СОДЕРЖАНИЕ

СПИСОК СОКРАЩЕНИЙ И УСЛОВНЫХ ОБОЗНАЧЕНИЙ

– КА – конечный автомат

– ПО – программное обеспечение

– ICFG – inter-procedural control flow graph

– JAR – Java archive

– JVM – Java virtual machine

– MBT – model based testing

– PBT – property based testing

ТЕРМИНОЛОГИЯ

– Динамический анализ – метод анализа программ, основанный на их реальном выполнении
– Мокирование – замена реальных компонентов программы на искусственные для имитации поведения
– Обратная совместимость – способность работы новой версии программного обеспечения с артефактами старой версии
– Символьное исполнение – техника анализа программ, основанная на трансляции кода в уравнения и их последующем решении
– Статический анализ – метод анализа программного кода без его выполнения
– Фаззинг – метод тестирования, при котором на вход программе подаются сгенерированные данные
– Фреймворк – определенный набор инструментов, определяющий шаблоны их использования
– Identity hash code – уникальное числовое значение, которое определяется для каждого объекта на основе его памяти адреса
– Reflection – механизм изменения и исследования программы во время её выполнения

ВВЕДЕНИЕ

С развитием области анализа программ при решении большого количества задач в этом направлении разработчики все чаще стали сталкиваться с необходимостью использования формальных спецификаций. Формальная спецификация - описание поведения программы на специальном или её исходном языке, включающее в себя такие детали как пред и пост условия вызовов, состояния и переходы между ними, что и делает спецификации идеальным кандидатом для применения в области анализа программного обеспечения (ПО). Например, спецификации не заменимы в символьном исполнении. Там они используются для аппроксимации поведения внешних библиотек или сложных частей программы, тем самым ускоряя анализ или вовсе делая его возможным в определенных местах. Реализацию данного подхода можно увидеть в USVM[1] и UtBot[1]. Еще один пример использования формальных спецификаций - Taint анализ, когда в них отмечаются потенциальные места ввода и утечки информации. Наличие подобных данных позволяет анализаторам сфокусироваться на проверке размеченных мест, представляющих возможные ошибки, тем самым сильно повышая эффективность[2]. Кроме этого, спецификации могут использоваться в тестировании, основанном на модели (MBT) и основанном на свойствах (PBT), в качестве тестового оракула, отвечая на вопросы о корректности состояния ПО и переходов между ними, а также о том, удовлетворяют ли входные и выходные данные для различных методов соответствующим предикатам. Сами формальные спецификации при этом могут быть представлены с

[1] https://github.com/UnitTestBot/usvm

помощью подмножества используемых языков программирования, как в UtBot, или на специальных языках спецификации, например LibSL[3].

При всем этом формальные спецификации в общем случае не поставляются с программными библиотеками или другим ПО, что заставляет задуматься о способах их создания. Полностью ручное составление спецификаций это довольно монотонная работа, при этом требующая от человека высокой квалификации в области разработки и анализа программ. Однако составление спецификаций можно частично автоматизировать.

Часть формальной спецификации можно найти в исходном коде библиотек или ПО. Если приводить в пример языки в парадигме объектно-ориентированного программирования, то это классы и интерфейсы программы, их поля и сигнатуры методов. Другую же часть спецификации, описывающую поведение программы, а именно состояния и переходы между ними, можно попытаться извлечь из реальных примеров использования.

Именно автоматизации получения поведенческих моделей библиотек посвящена данная работа. В рамках её выполнения будет разработан комплексный подход для получения моделей и для его апробации реализована утилита, позволяющая автоматически получать общедоступные Java проекты и с помощью методов статического и динамического анализа извлекать из них сценарии работы определенной библиотеки с целью последующего восстановления поведенческой модели в виде конечного автомата (КА). На данный момент комплексных автоматических решений для подобных задач нет, однако имеются работы в области восстановления КА из трасс и описаны способы получения трасс. Основные задачи, решаемые в этой работе, это автоматиза-

ция, интеграция и применение на практике существующих наработок в области восстановления поведенческой модели библиотек.

Важно сказать, что предполагается ручная обработка полученных с помощью реализованной утилиты автоматов. Необходимость этого является следствием фундаментальных ограничений алгоритмов восстановления КА. В дополнение к этому предполагается использование пользовательских репозиториев, которые не гарантируют корректного применения библиотек. Также свои ограничения накладывает статический и динамический анализ. Использование points-to анализа в статическом подходе не позволяет точно различать объекты, методы которых вызываются, что влияет на корректность получаемых трасс. Также при использовании статического анализа, по крайней мере в рамках данной работы, не будут предприниматься попытки обработать многопоточную работу и получить данные о состоянии программы в моменты вызовов. Что касается динамического анализа, то здесь на результат могут содержательность имеющихся тестов, точки входа в программу и необходимое для запуска окружение.

В первом разделе представлен обзор существующих решений, связанных с задачами поиска проектов, извлечением из них трасс вызовов и восстановлением модели. Также в данном разделе будет уделено внимание предшествующей работе по данной теме. На основе первого раздела сделан выбор в пользу определенных подходов и решений, составляющих общий подход и используемых в реализации инструмента. Во втором разделе формулируются требования к создаваемому инструменту и описываются пути решения каждой из задач, стоящих на пути реализации. Третий раздел посвящен разработке подхода. В нем будет подробно описана задуманная схема работы, способы извлече-

ния трасс и их восстановления, детали работы с репозиториями. Четвертый раздел содержит описание реализации инструмента. Пятый раздел посвящен тестированию полученной утилиты. Тестирование заключается в сравнении получаемых автоматов с несколькими заготовленными эталонами, а также демонстрацией получаемых КА для некоторого набора библиотек. Помимо этого, будет проанализировано количество успешно автоматически собранных проектов и полученных различными методами трасс. В заключении проведен анализ полученных результатов, отмечены преимущества и недостатки предложенного подхода, а также рассмотрены пути развития.

1 АНАЛИЗ РЕШЕНИЙ В ОБЛАСТИ ВОССТАНОВЛЕНИЯ ПОВЕДЕНЧЕСКИХ МОДЕЛЕЙ

При изучении предметной области было выявлено, что на текущий момент нет исследований и инструментов, целью которых являются полностью автоматический процесс получения спецификаций, начиная от получения проектов, использующих заданную библиотеку, и заканчивая извлечения из нее поведенческой модели. Тем не менее, в данной области достаточное количество работ, сосредоточенных на методах извлечения трасс и последующего восстановления модели библиотек, подразумевающих применение подходов к подготовленным для анализа программам. В данном разделе рассмотрим и сравним существующие способы извлечения трасс из программ и алгоритмы восстановления поведенческих моделей в виде КА.

1.1 Критерии сравнительного анализа

Выделим определенные критерии, на которые будем обращать внимание при обзоре работ.

1.1.1 Метод извлечения трасс

Глобально методы можно поделить на статические и динамические. Первые подразумевают анализ исходного кода или байт-кода программы без его запуска. Динамические методы наоборот, предполагают запуск анализируемой программы. Статические методы уступают в точности, однако позволяют покрыть все возможные пути исполнения программы. Это позволяет находить ошибки в тех участках кода, которые не покрытых тестами и до которых исполнение не доходит при штатной работе программы. Динамические методы, в

свою очередь, за счет реального исполнения обеспечивают точность получаемых результатов, но с их помощью сложно получить все возможные состояния программы. Для восстановления поведенческой модели мы заинтересованы в получении как можно большего количества трасс, чему сопутствует использование статических методов, но в тоже время ошибки в трассах неизбежно приведут к ошибкам в модели. Таким образом, недостатки одного метода являются преимуществом другого и наоборот. В работах нас интересует, как авторы реализовали преимущества и нивелировали недостатки выбранных методов.

1.1.2 Алгоритм восстановления модели

Алгоритм восстановления непосредственно влияет на качество получаемых моделей. При этом он определяет, какие данные нам необходимо извлечь из программы. Например, базовый алгоритм k-tail[4] требует на вход исключительно последовательности вызовов и параметра k, определяющего длину соединяемых цепочек. Другой алгоритм, gk-tail[5], дополнительно требует на вход значения аргументов.

Также существуют различные алгоритмы, основанные на использовании инвариантов или состоянии программы в моменте вызова библиотеки. В рамках обзора важно обратить внимание, каких дополнительных усилий требует применение сложных алгоритмов восстановления, какие ограничения это накладывает и какой дает прирост в точности и полноте получаемых автоматов.

Перед тем, как перейти к обзору современных работ, следует уделить особое внимание алгоритмам k-tail[4] и gk-tail[5], на которых основано большинство современных методов восстановления автоматов из трасс. K-tail принимает на вход последовательность вызовов, полагая что трасса - это КА, где

переходами являются вызовы. Для каждого состояния рассматриваются хвосты длинной k (обычно равной один или два) и если эти хвосты эквивалентны, то они сливаются. Безусловным плюсом данного алгоритма является высокая точность при простоте применения - алгоритм не может породить модель, разрешающую несуществующие трассы, а также не требует дополнительных обработок входных данных. Но не смотря на то, что получаемая модель описывает корректные последовательности, получаемые состояния КА не отражают реальные и являются сильной аппроксимацией сверху реальной модели.

Gk-tail основан на k-tail, однако помимо самих вызовов, также учитывает информацию об аргументах вызовов и контекстных переменных. Сначала трассы, состоящие из одинаковых вызовов объединяются вместе с данными, накапливая множество возможных значений для переменных. Далее с помощью Daikon[6] (используется в оригинальной статье, возможно использование других подобных инструментов) выполняется вывод инвариантов для каждого перехода, основанный на накопленных данных. Затем применяется алгоритм k-tail, однако для потенциально сливаемых хвостов происходит проверка инвариантов на конфликты. Данный алгоритм более трудозатратный ввиду необходимости получения информации о значении аргументов и контекстных переменных, а также обязательного вывода инвариантов. Но взамен мы получаем более осмысленное деление на состояния, чем при использовании K-tail.

1.1.3 Применимость к реальным проектам и возможность автоматизации

Определенные подходы могут показывать отличные результаты и иметь минимальные недостатки, но при этом иногда они совершенно не применимы к реальным проектам, что обусловлено либо новизной, либо фундаментальными

ограничениями подхода. Также применение некоторых методов, в частности основанных на динамическом анализе, требует определенной ручной работы, например связанной с подготовкой анализируемой программы и её окружения. Это может сильно влиять на массовость применения подхода и его автоматизацию.

1.1.4 Доступ к исходному коду

Если авторы предоставляют доступ к инструментам, это позволяет убедиться в результатах проведенных экспериментов. И что не менее важно, появляется возможность применять и развивать разработанный в рамках исследований подход и инструмент.

1.2 Обзор работ по извлечению спецификаций

В работе «Static Specification Mining Using Automata-Based Abstractions»[7] авторы статически собирают трассы в виде последовательностей объектов одного типа, используя абстрактную интерпретацию[8]. Для получения последовательностей вызовов над конкретным экземпляром объекта в исследовании используется points-to анализ на основе алгоритма Андерсена[9] (не чувствительный к потоку) и чувствительный к потоку access-paths анализ. При этом авторы не объединяют результаты применения анализов, а используют их в зависимости от потребности в максимально подробных и избыточных трассах (flow-insensitive) или точных и ограниченных (flow-sensitive). Для восстановления поведенческой модели авторы используют собственный подход, основанный на эвристических правилах слияния состояний. Предложенные алгоритмы выглядят интересно, однако сложно оценить их точность, так как в исследовании приводится сравнение результатов вари-

аций описанных алгоритмов между собой, хотя было бы уместно провести сравнение с классическим алгоритмом k-tail. В ограничениях подхода авторы описывают невозможность его применения для анализа проектов состоящих из десяток тысяч строк кода. Это ожидаемо, поскольку flow-sensitive подходы сталкиваются с проблемой взрыва состояний и применение access-paths анализа к реальной программе требует большого количества памяти даже при минимальной глубине анализа[10]. Авторы делают вывод, что получаемые поведенчески модели достоверно описывают поведение библиотек, однако являются сильной аппроксимацией сверху истиной модели и содержат множество лишних состояний и переходов. Однако предполагается, что выявление чистых функций в исходном коде библиотеки позволит избежать появления избыточных состояний, так как на этапе восстановления будет известно, что определенные вызовы не изменяют состояния программы, а значит конечную модель можно упростить. В статье явно упоминается разработанный инструментарий для проведения анализа, однако ссылки на них не приводятся.

