Санкт-Петербургский государственный политехнический университет
Факультет технической кибернетики
Кафедра компьютерных систем и программных технологий

Диссертация допущена к защите
зав. кафедрой

_______________ В.Ф. Мелехин

«____» _________________ 2011 г.

ДИССЕРТАЦИЯ
на соискание ученой степени
МАГИСТРА

Тема: РЕВЕРС-ИНЖИНИРИНГ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ДЛЯ ИЗВЛЕЧЕНИЯ ПОВЕДЕНЧЕСКИХ МОДЕЛЕЙ

230100 – Информатика и вычислительная техника
230100.68.01 – Высокопроизводительные вычислительные системы

Выполнил студент гр. 6081/11м _______________ Н.Н. Васильев

Научный руководитель,
к. т. н., доц. _______________ В.М. Ицыксон

Консультант по нормоконтролю,
ст. преп. _______________ С.А. Нестеров

Санкт-Петербург – 2011

РЕФЕРАТ

Отчет, 64 стр., 19 рис., 3 табл., 23 ист., 2 прил.

АБСТРАКТНОЕ СИНТАКСИЧЕСКОЕ ДЕРЕВО, ГРАФ ПОТОКА УПРАВЛЕНИЯ, СТАТИЧЕСКИЙ АНАЛИЗ КОДА, РЕВЕРС-ИНЖИНИРИНГ, КОНЕЧНЫЙ АВТОМАТ

Современное программное обеспечение часто сопровождается недостаточной проектной документацией. Восстановление документации осуществляется различными способами, от ручной инспекции программного кода до автоматического извлечения программных моделей. Процесс восстановления такого рода информации называется реверс-инжинирингом.

При разработке ПО используются различные шаблоны проектирования, модели разного уровня обобщения. Одной из таких моделей является модель конечного автомата. В рамках данной работы разрабатывается технология реверс-инжиниринга программ с целью выделения моделей конечных автоматов.

Основой предлагаемой технологии является использование средств статического анализа. Аннотирование используется для выделения переменных состояния. API компилятора используется для получения AST программы. С помощью модели графа потока управления осуществляется анализ поведения программы. Выделяются переходы между состояниями и их условия, невозможные переходы исключаются. Диаграмма конечного автомата записывается в текстовом формате DOT, и затем преобразуется в изображение с помощью GraphViz.

Разработан прототип системы выделения моделей для языка Java, позволяющий выделять граф конечного автомата для одной переменной состояния для исходного кода, находящегося внутри методов.

Проведено тестирование прототипа. Приведен способ тестирования и результаты для нескольких примеров, в том числе использованы примеры автоматных программ других авторов.

ABSTRACT

Report, 64 pages, 19 figures, 3 tables, 23 references, 2 appendices

ABSTRACT SYNTAX TREE, CONTROL FLOW GRAPH, STATIC CODE ANALYSIS, REVERSE-ENGINEERING, FINITE-STATE MACHINE

Software often comes undocumented. There are several ways to redocument it, from manual code review to automated reverse-engineering.

Software development requires more and more complex solutions, so design patterns and models of many kinds are used to reduce software complexity to human-understandable level. One of the models used is Finite-State Machine. The goal of this research is to introduce a method to reverse-engineer the code to extract FSM model.

The method being introduced is based on static analysis of program code. State variables are annotated, the compiler is used to produce AST, compiler tree API is used to fetch facts from AST, and to build intraprocedural CFG, which is analyzed for paths leading from one state variable assignment to another, and their conditions. Impossible paths are excluded, paths left are combined and the graph of FSM is then produced.

The prototype is implemented for Java language, allowing to extract one variable' FSM for intraprocedural code areas.

Prototype testing method and number of test results are provided. Test results include automata-based programs designed by third parties.

СОДЕРЖАНИЕ

СПИСОК ОБОЗНАЧЕНИЙ И СОКРАЩЕНИЙ

API	Application Programming Interface
ASG	Abstract Semantic Graph
AST	Abstract Syntax Tree
CFG	Control-Flow Graph
DDG	Data Dependency Graph
IDE	Integrated Development Environment
JDK	Java Development Kit
UML	Unified Modeling Language
XML	Extensible Markup Language
КА	Конечный автомат
ООП	Объектно-Ориентированное Программирование
ПО	Программное обеспечение
РИ	Реверс-инжиниринг

ВВЕДЕНИЕ

Жизненный цикл разработки программного обеспечения как правило включает проектирование, разработку, поддержку исходного кода. Нередко разработкой программной системы занимаются одни люди, а его поддержкой – другие. Это происходит потому, что затраты на поддержку существующей реализации некоторой программной системы могут быть существенно меньше затрат на ее полную переработку. В таких случаях говорят об унаследованном исходном коде.

Для успешной поддержки программной системы необходимо обладать знаниями о ее внутреннем устройстве. В случае унаследованного исходного кода такие знания часто не могут быть переданы от предыдущих разработчиков к последующим, поэтому все знания о системе ограничены проектной документацией. Одним из способов расширить понимание системы является реверс-инжиниринг ПО.

Реверс-инжиниринг ПО – это процесс, при котором восстанавливаются исходные принципы построения программного продукта. Это могут быть, например, способ размещения программы в памяти, взаимодействие с процессором и операционной и файловой системами, сетевая активность и другие особенности поведения программы.

При разработке современного ПО используются устоявшиеся способы проектирования, начиная от парадигм программирования и заканчивая шаблонами проектирования и программными моделями. Это позволяет обобщать элементы реализации системы и представлять структуру системы в доступном для понимания человеком виде. Такого рода информация обычно содержится в проектной документации программного продукта.

Однако часто проектная документация является неполной или отсутствует, но она необходима для поддержки и развития ПО. В связи с этим встает задача реверс-инжиниринга ПО с целью восстановления проектной документации.

Одной из используемых при проектировании программных моделей является *модель конечного автомата*, которая позволяет описывать поведение системы с конечным числом состояний. Чаще всего данная модель используется во встраиваемых системах, однако она также находит применение и в других областях отрасли программной инженерии. Поскольку существует потребность в реверс-инжиниринге проектной документации ПО в целом, также существует потребность в реверс-инжиниринге моделей конечных автоматов.

Целью данного диссертационного исследования является разработка технологии извлечения моделей КА из исходного кода средствами статического анализа ПО. В рамках данной работы также производится построение прототипа системы извлечения моделей для языка Java.

Работа состоит из пяти разделов. В разделе 1 вводится проблема извлечения моделей КА из ПО, приводится классификация и разбор методик анализа ПО, производятся обзоры существующих подходов к выделению моделей КА из реализации программной системы и средств реверс-инжиниринга ПО.

Раздел 2 посвящен постановке задачи выделения моделей, формулировке требований к разрабатываемому методу, выбору целевого языка программирования и пути решения задачи.

В разделе 3 описывается предлагаемая технология выделения моделей КА из исходного кода. Приводятся общая схема процесса извлечения моделей, способ отображения состояний программы в множество состояний модели, алгоритмы построения специфических структур данных и их анализа, способ генерации выходных данных.

Раздел 4 посвящен описанию реализации прототипа системы. Приводятся иерархическая схема организации прототипа, описываются используемые шаблоны проектирования, реализации предлагаемых алгоритмов, взаимосвязи сущностей программы. Исходные коды некоторых элементов прототипа приведены в приложении А.

В разделе 5 производится тестирование разработанного прототипа системы. Вводится способ тестирования, выполняется сравнение ожидаемых и полученных результатов, и их анализ.

1 ОБЗОР МЕТОДОВ ИЗВЛЕЧЕНИЯ ПОВЕДЕНЧЕСКИХ МОДЕЛЕЙ ИЗ ПРОГРАММ

В данном разделе производится обзор существующих подходов к выделению моделей КА из реализации программной системы, средств реверс-инжиниринга программ.

В подразделе 1.1 изложена краткая справка об истории создания модели КА и ее место в современном процессе проектирования ПО. Подраздел 2.1 посвящен формальному описанию модели и решаемой задачи. В подразделе 1.3 приведена классификация методов выделения поведенческих моделей[1]из их программной реализации. Подраздел 1.4 логически продолжает 1.3 более детальным сравнением статического и динамического анализа программ. В подразделе 1.5 рассматриваются существующие подходы к решению задачи о восстановлении моделей КА. Подраздел 1.6 посвящен краткому обзору существующих средств реверс-инжиниринга программ.

1.1 Роль конечных автоматов при проектировании ПО

Модель конечного автомата как системы с конечным числом состояний появилась в середине XX века как разрешение потребности в эффективном проектировании электрических переключательных схем с памятью. Работы Хаффмана, Шеннона, Мили и Мура в этом направлении принято считать истоками теории [1]. Примерно в то же время было положено начало изучению модели нейронных сетей, также являющихся системами с конечным числом состояний.

В настоящее время модель конечного автомата находит применение в таких сферах инженерного творчества, как разработка и верификация дискретных устройств, построение лексических анализаторов, задачи поиска шаблонов в тексте, разработка и реализация сетевых протоколов, верификация моделей программного обеспечения. Этот список можно продолжить.

[1]Здесь и далее мы будем считать термины «*поведенческая модель*» и «*конечный автомат*» синонимами, если не указано обратное.

Отрасль программной инженерии в целом испытывает увеличение темпов развития. Это обусловлено в первую очередь ростом информационных потребностей современных пользователей, повсеместной интеграцией сети Интернет в процессы жизнедеятельности, доступностью технологий разработки ПО.

Программный код очень быстро становится унаследованным (или англ., «legacy»), например, при скудном ведении документации в одной и той же команде разработчиков за несколько лет, или при смене команды разработчиков. Это порождает проблему понимания, освоения кода, который необходимо поддерживать или использовать.

В связи с ростом сложности ПО, возникла потребность в эффективном представлении его внутренней структуры. Данная потребность обычно удовлетворяется путем построения эквивалентных моделей, которые могут применяться как на стадии проектирования системы (то есть до непосредственной реализации), так и на стадии документирования уже реализованной системы.

Однако составление документации связано с задачей восстановления исходных требований, заложенных на этапе проектирования. Здесь удобно ввести термин *реверс-инжиниринга* как процесса воспроизведения некоторых характеристик, моделей или внутренней структуры программы по ее реализации. Таким образом, автоматическое восстановление моделей может быть использовано в качестве способа получения проектной документации.

Другой потребностью, связанной с растущей сложностью ПО, является обеспечение его качества. Использование моделей (таких, как КА) возможно в таких методах обеспечения качества ПО, как верификация на модели (автоматическое выделение верифицируемых моделей) и тестирование (оценка тестового покрытия, генерация тестовых данных).

В рамках данной работы разрабатывается метод выделения моделей конечных автоматов из исходного кода.

1.2 Задача восстановления КА

Термин «Конечный Автомат» в отечественной научной школе ассоциируется, в первую очередь, с построением дискретных устройств (как, например, в учебном пособии [2]). Это означает, что автомат является преобразователем цепочек входного алфавита в цепочки выходного.

В англоязычной литературе эквивалентное понятие – «Finite-State Machine», или «Automata» – имеет несколько более широкий смысл, объединяя класс моделей с конечными множеством состояний, множеством входных воздействий, и функцией перехода.

В рамках данной работы мы будем считать, что Конечный Автомат – это:

$$\{\Sigma, S, s_0, \delta, F\}$$

где Σ – это множество входных воздействий, S – это множество состояний, $s_0 \in S$ – это начальное состояние, $\delta : \Sigma \times S$ – это функция переходов, а $F \subset S$ – множество допустимых состояний. Более того, последний элемент этой пятерки мы будем, как правило, опускать, считая это множество пустым.

Введем определение программы. Пусть программа – это участок памяти, разделенный на память команд и память данных. Будем считать, что внешние воздействия передаются программе через память данных, а память команд является неизменяемой (во время исполнения). На данном этапе также положим, что размер памяти ограничен и с течением времени не меняется. Тогда множество состояний программы – это множество возможных состояний памяти данных. Модель КА обычно имеет не более 10–20 состояний, а память данных редко бывает менее 1 Кб. Следовательно, если в основу программы заложен КА, его состояниям соответствуют подмножества состояний программы. На практике это означает, что существует одна или несколько переменных, проверка состояния которых позволяет точно определить состояние, в котором находится программа.

Одним из возможных подходов реверс-инжиниринга, в таком случае, является анализ переменных, их значений и путей изменений значений.

С другой стороны, модель КА определяет, как программа реагирует на те или иные воздействия. Следовательно, противоположный предложенному выше подход реверс-инжиниринга содержится в анализе поведения программы по внешним признакам.

1.3 Классификация методов выделения моделей из ПО

Излагаемая ниже классификация практически полностью повторяет классификацию, изложенную в [3]. Несмотря на то, что предметная область цитируемой классификации – методы обеспечения качества ПО, она является достаточно общей для всех методов анализа ПО, и может быть принята в нашем случае с небольшими поправками.

По формализованности методы выделения моделей могут быть разделены на *формальные* и *неформальные*.

Несмотря на то, что выделяемая модель имеет строгое математическое описание (так как мы в данном случае говорим о КА), метод ее выделения может быть неформальным. Как пример формального метода можно привести поиск определенных связей в структуре исходного кода, как пример неформального метода – ручное выделение модели в процессе аудита (просмотра) кода.

По степени автоматизированности процедур методы выделения моделей можно разделить на *ручные*, *автоматизированные* и *автоматические*.

Ручные методы, как правило, в принципе не подлежат автоматизации, что часто является следствием невозможности задания четко формализованной модели.

Автоматизированные методы – это группа методов, которые состоят как из ручных действий (например, подготовки данных), так и из автоматических. Такие методы можно частично формализовать.

Автоматические методы используют полную автоматизацию процедур выделения моделей. На практике, выделение моделей редко бывает полностью ручным или полностью автоматическим. Как правило, требуется хотя бы простая, базовая информация, как, например, указание пользователем участка кода, предположительно содержащего выделяемую модель, или аннотирование переменных метками состояний.

