



ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ  
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

**САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ  
ИССЛЕДОВАТЕЛЬСКИЙ АКАДЕМИЧЕСКИЙ УНИВЕРСИТЕТ  
РОССИЙСКОЙ АКАДЕМИИ НАУК**

---

На правах рукописи

Диссертация допущена к защите

Зав. кафедрой

---

“ ” \_\_\_\_\_ 20... г.

**ДИССЕРТАЦИЯ  
НА СОИСКАНИЕ УЧЕНОЙ СТЕПЕНИ  
МАГИСТРА**

Тема: Построение регулярной аппроксимации встроенных языков

Направление: 03.04.01 – Прикладные математика и физика

Выполнил студент

(подпись)

*М.Р. Хабибуллин*

Руководитель:

(подпись)

*С.В. Григорьев*

Рецензент:

(подпись)

*В.С. Полозов*

Санкт-Петербург  
2015 г

## Реферат

Стр. 50, листингов 6, рис. 17, табл. 2.

В данной работе представлена реализация инструмента построения регулярной аппроксимации встроенных языков сверху. Инструмент разделён на две части: back-end, который реализует обобщённый алгоритм построения аппроксимации, независимый от языка, порождающего строки со встроенным кодом, и front-end, который конвертирует специфичные для конкретного языка входные данные в обобщённые для передачи в back-end. Такое разделение позволяет сравнительно просто строить аппроксимацию для различных языков, порождающих строки со встроенным кодом, путём реализации соответствующего front-end'a. Реализованный алгоритм построения аппроксимации умеет обрабатывать большинство конструкций языка, манипулирующего строками: циклы, условные операторы, вызовы функций (в том числе с хвостовой рекурсией), объявления переменных, присваивание, конкатенацию, присваивание с конкатенацией, замену в строках и строковые литералы.

Разработанный инструмент планируется включить в состав статического анализатора встроенного кода, разрабатываемого в рамках платформы YaccConstructor. Благодаря обобщённости алгоритма аппроксимации и модульной архитектуре реализованного инструмента статический анализатор будет способен работать с различными языками, использующими встраивание кода в строки, чем будет выгодно отличаться от существующих аналогов, работающих с конкретными языками.

*Ключевые слова:* встроенные языки, статический анализ, регулярная аппроксимация строковых выражений.

## Содержание

1. Введение.....	3
2. Цели и задачи .....	6
3. Обзор .....	7
3.1. Существующие решения.....	7
3.1.1. PhpStorm .....	7
3.1.2. IntelliLang.....	9
3.1.3. Alvor .....	9
3.1.4. Varis.....	10
3.1.5. Java String Analyzer и Php String Analyzer.....	11
3.1.6. Особенности существующих решений .....	12
3.2. Поиск строк со встроенным кодом .....	12
3.3. Построение регулярной аппроксимации сверху .....	13
3.4. Платформа для реализации и вспомогательные технологии .....	22
4. Реализация .....	24
4.1. Общая схема работы инструмента.....	24
4.3. Обобщённый CFG.....	29
4.4. Особенности обработки графов .....	33
4.5. Построение TCFS.....	35
4.6. Построение конечного автомата по TCFS .....	41
4.7. Межпроцедурная аппроксимация .....	45
4.7.1. Основные сведения о реализации .....	45
4.7.2. Обработка рекурсивных методов и функций .....	46
4.7.3. Особенности реализации межпроцедурной аппроксимации для JavaScript.....	47
5. Заключение .....	48
6. Библиографический список .....	50

## 1. Введение

При написании определённых типов компьютерных программ на каком-либо языке нередко возникает необходимость формировать код на другом языке. Примером может служить создание и отправка SQL-запросов к базе данных, с которой работает программа, написанная на Java, C#, PHP и т.д. В коде такой программы SQL-запросы могут быть записаны в виде строковых литералов, а также формироваться динамически с использованием строковых операций (конкатенация, замена в строке) и общих языковых конструкций (циклы, условные операторы). Другие примеры использования данного подхода – формирование кода на JavaScript внутри кода на Java при написании web-приложений, генерация кода на HTML, CSS и JavaScript с помощью PHP для создания динамических web-страниц, составление динамических SQL-запросов с помощью DynamicSQL, генерация xml-файлов и другие.