Авторы статьи «Automatic mining of specifications from invocation traces and method invariants»[11] подробно рассмотрели и сравнили четыре алгоритма восстановления моделей из трасс. При этом был рассмотрел базовый алгоритм k-tail, предложены улучшения для алгоритма Contractor[12], основанного на получении КА из инвариантов, а также разработаны новые подходы: SEKT и TEMI, заключающиеся в извлечении поведенческой модели из трасс, усиленных инвариантами, и инвариантов, усиленных трассами, соответственно. В рамках исследовании авторы получали трассы и инварианты с помощью инструмента динамического анализа Daikon[6]. Daikon очень мощный и

полезный инструмент, однако его применение сложно автоматизировать для сторонних проектов, так как даже чтобы получить полные трассы и полезные инварианты из собственного целевого проекта, нужно проделать определенную нетривиальную работу. В статье очень большое внимание уделено сравнению методов восстановления КА. Авторы ввели метрики precision и recall, где под precision понимается доля трасс, сгенерированных по восстановленной модели и подходящих под эталонную модель, а recall определяется как доля сгенерированных трасс по эталонной модели, не противоречащих восстановленной. В результате все методы, включая k-tail, показали precision близкий к 100 процентам для девяти эталонных моделей библиотек. Что касается recall, k-tail и SEKT показали результат от 20% до 60%, имя примерно одинаковые значения в рамках конкретной библиотеки. TEMI и Contractor++ показали лучшие результаты, достигая значений 100% для некоторых библиотек, однако также сохранялся большой разброс и худшие результаты были на уровне 40%. Стоит заметить, что не смотря на очевидное преимущество более сложных алгоритмов, k-tail, требующий минимальные входные данные, показывает конкурентноспособный результат. В данном исследовании авторы не делятся реализацией алгоритмов, хоть и детально описывают принцип их работы.

Интересный метод восстановления, а также его реализацию[2] в открытом доступе предлагают авторы статьи «Inferring Extended Finite State Machine models from software executions»[13]. Авторы развивают идею алгоритма gk-tail и предлагают использовать информацию о значении аргументов в анализируемых вызовов. Однако новизна заключается в том, что для поиска конфликтов

[2]https://github.com/neilwalkinshaw/mintframework

слияния применяются обучаемые классификаторы. Под конфликтами понимаются слияния таких трасс, где из одного и того же состояния при одних и тех же вызовах осуществляется переход в отличные друг от друга состояния. Это говорит о том, что на самом деле начальное состояние было не одно и то же. В gk-tail поиск конфликтов между инвариантами происходит локально для отдельных участков трасс длиной k, из за чего можно объединить состояния, где позже возникает конфликт. В предлагаемом подходе классификаторы используют трассы как источник данных для обучения, что позволяет осуществлять поиск конфликтов из всей совокупности данных. Также классификаторы избавляют от необходимости использовать тяжеловесные утилиты по типу Daikon, неявно выполняя задачу вывода инвариантов. Благодаря одновременному учету всех трасс обеспечивается высокий уровень обобщенности получаемой поведенческой модели. Однако при этом на пользователя ложится задача подбора алгоритма для классификации данных, поскольку разные алгоритмы могут показывать разный результат в зависимости от входных данных. Авторы в своем исследовании приводят сравнение алгоритмов, а также предоставляют в реализованном инструменте возможность удобно его менять. Что касается получения трасс, в исследовании используются трассы полученные из двух проектов с помощью Daikon. Отдельно стоит поблагодарить авторов за реализацию алгоритмов k-tails и gk-tails в предоставляемом инструменте.

Еще один выделяющийся подход реализован в инструментах Tautoko[14] для генерации тестов и ADABU[15], представленных в соответствующих исследованиях. ADABU решает задачу получения трасс и состояний программы на основе инструментации и реализует предложенный в исследовании метод

восстановления поведенческой модели. Общий подход заключается в отслеживании состояния программы до и после вызова библиотеки. После сбора трасс, собранные состояния классифицируются по определенным правилам, тем самым образуя состояния КА. Tautoko же является генератором тестов, позволяющим получить ранее не обнаруженные варианты поведения библиотеки. Авторы предлагают реализованный подход как решение проблемы ограниченного набора тестов при использовании инструментов по типу Daikon. К сожалению, в исследованиях не представлены сравнения с существующими алгоритмами восстановления и инструментами генерации тестов. Однако представленных результатов достаточно, чтобы убедиться в работоспособности предложенных походов, а наличие инструментов в открытом доступе[3] делает полученные результаты очень полезными. Но важно отметить, что инструменты реализованы в 2012 году и не получали никаких обновлений и они скорее всего не актуальны для анализа современных версий Java. Также данный подход имеет ограничение в виде необходимости работы над конкретными проектами для извлечения трасс, так как требуется плотное взаимодействие с исполняемыми файлами анализируемого ПО.

1.3 Результаты анализа

Все описанные работы предлагают работоспособные решения, подтверждаемые авторами в рамках проведенных экспериментов. Однако нигде не уделяется внимании автоматизации решения – зачастую авторы извлекают трассы из одного и того же проекта (даже в рамках разных исследований разных авторов). Причиной этого является применение чисто динамических

[3]https://www.st.cs.uni-saarland.de/models/

подходов для получения информации о состоянии программы в момент вызовов, что принуждает к плотному взаимодействию с бинарными файлами программы и её необходимым окружением, а это довольно трудозатратно. Одна из описанных работ[7] использует подход на основе статического анализа и имеет потенциал для автоматизации, однако авторы применяют flow-sensitive алгоритм для анализа псевдонимов, что делает подход неприменимым для реальных проектов. Краткий итог сравнения представлен в Таблица 1.

Таблица 1 — Анализ работ

Название	Извлечение трасс	Восстановление модели	Исх. код	Ограничения
Static Spec. Mining[7]	Абстрактная интерпретация	трассы	Нет	Невозможен анализ реальных проектов
SEKT/TEMI[11]	Daikon	трассы, состояния	Нет	Частный подход к проектам
MINT[13]	Daikon	трассы, состояния	Да	Частный подход к проектам
Tautoko[14]/ ADABU[15]	Собственная инструментация и генерация тестов	трассы, состояния	Да, устаревший	Частный подход к проектам
Gk-tail	Daikon	трассы, состояния	Да, в MINT	Частный подход к проектам
K-tail	-	трассы	Да, в MINT	Получаемая модель далека от реальной

Все это наводит на мысль о необходимости создания комплексного автоматизированного решения для извлечения трасс и поведенческих моделей библиотек. Безусловно, для начала автоматизация потребует некоторых уступ-

ков в требованиях к качеству получаемых автоматов и решение специфичных проблем. Однако это положит начало развитию подобных автоматизированных методов и, возможно, позволит использовать извлечение автоматов в реальной жизни с меньшими усилиями.

2 ЗАДАЧА ИЗВЛЕЧЕНИЯ ПОВЕДЕНЧЕСКИХ МОДЕЛЕЙ И АНАЛИЗ ПУТЕЙ ЕЁ РЕШЕНИЯ

Задача извлечения поведенческих моделей библиотек состоит из нескольких составляющих:

– Поиск проектов, использующих заданную библиотеку

– Подготовка проектов к анализу

– Извлечение трасс и состояний

– Восстановление автоматов из трасс

Каждая из этих задач требует отдельного внимания, а также определенного уровня согласованности с остальными. Из анализа существующих решений в предыдущем разделе видно, что методы извлечения трасс могут зависеть от требований к масштабированию подхода и от самих используемых для анализа программ (исходный код, исполняемые файлы, наличие тестов, необходимое окружение). Метод извлечения трасс в свою очередь определяет применимые для восстановления моделей методы. Поэтому важно комплексно подходить к выбору путей решения каждой из указанных составляющих.

Выбор путей решения в рамках данной работы будет основан на гипотезе о том, что в текущем состоянии предметной области получение части спецификаций, связанных с поведенческими моделями библиотек, является очень трудозатратным и требует автоматизации. Даже при использовании инструментов, предоставленных авторами статей, пользователю необходимо самостоятельно найти подходящий проект. Затем, в случае применения динамических методов, выполнить инструментирование, убедиться в наличии тестов или заняться их генерацией и оценить содержательность получаемых трасс. Если говорить про

статические подходы, то готовых решений обнаружено не было. Пользователь будет вынужден самостоятельно разрабатывать анализы на основе фреймворков статического анализа или изучать доступные символьные машины с целью применения их для сбора трасс. Только после успешного решения подобных задач и связанных с ними проблем, можно перейти к самому восстановлению трасс. К счастью, забегая вперед, MINT[13] предлагает действительно удобный и рабочий модульный инструмент для применения собственного алгоритма восстановления, а также k-tail и gk-tail. Тем не менее, для получения КА на данный момент требуется преодолеть ряд сложных и неочевидных задач, требующих определенного погружения в область анализа ПО.

2.1 Поиск проектов

Для апробации подхода остановимся на Java проектах по нескольким причинам:

- Большинство исследований в области восстановления поведенческих моделей библиотек используют проекты на Java в качестве целевых, что позволяет перенимать практический опыт данных работ, а также их реализацию
- Для языка Java разработано большое количество библиотек, в том числе для статического и динамического анализа
- Не смотря на определенные недостатки, Java по прежнему является популярным языком, на котором реализовано множество проектов, число которых продолжает расти

В качестве источников проекта предлагается использовать GitHub и Maven Central Repository. Рассмотрим данные варианты подробнее.

GitHub является крупнейшим сервисом хранения программных репозиториев и на текущий момент содержит более 420 миллионов репозиториев[16]. При этом GitHub предоставляет возможности поиска по коду и фильтрации, реализованные в рамках API. При этом репозитории могут содержать артефакты программ, а именно JAR (Java Archive) файлы. Их наличие позволит избежать этапа сборки, тем самым ускоряя процесс анализа проекта и потенциально повышая число успешно обработанных проектов, так как для статического анализа необходимы бинарные файлы.

Не смотря на большое количество проектов на GitHub, среди них могут быть как серьезные рабочие проекты, так и множество неработоспособных программ. Поэтому в качестве альтернативы есть возможность использовать Maven Central Repository, где среднее качество проектов должно быть значительно выше. Репозиторий Maven предлагает поиск библиотек по API. Для каждого пакета можно получить список зависимых от него, что и требуется для поиска примеров использования.

2.2 Подготовка проектов к анализу

Для применения статических подходов необходимы скомпилированные программы, при этом наличие зависимостей не влияет на анализ. Для динамических же необходим еще и исходный код, поскольку зависимости и тесты редко упаковываются в JAR файл и находятся непосредственно с самим кодом. Из этого возникает необходимость выполнять сборку проектов перед извлечением трасс. Опираясь на то, что Java Virtual Machine (JVM) решает вопрос зависимости от платформы, а Java гарантирует обратную совместимость, можно предположить, что при существовании зависимостей, указанных в проекте,

и совпадении версий сборщиков Maven или Gradle, будет возможно успешно собрать определенную долю проектов.

2.3 Извлечение трасс

Для извлечения трасс возможно применение статического и динамического анализа. В рамках данной работы предполагается, что с помощью динамического подхода в автоматическом режиме будет получено меньше трасс, чем с помощью статического. Более того, уже имеющиеся тесты не гарантируют целостность получаемых трасс ввиду возможного мокирования и искусственного создания объектов с помощью reflection, что будет необходимо учесть при проектировании инструмента.

Предлагаемый статический подход заключается в анализе межпроцедурного графа потока управления (в дальнейшем ICFG), поиску в нем вызовов искомой библиотеки и последующему применению Points-to анализа для определения объектов вызовов и формирования трасс. Предполагается использование анализа на основе алгоритма Андерсена[9] – для возможности применения такого подхода к реальным большим проектам мы вынуждены использовать не чувствительные к контексту или потоку алгоритмы. Хотя худшая сложность алгоритма Андерсена кубическая, что может стать проблемой в межпроцедурном анализе, в реальных программах она оказывается близкой к квадратичной[17].