По необходимости запуска методы делятся на *статические* и *динамические*. Классификацию по этому признаку мы рассмотрим подробнее в следующем подразделе.

1.4 Статический и динамический анализ

Анализ поведения программы можно производить на двух базовых уровнях: на уровне ее исходных кодов (без запуска программы) и на уровне трасс выполнения программы (с запуском программы). Эти уровни принято называть соответственно статическим и динамическим анализом. Возможно применять методы статического и динамического анализа совместно.

1.4.1 Применение статического анализа в рамках решаемой задачи

Статический анализ – это обобщенное название группы методов анализа ПО, ключевыми особенностями которых являются [3]:

- отсутствие необходимости запускать программу;
- формальность применяемых правил и методик;
- автоматизация процесса анализа.

Применение средств статического анализа для решения задачи выделения поведенческих моделей из реализации программ пока не имеет достаточной теоретической базы, которая бы позволила привести к общему знаменателю все существующие исследования. Тем не менее, общим местом являются анализ абстрактного синтаксического дерева программы (Abstract Syntax Tree, AST), графов потока управления и зависимостей данных (Control Flow Graph, CFG и Data Dependency Graph, DDG). Модель, объединяющая две вышеприведенные, – это модель абстрактного семантического графа (Abstract Semantic Graph, ASG), которая, однако, в силу своей сложности редко применяется на практике [3].

Разнородность подходов обусловлена в первую очередь глубокими различиями языков программирования. Редко один и тот же подход можно применить сразу к нескольким слабо связанным языкам. Как правило, исследователи ограничиваются выбором одного языка, например, C (C++) или Java. Данные языки выбираются исходя, во-первых, из их популярности (руководствуясь, например, рейтингом TIOBE [4]), и во-вторых, в следствие промышленной направленности таких исследований, то есть выполнением научно-исследовательской работы (НИР) в рамках реверс-инжиниринга моделей одного конкретного программного продукта.

Данный подход имеет недостаток, связанный с невозможностью решить задачу в общем случае, то есть может применяться только с введением некоторых ограничений.

1.4.2 Применение динамического анализа в рамках решаемой задачи

Динамический анализ – это анализ трасс выполнения программ, последовательной или иерархической информации о прохождении программой определенных контрольных точек, вызовов процедур, методов, их вложенности и так далее. Примитивным (то есть не автоматизированным) средством динамического анализа является отладчик.

С точки зрения поведения программы, трасса выполнения – это поток событий, которые могут иметь или не иметь отношения к переходам между состояниями выделяемого автомата.

Таким образом, задача выделения конечного автомата сводится к задаче вывода грамматики автомата, на основании некоторой обучающей последовательности. Следует однако отметить, что здесь прослеживается определенная особенность выделения именно конечных автоматов как поведенческих моделей, а именно наличие грамматики. Например, нельзя говорить о «допустимых цепочках» или «грамматиках» в случае таких поведенческих моделей, как Sequence Diagram (диаграмма последовательностей) и высокоуровневой StateChart Diagram (диаграмма состояний).

Такая задача, напротив, решается много лучше и формальнее (в отличие от предыдущего случая), так как она избавлена от подробностей уровня языка, и опирается на хорошо изученный математической аппарат модели конечного автомата. Подробнее о грамматике конечных автоматов можно прочитать в книге [1].

Существенный недостаток данного метода заключается в том, что анализируемая выборка трасс выполнения программы скорее всего, не является полной, а восстановленная модель будет отражать не полную реализацию, а только те пути исполнения, которые были затронуты при генерации выборки.

1.4.3 Комбинирование подходов

Избежать неточностей динамического анализа может помочь применение в паре с ним средств статического анализа, и наоборот, ограничения, накладываемые на статический анализ могут быть настраиваемыми за счет привнесения дополнительной информации, напрямую не содержащейся в исходном коде (например, через предварительный анализ трасс выполнения программы).

1.5 Существующие методы решения задачи

В данном подразделе мы рассмотрим существующие методы решения задачи реверс-инжиниринга ПО с целью выделения поведенческих моделей. Для того, чтобы обеспечить наглядность и формальность анализа этих методов, введем критерии качества решения задачи:

- Автоматизируемость – степень участия инженера-исследователя ПО при использовании метода. Данный критерий напрямую связан с затратами при использовании метода, потому что ручные методы требуют больших временных затрат, в сравнении с автоматизированными.

- Универсальность – мера адаптируемости метода к различным объектам анализа. Мерой универсальности можно считать количество усилий, требуемых для адаптации реализации к новому объекту анализа.

- Наглядность – мера наглядности получаемого результата. Целью данного проекта является решение задачи Code Understanding (понимания программного кода). Данный критерий призван оценить пригодность метода для решения указанной задачи.

- Формальность – мера формальности метода. Этот критерий тесно связан с критерием автоматизируемости, так как чем формальнее метод, тем большую степень автоматизации можно ввести.

Таким образом, данная работа направлена на получение автоматизированного универсального формального метода решения задачи об изучении программного кода (на примере выделения моделей КА).

Рассмотренные далее существующие решения содержат черты, как присущие разработанному методу, так и совершенно для нее не характерные.

1.5.1 Слайсинг и введение уровней абстракции

Авторы статьи [5] разработали средство Bandera для выделения моделей с конечным числом состояний для их последующей верификации в различных системах проверки моделей (как, например SPIN [6]). Основами предлагаемого метода являются «slicing», «слайсинг», то есть отсечение элементов реализации программы, не имеющих прямого отношения к выделяемой модели, и введение уровней абстракции, которые позволяют представлять сложные объекты как переменные с малым количеством состояний. Разработанное средство является интерактивным, то есть решения об уровне детализации и абстракции принимает пользователь.

Таким образом, основной целью применения данного инструмента является упрощение процедуры верификации модели КА, то есть обнаружения нежелательного поведения программы. Несмотря на то, что Bandera экипирован средствами статического анализа, адекватность модели напрямую зависит от инженера, использующего этот инструмент.

Данное решение отвечает критериям универсальности и формальности, но не полностью отвечает критерию автоматизируемости, так как предполагает интерактивную работу с инженером-исследователем, а также критерию наглядности, так как выделяемая модель используется для решения другой задачи, а именно автоматизации выделения модели для ее проверки.

1.5.2 Поиск шаблонов в коде

В статье [7] описан несколько иной подход. Авторы изначально ориентируются на выделение диаграмм КА из унаследованного кода некоторой встраиваемой системы, реализованной на C++. Их выбор обусловлен, главным образом, требованиями заказчика исследований. Авторы заметили, что часто проектируемый КА реализуется как «switch-внутри-switch», и предложили выполнять поиск таких конструкций в исходном коде. На момент написания статьи [7] поддержка разработанным программным средством других шаблонов только планировалась.

В качестве исходной модели при анализе программного кода используется AST, получаемый средствами CPP2XMI [8]. Данный инструмент также производит некоторую редукцию дерева. Используетмый формат для представления дерева – XML.

Как бы то ни было, такой подход оправдан в случае конкретного промышленного образца исходного кода. Он обладает малой масштабируемостью и, вероятно, ограничен внутрипроцедурным статическим анализом.

Поиск шаблонов по AST в целом обладает рядом недостатков:

1. Структура AST не отражает семантику программ, а использование шаблонов лишь позволяет интерпретировать смысл некоторых синтаксических конструкций.

2. Затраты на поддержку такого метода довольно высоки. Они определяются сложностью самой модели AST, и следовательно, сложностью шаблонов. (Поэтому, в частности, данный подход использует редукцию дерева до некоторого его подмножества.)

3. Метод зависит от количества шаблонов и размера кода. Таким образом, метод обладает ограничениями с точки зрения вычислимости.

Исходя из приведенных выше соображений, данная реализация обладает свойствами автоматизируемости и наглядности, но не удовлетворяет критерию универсальности, так как использует жестко заданные шаблоны исходного кода. Предложенный метод является достаточно формальным, так как оперирует точной моделью исходного кода – AST.

1.5.3 Эвристические абстракции

Альтернативным предыдущему вариантом является введение некоторых абстракций, позволяющих объединить элементы программы в систему, которая затем будет проанализирована как потенциально содержащая КА.

Такой подход встречается, например, в статье [9]. Данное исследование также направлено на извлечение диаграмм КА из исходного кода встраиваемых систем на языке С, и также тесно связано с заказчиком.

Авторы статьи предлагают выделять входные, выходные переменные, а также переменные, которые могут менять свое значение внутри исследуемого блока кода (или переменные «обратной связи»). Состояние автомата предлагается выражать как состояние некоторого набора переменных, или участок кода, соответствующий определенному условию.

Далее авторы приводят набор эвристических правил, которые позволяют выделять:

- внутренние и внешние *события*;
- *ограничения*;
- *действвия*;
- *состояния.*

Затем выполняется собственно построение диаграммы.

Сами авторы статьи [9] утверждают, что для успешного выделения автомата необходимо, чтобы разработчик хотя бы на ментальном уровне пользовался моделью конечного автомата.

Данное решение обладает всеми желаемыми свойствами, кроме свойства универсальности, которое обеспечивается пересмотром эвристических правил.

1.5.4 Применение нейронных сетей

Применение нейронных сетей доступно в рамках динамического подхода. Основанием в данном случае служит представление трасс выполнения программы как потока событий.

Определим ряд необходимых терминов.

Символом будем называть атомарное событие, которое может произойти в рамках модели конечного автомата.

Цепочкой будем называть входную последовательность символов.

Грамматикой будем называть множество допустимых цепочек КА.

Кластеризацией будем называть анализ внутренней структуры нейронной сети с целью выделения доменов.

В качестве примера мы приведем исследование [10]. Целью данного исследования является построение нейронной сети, распознающей грамматику КА. В основу метода заложено использование сети Элмана, которая является рекуррентной (то есть сети с обратной связью), с одним слоем скрытых нейронов.

Данная сеть, обладая информацией о предыдущем элементе последовательности (сохраняемом в рекуррентном скрытом слое) и о текущем элементе (находящемся на входах сети), может довольно точно предсказать следующий элемент в цепочке. Если количество узлов в скрытом слое минимально, то узлы сети соответствуют узлам грамматики (что подтверждается при кластеризации обученной сети).

Следует отметить, что обученная нейронная сеть становится не полным, а вероятностным отображением КА. В этом состоит главное отличие динамических методов от статических: первые являются приближенными, и сильно зависят от обучающих (и тестовых) данных, а вторые являются точными (в рамках принятых допущений).

В статье [11] решается та же задача, что и в рассмотренном выше исследовании. Авторы фокусируются на несколько другой, более практической предметной области – области выделения моделей уровня бизнес-логики приложений. Примером использования метода авторами приводится вывод модели обработки заявки в CRM-системе (англ. Customer Relationship Management).

В этой обширной статье приводятся три метода решения задачи о выводе грамматики КА:

1. алгоритмический;
2. эвристический (с использованием нейронной сети);
3. вероятностный (в терминах Марковских процессов).

Примечательно, что предложенный вариант решения с помощью нейронной сети авторами был признан как нестабильный и сложный в настройке, тогда как в статье [10] было показано, что возможно получить приемлемые результаты.

Вероятностный способ заключается в рассмотрении потока событий как Марковского процесса с неизвестными параметрами, то есть в нахождении этих параметров. Итоговая модель есть явно заданная матрица с весовыми коэффициентами (что также отсылает нас к упомянутому выше неявном вероятностном отображению модели в обученной нейронной сети).

Алгоритмический способ мы не будем детально рассматривать в текущем подразделе, так как далее он будет рассмотрен на примере другого исследования.

Приведенные решения достаточно универсальны, в рамках введенных исследователями ограничений, однако задача собственно выделения потока событий решается неформально и не универсально.

1.5.5 Восстановление сетевых протоколов

Одним из типов динамических трасс выполнения программы является срез ее сетевой активности. Такая информация используется, например, в [12] для выявления нежелательной сетевой активности приложений (таких, как botnet – анонимной распределенной вычислительной сети). В качестве базовой модели здесь используется КА протокола.

Первым шагом на пути анализа является кластеризация сообщений по типам, которая производится по эвристическому критерию «похожести» пакетов, а также реакции на эти сообщения. Когда все возможные сообщения сведены к конечному набору типов, последовательности сообщений обобщаются и выводится префиксное дерево. Данное дерево приводится к минимальному детерминированному КА с использованием известных алгоритмов (подробности см. [12]). Такая задача является NP-полной.

Авторами приводятся в качестве результатов работы выделенные диаграммы протоколов SMTP и SMB.

На примере данного подхода был рассмотрен алгоритмический способ выделения КА из динамической информации.

Приведенное решение является узкоспециальным в области выделения сетевых протоколов. Для этой области оно обладает всеми желаемыми свойствами, однако не может быть применено для анализа моделей, не имеющих отношения к сетевой активности приложения, то есть не выполнен критерий универсальности.

1.5.6 Комбинированное решение

В данном подразделе мы рассмотрим решение, использующее комбинацию статического и динамического подходов. Авторы исследования [13] фокусируются на выделении интерфейсов Объектно-Ориентированных (ОО) компонентов, используя в качестве базовой модели КА. Исследование ведется в рамках языка Java, главным отличием которого от таких языков, как C++, является полностью ОО-подход [14].

Модель выделяется для одной реализации интерфейса, и отражает последовательность вызовов методов класса. Модель имеет вид упрощенного КА, где узлы графа имеют смысл вызовов методов класса, а дуги отражают принципиальную возможность последовательного вызова двух методов.

Статический анализ в рамках данного подхода применяется для выделения невозможных последовательностей выполняется поиск источника и приемника, таких что источник – это присваивание полю класса некоторого константного значения x, а приемник – это выбрасывание исключения, условием для которого является равенство поля класса значению x.

