ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ УЧРЕЖДЕНИЕ
НАУКИ ИНСТИТУТ СИСТЕМНОГО ПРОГРАММИРОВАНИЯ
РОССИЙСКОЙ АКАДЕМИИ НАУК

На правах рукописи
УДК 519.686.4

Кошелев Владимир Константинович

МЕЖПРОЦЕДУРНЫЙ СТАТИЧЕСКИЙ АНАЛИЗ ДЛЯ ПОИСКА ОШИБОК В ИСХОДНОМ КОДЕ ПРОГРАММ НА ЯЗЫКЕ C#

Специальность 05.13.11 —
«Математическое и программное обеспечение вычислительных машин, комплексов и компьютерных сетей»

Диссертация на соискание учёной степени
кандидата физико-математических наук

Научный руководитель:
канд. физ.-мат. наук, ведущий научный сотрудник
Белеванцев Андрей Андреевич

Москва — 2017

Оглавление

Стр.

Стр.

Введение

Наряду с тестированием и динамическим анализом, статический анализ исходного кода широко используется для поиска дефектов в программах. Многие современные стандарты сертификации требуют, чтобы программное обеспечение было проверено на наличие дефектов, в том числе при помощи инструментов статического анализа. Например, таким стандартом является ГОСТ Р 56939-2016 для разработки безопасного программного обеспечения, который начал действовать в РФ 06.01.2017.

От статических анализаторов для применения в промышленном цикле разработки требуются: высокая скорость анализа (не более 2-3 часов для полного анализа проектов из нескольких миллионов строк кода); высокое качество анализа (высокий процент истинных срабатываний (50-70%) и поддержка поиска популярных классов ошибок, включающая поиск типичных ошибочных ситуации в рамках данных классов). Для соблюдения этих требований статические анализаторы вынужденно используют нестрогий анализ, позволяющий достичь компромисса между временем работы, количеством пропусков дефектов и процентом ложных срабатываний. При этом для поиска сложных межпроцедурных дефектов необходимо использовать анализ программ, учитывающий зависимости по данным (чувствительность к потоку), условия переходов (чувствительность к путям), контексты вызовов (чувствительность к контексту) и работу с динамической памятью.

В течение последних десяти лет язык программирования C# стабильно входит в шестерку самых распространенных языков программирования про версии TIOBE. В настоящий момент чувствительный к контексту, потоку и путям анализ с целью выявления дефектов для языка программирования C# осуществляют лишь закрытые коммерческие инструменты, для которых отсутствует детальное описание используемых подходов и алгоритмов. Существующие открытые решения для поиска дефектов ограничиваются обходом абстрактного синтаксического дерева или простым внутрипроцедурным анализом, не проводя детального анализа потоков управления и данных.

Методы проведения статического анализа программ разрабатываются как во многих научных центрах, таких как университет Стенфорд, университет Беркли, университет Британской Колумбии, ИСП РАН, так и в коммерческих

организациях, таких как Coverity, Klocwork, JetBrains. Общепризнанным способом организации масштабируемого межпроцедурного анализа является использование резюме функций, сжато описывающие их поведение. Конкретная форма и алгоритмы построения резюме целиком зависят от использующихся методов внутрипроцедурного анализа. Методов статического анализа, использующихся в открытых инструментах, на поверку оказывается недостаточно для удовлетворения указанных выше требований. В свою очередь, для коммерческих инструментов анализа отсутствует описание использованных методов.

Метод символьного выполнения, являясь чувствительным к потоку и путям, успешно применяется при анализе программ для поиска ошибочных входных данных, используя для этого решатели задачи выполнимости формул. В данной работе предлагается воспользоваться методом символьного выполнения для проведения внутрипроцедурного анализа. Метод символьного выполнения является ресурсоёмким, поэтому для достижения высокой скорости анализа в данной работе предлагается разработать:

- алгоритм объединения состояний символьного выполнения;
- алгоритм упрощения формул, основанный на свойствах доминаторов;
- подход для поддержки циклов с фиксированным числом итераций;
- метод моделирования вызовов с использованием резюме;
- подход для уменьшения числа запросов к решателю за счет доказательства свойств программы методами статического анализа.

Использование данных подходов позволяет достичь требуемой скорости при сохранении высокого качества анализа.

Методы статического анализа применяют различные критерии для выдачи предупреждений о наличии дефектов в зависимости от целей анализатора и использующихся алгоритмов анализа. Для успешного применения символьного выполнения для поиска дефектов необходимо разработать метод определения критерия выдачи предупреждений, учитывающий особенности символьного выполнения и позволяющий соблюсти баланс между числом пропусков ошибок и количеством ложных предупреждений.

Цели и задачи диссертационной работы: разработка внутрипроцедурных алгоритмов анализа и алгоритмов межпроцедурного анализа, чувствительных к потоку, контексту и путям, метода определения критерия выдачи предупреждений о наличии дефекта, обеспечивающих поиск дефектов в исходном коде программ, написанных на языке C#. Разработанные алгоритмы долж-

ны обеспечивать качество анализа, соответствующее современным требованиям для промышленных статических анализаторов, включая масштабируемость, достаточную для анализа программ, состоящих из миллионов строк кода.

Для достижения поставленных целей были сформулированы и решены следующие задачи:

1. Разработка внутрипроцедурных чувствительных к потоку и путям алгоритмов анализа, позволяющих на основе вычисленной ими информации выполнять поиск широкого класса дефектов; доказательство корректности алгоритмов.
2. Разработка межпроцедурных чувствительных к потоку, контексту и путям алгоритмов анализа, включающих алгоритм построения резюме метода по результатам внутрипроцедурного анализа и алгоритм применения резюме в точках вызова.
3. Разработка метода определения критерия выдачи предупреждения о наличии дефекта с учетом особенностей проводимого анализа, подходящего для широкого класса дефектов. Разработка конкретных детекторов для приведенного критерия выдачи предупреждения.
4. Оценка на практике характеристик предложенных методов и алгоритмов на соответствие заявленным требованиям.

Научная новизна. В работе были получены следующие результаты, обладающие научной новизной:

1. Разработаны алгоритмы внутрипроцедурного анализа, обладающие чувствительностью к потоку и путям, не теряющие информацию в точках объединения.
2. Разработаны алгоритмы межпроцедурного анализа, чувствительного к контексту, потоку и путям, обобщающие результаты внутрипроцедурного анализа в виде резюме метода, и применяющие резюме при обработке вызова данного метода.
3. Разработан метод определения критерия выдачи предупреждений, учитывающий чувствительность к путям при поиске дефектов, использование которого позволяет достичь приемлемого уровня ложных предупреждений.
4. Доказательства корректности предложенного алгоритма внутрипроцедурного анализа и соответствия разработанных детекторов выбранному критерию выдачи предупреждений.

Теоретическая и практическая значимость. Предложены алгоритмы чувствительного к путям внутрипроцедурного и межпроцедурного анализа для задачи поиска широкого класса дефектов. Разработан метод определения критерия выдачи предупреждений, учитывающий чувствительность к путям.

Все предложенные алгоритмы были реализованы в статическом анализаторе SharpChecker. Анализатор SharpChecker предполагает промышленное использование при разработке программного обеспечения для раннего обнаружения программных дефектов. Данный анализатор также может быть использован в сертифицирующих центрах для оценки качества сертифицируемого программного обеспечения. Разработанный анализатор SharpChecker внедрён для промышленного использования в компании Samsung.

Методология и методы исследований. Результаты диссертации были получены с использованием методов и моделей, применяемых при проведении статического анализа. Математическую основу данной работы составляют теория графов, теория множеств, математическая логика, теория алгоритмов.

Основные положения, выносимые на защиту:

1. алгоритм внутрипроцедурного анализа, обладающий чувствительностью к потоку и путям, при этом не теряющий информацию в точках объединения;
2. алгоритм межпроцедурного анализа, чувствительного к контексту, потоку и путям, обобщающий результаты внутрипроцедурного анализа в виде резюме метода и применяющий резюме при обработке вызова данного метода;
3. метод определения критерия выдачи предупреждений, учитывающий чувствительность к путям при поиске дефектов, использование которого позволяет достичь приемлемого уровня ложных предупреждений.

Апробация работы. Основные результаты работы докладывались на следующих конференциях:

1. XXI Международная научная конференция студентов, аспирантов и молодых учёных «Ломоносов-2014» (Москва, Россия 2014);
2. SPB .Net Meetup №10 (Санкт-Петербург, Россия, 2016);
3. Открытая конференция ИСП РАН 2016 (Москва, Россия, 2016);
4. Научно-исследовательский семинар Института системного программирования РАН.

Личный вклад. Все представленные в диссертации результаты получены лично автором.

Публикации. Основные результаты по теме диссертации изложены в 6 печатных изданиях [1–6], 5 из которых изданы в журналах, рекомендованных ВАК. Вклад автора в работе [2] заключается в разработке алгоритмов внутрипроцедурного и межпроцедурного анализа и разработке детектора для поиска доступа к нулевому указателю (`null`). В статье [3] вклад автора состоит в разработке алгоритмов оптимизации размера формул и реализации инфраструктуры инструмента SharpChecker. В работе [4] вклад автора состоит в разработке метода определения критерия выдачи предупреждений. В публикации [5] вклад автора заключается в разработке ядра анализа помеченных данных для программ на языках C/C++.

Статический анализатор языка C# SharpChecker, в котором реализованы предложенные методы анализа, включен в единый реестр российских программ для электронных вычислительных машин и баз данных по Приказу Минкомсвязи России от 09.03.2017 №103, Приложение 1, пп. №37, реестровый № 2910.

Объем и структура работы. Диссертация состоит из введения, четырёх глав, заключения и двух приложений. Полный объём диссертации составляет 104 страницы, включая 2 рисунка и 1 таблицу. Список литературы содержит 89 наименований.

Глава 1. Методы поиска ошибок в исходном коде

Статический анализ используется для обнаружения программных ошибок в исходном коде. Ключевая сложность разработки статического анализа заключается в поиске компромисса между масштабируемостью, классом обнаруживаемых ошибок, процентом ложных срабатываний и пропущенных ошибок и дополнительными ограничениями, накладываемыми на анализируемую программу. Идеальный статический анализатор должен хорошо масштабироваться, обнаруживать все ошибки времени выполнения, не иметь ложных срабатываний и пропусков ошибок и применяться ко всем программам без ограничений. Создание идеального анализатора невозможно в силу теоремы Райса.

1.1 Задачи статического анализа.

На практике часть из перечисленных выше свойств фиксируют, а предметом исследования становится достижение компромисса между оставшимися. Рассмотрим типичные постановки задачи для инструментов статического анализа:

- Задача доказательства корректности с пользовательскими аннотациями. Для каждой функции известны пред-, пост-условия и, возможно, инварианты циклов. Задача анализатора заключается в доказательстве соблюдения всех пред-,пост-условий и инвариантов цикла и отсутствия ошибок времени выполнения. Допускается наличие ложных срабатываний, гарантируется отсутствие пропуска ошибок. Сложность разработки заключается в поиске компромисса между временем работы и минимизацией класса корректных программ, для которых верификация не удалась. Примеры инструментов: frama-c/Jessie, Clousot.
- Задача доказательства корректности для подмножества языка. Анализу подвергаются специализированные программы, не использующие некоторых возможностей языка. Задача анализатора заключается в доказательстве отсутствия ошибок времени выполнения. Допускается наличие ложных срабатываний, гарантируется отсутствие пропуска ошибок.

Сложность разработки заключается в поиске компромисса между временем работы и процентом ложных срабатываний. Пример инструмента: Astree.

– Задача доказательства корректности без ограничений. Никаких дополнительных ограничений на анализируемые программы не накладывается. Задача анализа заключается в доказательстве отсутствия некоторых классов ошибок времени выполнения. Гарантируется отсутствие как ложных срабатываний, так и пропусков ошибок, однако анализ может зацикливаться. Сложность разработки заключается в минимизации класса программ, на который анализ зацикливается. Примеры инструментов: Blast, CPACheck.
– Задача поиска дефектов. Никаких дополнительных ограничений на анализируемые программы не накладывается. Задача анализа — поиск потенциальных ошибок времени выполнения. Допускается присутствие ложных срабатываний и пропусков ошибок. Сложность разработки заключается в обнаружении практически значимого класса ошибок при сохранении низкого процента ложных срабатываний и хорошей масштабируемости. Примеры инструментов: Svace, Klocwork, Coverity.

Данная работа прежде всего посвящена задаче поиска дефектов. Однако, несмотря на разницу в постановках задач поиска дефектов и доказательства корректности, инструменты, решающие эти задачи, основаны на сходном наборе методов. Следовательно, для разработки инструмента статического анализа необходимо понимание использующихся на практике методов.

1.2 Дефекты в исходном коде

Понятие дефекта в статическом анализе отличается от понятия ошибки времени выполнения. Для выполняемого файла программы возникновение ошибки времени выполнения зависит в том числе от операционной системы, окружения, компилятора и т.д. Статический анализатор абстрагируется от данных зависимостей, проводя поиск дефектов в соответствии с семантикой анализируемого языка программирования. Примерами дефектов, которым подвер-

жены программы на большинстве популярных языках программирования, являются разыменования нулевого указателя и утечка ресурсов.

Разыменованием нулевого указателя будем называть ситуацию, при которой происходит обращение по указателю, имеющему специальное значение (`null`), означающее, что обращение по данному значению недопустимо. В зависимости от языка программирования может привести как к падению программы (C/C++), так и к возникновению исключительной ситуации (Java/C#).

Утечка ресурсов происходит, если выделенный ресурс не освобождается после окончания использования. Одиночная утечка ресурсов не приводит к падению программы, однако в случае исчерпания ресурсов, например, лимита на число открытых файлов или сетевых соединений, программе будет отказано в выделении очередного ресурса, что может привести к её аварийному завершению. Дефекты, обнажаемые статическим анализатором, не обязательно связаны с ошибками выполнения. Статический анализатор может обнаружить ошибку в неиспользующемся коде. Не использоваться код может не только потому, что он больше не нужен, но и потому, что использование данного кода ещё не было реализовано программистом.

Кроме того, статический анализатор может обнаружить не саму ошибочную ситуацию, а избыточность в исходном коде программы, говорящую о возможности ошибки. Примером такой избыточности является сравнение указателя с `null` с последующем его разыменованием. Данная ситуация является потенциально ошибочной, потому что либо указатель никогда не равен `null`, и тогда сравнение избыточно, либо далее может произойти обращение по нулевому указателю.

Поиск подобных ошибочных шаблонов позволяет обнаружить подозрительные ситуации в исходном коде программы. Существование корреляции между ошибками и избыточными вычислениями в программе было показано в работах [7; 8].

1.3 Методы статического анализа

Одним из самых известных инструментов статического является утилита lint [9] операционной системы Unix. Lint проводит анализ использования подо-

зрительных и непереносимых конструкций в программах на языке C. Инструмент lint был включён в седьмую версию операционной системы Unix в 1979 году и по сей день остаётся в составе операционной системы FreeBSD. К числу обнаруживаемых инструментом lint дефектов относятся:

- использования неинициализированных переменных;
- арифметическое переполнение;
- недостижимый код;
- пропущенный оператор “return”.

Поиск дефектов в инструменте lint организован таким образом, что каждый дефект, найденный им, зачастую нуждается в исправлении. Благодаря этому свойству дефекты, представленные в анализаторе lint, сейчас выдаются современными компиляторами в виде ошибок и предупреждений. Трактовка дефекта как ошибки компиляции возможна только в том случае, когда дефект одновременно не требует ресурсоёмкого анализа для обнаружения, и его ошибочность очевидна программисту.

Благодаря своей популярности, название анализатора “lint” стало нарицательным, обозначая инструмент для поиска подозрительных конструкций, особенно для скриптовых языков программирования. Такие инструменты могут быть успешно использованы для поиска ошибок, допущенных программистом по невнимательности. Однако ввиду локальности проводимого анализа данные инструменты не подходят для обнаружения сложных ошибок, требующих анализа потоков данных и управления программы.

Примером сложной ошибки является межпроцедурное обращение к нулевому указателю. Под межпроцедурностью в данном случае понимается то, что присваивание указателю значения `null` и обращение к нему происходит в разных функциях. Более того, ошибка может произойти из-за того, что функции, в которых происходят присваивание и обращение, были некорректным образом вызваны третьей функцией. Далее в данной работе будут рассматриваться только анализы, способные обеспечить обнаружение межпроцедурных дефектов, учитывая зависимости по данным и управлению.

1.3.1 Задача точного поиска дефектов

Задача точного поиска дефектов в соответствии с семантикой исходной программы, наличие которых потенциально может служить причиной аварийного завершения, алгоритмически неразрешима по теореме Райса. На практике задача точного поиска дефектов формулируется с использованием дополнительных упрощений. Некоторыми из классических вариантов упрощения являются игнорирование вызываемых функций (внутрипроцедурный анализ), условий переходов и порядка операторов программы. Если же в алгоритме частично или полностью имеется соответствующая поддержка, то говорят о том, что он обладает соответствующей чувствительностью:

- чувствительность к контексту, алгоритм анализа является межпроцедурным, учитывающим при моделировании вызванной функции состояние программы в точке её вызова;
- чувствительность к путям, алгоритм анализа учитывает в результатах своей работы предикаты условных переходов;
- чувствительность к потоку, алгоритм анализа учитывает порядок операторов языка программирования.

Задача точного поиска дефектов, связанных с обращениями к памяти, неразрешима даже в случае, если от анализа требуется только чувствительность к потоку. Данное утверждение следует из того, что задача чувствительного к потоку точного анализа псевдонимов алгоритмически неразрешима [10; 11].

Замечание о алгоритмической неразрешимости чувствительного к потоку поиска дефектов, связанных с обращениями к памяти, существенно. Например, точный анализ инициализированных локальных переменных алгоритмически неразрешим, а аналогичный чувствительный лишь к потоку анализ реализован практически во всех современных компиляторах. Например, компилятор для языка программирования C# выдаст ошибку компиляции, если локальная переменная не была явно инициализирована перед использованием на всех путях выполнения, даже если она не инициализирована только на невозможных путях.

Ввиду данной неразрешимости инструментам статического анализа приходится вводить дополнительные оптимистичные предположения. Далее рассмотрим различные подходы к анализу программ, включающие себя как алгоритмы

анализа, так и упрощающие предположения, при которых данные алгоритмы работают, а также задачи, которые они призваны решать.

Представим результат работы алгоритма статического анализа в виде доказательства импликации $P \to Err$, где P — анализируемая программа, а Err — факт наличия ошибки. Так как осуществить доказательство исходной импликации не представляется возможным, вместо неё рассмотрим программу $P' : (P' \to Err) \to (P \to Err)$. Для точного алгоритма поиска дефектов в P' верно, что все найденные им дефекты также являются дефектами и для P, однако он мог пропустить часть дефектов в P. Аналогично рассмотрим программу $P'' : (P \to Err) \to (P'' \to Err)$. Алгоритм точного анализа для программы P'' найдет также все дефекты в программе P, однако часть найденных им дефектов не будет являться дефектами с точки зрения P, иными словами они будут ложными срабатываниями. Процедуру получения программы P' по программе P будем называть аппроксимацией снизу, а программы P'' по P — аппроксимацией сверху.

1.3.2 Абстрактная интерператция

Для построения корректных аппроксимаций семантики может быть использована теория абстрактной интерпретации [12; 13]. Абстрактная интерпретация была предложена французскими математиками Патриком Кузо и Радией Кузо в конце 70-х годов XX века. Абстрактная интерпретация описывает теорию корректной аппроксимации программ, основанную на монотонных функциях над элементами решёток [14]. Данная теория позволяет заменить обычную семантику программ, именуемую конкретной семантикой, на абстрактную семантику, оперирующую множествами состояний программ.

Идея применения абстрактной интерпретации заключается в аппроксимации сверху множества возможных состояний программы. Если аппроксимированное множество возможных состояний не содержит ошибочных состояний, тогда анализируемая программа гарантированно не содержит дефектов. Однако наличие ошибочных состояний ещё не говорит об ошибочности исходной программы. Выдача предупреждений в случае, если аппроксимированное мно-

жество состояний содержит ошибочное состояние, неизбежно приведёт к появлению ложных срабатываний.

Пожалуй, самым известным инструментом статического анализа, использующим абстрактную интерпретацию, является инструмент Astree [15–17], в разработке которого приняли активное участие Патрик и Радия Кузо. Данный инструмент нацелен на доказательство отсутствия ошибок времени выполнения в критических системах реального времени написанных на языке C. Например, при помощи данного инструмента было доказано отсутствие ошибок в основной системе управления полётом самолёта AirBus 340.

Авторы инструмента Astree утверждают, что при анализе целевого программного обеспечения Astree показывает крайне малое число ложных срабатываний, обладая при этом достаточной масштабируемостью для анализа существующих систем реального времени. Анализ программ, состоящих из нескольких сот тысяч строк кода, занимает несколько часов. Однако для достижения этих результатов, авторами Astree на анализируемые программы были наложены принципиальные ограничения, а именно отсутствие:

- рекурсивных функций;
- динамического выделения памяти;
- обратных операторов goto;
- функций setjmp/longjump;
- параллельности;
- сложных структур данных;
- более двух вложенных циклов.

Так как в подавляющем большинстве программ на языке C присутствует динамическое выделение памяти, подход инструмента Astree в явном виде не применим для их анализа.

1.3.3 Метод CEGAR

Метод CEGAR(Counter Example Guided Abstraction-Refinement) [18] также использует аппроксимацию сверху для доказательства того, что программа не содержит дефектов. Однако в отличие от абстрактной интерпретации, метод CEGAR позволяет на лету подстраивать использующиеся абстракции

под конкретное правило корректности и особенности анализируемой программы. Данный метод имеет ограниченную применимость, позволяя анализировать программы лишь до 50-100 тыс. строк кода для языка C. Также время работы существенно зависит от типа проверяемых дефектов. Методы уточнения абстракции организованы таким образом, что при неудачной комбинации условий в программе они не завершаются.

Данный метод реализован в инструментах BLAST [19; 20], CPAchecker [21;22], SLAM [23–25]. Эти инструменты применяются преимущественно для верификации драйверов операционных систем, поскольку размер одного драйвера обычно не превосходит нескольких тысяч строк кода. Основная часть ядра операционной системы не анализируется, вместо этого используется заранее определённая модель поведения, для создания и поддержания которой в актуальном состоянии нужно приложить значительные усилия. Тем не менее, данный метод показал свою применимость при разработке драйверов операционных систем. В рамках проекта Linux Driver Verification [26–30], использующего анализаторы BLAST и CPAChecker, регулярно находятся подтверждённые разработчиками ошибки в драйверах ядра Linux. В свою очередь, компания Microsoft включила проект SLAM в набор для разработчиков драйверов для операционных систем Windows и в Visual Studio.

Однако метод CEGAR, в силу ограничений на размер анализируемого кода, не может быть применён для анализа промышленных программ.

1.3.4 Сведение к выполнимости формул

Подходами, использующими при анализе программ аппроксимацию снизу, являются ограничиваемая проверка моделей (Bounded Model Checking) [31;32] и символьное выполнение (Symbolic Execution) [33]. Оба подхода рассматривают лишь конечное множество путей выполнения, вводя ограничения на рассматриваемые пути. Примерами таких ограничений, обеспечивающих конечность множества путей, являются ограничения на максимальное число пройдённых обратных рёбер циклов и максимальную глубину вызовов. Для того чтобы анализы, построенные на данных подходах, не выдавали ложных срабатываний, необходимо, чтобы, во-первых, у анализируемых программ была единственная точка

входа, во-вторых, для используемых в программе внешних функций должны быть предоставлены спецификации, моделирующие их поведение. При соблюдении этих двух свойств для программ на языке C гарантируется отсутствие ложных срабатываний проводимого анализа.

Для проведения анализа в ограничиваемой проверке моделей проводится развертку циклов и вставку функций, которая превращает программу в ациклический граф, соответствующий анализируемым путям выполнения. Далее производится трансляция программы, представленной в виде ациклического графа, в набор уравнений над битовыми векторами. При данной трансляции обращения к памяти моделируются при помощи неинтерпретируемой функции, а в условиях равенства допускается условный оператор, использующий ранее определённые переменные. Таким образом, осуществляется трансляция программы в систему уравнений, которая может быть проверена на совместность с условием возникновения ошибки с помощью SAT/SMT-решателей [34]. Данным подход был реализован в инструментах CBMC, LLBMC [35].

Символьное выполнение также использует системы уравнений над битовыми векторами для выражения семантики программы. Основное отличие методов, основанных на символьном выполнении, от ограничиваемой проверки моделей заключается в представлении в явном виде информации, описывающей возможные состояния программы. Примером инструмента, использующего символьное выполнение, является инструмент KLEE [36]. Другим примером использования символьного исполнения для поиска дефектов является Clang static analyzer (CSA) [37].

К сожалению, такие анализаторы имеют ограниченную масштабируемость в силу использования встраивания при обработке вызовов. На практике многие современные промышленные и исследовательские анализаторы, ставящие перед собой задачу поиска максимального числа дефектов, используют нестрогий анализ. Алгоритм статического анализа программ является нестрогим, если он допускает как пропуск ошибок, так и наличие ложных срабатываний.