Что касается динамического подхода, в рамках данной работы мы остановимся на сборе исключительно последовательностей вызовов, игнорируя значения аргументов и состояние программы. Сделано это с целью ограничить сложность работы, так как первоочередной целью является апробация идеи автоматизированного подхода к извлечению поведенческих моделей. Сбор информации должен осуществляться через инструментацию и запуск тестов.

Также предлагается воспользоваться методом фаззинга входных точек в программу. Он позволит обеспечить лучшую целостность получаемых трасс в случае успешного запуска и корректного выбора точек входа. Важно заметить, что выполнять запуск случайных сторонних программ небезопасно, поэтому предполагается, что использование динамических подходов должно производиться на специально выделенной для этого или виртуальной машине.

2.4 Восстановление поведенческой модели

Так как в рамках данной работы мы оперируем трассами, состоящими исключительно из последовательностей вызовов, то это сильно ограничивает выбор алгоритма восстановления. По этой причине предлагается использовать алгоритм k-tail. Тем не менее, алгоритм обладает рядом преимуществ:

- Он не уступает в точности более сложным алгоритмам, то есть сгенерированные по получаемому КА трассы будут корректны
- Простота алгоритма обеспечивает сравнительно высокую скорость работы, так как нет необходимости выводить инварианты и искать конфликты между сливаемыми состояниями
- Благодаря прозрачному принципу восстановления, интерпретация получаемых КА будет проще

Для апробации общего подхода будем полагать, что возможностей данного алгоритма достаточно.

2.5 Общий подход

Из предложенных способов решения задач складывается общий подход. Мы должны получать проекты с GitHub и репозитория Maven, использующие заданную библиотеку. Далее необходимо подготовить проекты к извлечению

трасс, выполнив компиляцию исходного кода и тестов. Затем необходимо извлечь трассы с применением описанных выше подходов и подать полученные результаты на вход алгоритму восстановления поведенческой модели в виде КА. Каждый из шагов имеет множество сопутствующих проблем и будет подробно рассмотрен в следующем разделе, посвященном проектированию инструмента.

3 ПРОЕКТИРОВАНИЕ

Перед реализацией предложенного подхода по получению поведенческих моделей библиотек необходимо сформулировать схему работы основных его составляющих и их взаимодействия друг с другом. В данном разделе рассмотрены общая схема работы инструмента, получение и подготовка проектов, извлечение трасс и восстановление из них поведенческих моделей.

3.1 Общая схема работы

На Рис. 1 в общих чертах представлен полный цикл работы разрабатываемого инструмента. Предполагается возможность выбора режима работы, что позволит пропускать определенные этапы и, например, восстанавливать КА из заранее накопленных данных или извлекать трассы из уже подготовленных проектов, что не отражено на схеме. Цифрами обозначены основные части инструмента, которые будут рассмотрены в данной главе:

1) Получение и подготовка проектов
2) Извлечение трасс
3) Восстановление поведенческих моделей

Общая последовательность действий заключается в том, что изначально пользователь конфигурирует инструмент – задает целевую библиотеку, определяет параметры используемых алгоритмов, указывает пути к требуемому окружению и выбирает режим работы. Далее итеративно выполняется поиск проектов, их подготовка и извлечение трасс с последующим сохранением. После анализа заданного количества проектов или каких-либо других установленных условий остановки выполняется обработка полученных трасс и

восстановление КА. На более детальном уровне каждый шаг разобран в соответствующих подразделах.

Рисунок 1 — Общая схема работы

3.2 Получение и подготовка проектов

В качестве источника проектов предлагается использовать GitHub и Maven Central Repository. Каждый из них имеет свои механизмы поиска и получения проектов, поэтому рассмотрим их раздельно. Дополнительно должна быть предусмотрена возможность использования заранее подготовленных проектов. Также для анализа необходимы JAR файлы проектов и их зависимости, поэтому необходимо реализовать их автоматическую сборку.

3.2.1 GitHub

Для GitHub мы вынуждены использовать поиск по коду, так как других вариантов найти проекты, использующие заданную библиотеку, отсутствуют.

Пользователь может указать в качестве исходного кода импорт основного пакета библиотеки или момент её подключения в build.gradle или pom файле. При этом репозитории помимо исходного кода могут содержать другую полезную информацию:

– Тесты. Они необходимы для получения трасс с помощью динамического анализа
– JAR файлы. Наличие бинарного файла увеличивает вероятность того, что проект рабочий, а также избавляет от необходимости выполнять его компиляцию для статического извлечения трасс. В случае применения динамических подходов необходимо компилировать программу для получения её зависимостей
– Конкретные версии программы. При выпуске версии фиксируется коммит, определяющий состояние исходного кода на тот момент. Это также повышает вероятность того, что проект рабочий

При реализации получения проектов должна быть предусмотрена возможность сбора всей полезной информации и фильтрация по ней, чтобы обеспечить возможность работы только с репозиториями, содержащими тесты или скомпилированные программы. Также необходимо отслеживать обработанные репозитории, так как в результатах поиска один проект может содержаться несколько раз.

Для поиска и получения проектов GitHub предоставляет бесплатное API. В случае отсутствия артефактов, получение проектов может быть реализовано через git.

3.2.2 Maven Central

Maven Central Repository содержит информацию об общедоступных библиотеках, а также об их зависимостях и зависящих от них проектов. Таким образом, есть возможность узнать где используется заданная библиотека – это нас и интересует. Для работы с Maven Repository пользователь должен указать библиотеку и её версию.

Получение проектов предлагается выполнять с помощью Gradle, так как это позволит автоматически собрать её и получить зависимости. Получение исходного кода, хранящегося в Maven Central, не имеет смысла, так как он не содержит тестов. На странице библиотек в Maven Central иногда расположены ссылки на GitHub репозитории, однако их нельзя получить с помощью API, а автоматическое получение и парсинг страниц может привести к блокировке на ресурсе.

Для получения информации о зависящих библиотек от заданной также не предусмотрен общий API. Однако после определенного исследования было обнаружено, что существует некий внутренний API, который реализует данную возможность. Чтобы убедиться в возможности использования данного API был сделан запрос на почту по контактам Maven, где сообщили, что проблемы могут возникнуть только в случае попытки массовой выгрузки всего Maven Central, а это не входит в план работы.

3.2.3 Сборка

В рамках работы будут рассматриваться только проекты, использующие Gradle и Maven в качестве системы сборки, так как они используются чаще всего и относительно легко поддаются автоматизации.

Для работы с Gradle существует Gradle Tooling API, позволяющий автоматически получать необходимую версию сборщика и программно вызывать необходимые задачи. При этом остается возможность использовать конкретную версию для всех проектов или использовать предустановленный Gradle. Maven не имеет подобного API, в связи с чем необходима предустановленная система сборки.

Java гарантирует обратную совместимость со старыми версиями, а JVM решает большинство проблем зависимости от платформы. Благодаря этому можно рассчитывать на успешную сборку определенной доли программ. Тем не менее, есть несколько причин, по которой не все проекты могут быть собраны:

– Проект может использовать специфические зависимости, получение и сборка которых выполняется вручную или иными способами
– Проект может использовать зависимости, которые больше не доступны из заданных репозиториев библиотек

Решение подобных проблем представляет отдельную большую задачу, позволяя увеличить процент собираемых программ, но в рамках данной работы она не будет рассматриваться. Мы вынуждены принять тот факт, что не все программы будут собраны.

3.3 Извлечение трасс

Для извлечение трасс предлагается использовать статический и динамический подходы. Каждый из них имеет сложную схему работы и определенные детали, что требует тщательной разработки методов извлечения.

3.3.1 Статический подход

Примерный алгоритм извлечения трасс с применением статического анализа представлен на Рис. 2. Общий подход заключается в том, что строится межпроцедурный граф потока управления для программы. Также необходимо получить набор точек входа в программу. Из каждой такой точки выполняется обход ICFG, в рамках которого выполняется обнаружение и сбор вызовов заданной библиотеки. Для каждого вызова определяется объект, над которым он выполняется, и с помощью points-to анализа формируются трассы для каждого объекта. Полученные трассы сохраняются для последующего восстановления поведенческой модели.

Что касается получения точек входа, предлагается предоставить пользователю возможность создавать определенные фильтры для методов, содержащие следующие параметры:

- Аннотации метода
- Аннотации класса
- Регулярное выражение для названия метода
- Регулярное выражение для названия класса
- Вид метода (конструктор, публичный метод, блок статической инициализации)
- Типы аргументов метода

Подобный набор должен обеспечить возможность поиска специфических точек входа. Как пример, это могут быть методы, отвечающие за обработку запросов по URL путям в различных фреймворках для создания веб-приложений.

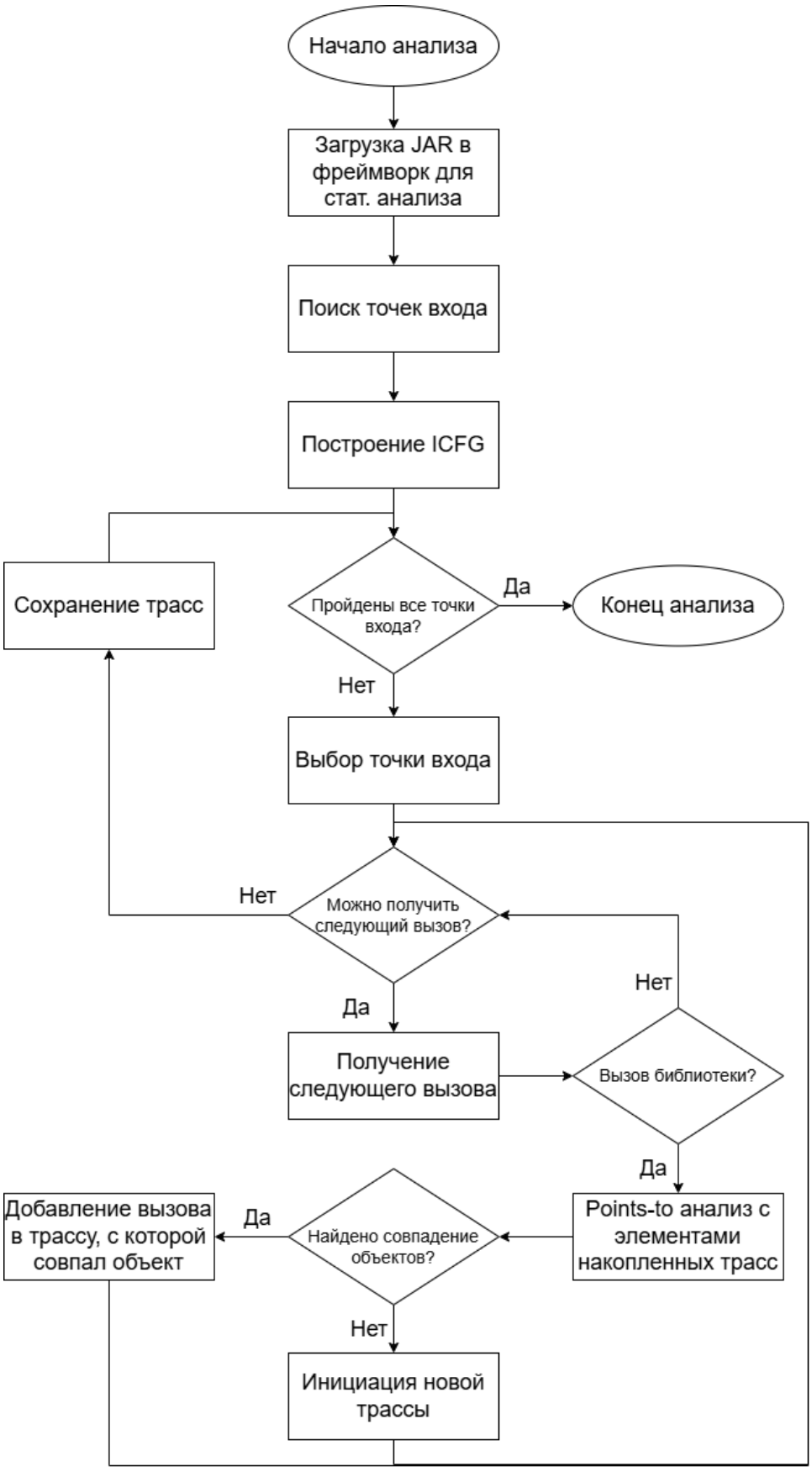

Рисунок 2 — Схема извлечения трасс с помощью статического анализа

Необходимость выполнения points-to анализа обусловлена тем, что общая последовательность нерепрезентативна и не содержит в себе реальных переходов и состояний библиотеки. Рассмотрим простой пример на Листинг 1.

```
1 FileOutputStream dest = new FileOutputStream(file);
2 ZipOutputStream zip1 = new ZipOutputStream(new BufferedOutputStream(dest));
3 ZipOutputStream zip2 = new ZipOutputStream(new BufferedOutputStream(dest));
4 zip1.write(1);
5 zip1.close();
6 zip2.write(1);
7 zip2.close();
```