Динамический анализ применяется для выделения возможных последовательностей вызовов методов класса. Для этого выполняется инструментализация байт-кода, которая позволяет получить такую информацию.

Данное решение обладает всеми желаемыми свойствами, кроме универсальности. Не универсальным данное решение делают следующие черты.

- Применимо только для языка Java.
  - Генерация трасс возможна только для языков, компилируемых в байт-код Java.
  - Извлечение модели возможно только для случая ОО-подхода при проектировании ПО.
- Выделяются только КА, построенные по принципу «1 состояние – 1 метод».

Тем не менее, из всех рассмотренных средств данное решение представляется наиболее сильным, так как промышленная апробация метода велась более широко, и производилась на элементах стандартной библиотеки Java.

1.5.7 Сравнительный анализ методов

Резюмируем соответствие введенным критериям рассмотренных методов в виде таблицы 1.1.

Использование статического анализа, как правило, ограничивает область выделения КА до программного кода, содержащегося внутри процедур. Методы, использующие статический анализ, лучше формализуются и обладают большей универсальностью.

Динамический анализ, напротив, чаще всего ограничивается информацией о последовательности вызовов процедур (или методов), так как информация о прохождении каждого оператора программы

Таблица 1.1
Сравнительный анализ методов

Метод	Автом.	Универс.	Наглядн.	Формальн.
Слайсинг и введение уровней абстракции	±	+	±	+
Поиск шаблонов в коде	+	−	+	−
Эвристические абстракции	+	−	+	+
Применение нейронных сетей	±	±	+	±
Восстановление сетевых протоколов	+	±	±	+
Комбинированное решение	+	−	+	±

слишком разнородна и обильна для последующего успешного анализа. Методы, использующие динамический анализ, формализуются не полностью, и целиком зависят от способа извлечения динамической информации.

Критерий, которому хуже всего удовлетворяют рассмотренные средства, это критерий универсальности. Вероятнее всего, это происходит потому, что не существует конечного способа решения задачи о синтезе КА средствами универсального языка программирования, и критерием универсальности пренебрегают для получения большей робастности получаемых результатов (точности и полноты модели).

Таким образом, ни один из рассмотренных подходов не отвечает всем рассмотренным критериям полностью.

1.6 Средства реверс-инжиниринга Java-программ

В данном подразделе рассмотрим средства реверс-инжиниринга моделей, заложенных в основу ПО. Для этого введем следующие критерии сравнения.

- *Целевой язык программирования.* Данный критерий необходим, потому что реверс-инжиниринг – задача, зависящая от языка реализации. Будем рассматривать программные средства, поддерживающие язык Java.

- *Возможность выделения поведенческих моделей. Типы выделяемых моделей и диаграмм.* Поведенческими моделями будем считать КА, диаграммы последовательностей, дерево вызовов.

В ходе составления этого обзора средств, позволяющих выделять из исходного кода модели КА обнаружено не было. Данный обзор предназначен для создания представления о современных средствах реверс-инжиниринга программ.

1.6.1 Borland Together

Borland Together [15] – продукт компании Borland, являющийся средством документирования ПО, средством разработки требований и ПО. Основной идеей является объединение разработки ПО на всех уровнях, начиная с бизнес-процессов и заканчивая шаблонами проектирования.

Продукт поддерживает различные стандарты моделирования ПО (например, UML), различные языки программирования (в том числе, Java и C#). Позволяет версионировать, сравнивать, комбинировать разрабатываемые модели. Поддерживает ER и IDEF1 диаграммы для моделирования данных, представляет возможность проектирования и реверс-инжиниринга для наиболее популярных СУБД (Oracle™, MS SQL Server™).

Продукт позволяет вести разработку на основе моделей за счет генерации кода, поддержки шаблонов проектирования, средств создания и повторного использования собственных шаблонов, а также с помощью возможностей импорта и экспорта моделей из различных форматов и источников (XMI 2.0, Rose, XDE).

Together позволяет моделировать бизнес-процессы в нотации BPMN, с возможностью валидации, импортировать и экспортировать модели бизнес-процессов для веб-сервисов (BPEL4WS). Продукт интегрируется в Eclipse IDE.

1.6.2 Understand

Understand [16] – продукт, позволяющий решать задачи понимания программного кода (*code understanding*). Поставляется в формате среды разработки (IDE, Integrated Development Environment), включающей текстовый редактор, файловый браузер, а также средства реверс-инжиниринга.

Продукт поддерживает большой спектр языков программирования. Среди них Java, C, Ada и другие. Позволяет производить анализ межфайловых зависимостей.

Understand позволяет автоматизировать процесс соответствия программного кода известным стандартам кодирования (как, например, MISRA-C), или определить свои собственные стандарты.

Продукт позволяет упростить и визуализировать процесс документирования кода посредством графов декларации процедур, дерева вызовов, потока управления, диаграмм классов и пакетов, а также вводить собственные типы диаграмм.

Возможны также вычисление метрик сложности и объема исходного кода. Единый интерфейс для расширений возможностей продукта предоставляется посредством API на языке Perl, который позволяет формулировать запросы к базе данных фактов об исследуемой программе.

1.6.3 WithClass

WithClass [17] – это продукт компании MicroGold, позволяющий вести проектную документацию в формате UML-диаграмм, обладающий также возможностями реверс-инжиниринга диаграмм классов для поддерживаемых языков.

Продукт предоставляет множество средств для настройки представления диаграмм (способ рендеринга, шрифты, цвета). Поддерживает все основные типы диаграмм из группы стандартов UML 1.x: диаграммы классов, примеров использования, конечных автоматов, последовательностей, взаимодействия, диаграммы компонентов и диаграмм поставки (deployment).

WithClass поддерживает множество языков программирования: C++, Java, C#, Visuak Basic, Delphi. Позволяет генерировать код по шаблонам, написанным на специальном проприетарном языке, разработанном компанией MicroGold, также позволяет создавать собственные сценарии генерации кода. Среди доступных шаблонов есть шаблон конечного автомата.

1.6.4 Imagix 4D

Imagix 4D [18] – продукт, позволяющий инструментализировать освоение унаследованного исходного кода. Поддерживаются следующие языки программирования: C, C++, Java.

Продукт выполняет точный и полный анализ семантики программы. Точность достигается использованием для получения информации об исходном коде через те же компиляторы, которыми осуществляется сборка ПО. Ввиду объемности получаемой информации для ее хранения используется специализированная база данных.

Продукт позволяет получать диаграммы классов, зависимостей по данным, граф потока управления программы и другие. Одной из ключевых возможностей являются взаимные связи моделей высокого уровня (как, например, диаграммы классов) с моделями низкого уровня (как, например, граф потока управления).

Imagix 4D производит анализ потока данных, что позволяет отследить все цепочки «определения-и-использования» («def-use chains»), причем результат представляется в виде специальной диаграммы.

Также позволяет получать некоторые метрики программного кода и осуществляет поиск потенциальных дефектов программного кода средствами статического анализа.

1.6.5 yDoc

yDoc [19] – это расширение технологии Javadoc, позволяющее встраивать в процесс генерации Java-документации реверс-инжиниринг UML-диаграмм. Данный продукт распространяется компанией yWorks по лицензии «shareware» с возможностью обработки до 10 пользовательских классов, или по коммерческой лицензии с отсутствием такого ограничения.

Продукт позволяет извлекать следующие UML-диаграммы: диаграммы структуры пакетов, диаграммы наследования (с учетом межпакетных связей), диаграммы классов. Для построения диаграмм используются алгоритмы автоматического расположения элементов диаграммы из библиотеки yFiles.

Возможна настройка стиля диаграмм, списка обрабатываемых классов, способа рендеринга изображений (типа графических файлов). Выделяемые диаграммы содержат ссылки на компоненты программы. Поддерживается ряд особенностей языка Java версии 5.0.

Использование продукта может производиться на основе JDK (Java Development Kit, пакет разработчика на Java) из командной строки, может быть интегрировано в процесс сборки проекта, основанного на ANT, или разрабатываемого в IDE (Eclipse, Idea, NetBeans) благодаря возможностям пользовательских расширений технологии Javadoc.

1.6.6 Сравнительный анализ средств реверс-инжиниринга

Резюмируем проведенный обзор средств реверс-инжиниринга в таблице 1.2

Таблица 1.2

Сравнительный анализ средств реверс-инжиниринга

Средство	**Языки программирования**	**Выделяемые диаграммы**
Borland Together	Java, C#, C++, Delphi	Диаграммы классов, последовательностей
Understand	Java, C, Ada	Диаграммы классов, зависимостей файлов, граф потока управления
WithClass	Java и байт-код, C++, C, C#, Visual Basic	Диаграммы классов
Imagix 4D	Java, C, C++	Диаграммы классов, зависимостей по данным, граф потока управления
yDoc	Java	Диаграммы классов, пакетов, наследования

Исходя из изложенных данных, можно сделать вывод, что наиболее востребованным является реверс-инжиниринг диаграмм классов, так как он поддерживается всеми рассмотренными средствами.

С другой стороны, граф зависимостей по данным встретился только один раз, в продукте Imagix 4D, который также позволяет средствами статического анализа обнаруживать некоторые программные дефекты. Выделение такого рода структур данных необходимо для статического анализа. Таким образом, предоставление графических диаграмм – это скорее побочный эффект.

Наконец, ни одно средство не позволяет выделять КА из исходного кода. Вероятно, это обусловлено тем, что на данный момент не существует универсального метода реверс-инжиниринга модели такого рода.

1.7 Выводы

В данной главе рассмотрены вопросы, связанные с проблемой выделения КА из реализации программы, а также несколько путей ее решения. Было введено понятие КА, статического и динамического анализа, представлены критерии анализа существующих методов решения задачи, которые отражают ожидаемый набор качеств разрабатываемого метода выделения КА из программного кода, а также ключевые отличия от существующих методов. Приведен обзор существующих программных решений реверс-инжиниринга программ.

2 ПОСТАНОВКА ЗАДАЧИ ВЫДЕЛЕНИЯ ПОВЕДЕНЧЕСКИХ МОДЕЛЕЙ ИЗ ИСХОДНОГО КОДА И ВЫБОР ПУТИ РЕШЕНИЯ

В данном разделе формулируются требования к разрабатываемому методу, производится обоснование и выбор целевого языка программирования, производится выбор пути решения.

2.1 Постановка задачи

В рамках данного диссертационного исследования необходимо разработать метод РИ программного кода, позволяющий выделять модель КА.

В качестве исходных данных должны использоваться исходные коды программ, которые следует преобразовывать к представлению AST.

Основным принципом метода следует принять анализ состояний и переходов между состояниями переменных программы. Для выделения переменных, отражающих состояние программы, следует прибегать к их аннотированию в программном коде. При разработке прототипа системы можно ограничиться одной такой переменной.

Анализ переходов между состояниями следует производить на внутрипроцедурном уровне, в качестве модели данных для анализа следует использовать граф потока управления.

Для каждой процедуры следует строить отдельный граф КА, и в графическом виде выводиться его диаграмма. Представление выходных данных может быть как графическим, так и любым другим, легко преобразуемым к графическому.

2.2 Выбор пути решения

2.2.1 Выбор целевого языка программирования

При выборе целевого языка программирования следует выбрать такой язык, для которого наиболее актуальна задача реверс-инжиниринга с целью выделения поведенческих моделей. Однако не существует статистики, согласно которой модель конечного автомата реализуется на одном языке чаще, чем на другом. Поэтому будем опираться на обобщенный рейтинг популярности языков, такой, как TIOBE [4].

По данным за май 2011 года, он имеет следующий вид (приведенные величины отражают относительную популярность) :

1. Java — 18.160%;
2. C — 16.170%;
3. C++ — 9.146%;
4. C# — 7.539%;
5. PHP — 6.508%.

Таким образом, наиболее популярным языком является Java. Его и предлагается выбрать в качестве целевого языка.

2.2.2 Используемые алгоритмы и структуры данных

В качестве входных данных будем использовать AST, построенный по исходному коду исследуемого ПО. Получение AST следует производить с помощью интерфейса доступа к компилятору.

Для аннотирования переменных состояния будем использовать технологию Java Annotations [20]. Аннотации будут иметь смысл специального маркера, отражающегося в AST программы.

Алгоритм построения внутрипроцедурного CFG будет разработан самостоятельно. При разработке будут рассматриваться следующие типы конструкций языка Java:

- циклы (*for*, *while*, *do-while*);
- операторы ветвления (*if*, *switch*);
- операторы безусловного перехода (*break* и *continue* без меток, *return*);
- последовательности операторов (в том числе *try* без *catch*);
- простые операторы.

Многопоточность, исключения, java reflection при построении внутрипроцедурного CFG учитываться не будут.

Множество возможных состояний аннотированной переменной будет редуцироваться до множества возможных константных значений, которые могут быть присвоены этой переменной.

Множество возможных переходов между состояниями будет определяться посредством анализа внутрипроцедурного CFG. Это множество будет отображаться в граф КА.

Синтез диаграмм КА будет производиться по их текстовому описанию в формате DOT [21] с помощью пакета программ GraphViz [22].

3 РАЗРАБОТКА ТЕХНОЛОГИИ ИЗВЛЕЧЕНИЯ МОДЕЛЕЙ ИЗ ИСХОДНОГО КОДА

В соответствии с поставленной задачей, необходимо разработать технологию извлечения моделей КА из исходного кода путем анализа конечного множества состояний одной переменной исследуемого Java-класса, и переходов между состояниями, и генерации графической диаграммы конечного автомата.

В данном разделе излагаются основные принципы, заложенные в разработанную технологию: аннотирование переменной состояния, выделение конечного множества возможных значений, анализ внутрипроцедурного графа потока управления для нахождения переходов между состояниями, формат выходных данных.