Язык, который манипулирует строками, содержащими код, называется основным языком, а язык, код которого записан внутри строк, называется встроенным. Понятие “встроенный язык” имеет и другие устоявшиеся значения, однако, в рамках данной работы, для краткости, мы будем пользоваться именно им.

Для программ указанного типа необходим статический анализ встроенного кода. С одной стороны, он позволит предоставлять поддержку встроенных языков в средах разработки, то есть выполнять подсветку синтаксиса, автодополнение и подобные функции, доступные для основных языков, прямо внутри строк.

С другой стороны, описанный способ построения программ в некоторых случаях начинает вытесняться другими, более совершенными подходами. К примеру, в случае со встроенным SQL, этими подходами являются ORM

(object-relational mapping, объектно-реляционное отображение) и LINQ (language integrated query, встроенные в язык запросы). Но, несмотря на это, по-прежнему существует много программ, написанных с использованием встраивания кода в строки. Поскольку эти программы до сих пор используются, им необходимы сопровождение и поддержка. Одно из средств облегчения поддержки – это реинжиниринг. Статический анализ встроенных языков полезен для реинжиниринга как минимум в двух направлениях.

Первое направление – сбор различной статистики для встроенного кода. Пользуясь формальными метриками (например, цикломатической сложностью [1]), можно оценить структурную сложность встроенного кода и, если участок кода со сложной структурой часто используется, сделать вывод о необходимости его рефакторинга. Другой пример подсчёта метрик (в случае со встроенным кодом на SQL) – это оценка частоты обращений к тем или иным таблицам. Если на основе результатов оценки становится ясно, что какие-то таблицы используются очень редко или не используются совсем, то можно задуматься о переносе данных из них в другие таблицы или удалении их из базы данных.

Второе направление – создание средств автоматизированной трансформации кода с целью замены старого подхода к встраиванию на современные. Примером может служить трансформация встроенного в строки SQL в LINQ. Важно отметить, что возможность подобных трансформаций в общем случае находится под вопросом даже с теоретической точки зрения. Однако в частных случаях их можно выполнять автоматизировано.

Приведённые примеры подтверждают актуальность задачи статического анализа встроенного кода. Несмотря на это, существующие решения в основном направлены на поддержку встроенного кода в средах разработки и

слабо применимы к задачам реинжиниринга. Кроме того, наиболее универсальные из них ориентированы на работу с каким-либо конкретным основным языком. Необходим инструмент, способный работать с различными основными языками, и позволяющий легко добавлять различные функции (от подсветки встроенного кода до сбора метрик).

Одна из основных задач, которую должен уметь решать такой инструмент, заключается в построении множества строк со встроенным кодом, которые может породить программа, и представлении его в виде некоторой структуры конечного размера. Данное множество можно рассматривать как некоторый язык. Удобнее всего работать с регулярными языками, так как для этого класса языков разрешимы многие задачи, его можно задать с помощью конечного автомата. Однако, имеющийся язык в общем случае является рекурсивно перечислимый, а не регулярным. Несмотря на это, можно построить аппроксимацию имеющегося языка регулярным сверху, то есть такой регулярный язык, который содержит все слова имеющегося языка и, возможно, некоторые другие.

Выполнение статического анализа на основе регулярной аппроксимации является одним из возможных подходов к созданию статического анализатора встроенного кода. Если научиться строить аппроксимацию множества порождаемых строк с кодом независимо от того, какой основной язык это множество порождает, то создание анализатора, способного решать обозначенные ранее задачи, становится реальным.