Листинг 1 — Пример последовательностей вызовов

Извлекаемая трасса при игнорировании объектов будет выглядеть примерно как на Листинг 2.

```
1 1. init
2 2. init
3 3. write
4 4. close
5 5. write
6 6. close
```

Листинг 2 — Пример некорректной трассы

При восстановлении поведенческой модели окажется, что мы допускаем запись после того, как закрыли ZipOutputStream. Очевидно, такая модель не соответствует действительности. Применив points-to анализ мы рассчитываем получить две идентичные трассы как на Листинг 3.

```
1 1. init
2 2. write
3 3. close
```

Листинг 3 — Пример правильной трассы

Предполагается применять points-to анализ на основе алгоритма Андерсена, так как это один самых точных алгоритмов, которые могут быть применимы в для всей программы межпроцедурно. Конечно, существуют гораздо более точные алгоритмы, однако ввиду экспоненциальной сложности их использо-

вание в таких условиях невозможно. В качестве инструмента для статического анализа выбран Soot[4], так как он предоставляет возможности для построения ICFG, конкретные реализации points-to анализа, а также предлагает средства для инструментирования программ, что будет необходимо для динамического подхода по извлечению трасс.

3.3.2 Динамических подход

Извлечение трасс с помощью динамического анализа заключается в предварительной инструментации на уровне байт-кода и последующем запуске программы. Инструментация может быть выполнена с помощью Soot и заключается во вставке специального кода после каждого вызова библиотеки, который позволяет получить информацию о вызове и объекте. Для того, чтобы точно различать экземпляры объектов между собой, может быть использован identity hash code.

Чтобы получить трассы после инструментации, программу необходимо запустить. Для этого предлагается использовать тесты при их наличии и фаззинг входных точек в программу. Входные точки предлагается получать тем же образом, что и при статическом анализе.

В качестве инструмента для фаззинга предлагается использовать Jazzer[5]. Он поставляется в виде собранного JAR файла и имеет режим автоматического фаззинга для указанных точек входа без предварительной подготовки целевого проекта. Для генерации данных Jazzer создает случайные массивы байт, интерпретируемые как необходимый объект и в последствии их мутирует для увеличения покрытия кода.

[4]https://github.com/soot-oss/soot

[5]https://github.com/CodeIntelligenceTesting/jazzer

Общая схема динамического извлечения трасс представлена на Рис. 3.

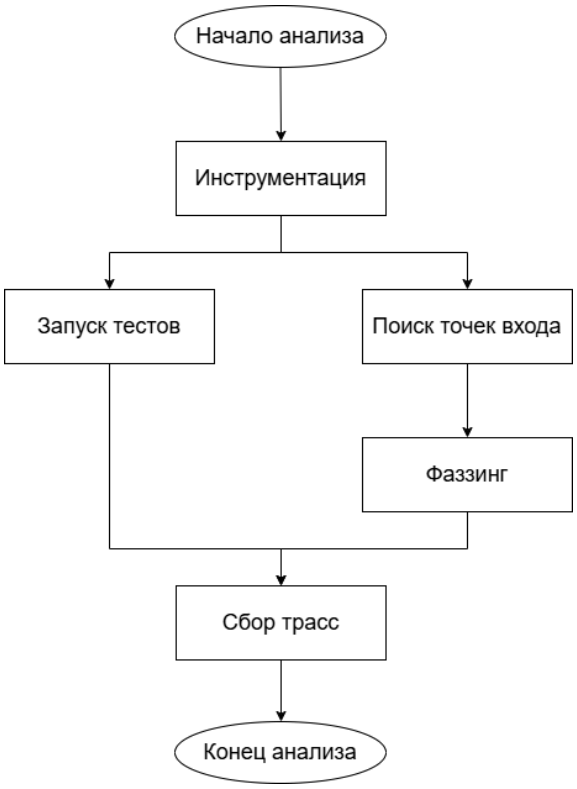

Рисунок 3 — Схема извлечения трасс с помощью динамического анализа

3.4 Восстановление поведенческих моделей

Из накопленных трасс необходимо восстановить поведенческую модель в виде КА. Для этого необходимо обработать полученные трассы и сформировать входные данные для алгоритма k-tail. Что касается обработки, то ввиду использования двух подходов для извлечения трасс, мы можем реализовать несколько режимов работы:

– Для восстановления КА полученные разными способами трассы объединяются в единое множество
– Для каждого способа получения трасс строятся соответствующие КА
– Динамические трассы являются верификатором для статических, подтверждая их в случае совпадения динамической трассы с подпоследовательностью статической

Можно попытаться уменьшить количество некорректных трасс. Трассы могут повторяться между собой, и если считать количество таких повторений для каждой трассы, мы получим частотное распределение. Чтобы исключить погрешности, можно воспользоваться фильтрацией по абсолютному или относительному порогу.

Получаемый в результате работы алгоритма КА может содержать ложные конечные состояния из-за слияний. Для правильного определения состояния необходимо их обойти и обозначить как конечные лишь те, из которых отсутствуют какие либо переходы. После этого конечные состояния следует объединить, что приведет к сокращению состояний КА без потери информации о переходах.

Вместо собственной реализации алгоритма k-tail предлагается использовать фреймворк MINT, рассмотренный в рамках обзора существующих работ в области восстановления поведенческих моделей. Помимо этого, MINT содержит реализации алгоритмов, использующих информацию о переменных, что будет полезно при развитии инструмента.

Получаемые с помощью MINT КА выводятся в dot формате. Для более универсального представления, предлагается транслировать их в JSON.

3.5 Результат проектирования

В результате рассмотрены основные задачи и способы их решения, включая используемые инструменты и различные детали, которые необходимо учесть при реализации. Для общего понимания представлена схема предлагаемого подхода, а также схемы наиболее сложных его частей.

4 РЕАЛИЗАЦИЯ

Создание инструмента для применения предложенного метода извлечения поведенческих моделей библиотек заключается в реализации каждой из его частей, разобранных ранее, и их связи между собой и пользователем. В данном разделе приведена общая архитектура инструмента и детали реализации его модулей, включая различные инженерные проблемы и пути их решения.

4.1 Общая архитектура

Реализованный инструмент может быть использован как консольное приложение или как библиотека, с помощью которой пользователь сможет реализовывать свои сценарии анализа или пользоваться отдельными её частями. Основой инструмента являются несколько пакетов:

1) analysis – пакет, содержащий реализацию извлечения трасс с помощью статического и динамического анализа, а именно построение ICFG, points-to анализ, фаззинг и сбор точек входа в программу
2) inference – пакет, обеспечивающий обработку трасс и восстановление из них поведенческих моделей
3) repository – пакет, реализующий получение репозиториев и работу с ними
4) storage – пакет, отвечающий за работу с БД
5) workflow – пакет, связывающий все реализованные возможности между собой с помощью определенных сценариев работы. Стоит отметить, что по сути пакет содержит примеры использования реализованных пакетов и не требует детального рассмотрения

4.2 Пакет repository

Данный пакет решает задачи поиска, получение и сборки проектов из GitHub и Maven Central Repository. Также в рамках этого пакета реализована возможность запуска тестов, имеющихся в репозитории. Рассмотрим каждую из задач подробнее.

4.2.1 Получение проектов с GitHub

Не смотря на то, что GitHub предоставляет открытый API для поиска по коду, при разработке возникли несколько проблем:

1) Поиск происходит по файлам, соответственно, сами репозитории могут повторяться. Необходимо это учитывать, чтобы не тратить лишнее время на получение и анализ уже рассмотренного ранее проекта.
2) Один поисковой запрос может найти не более 1000 результатов, что указано в документации[6]. С учетом повторения репозиториев, этого может быть недостаточно для автоматической работы длительное время с целью изучить большое количество проектов. При этом сторонние системы поиска по коду, например SearchCode, имеют более сильные ограничения.
3) Частые запросы к API приводят к достижению secondary rate limit[7]. При этом, основного лимита достаточно для выполнения запросов. Каким-то образом необходимо избежать введения искусственных задержек.

Для каждой из этих проблем были реализованы следующие решения:

[6]https://docs.github.com/en/rest/search/search?apiVersion=2022-11-28#about-search

[7]https://docs.github.com/en/rest/using-the-rest-api/rate-limits-for-the-rest-api?apiVersion=2022-11-28#about-secondary-rate-limits

1) Для исключения повторений следует хранить авторов и названия проанализированных репозитории в БД. Это также будет полезно для остановки и возобновлений поисковых сессий. Чтобы не обращаться каждый раз в БД для проверки репозитория на повторение, следует реализовать кэш.
2) Чтобы обойти лимит результатов поиска, реализован адаптивный фильтр по размеру файла. Способ фильтрации предлагает сам GitHub. Таким образом, мы можем задать диапазон от 0 до максимального размера файла, который нас интересует, и проходить его определенными шагами, например, изначально 100 байт. Если количество результатов поиска находится между нулем и 1000, то не меняем шаг. Если количество результатов равно нулю, то мы увеличиваем шаг в два раза, а если максимальному значению – уменьшаем шаг в два раза.
3) Во избежание secondary rate limit применяется ленивое получение репозиториев и их последовательный анализ, реализованное с помощью наследования Sequence в языке Kotlin. Каждый запрос к API возвращает набор файлов, из которых извлекаются уникальные репозитории, и до тех пор, пока все они не будут проанализированы, API использоваться не будет. Таким образом создается естественная задержка, позволяющая избегать лимита. Если он все же был достигнут (например, подряд выполнялись несколько уменьшений или увеличений размера шага), то используется sleep перед очередной попыткой запроса.

Для поиска и получения проектов с GitHub необходимо предварительно получить и указать свой токен для API в конфигурации инструмента, а на вход подать поисковой запрос и шаблон имени файла. Это необходимо, так как можно искать по импорту среди:

– .java файлов (path:**/*.java)
– build.gradle (path:**/build.gradle)
– build.gradle.kts (path:**/build.gradle.kts или path:**/build.gradle* для всех вариаций)
– pom.xml (path:**/pom.xml)

Для работы с удаленными GitHub репозиториями реализован класс GhRemoteRepository, позволяющий получать доступные артефакты и исходный код всего проекта:

1) Определяется наличие выпущенной версии
2) Проверяется наличие JAR файла соответствующего версии
3) Проверяется наличие исходного кода соответствующего версии
4) Если найдены, скачиваются JAR файл и архив с кодом версии, иначе выполняется клонирование с помощью git

4.2.2 Получение проектов с Maven Central

Веб-сайт Sonatype[8] предлагает возможность получения зависимых библиотек от выбранной. Однако публичного API для этого не предоставлено. С помощью инструментов разработчика в браузере было найдено внутреннее API, реализующее необходимую в рамках работы задачу. Способ нахождения и структура тела запроса представлена на Рис. 4. Чтобы исключить вероятность

[8]https://central.sonatype.com/

блокировки при использовании данного API, было направлено электронное письмо в поддержку Sonatype. В ответе говорилось, что допустимо использовать внутренний API, если нет намерений просканировать весь Maven Central, что в рамках выполняемой работы нас устраивает.