3.1 Процесс извлечения диаграммы КА

Процесс автоматизированного извлечения диаграммы КА представлен на рисунке 3.1. Все этапы этого процесса, кроме составления текста аннотаций, являются автоматическими. Поясним более детально каждый этап процесса.

Первым этапом процесса является внесение в текст программы аннотаций. Предусмотрены два типа аннотаций: маркировка поля класса как переменной состояния и маркировка метода класса как процедуры с побочными эффектами.

Следующим этапом является вызов компилятора Java, которому передается также специализированный процессор аннотаций. Результатом работы компилятора является AST программы, а результатом работы процессора аннотаций – списки узлов AST, отмеченных аннотациями.

Полученная на предыдущем этапе информация направляется в модуль выделения КА, результатом работы которого является диаграмма на языке DOT [22].

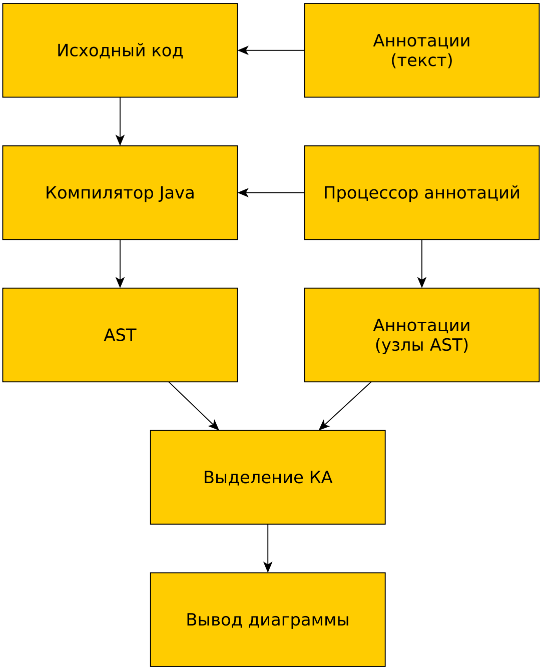

Рисунок 3.1. Схема процесса извлечения диаграммы КА

3.2 Технология аннотирования исходного кода

3.2.1 Технология Java-аннотаций

Язык Java обладает встроенными возможностями внесения мета-информации в исходный код [20]. Данная технология состоит из следующих компонент:

1. синтаксис определения типов аннотаций;
2. синтаксис аннотирования;
3. API для работы с аннотациями;

4. способ помещения аннотаций в скомпилированный java-файл (имеющий расширение *.*class*);

5. утилита обработки аннотаций.

Аннотации не влияют напрямую на семантику программы, однако влияют на способ взаимодействия библиотек с программой. В нашем случае, аннотации позволяют искать модель КА для определенного набора переменных состояния.

Обработка аннотаций может производиться в момент компиляции программы (тогда возможна генерация кода в зависимости от аннотаций), статически по скомпилированному коду или в процессе выполнения через механизм объектного отражения внутренней структуры приложения (называемый «*reflection*»). В нашем случае входными данными системы являются исходные коды, поэтому обработка аннотаций производится в момент компиляции программы.

Примеры определения типов аннотаций и их использования приведены в [20] и далее, в подразделе 4.2.

3.3 Извлечение информации о состояниях

3.3.1 Процесс извлечения информации о состояниях

Процесс извлечения информации о состояниях проиллюстрирован на рис. 3.2.

Информация о возможных состояниях используется при анализе допустимости дуг, изначальное состояние которых неизвестно.

3.3.2 Извлечение множества возможных состояний

Базируясь на предположении, что состояние автомата не может быть, в классическом смысле, вычислено, а может быть только загружена, а также на самой модели КА, допускающей только конечное множество состояний, можно придти к выводу, что множество возможных состояний переменной, имеющей смысл переменной состояния автомата, является множеством всех константных значений, которые могут быть присвоены этой переменной. Приведем алгоритм получения такого множество на рис. 3.3.

Данный алгоритм позволяет проанализировать все присваивания переменной состояния. Если среди присваиваний найдется такое, что

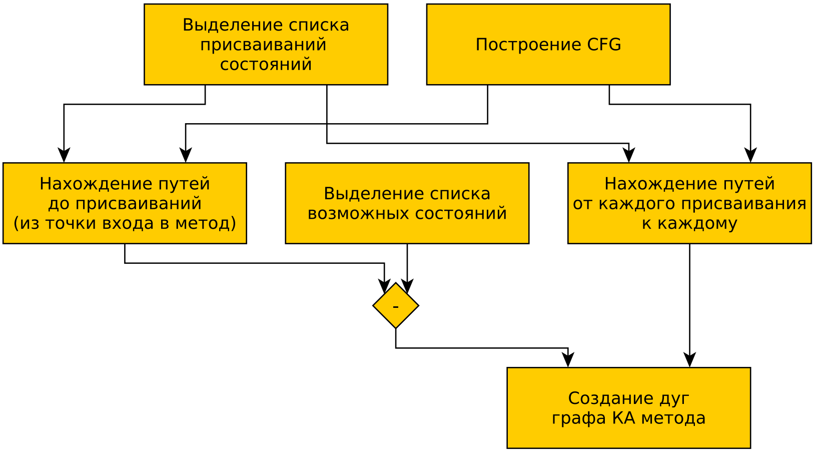

Рисунок 3.2. Схема процесса извлечения информации о состояниях КА

Вход: S – Имя переменной состояния
Вход: M – Множество методов класса
Выход: V – Множество возможных состояний переменной S

$V \leftarrow \emptyset$
for $m \in M$ **do**
  for $s \in m$ **do**
    if $\textsc{match}(s, S)$ **then**
      $V \leftarrow V \cup right(s)$
    end if
  end for
end for
procedure $\textsc{match}(s, S)$
  return $isAssignment(s) \wedge left(s) = S \wedge right(s) \in (Const, Literal, EnumElement)$
end procedure

Рисунок 3.3. Представление алгоритма извлечения возможных присваиваний констант на псевдокоде

$right(s) \notin (Const, Literal, EnumElement)$, значение, получаемое в правой части присваивания, окажется не проанализированным. Таким образом, приведенный алгоритм не является полным, однако такое качество от него не требуется.

3.3.3 Извлечение точек присваивания состояний

Точка присваивания переменной состояния – это инструкция, в которой автомат выполняет переход из одного состояния в другое (или сохраняет его). Если предположить, что в других инструкциях метода состояние не изменяется, то выделение точек изменения состояния можно произвести с помощью алгоритма, представление которого на псевдокоде приведенного на рис. 3.4.

Вход: S – Имя переменной состояния
Вход: M – Множество операторов метода
Выход: Q – Множество присваиваний состояний

$Q \leftarrow \emptyset$
for $s \in M$ **do**
    if $\textsc{match}(s, S)$ **then**
        $Q \leftarrow Q \cup s$
    end if
end for

Рисунок 3.4. Представление алгоритма извлечения присваиваний состояния на псевдокоде

Для того, чтобы отделить задачу обхода дерева от обработки его узлов, в данном и предыдущем случаях вводится допущение, что известно множество всех операторов (*statement*) метода.

3.3.4 Построение внутрипроцедурного CFG

Перед тем, как будет введен алгоритм построения внутрипроцедурного CFG, введем абстракцию, отражающую узел дерева AST.

$$T = (id, k, l)$$

Здесь T – это узел дерева, id – уникальный идентификатор узла, $k \in K$ – значение из множества типов узлов, l – список поддеревьев. l может быть пустым, если узел является листом дерева.

Введем также абстракцию, отражающую узел графа потока управления.

$$C = (id, c, n, a)$$

Здесь C – это узел графа, id – уникальный идентификатор узла, c – множество условий, проверяемых внутри узла (может быть пустым), n – следующий узел, переход к которому происходит при выполнении условия $c \vee c = \emptyset$, a – узел, переход к которому происходит при выполнении условия $\neg c \wedge c \neq \emptyset$.

procedure BUILDCFG($tree, parent$)
    if $tree$ – простой оператор **then**
        $c \leftarrow$ новый узел CFG
        задать $parent$ следующий узел – c
        return c
    else if $tree$ – оператор со сложной структурой **then**
        $c \leftarrow$ результат обхода структуры оператора
        задать $parent$ следующий узел – c
        return c
    end if
end procedure

Рисунок 3.5. Представление алгоритма построения внутрипроцедурного CFG на псевдокоде

Представление алгоритма построения внутрипроцедурного CFG приведено на рис. 3.5. Данный алгоритм является рекурсивным и построен по принципу обхода иерархической структуры дерева от его основания к листьям. Нет большой необходимости пояснять, как именно производится обход операторов со сложной структурой, т.к. это будет детально рассмотрено в подразделе 4.5. Следует однако отметить, что в данном алгоритме также неявно используется восходящий анализ иерархической структуры (от листьев к корню), для определения точек слияния путей, например, при обходе *break* или *return*.

3.3.5 Извлечение функции переходов

Извлечение функции переходов производится путем заполнения матрицы $S \times S$, где S – множество состояний конечного автомата. Если переход между двумя состояниями отсутствует, соответствующая ячейка остается пустой. Определение наличия перехода осуществляется с помощью алгоритма, решающего задачу поиска пути на графе по принципу поиска в глубину. Если переход имеет место, то в матрицу заносится условие. Представление алгоритма обхода CFG на псевдокоде приведено на рис. 3.6.

Вход: S – множество состояний КА
Вход: $root$ – точка входа в метод
Вход: $stop$ – точка выхода из метода
Вход: V – множество точек присваивания состояний
Выход: G – множество дуг графа КА

$G \leftarrow \emptyset$
$M \leftarrow$ FIND$(root, stop)$ ▷ Пути, не содержащие изменение состояния
for $v \in V$ **do**
    $M \leftarrow M \cup$ FIND$(root, v)$
    for $w \in V$ **do**
        $M \leftarrow M \cup$ FIND(v, w)
    end for
end for
for $m \in M$ **do**
    if *Путь m не содержит противоречивых условий* ***then***
        $G \leftarrow G \cup (getFrom(m), getTo(m), reduceConditions(m))$
    end if
end for

Рисунок 3.6. Представление алгоритма выделения дуг графа КА на псевдокоде

Каждый выделенный путь преобразуется к следующей тройке:

$$G = (from, to, cond)$$

где $from$ – исходное состояние, to – конечное состояние, $cond$ – условие перехода. Исходное состояние может быть неизвестно, или, иначе говоря, может принимать множество значений (например, точка входа в метод). Будем исходить из предположения, что при выполнении любого фрагмента программы состояние автомата может быть одним из допустимых (об этих состояниях говорится в подразделе 3.3.2). Следовательно, каждый такой путь следует разбить на множество путей, мощность которого совпадает с мощностью множества возможных состояний, и исследовать каждый такой путь отдельно на наличие в нем условий, противоречащих исходному состоянию пути. Иначе говоря, если путь содержит условие $s = S1$, то он не имеет смысла, если $from \neq S1$.

3.4 Вывод графа КА

Вывод графа конечного автомата производится для каждого метода класса, в котором изменяется состояние выбранной переменной (то есть $G \neq \emptyset$), путем отображения элементов из множества G в описание на языке DOT. Грамматика языка DOT приведена в [21]. Данный язык позволяет описывать структуру и способ визуализации графа в виде текстового файла. Пакет программ Graphviz [22] позволяет автоматически построить графическое представление по заданному текстовому описанию на языке DOT. На рис. 3.7 приведен пример описания некоторого графа на языке DOT, на рис. 3.8 – его графическое представление, полученное с помощью GraphViz.

```
digraph A2{
    3
    2
    1
    3 -> 2          [label="A"]
    2 -> 1          [label="A"]
    1 -> 3          [label="A"]
    1 -> 2    [label="B"]
}
```

Рисунок 3.7. Пример описания на языке DOT

3.5 Ограничения предложенной технологии

Данная технология обладает следующими ограничениями.

1. *Анализ только примитивных или перечисляемых типов.* Данное ограничение не позволяет проанализировать автоматные

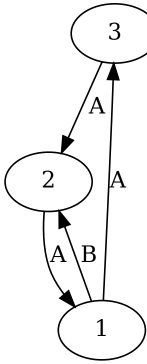

Рисунок 3.8. Пример графического файла, созданного с помощью DOT и GraphViz

программы, состояние которых хранится в ином виде.

2. *Анализ только прямых присваиваний констант.* Очевидно, что изменение значения переменной может происходить также и в других случаях, например, как присваивание результата вызова процедуры, или через составные операторы присваивания. Это ограничение может быть частично снято путем внесения дополнительных проверок в процедуру $MATCH()$, приведенную на рис. 3.3.

3. *Анализ только внутрипроцедурного CFG.* Это ограничение не позволяет корректно проанализировать автоматные программы, изменение состояния которых может происходить в нескольких методах.

Часть из приведенных ограничений могут быть частично устранены. Эти ограничения поставлены на этапе анализа исходной задачи с тем, чтобы получить базис для ее решения. Первое ограничение, вероятно, не столь существенно, потому что наиболее естественно решать задачу о синтезе КА именно с использованием констант и встраиваемых типов, или перечисляемых типов. Второе ограничение с практической точки зрения более значимо, однако даже в этом случае предложенная технология имеет смысл. В самом деле, предлагаемое решение базируется на «закрашивании» узлов CFG, в которых изменяется состояние, больше, чем на выборе «закрашиваемых» узлов.

Последнее ограничение является наиболее существенным, однако и наиболее труднопреодолимым. Тем не менее, можно предложить следующие способы: выполнение подстановки графов CFG друг в друга, согласно схеме взаимного вызова методов, и введение некоторых дополнительных атрибутов узлам CFG (например, отражающих степень влияния на состояние КА).

3.6 Выводы

В данном разделе представлена технология извлечения моделей КА из исходного кода путем анализа конечного множества состояний одной переменной Java-класса, и переходов между состояниями, и генерации графической диаграммы конечного автомата. Приведены основные принципы построения технологии и алгоритмы: способ аннотирования переменных; выделение конечного множества состояний; построение внутрипроцедурного CFG; анализ переходов между состояниями; способ создания графической диаграммы КА.

