_1) \wedge mkPrime(p_3.\varphi) \wedge \top \equiv \top \wedge x' = 1 \wedge y' = 1 \wedge 1 + y' <= 0 \wedge y' <= 0 \wedge \top \equiv x' = 1 \wedge y' = 1 \wedge 1 + y' <= 0 \wedge y' <= 0 \equiv \bot$ является невыполнимой.

Строка 15. $witnesses(p_3) = \varnothing$. $blockedLocs(p_3) = \{2; 5\}$.

Строка 17. На запрос p_3 получен отрицательный ответ.

Строка 18. $apxm \equiv L(2,0) \wedge encode(s_1) \vee L(5,-1) \wedge encode(s_3) \equiv \top \wedge x' = 1 \wedge y' = 1 \vee \bot \equiv x' = 1 \wedge y' = 1$.

Строка 19. $generalize(x' = 1 \wedge y' = 1; y' <= 0 \wedge 1 + y' <= 0)$ вернет лемму $x >= 1 \wedge y >= 1$. $L(3,0) = \{x = 1; x >= 1 \wedge y >= 1\}$.

Строка 20. Вызов **checkInductive**, который не протолкнет лемму вперед.

Вызов PropDirSymExec вызывает backward(p_2, s_2, P).

Строка 5. Формула $\varphi \equiv WLP(s_2, x <= 0 \wedge y <= 0) \wedge L(3,0) \equiv encode(s_2) \wedge mkPrime(x <= 0 \wedge y$ $x = 1 \wedge x >= 1 \wedge y >= 1 \equiv x' = x + y \wedge y' = y \wedge x' <= 0 \wedge y' <= 0 \wedge x = 1 \wedge x >= 1 \wedge y >= 1 \equiv x + y <= 0 \wedge y <= 0 \wedge x >= 1 \wedge y >= 1$ является невыполнимой.

Строка 15. Локация 3 была заблокирована для запроса p_2. $witnesses(p_2) = \varnothing$. $blockedLocs(p_2) = \{3\}$.

Строка 17. На запрос p_2 получен отрицательный ответ.

Строка 18. $apxm \equiv L(3,0) \wedge encode(s_2) \equiv x = 1 \wedge x >= 1 \wedge y >= 1 \wedge x' = x + y \wedge y' = y \equiv x = 1 \wedge y >= 1 \wedge x' = x + y \wedge y' = y$.

Строка 19. $generalize(x = 1 \wedge y >= 1 \wedge x' = x + y \wedge y' = y; x' <= 0 \wedge y' <= 0)$ вернет лемму $x >= 1 \wedge y >= 1$. $L(5,0) = \{x >= 1 \wedge y >= 1\}$.

Строка 20. Вызов **checkInductive**. Леммы окажутся индуктивными — сначала лемма $l_1 \equiv x >= 1 \wedge y >= 1 \in L(3,0)$ перейдет в новое множество $L(3,1)$, соответствующее уровню 1. Затем лемма $l_2 \equiv x = 1 \in L(3,0)$ не будет переведена на уровень 1. Затем лемма $l_3 \equiv x >= 1 \wedge y >= 1 \in L(5,0)$ с помощью только что переведенной леммы l_1 будет переведена в новое множество $L(5,1)$. После чего уровень рассматриваемых лемм увеличится с 0 до 1, и все леммы первого уровня l_1, l_3 будут переведены на уровень ∞. На запрос $p_1 = (3, x <= 0, 1)$ будет получен отрицательный ответ в связи с переводом леммы l_1 на бесконечный уровень.

Вызов PropDirSymExec вызывает backward(p_0, s_4, P).

Строка 5. Формула $\varphi \equiv WLP(s_4, \top) \wedge L(3,1) \equiv encode(s_4) \wedge mkPrime(\top) \wedge x >= 1 \wedge y >= 1 \equiv x <= 0 \wedge x' = x \wedge y' = y \wedge \top \wedge x >= 1 \wedge y >= 1 \equiv x <= 0 \wedge x >= 1 \wedge y >= 1 \equiv \bot$ является невыполнимой.