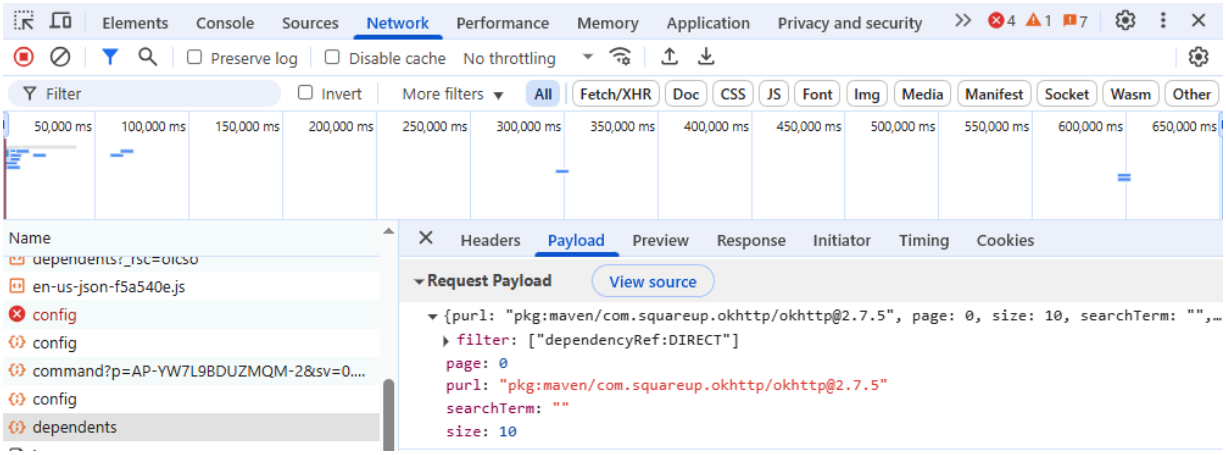

Рисунок 4 — Поиск необходимого API

Для поиска проектов следует указать группу, название и версию библиотеки.

Получение найденных проектов осуществляется с помощью специально созданного build.gradle файла на одну зависимость, позволяющего подставлять в него библиотеку для анализа и выполнять задачу разрешения зависимостей. Таким образом происходит автоматическое получение и сборка библиотеки для анализа и всех её зависимостей в заданную директорию.

Как уже было упомянуто в разделе, посвященному проектированию, получение исходного кода из Maven Central бессмысленно, так как он не содержит тестов, а сборка выполнена автоматически при получении проекта с помощью gradle.

4.2.3 Сборка проектов

Проекты из Maven Central собираются автоматически из-за выбранного метода их получения. Что касается проектов с GitHub, то даже при наличии JAR

файла в репозитории, он может не содержать зависимости, а следовательно, он может не запуститься. Также необходима компиляция тестов, если они есть в проекте.

Для этого требуется в первую очередь определить, какая система сборки используется в проекте. Это делается с помощью обхода файлов и поиска характерых файлов сборки: build.gradle, build.gradle.kts, pom.xml. Если ни один из файлов не обнаружен в корне проекта, то выполняется обход вложенных директорий, так как репозиторий может иметь нестандартные вложения или состоять из нескольких модулей.

Если проект использует Gradle, то вся работа с ним может быть выполнена с помощью Gradle Tooling API. В зависимости от конфигурации инструмента версия Gradle может быть установлена автоматически для каждого проекта, зафиксирована или может быть использован уже установленный на машине сборщик.

Maven не предполагает подобного API, поэтому в зависимости от конфигурации используется сборщик, к которому указан путь или из переменных окружения.

Класс локальных репозиториев предусматривает методы сбора трасс из файлов, которые создаются в корне проекта как результат запуска инструментированного кода.

4.3 Пакет analysis

Данный пакет содержит реализацию статического анализа, инструментацию и запуск фаззинга, в совокупности предоставляя возможности для извлечения трасс вызовов библиотеки с применением статических и динамических подходов.

4.3.1 Сбор точек входа в программу

Как для статического извлечения трасс, так и для фаззинга необходимо предварительно получить возможные точки в программу. Помимо main метода, это могут быть:

– Обработчики событий. Например методы, отвечающие за обработку запросов
– Конструкторы
– Конкретные известные пользователю методы

Чтобы иметь гибкость вопросе выбора входных точек в программу, реализован механизм фильтрации для методов. Фильтры пишутся в YAML формате, в процессе работы инструмента десериализуются и используются для определения входных точек при обходе всех методов в программе. Фильтр содержит следующие поля:

1) methodAnnotation – множество аннотаций метода, доступных в runtime
2) methodName – регулярное выражение для названия метода
3) classAnnotation – множество аннотаций класса, доступных в runtime
4) className – регулярное выражение для названия класса
5) kind – вид точки входа: метод, конструктор (init) или статический инициализатор (clinit)
6) args – список типов аргументов метода в их оригинальном порядке
7) returnType – возвращаемый тип
8) methodModifiers – модификаторы для метода: static, public, protected, private, final, synchronized, native

9) classModifiers – модификаторы для класса: static, public, protected, private, final, synchronized, enum

Если значения равны null, то параметр фильтра игнорируется. Пример фильтра для поиска публичных методов с названием, начинающимся на main, принимающих в себя массив строк, приведен на Листинг 4.

```
methodTags: ["API"]
methodName: "main.*"
classTags: null
className: null
kind: "method"
returnType: null
args: ["java.lang.String[]"]
methodModifiers: null
classModifiers: null
```

Листинг 4 — Пример фильтра в yaml формате

4.3.2 Статическое извлечение трасс

Извлечение трасс с применением статического анализа состоит из нескольких основных шагов:

1) Получить точки входа в программу. Эта задача подробно разобрана чуть выше.
2) Построить ICFG. Решается использованием стандартного API Soot, однако построение межпроцедурного графа потока управления требует запуска Soot в режиме «whole-program».
3) Выполнить обход ICFG для каждой точки входа, собирая вызовы и группируя их посредством применения points-to анализа к объектам. Рассмотрим данную задачу отдельно, так как она требует разработки определенного алгоритма обхода.

Для обхода ICFG и выполнения всех необходимых для извлечения трассы действий следует реализовать SceneTransormer и добавить его PackageManager. Именно созданный SceneTransormer будет определять проводимый анализ.

ICFG представляет собой все возможные пути выполнения программы и, очевидно, в среднем содержит множество циклов и ветвлений. Так как нас интересуют именно трассы исполнения, их длинна может быть бесконечной и ее следует ограничивать. Для этого были введены два параметра:

– Глубина обхода. Изначально устанавливается некоторое число, которое уменьшается каждый раз, когда мы уходим в реализацию вызова. Когда глубина равна 0, обход не может попасть в реализацию какого-либо вызова в рассматриваемом методе и обязан пройти его до конца. Возвращаясь на уровень выше, глубина инкрементируется.
– Длина. Задает лимит по количеству вершин графа в рамках обхода.

Извлечение трасс из ICFG представляет собой рекурсивный обход в глубину. Псевдокод алгоритма приведен на Листинг 5. Каждое ветвление имеет собственный набор трасс, при этом оно содержит общую часть с трассами, которые будут собраны для других веток (имеются ввиду ветки в одной и той же точке программы). Для того, чтобы не копировать наборы трасс, в методе фиксируется, какие из них были дополнены на данной итерации. В конце каждого метода, после выполнения рекурсивных вызовов, с помощью сохраненных индексов трассы приводятся в свое изначально состояние, которое было до ветвления. Сами трассы сохраняются в БД по достижению ограничений на обход или обходу всей программы. Для обхода в глубину самих вызовов в специальном стеке сохраняется точка в программе, к которой необходимо

вернуться после рассмотрения метода до конца. Когда вызываемый метод рассмотрен, точка возврата снимается со стека.

Для поиска вызовов конкретной библиотеки выполняется сравнение с указанным префиксом названий пакетов. Обнаружив вызов, необходимо применить points-to анализ для объекта вызова и объектов из уже накопленных трасс, получив множества переменных, которые могут указывать на тот же объект. Если полученные множества пересекаются, то переменные могут указывать на один и тот же объект, и вызов помещается в соответствующую трассу, иначе создается новая. Таким образом, через points-to выражается alias анализ.

При обходе перебираются все возможные в рамках наложенных ограничений по длине и глубине комбинации ветвлений. Благодаря этому возможно получение высокого покрытия кода и сбора содержательных трасс вызовов. Собранные трассы представляют собой последовательность из MethodData, содержание которой можно увидеть в Листинг 6 как часть динамически получаемой трассы.

```
graphTraverseLib(
    startPoint: Unit,
    isMainMethod: Boolean,
    ttl: Int = configuration.traversJumps,
    depth: Int = configuration.traversDepth
) {
    val currentSuccessors = icfg.getSuccsOf(startPoint)
    if (currentSuccessors.size == 0 || ttl <= 0) {
        if (ttl <= 0 || isMainMethod) {
            save(extracted)
        } else {
            val succInfo = continueStack.removeLast()
            graphTraverseLib(succInfo.first, succInfo.second, ttl - 1, depth + 1)
            continueStack.add(succInfo)
        }
    } else {
        for (succ in currentSuccessors) {
            var method: SootMethod? = null
            var continueAdded = false
            var klass: String? = null
            var indexesOfChangedTraces: List<Int>? = null
                if (succ is JInvokeStmt || succ is JAssignStmt) {
                    succ as AbstractStmt
                    if (succ.invokeExpr.method.declaringClass in Scene.v().applicationClasses)
                        method = succ.invokeExpr.method
                    if (method?.foundLib(lib)) {
                        klass = method.declaringClass.toString()
                        if (extracted[klass] == null) extracted[klass] = mutableListOf()
                        indexesOfChangedTraces = saveInvokeToTrace(succ.invokeExpr, extracted[klass])
                    }
                }
            if (method != null && depth > 0) {
                continueAdded = continueStack.add(Pair(succ, isMainMethod))
                icfg.getStartPointsOf(method).forEach { methodStart ->
                    graphTraverseLib(methodStart, false, ttl - 1, depth - 1)
                }
            } else graphTraverseLib(succ, isMainMethod, ttl - 1, depth)

            if (indexesOfChangedTraces != null) resetTraces(indexesOfChangedTraces, extracted[klass])
            if (continueAdded) continueStack.removeLast()
        }
    }
}
```

Листинг 5 — Псевдокод алгоритма обхода

4.3.3 Динамическое извлечение трасс

Динамическое извлечение трасс в первую очередь требует инструментации. Она реализована с помощью создания собственного SceneTransormer в Soot и заключается во вставке кода после вызовов библиотеки. Вызовы

библиотеки определяются тем же способом, что и при статическом анализе. Вставленный код собирает следующие данные:

1) Identity hash code объекта вызова
2) Время вызова в наносекундах
3) Uuid запуска
4) Название метода
5) Аргументы
6) Возвращаемый тип
7) Класс метода

Для инструментации реализован вспомогательный класс, отвечающий за запись собранной информации в файл, который подкладывается в JAR. Файлы создаются для каждого потока, чтобы избежать конфликтов при многопоточной работе. Uuid необходим, чтобы различать попытки запуска и исключить совпадение identity hash code в рамках разных рабочих сессий. Записи в файле представляют собой сериализованные объекты InvokeData и MethodData, которые успешно десериализуются при сборе трасс и группируются по identity hash code и uuid.

Soot при инструментации JAR файла теряет директорию META-INF, влияющую на его запуск. Не смотря на то, что тесты не используют JAR, а фаззинг требует на вход путь к конкретному методу, это учтено и реализован механизм сохранения изначальной мета-информации и добавления её в инструментированный JAR. Это полезно для пользовательских экспериментов по извлечению трасс, позволяя сохранять JAR действительно исполняемым.

Для получения трасс необходимо выполнить запуск тестов или фаззинг. Для запуска тестов используется Gradle Tooling API или предустановленный

Maven тем же образом, что описан в Раздел 4.2.3. При вызове задачи явно исключается сборка проекта, чтобы не потерять результаты инструментирования.

Для фаззинга автоматически при первом запуске происходит получение архива с исполняемыми файлами Jazzer под целевую операционную систему (в дальнейшем проверяется наличие исполняемых файлов). Сам фаззер вызывается как сторонний процесс для каждой точки входа и имеет следующие опции:

– -runs=N. Количество запусков указанного метода, где N задается пользователем. Если N = 0, то ограничение отсутствует.
– -max_total_time=N. Длительность работы фаззера в секундах, где N задается пользователем. Для бесконечной работы опция не указывается.
– --keep_going=0. Игнорирует найденные фаззером ошибки. Эта опция нужна, так как мы заинтересованы в трассах, а не в тестировании программы. На самом деле вместо нуля можно указать любое другое число, означающее количество игнорируемых ошибок.
– --autofuzz=qualified.reference.to.Class::method. Указывает цель для фаззинга.