## **2. Цели и задачи**

Целью данной работы является построение регулярной аппроксимации строковых выражений без привязки к основному языку. Для достижения цели решаются четыре задачи:

1. Реализовать алгоритм построения аппроксимации строковых выражений с поддержкой основных строковых операций и конструкций языка;
2. Сделать решение независимым от основного языка;
3. Интегрировать результат работы в платформу, в рамках которой разрабатываются другие части инструмента статического анализа встроенных языков, для дальнейшего объединения в цельное решение;
4. Реализовать построение аппроксимации для конкретных основных языков для проверки независимости алгоритма от основного языка.

### 3. Обзор

#### 3.1. Существующие решения

Целью данной работы является создание инструмента построения регулярной аппроксимации множества строк со встроенным кодом. Подобные инструменты чаще всего являются частью статического анализатора, а не используются самостоятельно. Поэтому в данном разделе рассматриваются существующие решения задачи статического анализа встроенных языков в целом. На основе их особенностей и недостатков делается вывод о необходимости альтернативного решения с описанными ранее свойствами, а значит и инструмента регулярной аппроксимации как одной из его основных частей.

##### 3.1.1. PhpStorm

PhpStorm<sup>1</sup> – среда разработки для создания web-приложений на Php. Программы на Php часто содержат встроенный код на JavaScript, CSS и HTML, а также на SQL, и данная среда разработки предоставляет некоторый функционал для поддержки такого кода.

---

<sup>1</sup> Сайт среды разработки PhpStorm (посещён: 28.07.2015): <https://www.jetbrains.com/phpstorm/>



```

1  <?php
2      $usualString = 'simple string';
3      $hello = '<html><body>Hello, world!</body></html>';
4      $string = '<';
5      if (cond)
6          $string .= 'html';
7      else
8          $string .= 'body';
9      $string .= '>';
10
11     $error = 'SELECT * FROM table1 FROM table1';
12  ?>

```

Рисунок 1. Окно текстового редактора PhpStorm

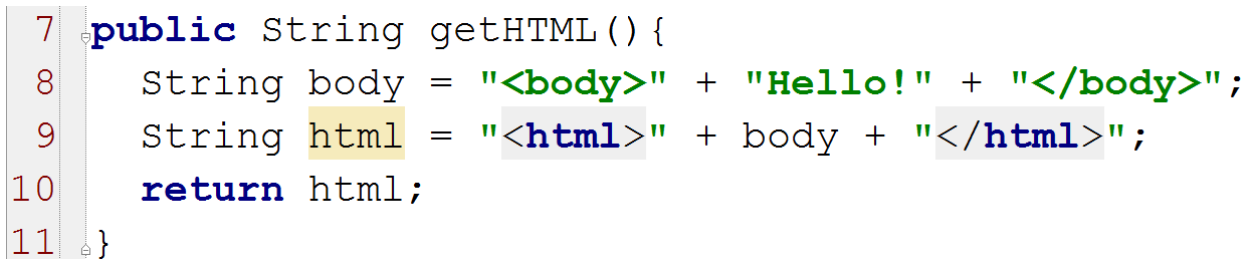
На Рисунке 1 показано окно текстового редактор PhpStorm. В строках 2 и 3 объявляются две переменные. Переменная `$usualString` содержит обычный текст, а `$hello` содержит код на языке HTML. PhpStorm смог определить, что последняя переменная содержит код и выполнил его подсветку. В строках 4 - 9 формируется значение переменной `$string` с помощью строковых операций и оператора условия. Эта переменная также содержит код на HTML. Однако PhpStorm не смог определить это и не выполнил подсветку. Таким образом, данная среда разработки не поддерживает обработку динамически формируемых строк со встроенным кодом. В строке 11 PhpStorm выполнил подсветку кода на SQL внутри строки, но этот код содержит ошибку, о которой статический анализатор не сообщает пользователям.

В целом можно сделать вывод, что PhpStorm предоставляет сравнительно бедный функционал по поддержке встроенного кода, выполняя в основном подсветку кода, целиком встроенного в строку, а не формируемого динамически.