Строка 15. Локация 3 была заблокирована для запроса, из чего $witnesses(p_0) = \varnothing$. $blockedLocs(p_0) = \{3, 5, 2\}$.

Строка 17. На запрос p_0 получен отрицательный ответ.

Строка 18. $apxm \equiv L(3,1) \wedge encode(s_4) \equiv x >= 1 \wedge y >= 1 \wedge x <= 0 \wedge x' = x \wedge y' = y \equiv$

$\bot$.

Строка 19. $L(7,1) = \{\bot\}$.

Строка 20. Вызов **checkInductive** покажет, что лемма $l_1 \equiv \bot \in L(7,1)$ взаимно индуктивна с леммой $l_2 \equiv x >= 1 \wedge y >= 1 \in L(3,\infty)$, после чего будет переведена на бесконечный уровень ($l_1 \in L(7,\infty)$). И уже после этого будет закрыт главный запрос p_M.

Хочется отметить, что важную роль в алгоритме играет процедура вывода лемм (*generalize*), поскольку от нее зависит их взаимная индуктивность. Например, валидными леммами для нашего примера были бы леммы, кодирующие состояния, в точности исполнившие тело цикла «уровень» число раз: $L(3,0) = \{x = 1 \wedge y = 1\}$, $L(3,1) = \{x = 1 \wedge y = 1 \vee x = 2 \wedge y = 2\}$, $L(3,2) = \{x = 1 \wedge y = 1 \vee x = 2 \wedge y = 2 \vee x = 4 \wedge y = 4\}$ и т.д. и т.п. Однако такая последовательность лемм не является индуктивной и не смогла бы ответить на *главный* запрос.

3. Реализация на платформе .NET

Предложенный алгоритм был реализован на платформе .NET в рамках системы V#.

3.1. Система V#

Система V# представляет собой композициональную символьную виртуальную машину для анализа программ, предназначенных для исполнения на платформе .NET. Система V# может символьно исполнять инструкции промежуточного языка (CIL) платформы .NET и получать новые символьные состояния при помощи композиции имеющихся.

Возможность выполнять композицию символьных состояний не только позволяет избежать повторного исполнения одного и того же кода в программе, но и является средством вычисления слабейшего предусловия для формул (**WLP**). Данное предусловие используется в алгоритме для делегирования запросов к целевым локациям кода исполняемой программы.

Схема работы тестовой подсистемы V# представлена на рис. 1. Тестовая подсистема позволяет символьно исполнить произвольный метод программы и породить финальные состояния, включающие следующую информацию: содержимое кучи программы, значения статических переменных, условия пути, возвращаемое значение метода и т.д.

Сначала исходный код метода подается на вход парсеру, который выполняет синтаксический анализ и выдает мета-информацию, необходимую для работы алгоритма символьного исполнения: граф потока управления, компоненты сильной связности этого графа, обратные ребра, ведущие в начальные вершины циклов и т.д. Далее стартует классический алгоритм символьного исполнения, который пытается достичь инструкции выхода из метода, и тем самым получить финальные состояния. После достижения всех финальных состояний последние проверяются тестовой подсистемой на соответствие с ожидаемыми результатами. После этой проверки появляется результат теста.