Рассмотрены ограничения предложенного метода, причины их возникновения и способы их устранения.

4 РЕАЛИЗАЦИЯ ПРОТОТИПА СИСТЕМЫ

В данном разделе рассматривается реализация прототипа системы извлечения модели конечного автомата из исходного кода путем анализа конечного множества значений одной переменной Java-класса.

В подразделе 4.1 приведена иерархическая структура пакетов разработанной системы. В подразделе 4.2 изложен способ аннотирования переменной состояния и обработки аннотаций. Подраздел 4.3 посвящен реализации алгоритма выделения множества возможных состояний. В подразделе 4.4 изложен способ реализации алгоритма выделения точек присваивания состояний. Подраздел 4.5 посвящен описанию реализации алгоритма построения внутрипроцедурного CFG. В подразделе 4.6 изложены детали реализации обхода графа потока управления. Подраздел 4.7 посвящен реализации алгоритмов анализа выражений, используемых для описания условий переходов в КА. В подразделе 4.8 описан принцип вывода графа выделенного КА.

4.1 Иерархическая организация системы

Реализованная система имеет следующую структуру пакетов.

- Пакет *botoo* является корневым. Он содержит следующие классы:
  - *botoo.Botoo* – класс, содержащий обработку аргументов командной строки, вызов компилятора Java со специфическим процессором аннотаций.
  - *botoo.StateAnnotationProcessor* – класс, являющийся реализацией специфического процессора аннотаций. Управление порядком реверс-инжиниринга находится в этом классе.
- Пакет *botoo.annotate* содержит определения типов аннотаций.
- Пакет *botoo.flownode* содержит определение интерфейса и нескольких реализаций узла CFG, а также алгоритмов построения и обхода этого графа.

- Пакет *botoo.graph* содержит определения классов узла и ребра графа КА.
- Пакет *botoo.samples* содержит демонстрационные примеры реализации конечных автоматов.
- Пакет *botoo.scanner* содержит определения сканеров AST.
- Пакет *botoo.scanner.base* содержит определение одного базового класса-сканера, позволяющего получать в результате обхода дерева список узлов.
- Пакет *botoo.tree* содержит реализации узлов AST, используемые при построении и редукции условий перехода между состояниями.
- Пакет *botoo.util* содержит несколько классов общего назначения.
- Пакет *botoo.visitor* содержит определения классов-посетителей узлов AST, позволяющих выполнять одинаковые операции для разных типов узлов.

4.2 Система аннотирования

Определение типов аннотаций, позволяющего помечать переменные состояния и процедуры с эффектом приведено на рис. 4.1. Процедурой с эффектом будем называть такие процедуры, информацию о вызовах которых следует отображать на диаграмме КА. Пример использования аннотаций приведен в приложении Б.1.

```
package botoo.annotate;

public @interface StateVariable {

}
public @interface SideEffectSubroutine {

}
```

Рисунок 4.1. Типы аннотаций переменной состояния и процедур с эффектами

4.3 Выделение множества возможных состояний

Выделение множества возможных состояний выполняется классом `botoo.scanner.ClassFinalFieldsScanner`, который является расширением реализации класса `TreePathScanner`. Последний позволяет производить произвольный обход дерева AST, с сохранением текущего пути (набора узлов AST от корня до текущего). Расширение этого класса заключается в задании пользовательских методов обхода конкретных узлов; дерево же в целом будет обходиться автоматически.

Реализация рассматриваемой функции заключается в сборе всех деклараций константных переменных, являющихся полями любого класса, входящего в область рассмотрения.

Следует отметить, что перечисляемые типы в версии Java 1.6 реализуются как классы с константными полями, следовательно разработанная реализация позволяет обнаруживать возможные состояния, описанные в том числе перечисляемыми типами.

Для того, чтобы учитывать только константные поля определенного типа, реализован класс `botoo.util.TreeCheck`, позволяющий сравнивать типы двух узлов AST, содержащих определение переменной.

4.4 Выделение множества присваиваний состояний

Выделение множества присваиваний указанной переменной состояния константных значений осуществляется с помощью класса `botoo.scanner.StateChangeScanner`. Как и в предыдущем случае, данный класс является расширением реализации сканера AST.

Реализация рассматриваемой функции заключается в обходе всех узлов AST, являющихся присваиваниями. Из этого множества выделяются те присваивания, в правой части которых находятся следующие выражения:

1. *константное* – идентификатор из списка возможных состояний;

2. *перечисляемое* – обращение к полю перечисляемого типа;

3. *литеральное* – константа, указанная в исходном коде явно.

В результате обхода дерева с помощью `StateChangeScanner` также актуализируется множество возможных состояний: в него добавляются литеральные константы, и удаляются состояния, присваивания которых не были обнаружены.

4.5 Построение внутрипроцедурного CFG

Построение внутрипроцедурного CFG базируется на использовании шаблона проектирования «visitor», посетитель, который позволяет единообразно обрабатывать полиморфные объекты. Обобщенный алгоритм построения CFG приведен на рис. 3.5.

Класс `com.sun.source.util.SimpleTreeVisitor` является обобщенной реализацией этого шаблона для элементов AST. При использовании этого класса в качестве базового, необходимо указать параметры шаблона: тип объектов-параметров и тип объектов-возвращаемых значений при обработке узла.

В рамках такого подхода удобно в качестве параметров и выходных значений использовать узлы графа потока управления. В самом деле, каждая программная конструкция[1] имеет точки входа и точки выхода из нее. Точки входа можно передавать в процедуру анализа программной конструкции в качестве параметра, а точки выхода передавать в качестве возвращаемого значения процедуры. Наконец, анализ программных конструкций следует производить рекурсивно.

4.5.1 Типы узлов CFG

Интерфейс узла графа потока управления описывается классом `botoo.flownode.FlowNode`, и изображен на рис. 4.2.

Данный интерфейс требует от его реализации, чтобы узел CFG был связан с узлом AST, имел ссылки на следующий и альтернативный узел CFG, позволял получить условие, проверяемое в узле, а также эффект, порождаемый узлом. Класс `botoo.flownode.Stub` является базовой реализацией этого интерфейса, и удовлетворяет всем его требованиям.

Сложные программные конструкции могут иметь несколько точек выхода. Объединение этих точек в одну часто называют ϕ-функцией (как, например, в [3]); в диаграммах далее мы будем помечать такие точки как fi. Реализация ϕ-функции представлена классом `botoo.flownode.CombineStub` (исходный код в приложении A.2).

В целях соответствия выбранному алгоритму построения CFG, требуется также узел, позволяющий «поменять местами» прямую и

[1] Здесь под программной конструкцией понимается узел AST, который имеет четко определенную структуру и может содержать другие программные конструкции.

```java
package botoo.flownode;

import com.sun.source.tree.ExpressionTree;
import com.sun.source.tree.Tree;

public interface FlowNode {

    public enum Type{
        DEFAULT,
        BREAK,
        RETURN,
        CONTINUE
    }

    Tree getTree();

    String getName();

    FlowNode getNext();

    FlowNode getAltNext();

    Type getType();

    ExpressionTree getConditional();

    String getEffect();
}
```

Рисунок 4.2. Интерфейс узла CFG

альтернативную ветвь исполнения. Реализация такого узла находится в классе `botoo.flownode.AltNextStub`.

4.5.2 Анализ сложных операторов

На рис. 4.3 а) приведена схема анализа оператора *if*. Здесь *parent* – это узел, предшествующий текущему, *cond* – проверяемое в if условие, *stmt1* – набор инструкций, который будет выполнен при выполнении условия, *stmt2* – набор инструкций, который будет выполнен в противном случае, а *fi* – узел, объединяющий две ветви в одну. Альтернативная ветвь может отсутствовать, а любой из *stmt1* и *stmt2* может в свою очередь быть составным оператором, в том числе – if.

Схема анализа оператора цикла *for* приведена на рис. 4.3 б). По сравнению с предыдущим случаем, есть несколько отличиев. Во-первых, выход из цикла осуществляется по альтернативной ветви узла

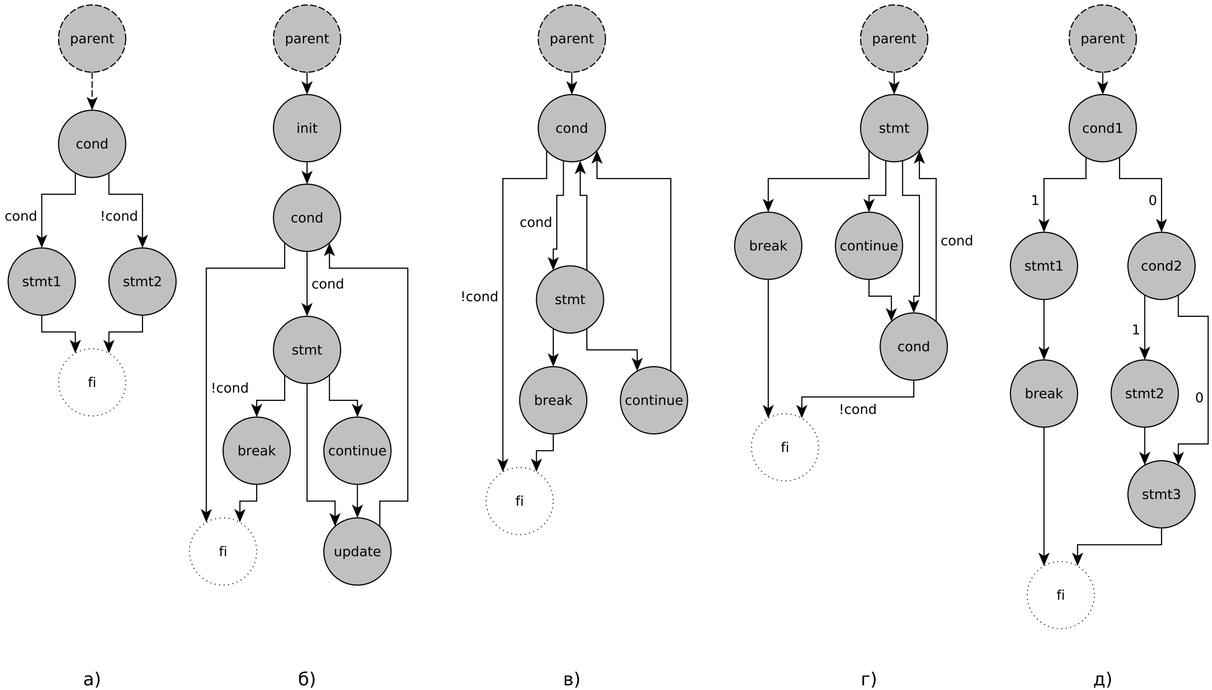

Рисунок 4.3. Схемы анализа сложных операторов: а) if, б) for, в) while, г) do-while, д) switch.

проверки условия. Во-вторых, возможно, что среди операторов внутри цикла встретится *break*, который порождает дополнительный путь выхода из цикла. В-третьих, возможно появление оператора *continue*, который создает дополнительную дугу для перехода на изменение счетчика цикла (*update*). Эти особенности легко реализовать с помощью введенных ранее типов узлов графа. Для корректного объединения путей необходимо «поменять местами» прямой и альтернативный пути узла проверки условия *cond*. Для этого создается промежуточный узел – объект класса `AltNextStub`.

Решение задачи об объединении путей от узлов *break*, находящихся внутри цикла, осложняется тем, что циклы могут быть вложенными, и необходимо правильно определять, какой именно цикл прерывает этот оператор. Для этого предлагается использовать стек операторов. Каждый узел CFG обязан обладать типом (см. рис. 4.2); каждый уровень стека хранит ссылки на узлы *break*, *continue*, *return*, встреченные при анализе на текущем уровне вложенности циклов. Анализ оператора цикла завершается слиянием всех найденных *break* текущего уровня вложенности циклов, с помощью объекта класса `CombineStub`.

Анализ операторов цикла *while* и *do-while* приведен на рис. 4.3 в) и г) соответственно, и принципиально не отличается от анализа оператора *for*.

Анализ оператора *switch* приведен на рис. 4.3 д). Предлагается рассматривать данный оператор как составной оператор ветвления, с учетом того, что *break* в данном контексте следует объединять по тому же принципу, что и внутри циклов.

Графические схемы анализа сложных операторов призваны проиллюстрировать метод построения CFG в целом. Так, например, если в программном коде за *if* следует *for*, *fi* в схеме *if* станет *parent* в схеме *for*, или если *stmt1* узла *if* будет являться оператором *for*, то для его схемы узлом *parent* будет узел *cond* схемы *if*, а *fi* двух схем объединятся.

Приведенные схемы реализованы в классе `botoo.flownode.FlowNodeBuilder`, листинг которого находится в приложении А.3.

4.6 Алгоритмы анализа CFG

4.6.1 Алгоритм поиска пути на графе

Алгоритм поиска пути на графе применяется для поиска возможных путей между точками с фиксированными состояниями автомата: точками входа и выхода из метода, а также точками присваивания значений аннотированной переменной состояния. Поэтому при поиске пути также следует учитывать, что путь между состояниями не может содержать узлов, где состояние может измениться.

Необходимо обеспечить два режима поиска. В первом режиме находится список возможных путей, и каждый путь отображается на диаграмме КА (в виде последовательности условий, выделенных из пути). Во втором режиме выполняется выделение подграфа, отражающего переход из одного узла в другой. Условия в подграфе затем анализируются и сворачиваются в одно условие. На диаграмме, таким образом, появляется не более одного перехода между состояниями.

Оба режима можно обеспечить с использованием поиска в глубину. На рис. 4.4 приведен пример реализации, позволяющей установить факт наличия пути между двумя узлами.