1.3.5 Нестрогий анализ

В работе [16] авторы инструмента Astree рассматривают основные причины, по которым инструменты статического анализа являются нестрогими. К данным причинам относятся:

– Преднамеренное игнорирование семантики операторов. Статический анализатор может игнорировать частично или полностью семантику сложных операторов языка программирования, например, оператор вызова функции по указателю.
– Поиск ошибок в частных случаях. Статический анализатор может рассматривать частный случай ошибочных ситуаций, например, только внутрипроцедурный случай.
– Дополнительная фильтрация найденных предупреждений. Статический анализатор может подвергать найденные предупреждения дополнительной фильтрации. Например, он может не выдавать предупреждения в случае подавления данных предупреждений пользователем.
– Пропуск путей выполнения. Статический анализатор может игнорировать часть путей выполнения. Например, такое поведение может быть связано с анализом лишь нескольких первых итераций цикла.

Современные статические анализаторы для поиска дефектов, как правило, обладают всеми перечисленными выше причинами нестрогости анализа. Общая идея реализации такого рода анализаторов заключается в комбинации одного из приведённых выше вариантов внутрипроцедурного анализа с межпроцедурным анализом на основе резюме. Анализ на основе резюме заключается в построении для каждой анализируемой функции краткого описания её поведения — резюме. Преимуществом анализа на основе резюме является возможность переиспользования результатов анализа функции при обработке её вызовов.

Первым промышленным инструментом для поиска дефектов, использовавшим комбинацию символьного выполнения и резюме, был инструмент Prefix [38] для анализа программ на C/C++. Разработчиками данного инструмента были сформулированы требования к промышленным инструментам поиска дефектов, а именно:

– анализ должен быть эффективен при проверке промышленных программ на C/C++;

– анализ должен основываться на исходном коде, а не на аннотациях пользователя;
– анализ должен учитывать условия переходов, не ограничиваясь анализом потоков данных и управления;
– сообщение об ошибке должно содержать подробное описание причины её возникновения.

Для достижения заявленных требований в инструменте Prefix были применены следующие подходы:

– внутрипроцедурное символьное выполнение (авторы используют термин "симуляция") каждой функции;
– анализ функций в обратном топологическом порядке графа вызовов с использованием результатов символьного выполнения для построения резюме (авторы используют термин "модель");
– моделирование вызовов функций при помощи построенных резюме;
– использование специализированных решателей для проверки математических формул, описывающих возможность возникновения ошибочных ситуаций.

В дальнейшем данные подходы использовались во многих инструментах, ориентированных на поиск программных дефектов.

Первым нестрогим инструментом статического анализа, пользующимся аннотациями, был ESC/Modula-3 [39]. В дальнейшем подход, использующийся в ESC/Modula-3, был доработан для языка Java в инструменте ESC/Java [40]. В отличие от инструмента Prefix, для межпроцедурного анализа использовались пред- и пост-условия, написанные разработчиками, а не исходный код функции. В данных инструментах впервые использовался полноценный решатель для проверки выполнимости полученных в ходе анализа формул.

1.3.6 Инструмент Saturn

Следующим значимым инструментом для поиска дефектов является анализатор Saturn [41–45]. Его авторы заявляют основной целью создание масштабируемого средства поиска дефектов, работающего без участия программиста. Внутрипроцедурный анализ в инструменте Saturn основан на точной трансля-

ции операторов исходной программы в SAT-формулу. В отличие от остальных существовавших на тот момент инструментов, основанных на SAT-формулах, Saturn достиг масштабируемости на проектах размером в несколько миллионов строк кода. Одним из ключевых отличий, позволивших достичь таких результатов, было использование резюме для проведения межпроцедурного анализа. Использование резюме вместо встраивания функций позволяет избежать экспоненциального роста сложности анализа вызова.

В инструменте Saturn рассматриваются два базовых типа: булевый и знаковый или беззнаковый целочисленный тип заданной битовой размерности N. Целочисленные типы из N бит моделируются с помощью вектора булевых переменных размерности N, а операции над типами моделируются при помощи соответствующих булевых схем.

Для моделирования памяти при проведении внутрипроцедурного анализа авторы Saturn вводят понятие множеств условных ячеек памяти (Guarded Location Sets). Для каждого указателя в каждой конкретной точке программы множество условных ячеек памяти задает набор ячеек, на который данный указатель может указывать. Каждое отношение «указывает на» в данных множествах сопровождается формулой G, указывающей условие, при котором отношение достигается. В точках слияния потоков управления возможные значения переменных выражаются через значения, пришедшие по данным потокам, учитывая условия для каждого входящего потока. Таким образом, в точках слияния не происходит потери точности анализа.

В работе [44], авторы приводят причины, по которым инструмент Saturn является нестрогим, а именно:

– Поддержка циклов. Авторы используют два подхода для анализа циклов: развертку на определённое число итераций и замену правых частей операторов присваивания на неизвестные значений (loop havocing). Первый подход ведет к пропуску ошибок, а второй как к пропуску, так и к появлению ложных срабатываний.
– Поддержка рекурсии. Функции из компонент сильной связности в графе вызовов анализируются в произвольном порядке. Данный подход ведет как к пропуску ошибок, так и к появлению ложных срабатываний. Однако на практике рекурсия используется относительно редко, поэтому влияние отсутствия такой поддержки невелико.

– Межпроцедурные псевдонимы. Анализ проходит в предположении, что любые два неизвестных значения указателей не являются псевдонимами.
– Неполнота резюме. Резюме не содержит полного описания поведения функции, что может приводить как к пропускам ошибок, так и к ложным срабатываниям.
– Неполная поддержка конструкций языка C. В инструменте не поддерживаются некоторые конструкции языка C, например, отсутствует поддержка чисел с плавающей точкой.

Идеи, использованные в Saturn, были в дальнейшем доработаны в инструменте Calysto [46; 47]. В отличие от Saturn, Calysto использует SMT-решатель и улучшенную модель резюме. В нём резюме представляется в виде графа символьных значений, схожем по структуре с графом значений. Данный граф лениво встраивается в код анализируемой функции в случае, если содержит вычисление условий, влияющих на возникновение ошибки. Автор утверждает, что Calysto имеет большую точность и лучшее время работы по сравнению с анализатором Saturn. Однако в отличие от Saturn, Calysto недоступен для использования.

Одним из немногих современных промышленных статических анализаторов для поиска дефектов, имеющих подробное описание принципов работы, является анализатор VARVEL [48; 49], разработанный в NEC. Анализатор Varvel состоит из трех основных компонент: абстрактного интерпретатора, BMC-анализатора и инфраструктуры создания резюме функций. Анализ начинается с применения абстрактной интерпретации для доказательства отсутствия дефектов. После этого при помощи слайсинга из программы исключаются инструкции, не влияющие на недоказанные потенциально уязвимые инструкции. По словам авторов, при помощи такой комбинации удается сократить объем анализируемого кода на 60%. Далее, подобно Saturn, в Varvel рассматриваются все функции как возможные точки входа с произвольными входными параметрами. Для каждой такой функции запускается BMC-анализ с ограничением на глубину вызовов. Все функции, имеющие глубину вызова относительно данной функции больше заданной величины, заменяются на свои резюме, содержащие предусловия и постусловия их выполнения. Авторы утверждают, что данный подход может быть эффективно применён для анализа проектов, состоящих из нескольких десятков миллионов строк кода.

1.3.7 Промышленные статические анализаторы

В настоящее время на рынке представлены несколько промышленных статических анализаторов, а именно Coverity [50; 51], Klocwork [52], GrammaTech CodeSonar [53], MathWorks Polyspace [54] и другие. К сожалению, для данных инструментов отсутствует детальное описание принципов их работы.

Инструмент Coverity на данный момент является одним из лидеров рынка статических анализаторов. Он осуществляет поиск таких дефектов, как разыменование нулевого указателя, утечки памяти и ресурсов, некорректная работа с освобожденной памятью, переполнения буфера и другие. Coverity доступен для бесплатного использования при анализе проектов с открытым исходным кодом. Результаты анализа доступны для третьих лиц с одобрения авторов проекта. Однако по лицензионному соглашению сервиса результаты работы анализатора не могут разглашаться третьим лицам и не могут использоваться для сравнения.

Тем не менее, по косвенным данным можно утверждать, что идейно подход, используемый в анализаторе Coverity, схож с подходами инструментов Saturn и VARVEL. В нем используется один из вариантов чувствительного к путям внутрипроцедурного анализа, предположительно, символьное выполнение вместе с резюме для моделирования вызовов функций.

Согласно данным с официального сайта, анализатор Klocwork осуществляет поиск схожего множества дефектов с анализатором Coverity. К сожалению, последние результаты сравнения данных анализаторов, которые удалось обнаружить, датированы 2008 годом [55]. Очевидно, что в настоящий момент результаты сравнения не являются релевантными. Однако примечательным является то, что результаты этих инструментов пересекались не более, чем на 20 процентов. Особенности реализации алгоритмов анализа в инструменте Klocwork также неизвестны.

В 2015 году Toyota ITC разработала тестовый набор для статических анализаторов языков C/C++ [56]. Данный тестовый набор содержит 1276 тестов, подавляющее большинство которых написано на языке C. Тесты разделены на категории в зависимости от классов ошибок. Например, в тестах присутствуют такие категории, как выход за границы массива для статической и динамической памяти, ошибки работы с ресурсами, ошибки при работе с `null` и осво-

бождённой памятью. Каждый тест содержит либо одну ошибку, либо схож с соответствующим ошибочным, но ошибки не содержит.

Разработчики тестового набора провели тестирования промышленных анализаторов GrammaTech CodeSonar и MathWorks Polyspace, а также собственного анализатора RV-Match. Методика их тестирования показала, что анализатор GrammaTech незначительно превосходит MathWorks Polyspace. Анализатор RV-Match показал наилучший результат на данном тесте.

Тем не менее, сама методика тестирования вызывает несколько нареканий. Авторы анализатора утверждают, что тесты были составлены, исходя из реальных примеров ошибочных ситуаций, однако детальное обоснование для каждой группы тестов, к сожалению, отсутствует. Что более важно, размер тестов в большинстве случаев не превосходит пары десятков строчек, что делает допустимым использование практически любых методов анализа, независимо от их масштабируемости.

Ввиду этих недостатков в данной работе подобные тестовые наборы не будут рассматриваться в качестве релевантного способа оценки качества работы статического анализатора. Предпочтение будет отдано в пользу анализа набора промышленных проектов и оценки результата работы на них.

1.3.8 Существующие анализаторы программ на языке C#

Прежде всего отметим, что такие промышленные статические анализаторы, как Coverity и Klocwork, поддерживают C#. Для обоих анализаторов известен список предупреждений, которые они способны обнаружить в языке C#. Также для анализатора Coverity известны его результаты на ряде проектов с открытым исходным кодом, однако, как говорилось ранее, данные результаты не могут быть приведены в данной работе.

Для языка C# существует ряд коммерческих решений, нацеленных в том числе на поиск нарушений установившихся практик (best practices). К данным инструментам относятся CodeIt.Right [57], FxCop [58], StyleCop [59], NDepend [60] и CodeRush [61]. В отличие от Coverity и Klocwork, данные инструменты не осуществляют поиск сложных ошибок, поэтому далее в данной работе рассмотрены не будут.

Статический анализатор Clousot [62] включён в набор инструментов CodeContracts, предназначенный для проверки соблюдения контрактов методов и инвариантов классов. Данный инструмент использует нестрогий анализ, предполагая отсутствие псевдонимов и моделируя вызовы функции при помощи их контрактов, игнорируя реализацию. Для проведения внутрипроцедурного анализа используется абстрактная интерпретация. Данный инструмент предназначен прежде всего дня проверки выполнения контрактов и не может успешно применяться к программам, в которых отсутствуют контракты. Так как явное задание контрактов далеко не всегда используется при промышленной разработке на C#, данный инструмент имеет ограниченное применение.

В 2014 году компания Microsoft выпустила открытую компиляторную платформу Roslyn [63]. Данная платформа поддерживает возможность интеграции в компилятор статических анализаторов, основанных на анализе абстрактного синтаксического дерева. На данный момент существует достаточно большое число проектов с открытым исходным кодом, реализующих различные наборы анализаторов, основанные на Roslyn. Данные анализаторы осуществляют поиск относительно простых ошибочных ситуаций, не требующих проведения анализа потоков управления и данных. Примером хорошего набора таких анализаторов является проект SonarLint.

На компиляторной платформе Roslyn основан российский коммерческий анализатор PVS-Studio [64]. Данный анализатор также представляет собой коллекцию анализаторов для Roslyn. На момент написания данной работы PVS-Studio поддерживает 126 различных типов дефектов, которые, по словам авторов, реализованы в виде анализаторов абстрактного синтаксического дерева. Необходимо отметить, что несмотря на то, что в PVS-Studio многие ошибочные паттерны выбраны удачно, проводимого анализа недостаточно для обнаружения сложных ошибок уровня инструментов Coverity и Klocwork.

Пожалуй, самым популярным инструментом упрощения разработки программ на языке C#, включающим в себя статический анализатор, является ReSharper от российской компании JetBrains [65]. Статический анализатор ReSharper проводит не только анализ абстрактного синтаксического дерева, но и внутрипроцедурный анализ потоков данных, что, в частности, позволяет ему обнаруживать внутрипроцедурные ошибки на уровне анализаторов Coverity и Klocwork. ReSharper проводит внутрипроцедурный анализ инкрементально, что позволяет отображать его результаты во время разработки в IDE. Разумеется,

наличие требования о достаточно быстром для интеграции в IDE анализе накладывает ограничение на его сложность. В результате для достижения хорошей производительности ReSharper не использует при межпроцедурном анализе информацию о коде метода, опираясь лишь на пользовательские аннотации в случае их наличия.

Таким образом, можно сделать вывод, что на данный момент для языка C# существуют лишь несколько закрытых коммерческих инструментов, осуществляющих межпроцедурный статический анализ для поиска дефектов. Следовательно, задача разработки масштабируемых алгоритмов поиска сложных дефектов, аналогичных схожим дефектам в программах на C/C++, является актуальной.

1.3.9 Инструмент статического анализа Svace

В Институте системного программирования РАН разработан программный продукт Svace [66–80], осуществляющий анализ программ на языках C/C++, Java и C#. Svace включает в себя несколько независимых анализаторов, а именно:

- одноименный анализатор Svace для программ на C/C++ и Java;
- набор анализаторов абстрактного синтаксического дерева на основе компилятора Clang;
- набор анализаторов на основе компилятора javac для языка Java;
- анализатор SharpChecker, основанный на компиляторной инфраструктуре Roslyn.

В диссертации [81] описывается устройство анализатора Svace для программ на C/C++. Подход, используемый в анализаторе Svace, показал свою эффективность при анализе больших проектов. Общее устройство анализатора идеологически схоже со схемой, использующейся инструментами Prefix, Saturn и Varvel. Отметим общие для данных анализаторов особенности:

- В анализаторе Svace используются резюме для организации межпроцедурного анализа.

– Для анализа циклов в анализаторе Svace применяется схожий с анализатором Saturn прием: развертка цикла в комбинации с заменой правых частей некоторых операторов присваивания на неизвестные значения.
– Предположение об отсутствии указателей-псевдонимов между различными неизвестными значениями, в частности, отсутствие указателей-псевдонимов среди аргументов функции.

Алгоритмы анализа, предлагаемые в данной работе, в целом следуют общему подходу анализатора Svace, имея перечисленные выше особенности. Однако внутреннее устройство анализатора SharpChecker существенным образом отличается от внутреннего устройства анализатора Svace.

В качестве основного критерия выдачи предупреждений автор диссертации [74] рассматривает критерий наличия дефекта на основе критических ребер. Данный критерий говорит, что в программе содержится дефект определённого класса, если он неизбежен на всех путях, проходящих через некоторое ребро в графе потока управления. Как отмечается в диссертации, данного критерия недостаточно для обнаружения многих существенных предупреждений, отброшенных ввиду отсутствия чувствительности к путям. Однако автор не предлагает конкретного метода реализации чувствительности к путям, оставляя её на откуп авторам детекторов дефектов. В отличие от алгоритмов анализатора Svace, алгоритмы анализатора SharpChecker изначально разрабатывались с целью поддерживать чувствительность к путям на всех этапах анализа.

Критерий обнаружения дефектов, использованный в анализаторе Svace, показал свою эффективность на практике. Однако данный критерий не может быть использован для обнаружения дефектов с учетом чувствительности к путям. В данной работе предлагается метод построения критериев, одновременно обобщающий критерий анализатора Svace и позволяющий обнаруживать дефекты с учетом чувствительности к путям, сохраняя при этом приемлемый процент ложных препупреждений.

В анализаторе Svace большое значение играют алгоритмы нумерации значений, использующиеся для поисков эквивалентных выражений. Ввиду полной чувствительности к путям, анализатор SharpChecker не производит нумерацию значений, поскольку данная задача решается на стороне решателя формул.

Отдельное внимание в инструменте SharpChecker отводится алгоритмам, позволяющим минимизировать размер формул. В данной работе рассматрива-

ются две оптимизации, основанные на свойствах дерева доминаторов, позволяющие существенно уменьшить размер формул.

Представленные в данной работе алгоритмы реализованы в анализаторе SharpChecker.

Глава 2. Внутреннее представление для языка C#

В данной главе рассматриваются особенности анализа языка C#, а также вводится внутреннее представление для программ на языке C#.

2.1 Особенности анализа языка C#

Язык программирования C# — объектно-ориентированный статически типизированный язык высокого уровня. Компания Microsoft активно его развивает, выпуская новую версию каждые два-три года. С каждой новой версией, в язык C# добавляются новые языковые возможности, что требует от инструментов статического анализа адаптироваться к ним. Данная проблема может быть решена с помощью проведения анализа над ассемблером виртуальной машины .NET Common Intermediate anguage (сокращённо CIL). В отличие от языка C#, спецификация CIL практически не меняется при переходе к новой версии языка. Однако работа с представлением, основанным на исходном коде, даёт возможность специфицировать поведение анализа для языковых конструкций, не представленных в явном виде в CIL. Восстановление таких конструкций в CIL потребует проведения дополнительных анализов.

С выходом компиляторной инфраструктуры Rolsyn задача поддержки новых языковых возможностей для основанных на нем анализаторов значительно упростилась. Так как Roslyn разрабатывается компанией Microsoft, все новые возможности языка C# реализуются именно в данном компиляторе. Следовательно, разработчики статического анализатора автоматически получают абстрактное синтаксическое дерево, содержащее новые конструкции.

Учитывая этот факт, при разработке статического анализатора SharpChecker было принято решение использовать компиляторную инфраструктуру Roslyn для получения набора абстрактных синтаксических деревьев и семантической информации для анализируемой программы. Дальнейшие структуры данных, такие, как граф потока управления [82; 83] (сокращённо ГПУ) или граф вызовов, строятся на основе полученной от Roslyn информации.

Постоянное добавление новых возможностей приводит к тому, что в языке присутствует достаточно больше количество неявных конструкций, точная поддержка многих из которых невозможна при статическом анализе. Вопрос о поддержки возможностей решался, исходя из их важности для поиска доступа к `null` и утечки ресурсов и возможности осуществить их полноценную поддержку методами статического анализа.

В данной работе поддерживаются следующие возможности. Подробнее поддержка данных возможностей обсуждается в последующих главах.

Поддержка динамический памяти. В языке C# отсутствует адресная арифметика, что позволяет существенно упростить модель памяти по сравнению с языками C/C++. При анализе, в частности, не учитываются записи в глобальные статические поля классов, во-первых, такие поля зачастую не используются на практике, и, во-вторых, запись в статические поля может происходить из разных потоков, что не учитывается при анализе.

Динамическая проверка типа. Во время выполнения для любого объекта языка C# может быть произведена проверка его фактического типа. Так как результат этой проверки зачастую влияет на поток выполнения программы, то инструмент статического анализа должен учитывать его при вычислении условий переходов.

Поддержка исключений. В языке C# для обработки нештатных ситуаций используется механизм исключений. Наличие в языке исключений приводит к необходимости поддержки множества точек выхода из функции, что приводит как к необходимости учета дополнительных зависимостей по управлению при построении ГПУ, так и к существенному усложнению межпроцедурного анализа.

Поддержка конструкций управления ресурсами. Для управления ресурсами в языке C# используется концепция Disposable-объектов. Считается, что объекты, реализующие управление ресурсами, должны реализовывать интерфейс IDisposable, содержащий единственный метод Dispose — освобождение ресурса. Для безопасного управления жизненным циклом таких объектов в язык введен специальный using-блок, гарантирующий, что метод Dispose соответствующего объекта будет вызван при выходе из using-блока. Статический анализ, обнаруживающий неправильное использование ресурсов, должен отслеживать работу с Disposable объектами, включая using-блоки.

Частично учитываются при анализе учитываются следующие возможности:

Косвенные вызовы. В языке C# представлено два механизма выполнения косвенных вызов функций: виртуальный вызов и вызов делегата. При виртуальном вызове вызываемый метод определяется по фактическому типу объекта. При вызове делегата фактически выполняется вызов набора методов, хранящихся в данном делегате. Таким образом, делегаты в языке C# являются безопасным аналогом вызова функции по указателю. При статическом анализе точное определение вызываемого метода при косвенном вызове зачастую невозможно. Поэтому на практике используются различные эвристики, зависящие от целей проводимого анализа.

При анализе косвенных вызовов детекторы имеют возможность получить список возможных кандидатов.

Поддержка многопоточности. Язык C# имеет несколько конструкций, упрощающих программисту разработку многопоточного приложения. Во-первых, язык C# поддерживает lock-блоки, позволяющие создавать критические секции. Во-вторых, поля классов, доступ к которым осуществляется из разных потоков, должны быть помечены специальным квалификатором volatile. Данный квалификатор гарантирует, что данные поля не будут подвергаться оптимизациям. В-третьих, для организации асинхронного выполнения методов в языке C# используется специальный синтаксис вызова методов с использованием ключевых слов async / await. Вызовы методов, помеченных ключевым словом async, принудительно выполняются в отдельном потоке. Для получения результатов вызова этих методов используется ключевое слово await.

При анализе не учитываются результаты записи в volatile переменные, т.к. предполагается, что другой поток может недетерминированно изменить их значения.

Поддержка замыкания. Язык C# позволяет определять анонимные функции, использующие локальные переменные метода, в котором они объявлены. Таким образом, локальные переменные могут быть неявно изменены вне метода объявления.

Анализ учитывает тот факт, что некоторая локальная переменная была использована в замыкании, однако полный анализ поведения анонимных функций с учётом замыканий не осуществляется.

Linq-выражения. Работа с коллекциями в языке C# зачастую осуществляется с помощью специальных Linq-выражений. Linq-выражения позволяют выполнять запросы к коллекциям, аналогичные запросам к базам данных.

Для некоторых Linq-выражений построены модели их поведения.

В данной работе не рассматриваются, в том числе, следующие возможности языка C#:

Небезопасные секции. В языке C#имеется возможность задавать специальные небезопасные секции, в которых разрешены адресные операции.

Использование отражений. Язык C#поддерживает отражение, позволяя модифицировать структуру и поведение программы прямо во время выполнения. Отражение включает в себя возможность неявного чтения и записи в поля объекта, вызова методов, генерацию кода на лету и т.д.

2.2 Описание внутреннего представления

В данной работе вводится внутреннее представление, описывающее основные операторы языка C#. Отметим, что инструмент SharpChecker в качестве инструкций внутреннего представления использует вершины абстрактного синтаксического дерева. Рассматривать внутреннее представление, аналогичное использующемся в SharpChecker, нецелесообразно по причинам громоздкости и избыточности. Предлагаемое в данной работе внутреннее представление состоит из описаний использующихся типов и реализаций соответствующих методов.

Описания типов совпадают с соответствующими описаниями в языке C#. Таким образом, для каждого типа известны списки полей, свойств и методов с соответствующими модификаторами, список реализуемых интерфейсов и класс-предок.

Для каждого метода класса, имеющего реализацию, внутреннее представление содержит ГПУ с дополнительными метаданными. Метаданные используются для связывания базовых блоков с конструкциями исходной программы. Примерами таких конструкций являются finally-блоки.

В ГПУ допускается наличие пустых базовых блоков. Для поддержки исключений в ГПУ выделено две специальные вершины: штатный выход из функции и выход из функции по исключению.

Каждый базовый блок содержит последовательность инструкций. Каждой инструкции соответствует дополнительная локальная переменная, значение которой равно результату выполнения этой инструкции. С точки зрения языка C# дополнительные локальные переменные соответствуют результатам вычисления выражений. Дополнительные локальные переменные могут быть использованы в базовых блоках, доминируемых базовым блоком, содержащим инструкцию. Переопределение дополнительных локальных переменных недопустимо.

Кроме дополнительных локальных переменных, для всего ГПУ определён набор параметров функции и набор локальных переменных. Тип параметров задан прототипом метода в соответствующем классе. Тип локальных переменных определяется по типу переменных, которыми они инициализируются.