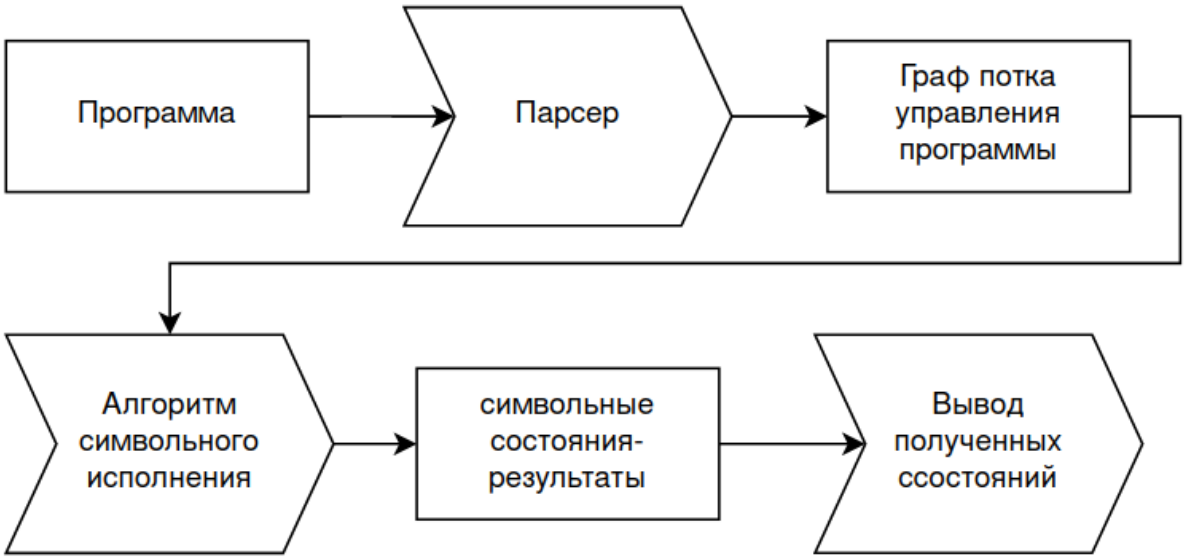

Рис. 1: Схема работы существующей тестовой подсистемы системы V#

3.2. Схема работы направляемой тестовой подсистемы

Поскольку в систему V# был встроен новый алгоритм направляемого свойством символьного исполнения, целью которого является проверка достижимости заданных локаций кода, в системе реализована новая схема работы тестовой подсистемы — см. рис. 2. Исходный код можно найти на GitHub [2]