Пример получаемой трассы для поиска вызовов класса java.io.File приведен в Листинг 6.

```
1  {"methodData":{"name":"<init>","args":
   ["java.lang.String"],"returnType":"void","isStatic":false,"klass":"java.io.File"},
2  "iHash":"443942537","date":"170309763...","uuid":"6b4..."}
3  {"methodData":{"name":"<init>","args":
   ["java.lang.String"],"returnType":"void","isStatic":false,"klass":"java.io.File"},
4  "iHash":"1243171897","date":"170309805...","uuid":"6b4..."}
5  {"methodData":{"name":"exists","args":[],"returnType":"boolean","isStatic":false,"klass":"java.io.File"},
6  "iHash":"1243171897","date":"170309805...","uuid":"6b4..."}
7  {"methodData":{"name":"createNewFile","args":[],"returnType":"boolean","isStatic":false,"klass":"java.io.File"},
8  "iHash":"1243171897","date":"170309806...","uuid":"6b4..."}
9  {"methodData":{"name":"toPath","args":[],"returnType":"java.nio.file.Path","isStatic":false,"klass":"java.io.File"},
10 "iHash":"1243171897","date":"170309807...","uuid":"6b4..."}
11 {"methodData":{"name":"<init>","args":
   ["java.lang.String"],"returnType":"void","isStatic":false,"klass":"java.io.File"},
12 "iHash":"1376151044","date":"170309807...","uuid":"6b4..."}
13 {"methodData":{"name":"delete","args":[],"returnType":"boolean","isStatic":false,"klass":"java.io.File"},
14 "iHash":"1376151044","date":"170309809...","uuid":"6b4..."}
15 {"methodData":{"name":"createNewFile","args":[],"returnType":"boolean","isStatic":false,"klass":"java.io.File"},
16 "iHash":"1243171897","date":"170309809...","uuid":"6b4..."}
```

Листинг 6 — Пример получаемой трассы

4.4 Пакет storage

Для хранения результатов поиска проекта и извлечения из них трасс использована SQLite. Так как это локальная БД, она позволяет легко создавать переносимые базы данных в виде файла для различных запусков инструмента. Это позволяет в дальнейшем использовать их как отправную точку для поиска репозиториев, дополнять уже накопленные трассы, проводить эксперименты по восстановлению с разными параметрами и делиться накопленными данными. Также она не требует специального окружения и может создаваться с использованием соответствующей библиотеки. Работа с БД реализована с помощью ktorm.

Сама БД состоит из нескольких таблиц:

– Repository. Таблица используется для хранения уже полученных репозиториев и краткой информации о них. Содержит в себе следующие поля:

  - name – название репозитория
  - namespace – иначе groupId, в случае если проект – библиотека из Maven
  - version – версия
  - author – автор
  - locator – локация репозитория, для GitHub это URL, а для Maven – PURL
  - source – источник, т.е. GitHub или Maven
  - date – дата получения репозитория
- Trace. Данная таблица предназначена для хранения трасс и содержит в себе следующие поля:
  - trace - трасса из вызовов в JSON формате. На самом деле подобное решение идет вопреки с нормализацией БД и является неоптимальным, можно сохранять методы в отдельную таблицу и хранить исключительно идентификаторы методов из неё. Но в рамках работы выбор сделан в пользу удобства просмотра накопленных трасс без средств обработки, а потенциальный объем данных в БД позволяет пренебречь нормальной формой
  - class - класс, для которого собрана трасса
  - count - количество найденных идентичных трасс
  - isStatic - содержит трасса статические вызовы или обычные. Так как в рамках предлагаемого подхода мы не можем определить контекст статического вызова, было решено выделить их в отдельную трассу

4.5 Пакет inference

После извлечения и сохранения определенного количество трасс появляется возможность восстановить поведенческую модель библиотеки. Процесс восстановления происходит в несколько этапов:

1) Выбор трасс и их предобработка в соответствии с режимом работы. Это могут быть трассы полученные только статическим или динамическим путем, а также статически полученные трассы, частично совпадающие с динамическими. Если был установлен порог погрешности, то также выполняется фильтрация трасс, которые будут использованы для восстановления
2) Формируется входной файл для фреймворка MINT, состоящий из предварительно указанных названий переходов и трасс
3) Применяется алгоритм k-tail, реализованный в рамках MINT. Результатом является граф в DOT формате
4) Определяются конечные состояния, т.е. из которых отсутствуют любые переходы.
5) Выполняется объединение конечных состояний. Для этого выбирается основное конечное состояние и у переходов, ведущих в остальные конечные состояния, подменяется точка назначения на основное. Лишние конечные состояния удаляются. Как уже было описано в разделе, посвященному проектированию, это действие не несет какой либо потери информации
6) Выполняется трансляции DOT графа в JSON для унификации формата хранения и какой-либо потенциальной работы с ним (например,

интерактивное редактирование). При этом DOT остается в качестве артефакта работы инструмента

Для восстановления можно управлять параметром k, влияющим на длину минимально совпадающих частей трассы для слияния, а также выбирать другие предоставляемые mintframework стратегии восстановления и алгоритмы классификации.

4.6 Реализованный инструмент

В результате разработан инструмент, реализующий подход автоматического извлечения поведенческих моделей библиотек из доступных программных репозиториев. В разработке учтена возможность использования модулей независимо друг от друга, а также предоставлены возможности настройки используемого окружения и библиотек. Исходный код проекта доступен в репозитории LibMiner[9] на GitHub.

[9]https://github.com/kechinvv/LibMiner

5 ТЕСТИРОВАНИЕ

Для тестирования предложенного подхода и его реализации были получены поведенческие модели для классов нескольких библиотек:

– java.util.zip.ZipOutputStream

– java.lang.StringBuilder

– java.security.Signature

– java.net.Socket

– org.columba.ristretto.smtp.SMTPProtocol

Все из этих классов, имеют модели разной сложности и, за исключением StringBuilder, рассматривались в упомянутых исследованиях[11,14,18], авторы которых сравнивали полученные КА с эталонными моделями. Эти модели расположены в репозитории GitHub[10]. Не смотря на то, что авторы использовали их как эталон, некоторые из представленных КА явно неполны и ограничены определенным набором вызовов (Signature и SMTPProtocol), а модель класса Socket похожа на полученную автоматически и отредактированную человеком. По этой причине сравнение и оценка полученных автоматов будет приблизительна и субъективна.

Стоит сказать, что это можно решить созданием своих эталонов, однако это трудоемкая задача, о чем сообщалось в каждом исследовании с проведенным тестированием. Решалась она работой нескольких человек, а все модели, которые можно было переиспользовать из других исследований – переиспользовались. Поэтому в рамках выполняемой работы задача составления эталонов не выполнялась.

[10]https://github.com/ModelInference/SpecForge

Оценка точности и полноты получаемых поведенческих моделей не проводилась, так как работа не предлагает собственные алгоритмы восстановления. Подробное сравнение алгоритмов проведено в исследовании «Automatic mining of specifications from invocation traces and method invariants»[11]. На трассы также влияет корректность получаемых трасс:

– Обход ICFG может получать все возможные пути исполнения, однако часто программы пишутся так, что не все состояния достижимы при реальном исполнении
– Фаззинг способен создавать некорректные входные объекты, что в теории может привести к появлению трасс, невозможных при реальном использовании (проблема, аналогичная предыдущему пункту)
– Анализ указателей имеет определенную погрешность

Влияние первых двух факторов сложно оценить и учесть. Что касается анализа указателей, то определением точности алгоритма Андерсена на практике занимались авторы работы «The Flow-Insensitive Precision of Andersen's Analysis in Practice»[19]. Результат показал, что точность очень сильно варьируется от проекта к проекту. Для полноты проведенного тестирования, следует оценивать влияние качества трасс на получаемые КА, но ввиду отсутствия качественных эталонов и сложности их составления, эта часть не будет выполнена в рамках данной работы.

5.1 Полученные модели

Все модели получены в результате объединения трасс, полученных статическим и динамическим путем. Сделано это по той причине, что динами-

ческие трассы не содержат новых покрытий, но полученные таким образом последовательности вызовов имеют более высокий уровень доверия.

В качестве источника проектов использовался GitHub, так как наиболее демонстративными и понятными примерами являются классы стандартной библиотеки, не хранящиеся в Maven Central. При этом библиотеки с Maven Central в результате локальных экспериментов обладали наиболее высоким процентом успешного применения фаззинга, что повышает качество получаемых моделей.

Запуск инструмента для каждого целевого класса выполнялся в следующих условиях:

- Лимит получаемых проектов: 1000
- Лимит получаемых трасс: 1000000
- Лимит длины трассы: 200 вызовов
- Ограничение глубины обхода: 10
- Параметр k для k-tail: 1 или 2
- Количество запусков фаззера: 10000
- Ограничение времени работы фаззера: 300 секунд
- Системные характеристики: 20 гигабайт оперативной памяти, процессор Intel Core i5-8300H CPU 2.30GHz, 4 физических ядра

5.1.1 ZipOutputStream

Полученная модель представлена на Рис. 5, эталонная на Рис. 6. Важно отметить, что к данному эталону не возникает вопросов о его полноте и методу получения. Из рисунков видно, что обе модели покрывают все необходимые вызовы. Можно заметить, что получаемая модель не противоречит эталону,

но не содержит всех переходов и является неполной. Более длительный поиск проектов и трасс может улучшить ситуацию, но вряд ли даст эталонный результат.

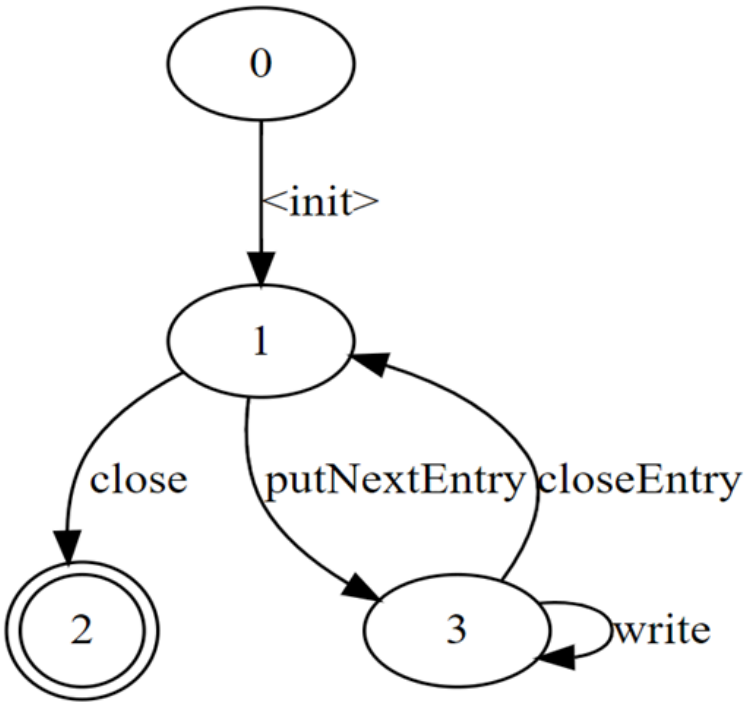

Рисунок 5 — Полученная модель ZipOutputStream

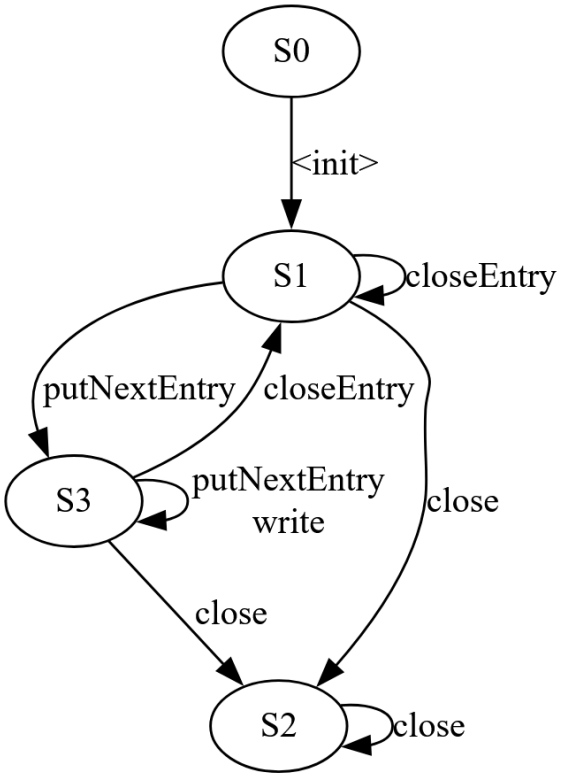

Рисунок 6 — Эталонная модель ZipOutputStream

5.1.2 StringBuilder

Для данной модели на Рис. 7 нет эталона, но легко заметить, что модель не противоречит реальному использованию StringBuilder. Тем не менее, получены не все возможные вызовы библиотеки и полученный КА очень простой. Более длительное применение инструмента точно обогатит данную модель переходами.