Рассмотрим список основных инструкций, использующихся во внутреннем представлении.

$$val = a.f; \tag{2.1}$$

Инструкция чтения поля f по ссылке, находящейся в переменной a. Дополнительная локальная переменная val равна считанному значению.

$$val = a[i]; \tag{2.2}$$

Инструкция чтения i-того элемента массива a, val равна считанному значению.

$$a[i] = b; \tag{2.3}$$

Инструкция записи значения b в i-ый элемент массива a.

$$val = a \triangledown b \tag{2.4}$$

$\triangledown$ — одна из бинарных операций, присутствующих в языке C#. a, b — её операнды, val — результат операции.

$$val = \triangledown a \tag{2.5}$$

∇ — одна из унарных операций, присутствующих в языке C#. a — её операнд, val — результат операции.

$$val = a.f = b; \tag{2.6}$$

Запись в значение переменной b в поле f объекта a, val — результат записи.

$$val = a = b; \tag{2.7}$$

Присваивание в переменную a значения переменной b, val — записанное значение.

$$val = const; \tag{2.8}$$

Инициализация константой дополнительной локальной переменной.

$$val = foo(params); \tag{2.9}$$

Вызов метода foo с набором параметров $params$. Если необходимо, первым параметром передается объект, метод которого вызывается. В случае виртуального вызова вызванная функция определяется по первому параметру.

$$val = new\ T(params); \tag{2.10}$$

Создает новый объект типа T и вызывает конструктор, соответствующий переданному набору параметров.

$$return\ var; \tag{2.11}$$

Завершает выполнение метода, возвращая значение, находящееся в переменной var. Предполагается, что в ГПУ содержится лишь одна инструкция возврата.

$$throw\ var; \tag{2.12}$$

Завершает выполнение метода, бросая исключение, находящееся в переменной var.

В том случае, если из базового блока выходит более одного ребра, каждое ребро аннотировано локальной переменной типа bool. Переход по ребру возможен только в том случае, если данная переменная имеет значение $true$. Для каждого базового блока по построению верно, что нет такого пути выполнения, прошедшего через данный блок, на котором сразу две переменные, которыми аннотированы исходящие ребра, имеют значение $true$.

Для моделирования исключений в ГПУ добавляются дополнительные ребра, соответствующие возникновению исключения при выполнении последней инструкции базового блока. В случае инструкции $throw$ данное ребро будет единственным, выходящим из содержащего её базового блока. Инструкции вызова метода и конструктора во время своего выполнения, вообще говоря, могут бросить любое исключение. Однако при построении ГПУ необходимо учитывать только те исключения, которые обрабатываются в данном методе. Следовательно, в ГПУ добавляются ребра только для тех исключений, чьи обработчики присутствовали в исходном коде. Подробнее построение ГПУ с учётом брошенных исключений описано в разделе 6.3

Глава 3. Внутрипроцедурный анализ

В данной главе рассматривается вопрос использования символьного выполнения для проведения чувствительного к путям и к потоку внутрипроцедурного анализа. Для упрощения повествования в данной главе не рассматриваются вызовы других методов, соответственно путь выполнения полностью определяется начальным состоянием. Символьное выполнение с межпроцедурным анализом будет рассмотрено в следующей главе.

3.1 Символьное выполнение

При символьном выполнении будем считать, что каждая функция является точкой входа в программу. Входные параметры функции и состояние памяти могут иметь произвольные значения. Тогда параметризуем начальное состояние в точке входа в метод набором символьных переменных. Символьные переменные являются типизированными в соответствии с типами исходных полей и переменных. Дополнительно для символьных переменных, имеющих ссылочный тип, потребуем, чтобы их значения не являлись псевдонимами друг друга. Данное предположение, разумеется, ограничивает множество рассматриваемых состояний.

Для описания параметризации введем следующие множества:

– V — множество переменных, определяемых в функции;

– $P, P \subseteq V$ — множество параметров функции;

– S — множество символьных переменных;

– $S_R, S_R \subseteq S$ — множество символьных переменных ссылочного типа;

– $\vec{s}$ — вектор, состоящий из всех символьных переменных;

– F— множество полей, по которым осуществляется доступ.

Тогда множество начальных состояний программы на входе в метод будет описываться следующим образом:

– $\mathcal{P}_{entry} : P \to S$ — символьная параметризация параметров метода;

– $\mathcal{H}_{entry} : S_R \times F \rightharpoonup S$ — символьная параметризация состояния кучи.

Здесь и далее $entry$ — точка входа в метод.

Пару отображений $\mathcal{P}_{entry}$ и $\mathcal{H}_{entry}$, задающую символьную параметризацию на входе в метод, будем называть начальным символьным состоянием или $State_{entry}$. Подставляя вместо символьных переменных, содержащихся в $\vec{s}$, их допустимые конкретные значения, получим множество способов запустить метод. Обозначим данное множество как Σ. Предполагается, что значения символьных переменных полностью определяют результаты вычисления всех булевых выражений, а, следовательно, и путь выполнения. Если результат булевых выражений зависит от неизвестных значений, то для данных значений также введём символьные переменные, определяемые в точке входа. Так как вектор конкретных значений $\vec{\sigma} \in \Sigma$ однозначно определяет путь выполнения, то обозначим как $L(\vec{\sigma})$ путь выполнения, соответствующий вектору конкретных значений $\vec{\sigma}$.

Введем вспомогательные обозначения:

- $L(\vec{\sigma})_e$ — значения переменных и состояние памяти на ребре e пути выполнения $L(\vec{\sigma})$;
- Σ_l — множество векторов конкретных значений, для которых $L(\vec{\sigma})$ соответствует пути на ГПУ l.

Для определения символьного состояния в произвольной точке ГПУ введем SE — множество символьных выражений. Зададим SE при помощи следующего набора правил вывода:

- все числовые константы и `null` являются символьными выражениями;
- все символьные переменные являются символьными выражениями;
- если $x, y \in SE$ и типы xy допускают операцию $x \diamond y$, то $x \diamond y \in SE$;
- если x — символьное выражение, и его тип допускает операция ∇x, то $\nabla x \in SE$;
- если x, y — символьные выражения с одинаковым типом, а c — символьное выражение типа $bool$, то $ite(c, x, y) \in SE$. Функция ite является условным оператором и возвращает свой второй аргумент, если первый — истина, и третий аргумент в противном случае.

В отличие от обычного выполнения, при символьном выполнении зачастую переход при ветвлении может быть произведен по обеим веткам. Выполнение такого перехода накладывает дополнительные ограничения на значения символьных переменных. Для задания данных ограничений в символьном состоянии явно вводится предикат пути. Предикатом пути будем называть символь-

ное выражение $\mathcal{G}$ логического типа, задающее множество допустимых значений вектора символьных переменных — $\Sigma_{\mathcal{G}}$, $\vec{\sigma} \in \Sigma_{\mathcal{G}} \iff \mathcal{G}(\vec{\sigma})$.

Тогда символьное состояние в конкретной вершине пути ГПУ будем описывать набором $State = \langle \mathcal{V}, \mathcal{H}, \mathcal{G} \rangle$, где:

– $\mathcal{V} : V \to SE$ — параметризованные значения локальных переменных;

– $\mathcal{H} : S_R \times F \rightharpoonup SE$ — параметризованное состояние кучи;

– $\mathcal{G}$ — предикат пути.

Введём функцию конкретизации для символьного состояния — $\varphi(State = \langle \mathcal{V}, \mathcal{H}, \mathcal{G} \rangle, \vec{\sigma})$, которая в случае, если $\vec{\sigma} \in \Sigma_{\mathcal{G}}$, строит конкретное состояние для локальных переменных и памяти, подставляя в $\mathcal{V}$ и $\mathcal{H}$ вместо символьных переменных конкретные значения, соответствующие $\vec{\sigma}$.

Задача символьного выполнения заключается в построении символьного состояния для выбранного пути на ГПУ. Формально данная задача ставится следующим образом: для заданного пути l на ГПУ, начинающегося в $entry$ и заканчивающегося ребром e, необходимо вычислить символьное состояние $State_e$ такое, что:

$$\mathcal{G} = \Sigma_l \wedge \forall \vec{\sigma} \in \Sigma_l (L(\vec{\sigma})_e = \varphi(State_e, \vec{\sigma})). \tag{3.1}$$

Здесь и далее операция $\uplus$ обозначает переопределение отображения для заданных значений:

$$(A \uplus B)(x) = \begin{cases} B(x), & \textit{если } x \in domain(B); \\ A(x), & \textit{иначе} \end{cases} \tag{3.2}$$

Для построения символьных состояний для точек на пути l определим правила интерпретации инструкций относительно начального состояния в соответствии с их семантикой. Пусть e_{pred}, e_{succ} — ребра пути на ГПУ до и после инструкции $\mathcal{I}$ соответственно. Тогда введем следующие правила интерпретации для соответствующих инструкций.

$$\frac{\langle e_{pred} \rangle \models \langle \mathcal{V}, \mathcal{H}, \mathcal{G} \rangle \quad \mathcal{I} = "a = b;"}{\langle e_{succ} \rangle \models \langle \mathcal{V} \uplus \{a \mapsto \mathcal{V}(b)\}, \mathcal{H}, \mathcal{G} \rangle}$$

$$\frac{\langle e_{pred} \rangle \models \langle \mathcal{V}, \mathcal{H}, \mathcal{G} \rangle \quad \mathcal{I} = "a = const;"}{\langle e_{succ} \rangle \models \langle \mathcal{V} \uplus \{a \mapsto const\}, \mathcal{H}, \mathcal{G} \rangle}$$

$$\frac{\langle e_{pred}\rangle \models \langle \mathcal{V}, \mathcal{H}, \mathcal{G}\rangle \quad \mathcal{I} = "a = b \diamond c;"}{\langle e_{succ}\rangle \models \langle \mathcal{V} \uplus \{a \mapsto \mathcal{V}(b) \diamond \mathcal{V}(c)\}, \mathcal{H}, \mathcal{G}\rangle}$$

$$\frac{\langle e_{pred}\rangle \models \langle \mathcal{V}, \mathcal{H}, \mathcal{G}\rangle \quad \mathcal{I} = "a = \nabla b;"}{\langle e_{succ}\rangle \models \langle \mathcal{V} \uplus \{a \mapsto \nabla b\}, \mathcal{H}, \mathcal{G}\rangle}$$

$$\frac{\langle e_{pred}\rangle \models \langle \mathcal{V}, \mathcal{H}, \mathcal{G}\rangle \quad \mathcal{I} = "assume(c);"}{\langle e_{succ}\rangle \models \langle \mathcal{V}, \mathcal{H}, \mathcal{G} \wedge \mathcal{V}(c)\rangle)}$$

$$\frac{\langle e_{pred}\rangle \models \langle \mathcal{V}, \mathcal{H}, \mathcal{G}\rangle \quad \mathcal{I} = "a = b.f;"}{\langle e_{succ}\rangle \models \langle \mathcal{V} \uplus a \mapsto \mathcal{H}(\mathcal{V}(b), f), \mathcal{H}, \mathcal{G}\rangle)}$$

$$\frac{\langle e_{pred}\rangle \models \langle \mathcal{V}, \mathcal{H}, \mathcal{G}\rangle \quad \mathcal{I} = "a.f = b;"}{\langle e_{succ}\rangle \models \langle \mathcal{V}, \mathcal{H} \uplus \langle \mathcal{V}(a), f\rangle \mapsto \mathcal{V}(b), \mathcal{G}\rangle)}$$

При задании правил интерпретации не были рассмотрены инструкции вызова метода и доступа к массиву. Интерпретация инструкций вызова с учетом резюме вызванных функций будет рассмотрена далее. В данной работе не рассматриваются методы, позволяющие полностью поддерживать массивы, ввиду редкого их использования в программах на C#. Чтение из массивов предлагается трактовать как вызовы чистых функций. Более подробно поддержка массивов будет разобрана вместе с межпроцедурным анализом.

Лемма 3.1. Для всех символьных состояний, полученных в результате применения правил интерпретации, начиная из начального символьного состояния для заданного пути на ГПУ l, выражение (3.1) верно.

Для доказательства леммы заметим, что правила интерпретации инструкций были заданы в соответствии с семантикой. Проведем доказательство по индукции. Для точки входа равенство (3.1) верно по построению начальной параметризации. Пусть для первых k рёбер пути правила интерпретации построили корректное символьное состояние. Тогда ввиду того, что правила интерпретации сохраняют корректность, то и для ребра $k+1$ получим корректное символьное состояние.

Ввиду условия на отсутствие псевдонимов между символьными переменными считается, что результат сравнения двух символьных переменных на равенство задаётся некоторым неизвестным значением.

Используя данные правила интерпретации, можно организовать символьное выполнение. Однако предложенный способ вынужден перебирать все пути выполнения по отдельности. Даже без циклов число различных путей в ГПУ

экспоненциально зависит от числа ветвлений. Количество рассматриваемых путей можно сократить с помощью объединения нескольких символьных состояний в точках слияния. Такое объединение позволяет существенно уменьшить время анализа, однако требует доработки правил интерпретации и разработки правил объединения символьных состояний.

3.2 Развертка графа потока управления

Так как символьное выполнение может проанализировать лишь определённый набор путей выполнения, сразу определим пути, которые будут подвергаться анализу. Для этого введем понятие графа развертки.

Графом развертки для ГПУ $G = \langle V,E,v_{entry},v_{exit}\rangle$ будем называть ациклический ориентированный связанный граф с выделенными истоком и стоком $G' = \langle V',E',v'_{entry},v'_{exit}\rangle$, для которого определена функция соответствия $\varphi : V' \to V$ такая, что верно:

$$\varphi(v'_{entry}) = v_{entry} \quad \varphi(v'_{exit}) = v_{exit} \tag{3.3}$$

$$\forall v' \in V' \setminus \{v'_{entry}, v'_{exit}\} : \varphi(v') \notin \{v_{entry}, v_{exit}\} \tag{3.4}$$

$$\langle v', u'\rangle \in E' \implies \langle \varphi(v'), \varphi(u')\rangle \in E' \tag{3.5}$$

$$\langle v',u'\rangle \in E' \wedge \langle v',w'\rangle \in E' \wedge u' \neq w' \implies \varphi(u') \neq \varphi(w') \tag{3.6}$$

Утверждение (3.3) говорит о том, что выделенным вершинам v'_{entry}, v'_{exit} соответствуют v_{entry},v_{exit}. Утверждение (3.4) заявляет, что никакие другие вершины графа развертки не соответствуют v_{entry},v_{exit}. Утверждение (3.5) говорит, что ребро в графе развертки присутствует только в том случае, если аналогичное ребро представлено в ГПУ. И, наконец, (3.6) требует детерминированности от графа развертки: для каждой вершины исходящее ребро однозначно определяется вершиной ГПУ.

Ввиду утверждения (3.5) любой путь графа развертки соответствует некоторому пути ГПУ. Более того, ввиду (3.6) различные пути графа развертки соответствуют различным путям ГПУ. Очевидно, граф развертки может быть построен для заданного множества путей ГПУ различными способами. Например, графом развертки является дерево выполнений программы, проходящих

по обратным рёбрам не более k раз. Так как размер такого дерева экспоненциально зависит от размера исходной программы, предпочтительно использовать ациклические графы.

Рассмотрим один из возможных алгоритмов построения графа развертки для путей, проходящих по обратным рёбрам не более k раз. Множество обратных ребёр ГПУ обозначим как $BE, BE \subset E$. Введем операцию R, которая для графа потока управления G и вершины $v \in V$ строит новый граф $P = \langle V_P, E_P \rangle$, изоморфный подграфу G, состоящему из вершин и ребер, достижимых из v только по прямым ребрам. Изоморфизм между вершинами P и подграфом G задается отображением $T : V_P \to V$.

Итоговый граф развертки будет состоять из дерева компонент, для которых верно, что:

1. каждая компонента соответствует ациклическому подграфу G;
2. компоненты соединены ребрами, соответствующими обратным ребрам графа G;
3. путь от корня дерева до конкретной компоненты задает последовательность обратных ребер; сама компонента содержит все пути ГПУ, содержащие данную последовательность как подпоследовательность и не включающие других обратных ребер.

Первую компоненту — корень дерева — построим как $Comp_1 = R(entry)$, где $entry$ — точка входа в CFG, соответственно за T_{Comp_1} обозначим соответствие между вершинами $Comp_1$ и CFG. Рассмотрим алгоритм построения из некоторой компоненты графа развертки $Comp$ достижимых из неё компонент. Для этого рассмотрим все вершины $\{v_{comp}\}$ из $Comp = \langle V_{comp}, E_{comp} \rangle$ такие, что $\exists u \in V : \langle T_{Comp}(v_{comp}), u \rangle \in BE$. Таким образом, каждой из рассматриваемых вершин было поставлено в соответствие некоторое ребро $be \in BE$, обозначим множество данных ребер как BE_{comp}. Пронумеруем рассматриваемые обратные ребра в порядке топологической сортировки, i-тое ребро из данного множества обозначим как $be_i = \langle T_{Comp}(v_i), u_i \rangle$. Тогда компоненту $Comp_i$ из компоненты $Comp$ получим применением операции R к вершине u_i. Соответственно, добавим к дереву компоненту $R(u_i)$ и соединим её с компонентой $Comp$ при помощи ребра $\langle v_i, entry \rangle$, где $entry$ — точка входа в граф $R(u_i)$. При этом ребро $\langle v_i, entry \rangle$ будет соответствовать обратному ребру be_i.

На Рисунке 1 изображен пример применения развертки к графу потока управления. Обратные ребра обозначены пунктирными линиями. Имена компо-

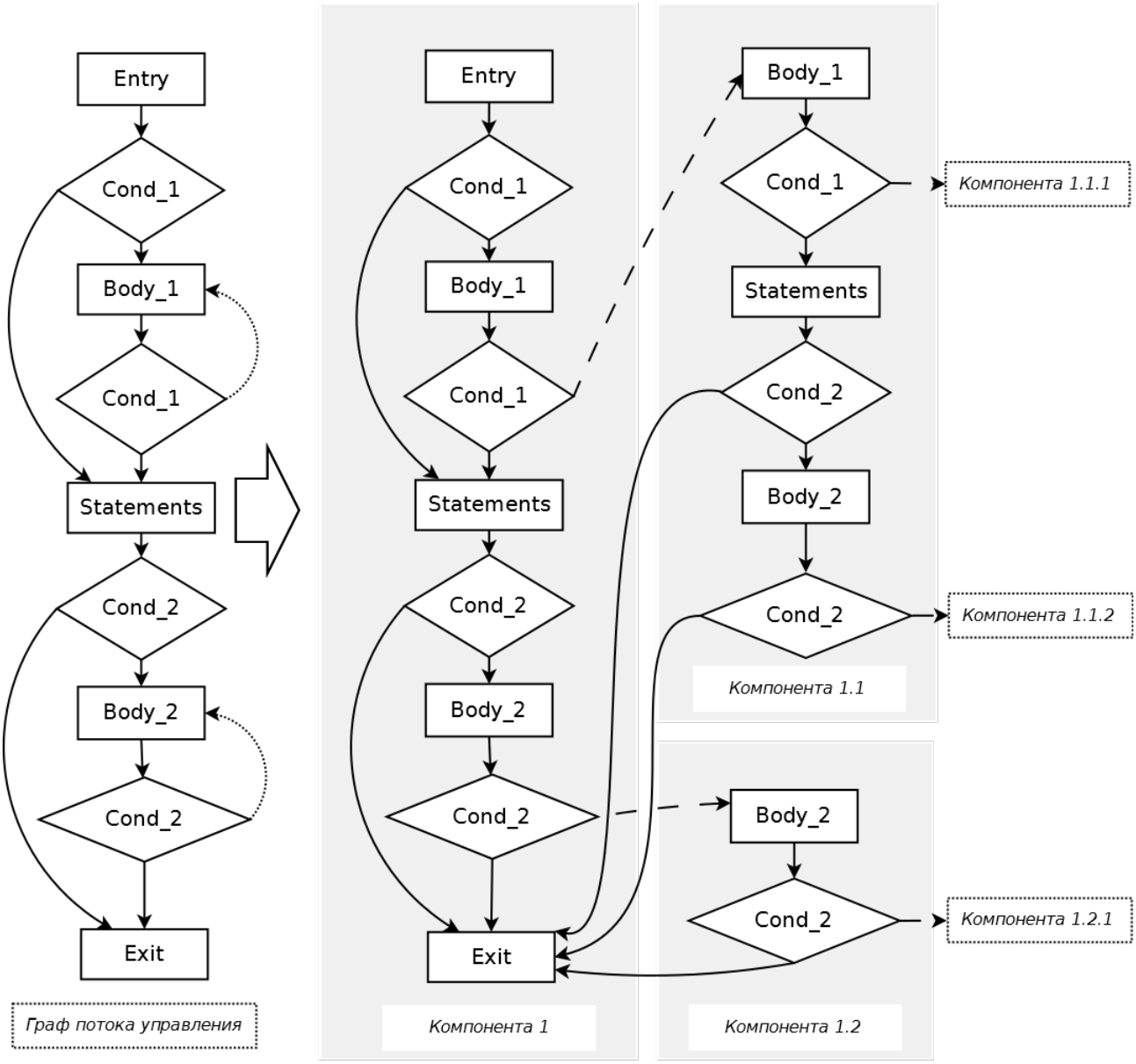

Рисунок 3.1 — Развертка графа

нент содержат полную информацию о том, как данная компонента была построена. Например, имя «Компонента 1.2» означает, что данная компонента была построена из «Компонента 1» при помощи обратного ребра номер 2 в топологическом порядке.

Рассмотрим все компоненты развертки, находящиеся в дереве на глубине не более k. Эти компоненты образуют граф развертки, и по свойству компонент (3) содержит все пути ГПУ, проходящие по обратным ребрам не более k раз. Таким образом, для того, чтобы ограничиться лишь набором конкретных путей, проходящих по обратным ребрам не более k раз, достаточно рассмотреть граф развертки, включающий все компоненты глубины не более k.

Переход от путей на графе потока управления является аппроксимацией снизу, т.к. происходит ограничение рассматриваемых путей. При этом данная аппроксимация уменьшает множество ошибок, которые способен обнаружить анализ.

3.3 Объединение состояний

Для перехода от анализа конкретного пути ГПУ к анализу графа развертки необходимо определить операции объединения символьных состояний, а также переопределить правила вывода для чтения и записи в память. Не нарушая общности, можно считать, что объединяются всегда лишь два состояния. В том случае, если в базовый блок входит более двух рёбер, добавлением пустых базовых блоков можно добиться того, что объединению будут подвергаться не более двух блоков.

Тогда задачу объединения символьных состояний сформулируем следующим образом. Пусть дана тройка вершин $\langle l,r,j\rangle$ таких, что есть ребра $\langle l,j\rangle$ и $\langle r,j\rangle$. Для вершин l и r известны состояния $\langle \mathcal{V}_l, \mathcal{H}_l, \mathcal{G}_l\rangle$ и $\langle \mathcal{V}_r, \mathcal{H}_r, \mathcal{G}_r\rangle$. Необходимо построить символьное состояние для вершины j, объединяющее информацию о путях, пришедших по ребрам $\langle l,j\rangle$ и $\langle r,j\rangle$.

Предикат при объединении вычисляется следующим образом: $\mathcal{G}_j = \mathcal{G}_l \vee \mathcal{G}_r$. По построению данный предикат допускает все возможные значения начальных состояний, соответствующих предикатам $\mathcal{G}_l$ и $\mathcal{G}_r$.

Для построения $\mathcal{V}_j$ и $\mathcal{H}_j$ необходимо найти такое символьное выражение C, что одновременно:

$$\Sigma_{\mathcal{G}_l} \subset \Sigma_C, \quad \Sigma_{\mathcal{G}_r} \cap \Sigma_C = \varnothing \tag{3.7}$$

Если выполнение дошло до вершины j из начального состояния, соответствующего значениям $\vec{\sigma}$, то можно гарантировать, что если $\vec{\sigma} \in \Sigma_C$, то было пройдено ребро $\langle l,j\rangle$, иначе $\langle r,j\rangle$. Тогда состояние памяти и переменных может быть выражено при помощи условного оператора от C следующим образом:

$$\mathcal{V}_j(v) = \begin{cases} \mathcal{V}_l(v), \ \textit{если } \mathcal{V}_l(v) = \mathcal{V}_r(v) \\ (C)?(\mathcal{V}_l(v)) : (\mathcal{V}_r(v)), \ \textit{иначе} \end{cases} \tag{3.8}$$

$$\mathcal{H}_j(s,f) = \begin{cases} \mathcal{H}_l(s,f), \ \textit{если } \mathcal{H}_l(s,f) = \mathcal{H}_r(s,f) \\ (C)?(\mathcal{H}_l(s,f)) : (\mathcal{H}_r(s,f)), \ \textit{иначе} \end{cases} \tag{3.9}$$

В качестве C может быть использован предикат $\mathcal{G}_l$. Действительно, для предиката $\mathcal{G}_l$ выполняется требование (3.7).

Заметим, что $\mathcal{V}(v)$ может быть определён только в одном из символьных состояний. В таком случае включать его в итоговое объединённое символьное состояние не требуется, т.к. далее переменная v использована не будет.

В результате операции объединения образуются символьные выражения ссылочного типа, содержащие условный оператор. Определённые ранее правила интерпретации, за исключением чтения и записи поля, сохраняют корректность. Определим правила интерпретации для операций чтения и записи поля для случая условных операторов.