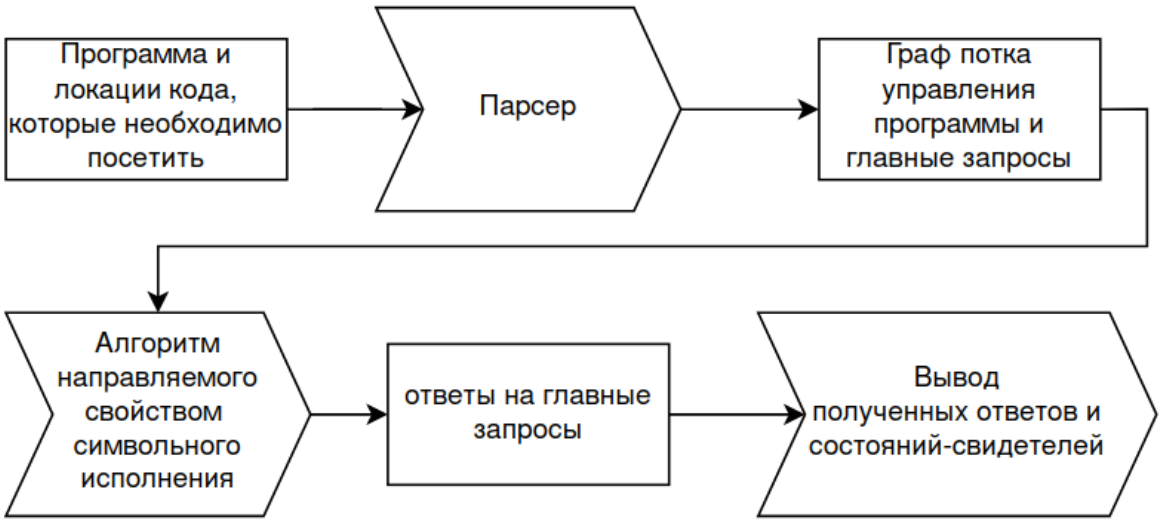

Рис. 2: Новая схема работы тестовой подсистемы

Опишем подробно эту схему. На системе V# подается *множество методов и локаций кода*. Далее парсер байткода, как и прежде, создает мета-информацию для методов и *главные запросы*, на которые необходимо ответить. Затем запускается алгоритм направляемого свойством символьного исполнения, которые пытается ответить на главные запросы при помощи мета-информации. Результаты работы алгоритма — полученные ответы на главные запросы, — далее анализируются тестовой подсистемой, которая не только сравнивает статус ответа на главные запросы, но и создаёт отчет о проделанной работе (время работы алгоритма, количество порожденных состояний, количество исполненных инструкций). Этот отчет нужен для сравнения работы алгоритма для разных стратегий исследования кода.

3.3. Реализация стратегий исследований кода

Эффективность алгоритмов символьного и направляемого свойством символьного исполнения зависят от выбора стратегии исследования (см. раздел 2.3). В рамках данной работы, для предложенного алгоритма, были реализованы следующие стратегии исследования: *dfs-стратегия*, *bfs-стратегия*, *направленная стратегия*. Первые две являются классическими в символьном исполнении [13], последняя разработана в рамках данной работы специально для поддержки направленного к локациям символьного исполнения в системе V#.

3.3.1. DFS- и BFS- стратегии

DFS-стратегия исследует программу и строит трассу исполнения, уходящую как можно дальше вглубь исследования.

Она реализует следующую последовательность действий. Символьное исполнение начинается из входной точки программы. Если множество распространения «назад» Q_b не пусто, то стратегия выбирает какой-то элемент и распространяет его «назад». Если же «множество» Q_b пусто, то очередь выбирает состояние из множества состояний для распространения «вперед» Q_f. Для данной стратегии «множество» Q_f

представляет из себя структуру данных в виде стека. Как раз с помощью стека символьное исполнение идет вглубь: стратегия выбирает одного из потомков выбранного на предыдущей итерации символьного состояния.

BFS-стратегия выбирает по очереди все пути исполнения программы, не выделяя какой-либо отдельный путь. Она придерживается такого же порядка действий, что и DFS-стратегия, только в данном случае множество состояний для распространения «вперед» Q_f представляет из себя структуру данных очередь.

3.3.2. Направленная стратегия

Направленная стратегия пытается достигать локаций с запросами, начиная с входной точки программы и учитывая структуру программы. А именно, в самом начале работы алгоритма, стратегия строит отображения *reachableLocationsOfLoc*, *reachableMethodsOfLoc*, *reachableMethodsOfMethod*. Эти отображения используются для определения достижимости локации с запросом, а также для оценки количества действий, которые будут выполнены.

- Отображение *reachableLocationsOfLoc* сопоставляет локации множество достижимых локаций внутри данного метода. Данное отображение дает наименее грубую оценку достижимости локации с запросом.

- Отображение *reachableMethodsOfLoc* сопоставляет локации множество методов, в которых может оказаться символьное исполнение за 1 вызов функции. Данное отображение все еще достаточно точно оценивает достижимость локации с запросом.

- Отображение *reachableMethodsOfMethod* сопоставляет методу транзитивно-замкнутое множество методов, в которых он может принципе оказаться. Стоит заметить, что учитываются только методы программы, которую исполняем, поскольку методы из сторонних библиотек, например, MsCoreLib, точно не могут вызвать

метод программы. Данное отображение дает самую грубую (относительно предыдущих двух случаев) оценку достижимости локации с запросом.

Порядок действий в этой стратегии почти такой же, как и для *dfs-стратегии*: стратегия стартует исполнение из входной точки программы; если очередь распространения «назад» не пуста, то стратегия распространяет какой-то запрос «назад». Отличие заключается в способе выбора очередного состояния для исполнения: выбирается состояние с наименьшей оценкой достижимости локации с запросом.