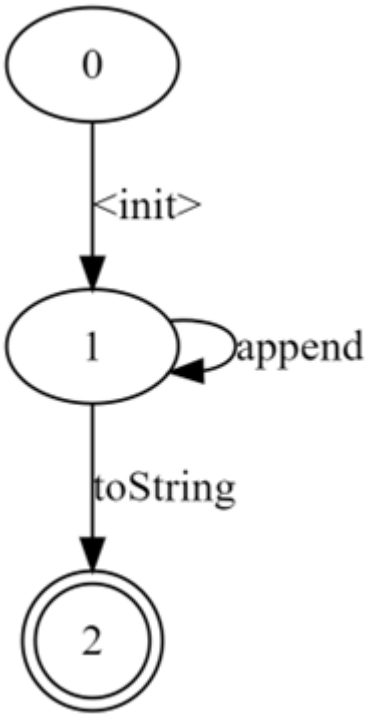

Рисунок 7 — Полученная модель StringBuilder

5.1.3 Signature

Для полученной модели, представленной на Рис. 8, существует эталонная модель, но она содержит лишь небольшую часть действительно возможных вызовов и переходов. Тем не менее, из модели видно, что явно выделяются определенные повторяющиеся паттерны переходов между состояниями, а также есть дубликаты переходов. Это является следствием работы алгоритма и

не может быть решено без потери подробностей или использования другого алгоритма.

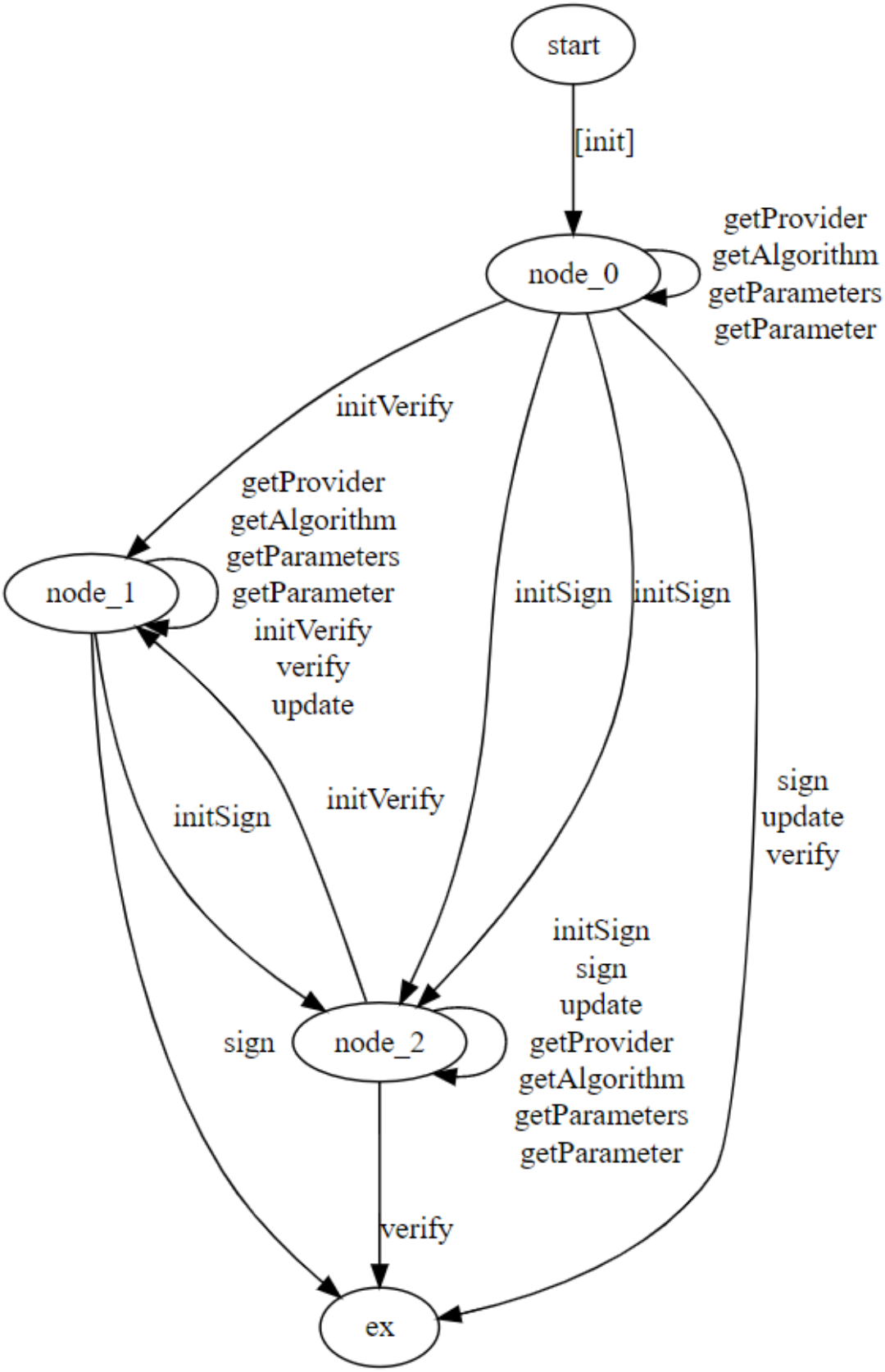

Рисунок 8 — Полученная модель Signature

5.1.4 Socket

При сравнении полученной модели с Рис. 9 и эталона Рис. 10 видно, что полученный КА явно не содержит всех вызовов, а также имеет слабое обобщение. Также здесь видна погрешность статического подхода в том, что потеряно создание объекта и некоторые переходы происходят без вызова конструктора. Тем не менее, из полученного КА видны дальнейшие точки для ручного исправления. Например, объединение get методов в один переход.

Что касается эталонной модели, к ней есть вопросы, так как она явно может быть обобщена лучше, а состояние S9 играет роль начального, хотя вызова конструктора для нее нет. В целом это допустимо, если библиотека содержит фабрики объектов. Тем не менее, складывается впечатление, что это обработанный вручную и автоматически полученный КА.

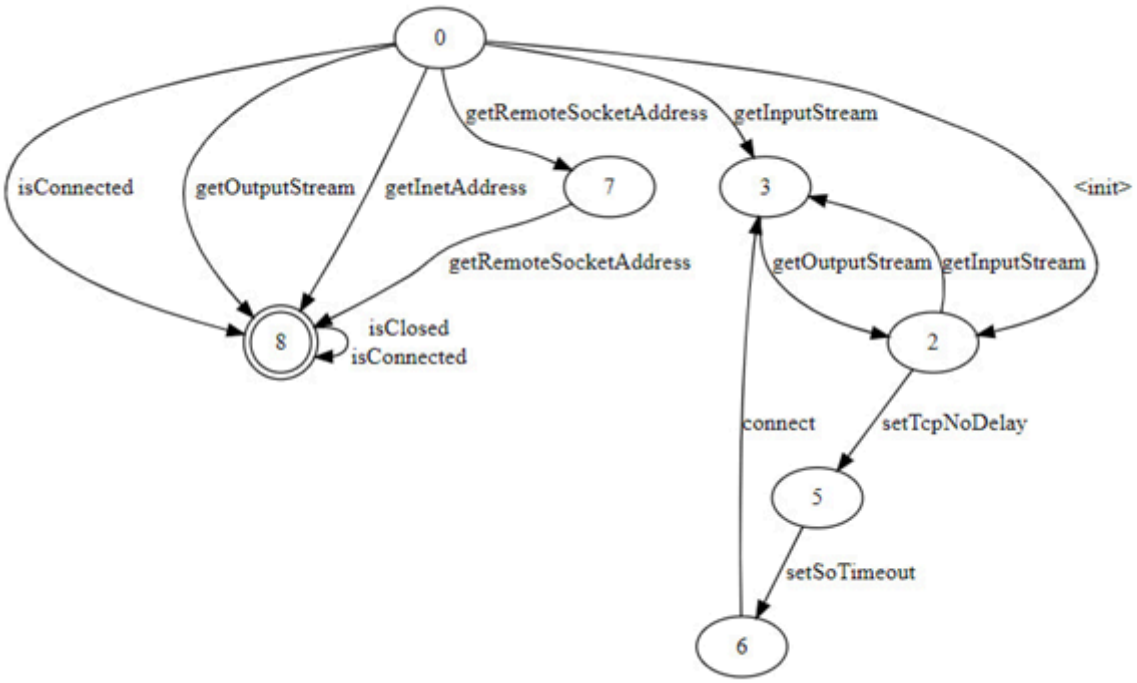

Рисунок 9 — Полученная модель Socket

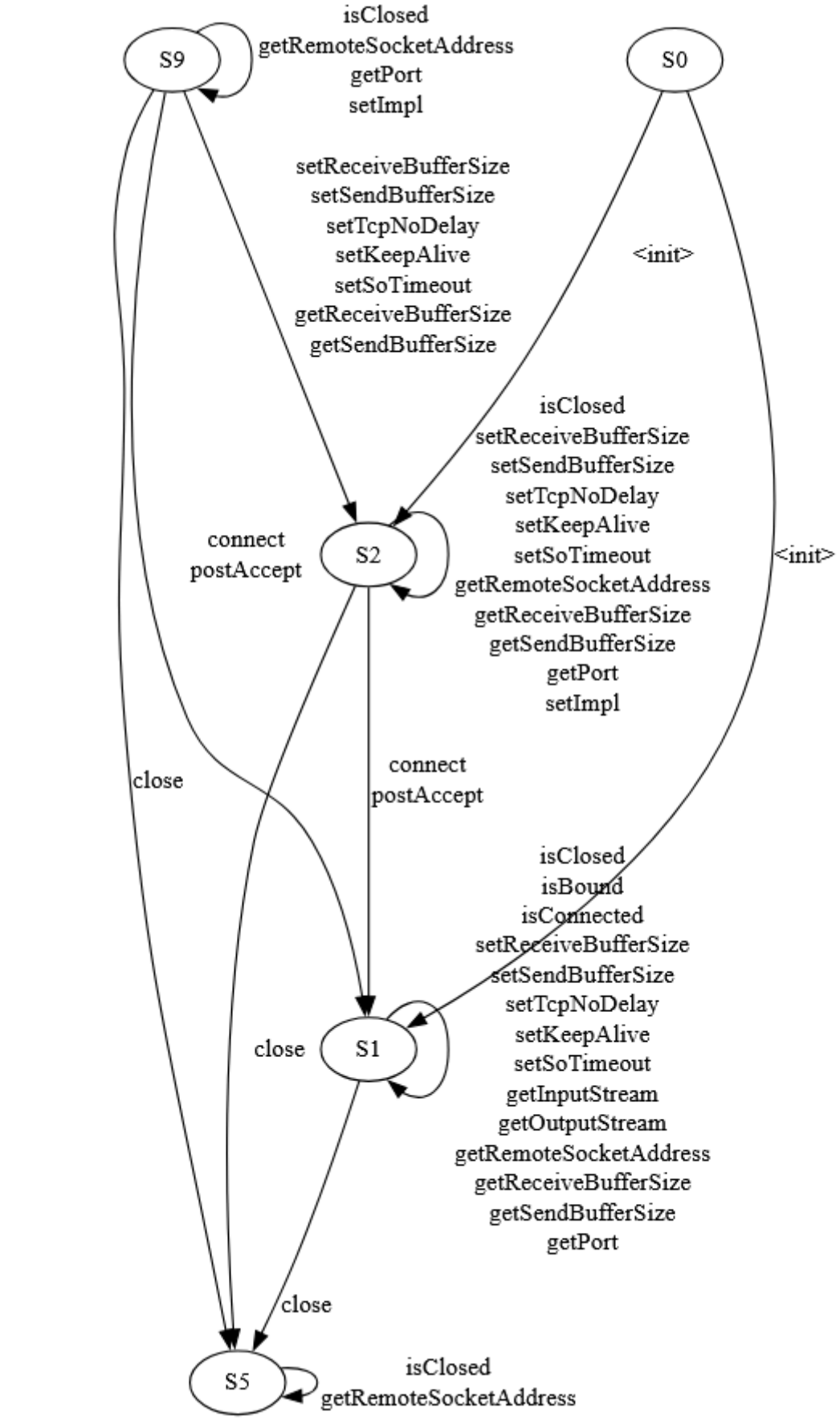

Рисунок 10 — Эталонная модель Socket

5.1.5 SMTPProtocol

Для модели с Рис. 11 также существует неполная эталонная модель, но она содержит очень малое количество доступных вызовов. Не смотря на это, из полученного КА также видно, что он имеет слабую обобщенность и визуально выделяющиеся повторяющиеся паттерны вызовов, которые скорее всего могут быть объединены вручную. Также следует отметить наличие большинства методов класса, что говорит об успехе сбора трасс.