Введём операцию $Decompose : SE \to 2^{S \times SE}$, которая для символьного выражения ссылочного типа строит набор пар из символьной переменной и условия, при котором исходное выражение равно данной символьной переменной. Все условия являются попарно несовместными. Результат $Decompose$ может быть вычислен с помощью обхода дерева выражений, содержащих условные операторы.

Введём операцию взятия поля $DEREF : SE \times F \to SE$, которая производит чтение из символьного выражения SE по полю f:

$$Decompose(se) = \{\langle s_i, c_i \rangle | i \in [1 \dots n]\}$$

$$|Decompose(se)| = n$$

$$h_i = \mathcal{H}(s_i, f)$$

$$DEREF(se, f) = \begin{cases} ite(c_1, h_1, ite(c_2, h_2, ...ite(c_{n-1}, h_{n-1}, h_n))), \textit{для } n > 1 \\ h_1, \textit{для } n = 1 \end{cases}$$

При помощи данных операций зададим интерпретацию для инструкций чтения и записи поля.

$$\frac{\langle e_{pred}\rangle \models \langle \mathcal{V}, \mathcal{H}, \mathcal{G}\rangle \quad \mathcal{I} = "a = b.f;"}{\langle e_{succ}\rangle \models \langle \mathcal{V} \uplus \{a \mapsto \mathit{DEREF}(\mathcal{V}(b), f)\}, \mathcal{H}, \mathcal{G}\rangle} \tag{3.10}$$

$$\frac{\langle e_{pred}\rangle \models \langle \mathcal{V}, \mathcal{H}, \mathcal{G}\rangle \quad \mathcal{I} = "a.f = b;" \quad \mathit{Decompose}(\mathcal{V}(a)) = \{s_i, c_i\}}{\langle e_{succ}\rangle \models \langle \mathcal{V}, \mathcal{H} \uplus_{i=1}^{n} \{\{s_i, f\} \mapsto \mathit{ite}(c_i, \mathcal{V}(b), \mathcal{H}(s_i, f))\}, \mathcal{G}\rangle} \tag{3.11}$$

Будем говорить, что символьное состояние $\langle \mathcal{V}, \mathcal{H}, \mathcal{G}\rangle$ относительно точки e определено корректно, если $\Sigma_{\mathcal{G}}$ содержит в точности все $\vec{\sigma}$, из которых e достижима, и для каждого $\vec{\sigma} \in \Sigma_{\mathcal{G}}$ верно, что $\varphi(\langle \mathcal{V}, \mathcal{H}, \mathcal{G}\rangle, \vec{\sigma}) = L(\vec{\sigma})_e$

Теорема 3.1. Определённые выше правила интерпретации строят корректные символьные состояния.

Для доказательства данного утверждения достаточно показать, что определённые выше операции объединения, записи в поле и чтения из поля переводят корректные символьные состояния в корректные символьные состояния.

Для начала докажем корректность определения операции объединения. Для этого покажем, что конкретизации полученного в результате объединения символьного состояния совпадают с конкретизациями состояния $\langle \mathcal{V}_l, \mathcal{H}_l, \mathcal{G}_l\rangle$ при $\vec{s} \in \Sigma_{\mathcal{G}_l}$ и с конкретизациями состояния $\langle \mathcal{V}_r, \mathcal{H}_r, \mathcal{G}_r\rangle$ при $\vec{s} \in \Sigma_{\mathcal{G}_r}$. Условие C всегда истинно при $\vec{s} \in \Sigma_{\mathcal{G}_l}$, а, значит, по определению $\mathcal{H}_j = \mathcal{H}_l$ и $\mathcal{V}_l = \mathcal{V}_j$, следовательно, их конкретизации также совпадут. Аналогично, условие C всегда ложно при $\vec{s} \in \Sigma_{\mathcal{G}_r}$, следовательно, $\mathcal{H}_r = \mathcal{H}_j$ и $\mathcal{V}_r = \mathcal{V}_j$. Предикат пути $\mathcal{G}_j = \mathcal{G}_l \vee \mathcal{G}_r$ задает все векторы символьных значений, из которых достижима точка j, так как в точку j можно попасть, лишь пройдя одну из вершин l, которая достижима из $\mathcal{G}_l$, или r, которая достижима из $\mathcal{G}_r$. Следовательно, объединение состояний сохраняет корректность.

Покажем теперь, что доступ к символьному условному выражению по полю f определён корректно. Заметим, что условия результата операции Decompose разбивают множество возможных начальных состояний Σ на множества непересекающихся подмножеств $\{\Sigma_{c_i}\}$. Действительно, если бы нашёлся $\vec{\sigma} \in \Sigma$ такой, что он не принадлежит ни одному Σ_{c_i}, то исходное условное выражение оказалось бы не определено для $\vec{\sigma}$, что невозможно. Аналогично, если бы нашелся такой $\vec{\sigma}$, что $\vec{\sigma} \in \Sigma_{c_j}$ и $\vec{\sigma} \in \Sigma_{c_k}$, при этом $j \neq k$, то для данного $\vec{\sigma}$ исходное условное выражение принимало бы сразу два различных значения, что невозможно. Заметим, что на каждом из Σ_{c_i} исходное условное символьное выражения равно символьной переменной ссылочного типа s_i. Из корректности объединённого символьного состояния в точке перед доступом следует, что для

любого пути l в ГПУ, содержащего данную инструкцию доступа, верно, что значение переменной при символьном выполнении без объединения состояний, к которой происходит доступ, совпадает со значением условного символьного выражения в объединённом состоянии при $\vec{\sigma} \in \Sigma_l$.

Пусть для пути l значение переменной равно s_i, тогда в случае символьного выполнения без объединения из данной переменной будет считано значение $\mathcal{H}(s_i, f)$. В случае символьного выполнения с объединением состояний, исходя из корректности, для пути l будет выполнено условие c_i, следовательно, по формуле (3.10) также будет прочитано значение $\mathcal{H}(s_i, f)$. А так как для символьного состояния относительно пути l корректность верна в силу леммы 3.1, то считывание одинакового символьного выражения означает, что конкретизации объединённого символьного состояния совпадет с $L(\vec{\sigma})_e$ при всех $\vec{\sigma} \in \Sigma_l$.

Аналогично показывается корректность операции записи в поле. В случае записи для пары $\langle s_i, c_i \rangle$ согласно правилу вывода (3.11) значение в поле f для переменной s_i будет изменено только в случае $\vec{s} \in \Sigma_{c_i}$, что соответствует всем путям, на которых значение переменной равно s_i.

3.4 Оптимизация предикатов

В предыдущей части были рассмотрены способы построения символьных выражений $\mathcal{G}_j$ и C, использующихся при объединении символьных состояний. Однако в силу того, что объединению зачастую подвергаются ветки оператора if с противоположными условиями, данные символьные выражения могут быть сильно упрощены. Такое упрощение необходимо как для уменьшения потребления памяти при сохранении выражений в резюме, так и для ускорения решения совместности формул.

Для упрощения выражений можно использовать два подхода. Первый подход заключается в использовании специального представления, автоматически упрощающего выражение во время его построения. Примером такого представления являются ROBBD [84].

Второй подход заключается в использовании свойства графа развертки: если две вершины достижимы из одного и того же множества начальных состо-

яний, то и предикаты путей в этих вершинах будут совпадать. Рассмотрим его подробнее.

Алгоритм 3.1. Выберем конкретную вершину в графе развертки — u. Будем считать, что все вершины, которые топологически меньше u, уже обработаны и их предикаты пути посчитаны. Найдем для u её область постдоминирования. Путь u не постдоминирует точку входа. Рассмотрим граф вершин, топологически меньших либо равных u, назовём его G'. Построим разрез $\langle S,T\rangle$ такой, что к S относятся все вершины G', которые в исходном графе не постдоминирует данная вершина, а к T — все вершины из области постдоминирования. Пусть $\langle s_i, t_i\rangle$ — список ребер, лежащих на разрезе. Тогда предикат пути для u построим следующим образом: $\langle \vee_i(\mathcal{G}(s_i) \wedge Cond(s_i,t_i)), \rangle$, где $\mathcal{G}(s_i)$ — предикат пути для вершины s_i, а $Cond(s_i,t_i)$ — условие перехода по ребру. В данном случае под условием перехода по ребру понимается условие в инструкции `assume` базового блока, в который входит ребро. Если у блока отсутствует `assume`, то условие считается тождественным истине.

Теорема 3.2. Алгоритм 3.1 корректно вычисляет предикат пути.

Доказательство. Если $S = \varnothing$, значит T постдоминирует точку входа и, следовательно, $\mathcal{G}(u)$ — тождественная истина. Если же $S \neq \varnothing$, то, так как любой путь начинается в S, то единственный способ попасть в T заключается в прохождении по одному из ребёр разреза $\langle S, T\rangle$. Следовательно, построенные условия $\langle \vee_i(\mathcal{G}(s_i) \wedge Cond(s_i,t_i)), \rangle$ включают в себя все начальные состояния, из которых достижима u. Однако любой путь, попав в границу постдоминирования u, гарантированно достигнет вершины u. Следовательно, $\langle \vee_i(\mathcal{G}(s_i) \wedge Cond(s_i,t_i)), \rangle$ в точности описывает начальные состояния, из которых достижима u. Что и требовалось доказать.

Стоит отметить, что оба подхода к минимизации размера формул могут применяться одновременно.

Перейдем к вычислению условия объединения. Пусть дана тройка вершин $\langle lhs,rhs,join\rangle$ таких, что есть ребра $\langle lhs,join\rangle$ и $\langle rhs, join\rangle$. Заранее известно, что вершина $join$ была достигнута либо на пути, прошедшем через ребро $\langle lhs,join\rangle$, либо на пути, прошедшем через ребро $\langle rhs, join\rangle$. Для того, чтобы различать, по какому из этих двух ребер была достигнута вершина $join$, достаточно воспользоваться условием $lhs_{cond} = \mathcal{G}(lhs) \wedge Cond(lhs,join)$, соответствующим тому, что путь прошел по ребру $\langle lhs,join\rangle$. Аналогично, можно рассмотреть обратное условие $rhs_{cond} = \mathcal{G}(rhs) \wedge Cond(rhs,join)$ для ребра

$\langle rhs,join\rangle$. В силу детерминированности выбранной параметризации условия lhs_{cond} и rhs_{cond} несовместны. Однако эти условия зачастую слишком громоздкие, и для хорошей производительности их необходимо упростить.

Задачу упрощения можно сформулировать следующим образом. Необходимо найти такое условие $Interpol$, что $Interpol \to lhs_{cond}$ и $\neg Interpol \to rhs_{cond}$ при условии того, что верно $lhs_{cond} \vee rhs_{cond}$. Это условие можно получить, вычислив интерполянт Крейга [85] для формул lhs_{cond} и rhs_{cond}. Вычисление интерполянта предлагается выполнять с помощью интерполирующего решателя [86]. Применение такого решателя на каждую операцию объединения достаточно затратно, поэтому рассмотрим алгоритм построения условия $Interpol$ на основе дерева доминаторов.

Алгоритм 3.2. Пусть dom — это непосредственный доминатор вершин lhs и rhs, тогда рассмотрим множество вершин, топологически меньше либо равных lhs и больше либо равных dom, назовём его L. Рассмотрим аналогичное множество для rhs, назовем его R. Рассмотрим тогда $L \cap R$ и $L \setminus R$, без ограничения общности будем считать, что $L \setminus R$ не пусто. Тогда $Interpol = \vee_i(\mathcal{G}_{dom,u_i} \wedge Cond(u_i,v_i))$, где $\langle u_i,v_i\rangle$ —- ребра, лежащие на разрезе $\langle L \cap R,L \setminus R\rangle$, а $\mathcal{G}_{dom,u_i}$ — условие того, что путь выполнения дойдет до u_i, пройдя dom.

Теорема 3.3. Алгоритм 3.2 корректно вычисляет условие $Interpol$.

Доказательство. Вершины топологически меньше либо равные lhs, но больше либо равные dom — это такие вершины, из которых одновременно есть путь до lhs и они достижимы из dom. Таким образом, L — это множество вершин, которые одновременно топологически больше или равны dom и через которые проходит путь в lhs. Заметим, что по построению не существует вершин топологически больших, чем dom, из которых достижимо lhs, но которые не содержатся в L. Аналогичное утверждение верно для R. Следовательно, $L \cup R$ содержат все пути, идущие к вершинам lhs и rhs. Тогда через разрез $\langle L \cap R,L \setminus R\rangle$ проходят все пути, идущие до lhs. Тогда, если для каждого ребра записать условие, что при условии того, что путь прошел через dom, путь пройдет именно по ребру e из разреза, то дизъюнкция условий всех ребер даст условие $Interpol$. Действительно, если для некоторого пути l известно, что он проходит либо через lhs, либо через rhs, то прохождение его по ребру данного разреза гарантирует то, что он попадёт в lhs. Аналогично, если путь l не прошёл ни по одному ребру разреза, то он не может попасть в lhs, следовательно, он проходит через rhs. Осталось показать, что условие $\mathcal{G}_{dom,u_i} \wedge Cond(u_i,v_i)$ яв-

ляется условием прохождения по ребру $\langle u_i, v_i \rangle$ при условии прохождения пути через dom. Действительно, условие $\mathcal{G}_{dom,u_i}$ соответствует тому, что путь дошёл до вершины u_i, а условие $Cond(u_i,v_i)$ — тому, что при достижении вершины u_i путь прошел по ребру $\langle u_i, v_i \rangle$. Следовательно, условие для ребра построено корректно, а, значит, и $Interpol$ построен корректно. Что и требовалось доказать.

Заметим, что в формулах, построенных с использованием алгоритмов 3.1 и 3.2, используются лишь символьные выражения из инструкций `assume` либо их отрицания. Следовательно, условия $\mathcal{G}_e$ и C можно рассматривать как условия, зависящие от лишь символьных переменных булевого типа, которые, в свою очередь, определяются через остальные символьные переменные. Данное замечание будет использовано в пятой главе.

3.5 Поддержка циклов с фиксированным числом итераций

Предложенный алгоритм символьного выполнения имеет очевидные проблемы при анализе циклов с фиксированным числом итераций. Рассмотрим следующий пример:

```
...
int a[] = new a[10000];

for (int i = 0; i < 10000; ++i) {
  a[i] = i*i;
}

...
```

Очевидно, что для того, чтобы символьному выполнению выйти из приведённого выше цикла, необходимо выполнить 10000 итераций. Для того, чтобы обеспечить возможность выхода из таких циклов, и, соответственно, анализ последующего кода, предлагается изменить семантику арифметических операций внутри циклов.

Предлагаемый подход схож с подходом, использующемся в инструменте Saturn. Однако в отличие от Saturn, где на неизвестные значения заменялись все правые части операций присваивания, в данной работе на неизвестные зна-

чения заменяются результаты арифметических выражений, из-за которых в большинстве случаев цикл имеет фиксированное количество итераций.

Благодаря применению данного подхода становится возможным корректный поиск ошибок в коде, находящимся после цикла, в случае, если наличие ошибки не зависит от вычисляемых в цикле значений. В приведённом выше примере, благодаря использованию данного подхода, выход из цикла произойдет после первой итерации цикла.

Глава 4. Межпроцедурный анализ

Данная глава посвящена вопросу учета вызовов при символьном выполнении. В главе рассматриваются три типа вызовов: прямые вызовы известных методов, косвенные вызовы и вызовы методов из сторонних библиотек.

4.1 Построение резюме

Для моделирования прямых вызовов необходимо разработать процедуру построения резюме метода. Резюме метода хранит в себе неполную информацию о его поведении, с точки зрения анализа являясь нестрогим. Информация, хранящаяся в резюме, и методы её сбора могут существенно различаться в зависимости от использующихся алгоритмов внутрипроцедурного анализа. Предлагаемый в данной работе метод построения резюме основан на опыте практической разработки анализаторов Svace и SharpChecker. Основной идеей является хранение в резюме состояния памяти до и после выполнения метода.

$$\langle \mathcal{V}_{PRE}, \mathcal{H}_{PRE}, \mathcal{V}_{POST}, \mathcal{H}_{POST} \rangle \tag{4.1}$$

Для построения резюме такого вида могут быть использованы результаты внутрипроцедурного символьного выполнения. В качестве $\mathcal{V}_{PRE}$ и $\mathcal{H}_{PRE}$ предлагается использовать начальное символьное состояние, а в качестве $\mathcal{V}_{POST}$ и $\mathcal{H}_{POST}$ — символьное состояние, полученное в точке выхода. Заметим, что во время выполнения метода может произойти исключение. Например, исключение может быть явно брошено посредством инструкции *throw*. Отметим, что для исключительного завершения выполнения метода не сохраняется состояние памяти. Данное решение принято с целью уменьшения размера резюме. Кроме того, не сохраняется информация об объекте, брошенном при возникновении исключения.

В случае, если некоторое выражение по той или иной причине должно быть исключено из резюме, то вводится новая символьная переменная, и данное выражение заменяется на неё. Например, громоздкие условные выражения,

содержащие большое количество условных операторов, предлагается заменять новыми символьными значениями.

На сохранённое резюме наложим ряд ограничений. Для начала ограничим количество записей в $\mathcal{H}_{PRE}$ и $\mathcal{H}_{POST}$ некоторой константой H_{SIZE}. Далее, ограничим число вершин в сохраненном ациклическом графе символьных выражений константой G_{SIZE}. Детекторы ошибок также могут сохранять информацию в резюме. Пусть детекторы сохраняют информацию с использованием символьных выражений. Будем считать, что для каждого анализатора $ANALYZER$ введена своя константа $G_{ANALYZER}$, ограничивающая число дополнительных символьных выражений, сохранённых в резюме для данного детектора.

Рассмотрим алгоритм сжатия кучи $\mathcal{H}_{PRE}$ и $\mathcal{H}_{POST}$. Для начала выделим символьные выражения, которые упоминаются в резюме каждого из детекторов. Обозначим такие символьные выражения как $SE_{ANALYZERS}$. Пары $\langle \mathcal{V}_{PRE}, \mathcal{H}_{PRE} \rangle$ и $\langle \mathcal{V}_{POST}, \mathcal{H}_{POST} \rangle$ задают графы памяти, в которых символьные переменные являются вершинами, а элементы $\mathcal{H}_{PRE}$ и $\mathcal{H}_{POST}$ — ребрами. В случае, если элемент $\mathcal{H}_{POST}$ указывает на символьное выражение se, содержащее условный оператор, то будем считать, что этот элемент задает набор рёбер, ведущих в вершины $Decompose(se)$. Добавим искусственную вершину $root$, ребра из которой будут вести в вершины, соответствующие символьным переменным, содержащимся в $\mathcal{V}_{PRE}$ и $\mathcal{V}_{POST}$ соответственно. Увеличим значение H_{SIZE} на число добавленных фиктивных рёбер.

Найдем на графе памяти $\langle V, E \rangle$ расстояние от вершины $root$ до всех ребер. Будем считать, что расстояние до ребра равно расстоянию до вершины, в которое данное ребро входит. Далее, каждому элементу $e \in E$ сопоставим число $\mathcal{D}_e$, равное расстоянию до соответствующего ребра в графе памяти.

Сжатие графа памяти будет проводить, учитывая использованные в резюме символьные выражения $SE_{ANALYZERS}$ и расстояния $\mathcal{D}_e$. Для начала выделим подграф графа памяти, вершинами которого являются символьные переменные, упомянутые в $SE_{ANALYZERS}$. Если размер данного подграфа больше, чем H_{SIZE}, то вначале удалим все ребра, не относящиеся к подграфу, а затем будем удалять из подграфа рёбра по убыванию $\mathcal{D}_\eta$. Если же размер подграфа меньше или равен H_{SIZE}, то будем удалять ребра по убыванию $\mathcal{D}_e$ из оставшейся части графа до тех пор, пока число ребер не окажется равным H_{SIZE}.

Для того, чтобы процедура удаления была детерминированной, будем считать, что всем вершинам графа памяти детерминированно назначаются номера,

и в случае равенства расстояний удаление производится по возрастанию номеров вершин.

Рассмотрим сжатие графа символьных выражений. Вершинами этого графа будем считать все символьные выражения, упомянутые в графе памяти. Между двумя символьными выражениями se_{from} и se_{to} есть ребро, если выражение se_{to} является оператором, операндом которого является выражение se_{from}. Аналогично, введём вспомогательную вершину $root$ и добавим ребра из неё во все вершины, являющиеся символьными переменными. Тогда по аналогии вычислим минимальное расстояние от $root$ до каждой вершины в графе — $\mathcal{D}_{se}$.

Также удалим из графа все выражения, которые не содержатся в памяти и не упоминаются среди $SE_{ANALYZERS}$. Если размер оставшегося графа выражений больше, чем G_{SIZE}, то воспользуемся схожей стратегией удаления. Будем удалять вершины из графа выражений в порядке убывания $\mathcal{D}_{se}$. Однако каждому анализатору $ANALYZER$ представляется возможность вернуть часть удалённых выражений в размере $G_{ANALYZER}$. Возвращение выражений происходит по возрастанию величины $\mathcal{D}_{se}$ независимо для каждого анализатора, то есть одно и тоже выражение может быть возвращено несколькими анализаторами. Данное решение было принято для того, чтобы обеспечить независимость множества возвращённых выражений от числа анализаторов.

Возможность возврата удалённых символьных выражений обусловлена тем, что размер графа выражений зачастую превосходит допустимое значение G_{SIZE}, таким образом, анализаторам необходимо дать возможность приоритизировать сохранение необходимых им символьных выражений.

4.2 Вспомогательные части резюме

На основе предложенной выше модели построения резюме для памяти и выражений построим вспомогательные резюме для сохранения возвращаемого значения методов и сохранения информации об исключениях.

Для сохранения возвращаемого значения достаточно сохранить символьное выражение $\mathcal{R}$, описывающее возвращаемое значение. В случае, если в методе несколько инструкций $return$, то данное символьное выражение получается

из возвращаемых выражений при помощи соответствующих условных операторов.

Информация о возможных исключениях сохраняется с помощью набора пар из типа исключений и условия его возникновения, представленного символьным выражением: $\mathcal{E} \subset 2^{Type \times SE}$. В явных инструкциях $throwObj$ в $\mathcal{E}$ добавляется информация о том, что исключение типа $Type$ будет брошено при условии $\mathcal{G}_{throw}$, где $\mathcal{G}_{throw}$ — предикат пути у вершины, содержащей инструкцию $throw$.

В резюме вызываемого метода может содержаться информация о том, что исключение типа T будет брошено при условии se. Тогда, если для точки вызова этого метода нет подходящего обработчика в анализируемом методе, то в $\mathcal{E}$ добавится информация о исключении типа T. Для моделирования условий возникновения недетерминированных исключений предлагается в качестве условия использовать неизвестную булеву переменную.

4.3 Применения резюме

Для применения резюме в точке вызова метода необходимо построить соответствие между памятью в точке вызова и состояниями памяти $\langle \mathcal{V}_{PRE}, \mathcal{H}_{PRE} \rangle$ и $\langle \mathcal{V}_{POST}, \mathcal{H}_{POST} \rangle$. Пусть метод вызван с набором параметров $\{se_i\}$, где se_i — символьное выражение, соответствующее i-тому аргументу вызова. Будем считать, что построено отображение $Formal2Actual_{\mathcal{V}}$, отображающее символьные переменные, соответствующие формальным параметрам метода, в символьные выражения, с которыми данный метод был вызван.

Опишем процедуру, позволяющую построить сопоставление между символьными выражениями вызываемой и вызывающей функции. Для начала сопоставим начальное состояние вызываемой функции, заданной парой $\langle \mathcal{V}_{PRE}, \mathcal{H}_{PRE} \rangle$, и текущее символьное состояние $\langle \mathcal{V}, \mathcal{H} \rangle$. Для этого воспользуемся следующим алгоритмом (4.1).

Использующаяся функция $DEREF$ аналогична по своей семантике функции (3.10).

Благодаря построенному отображению из символьных переменных вызванного метода в символьные выражения вызывающего возможно осуще-

```
Array of Type --- массив элементов типа Type;
Queue of Type --- очередь элементов типа Type;
Type1 --> Type2 --- отображение элементов типа Type1 в элементы
   типа Type2;
Set of Type --- множество элементов типа Type;
Type1 x Type2  --- кортеж из элементов типа Type1 и Type2.

PROC BuildTranslation(Formal2Actual : Array of  SE x SE,
  CallerHeap: S x F --> SE, CalleeHeap : S x F --> S) : SE -->
     SE
BEGIN
  result : SE --> SE = Empty
  visited: Set of SE  = Empty
  queue:  Queue of <SE, SE>  = Empty

  FOREACH <callee, caller> in Formal2Actual; DO
    visited.Add(callee)
    queue.Add(<callee, caller>)
    result.Add(<callee, caller>)
  DONE

  WHILE Queue.Size > 0 DO
    <callee, caller>  = Queue.Dequeue();
    FOREACH field in CallerHeap.Keys.Where(s == caller) DO
      pointsToCallee = CalleeHeap.Get(<callee, field>)
      pointsToCaller = DEREF(CalleeHeap, caller, field)
      result.Add(pointsToCallee, pointsToCaller);
      visited.Add(pointsToCallee);
      queue.Enqueue(<pointsToCallee, pointsToCaller>);
    DONE
  DONE

  return result
END
```

Листинг 4.1 Алгоритм построения функции $Translate$

ствить необходимую нам трансляцию символьных выражений. Для этого достаточно заменить все вхождения символьных переменных вызванного метода на соответствующие выражения вызывающего. Функцию, осуществляющую такую трансляцию, определим как $Translate : SE_{callee} \to SE_{caller}$. В случае, если

для символьного выражения вызванного метода нет аналогичного символьного выражения в вызывающем, то заводится новая неизвестная символьная переменная, соответствующая выражению вызванного метода.