4.6.2 Алгоритм сворачивания условий перехода

Принцип объединения условий перехода выражается таблицей 4.1. Здесь буквами A, B, C, D обозначены условия, символом $\emptyset$ обозначен случай, когда по прямому или альтернативному не существует пути к искомому узлу. Если то или иное условие не участвует в результирующем условии, это значит, что такое условие может быть любым (в том числе и $\emptyset$).

Данная таблица используется в тот момент, когда известно, какое условие содержит текущий узел CFG, и какие условия необходимо выполнить, чтобы выполнить переход по прямому и альтернативному пути.

API компилятора позволяет доступ к AST класса только в режиме чтения. Поэтому, чтобы обеспечить возможность синтеза выражения, разработаны реализации интерфейсов `com.sun.source.tree`: `BinaryTree`, `ExpressionTree`, `LiteralTree`, `UnaryTree`.

Рассмотрим данные классы подробнее. Класс `botoo.tree.BinaryExpressionTree` позволяет создать псевдо-

```java
public class PathFinder {

    List<FlowNode> visited;

    public boolean find(FlowNode from, FlowNode to) {
        if (from == null || to == null) {
            return false;
        }
        if (visited.contains(from)) {
            return false;
        } else {
            if (from == to) {
                return true;
            } else {
                visited.add(from);
                boolean next = find(from.getNext(), to);
                boolean alt = find(from.getAltNext(), to);
                visited.remove(from);
                return next || alt;
            }
        }
    }
}
```

Рисунок 4.4. Реализация поиска пути на графе.

Таблица 4.1
Объединение условий

Текущее условие	**Прямой потомок**	**Альтернативный потомок**	**Результат**
A	B	C	B
A	$\emptyset$	$\emptyset$	$\emptyset$
A	$\emptyset$	C	$\neg A \vee C$
A	B	$\emptyset$	$A \vee B$
A	B	C	$(A \vee B) \wedge (\neg A \vee C)$
A	D	D	D

узел AST, объединяющий два логических выражения. Класс `botoo.tree.UnaryExpressionTree` позволяет добавить логическую инверсию в синтезируемое выражение. Класс `botoo.tree.LiteralTree` позволяет создавать произвольные литералы, например, для отражения в синтезируемом выражении вызовов процедур со сторонними эффектами. Класс `botoo.tree.StubExpressionTree` позволяет создавать «пустое» выражение, которое возвращается при обнаружении искомого узла (так как искомый узел по определению не содержит условий).

4.6.3 Алгоритм выделения переходов между состояниями

Алгоритм выделения переходов между состояниями базируется на анализе CFG метода. Как уже было сказано ранее, необходимо поддерживать два режима выделения переходов: с объединением дуг и без объединения. Входными параметрами алгоритма являются список узлов, в которых происходит фиксация (присваивание) состояния, и флаг, устанавливающий режим работы. Код алгоритма приведен на рис. 4.5.

```
FlowNodeBuilder f = new FlowNodeBuilder();
FlowNode root = new NamedStub("START:" + methodName);
FlowNode stop = new NamedStub("STOP:" + methodName);

List<FlowNode> changes;
boolean separateEdges;

Graph g = new Graph();

SliceOffStateName slicer = new SliceOffStateName(stateVar, stateNames);
PathFinder finder = new PathFinder(slicer);

List<Edge> sep = finder.findEdges(root, stop, separateEdges);
g.putEdge(sep);

for (int i = 0; i < len; i++) {
    for (int j = i; j < len; j++) {
        FlowNode from = changes.get(i);
        FlowNode to = changes.get(j);
        sep = finder.findEdges(from, to, separateEdges);
        g.putEdge(sep);
        if (i != j) {
            sep = finder.findEdges(to, from, separateEdges);
            g.putEdge(sep);
        }
    }
    FlowNode n = changes.get(i);
    sep = finder.findEdges(root, n, separateEdges);
    g.putEdge(sep);
}
```

Рисунок 4.5. Поиск переходов между состояниями КА.

4.7 Алгоритмы анализа выражений

В реализуемом прототипе системы выделения моделей КА из исходного кода используются алгоритмы объединения и анализа логических выражений. Первые нужны для того, чтобы логически сгруппировать все условия, которые должны быть выполнены для выполнения перехода между состояниями. Вторые нужны для того, чтобы исключить невозможные переходы и упростить эти выражения.

Группирование логических выражений осуществляется по схеме, представленной в таблице 4.1. Результатом группировки является дерево логического выражения, содержащее участки исходного AST, объединенные узлами, совместимыми с интерфейсом AST.

Анализ логических выражений реализован в виде класса `botoo.visitor.SliceOffStateName`, исходный код которого приведен в приложении A.4. Анализ можно условно разделить на две части. С одной стороны, проверяется, выполняется ли в некотором подвыражении сравнение переменной состояния с некоторым значением. Если такое сравнение имеет место, его значение вычисляется, так как информация об исходном состоянии, из которого выполняется переход, известна. С другой стороны, информация о значении подвыражений распространяется вверх по дереву, с учетом логических операций «или» и «и».

Пример анализа выражений приведена на рис. 4.6. На части *а)* рисунка приведено исходное логическое выражение, на части *б)* приведен результат его сокращения. Здесь S – переменная состояния. Исходное состояние – $S = 1$. Черным отмечены узлы, вычисленным значением которых является «*ложь*», серым – «*истина*». Выражение, в котором ложное значение распространяется до вершины, вызывает удаление перехода, соответствующего данному выражению.

Экземпляр класса передается в качестве параметра в такие сущности, как реализация поиска пути на графе (класс `botoo.flownode.PathFinder`) и построитель диаграммы КА (класс `botoo.graph.Graph`).

4.8 Построение диаграммы КА

Построение диаграммы КА выполняется процессором аннотаций – *botoo.StateAnnotationProcessor*: строится структура конечного автомата, записывается в файл описание на языке DOT и выполняется

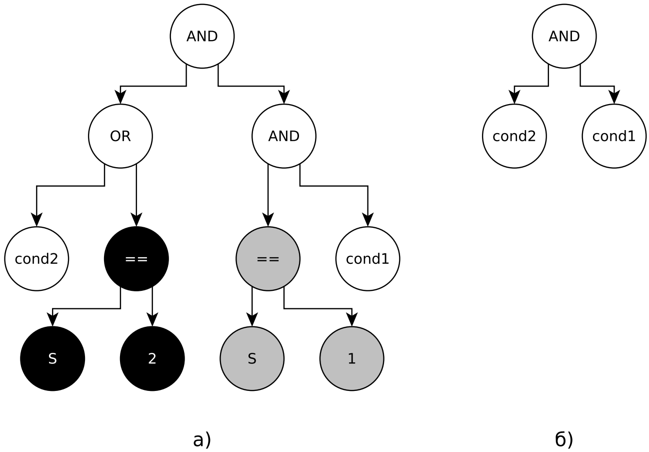

Рисунок 4.6. Пример анализа выражений. (Исходное состояние: $S = 1$.)

вызов GraphViz для синтеза диаграммы КА через механизм системных вызовов Java (обращение к операционной системе).

Структура графа КА описывается классами `botoo.graph.Graph` и `botoo.graph.Edge`. Последний отражает структуру перехода между состояниями и способ определения идентичности двух переходов (методы `equals()` и `hashCode()`, а первый хранит множество переходов и содержит механизм отображения структуры графа в формат DOT.

4.9 Выводы

В данном разделе рассмотрена реализация прототипа системы извлечения модели конечного автомата из исходного кода путем анализа конечного множества значений одной переменной Java-класса.

Приведены иерархическая структура пакетов разработанной системы, детальный разбор ключевых моментов реализации системы: аннотация переменной и обработка аннотации, построение множества возможных состояний, построение внутрипроцедурного CFG, реализации алгоритмов анализа графа потока управления и упрощения выражений, реализация вывода диаграммы КА.

5 ТЕСТИРОВАНИЕ ТЕХНОЛОГИИ ВЫДЕЛЕНИЯ МОДЕЛЕЙ КОНЕЧНЫХ АВТОМАТОВ ИЗ ИСХОДНОГО КОДА И АНАЛИЗ РЕЗУЛЬТАТОВ

В данном разделе производится апробация разработанного прототипа системы выделения моделей КА из исходного кода.

В подразделе 5.1 изложен метод тестирования системы. В подразделах 5.2 и 5.3 приведены тестовые примеры исходного кода, искомые и выделенные диаграммы и анализ полученных результатов.

5.1 Метод тестирования системы

Тестирование разработанного прототипа преследует две основные цели: отладка алгоритмов и демонстрация работоспособности системы. Тестирование также позволяет оценить качество прототипа как программного продукта.

Для оценки качества выделяемых диаграмм будет использоваться визуальное сравнение отличия полученного результата от ожидаемого. Таким образом, формальные критерии использоваться не будут. Схема процесса тестирования прототипа системы приведена на рис. 5.1.

Таким образом, итерация тестирования системы представляется следующим набором данных:

- исходный код тестового примера;
- исходная диаграмма конечного автомата;
- выделенная диаграмма конечного автомата (возможно, в нескольких вариантах);
- заключение об успешности итерации тестирования.

5.2 Тестовый пример `botoo.samples.StateMachine`

В качестве первого тестового примера возьмем класс `botoo.samples.StateMachine`, специально разработанный для

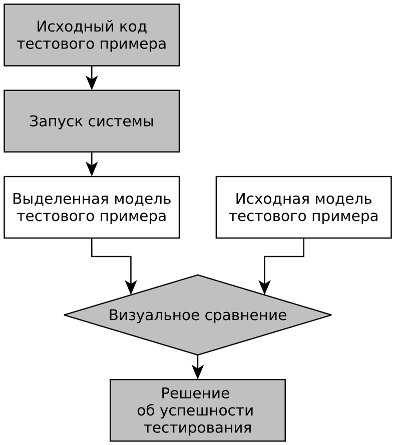

Рисунок 5.1. Схема процесса тестирования прототипа

демонстрации возможностей реализованной системы выделения моделей. Его исходный код приведен в приложении Б.1.

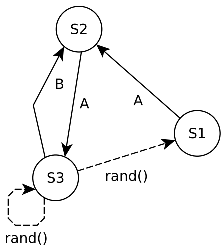

Рисунок 5.2. Искомая диаграмма КА класса
`botoo.samples.StateMachine`

Искомая диаграмма КА приведена на рис. 5.2. Пунктирными линиями с меткой *rand()* обозначены не определенные переходы. Это означает, что переходы могут активироваться спонтанно.

Выделенная диаграмма приведена на рис. 5.3. В синтезированных выражениях «&» означает конъюнктивное объединение условий,а «|»

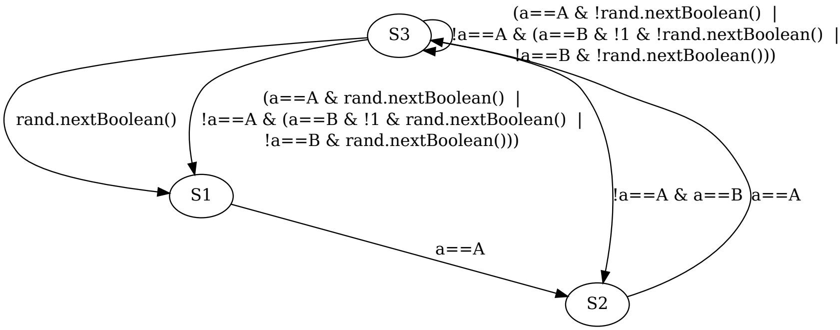

Рисунок 5.3. Выделенная диаграмма КА класса `botoo.samples.StateMachine`

– дизъюнктивное. Знак «!» является знаком логической инверсии.

Сравним две диаграммы и проанализируем полученный результат. Можно сделать два следующих вывода:

1. Переходы по символу входного алфавита (`Input`) выделены верно. Это справедливо для переходов $S1 \to S2$, $S2 \to S3$: в искомой диаграмме условием перехода является входное воздействие A, в выделенной диаграмме – логическое выражение $a = A$. Это также справедливо для перехода $S3 \to S1$, условием которого в искомой диаграмме является входное воздействие B, а в выделенной – по логическое выражение $a \neq A \lor a = B$.

2. Переход по случайному условию наличествует, однако его отображение переусложнено. Так, дополнительный анализ выражений и их упрощения позволили бы свести условие перехода $S3 \to S1$ до выражения $rand.nextBoolean()$, а для перехода $S3 \to S3$ – до выражения $!rand.nextBoolean()$.

Данный пример показывает, что разработанная система позволяет работать с такими конструкциями языка Java, как *if*, *switch*, *enum*, а также что выполняется анализ последовательности выполнения операторов.

Заключение. Выделенная диаграмма в целом верно отражает все переходы, заложенные при проектировании, но содержит точные условия возникновения событий. Однако способ представления несколько не очевиден. Это вызвано недостатком степени упрощения условий, а также использованием не оптимальных способов синтеза диаграммы.

5.3 Тестовый пример `botoo.samples.Alarm`

Тестовый пример `botoo.samples.Alarm` позаимствован из книги [23], как и искомая диаграмма КА (изображенная на рис. 5.4). В оригинале данный пример приведен автором для иллюстрации подхода к проектированию автоматных программ, в частности, программы, эмулирующей работу часов с будильником. Исходный код примера переработан для совместимости с Java, и приведен в приложении Б.2.

Рисунок 5.4. Искомая диаграмма КА класса `botoo.samples.Alarm`

Выделенная диаграмма изображена на рис. 5.5. В отличие от предыдущего примера, данный пример содержит процедуры со сторонним эффектом, метки о вызове которых помещены на диаграмме в квадратных скобках (например, [$z7$]).