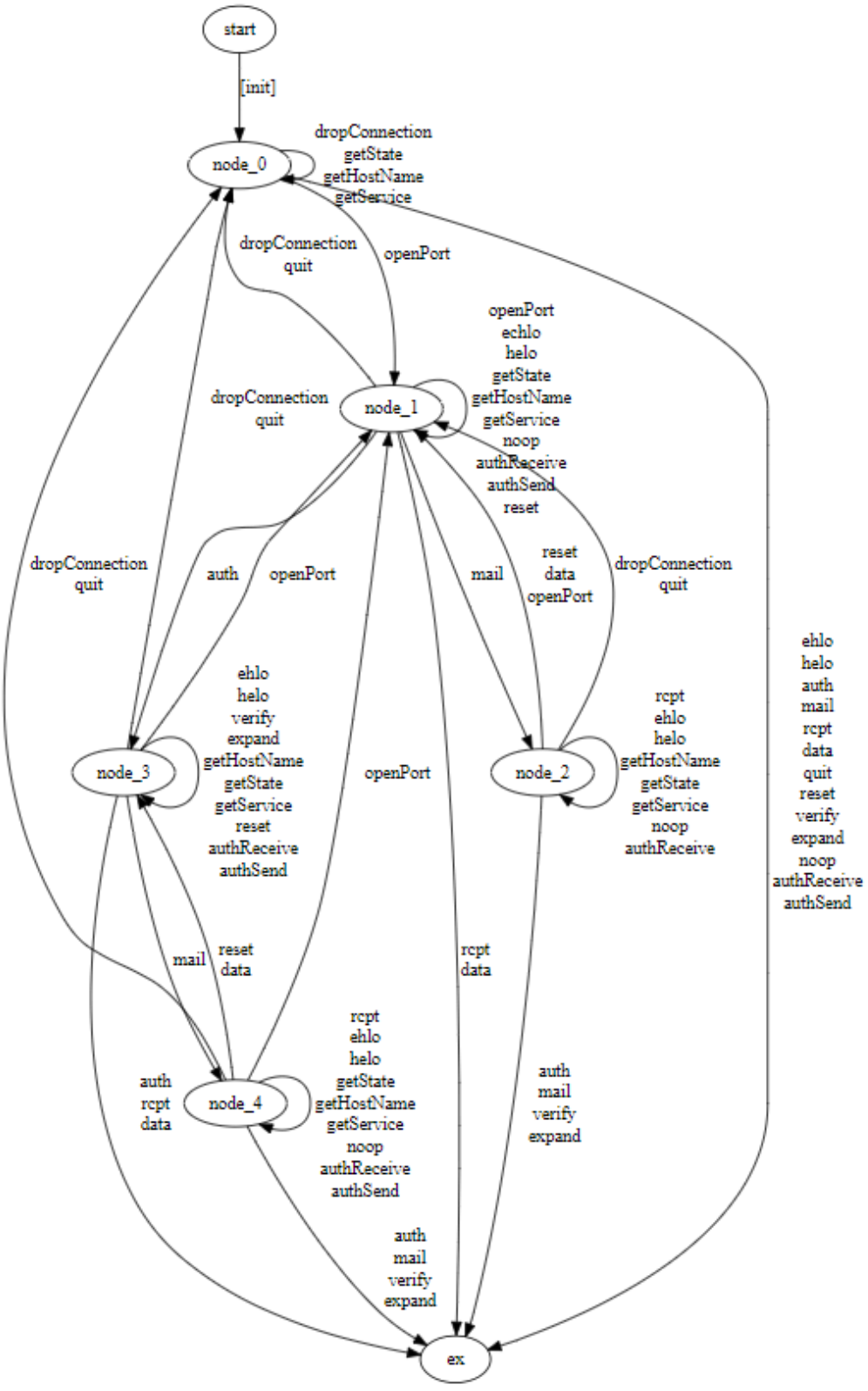

Рисунок 11 — Полученная модель SMTP

5.2 Выводы

Получение поведенческих моделей с помощью предложенного подхода и его реализации показало свою работоспособность. Получаемые модели

действительно являются аппроксимацией сверху, а представленный подход требует улучшений в области извлечения трасс и использованию алгоритмов восстановления. Тем не менее, в автоматическом режиме успешно выполняется получение проектов, использующих заданную библиотеку, происходит извлечение трасс как статически, так и динамически, а восстанавливаемые модели соответствуют полученным трассам и могут быть интерпретированы человеком для дальнейшей доработки. Более того, манипуляция параметрами поиска проектов, входных точек и параметров алгоритма восстановления может сильно улучшить результат.

ЗАКЛЮЧЕНИЕ

В результате работы предложен метод автоматического извлечения поведенческих моделей библиотек из открытых программных репозиториев. Метод заключается в применении статических и динамических подходов к полученным с GitHub или Maven Central Repository проектам для извлечения трасс и последующем применении алгоритма восстановления КА. Для апробации метода реализован инструмент, позволяющий запускать автоматические сценарии для получения поведенческих моделей в виде КА и предоставляющий элементы управления настройками поиска, анализа, восстановления и используемого окружения.

Результат исследования показал применимость предложенного подхода. С помощью поиска по GitHub и Maven Central Repository была решена задача получения общедоступных программ, использующих заданную библиотеку. Для извлечение трасс применены статический и динамический анализ. Статический заключается в обходе ICFG и применении Points-to анализа на основе алгоритма Андерсена, а динамический – в инструментации и запуске тестов или фаззинга входных точек в программу на основе Jazzer. Восстановление поведенческих моделей выполнено с применением алгоритма k-tail, реализованного в рамках инструмента MINT.

Представленный метод и инструмент требуют дальнейшего развития в нескольких направлениях.

Одна из проблем в том, что получаемые КА являются сильной аппроксимацией сверху и на сложных библиотеках имеют мало общего с истинными поведенческими моделями. Это является следствием используемого алгоритма восстановления и игнорирования состояния программы, включая значения

аргументов. Для решение этого необходимо провести работу над инструментаций, реализовав сбор значений аргументов и доступных в контексте переменных. Это позволит применять более сложные алгоритмы восстановления, обеспечивающие лучшую обобщенность модели. При этом важно сохранить простоту применения инструментации для работы в автоматическом режиме. Другой способ получения информации о состоянии программы, причем сразу в форме предикатов - использование символьного исполнение, но это также объект для отдельного исследования.

Еще одна проблема, требующая решения – отсутствие тестов в проектах, их малое количество или плохая репрезентативность в контексте получаемых трасс. В данной работе для решения этой проблемы применяется фаззинг, но, вероятно, лучший результат покажет генерация тестов, направленная на получения трасс, содержащих поведение заданной библиотеки (схожее с Tautoko). Однако, это отдельная сложная задача, требующая соответствующего исследования.

Сильным ограничением для автоматизации динамических методов является зависимость программ от определенного внешнего окружения - базы данных, ПО, переменные окружения. Идеальным решением этого было бы мокирование подобного кода, интегрированное в задачу генерацию тестов. Это бы обеспечило высокую долю успешно исполняемых тестов. На текущий момент нет инструментов, позволяющих это осуществить – в связи со сложностью решения и узким направлением его применения актуален вопрос релевантности исследований по данной теме.

Не смотря на указанные недостатки, требующие дальнейшего развития и дополнительных исследований, предложенный метод является комплексным

решением задачи получения поведенческих моделей. При этом для его применения требуется сравнительно небольшое участие человека. Что касается необходимости вручную обрабатывать полученные КА – это ограничение всех существующих подходов к восстановлению, на данный момент не имеющее другого решения, кроме как использовать различные алгоритмы и экспериментировать с настройкой их параметров. Хочется надеяться, что инструменты, позволяющие автоматизировать получение поведенческих моделей, в будущем смогут улучшить опыт разработки и применения анализов, основанных на использовании формальных спецификаций.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Ivanov D. и др. UTBot Java at the SBST2022 tool competition. 2023. сс. 39–40.
2. Ицыксон В.М. и др. Моделирование поведения функций стандартной библиотеки в задачах анализа программ // Информационно-управляющие системы. 2024. № 4. сс. 24–39.
3. Itsykson V. LibSL: Language for Specification of Software Libraries // Programmnaya Ingeneria. 2018. т. 9, № 5. сс. 209–220.
4. Biermann A., Feldman J. On the Synthesis of Finite-State Machines from Samples of Their Behavior // IEEE Transactions on Computers. 1972. № 6. сс. 592–597.
5. Lorenzoli D., Mariani L., Pezzè M. Automatic generation of software behavioral models // Proceedings - International Conference on Software Engineering. 2008. сс. 501–510.
6. Ernst M.D., Perkins J.H., Guo P.J. The Daikon system for dynamic detection of likely invariants // Sci. Comput. Program. 2007. т. 69. сс. 35–45.
7. Shoham S., Yahav E., J. S.F. Static Specification Mining Using Automata-Based Abstractions // IEEE Transactions on Software Engineering. 2008. т. 34, № 5. сс. 651–666.
8. Cousot P., Cousot R. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. 1977. сс. 238–252.
9. Andersen L.O., Lee P. Program Analysis and Specialization for the C Programming Language. 2005.
10. Lerch J. и др. Access-Path Abstraction: Scaling Field-Sensitive Data-Flow Analysis with Unbounded Access Paths (T). 2015. сс. 619–629.

11. Krka I., Brun Y., Medvidovic N. Automatic mining of specifications from invocation traces and method invariants // Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. 2014. т. 9, № 5. cc. 178–189.
12. Caso G. и др. Automated Abstractions for Contract Validation // IEEE Trans. Software Eng. 2012. т. 38. cc. 141–162.
13. Walkinshaw N., Taylor R., Derrick J. Inferring Extended Finite State Machine models from software executions // 20th Working Conference on Reverse Engineering (WCRE). 2013. cc. 301–310.
14. Dallmeier V., Knopp N., Mallon C. Automatically Generating Test Cases for Specification Mining // IEEE Transactions on Software Engineering. 2012. т. 38, № 2. cc. 243–257.
15. Dallmeier V. и др. Mining object behavior with ADABU // WODA. 2006. с. .
16. About [электронный ресурс]. URL: https://github.com/about.
17. Sridharan M., Fink S.J. The Complexity of Andersen's Analysis in Practice // Static Analysis / под ред. Palsberg J., Su Z. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. cc. 205–221.
18. Le T.-D.B. и др. Synergizing Specification Miners through Model Fissions and Fusions // Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE). 2015. cc. 115–125.
19. Blackshear S. и др. The Flow-Insensitive Precision of Andersen's Analysis in Practice // Static Analysis / под ред. Yahav E. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. cc. 60–76.