Для завершения применения резюме необходимо обновить состояние памяти в точке вызова в соответствии с $\langle \mathcal{V}_{POST}, \mathcal{H}_{POST} \rangle$.

```
PROC UpdateMemoryGraph(Callee2Caller : Array of SE x SE,
   CallerHeap: S x F --> SE, CalleeHeap : S x F --> SE)
BEGIN

visited: Set of SE  = Empty
queue:  Queue of <SE, SE>  = Empty

FOREACH <callee, caller> in Callee2Caller; DO
    visited.Add(callee)
    queue.Add(<callee, caller>)
DONE

WHILE Queue.Size > 0 DO
  <callee, caller>  = Queue.Dequeue();
  FOREACH field in CallerHeap.Keys.Where(s == caller) DO
    pointsToCallee = CalleeHeap.Get(<callee, field>)
    pointsToCaller = DEREF(CalleeHeap, caller, field)
    translated = Translate(pointsToCallee)
    IF translated != pointsToCaller THEN
      CallerHeap[caller, field] = translated
      visited.Add(pointsToCallee);
      queue.Enqueue(<pointsToCallee, translated>);
    ENDIF
   DONE
DONE

END
```

Данный алгоритм позволяет обновить содержимое памяти в соответствии с обновлениями, произошедшими в вызванном методе. После выполнения данного обновления основное резюме метода считается применённым, и начинается применение вспомогательных частей резюме, определённых анализаторами.

Каждому анализатору для применения резюме предоставляется следующая информация:

– набор пар $\langle callee, caller \rangle$, добавленных в $queue$ во время работы $GetCallee2Caller$;
– набор пар $\langle callee, caller \rangle$, добавленных в $queue$ во время работы $ApplyMemoryGraph$;
– функция $Translate$ для трансляции символьных выражений вызываемого метода.

Рассмотрим, как данная информация может быть использована при трансляции дополнительный частей резюме $\mathcal{R}$ и $\mathcal{E}$. Для трансляции $\mathcal{R}$ достаточно воспользоваться функцией $Translate(\mathcal{R})$. Аналогично, для $\mathcal{E}$ достаточно произвести трансляцию условий возникновения всех перечисленных исключений.

4.4 Поддержка чистых методов и массивов

Некоторые методы, будучи вызванными с одинаковыми аргументами, всегда возвращают одно и то же значение. Такие методы будем называть чистыми. Одним из популярных примеров является работа с коллекциями. Рассмотрим следующий пример.

```
public string ToUpperIth(IList<string> strings, int index) {
  if (strings[i] != null) {
    strings[i] = "NULL";
  }
  return strings[i].ToUpper();
}
```

Приведенный выше код содержит описку на второй строчке. В ней оператор "не равно"использован вместо оператора "равно". Данная ошибка повлекла за собой возможное обращение к `null` на пятой строчке. Такое обращение может быть обнаружено предлагаемыми в данной работе алгоритмами анализа нулевых указателей. Однако для этого необходимо установить, что обращения `strings[i]` на 2 и 5 строчках возвращают одно и то же значение в случае, если сравнение на 2 строчке неверно.

Для поддержки таких ошибок добавим в анализ поддержку чистых вызовов. Введем глобальное для внутрипроцедурного анализа хранилище, сопоставляющее результаты чистых методов и наборы символьных выражений, соответствующие начальным состояниям, с которыми они были вызваны.

Пусть для чистого метода `Pure` известно его резюме $\langle \mathcal{V}_{PRE}, \mathcal{H}_{PRE} \rangle$. Тогда в точке его вызова, воспользовавшись алгоритмом (4.1), получим набор пар $\{\langle se_{callee}, se_{caller} \rangle\}$. Упорядочив символьные выражения se_{caller} из данного набора, получим набор $\{se_i\}$, описывающий начальное состояние вызванного метода `Pure`. Введем новую символьную переменную ret_{pure} для обозначения результата чистого вызова. Тогда для каждого чистого метода введем дополнительное отображение $\mathcal{P} : 2^S E \rightharpoonup SE$ и следующие правила обработки вызова метода `Pure(Params)`:

– Если набор символьных выражений $\{se_i\}$, переданный в метод `Pure`, не содержится в $\mathcal{P}$, то введём новую символьную переменную ret_{pure} и изменим кеш — $\mathcal{P} \uplus (\{se_i\} \mapsto ret_{pure})$.
– Если набор символьных выражений $\{se_i\}$ содержится в $\mathcal{P}$, то в качестве возвращаемого значения используем $\mathcal{P}(\{\})$.

Для моделирования работы с внешними коллекциями воспользуемся следующей моделью. Будем считать, что коллекция имеет некоторое внутреннее состояние $InnerState$, которое меняется при вызове методов, изменяющих коллекцию. Тогда между операциями изменения коллекции использование описанного механизма резюме гарантирует, что одинаковые обращения к элементам коллекции вернут одинаковые значения.

Однако данного подхода недостаточно для того, чтобы обеспечить нахождение ошибки в рассмотренном выше примере, т.к между операциями чтения присутствует операция записи. Поэтому если $InnerState$ является условным выражением, то, воспользовавшись операцией $Decompose(InnerState)$, получим набор возможных состояний, с которыми коллекция достигла текущей точки. Для каждого возможного состояния найдем с помощью $\mathcal{P}$ соответствующие им возвращаемые значения. Далее, используя условия из $Decompose(InnerState)$, построим условное выражение, объединяющее возможные возвращаемые значения. Благодаря применению такого механизма становится возможным обнаружить ошибку в приведённом выше примере.

Отдельно отметим, что данный механизм применим для любых условных символьных выражений, включенных в $\{se_i\}$. Однако использование его для всех выражений приведет к значительным накладным расходам, поэтому его предлагается применять только для $InnerState$.

4.5 Организация анализа помеченных данных

Сформулируем задачу межпроцедурного анализа помеченных данных. Пусть в программе заданы два множества внешних методов: истоки и стоки. Задачей анализа помеченных данных является поиск всех таких параметров вызовов функций из множества стоков, что они зависят по данным от какого-либо параметра вызова функции из множества истоков. Результатом такого анализа помеченных данных является набор путей в программе, вдоль которых помеченные данные распространяются от функции из множества истоков до функции из множества стоков.

Данная задача может решаться как в рамках описанной выше инфраструктуры, так и при помощи комбинации слайсинга и внутрипроцедурного анализа. Использование предложенных методов позволит организовать поиск утечек помеченных данных в том случае, если путь, связывающий сток и исток, достаточно короткий. Проблема в данном подходе возникает, например, в случае пропуска копирования внутри сложного метода. Такое копирование может являться ключевым при передаче помеченных данных из истока в сток, а его игнорирование приведет к пропуску ошибки. С другой стороны, использование стандартной инфраструктуры кардинально снижает расходы на проведение анализа помеченных данных, позволяя запускать его вместе с остальными анализаторами.

Подход, использующий комбинацию слайсинга и внутрипроцедурного анализа, предполагает сведение задачи межпроцедурного анализа помеченных данных к IFDS-задаче [87; 88]. В этом случае решение IFDS-задачи даст возможные пути без учёта условий переходов, по которым данные распространяются от истока до стока. Далее предлагается анализировать на совместность условий переходов лишь найденные пути. Для этого может быть использован рассмотренный алгоритм внутрипроцедурного анализа. В инструкциях вызова и возврата из методов вместо применения резюме предлагается начинать анализ вызванной функции по аналогии с вставкой. Для этого проводится трансляция символьного состояния, схожая с алгоритмами применения резюме.

Глава 5. Поиск дефектов

В данной главе рассматриваются различные критерии выдачи предупреждения о наличии дефекта, проводится сравнительный анализ данных предупреждений. Для приведенных критериев выдачи дефекта предлагаются алгоритмы их эффективного поиска на примере ошибок типа «доступ по нулевому указателю» и «утечка ресурсов».

5.1 Символьное выполнение для поиска ошибок

При использовании символьного выполения для поиска ошибок требуется подобрать входные данные, на которых произойдет ошибка. Пусть e — ребро на пути, рассматриваемом символьным выполнением, $\mathcal{G}_e$ — предикат пути для шага e, а Err_e — условие возникновения ошибки на шаге e, тогда наличие ошибки на шаге e задается следующей формулой:

$$\exists \vec{s} : \mathcal{G}_e(\vec{s}) \wedge Err_e(\vec{s}) \tag{5.1}$$

Однако в нашем случае, когда каждый метод программы рассматривается как потенциальная точка входа, условие наличия ошибки (5.1) не может быть использовано ввиду чрезмерного количества ложных срабатываний. Рассмотрим пример 5.1.

Листинг 5.1 Ошибка при символьном выполнении

```
1) public class A {
2)   public int X;
3)   public static int Foo(A a, bool five) {
4)     if (five)
5)       return a.X;
6)     else
7)       return 5;
8)   }
9) }
```

Напишем условие ошибки `NullReferenceException` для точки (5).

$$\exists a, five : five \wedge a = \texttt{null} \tag{5.2}$$

Данная формула имеет решение $a = \texttt{null}$ $five = true$, однако метод Foo может иметь неявные предусловия, запрещающие переменной a иметь значение `null`. На практике использование условия ошибки (5.1) для поиска `NullReferenceException` приведет к обнаружению ошибки практически в каждом методе. Поэтому эти неявные предусловия, другими словами, неизвестный контракт метода, приходится учитывать. Для этого введем гибкое определение ошибки в конкретной точке программы, позволяющее в зависимости от целей детектора описывать различные классы ошибочных ситуаций.

5.2 Определения ошибочных ситуаций

Рассмотрим множество $\{\Sigma_i\} \subseteq 2^{\Sigma}$, где $\Sigma_i \subseteq \Sigma$. Элемент Σ_i назовём *абстракцией*, а $\{\Sigma_i\}$ — набором абстракций или просто абстракциями. Будем говорить, что в вершине графа развертки B содержится ошибка, если найдется такая абстракция Σ_i из набора, что на всех векторах конкретных значений $\vec{\sigma} \in \Sigma_i$ произойдет ошибка в точке B. Запишем данное условие формально.

Пусть Σ_i выражается некоторым образом через символьные переменные $\vec{s}$. Тогда обозначим как $\mathcal{G}_B^{\Sigma_i}(\vec{s})$ формулу от символьных переменных, задающую условие того, что одновременно $\vec{s}$ принадлежит абстракции Σ_i и управление дойдет до точки B. Как $Err_B(\vec{s})$ обозначим условие того, что в точке B произойдет ошибка. Тогда дадим определение ошибки для абстракции Σ_i следующим образом:

$$Err_B^{\Sigma_i} = (\exists \vec{s} : \mathcal{G}_B^{\Sigma_i}(\vec{s})) \wedge (\forall \vec{s} : \mathcal{G}_B^{\Sigma_i}(\vec{s}) \rightarrow Err_B(\vec{s})) \tag{5.3}$$

Тогда наличие ошибки в точке B определим как существование абстракции, на которой произойдёт ошибка:

$$\exists \Sigma_i : Err_B^{\Sigma_i} \tag{5.4}$$

Определение (5.4) будем называть общим определением ошибки, $\langle\{\Sigma_i\}, Err_B\rangle$ — определением ошибки, полученным подстановкой $\{\Sigma_i\}$ и Err_B в общее определение ошибки. В зависимости от выбора множества абстракций $\{\Sigma_i\}$ будут различаться множества обнаруживаемых ошибок.

В некоторых случаях можно показать, что имеет место вложение множеств обнаруживаемых ошибок. Рассмотрим два различных множества абстракций $\mathcal{A}' = \{\Sigma'_i\}$ и $\mathcal{A}'' = \{\Sigma''_i\}, \mathcal{A}', \mathcal{A}'' \in 2^{\Sigma}$ таких, что:

$$\forall i \exists \vec{J} : \Sigma'_i = \bigcup_{j \in \vec{J}} \Sigma''_j \tag{5.5}$$

Лемма 5.1. Пусть $Err_B^{\mathcal{A}'}$ — множество ошибок для точки B при использовании абстракций $\mathcal{A}'$, а $Err_B^{\mathcal{A}''}$ — для абстракций $\mathcal{A}''$, тогда верно:

$$\forall B : Err_B^{\mathcal{A}'} \subseteq Err_B^{\mathcal{A}''} \tag{5.6}$$

.

Для доказательства (5.6) заметим, что если верно $Err_B^{\Sigma'_i}$ то для всех $\Sigma''_j \subseteq \Sigma'_i$, при условии достижимости B (то есть при $\mathcal{G}_B^{\Sigma''_j}(\vec{s})$) будет верно и $Err_B^{\Sigma''_j}$. Так как $\mathcal{G}_B^{\Sigma'_i}(\vec{s})$ верно, и $\Sigma'_i = \bigcup_{j \in \vec{J}} \Sigma''_j$, то среди абстракций $\{\Sigma''_j\}$ найдется такая Σ''_k, что $\mathcal{G}_B^{\Sigma''_k}(\vec{s})$ верно. Тогда значение Σ''_k является решением (5.4). Таким образом, определение ошибки (5.4) может быть использовано для доказательства вложения между различными классами ошибок.

5.3 Примеры множеств абстракций

Построим абстракцию, соответствующую определению ошибки (5.1). Для этого пронумеруем все значения множества $\Sigma = \{\vec{\sigma}_i\}$ и определим множество абстракций как $\Sigma_i = \vec{\sigma}_i$. Для данных абстракций определение ошибки (5.4) будет записано в виде: $\exists \vec{\sigma}_i : P_B(\vec{\sigma}_i) \wedge Err_B(\vec{\sigma}_i)$, который совпадает с (5.1).

С другой стороны, рассмотрим $\Sigma_1 = \Sigma$. Тогда определение (5.4) принимает вид:

$$\exists \vec{s}(\mathcal{G}_B(\vec{s})) \wedge \forall \vec{s}(\mathcal{G}_B(\vec{s}) \rightarrow Err_B(\vec{s})) \tag{5.7}$$

Определение (5.7) утверждает, что если точка B такова, что она достижима хотя бы на одном конкретном состоянии, и в ней всегда происходит ошибка, то

точка B ошибочна. Данное определение является самым строгим из рассматриваемых, поэтому ему свойственен пропуск реальных ошибок.

По числу обнаруживаемых ошибок между абстракциями $\Sigma_i = \vec{\sigma}_i$ и $\Sigma_i = \mathcal{G}_B$ находится абстракция путей. Идея данной абстракции основана на предположении, что все пути на ГПУ, представленные в графе развертки, не запрещены контрактом метода.

Для определения абстракции путей введем набор булевых переменных b_j для каждого выражения в графе развертки, имеющего логический тип. Значения переменных b_j зависят от значений символьных переменных: $b_j = b_j(\vec{s})$. Будем считать, что вычисление выражения логического типа всегда представлено ветвлением, зависящим от результатов этого выражения. Тогда пронумеруем все пути в графе развертки $\{l_i\}$. Для конкретного пути l_i обозначим набор переменных b_j, вычисленных в истину, как B_i^+, а набор переменных, вычисленных в ложь, как B_i^-. Заметим, что b_j не может одновременно принадлежать к B_i^+ и B_i^-, но может не принадлежать ни одному из них.

$$\Sigma_i = \bigwedge_j \begin{cases} b_j, & b_j \in B_i^+ \\ \neg b_j, & b_j \in B_i^- \\ true, & \text{otherwise} \end{cases} = \vec{l_i}(\vec{s})$$

Таким образом, каждая Σ_i определяет результат вычисления всех булевых выражений, которые в свою очередь определяют путь l'_i. Формула (5.4) для данной абстракции имеет следующий вид:

$$\exists \vec{l_i} : (\exists \vec{s} : (\mathcal{G}_B^{\vec{l_i}}(\vec{s}))) \wedge \forall \vec{s} : ((\mathcal{G}_B^{\vec{l_i}}(\vec{s}) \rightarrow Err_B(\vec{s})) \tag{5.8}$$

где $\mathcal{G}_B^{\vec{l_i}}(\vec{s}) = (\forall j : \vec{l}_{i,j} = \vec{l}_{i,j}(\vec{s})) \wedge \mathcal{G}_B$

Заметим, что для введённых абстракций верно следующее соотношение:

$$Err^{\Sigma_i = \mathcal{G}_B} \subseteq Err^{\Sigma_i = \vec{l_i}} \subseteq Err^{\Sigma_i = \vec{\sigma}_i}$$

Кроме того, вместо конкретных путей выполнения можно рассмотреть в качестве Σ_i различные наборы значений переменных $\vec{b_j}$. Преимуществом такого выбора Σ_i является удобство использования итогового определения для поиска ошибок при помощи SMT-решателей. В данном случае Σ_i предлагается определить следующим образом:

$$\Sigma_i = \bigwedge_j \begin{cases} b_j, \text{if } \left|\frac{i}{2^j}\right| \equiv 0 (mod\ 2), \\ \\ \neg b_j, \text{otherwise} \end{cases} = \vec{b_i}$$

Формула (5.4) для данной абстракции имеет следующий вид:

$$\exists \vec{b} : (\exists \vec{s} : (\mathcal{G}_B^{\vec{b}}(\vec{s}))) \wedge \forall \vec{s} : ((\mathcal{G}_B^{\vec{b}}(\vec{s}) \rightarrow Err_B(\vec{s})) \tag{5.9}$$

где $\mathcal{G}_B^{\vec{b}}(\vec{s}) = (\forall i : b_i = b_i(\vec{s})) \wedge P_B$

Введённая абстракция включает в себя абстракцию пути:

$$Err^{\Sigma_i = \mathcal{G}_B} \subseteq Err^{\Sigma_i = \vec{l_i}} \subseteq Err^{\Sigma_i = \vec{b_i}} \subseteq Err^{\Sigma_i = \vec{\sigma}_i}$$

Между наборами абстракций $\Sigma_i = \mathcal{G}_B$ и $\Sigma_i = \vec{l_i}$ находится набор абстракций k критических точек. При использовании абстракций из k критических точек точка B считается ошибочной в том случае, если найдутся k точек графа развертки таких, что любой путь, проходящий через них и через B, приводит к ошибке. Как можно заметить, при достаточно большом числе k данный набор абстракций будет эквивалентен абстракциям $\Sigma_i = \vec{b_i}$. Рассмотрим случай $k = 1$. Пронумеруем все точки в графе развертки -- $\{B_i\}$, тогда $\Sigma_i = \mathcal{G}_{B_i}$ и $\mathcal{G}_B^{B_i} = \mathcal{G}_{B_i} \wedge P_B$. Тогда общая формула ошибки записывается следующим образом:

$$\exists B_i : (\exists \vec{s} (\mathcal{G}_B^{B_i}(\vec{s})) \wedge \forall \vec{s} ((\mathcal{G}_B^{B_i}(\vec{s}) \rightarrow Err_B(\vec{s})) \tag{5.10}$$

Учитывая последнюю введённую абстракцию, напишем итоговое соотношение:

$$Err^{\Sigma_i = \Sigma} \subseteq Err^{\Sigma_i = \mathcal{G}_{B_i}} \subseteq Err^{\Sigma_i = \vec{b_i}} \subseteq Err^{\Sigma_i = \vec{\sigma}_i} \tag{5.11}$$

5.4 Примеры поиска ошибок при помощи предложенных абстракций

Для приведённых в предыдущей главе наборов абстракций рассмотрим набор искусственных примеров, демонстрирующий различие между ними. Листинги 5.2 - 5.7 содержат шесть потенциально ошибочных методов. Здесь и далее в качестве условия возникновения ошибки используется

Листинг 5.2 Ошибка при всех абстракциях

```
string ex1(object obj) {
  obj = null;
  return obj.ToString();
}
```

Листинг 5.3 Ошибка при $\Sigma_i = \{\mathcal{G}^{B_i}, \vec{l_i}, \vec{b_i}, \vec{\sigma}_i\}$

```
string ex2(object obj, bool f) {
  if (f)
    obj = null;
  return obj.ToString();
}
```

Листинг 5.4 Ошибка при $\Sigma_i = \{\mathcal{G}^{B_i}, \vec{l_i}, \vec{b_i}, \vec{\sigma}_i\}$

```
string ex3(object obj, bool f1, bool f2) {
  if (f1)
    obj = null;
  if (f2)
    return obj.ToString();
  return "";
}
```

$Err^{NULL}_{access(se)}(\vec{s}) = se == \texttt{null}$ для всех точек доступа к переменной, содержащей символьное выражение se.

В примере 5.2 переменная `obj` явно присваивается, таким образом, ошибка при доступе к `obj` неминуема и будет обнаружена всеми абстракциями.

Листинг 5.5 Ошибка при $\Sigma_i = \vec{l_i}, \vec{b_i}$, $\Sigma_i = \vec{\sigma}_i$

```
string ex4(int* obj, bool f1, bool f2, bool f3) {
  if (f1)
    obj = null;
  if (f2)
    obj = "";
  if (f3)
    return obj.ToString();
  return "";
}
```

Листинг 5.6 Ошибка при $\Sigma_i = \vec{b_i}$, $\Sigma_i = \vec{\sigma}_i$

```
string ex5(object obj, bool f1) {
  if (f1)
    if (obj == null)
      return "null";
    else
      return "non-null";
  else
    return obj.ToString();
}
```

Листинг 5.7 Ошибка при $\Sigma_i = \vec{\sigma}_i$

```
string ex6(object obj) {
  return obj.ToString();
}
```

В примере 5.3 присваивание `null` происходит только в случае, если переменная `f` истинна. В данном примере ошибка содержится по всем определениям, кроме $Err^{\Sigma_i=\Sigma}$. В случае, если определение $Err^{\Sigma_i=\Sigma}$ будет применено к инструкции `obj.ToString()`, то данная точка не будет признана ошибочной, т.к. существует путь, на котором переменная `f` ложна, следовательно, переменная `obj` может быть не `null`. Однако данная точка будет ошибочной по определению $Err^{\Sigma_i=\Sigma}$ в случае, если во всех точках присваивания `null` использовать в качестве условия ошибки то, что к присвоенной переменной неминуемо будет осуществлён доступ.

В отличие от примера 5.3, пример 5.4 вне зависимости от выбора $Err_{\texttt{null}}$ не будет содержать ошибку по определению $Err^{\Sigma_i=\Sigma}$. Все остальные абстракции обнаружат ошибку в данном примере. Пример 5.5 содержит дополнительную проверку `f2`, обновляющую значение obj в случае истинности. Ввиду наличия данной проверки, для обнаружения ошибки необходимы две критические точки, следовательно, определение $\Sigma_i = P^{B_i}$, предполагающее наличие одной критической точки, в данном случае не обнаружит ошибку.

В примере 5.6 отсутствует путь в ГПУ, на котором бы гарантированно возникала ошибка. Однако определение $\Sigma_i = \vec{b_i}$ обнаружит ошибки даже в данном случае. Действительно, для значений $\vec{b}$, равных $f1 == false, obj == \texttt{null}$,

обращение $obj.ToString()$ будет ошибочным. Наконец, в примере 5.7 ошибку обнаружит только $Err^{\Sigma_i=\vec{\sigma}_i}$.

5.5 Поиск ошибок по определению на примере обращения к нулевому указателю

Рассмотрим задачу поиска доступа к нулевому указателю. Условие возникновение ошибки для инструкции доступа по символьному выражению выглядит следующим образом: $Err^{NULL}_{access(se)}(\vec{s}) = se == \texttt{null}$. Для набора абстракций $\Sigma_i = \vec{b_i}$ можно организовать поиск ошибок данного типа по определению (5.4). Однако такой подход очень затратен с точки зрения производительности, т.к., во-первых, проверку на наличие ошибки придется осуществлять в каждой инструкции доступа, и, во-вторых, формула, проверяемая на совместность, достаточно сложна для решения с помощью SMT-решателей. Вместо этого предлагается использовать подход, одновременно не требующий проверки каждой инструкции доступа и использующий формулы без предикатов всеобщности, ограничиваясь проверкой формулы на выполнимость.

Покажем, что из набора абстракций $\Sigma_i = \vec{b_i}$ можно построить алгоритм, находящий ошибки из $Err^{\Sigma_i=\vec{b_i}}$ и не требующий при этом поддержки кванторов существования и всеобщности.

Пусть в каждой точке программы e и символьного выражения se определена формула $IsNull^{se}_e(\vec{b})$ такая, что $\mathcal{G}_e \wedge IsNull^{se}_e(\vec{b}) \rightarrow se == \texttt{null}$. Заметим, что $IsNull^{se}_e(\vec{b})$ явно зависит только от переменных $\vec{b}$, которые, в свою очередь, зависят от $\vec{s}$.

Теорема 5.1. Из выполнимости формулы $\mathcal{G}_e \wedge IsNull^{se}_e(\vec{b})$ следует наличие ошибки по определению (5.9).