Сравним две диаграммы и проанализируем полученный результат. Так же как и в предыдущем случае, корректно выделены переходы

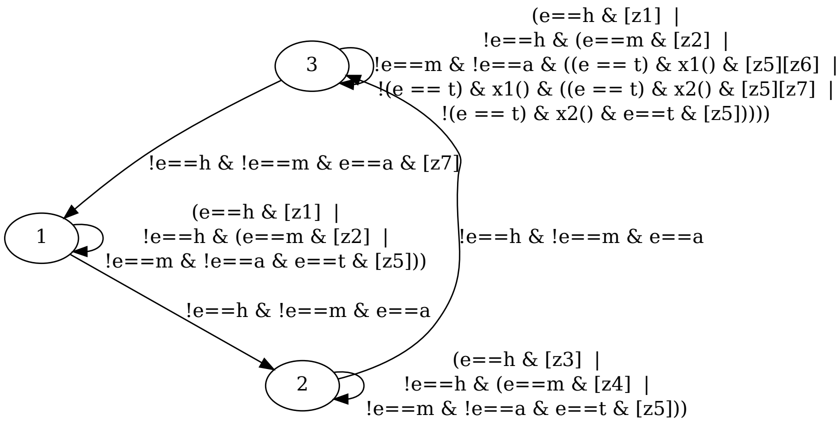

Рисунок 5.5. Выделенная диаграмма КА класса `botoo.samples.Alarm`

по событиям (значениям переменной e). По сравнению с искомой диаграммой, на выделенной диаграмме отсутствует переход $S3 \to S3$ по условию $T \vee \neg x_1 \vee \neg x_2$, однако если обратиться к исходному коду в приложении Б.2, такой переход в нем также отсутствует.

Заключение. Выделенная диаграмма отражает заложенную в программу модель КА, а также отражает выходные воздействия, порождаемые при переходах между состояниями (с помощью меток о процедурах со сторонним эффектом). Сравнение искомой и выделенной диаграмм позволяет обнаружить различие, которое говорит об отличии реализованной модели от ее проектного варианта.

5.4 Выводы

В данном разделе изложен способ тестирования системы, применяемый в целях демонстрации ее работоспособности, и проведен анализ двух тестовых примеров, показывающих точность и адекватность выделяемых моделей.

Рассмотрены негативные качества разработанной системы. Так, в подразделе 5.2 показано, что выделяемая модель может быть избыточна.

ЗАКЛЮЧЕНИЕ

Поддержка современного программного обеспечения требует от разработчика досконального понимания системы. В условиях унаследованного исходного кода и недостаточного документирования знания о системе можно получить только непосредственно из исходного кода. Эту задачу может упростить автоматизированный реверс-инжиниринг программ.

Одной из задач реверс-инжиниринга является выделение моделей из исходного кода, в числе которых находится модель конечного автомата. Данная модель используется для проектирования сложного поведения реагирующей системы.

Для решения задачи выделения моделей конечных автоматов из исходного кода было предложено использовать аннотирование переменных состояния и анализ графа потока управления поведения с точки зрения изменения состояний.

В работе выполнен анализ предметной области выделения поведенческих моделей из реализации системы. Рассмотрены существующие подходы к решению задачи и их сильные и слабые стороны. Показано, что не существует универсального формального метода. Рассмотрены существующие средства реверс-инжиниринга программного обеспечения, показано, что поддержка восстановления моделей конечных автоматов отсутствует.

Базисом предложенной технологии являются аннотирование переменных состояния и анализ внутрипроцедурного графа потока управления программы. Выполняется поиск присваиваний переменным состояния константных значений, которые интерпретируются как состояния, и анализируется граф потока управления на наличие путей между присваиваниями условия выполнения этих путей. Найденные пути объединяются, и строится граф конечного автомата.

Вывод диаграммы конечного автомата выполняется в текстовом формате DOT, а графический вариант диаграммы получается из текстового с помощью GraphViz.

В результате получен метод выделения моделей конечных автоматов из программ, при проектировании которых была использована модель конечного автомата. Разработанный метод является формальным и универсальным, так как он использует формальные алгоритмы и структуры данных, и анализирует изменение состояния, а не способ кодирования автомата.

Разработан прототип системы выделения моделей для языка Java и одной переменной состояния, который протестирован на нескольких примерах автоматных программ. Результаты тестирования показали, что выделяемые модели соответствуют ожидаемым. Сравнение модели из спецификации и выделенной модели позволяет обнаружить дефекты реализации. Тестирование также показало, что выделяемые модели могут быть избыточны и не наглядны.

Предложенная технология реверс-инжиниринга программного обеспечения может применяться в ходе поддержки унаследованного исходного кода в качестве автоматизированного средства выделения моделей конечных автоматов. От инженера-исследователя требуется только проаннотировать исходный код. Побочным результатом применения метода является получение внутрипроцедурных графов потока управления, которые легко могут быть визуализированы с помощью GraphViz, и использованы инженером-исследователем для анализа поведения программы.

С точки зрения развития предложенной технологии возможны два направления: развитие собственно метода и развитие прототипа. Развитие метода можно производить в направлении анализа полного графа потока управления программы. Этого можно достичь комбинацией нерекурсивных графов. Развитие метода также можно вести в направлении более полного анализа присваивания состояний, например, в виде анализа присваиваний переменной состояния не только константных значений. Возможна также разработка алгоритмов анализа и упрощения выражений, отражающих условия переходов конечного автомата.

Доработка прототипа системы должна включать в себя более полное тестирование, поддержку нескольких переменных состояния, поддержку альтернативных способов вывода графа конечного автомата.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. *Джон Хопкрофт, Раджив Мотвани, Джеффри Ульман.* Введение в теорию автоматов, языков и вычислений. — Москва, Вильямс, 2002. — С. 528.

2. *Лупал А.М.* Теория Автоматов. — СПбГУАП, 2000. — С. 120.

3. *Глухих М.И., Ицыксон В.М.* Программная инженерия. Обеспечение качества программных средств методами статического анализа. — Издательство Политехнического Университета, 2011. — С. 150.

4. TIOBE Programming Community Index for May 2011. — 2011. `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`.

5. *James C. Corbett, Matthew B. Dwyer, John Hatcliff et al.* Bandera: Extracting Finite-state Models from Java Source Code // IN PROCEEDINGS OF THE 22ND INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING. — ACM Press, 2000. — Pp. 439–448.

6. Spin – Formal Verification. — 2011. `http://spinroot.com/`.

7. *Mark van den Brand, Alexander Serebrenik, Dennie van Zeeland.* Extraction of State Machines of Legacy C code with Cpp2XMI.

8. *E. Korshunova, Marija Petkovic, M. G. J. van den Brand, Mohammad Reza Mousavi.* CPP2XMI: Reverse Engineering of UML Class, Sequence, and Activity Diagrams from C++ Source Code. // WCRE. — IEEE Computer Society, 2006. — Pp. 297–298.

9. *Ravindra Naik Rahul Jiresal, Hemanth Makkapati.* Statechart Extraction from Code – An Approach using Static Program Analysis and Heuristics Based Abstractions. — 2011. — India Workshop on Reverse Engineering, IWRE '11.

10. *Axel Cleeremans, David Servan-Schreiber, James L. McClelland.* Finite state automata and simple recurrent networks // *Neural Comput.* — 1989. — September. — Vol. 1. — Pp. 372–381.

11. *Jonathan E. Cook, Alexander L. Wolf.* Discovering models of software processes from event-based data // *ACM Trans. Softw. Eng. Methodol.* — 1998. — July. — Vol. 7. — Pp. 215–249.

12. *Paolo Milani Comparetti, Gilbert Wondracek, Christopher Krügel, Engin Kirda.* Prospex: Protocol Specification Extraction // IEEE Symposium on Security and Privacy. — 2009. — Pp. 110–125.

13. *John Whaley, Michael C. Martin, Monica S. Lam.* Automatic extraction of object-oriented component interfaces // *SIGSOFT Softw. Eng. Notes.* — 2002. — July. — Vol. 27. — Pp. 218–228.

14. *James Gosling, Bill Joy, Guy Steele, Gilad Bracha.* The Java™Language Specification, The (3rd Edition). — Addison-Wesley Professional, 2005. — P. 688.

15. Borland Together. — 2011. `http://www.borland.com/us/products/together/index.aspx`.

16. Understand: Source Code Analysis & Metrics. — 2011. `http://www.scitools.com/`.

17. WithClass. — 2011. `http://microgold.com/`.

18. Imagix 4D. — 2011. `http://www.imagix.com/`.

19. yDoc – Javadoc™UML Extension. — 2011. `http://www.yworks.com/en/products_ydoc.html`.

20. Java language annotations. — 2011. `http://download.oracle.com/javase/1.5.0/docs/guide/language/annotations.html`.

21. The DOT Language. — 2011. `http://www.graphviz.org/doc/info/lang.html`.

22. Graphviz – Graph Vizualization Software. — 2011. `http://www.graphviz.org/`.

23. *Поликарпова Н.И., Шалыто А.А.* Автоматное программирование. — Спб.: Питер, 2011. — С. 176.

ПРИЛОЖЕНИЕ А
ЛИСТИНГИ

Листинг А.1. Тестовая реализация КА. Класс `botoo.samples.StateMachine`.

```
package botoo.samples;

import botoo.annotate.StateVariable;
import java.util.Random;

public class StateMachine {

    public enum Input {
        A, B
    };
    public enum State {
        S1, S2, S3
    };
    @StateVariable
    private State state = State.S1;
    Integer field;
    Random rand = new Random();

    public void transit(Input a) {
        switch (a) {
            case A:
                switch (state) {
                    case S1:
                        state = State.S2;
                        break;
                    case S2:
                        state = State.S3;
                }
                break;
            case B:
                if (state == State.S3) {
                    state = State.S2;
                }
        }
        if (state == State.S3) {
            if (rand.nextBoolean()){
                state = State.S1;
            }
        }
    }
}
```

Листинг А.2. Реализация ϕ-функции. Класс `botoo.flownode.CombineStub`.

```
package botoo.flownode;
```

```java

import java.util.LinkedList;
import java.util.List;

public class CombineStub extends DefaultFlowNode {

    List<FlowNode> l;

    public CombineStub(List<FlowNode> l) {
        this.l = new LinkedList<FlowNode>();
        for(FlowNode f : l){
            if (f != null && !this.l.contains(f)){
                this.l.add(f);
            }
        }
    }

    @Override
    public void setNext(FlowNode c) {
        for (FlowNode f : l) {
            f.setNext(c);
        }
    }

    @Override
    public void setAltNext(FlowNode c) {
        for (FlowNode f : l) {
            f.setAltNext(c);
        }
    }

    @Override
    public FlowNode getNext() {
        if (l!=null && !l.isEmpty()){
            return l.get(0).getNext();
        } else {
            return null;
        }
    }

}
```

