

## СОДЕРЖАНИЕ

Список сокращений и условных обозначений .....	2
Терминология .....	3
Введение .....	4
1 Анализ решений в области восстановления поведенческих моделей .....	7
1.1 Критерии сравнительного анализа .....	7
1.1.1 Метод извлечения трасс .....	7
1.1.2 Алгоритм восстановления модели .....	8
1.1.3 Применимость к реальным проектам и возможность автоматизации .....	9
1.1.4 Доступ к исходному коду .....	10
1.2 Обзор работ по извлечению спецификаций .....	10
1.3 Результаты анализа .....	14
2 Пути решения .....	15
2.1 подсекция .....	15
3 Проектирование .....	16
4 Реализация .....	17
5 Тестирование .....	18
Заключение .....	19
Список использованных источников .....	20

# **СПИСОК СОКРАЩЕНИЙ И УСЛОВНЫХ ОБО- ЗНАЧЕНИЙ**

- КА - конечный автомат
- ООП - объектно-ориентированное программирование
- JVM - Java Virtual Machine
- MBT - Model Based Testing
- PBT - Property Based Testing
- ПО - программное обеспечение
- СА - статический анализ
- ДА - динамический анализ

# ТЕРМИНОЛОГИЯ

- SMT-решатель
- условия корректности

# 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 ее 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 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[ССЫЛКА] и UtBot[ССЫЛКА]. 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[ЕСТЬ ЛИ ССЫЛКА НА ПРОЕКТ?]. Кроме этого, спецификации могут использоваться в MBT и RBT 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.

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 ООП, 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 извлечения поведенческих моделей библиотек посвящена данная работа. В рамках ее выполнения будет 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 КА. На данный момент комплексных 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 автоматизация, 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 использования пользовательских репозиторий, 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 СА, по крайней мере в рамках данной работы, не будут предприниматься попытки обработать многопоточную работу программы. Что касается ДА, то здесь на результат могут влиять точки входа в программу. Так как трассы будут собираться из запусков имеющихся или сгенерированных тестов, невозможно гарантировать корректность начального состояния библиотеки.

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, 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 извлечения 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[1]` требует на вход исключительно последовательности вызовов и параметра `k`, определяющего длину соединяемых цепочек. Другой алгоритм, `gk-tail[2]`, 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[1]` и `gk-tail[2]`, 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 длиной  $k$  (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[3] (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 ее 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»[4] авторы статически собирают трассы в виде последовательностей объектов одного типа, используя абстрактную интерпретацию[5]. Для получения последовательностей вызовов над конкретным экземпляром объекта в исследовании используется points-to анализ на основе алгоритма Андерсона (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 десятков 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[6]. Авторы делают вывод, что получаемые поведенчески модели достоверно описывают поведение библиотек, однако являются сильной аппроксимацией сверху истинной 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 инструмент для проведения анализа, однако ссылки на них не приводятся.

Авторы статьи «Automatic mining of specifications from invocation traces and method invariants»[7] подробно рассмотрели и сравнили четыре алгоритма восстановления моделей из трасс. При этом был рассмотрел базовый алгоритм k-tail, предложены улучшения для алгоритма Contractor[8], 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 исследования авторы получали инварианты с помощью инструмента динамического анализа Daikon[3]. 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%, имея 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<sup>1</sup> в открытом доступе предлагают авторы статьи «Inferring Extended Finite State Machine models from software executions»[9]. Авторы развивают идею алгоритма *gk-tail* и предлагают использовать информацию о значении аргументов в анализируемых вызовах. Однако новизна заключается в том, что для поиска конфликтов

---

<sup>1</sup><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`[10] для генерации тестов и `ADABU`[11], 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 подходов, а наличие инструментов в открытом доступе<sup>2</sup> делает полученные результаты очень полезными. Тем не менее, 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**

---

<sup>2</sup><https://www.st.cs.uni-saarland.de/models/>

## **2 ПУТИ РЕШЕНИЯ**

### **2.1 подсекция**

### **3 ПРОЕКТИРОВАНИЕ**



## **4 РЕАЛИЗАЦИЯ**

## **5 ТЕСТИРОВАНИЕ**

## **ЗАКЛЮЧЕНИЕ**

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Biermann A., Feldman J. On the Synthesis of Finite-State Machines from Samples of Their Behavior // IEEE Transactions on Computers. 1972. № 6. cc. 592–597.
2. Lorenzoli D., Mariani L., Pezzè M. Automatic generation of software behavioral models // Proceedings - International Conference on Software Engineering. 2008. cc. 501–510.
3. Ernst M.D., Perkins J.H., Guo P.J. The Daikon system for dynamic detection of likely invariants // Sci. Comput. Program. 2007. т. 69. cc. 35–45.
4. Shoham S., Yahav E., J. S.F. Static Specification Mining Using Automata-Based Abstractions // IEEE Transactions on Software Engineering. 2008. т. 34, № 5. cc. 651–666.
5. Cousot P., Cousot R. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. 1977. cc. 238–252.
6. Lerch J. и др. Access-Path Abstraction: Scaling Field-Sensitive Data-Flow Analysis with Unbounded Access Paths (T). 2015. cc. 619–629.
7. 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.
8. Caso G. и др. Automated Abstractions for Contract Validation // IEEE Trans. Software Eng. 2012. т. 38. cc. 141–162.

9. 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.
10. 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.
11. Dallmeier V. и др. Mining object behavior with ADABU // WODA. 2006. с. .