Доказательство. Выполнимость формулы $\mathcal{G}_e \wedge IsNull^{se}_e(\vec{b})$ означает существование такого набора значений $\vec{\sigma}$, что верны $\mathcal{G}_e(\vec{\sigma})$ и $IsNull^{se}_e(\vec{b}(\vec{\sigma}))$. Подставим в (5.9) значение $\vec{b} = \vec{b}(\vec{\sigma})$. Тогда осталось доказать $\exists \vec{s}\ \mathcal{G}^{\vec{b}(\vec{\sigma})}_e(\vec{s})$ и $\forall \vec{s}\ \mathcal{G}^{\vec{b}(\vec{\sigma})}_e(\vec{s}) \rightarrow se == \texttt{null}$. Для доказательства первого утверждения достаточно подставить $\vec{s} = \vec{\sigma}$. Так как $\mathcal{G}_e(\vec{\sigma})$ верно, а $\vec{\sigma} \in \Sigma_{\vec{b}(\vec{\sigma})}$, то $\mathcal{G}^{\vec{b}(\vec{\sigma})}_e(\vec{\sigma})$ тоже верно. Для доказательства второго утверждения докажем импликацию $\mathcal{G}^{\vec{b}(\vec{\sigma})}_e(\vec{s}) \rightarrow IsNull^{se}_e(\vec{b}(\vec{s}))$. Для этого достаточно заметить, что $\mathcal{G}^{\vec{b}(\vec{\sigma})}_e(\vec{s})$ по опре-

делению верно только для $\vec{s}$ таких, что $\vec{b}(\vec{s}) = \vec{b}(\vec{\sigma})$, а так как $IsNull_e^{se}(\vec{b}(\vec{\sigma}))$ верно в силу выполнимости исходной формулы, то и импликация верна. Также по определению верно, что $\mathcal{G}_e^{\vec{b}(\vec{\sigma})}(\vec{s}) \to \mathcal{G}_e(\vec{s})$. Следовательно, верна следующая цепочка импликаций:

$$\forall \vec{s}\ \mathcal{G}_e^{\vec{b}(\vec{\sigma})}(\vec{s}) \to \mathcal{G}_e(\vec{s}) \wedge IsNull_e^{se}(\vec{b}(\vec{s})) \to se == \texttt{null}. \tag{5.12}$$

Значит, и исходное определение (5.9) верно.

Однако для снижения числа ложных срабатываний на практике предлагается использовать $Err^{\Sigma_i = \vec{l_i}}$. Примечательно то, что $IsNull_e^{se}(\vec{b})$ можно выбрать таким образом, что определениям $\mathcal{G}_e \wedge IsNull_e^{se}(\vec{b})$ и (5.8) будут соответствовать одни и те же ошибки, т.е определения будут эквивалентны. Фактически приведённый ниже алгоритм построения $IsNull_e^{se}(\vec{b})$ решает задачу помеченных данных, в которых истоками являются инструкции присваивания и сравнения с `null`, а стоками — инструкции доступа. Следовательно, из точного соответствия приведённого ниже алгоритма определению (5.8) следует эквивалентность формулировки (5.8) и анализа помеченных данных.

Однако не все виды ошибок, использующие (5.8), могут быть сформулированы в терминах анализа помеченных данных. Например, в том случае, если в определённых операциях с целочисленным типом их значения обязаны находиться в определённом интервале, то ошибки, заключающиеся в выходе за границу данного интервала, не удаётся сформулировать в терминах анализа помеченных данных. Причина, по которой такая формулировка невозможна, заключается в наличии арифметических операций, не учитывающихся должным образом в формулировке анализа помеченных данных.

Примерами ошибок, требующих подобный анализ, являются арифметический сдвиг или переполнение коллекции заданного размера. Данные ошибки на языке C# , с одной стороны, являются намного менее распространенными, чем аналогичные на языках C/C++. С другой стороны, поиск таких ошибок составляет существенную сложность в реализации и занимает достаточно времени анализа. Поскольку, судя по результатам других анализаторов, такие ошибки практически не встречаются в проектах на языке C#, в данной работе они рассматриваться не будут.

Перейдём к построению $IsNull_{e,l}^{se}(\vec{b})$ — условия того, что выражение se всегда равно `null` в точке e при всех $\vec{\sigma} \in \Sigma_l$. Рассмотрим причины доступа по указателю `null` на пути l в ГПУ. Так как все символьные переменные в началь-

ном состоянии могут принимать произвольные значения, то в точке доступа к переменной v существуют лишь две причины, по которым значение v для всех выполнений, следующих пути l, всегда равно `null`. Либо $\mathcal{V}(v) = \texttt{null}$, либо условие $\mathcal{G}_l$ таково, что $\mathcal{G}_l \to \mathcal{V}(v) == \texttt{null}$.

Воспользовавшись данным замечанием, зададим $IsNull^{se}_{e,l}(\vec{b})$ следующим образом:

$$SE^{-\texttt{null}} = SE \setminus \{\texttt{null}\} \tag{5.13}$$

$$\langle e_{entry} \rangle \models (\forall se \in SE^{-\texttt{null}} IsNull^{se}_{e,l}(\vec{b}) = false) \wedge (IsNull^{\texttt{null}}_{e,l} = true) \tag{5.14}$$

$$\frac{\langle e_{pred} \rangle \models \langle \mathcal{V}, \mathcal{H}, \mathcal{G}, IsNull_{e,l}(\vec{b}) \rangle \quad \mathcal{I} = \text{"}assume(v) \quad \mathcal{V}(v) \mapsto se == \texttt{null}}{\langle e_{succ} \rangle \models \langle \mathcal{V}, \mathcal{H}, \mathcal{G}, IsNull_{e,l}(\vec{b}) \uplus IsNull^{se}_{e,l}(\vec{b}) = true \rangle)} \tag{5.15}$$

Теорема 5.2. Ошибочные ситуации, соответствующие выполнимости формулы $\mathcal{G}^l_e \wedge IsNull^{se}_{e,l}(\vec{b})$ в точках доступа к символьному выражению se для всех возможных путей l, совпадают с ситуациями по определению (5.8).

Сначала докажем, что из (5.8) следует существование пути l такого, что $\mathcal{G}^l_e \wedge IsNull^{se}_{e,l}(\vec{b})$. Из (5.8) следует, что существует путь l и точка доступа к переменной v, e, такие, что $\Sigma_l \neq \varnothing$ и $\mathcal{V}(v) == \texttt{null}$ для всех $\vec{\sigma} \in \Sigma_l$. Σ_l — множество значений $\vec{s}$, при котором управление пойдет по пути l.

Пусть $\mathcal{V}(v) \mapsto \texttt{null}$. Так как $IsNull^{\texttt{null}}_{e,l}(\vec{b}) == true$ по определению, то осталось показать, что верна выполнимость $\mathcal{G}^l_e$. Заметим, что все $\vec{\sigma} \in \Sigma_l$ являются решением для $\mathcal{G}^l_e$ т.к. они соответствуют пути l, а путь l содержит e. А так как $\Sigma_l \neq \varnothing$, то у $\mathcal{G}^l_e$ есть решение. Следовательно, $\mathcal{G}^l_e \wedge IsNull^{se}_{e,l}(\vec{b})$ также верно.

Пусть в $\mathcal{V}(v)$ содержится не `null`. Тогда условие $\mathcal{G}^l_e$ таково, что $\mathcal{G}_e \to \mathcal{V}(v) == \texttt{null}$. В силу отсутствия псевдонимов, символьная переменная $\mathcal{V}(v)$ может присутствовать в $\mathcal{G}^l_e$ только в сравнении с `null`. Учитывая, что $\mathcal{G}_l$ — конъюнкция, получим, что $\mathcal{G}_l = \mathcal{V}(v) == \texttt{null} \wedge \mathcal{G}'_l$. Учитывая, что по мере прохождения по пути l предикат $\mathcal{G}^l$ не ослабевает, то найдется такая вершина u, $\mathcal{G}^l_{pred(u),u} \to \mathcal{V}(v) == \texttt{null}$ неверно, а $\mathcal{G}^l_{pred(u),u} \to \mathcal{V}(v) == \texttt{null}$ — верно. Такая ситуация могла возникнуть лишь в результате прохождения инструкции

$assume(u)$, где $\mathcal{V}(u) \mapsto \mathcal{V}(v) == \texttt{null}$. Следовательно, исходя из правила вывода (5.15), $IsNull_{e,l}^{\mathcal{V}(v)}(\vec{b})$ верно. Верность $\mathcal{G}_e$ показывается аналогично предыдущему случаю. А, значит, и $\mathcal{G}_e^l \wedge IsNull_e^{se}(\vec{b})$ также верно.

Докажем теперь обратное следствие. Пусть для пути l и точки доступа e к se верно $\mathcal{G}_e^l \wedge IsNull_{e,l}^{se}(\vec{b})$. Подставим в (5.8) путь l. Верность $\exists \vec{s} : (\mathcal{G}_e^{\vec{l}}(\vec{s}))$ следует из выполнимости $\mathcal{G}_e^l$. Осталось показать верность $\forall \vec{s}\, \mathcal{G}_e^{\vec{l}}(\vec{s}) \to \mathcal{V}(v) == \texttt{null}$. Опять же рассмотрим два случая. Либо $\mathcal{V}_e(v) \mapsto \texttt{null}$, либо в $\mathcal{V}_e(v)$ не $\texttt{null}$, но $IsNull_e^{\mathcal{V}_e(v)} == True$. Если для данного пути l верно $\mathcal{V}_e(v) \mapsto \texttt{null}$, то вне зависимости от s верно и $\mathcal{V}(v) == \texttt{null}$. Если же $\mathcal{V}_e(v)$ не $\texttt{null}$, но $IsNull_e^{\mathcal{V}_e(v)} == True$, то, значит, на пути l присутствует $assume(c)$ такой, что $\mathcal{V}_{assume(c)}(c) \mapsto \mathcal{V}_e(v) == \texttt{null}$. Следовательно, ввиду правил интерпретации, в конъюнкции $\mathcal{G}_e^l$ содержится конъюнкт $\mathcal{V}_e(v) == \texttt{null}$. Что в свою очередь означает, что $\forall \vec{s}\, \mathcal{G}_e^{\vec{l}}(\vec{s}) \to \mathcal{V}_e(v) == \texttt{null}$ имеет вид $\forall \vec{s} \mathcal{G}_e'^{\vec{l}}(\vec{s}) \wedge (\mathcal{V}_e(v) == \texttt{null}) \to \mathcal{V}_e(v) == \texttt{null}$, что, очевидно, верно вне зависимости от $\vec{s}$.

Следовательно, для каждой точки доступа верности (5.8) следует существование пути, для которого в точке доступа верно $\mathcal{G}_e^l \wedge IsNull_{e,l}^{se}(\vec{b})$, и наоборот.

Для применения $\mathcal{G}_e^l \wedge IsNull_{e,l}^{se}(\vec{b})$ на практике необходимо определить $IsNull_e^{se}$ без привязки к конкретному пути. Правило интерпретации (5.15) остаётся без изменений, однако необходимо определить правила объединения $IsNull_e^{se}$. Пусть дана тройка вершин $\langle l,r,j\rangle$ таких, что есть ребра $\langle l,j\rangle$ и $\langle r,j\rangle$. Для вершин l и r известны состояния $\langle \mathcal{V}_l, \mathcal{H}_l, \mathcal{G}_l, IsNull_l\rangle$ и $\langle \mathcal{V}_r, \mathcal{H}_r, \mathcal{G}_r, IsNull_r\rangle$. Аналогично, предположим, что имеется условие $C = C(\vec{b})$, позволяющее отличать ситуацию, когда путь выполнения прошел по ребру $\langle l,j\rangle$, от ситуации, когда путь прошел по $\langle l,r\rangle$. Тогда определим $IsNull_j$ в вершине j следующим образом:

$$IsNull_j^{se}(\vec{b}) = IsNull_l^{se}(\vec{b}) \wedge C(\vec{b}) \vee IsNull_r^{se}(\vec{b}) \wedge \neg C(\vec{b}) \tag{5.16}$$

Заметим, что ввиду того, что $\Sigma_C \cap \mathcal{G}_l = \mathcal{G}_l$ и $\Sigma_C \cap \mathcal{G}_r = \varnothing$, при $\vec{\sigma} \in \mathcal{G}_l$ оказывается, что $IsNull_j = IsNull_l$, и при $\vec{\sigma} \in \mathcal{G}_r$, аналогично, $IsNull_j = IsNull_r$. Таким образом, при объединении сохраняется корректность.

Тогда рассмотрим условие ошибки для точки программы без привязки к конкретному пути $\mathcal{G}_e \wedge IsNull_e^{se}(\vec{b})$. Заметим, что выполнимость $IsNull_e^{se}(\vec{b})$ означает существование пути l, для которого $IsNull_{e,l}^{se} == True$, и наоборот.

Тогда выполнимость условия $\mathcal{G}_e \wedge IsNull_e^{se}(\vec{b})$ эквивалентна существованию пути l, для которого верно $\mathcal{G}_e^l \wedge IsNull_{e,l}^{se}(\vec{b})$. Следовательно, ввиду теоремы 5.2 выполнимость $\mathcal{G}_e \wedge IsNull_e^{se}(\vec{b})$ эквивалентна определению (5.8).

5.5.1 Реализация поиска доступа к `null`

При реализации предлагается дополнить символьное состояние отображением $IsNull : SE \to SE$, реализующим вычисление $IsNull_e$. Заметим, что в случае, если для некоторого символьного выражения se $IsNull(se) = False$, то при реализации нет необходимости хранить $se \mapsto False$, т.к. отсутствие элемента в коллекции будет означать, что условие для данного элемента равно $False$.

В том случае, если в точке доступа к символьному выражению se, $IsNull(se) \neq False$, то выполним запрос к SMT-решателю, подав ему в качестве запроса условие $\mathcal{G} \wedge IsNull(se)$. Тогда, в случае, если SMT-решатель найдет решение для данного условия, то по теореме (5.2) в данной точке имеет место ошибка типа доступа к `null`. Более того, решение, найденное SMT-решателем, задает путь в графе развертки, для которого ошибка неминуема.

Для уменьшения числа запросов к SMT-решателям предлагается дополнительно вычислять также отображение $CannotBeNull : SE \to \{False, True\}$. Данное отображение переводит символьное выражение в True в случае, если символьная переменная гарантированно не равна `null` ввиду условий переходов. $CannotBeNull$ может быть построено следующим образом:

$$\frac{\langle e_{entry} \rangle \models \langle \mathcal{V}, \mathcal{H}, \mathcal{G}, CannotBeNull \rangle \quad \mathcal{I} = "assume(v) \quad \mathcal{V}(v) \mapsto se! = \texttt{null}}{\langle e_{succ} \rangle \models \langle \mathcal{V}, \mathcal{H}, \mathcal{G}, CannotBeNull \uplus se \mapsto True \rangle)} \tag{5.17}$$

$$CannotBeNull_j(se) = CannotBeNull_l(se) \wedge CannotBeNull_r(se) \tag{5.18}$$

В случае, если в точке доступа для символьного выражения se верно, что $CannotBeNull(se) = True$, доступ к `null` в данной точке невозможен, поэтому можно избежать запроса к SMT-решателю.

5.5.2 Межпроцедурный анализ доступа к `null`

Доступ к переменной, имеющей значение `null`, может произойти в вызванном методе, при этом инструкция, из-за которой данная переменная имеет значение `null`, находится в вызывающем методе. Для поддержки таких ситуаций предлагается дополнить резюме метода информацией о доступе к символьным выражениям. Для этого введём отображение $DerefInfo : SE \to SE$. Данное отображение ставит в соответствие символьному выражению se условие, задающее наборы входных значений вызванного метода, при которых выполнение дойдет до инструкции доступа к se.

Таким образом, в точках вызова для символьных выражений, к которым был произведён доступ внутри вызова, предлагается дополнительно проверять совместность условий $\mathcal{G} \wedge IsNull(se) \wedge DerefInfo(se)$. При этом предполагается, что символьные выражения вызванного метода, содержащиеся в $DerefInfo$, были транслированы в символьные выражения вызываемого метода. Методы, применяемые для уменьшения числа запросов к SMT-решателю, использующиеся при обычном доступе, также применимы в случае доступа в вызванном методе. Соответствие $se \mapsto False$ при реализации также задается как отсутствие se в соответствующей коллекции.

Так как $DerefInfo$ сохраняется в резюме, к нему также применяются правила, оптимизирующие его размер. Например, предлагается ограничить суммарный размер символьных выражений, содержащихся в $DerefInfo$. Кроме того, из $DerefInfo$ удаляются символьные выражения, которые более не доступны в $\langle \mathcal{V}_{pre}, \mathcal{H}_{pre} \rangle$ по причине уменьшения размера резюме.

5.6 Утечка ресурсов

В данном разделе рассматривается задача поиска дефектов типа утечка ресурсов и обсуждаются особенности реализации данного детектора.

5.6.1 Постановка задачи

В языке C# сборщик мусора автоматически освобождает память, связанную с управляемыми ресурсами, когда ресурс больше не используется. Но невозможно предсказать, когда это произойдёт. К тому же существуют неуправляемые ресурсы, такие, как открытые файлы или сетевые соединения. Для управления такими ресурсами существует интерфейс $IDisposable$, его метод $Dispose$ освобождает ресурсы явно.

Утечка ресурсов — это ситуация, когда неиспользуемый ресурс не освобождается. Ищутся случаи, когда объект класса, реализующего интерфейс $IDisposable$, перестаёт использоваться, но $Dispose$ не вызывается. Данная задача может быть сформулирована в терминах помеченных данных. Будем считать, что операции создания объекта класса-ресурса являются источником данных, а точка выхода из метода — стоком. Если найдется такой путь выполнения, вдоль которого созданный ресурс может добраться от точки создания до выхода из метода без сохранения в динамической памяти и освобождения, то будем говорить об утечке ресурса.

При обработке операций, создающих ресурсы, сохраняется информация о символьной переменной, содержащей ссылку на созданный ресурс. Если во время создания ресурса может произойти исключение, то ресурс помечается как несуществующий и не требующий освобождения.

5.6.2 Условие утечки

Для каждой символьной переменной s_r, соответствующей ресурсу r, будем хранить $LeakCond_{r,e}$ — условие того, что ресурс r не освобождён в точке выполнения программы e. Очевидно, что если в точку p_2 программы можно прийти только через точку p_1, то

$$LeakCond_{r,p_2} \to LeakCond_{r,p_1}$$

То есть условие утечки становится всё более строгим. В случаях, где неясно, стоит ли изменять $LeakCond$, предпочитается изменение, так как более строгие условия ошибки приводят к меньшему количеству ложных срабатываний.

Рассмотрим следующие правила вычисления и изменения $LeakCondition$:

1. В точке $Creation_r$ создания ресурса r

$$LeakCond_{r,Creation_r} := Cond_{Creation_r} \wedge s_r \neq \texttt{null}$$

2. В точке $Dispose_a$ вызова метода $Dispose$ переменной a для каждого ресурса r

$$LeakCond_{r,Dispose_a} := LeakCond_{r,Prev} \wedge s_r \neq \mathcal{V}(a),$$

где s_a — символьное значение a, $Prev$ — предыдущая точка программы. Так как предполагается, что для любых двух ссылочных переменных s_1 и s_2

$$s_1 = s_2 \leftrightarrow (s_1 = \texttt{null} \wedge s_2 = \texttt{null}),$$

то $LeakCond$ изменяется только для ресурсов r таких, что переменная s_r входит в выражение s_a. Аналогичная идея используется при изменении $LeakCond$ и в случаях, описанных ниже.

3. Оператор «using», являющийся синтаксическим сахаром, обрабатывается аналогично $Dispose$. При этом $LeakCond$ можно изменить при входе в $using(a)$, так как освобождение объекта a произойдёт раньше следующей точки выхода из метода, а порядок уточнения $LeakCond$ не важен.
4. При сохранении ресурса в переменной, не являющейся локальной (в глобальной переменной, поле класса, элементе массива, параметре, переданном по ссылке или лямбда-функции), считаем, что освобождение ресурса внутри анализируемого метода необязательно, и ответственность за его освобождение передаётся вовне. (Это предположение может заставить пропустить некоторые истинные срабатывания.) Таким образом, в точке присваивания $a = b$ для каждого ресурса r

$$LeakCond_{r,a=b} := LeakCond_{r,Prev} \wedge s_r \neq \mathcal{V}(b).$$

5. В точке $return\ a$ для каждого ресурса r

$$LeakCond_{r,return\ a} := LeakCond_{r,Prev} \wedge s_r \neq \mathcal{V}(a).$$

6. В точке $Call_f$ вызова метода f с аргументом a в качестве формального параметра x ($this$ рассматривается как параметр) из резюме метода f извлекается информация о том, сохраняет и освобождает ли она x (резюме не делает разницы между сохранением и освобождением). В случае, если по каким-то причинам резюме метода f недоступно, считается, что f сохраняет x. Возможно несколько вариантов.
   а) f не сохраняет x — ничего делать не нужно.
   б) f сохраняет x, и возвращаемое значение f не является ресурсом. В таком случае предполагаем, что f либо освобождает x, либо сохраняет его вне локальных переменных. Следить за ресурсом, записанным в a, после этого не надо. Для каждого ресурса r

$$LeakCond_{r,Call_f} := LeakCond_{r,Prev} \wedge s_r \neq \mathcal{V}(a).$$

   в) f сохраняет x, и возвращаемое значение f является ресурсом. Тогда объект, возвращаемый f, может
      – либо являться обёрткой над x (в таких случаях освобождение одного влечёт освобождение другого);
      – либо освобождать x, когда освобождается сам;
      – либо никак на x не влиять.

   Во всех этих случаях никаких изменений условий утечки не происходит, но сохраняется дополнительная информация. Чтобы отделить третий случай, в резюме f сохраняется информация о том, какие параметры f будут освобождены при освобождении возвращаемого значения. Чтобы различать первый и второй случаи, проверим, освобождает ли $Dispose$ возвращаемого значения f что-то, кроме x. Если нет, считаем, что возвращаемое значение f — обёртка, т.е. объект, содержащий в себе ресурс r. В первом и втором случаях в символьном состоянии сохраняется информация о связи ресурсов.

$LeakCond_{r,p}$ не является достаточным условием того, что в точке p ресурс r не освобождён, так как путь в ГПУ, по которому ошибка найдена, может быть невыполним (из-за неточности при работе с резюме). Это может породить ложные срабатывания. Оно также не является необходимым, так как сохранение

ресурса вне локальных переменных не гарантирует, что ресурс будет впоследствии освобождён.

Если параметр метода сохраняется хотя бы по одному пути исполнения (то есть его $LeakCond$ меняется хотя бы один раз), он считается сохранённым.

5.6.3 Связанные ресурсы

Как уже упомянуто в (5.6.2), сохраняется информация о связи ресурсов. Ниже более подробно объясняется, откуда она берётся и как используется.

Отслеживание освобождений Процесс получения информации о том, какие параметры метода освобождаются при освобождении возвращаемого значения, состоит из двух частей.

1. Для каждого класса, реализующего интерфейс $IDisposable$, анализируется метод $Dispose$. Для каждого выражения типа $a.Dispose()$ внутри этого метода, где a — поле класса, в резюме метода записывается информация об этом.
2. Для каждого метода, возвращающего ресурс, для каждого присваивания его параметра-ресурса полю класса в резюме записывается информация о том, что этот параметр будет освобождён при освобождении возвращаемого значения. Так как параметр может быть записан в поле только на некоторых путях исполнения и может быть затёрт другим присваиванием, теоретически это может привести к потере истинных срабатываний. Также возможны ложные срабатывания, так как не отслеживаются случаи, когда параметр был сначала записан в какую-то переменную, а уже она — в поле.

Условие утечки В случае, когда освобождение ресурса r_1 приводит к освобождению ресурса r_2, этот факт сохраняется в символьном состоянии. Когда в точке p происходит освобождение ресурса r', который может являться ресурсом r_1 (символьное значение освобождаемого ресурса содержит s_{r_1}), для каждой символьной переменной r, являющейся частью s_{r_2}, условие утечки должно

изменяться как

$$LeakCond_{r,p} = LeakCond_{r,Prev} \wedge (r_1 \neq r' \vee r_2 \neq r)$$

Для простоты полагаем, что

$$LeakCond'_{r,p} = LeakCond_{r,Prev} \wedge r_2 \neq r$$

Такое условие является более строгим ($LeakCond'_{r,p} \rightarrow LeakCond_{r,p}$) и может привести к потере истинных срабатываний.

5.6.4 Момент утечки

Если в точке выхода из метода (нормально или из-за исключения), в конце области видимости, либо в точке, где не осталось ссылок, по которым возможен доступ к ресурсу, условие утечки выполнимо, то сообщается о возможной ошибке.

Отслеживание ссылок и областей видимости не требуется непосредственно для поиска ошибок, так как ресурс, который был не освобождён и не имеет ссылок в текущей области видимости, всё ещё не будет освобождён к концу метода, и ошибка обнаружится там. Так что пропуск момента потери ссылки или выхода из области видимости не является критичным. Однако этот момент может быть полезен при показе ошибок пользователю.