Листинг А.3. Построение внутрипроцедурного CFG. Класс `botoo.flownode.FlowNode`.

```java
package botoo.flownode;

import botoo.scanner.FindMethodWithName;
import botoo.tree.BinaryExpressionTree;
import com.sun.source.tree.*;
import com.sun.source.util.SimpleTreeVisitor;
import java.util.Arrays;
import java.util.LinkedList;
import java.util.List;

public class FlowNodeBuilder extends SimpleTreeVisitor<FlowNode,
```

```
        FlowNode> {

    List<List<FlowNode>> unionStack;
    FindMethodWithName scanner;
    String stateVar = "";

    public FlowNodeBuilder(FindMethodWithName scanner) {
        this.scanner = scanner;
    }

    public FlowNodeBuilder(FindMethodWithName scanner, String
         stateVar) {
        this(scanner);
        this.stateVar = stateVar;
    }

    @Override
    protected FlowNode defaultAction(Tree node, FlowNode parent) {
        FlowNode flowNode;
        if (node != null) {
            flowNode = new Stub(node, node.toString().replace("\n", "
                 "),
                    scanner.scan(node));
            if (parent != null) {
                parent.setNext(flowNode);
            }
        } else {
            flowNode = parent;
        }
        return flowNode;
    }

    @Override
    public FlowNode visitMethod(MethodTree node, FlowNode p) {
        unionStack = new LinkedList<List<FlowNode>>();
        LinkedList<FlowNode> unionList = new LinkedList<FlowNode>();
        unionStack.add(unionList);

        FlowNode block = visitBlock(node.getBody(), p);
        List<FlowNode> ofTypeReturn = ofType(unionList,
             FlowNode.Type.RETURN);
        ofTypeReturn.add(block);

        return new CombineStub(ofTypeReturn);
    }

    @Override
    public FlowNode visitBlock(BlockTree node, FlowNode p) {
        return visitTreeList(node.getStatements(), p);
    }

    @Override
    public FlowNode visitForLoop(ForLoopTree node, FlowNode parent) {
        FlowNode init = visitTreeList(node.getInitializer(), parent);

        FlowNode cond;
        if(node.getCondition() == null) {
            cond = new Stub(null, "true");
```

```
            init.setNext(cond);
        } else {
            cond = node.getCondition().accept(this, init);
        }

        List<FlowNode> unionList = new LinkedList<FlowNode>();
        unionStack.add(unionList);

        FlowNode stmt = node.getStatement().accept(this, cond);

        FlowNode upd = visitTreeList(node.getUpdate(), stmt);

        upd.setNext(cond);

        List<FlowNode> ofTypeBreak = ofType(unionList,
             FlowNode.Type.BREAK);
        ofTypeBreak.add(new AltNextStub(cond));

        List<FlowNode> ofTypeContinue = ofType(unionList,
             FlowNode.Type.CONTINUE);
        new CombineStub(ofTypeContinue).setNext(cond);

        FlowNode combine = new CombineStub(ofTypeBreak);
        unionStack.remove(unionStack.size()-1);

        return combine;
    }

    private FlowNode visitTreeList(List<? extends Tree> l, FlowNode
         parent) {
        FlowNode last = parent;
        for (Tree st : l) {
            FlowNode next = st.accept(this, last);
            last = next;
        }
        return last;
    }

    @Override
    public FlowNode visitSwitch(SwitchTree node, FlowNode p) {
        String var = node.getExpression().toString();
        boolean switchState = var.equals(stateVar);

        FlowNode last = p;
        FlowNode lastCond = null;
        FlowNode lastAct = null;

        List<FlowNode> unionList = new LinkedList<FlowNode>();
        unionStack.add(unionList);

        for (CaseTree ct : node.getCases()) {

            FlowNode cond;
            if(ct.getExpression() == null){
                cond = last;
            } else {
                BinaryExpressionTree cttree = new BinaryExpressionTree(
                        node.getExpression(), ct.getExpression(),
```

```
                        Tree.Kind.EQUAL_TO);
                cond = cttree.accept(this, null);
            }

            FlowNode act = visitTreeList(ct.getStatements(), cond);

            if (lastAct != null) {
                lastAct.setNext(cond.getNext());
            }

            lastAct = act;

            if (lastCond != null) {
                lastCond.setAltNext(cond);
            } else {
                last.setNext(cond);
            }
            lastCond = cond;
            last = cond;
        }

        List<FlowNode> ofTypeBreak = ofType(unionList,
             FlowNode.Type.BREAK);
        ofTypeBreak.add(new AltNextStub(lastCond));
        ofTypeBreak.add(lastAct);

        FlowNode sum = new CombineStub(ofTypeBreak);

        unionStack.remove(unionStack.size()-1);

        return sum;
    }

    @Override
    public FlowNode visitWhileLoop(WhileLoopTree node, FlowNode p) {
        FlowNode cond = node.getCondition().accept(this, p);

        List<FlowNode> unionList = new LinkedList<FlowNode>();
        unionStack.add(unionList);

        FlowNode stmt = node.getStatement().accept(this, cond);

        stmt.setNext(cond);

        List<FlowNode> ofTypeBreak = ofType(unionList,
             FlowNode.Type.BREAK);
        ofTypeBreak.add(new AltNextStub(cond));

        List<FlowNode> ofTypeContinue = ofType(unionList,
             FlowNode.Type.CONTINUE);
        new CombineStub(ofTypeContinue).setNext(cond);

        unionStack.remove(unionStack.size()-1);

        return new CombineStub(ofTypeBreak);
    }

    @Override
```

```java
    public FlowNode visitIf(IfTree node, FlowNode p) {
        FlowNode cond = node.getCondition().accept(this, p);

        StatementTree thenStatement = node.getThenStatement();
        StatementTree elseStatement = node.getElseStatement();

        FlowNode thenStmt = null;
        FlowNode elseStmt = null;

        if (thenStatement != null) {
            thenStmt = thenStatement.accept(this, cond);
        }
        if (elseStatement != null) {
            elseStmt = elseStatement.accept(this, new
                AltNextStub(cond));
        } else {
            elseStmt = new AltNextStub(cond);
        }

        return new CombineStub(Arrays.asList(thenStmt, elseStmt));
    }

    @Override
    public FlowNode visitDoWhileLoop(DoWhileLoopTree node, FlowNode p)
        {

        List<FlowNode> unionList = new LinkedList<FlowNode>();
        unionStack.add(unionList);

        FlowNode stmt = node.getStatement().accept(this, p);
        FlowNode cond = node.getCondition().accept(this, stmt);

        cond.setNext(p.getNext());

        List<FlowNode> ofTypeBreak = ofType(unionList,
            FlowNode.Type.BREAK);
        ofTypeBreak.add(new AltNextStub(cond));

        List<FlowNode> ofTypeContinue = ofType(unionList,
            FlowNode.Type.CONTINUE);
        new CombineStub(ofTypeContinue).setNext(cond);

        unionStack.remove(unionStack.size()-1);

        return new CombineStub(ofTypeBreak);
    }

    @Override
    public FlowNode visitBreak(BreakTree node, FlowNode p) {
        FlowNode act = this.defaultAction(node, p);
        if (node != null) {
            act.setType(FlowNode.Type.BREAK);
            unionStack.get(unionStack.size()-1).add(act);
        }
        return act;
    }

    @Override
```

```
    public FlowNode visitReturn(ReturnTree node, FlowNode p) {
        FlowNode act = this.defaultAction(node, p);
        if (node != null) {
            act.setType(FlowNode.Type.RETURN);
            unionStack.get(0).add(act);
        }
        return act;
    }

    @Override
    public FlowNode visitContinue(ContinueTree node, FlowNode p) {
        FlowNode act = this.defaultAction(node, p);
        if (node != null) {
            act.setType(FlowNode.Type.CONTINUE);
            unionStack.get(unionStack.size()-1).add(act);
        }
        return act;
    }

    @Override
    public FlowNode visitTry(TryTree node, FlowNode p) {
        p = node.getBlock().accept(this, p);
        p = node.getFinallyBlock().accept(this, p);
        return p;
    }


    static List<FlowNode> ofType(List<FlowNode> l, FlowNode.Type t){
        List<FlowNode> ret = new LinkedList<FlowNode>();
        for(FlowNode f : l){
            if(f.getType() == t){
                ret.add(f);
            }
        }
        return ret;
    }

}
```

Листинг А.4. Анализ выражений. Класс `botoo.visitor.SliceOffStateName`.

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package botoo.visitor;

import botoo.tree.BinaryExpressionTree;
import botoo.tree.StubExpressionTree;
import botoo.tree.UnaryExpressionTree;
import com.sun.source.tree.BinaryTree;
import com.sun.source.tree.ExpressionTree;
import com.sun.source.tree.IdentifierTree;
import com.sun.source.tree.LiteralTree;
import com.sun.source.tree.MemberSelectTree;
```

```java
import com.sun.source.tree.Tree;
import com.sun.source.tree.UnaryTree;
import com.sun.source.util.SimpleTreeVisitor;
import java.util.LinkedList;
import java.util.List;
import java.util.regex.Pattern;

/**
 *
 * @author lonlylocly
 */
public class SliceOffStateName extends SimpleTreeVisitor<ExpressionTree, Void> {

    String stateVar;
    List<String> stateNames;
    /**
     * This flag indicates if state variable comparison
     * is to be kept in expression or to be removed
     */
    boolean keepStates = true;
    /**
     * This flag indicates if there's 'point' state,
     * which restricts possible from-states to a single one
     */
    boolean exactState = false;
    /**
     * This is 'point' state name
     */
    String statePoint = "";

    public SliceOffStateName(String stateVar, List<String> stateNames) {
        this.stateVar = stateVar;
        this.stateNames = new LinkedList<String>(stateNames);
    }

    private boolean matches(String node) {
        boolean m = Pattern.matches(stateVar, node);
        return keepStates ? m : !m;
    }

    private ExpressionTree exactState(BinaryTree node) {
        String left = node.getLeftOperand() == null
                ? ""
                : node.getLeftOperand().toString();
        String right = node.getRightOperand() == null
                ? ""
                : node.getRightOperand().toString();
        boolean point_in = left.equals(stateVar) && right.equals(statePoint)
                    || right.equals(stateVar) && left.equals(statePoint);
        boolean point_out = left.equals(stateVar) && !right.equals(statePoint)
                    || right.equals(stateVar) && !left.equals(statePoint);
        if (node.getKind() == Tree.Kind.EQUAL_TO){
```

```java
            // s == 1 || 1 == s
            if (point_in){
                return new StubExpressionTree();
            } else if (point_out){
                return null;
            }
        } else if (node.getKind() == Tree.Kind.NOT_EQUAL_TO){
            // s != 1 || 1 != s
            if (point_out){
                return new StubExpressionTree();
            } else if (point_in){
                return null;
            }
        }
        return node;
    }

    @Override
    protected ExpressionTree defaultAction(Tree node, Void p) {
        if (node != null
                && node instanceof ExpressionTree
                && matches("" + node)) {
            return (ExpressionTree) node;
        } else {
            return null;
        }
    }

    @Override
    public ExpressionTree visitBinary(BinaryTree node, Void p) {
        /**
         * When concrete state 'point' is defined, slice only correct
         */
        if (exactState) {
            ExpressionTree e = exactState(node);
            if (e != node){
                return e;
            }
        }
        ExpressionTree left = node.getLeftOperand();
        if (left != null) {
            left = left.accept(this, p);
        }
        ExpressionTree right = node.getRightOperand();
        if (right != null) {
            right = right.accept(this, p);
        }
        boolean equality = node.getKind() == Tree.Kind.EQUAL_TO
                || node.getKind() == Tree.Kind.NOT_EQUAL_TO;
        if (left != null && right != null) {
            if (left instanceof StubExpressionTree ){
                return right;
            } else if (right instanceof StubExpressionTree){
                return left;
            }
            return new BinaryExpressionTree(left, right,
                 node.getKind());
        } else if (!keepStates && equality
```

```
                || exactState && node.getKind() ==
                    Tree.Kind.CONDITIONAL_AND) {
            /**
             * When slicing off states, omit matches in [not] equal to
             */
            return null;
        } else if (left != null) {
            if (right == null) {
                right = node.getRightOperand();
                if (right instanceof LiteralTree
                        || right instanceof IdentifierTree
                        || right instanceof MemberSelectTree) {
                    return node;
                }
                return left;
            } else {
                return new BinaryExpressionTree(left, right,
                    node.getKind());
            }
        } else if (right != null) {
            left = node.getLeftOperand();
            if (left instanceof LiteralTree
                    || left instanceof IdentifierTree
                    || left instanceof MemberSelectTree) {
                return node;
            }
            return right;
        } else {
            return null;
        }
    }

    @Override
    public ExpressionTree visitUnary(UnaryTree node, Void p) {
        ExpressionTree expression = node.getExpression();
        if (expression != null) {
            ExpressionTree accept = expression.accept(this, p);
            if (accept != null) {
                return new UnaryExpressionTree(accept, node.getKind());
            } else if (exactState){
                return new StubExpressionTree();
            }
        } else if (!keepStates) {
            return node;
        }
        return null;
    }

    public boolean isKeepStates() {
        return keepStates;
    }

    public void setKeepStates(boolean keepStates) {
        this.keepStates = keepStates;
    }

    public List<String> getStateNames() {
        return stateNames;
```

```java
    }

    public String getStateVar() {
        return stateVar;
    }

    public boolean isExactState() {
        return exactState;
    }

    public void setExactState(boolean exactState) {
        this.exactState = exactState;
    }

    public String getStatePoint() {
        return statePoint;
    }

    public void setStatePoint(String statePoint) {
        this.statePoint = statePoint;
    }

}
```

ПРИЛОЖЕНИЕ Б
ТЕСТОВЫЕ ПРИМЕРЫ

Листинг Б.1. Тестовый пример. Класс `botoo.samples.StateMachine`.

```java
package botoo.samples;

import botoo.annotate.StateVariable;
import java.util.Random;

public class StateMachine {

    public enum Input {
        A, B
    };
    public enum State {
        S1, S2, S3
    };
    @StateVariable
    private State state = State.S1;
    Integer field;
    Random rand = new Random();

    public void transit(Input a) {
        switch (a) {
            case A:
                switch (state) {
                    case S1:
                        state = State.S2;
                        break;
                    case S2:
                        state = State.S3;
                }
                break;
            case B:
                if (state == State.S3) {
                    state = State.S2;
                }
        }
        if (state == State.S3) {
            if (rand.nextBoolean()){
                state = State.S1;
            }
        }
    }
}
```

Листинг Б.2. Тестовый пример. Класс `botoo.samples.Alarm`.

```java
package botoo.samples;

import botoo.annotate.SideEffectSubroutine;
import botoo.annotate.StateVariable;
```

```
public class Alarm {

    final int h = 1;
    final int m = 2;
    final int a = 3;
    final int t = 4;
    int e;
    @StateVariable
    int y;
    int hours;
    int minutes;
    int alarm_hours;
    int alarm_minutes;

    boolean x1() {
        if ((minutes == alarm_minutes - 1) && (hours == alarm_hours)
                || (alarm_minutes == 0) && (minutes == 59) && (hours
                    == alarm_hours - 1)) {
            return true;
        } else {
            return false;
        }
    }

    boolean x2() {
        if ((minutes == alarm_minutes) && (hours == alarm_hours)) {
            return true;
        } else {
            return false;
        }
    }

    @SideEffectSubroutine
    void z1() {
        hours = (hours + 1) % 24;
    }

    @SideEffectSubroutine
    void z2() {
        minutes = (minutes + 1) % 60;
    }

    @SideEffectSubroutine
    void z3() {
        alarm_hours = (alarm_hours + 1) % 24;
    }

    @SideEffectSubroutine
    void z4() {
        alarm_minutes = (alarm_minutes + 1) % 60;
    }

    @SideEffectSubroutine
    void z5() {
        minutes = (minutes + 1) % 60;
        if (minutes == 0) {
            hours = (hours + 1) % 24;
```

```
        }
    }
    @SideEffectSubroutine
    void z6() {
        System.out.println("Start ringing");
    }
    @SideEffectSubroutine
    void z7() {
        System.out.println("Stop ringing");
    }

    void A1() {
        switch (y) {
            case 1:
                if (e == h) {
                    z1();
                } else if (e == m) {
                    z2();
                } else if (e == a) {
                    y = 2;
                } else if (e == t) {
                    z5();
                }
                break;
            case 2:
                if (e == h) {
                    z3();
                } else if (e == m) {
                    z4();
                } else if (e == a) {
                    y = 3;
                } else if (e == t) {
                    z5();
                }
                break;
            case 3:
                if (e == h) {
                    z1();
                } else if (e == m) {
                    z2();
                } else if (e == a) {
                    z7();
                    y = 1;
                } else if ((e == t) && x1()) {
                    z5();
                    z6();
                } else if ((e == t) && x2()) {
                    z5();
                    z7();
                } else if (e == t) {
                    z5();
                }
                break;
        }
    }
}
```