4. Эксперименты

Целью экспериментов было определить эффективность предложенного алгоритма и стратегий символьного исполнения. В связи с этим было проведено сравнение классического и нового алгоритмов символьного исполнения, а также сравнение различных стратегий исследования кода для нового алгоритма направляемого свойством символьного исполнения.

Для достижения этой цели были поставлены следующие вопросы к экспериментальному исследованию.

RQ1:Насколько меньше инструкций исполняет алгоритм, используя стратегию, направленную к запросам? Этот вопрос направлен на выяснение того, сколько «непродуктивной» работы выполняет алгоритм, когда пользуется тривиальными стратегиями.

RQ2:Каковы преимущества нового алгоритма относительно классического? Другими словами, данный вопрос нацелен на прояснение того, насколько сильно уступают классические стратегии исследования (BFS, DFS) стратегии, направленная к запросам (TS).

4.1. Проектирование эксперимента

В качестве *тестовых наборов* для экспериментов использовались функциональные тесты системы V#, нацеленные на символьное исполнение различных конструкций языка C#, а также тесты из внешних библиотек LINQ, BlockChain, Regex.

Функциональные тесты V#. В подсистеме V# реализованы тесты, предназначенные для тестирования корректности символьной виртуальной машины. Они поддерживают различные конструкции языка C# такие, как создание объектов, выделение массивов, объявление делегатов и т.д.

Библиотека LINQ позволяет разработчикам для платформы .NET работать с различными коллекциями на уровне конструкций языка. Та-

ким образом, разработчику не нужно использовать отдельные виды запросов для различных форматов данных. Например, один и тот же код будет обрабатывать данные как из XML-файла, так и из SQL-таблицы.

Библиотека BlockChain предоставляет реализацию структуры данных, являющейся связным списком из блоков, где каждый блок связан не только нумерацией, но и помнит информацию о предыдущих блоках. Такая структура позволяет сохранять и передавать данные, которые не будут никак изменены.

Библиотека Regex служит для проверки соответствия строки шаблону. Например, при помощи регулярных выражений можно находить нужные файлы в файловой системе.

Для каждого функционального теста алгоритм пытался достичь всех локаций входного метода. Для дополнительных тестов с рекурсией и с циклами целевые локации кода задавались явно. Для библиотек LINQ, Blockchain, Regex использовался смешанный способ задания целевых локаций: алгоритм пытался посетить как все локации, так и множество целевых локаций, заданных явно.

В качестве метрик работы тестов были выбраны процент отвеченных запросов (Accuracy), а также количество исполненных инструкций.

Эксперимент проводился на следующем аппаратном и программном обеспечении. Аппаратное обеспечение состояло из ноутбука модели Dell XPS 9570. Его процессор Intel(R) Core(TM) i7-8750H с тактовой частотой 2.20 GHz обладал 12 логическими ядрами, а оперативная память была равна 16 GB. Программное обеспечение базировалось на операционной системе Ubuntu 20.04.2 LTS. Для запуска тестов была использована интегрированная среда разработки (IDE) JetBrains Rider версии 2020.2.4.

4.2. Результаты экспериментов

В таблице 1 представлены результаты проведенных экспериментов. Таблица содержит 4 главных тестовых набора: V#, BlockChain, Linq, Regex. Рядом с каждым набором содержится количество тестов. Для каждого набора вычисляется средняя точность ответов на запросы и суммарное количество исполненных инструкций. Рассмотрено выполнение предложенного алгоритма в рамках трех стратегий: BFS-стратегии (столбец BFS), DFS-стратегии (столбец DFS), а также направленная стратегия (столбец TS).

Тестовые наборы, кол-во	BFS	DFS	TS
V#, 436	0.90 26331	0.93 26331	0.90 7952
Blockchain, 4	1.0 30812	1.0 99839	1.0 10846
LINQ, 2	1.0 9129	1.0 7230	1.0 1370
Regex, 3	1.0 72437	1.0 71391	1.0 29471

Таблица 1: Таблица с результатами экспериментов

Ответим на поставленные вопросы.