Отслеживание ссылок Для того, чтобы отследить ситуацию, когда в локальных переменных не осталось ссылок на конкретный ресурс, будем хранить для каждой символьной переменной, соответствующей ресурсу, список всех переменных метода, которые могут указывать на этот ресурс. В точке присваивания $a = b$, где $\mathcal{V}(b)$ — ресурс:

1. просматриваются все символьные переменные, входящие в символьное выражение $\mathcal{V}(a)$, и для каждой символьной переменной s переменная a удаляется из списка символьных переменных, сохраняющих соответствующий ресурс;
2. просматриваются все символьные переменные, входящие в $\mathcal{V}(b)$, и для каждой символьной переменной s переменная a добавляется в список переменных, сохраняющих соответствующий ресурс.

Отслеживание области видимости Для каждой символьной переменной, соответствующей ресурсу, также хранится максимальная область видимости, в которой могут быть ссылки на этот ресурс. Отслеживание области видимости необходимо для корректного определения момента утечки ресурса. Если все локальные переменные, хранящие некоторый ресурс, становятся недоступны в текущей области видимости, то необходимо выдавать предупреждение об ошибке.

1. Для каждой переменной сохраняется область видимости, в которой она определена.
2. В точке присваивания $a = b$, где $\mathcal{V}(b)$ — ресурс, для каждой символьной переменной, входящей в $\mathcal{V}(s)$, её максимальная область видимости меняется на область видимости переменной a, если последняя больше.
3. В точке вызова метода f с аргументом a, хранящим ресурс, выбирается символьная переменная с максимальной областью видимости из символьного выражения, соответствующего возвращаемому значению f. Затем для каждой символьной переменной, входящей в $\mathcal{V}(a)$, её область видимости меняется на вышеописанную максимальную, если последняя больше.
4. Если возвращаемое значение f является обёрткой над a, изменение областей видимости происходит и в обратную сторону.
5. Если ресурсы r_1 и r_2 связаны таким образом, что освобождение r_1 приводит к освобождению r_2, то при изменении предположительной максимальной области видимости r_1 область видимости r_2 меняется также.

5.6.5 Объединение символьных состояний

Условие утечки Пусть символьные состояния для lhs и rhs объединяются в j, $LeakCond_r^j$ — условие утечки ресурса r в j, $LeakCond_r^{lhs}$ и $LeakCond_r^{rhs}$ — условия утечки ресурса r в lhs и rhs. C — условие, различающее рёбра $\langle lhs, j\rangle$ и $\langle rhs, j\rangle$

$$LeakCond_r^j := LeakCond_r^{lhs} \wedge C \vee LeakCond_r^{rhs} \wedge \neg C$$

Связанные ресурсы При объединении символьных состояний информация о связи ресурсов объединяется.

Ссылки При объединении символьных состояний наборы ссылок объединяются.

Область видимости Область видимости выбирается максимальной из представленных в символьных состояниях.

Несуществующие ресурсы Если ресурс существует хотя бы в одном из объединяемых символьных состояний, он считается существующим.

5.6.6 Поиск утечки ресурсов по критерию ошибки

Рассмотрим задачу поиска утечек ресурсов с точки зрения определения (5.4). Для этого будем считать, что для символьных переменных, являющихся ресурсами, дополнительно определены правила интерпретации в соответствии с изложенными выше правилами, позволяющие для каждого ресурса вычислить пометку, является ли он освобождённым или нет. Условием ошибки, соответственно, будем считать наличие неосвобождённого ресурса в точке выхода из метода. Тогда поиск утечки ресурса, по аналогии с поиском доступа к `null`, можно свести к проверки условия утечки на выполнимость, что и было описано выше.

5.7 Влияние нестрогости предложенного анализа на поиск ошибок

В главах 3 и 4 данной работы рассматривается комбинация методов внутрипроцедурного и межпроцедурного анализа, направленная на поиск дефектов. Подытожим причины, по которым данная комбинация методов представляет собой нестрогий анализ. Основными причинами нестрогости являются:

– развертка циклов;

– замена результатов арифметических выражений в цикле на неизвестные значения;
– предположение об отсутствии псевдонимов в точках входа;
– неточность построения резюме.

Развертка циклов. Использование развертки циклов приводит к пропуску ошибок, поскольку не рассматриваются все итерации циклов. Однако замена результатов арифметических выражений в цикле на неизвестные значения частично решает данную проблему ценой добавления ложных срабатываний. Такая замена позволяет выйти из циклов по целочисленной переменной с фиксированным числом итераций. Однако в случае, если в цикле происходит обработка событий, например, разбор входных данных, то некоторые последовательности событий могут остаться непроанализированными.

Замена результатов арифметических выражений в цикле на неизвестные значения. Такая замена приводит к тому, что множество рассматриваемых выполнений расширяется, допуская выполнимость формул, которые до замены были невыполнимыми. Следовательно, возможны ложные срабатывания, связанные с преждевременным выходом из цикла. Однако такие срабатывания относительно легко заметить на практике, т.к. в них присутствует невозможный переход. Также возможны и пропуски ошибок, поскольку ослабление формул может привести к тому, что определению (5.4) будет соответствовать меньше ошибок.

Предположение об отсутствии псевдонимов в точках входа. На практике псевдонимы в точках входа могут присутствовать. Хорошим примером метода, ожидающего псевдонимы в качестве параметров, является метод `Equals`, сравнивающий объекты на равенство. Наличие псевдонимов может приводить как к пропуску ошибок, так и к выдаче ложных срабатываний. Однако, во-первых, точная поддержка псевдонимов крайне сложна, во-вторых, как было показано в работах [44; 62; 81], отсутствие поддержки псевдонимов в точках входа незначительно влияет на результаты анализа.

Неточность построения резюме Неточность построения резюме приводит к тому, что, во-первых, некоторые значения, вычисленные вызванным методом, считаются неизвестными в вызывающем методе. Данная ситуация схожа с результатом арифметических операций в цикле, поэтому она также может приводить к ложным срабатываниям и к пропуску ошибок. Во-вторых, в результате неточности резюме может потеряться информация о присваиваниях.

Потеря этой информации может также приводить как к выдаче ложных предупреждений, так и к пропуску ошибок. Чистые функции, и массивы в частности, также могут служить причиной выдачи ложных предупреждений и пропуска ошибок. Кроме того, при построении резюме не учитываются рекурсивные вызовы.

Можно заметить, что ввиду корректности внутрипроцедурного анализа возникающие ложные срабатывания и пропуски ошибок вызваны одной или несколькими перечисленными выше причинами.

Глава 6. Инструмент SharpChecker

В данной главе описывается инструмент SharpChecker, в котором реализованы предлагаемые в данной работе алгоритмы. Проводится оценка производительности и анализ результатов инструмента SharpChecker.

6.1 Описание инструмента

Разработанный инструмент статического анализа SharpChecker способен различать более 100 типов ошибок, среди которых есть как использующие исключительно синтаксический анализ, так и анализ потоков данных, а также наиболее мощный, описанный в данной работе, чувствительный к контексту и путям выполнения межпроцедурный анализ. Для понимания особенностей обработки специфичных возможностей языка C# предварительно рассмотрим общую схему работы анализатора, представленную на рисунке 6.1.

Большинство проектов на языке C# используют для организации сборки механизмы, основанные на файлах проектов и решений, предоставляемые Microsoft Visual Studio. Разработанный анализатор использует инфраструктуру Roslyn для работы с файлами системы сборки, а также для компиляции исходного текста программы. Roslyn — это открытая компиляторная платформа, поддерживающая набор механизмов для разработки статических анализаторов. Инструмент SharpChecker использует только часть Roslyn, отвечающую за разбор файлов проектов, решений, в результате которой определяется множество файлов, используемое для сборки заданной программы или библиотеки, а также необходимое окружение. Кроме этого, Roslyn использован для построения абстрактного синтаксического дерева и таблицы символов каждого модуля компиляции. Для достижения хорошего качества результатов статический анализатор обязан учитывать правила сборки, определяющие, в первую очередь, правила для компоновщика, или, другими словами, граф вызовов.

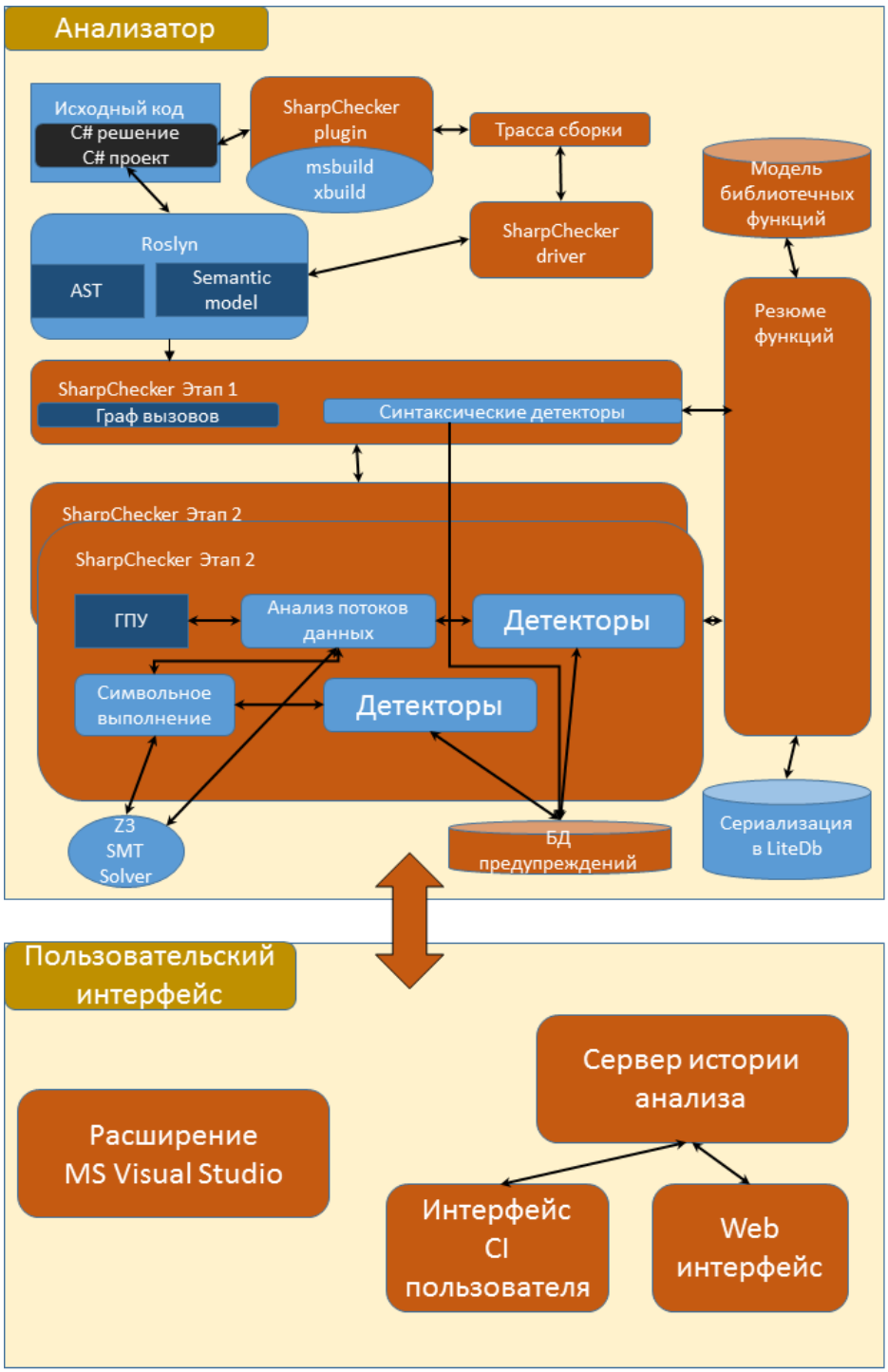

Рисунок 6.1 — Анализатор SharpChecker

6.2 Порядок анализа

Инструмент SharpChecker производит анализ проектов, либо используя собранную трассу сборки проекта, либо открывая проект с помощью встроенных средств Roslyn.

На первом этапе с помощью Roslyn выполняется построение абстрактного синтаксического дерева для всех файлов, участвующих в сборке, производится анализ всех возможных вызовов, в том числе неявных, строится иерархия наследования классов и происходит построение статического графа вызовов. Одновременно на этом же этапе выполняется поиск синтаксических ошибок, для обнаружения которых достаточно АСД.

Второй этап состоит в обходе графа вызовов в обратном топологическом порядке от вызываемого метода к вызывающему. При этом возможные циклы в графе вызовов разрываются в произвольном месте. В течение этой стадии анализа выполняются как анализ потоков данных и использующие его детекторы, так и чувствительные к путям детекторы. Кроме того, часть собранной информации сохраняется в резюме метода и используется при анализе методов, вызывающих данный. Обход методов производится параллельно, что позволяет существенно сократить время работы на многоядерных машинах.

По окончанию анализа метода выполняется проверка большинства правил, а также накопление информации о результатах анализа метода в резюме для последующего использования в вызывающих методах. Таким образом, в процессе обхода графа вызовов по предоставленным Roslyn АСД и таблице символов осуществляется построение графа потока управления для каждого метода. При этом анализатор использует два представления ГПУ для каждого метода, различающиеся обработкой лямбда-функций.

Классический анализ потоков данных включает в себя поиск неиспользованных значений, включающий как неиспользуемые переменные, так и результаты вычислений. При этом происходит обход базовых блоков ГПУ, во время которого детекторы вычисляют свои множества, соответствующие GEN и KILL в терминах классического анализа, а затем по правилам, заданным в тех же детекторах, обходится граф потока управления для построения IN и OUT. Наконец, вызывается обработчик завершения анализа, который использует собранную информацию для выдачи предупреждений.

После этапа анализа потоков данных работает чувствительный к путям анализ. Здесь выполняется накопление информации в резюме каждого метода и анализ всех возможных путей выполнения программы с учётом уже построенных резюме.

6.3 Построение графа вызовов

В рассматриваемом инструменте граф вызовов необходим для решения двух основных задач.

- Для каждого вызова в программе (с учётом делегатов, интерфейсов, геттеров и сеттеров свойств, лямбда-выражений) определить вызываемый метод или множество методов, которые могут быть вызваны.
- Построить правильный порядок обхода методов, чтобы вызываемые методы были проанализированы к моменту анализа вызывающих. Это позволяет достичь контекстной чувствительности за счёт использования резюме вызываемых методов.

Построение графа вызовов состоит из двух этапов. Сначала осуществляется обход АСД, во время которого происходит:

1. Анализ вершин, соответствующих объявлениям класса, для построения иерархии классов. Эта информация используется на втором этапе для уточнения графа в контексте виртуальных вызовов.
2. Анализ вершин АСД, в которых возможен вызов:
    а) непосредственный вызов метода, статического метода, конструктора;
    б) доступ к свойству объекта, когда возможен вызов геттера или сеттера;
    в) создание лямбда-выражений и анонимных методов;
    г) неявные вызовы переопределённых операторов;
    д) неявные вызовы операторов преобразования;
    е) вызовы финализаторов (деструкторов).

Для каждого узла АСД, в котором происходит вызов, осуществляется связывание по строковому идентификатору метода с существующим резюме или создаётся новое резюме. Полученный в результате граф топологически сортируется, после чего производится выбор методов, допускающих параллельный анализ.

6.4 Построение графа потока управления

Граф потока управления лежит в основе анализа и должен отражать все возможные пути выполнения программы. В рассматриваемом анализаторе базовые блоки состоят из вершин АСД. Это позволяет сохранить связь между сложными анализами путей исполнения и соответствующими оригинальными конструкциями языка. Использование языково-независимого представления, например, трехадресного кода, позволило бы упростить анализ за счет избавления от «синтаксического сахара». Однако такой подход приводит к потере информации об исходных конструкциях, что влияет на качество последующего анализа и, особенно, содержания предупреждений. В примерах 6.1,6.2 показан единственный содержательный базовый блок ГПУ для метода foo. Каждая инструкция базового блока — это вершина АСД, тип которой указан в правом столбце. Конструкция [число] означает номер инструкции в базовом блоке, результат которой используется в качестве аргумента в данной инструкции.

Листинг 6.1 C# код

```
                    
  public void foo()
  {
    int p = 1;
    bar(p,
    new MyClass());
  }
```

Листинг 6.2 внутреннее представление

```
0: 1                          LiteralExpression
1: int p = [0]              VariableDeclaration
2: MyClass                        IdentifierName
3: new [2]()          ObjectCreationExpression
4: p                              IdentifierName
5: bar                            IdentifierName
6: [5]([4], [3])          InvocationExpression
```

Ребра, соединяющие базовые блоки, хранят разные атрибуты, указывающие на то, является ли ребро обратным для циклов, соответствует положительной или отрицательной ветви условного оператора, является ли входом, выходом или замыканием лямбда-выражения, соответствует ли ребро переходу по пользовательскому или системному, явному или неявному исключению и некоторые другие. Кроме того, на рёбрах хранятся условия перехода для чувствительного к путям анализа.

Для упрощения работы с циклами на фазе чувствительного к путям анализа условие входа в цикл дублируется в его конце, чтобы позволить совершить выход из цикла без прохода по обратному ребру.

Существенное влияние на результаты анализа оказывает способ представления исключений в ГПУ. В языке C# практически каждая инструкция в исходном коде может вызвать исключение: например, `NullReferenceException`. Практика программирования также поощряет использование исключений. В итоге вызов практически любого метода может закончиться выбросом исключения. Практические эксперименты показали, что необходимо разделять пользовательские и системные исключения, выброшенные явно оператором `throw` и пришедшие из вызовов библиотечных методов. Такое разделение используется в детекторе утечки неуправляемой памяти в случае, например, отсутствия свободного места на диске или ошибки передачи данных по сети. Кроме того, для корректной обработки явных перехватов исключений требуется знать, какие исключения могут возникнуть во время выполнения метода.

Таким образом, при построении ГПУ для каждого вызова метода необходим список всех возможных в ней исключений. Для пользовательских методов этот список строится во время анализа, а информация о возможных исключениях при работе библиотечных методов загружается из базы данных резюме, более подробное описание которой содержится в части 5.

Разбиение базового блока и создание ребра для исключения после каждого вызова существенно увеличивает как сложность самого графа, так и время его последующей обработки. Поэтому на данный момент дробление базового блока осуществляется только для явных пользовательских исключений. Остальные добавляются как возможные выходы в конце базового блока. Однако это решение имеет недостатки. Рассмотрим пример 6.3, иллюстрирующий сравнительно частую ситуацию.

Если в методе Test не добавлять ребро между точкой вызова и присваиванием, то ошибку возможного использования `null` в reader.Dispose() найти не получится.

Широкое распространение в коде на C# имеют также лямбда-выражения и анонимные методы. Для упрощения будем называть все эти сущности одним словом лямбды. Без точного межпроцедурного чувствительного к путям анализа определить место их вызова невозможно. Поэтому для нужд различных детекторов используются различные эвристики. На этапе анализа потоков данных работают сравнительно простые детекторы, поэтому для них разумным компромиссом является непосредственное встраивание тела лямбды в ГПУ. Поскольку количество исполнений лямбды также неизвестно, добавляются ребро,

Листинг 6.3 Пропуск ошибки ввиду отсутствия ребра-исключения

```
public StreamReader GetStreamReaderOrThrow(bool b)
{
  if (b)
    return new StreamReader("stream.txt");
  throw new NotImplementedException();
}

public void Test(bool b)
{
  StreamReader reader = null;
  try
  {
    reader = GetStreamReaderOrThrow(b);
  }
  finally
  {
    reader.Dispose();
  }
}
```

минующее тело лямбды, и ребро, ведущее из ее выхода во вход. Такое встраивание позволяет повысить качество анализа по аналогии с компиляторной оптимизацией встраивания, давая анализу информацию о том, что будет происходить с переменными, использованными в лямбдах. Например, это позволяет существенно повысить качество анализа неиспользуемых значений без более слабых эвристик, считающих, что лямбды используют все переменные. По окончанию анализа потоков данных ребра, соединяющие тело лямбды с остальным методом, разрываются. Таким образом, в дальнейшем лямбды не учитываются.

Язык C# содержит достаточное количество удобного синтаксического сахара, который также требуется представлять в ГПУ. К таким конструкциям относятся операторы «??», «?.», `yield break`, `yield return`, `switch` по строкам и `goto` по вычисляемым выражениям, интерполированные строки и т.д. Основная сложность их поддержки в графе потока управления заключается в аккуратном создании базовых блоков и рёбер между ними, а также построении правильных условий переходов по этим рёбрам на этапе анализа путей.

Таким образом, построение ГПУ C# программы для последующего использования в статическом анализаторе — это поиск компромисса между слож-

ностью предшествующего построению анализа и последующего. Набор предложенных выше методов построения сохраняет большую часть информации в удобном для последующего обхода и анализа виде. Обходчик путей выполнения программы по ГПУ выдаёт набор предопределённых событий, обработчики которых реализованы в детекторах.

6.5 Резюме для внешних методов

С точки зрения анализатора методы делятся на пользовательские, т.е. те, для которых есть исходный код, и внешние, или библиотечные, — из системных или пользовательских библиотек. В связи с этим их резюме существенно отличаются.

Рассмотрим сначала, где можно взять информацию о внешних методах. Во-первых, их можно анализировать в бинарном виде. Поскольку большинство распространённых библиотек для C# скомпилировано в CIL — аналог байт-кода для языка C#— можно использовать, например, Mono.Cecil для дополнительного анализа бинарного представления. Одна из первых версий описываемого инструмента использовала этот подход, однако впоследствии решено было отказаться от анализа CIL, поскольку для получения качественных результатов необходима серьёзная инфраструктура, практически дублирующая функциональность анализатора исходного кода, что требует существенного времени на реализацию.

Поскольку большинство исходных текстов для популярных библиотек доступно для скачивания и анализа, другим способом является предварительный анализ исходного кода библиотеки по аналогии с пользовательскими методами или даже анализ на лету. Второй способ требует распространения огромного количества исходного кода библиотек вместе с инструментом, а также существенно увеличивает объем анализируемого кода, поскольку многие методы имеют сложную реализацию, кроме того, используют другие методы без исходного текста, включая неуправляемые библиотеки, например, WinAPI.

Предварительный анализ также не решает проблему с методами без исходного кода, а также требует постоянного обновления после исправления ошибок

или улучшения качества анализатора. Кроме того, для корректного анализа требуется поддержка различных версий одних и тех же библиотек.

Опыт разработки показывает, что для анализа библиотечных методов зачастую достаточно информации, представимой в виде нескольких десятков текстовых или логических атрибутов. Поэтому одним из используемых на данный момент решений является SQL база данных, содержащая записи для более 60000 популярных методов. База создавалась путём автоматизированного разбора и анализа официальной документации MSDN, а также исправлений и дополнений по результатам оценки работы инструмента. Для каждого метода хранится битовый вектор, представляющий логические свойства, например, что он создаёт Disposable объект, и различные другие свойства, например, список возможных исключений. Подобным образом хранится информация о параметрах, например, что они не могут быть `null`. Такой подход позволяет быстро, качественно и предсказуемо предоставлять информацию, достаточную для большинства детекторов анализатора.

Серьёзным недостатком данного подхода является сложность моделирования сторонних эффектов и условий их возникновения. Рассмотрим характерный пример 6.4.

Листинг 6.4 Пример сторонних эффектов

```
public string Foo()
{
  var list = new List<int>();
  list.Add(5);
  string check = null;
  foreach (var elem in list)
  {
    check = "Not null";
  }
  return check.ToString();
}
```

Код на C#, порождающий ложное срабатывание без дополнительного моделирования библиотечных методов. В результате добавления в *list* элемента 5 список *list* перестает быть пустым, поэтому тело цикла *foreach* всегда выполняется, и переменная *check* всегда инициализируется, следовательно, использование `null` невозможно. Однако анализатор предполагает, что *list* может быть пустым, потому что в описанной модели методов $List<T>.Add(T)$ нет воз-

можности прямо указывать подобные эффекты. Поэтому переход по ребру в обход $foreach$ возможен, что приводит к выдаче ложного предупреждения.

Для поддержки сложных эффектов используется моделирование внешних методов на языке C#. Модель метода в данном случае представляет собой упрощённую реализацию. Такая реализация компилируется и анализируется на лету как пользовательский метод, и для неё строится резюме, как для метода, имеющей исходный код.

Такое моделирование основывается на механизме сериализации резюме, позволяющему по имени сборки и полному имени метода сохранить его резюме. Далее при анализе проектов, в случае, если сериализованные данные будут доступны, при определении резюме вызываемого метода будет использована подготовленная заранее модельная версия. Сериализация выполняется с помощью стандартных средств языка C#.

6.6 Использование SMT-решателей

Для решения задач выполнимости формул, возникающих в ходе проведения анализа, предлагается использовать специализированные SMT-решатели. В данной работе используется решатель Z3 от компании Microsoft. В ходе работы анализатора решатель Z3 работает в отдельном процессе, взаимодействие с которым осуществляется с помощью потоков ввода/вывода. Запросы к решателю представляются в формате SMT-LIB2 [89]. В качестве основной теории используется теория битовых векторов (BV) в сочетании с теорией неинтерпретируемых функций (UF). Арифметические операции в языке C# представляются с помощью аналогичных операций с битовыми векторами. Ссылки моделируются как 32-битные битовые вектора. Операции с числами с плавающей точкой поддерживаются логикой BV, однако для ускорения анализа они не поддерживаются анализатором.