RQ1:Насколько меньше инструкций исполняет алгоритм, используя стратегию, направленную к запросам? Как и ожидалось, для всех тестовых наборов направленная стратегия исполняет меньше инструкций. Однако это обстоятельство не мешает сохранить качество точности символьного анализа.

RQ2:Каковы преимущества нового алгоритма относительно классического? Предложенный алгоритм и направленная стратегия позволяют не терять в точности ответов на запросы, причем количество исполненных инструкций значительно уменьшается. Это связано с тем, что направленная стратегия направляет символьное исполнение к заданным локациям кода и не исследует лишних путей исполнения.

5. Заключение

В рамках данной выпускной квалификационной работы были получены результаты.

- Проведен обзор прямого, обратного, двунаправленного и композиционального символьного исполнения; а также подхода PDR для верификации программного обеспечения.
- Разработан алгоритм направляемого свойством символьного исполнения, который способен доказывать недостижимость путей исполнения и достигать заданных локаций программы.
- Алгоритм был реализован на платформе .NET, в рамках системы V# (символьной виртуальной машины для .NET) на языках C# и F#.
- Проведены эксперименты предложенного алгоритма на тестовых наборов системы V#, а также на сторонних библиотеках LINQ, Blockchain, Regexp; для больших тестовых программ алгоритм достигал заданной локации кода значительно быстрее, чем классический алгоритм символьного исполнения; при этом количество исполненных инструкций значительно уменьшилось при сохранении точности посещенных локаций кода.

Список литературы

[1] Anand Saswat, Godefroid Patrice, Tillmann Nikolai. Demand-driven compositional symbolic execution // International Conference on Tools and Algorithms for the Construction and Analysis of Systems / Springer. — 2008. — P. 367–381.

[2] Batoev K.A. VSharp. — `https://github.com/kbatoev/VSharp/tree/pdse`. — 2021.

[3] Boyer Robert S, Elspas Bernard, Levitt Karl N. SELECT—a formal system for testing and debugging programs by symbolic execution // ACM SigPlan Notices. — 1975. — Vol. 10, no. 6. — P. 234–245.

[4] Bradley Aaron R. SAT-based model checking without unrolling // International Workshop on Verification, Model Checking, and Abstract Interpretation / Springer. — 2011. — P. 70–87.

[5] Bradley Aaron R. Understanding ic3 // International Conference on Theory and Applications of Satisfiability Testing / Springer. — 2012. — P. 1–14.

[6] Cadar Cristian, Sen Koushik. Symbolic execution for software testing: three decades later // Communications of the ACM. — 2013. — Vol. 56, no. 2. — P. 82–90.

[7] Chandra Satish, Fink Stephen J, Sridharan Manu. Snugglebug: a powerful approach to weakest preconditions // Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation. — 2009. — P. 363–374.

[8] Craig William. Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory // The Journal of Symbolic Logic. — 1957. — Vol. 22, no. 3. — P. 269–285.

[9] Dershowitz Nachum, Hanna Ziyad, Nadel Alexander. A scalable algorithm for minimal unsatisfiable core extraction // International Confer-

ence on Theory and Applications of Satisfiability Testing / Springer. — 2006. — P. 36–41.

[10] Dinges Peter, Agha Gul. Targeted test input generation using symbolic-concrete backward execution // Proceedings of the 29th ACM/IEEE international conference on Automated software engineering. — 2014. — P. 31–36.

[11] King James C. Symbolic execution and program testing // Communications of the ACM. — 1976. — Vol. 19, no. 7. — P. 385–394.

[12] Komuravelli Anvesh, Gurfinkel Arie, Chaki Sagar. SMT-based model checking for recursive programs // Formal Methods in System Design. — 2016. — Vol. 48, no. 3. — P. 175–205.

[13] A survey of symbolic execution techniques / Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia et al. // ACM Computing Surveys (CSUR). — 2018. — Vol. 51, no. 3. — P. 1–39.