Неинтерпретируемые функции используются для моделирования проверок типа. Проверки соответствия выражения типу во внутреннем представлении имеют вид $se\ is\ T$, где se — некоторое символьное выражение ссылочного типа, а T — тип в программе. При выполнении SMT-запроса для проверяемых символьных выражений строится список участвующих в них типов $\{T_i\}$, име-

ющий длину n. Каждому типу из $\{T_i\}$ ставится в соответствие константа c_i, имеющая тип n-битного вектора. j-тый бит данной константы равен 1 тогда и только тогда, когда тип $\{T_i\}$ включает, с учётом наследования, тип $\{T_j\}$. Введём неинтерпретируемую функцию $GetType(se)$, которая принимает 32-битный вектор, соответствующий символьному выражению ссылочного типа, и возвращает n-битный вектор. Будем считать, что для каждого ссылочного выражения функция $GetType(se)$ возвращает его тип. Тогда проверка $se\ is\ T_i$ эквивалентна тому, что $GetType(se)\ and\ c_i == c_i$. Действительно, для того, чтобы se считалось ссылкой типа T, её тип должен включать все типы, включённые в тип T.

Листинг 6.5 "Пример SMT-запроса"

```
(set-logic QF_UFBV)
(declare-fun GetType ((_ BitVec 32)) (_ BitVec 2))
(declare-fun N () (_ BitVec 32))
(declare-fun F () (_ BitVec 32))
(declare-fun T () (_ BitVec 32))
(declare-fun X2 () (_ BitVec 32))
(assert (and (= T (_ bv1 32)) (= F (_ bv0 32)) (= N (_ bv100 32))
   )))
(assert
(let ((Let0 (ite (and (= (bvand (GetType X2) (_ bv3 2))
 (_ bv3 2)) (= (bvand (GetType X2) (_ bv0 2))
  (_ bv0 2))) X2 N)))
(let ((Let1 (and (and (= (bvand (GetType X2) (_ bv2 2))
 (_ bv2 2)) (= (bvand (GetType X2) (_ bv0 2))
 (_ bv0 2))) (not (= X2 N)))))
(and (= Let0 Let0) (= Let0 N) Let1 ))))
(check-sat)
```

В листинге 6.5 приведён пример SMT-запроса к решателю, использующий неинтерпретируемую функцию GetType для определения типа ссылочного выражения.

6.7 Результаты

Для оценки характеристик предложенных методов и алгоритмов было проведено тестирование инструмента SharpChecker, реализующего предложенные методы и алгоритмы, на наборе проектов с открытым исходным кодом. Размеры проектов варьировались от пятидесяти тысяч строк кода до полутора миллионов. Тестирование проводилось на компьютере с процессором Intel Core i7 6700 и 32 гигабайтами оперативной памяти. Результаты тестирования приведены в таблице 1. Группа дефектов NRE содержит дефекты, связанные с доступом к `null` в соответствии с определением 5.8. В группу NRE также включены межпроцедурные ошибки. Группа утечка ресурсов содержит ошибки, связанные с утечкой ресурсов и соответствующие ошибочным ситуациям, описанным в разделе 5.6.

Название проекта	Размер (LoC)	Время анализа	NRE	Утечка ресурсов
Jil	50 тыс.	3 мин.	32	2
CSParser	60 тыс.	1 мин.	7	0
CSharpUtils	108 тыс.	2 мин.	21	107
Lucene.net	256 тыс.	19 мин.	33	876
SharpDevelop	1213 тыс.	23 мин.	140	182
CodeContracts	1534 тыс.	14 мин.	64	23

Таблица 1 — Результаты тестирования инструмента SharpChecker.

Была проведена оценка эффективности оптимизации размера формул, при помощи алгоритмов 3.1 и 3.2. На проекте Lucene.net суммарное время формирования и выполнения запросов к SMT-решателю в однопоточном режиме работы при использовании наивных алгоритмов построения предикатов составило 12 минут 35 секунд, а с использованием алгоритмов 3.1 и 3.2 — 8 минут 36 секунд. Таким образом, время, проведённое в решателях, сократилось на 32%. Кроме того, применение данных алгоритмов позволило уменьшить размер формул, сохраняемых в резюме.

Для оценки числа истинных срабатываний был использован следующий метод. Из общего набора срабатываний случайно равновероятно выбирались с повторениями срабатывания, после чего в случае, если тип срабатывания (истинное или ложное) ещё не был определён, то определялся тип данного срабатывания. Таким образом, была получена выборка, каждый элемент которой представлял собой либо истинное срабатывание, либо ложное срабатывание. Так как число истинных и ложных срабатываний в исходном наборе при всех операциях выбора не менялось, то данная выборка соответствует распределению Бернулли с неизвестным параметром p, соответствующим числу истинных срабатываний. Оценку параметра p предлагается проводить при помощи 95-процентных доверительных интервалов.

Для групп дефектов NRE и утечки ресурсов были сформированы выборки, по которым проводилась оценка отношения истинных и ложных срабатываний. Оценка показала, что для группы NRE процент истинных срабатываний находится в 95-процентом доверительном интервале 57-74%, а для группы утечки ресурсов — в интервале 72-87%.

Таким образом, разработанный инструмент удовлетворяет требованиям, предъявляемым к промышленным инструментам статического анализа.

Заключение

Основные результаты диссертационной работы

1. Разработан алгоритм внутрипроцедурного анализа, обладающий чувствительностью к потоку и путям, при этом не теряющий информацию в точках объединения.
2. Разработан алгоритм межпроцедурного анализа, чувствительного к контексту, потоку и путям, обобщающий результаты внутрипроцедурного анализа в виде резюме метода и применяющий резюме при обработке вызова данного метода.
3. Разработан критерий выдачи предупреждений, учитывающий чувствительность к путям при поиске дефектов, использование которого позволяет достичь приемлемого уровня ложных предупреждений. Для данного критерия разработаны детекторы, позволяющие обнаружить ошибки вида «доступ по нулевому указателю» и «утечка ресурсов».
4. Доказана корректность предложенного алгоритма внутрипроцедурного анализа и соответствие разработанных детекторов выбранному критерию выдачи предупреждений.
5. Разработан инструмент статического анализа, реализующий разработанные алгоритмы для анализа программ на языке C#. Для данного инструмента проведена экспериментальная оценка его характеристик на соответствие заявленным требованиям. Предложенные алгоритмы и методы позволяют проводить анализ проектов, состоящих из более миллиона строк кода, в отведённое время. При этом качество результатов анализа соответствует заявленным требованиям (более 50% истинных срабатываний для реализованных детекторов).

В дальнейших работах планируется увеличить число поддерживаемых ошибочных шаблонов с помощью поддержки косвенных вызовов. Также планируется разработать детекторы позволяющие обнаружить ошибки связанные с безопасностью, такие как SQL-инъекция или утечка конфиденциальных данных. Другой важной задачей, на которую предлагается обратить внимание, является поиск дефектов связанных с межпротоковым взаимосдействем, в частности поиск взаимных блокировок.

В заключение автор выражает благодарность и большую признательность научному руководителю Белеванцеву А.А. за поддержку, помощь, обсуждение результатов и научное руководство. Автор благодарит директора ИСП РАН Аветисяна А.И. и руководителя отдела компиляторных технологий Гайсаряна С.С. за неоценимый вклад в подготовку данной работы. Также автор благодарит команду разработчиков инструмента SharpChecker, в котором реализовывались все предложенные в данной работе методы и алгоритмы. Автор выражает благодарность Дудиной И.А. и Несову В.С. за конструктивное обсуждение математической составляющей данной работы. Отдельная благодарность моей жене Кошелевой Д.Л. и родителям Кошелеву К.Б и Кошелевой Е.Д. за оказанную поддержку в процессе подготовки данной работы.

Список литературы

1. *Кошелев В. К.* Формализация определения ошибок при статическом символьном выполнении // *Труды Института системного программирования РАН.* — 2016. — Т. 28, № 5. — С. 105–118.

2. Чувствительный к путям поиск дефектов в программах на языке C# на примере разыменования нулевого указателя / Кошелев В.К., Дудина И.А., Игнатьев В.Н., Борзилов А.И. // *Труды Института системного программирования РАН.* — 2015. — Т. 27, № 5. — С. 59–86.

3. *В.К. Кошелев, В.Н. Игнатьев, А.И. Борзилов.* Инфраструктура статического анализа программ на языке C# // *Труды Института системного программирования РАН.* — 2016. — Т. 28, № 1. — С. 21–40.

4. *И.А. Дудина, В.К. Кошелев, А.Е. Бородин.* Поиск ошибок доступа к буферу в программах на языке C/C++ // *Труды Института системного программирования РАН.* — 2016. — Т. 28, № 4. — С. 149–168.

5. *Koshelev V. K., Izbyshev A. O., Dudina I. A.* Interprocedural Taint Analysis for LLVM-bitcode // *Programming and Computer Software.* — 2015. — Vol. 41, no. 4. — Pp. 237–245.

6. *Кошелев В. К.* Статический масштабируемый межпроцедурный анализ помеченных данных // Материалы Международного молодежного научного форума «ЛОМОНОСОВ-2014». — 2014.

7. *Xie Y., Engler D.* Using redundancies to find errors // *ACM SIGSOFT Software Engineering Notes.* — 2002. — Vol. 27, no. 6. — Pp. 51–60.

8. *Dilling I., Dilling T., Aiken A.* Static error detection using semantic inconsistency inference // *ACM SIGPLAN Notices.* — 2007. — Vol. 42, no. 6. — Pp. 435–445.

9. *Johnson S. C.* Lint, a C Program Checker // COMP. SCI. TECH. REP. — 1978. — Pp. 78–1273.

10. *Landi William.* Undecidability of Static Analysis // *ACM Lett. Program. Lang. Syst.* — 1992. — Vol. 1, no. 4. — Pp. 323–337.

11. *Ramalingam G.* The Undecidability of Aliasing. // *ACM Trans. Program. Lang. Syst.* — 1994. — Vol. 16, no. 5. — P. 1467–1471.

12. *Cousot Patrick, Cousot Radhia.* Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints // Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages / ACM. — 1977. — Pp. 238–252.

13. *Cousot Patrick, Cousot Radhia.* Abstract interpretation frameworks // *Journal of logic and computation.* — 1992. — Vol. 2, no. 4. — Pp. 511–547.

14. *Davey B. A., Priestley H. A.* Introduction to lattices and order. — Cambridge university press, 2002.

15. The ASTREE analyzer / P. Cousot, R. Cousot, J. Feret et al. // *Programming Languages and Systems. – Springer Berlin Heidelberg.* — 2005. — Pp. 21–30.

16. Varieties of static analyzers: A comparison with ASTREE / Patrick Cousot, Radhia Cousot, Jerome Feret et al. // *Sixth International Symposium on Theoretical Aspects of Software Engineering.* — 2007. — no. 3. — Pp. 3–20.

17. Why does Astree scale up / Patrick Cousot, Radhia Cousot, Jerome Feret et al. // *Formal Methods in System Design.* — 2009. — Vol. 35, no. 3. — Pp. 229–264.

18. *Мандрыкин МУ, Мутилин ВС, Хорошилов АВ.* Введение в метод CEGAR – уточнение абстракции по контрпримерам // *Труды Института системного программирования РАН.* — 2013. — Vol. 24. — P. 37.

19. *Jhala Ranjit, Majumdar Rupak.* Path slicing // *SIGPLAN Not.* — 2005. — Vol. 40, no. 6. — Pp. 38–47.

20. *Швед П. Е., Мутилин В. С., Мандрыкин М. У.* Опыт развития инструмента статической верификации BLAST // *Программирование.* — 2012. — Т. 3. — С. 24–35.

21. *Beyer Dirk, Keremoglu M. Erkan.* CPAchecker: A Tool for Configurable Software Verification. — 2009. — Vol. SFU-CS-2009-02.

22. *Beyer Dirk, Keremoglu M. Erkan.* CPAchecker: a tool for configurable software verification // Proceedings of the 23rd international conference on Computer aided verification. — CAV'11. — Berlin, Heidelberg: Springer-Verlag, 2011. — Pp. 184–190.

23. SLAM and Static Driver Verifier: technology transfer of formal methods inside Microsoft / Thomas Ball, Byron Cook, Vladimir Levin, Sriram K. Rajamani. — 2004. — Vol. MSR-TR-2004-08. — URL: ftp://ftp.research.microsoft.com/pub/tr/tr-2004-08.pdf.

24. *Ball Thomas, Levin Vladimir, Rajamani Sriram K.* A decade of software model checking with SLAM // *Commun. ACM.* — 2011. — Vol. 54, no. 7. — Pp. 68–76.

25. Efficient evaluation of pointer predicates with Z3 SMT Solver in SLAM2 / T. Ball, E. Bounimova, V. Levin, L. De Moura. — 2010. — March. — Vol. Technical Report MSR-TR-2010-24. — URL: https://www.microsoft.com/en-us/research/publication/efficient-evaluation-of-pointer-predicates-with-z3-smt-solver-in-slam2/.

26. *Novikov Eugene.* One Approach to Aspect-Oriented Programming Implementation for the C programming language // *Proceedings of SYRCoSE.* — 2011. — Vol. 1. — Pp. 74–81.

27. *Мутилин В. С., Новиков Е. М., Хорошилов А. В.* Анализ типовых ошибок в драйверах операционной системы Linux // *Труды Института системного программирования РАН.* — 2012. — Т. 22. — С. 349–374.

28. Обзор инструментов статической верификации Си программ в применении к драйверам устройств операционной системы Linux / М. У. Мандрыкин, В. С. Мутилин, Е. М. Новиков, А. В. Хорошилов // *Труды Института системного программирования РАН.* — 2012. — Т. 22. — С. 293–326.

29. *Мутилин В. С., Мандрыкин М. У.* Интерполяция формул с кванторами в CSIsat на основе инстанцирования // *Труды Института системного программирования РАН.* — 2012. — Т. 22. — С. 327–348.

30. Использование драйверов устройств операционной системы Linux для сравнения инструментов статической верификации / М. У. Мандрыкин, В. С. Мутилин, Е. М. Новиков и др. // *Программирование*. — 2012. — Т. 5. — С. 54–71.

31. *Clarke Edmund, Kroening Daniel, Yorav Karen.* Behavioral Consistency of C and Verilog Programs Using Bounded Model Checking // Proceedings of the 40th Annual Design Automation Conference. — DAC '03. — New York, NY, USA: ACM, 2003. — Pp. 368–371. — URL: http://doi.acm.org/10.1145/775832.775928.

32. *Clarke Edmund, Kroening Daniel, Yorav Karen.* Behavioral Consistency of C and Verilog Programs Using Bounded Model Checking. — 2003. — Vol. Technical Report CMU-CS-03-126.

33. *King J. C.* Symbolic execution and program testing // *Communications of the ACM*. — 1976. — Vol. 19, no. 7. — Pp. 385–394.

34. *Moura L. De, Bjorner N.* Z3: An efficient SMT solver // *Tools and Algorithms for the Construction and Analysis of Systems. – Springer Berlin Heidelberg*. — 2008. — Pp. 337–340.

35. *Merz Florian, Falke Stephan, Sinz Carsten.* LLBMC: Bounded model checking of C and C++ programs using a compiler IR // International Conference on Verified Software: Tools, Theories, Experiments / Springer. — 2012. — Pp. 146–161.

36. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. / Cristian Cadar, Daniel Dunbar, Dawson R Engler et al. // OSDI. — Vol. 8. — 2008. — Pp. 209–224.

37. *Xu Zhongxing, Kremenek Ted, Zhang Jian.* A memory model for static analysis of C programs // International Symposium On Leveraging Applications of Formal Methods, Verification and Validation / Springer. — 2010. — Pp. 535–548.

38. *Bush W. R., Pincus J. D., Sielaff D. J.* A static analyzer for finding dynamic programming errors // *Software-Practice and Experience*. — 2000. — Vol. 30, no. 7. — Pp. 775–802.

39. Extended static checking / David L Detlefs, K Rustan M Leino, Greg Nelson, James B Saxe. — 1998.

40. PLDI 2002: Extended static checking for Java / Cormac Flanagan, K Rustan M Leino, Mark Lillibridge et al. // *ACM Sigplan Notices*. — 2013. — Vol. 48, no. 4S. — Pp. 22–33.

41. *Xie Y., Aiken A.* Scalable error detection using boolean satisfiability // *ACM SIGPLAN Notices*. — 2005. — Vol. 40, no. 1. — Pp. 351–363.

42. *Xie Y.* Static detection of software errors: Ph.D. thesis / Stanford University. — 2006.

43. *Aiken A., Bugrara S., Dillig I. et al.* Precise and Scalable Software Analysis. — 2006. — URL: http://saturn.stanford.edu.

44. An overview of the Saturn project / A. Aiken, S. Bugrara, I. Dillig et al. // *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. — 2007. — Pp. 43–48.

45. *Xie Y., Aiken A.* Context-and path-sensitive memory leak detection // *ACM SIGPLAN Notices*. — 2005. — Vol. 30, no. 5. — Pp. 115–125.

46. *Babic D., Hu A. J.* Calysto // *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on. - IEEE*. — 2008. — Pp. 211–220.

47. *Babić Domagoj.* Exploiting Structure for Scalable Software Verification: Ph.D. thesis / University of British Columbia, Vancouver, Canada. — 2008.

48. DC2: A framework for scalable, scope-bounded software verification / Franjo Ivancic, Gogul Balakrishnan, Aarti Gupta et al. // Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering / IEEE Computer Society. — 2011. — Pp. 133–142.

49. Scalable and scope-bounded software verification in Varvel / Franjo Ivančić, Gogul Balakrishnan, Aarti Gupta et al. // *Automated Software Engineering*. — 2015. — Vol. 22, no. 4. — Pp. 517–559.

50. A few billion lines of code later: using static analysis to find bugs in the real world / Al Bessey, Ken Block, Ben Chelf et al. // *Communications of the ACM*. — 2010. — Vol. 53, no. 2. — Pp. 66–75.

51. *Almossawi A., Lim K., Sinh T.* Analysis tool evaluation: Coverity prevent // *Pittsburgh, PA: Carnegie Mellon University.* — 2006.

52. *Klocwork.* Klocwork: the set of static code analysis tools. — URL: http://www.klocwork.com/.

53. CodeSonar : a source code and binary code analysis tool that performs a whole-program, interprocedural analysis on C, C++, Java, and binary executables. — URL: https://www.grammatech.com/products/codesonar.

54. *Deutsch A.* Static Verification of Dynamic Properties // *ACM SIGAda 2003 Conference.* — 2003.

55. *Emanuelsson P., Nilsson U.* A comparative study of industrial static analysis tools // *Electronic notes in theoretical computer science.* — 2008. — Vol. 217. — Pp. 5–21.

56. *Shiraishi Shinichi, Mohan Veena, Marimuthu Hemalatha.* Test suites for benchmarks of static analysis tools // Software Reliability Engineering Workshops (ISSREW), 2015 IEEE International Symposium on / IEEE. — 2015. — Pp. 12–15.

57. CodeIt.Right : a predefined design, style guidelines and best practices checker. — URL: http://submain.com/products/codeit.right.aspx.

58. FxCop : a free static code analysis tool from Microsoft that checks .NET managed code assemblies. — URL: https://msdn.microsoft.com/ru-ru/library/bb429476(v=vs.80).aspx.

59. StyleCop : an open source static code analysis tool from Microsoft that checks C# code for conformance to StyleCop's recommended coding styles. — URL: https://stylecop.codeplex.com.

60. NDepend : a static analysis tool for .NET managed code. — URL: http://www.ndepend.com/.

61. CodeRush : refactoring and productivity visual studio plugin. — URL: https://www.devexpress.com/products/coderush/.

62. *Logozzo Francesco.* Practical verification for the working programmer with code-contracts and abstract interpretation // International Workshop on Verification, Model Checking, and Abstract Interpretation / Springer. — 2011. — Pp. 19–22.

63. The .NET Compiler Platform (R̈oslyn)̈ provides open-source C# and Visual Basic compilers with rich code analysis APIs. — URL: https://github.com/dotnet/roslyn.

64. PVS-Studio : Static Code Analyzer for C#. — URL: https://www.viva64.com/ru/pvs-studio/.

65. ReSharper : refactoring, productivity and bug detection visual studio plugin. — URL: https://www.jetbrains.com/resharper/.

66. *Аветисян А. И., Бородин А. Е.* Механизмы расширения системы статического анализа Svace детекторами новых видов уязвимостей и критических ошибок // *Труды ИСП РАН.* — 2011. — Т. 21. — С. 39–54.

67. Использование статического анализа для поиска уязвимостей и критических ошибок в исходном коде программ / А. И. Аветисян, А. А. Белеванцев, А. Е. Бородин, В. С. Несов // *Труды ИСП РАН.* — 2011. — Т. 21. — С. 23–38.

68. Статический анализатор Svace для поиска дефектов в исходном коде программ / В. П. Иванников, А. А. Белеванцев, А. Е. Бородин и др. // *Труды ИСП РАН.* — 2014. — Т. 26, № 1. — С. 231–250.

69. Static analyzer Svace for finding defects in a source program code / V. P. Ivannikov, A. A. Belevantsev, A. E. Borodin et al. // *Programming and Computer Software.* — 2014. — Vol. 40, no. 5. — Pp. 265–275.

70. *Бородин А. Е., Белеванцев А. А.* Статический анализатор Svace как коллекция анализаторов разных уровней сложности // *Труды ИСП РАН.* — 2015. — Т. 27, № 6. — С. 111–134.

71. *Borodin Alexey.* Summary Based Static Analysis for Practical Search for Defects in C Programs and Libraries // Software Testing, Verification and Validation Workshops (ICSTW), 2014 IEEE Seventh International Conference on / IEEE. — Clevland: 2014. — Pp. 231–232.

72. *Бородин А. Е.* Статический поиск ошибок повторной блокировки семафора // *Труды ИСП РАН.* — 2014. — Т. 26, № 3. — С. 103–112.

73. *Бородин А Е.* Анализ на основе аннотаций для практического поиска дефектов в программах и библиотеках, написанных на языке Си // Труды XXI Международной научной конференции студентов, аспирантов и молодых учёных «Ломоносов-2014». — Москва: 2014. — С. 100–102.

74. *Бородин А Е.* Статический анализатор Svace как коллекция анализаторов разных уровней сложности // Открытая конференция по компиляторным технологиям. — Москва: 2015.

75. *Игнатьев В Н.* Использование легковесного статического анализа для проверки настраиваемых семантических ограничений языка программирования // *Труды ИСП РАН.* — 2012. — Т. 22. — С. 169–188.

76. *Игнатьев В Н.* Формализация ограничений языков C и C++ для их проверки методами статического анализа // Пятый сборник трудов молодых ученых и сотрудников кафедры Вычислительной Техники ИТМО / Под ред. Т И Алиева. — 2014. — С. 17–20.

77. *Маликов О. Р., Несов В. С.* Автоматический поиск уязвимостей в больших программах. // *Известия ТРТУ, Тематический выпуск «Информационная безопасность».* — 2006. — № 7. — С. 114–120.

78. *Nesov V.* Automatically Finding Bugs in Open Source Programs. // *Electronic Communications of the EASST.* — 2009. — Vol. 20.

79. *Маликов О. Р.* Исследование и разработка методики автоматического обнаружения уязвимостей в исходном коде программ на языке Си: Ph.D. thesis / ИСП РАН. — 2006.

80. *Аветисян А. И.* Современные методы статического и динамического анализа программ для автоматизации процессов повышения качества программного обеспечения. — 2012.

81. *Бородин А. Е.* Межпроцедурный контекстно-чувствительный статический анализ для поиска ошибок в исходном коде программ на языках Си и Си++: Ph.D. thesis / ИСП РАН. — 2006.

82. Компиляторы: принципы, технологии и инструментарий / А Ахо, М Лам, Р Сети, Д Ульман. — 2 изд. — Вильямс, 2008.

83. *Muchnick Steven S.* Advanced Compiler Design and Implementation. — Morgan Kaufmann, 1997. — Vol. 1.

84. *Andersen Henrik Reif.* An introduction to binary decision diagrams // *Lecture notes, available online, IT University of Copenhagen.* — 1997.

85. *Craig William.* Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory // *Journal of Symbolic Logic.* — 1957. — Vol. 22, no. 3. — Pp. 269–285.

86. *Christ Jürgen, Hoenicke Jochen, Nutz Alexander.* SMTInterpol: An interpolating SMT solver // International SPIN Workshop on Model Checking of Software / Springer. — 2012. — Pp. 248–254.

87. *Reps Thomas.* Program analysis via graph reachability // *Information and software technology.* — 1998. — Vol. 40, no. 11. — Pp. 701–726.

88. Highly precise taint analysis for Android applications / Christian Fritz, Steven Arzt, Siegfried Rasthofer et al. // *EC SPRIDE, TU Darmstadt, Tech. Rep.* — 2013. — P. 32.

89. The smt-lib standard: Version 2.0 / Clark Barrett, Aaron Stump, Cesare Tinelli et al. // Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England). — Vol. 13. — 2010. — P. 14.