### 3.1.2. IntelliLang

IntelliLang<sup>2</sup> – плагин для среды разработки IntelliJ IDEA<sup>3</sup>, который, в частности, расширяет её возможности в области работы со встроенными языками. Плагин умеет подсвечивать встроенный код и в некоторых случаях находить в нём ошибки. В отличие от предыдущего решения IntelliLang умеет обрабатывать динамически формируемые выражения. Однако особенностью работы плагина является то, что он не осуществляет поиск строк со встроенным кодом. Программист должен сам указывать, какие строки нужно анализировать. Это делает плагин не очень удобным в использовании, особенно в контексте задач реинжиниринга, обозначенных ранее.

На Рисунке 2 показано окно текстового редактора среды IntelliJ IDEA с плагином IntelliLang, где анализ выполнен только для переменной `html`.



```
7 public String getHTML() {
8     String body = "<body>" + "Hello!" + "</body>";
9     String html = "<html>" + body + "</html>";
10    return html;
11 }
```

Рисунок 2. Окно текстового редактора среды IntelliJ IDEA с плагином IntelliLang

### 3.1.3. Alvor

Alvor<sup>4</sup> [2] – плагин для среды разработки Eclipse<sup>5</sup>, предназначенный для статической проверки корректности кода на SQL, встроенного в Java. Плагин выполняет межпроцедурный анализ кода, поддерживает обработку условных

---

<sup>2</sup> Сайт плагина IntelliLang (посещён: 30.05.2015): <https://www.jetbrains.com/idea/help/intellilang.html>

<sup>3</sup> IntelliJ IDEA - среда разработки для JVM языков. Сайт (посещён: 30.05.2015): <https://www.jetbrains.com/idea/>

<sup>4</sup> Страница проекта Alvor на bitbucket.org (посещён: 30.05.2015): <https://bitbucket.org/plas/alvor>

<sup>5</sup> Сайт среды разработки Eclipse (посещён: 30.05.2015): <http://www.eclipse.org/ide/>

операторов, операций конкатенации и присваивания строк, сообщает о лексических и синтаксических ошибках. Если найдено больше одной ошибки, то показывается первая из них. Один из таких случаев представлен на Рисунке 3, где в строках 16 и 18 допущены опечатки. В строке 16 опечатка подчёркнута, а в строке 18 – нет.

```
11 public static void executeSQL(Connection connection)
12     throws SQLException{
13     String sql = "select id, first_name from person where ";
14     int a = 2;
15     if(a > 3){
16         sql += " b => 1 ";
17     }else{
18         sql += " c => 1 ";
19     }
20     sql += " order by first_name";
21     PreparedStatement preparedStatement =
22         connection.prepareStatement(sql);
23     ResultSet result = preparedStatement.executeQuery();
24 }
```

Рисунок 3. Окно текстового редактора Eclipse с плагином Alvor

К недостаткам Alvor'а можно отнести отсутствие поддержки циклов.

#### 3.1.4. Varis

Varis [3] - плагин для Eclipse, предоставляющий поддержку кода на HTML, CSS и JavaScript, встроенного в Php. В плагине реализованы функции подсветки встроенного кода, автодополнения, перехода к объявлению (jump to declaration), построения графа вызовов (call graph) для встроенного JavaScript. Рисунок 4 демонстрирует функции подсветки и автодополнения.



Рисунок 4. Окно текстового редактора Eclipse с плагином Varis

Varis – был представлен в 2015 году, и является одним из самых новых и богатых по функционалу решений в области поддержки встроенных в Php языков со стороны сред разработки.

### 3.1.5. Java String Analyzer и Php String Analyzer

Java String Analyzer<sup>6</sup> [4] и Php String Analyzer<sup>7</sup> [5] предназначены для определения корректности кода, встроенного в Java и Php соответственно. Впрочем, их подходы к определению корректности различаются. Эти инструменты легко расширяются для поддержки любых встроенных языков, однако плохо расширяются для поддержки какого-либо дополнительного функционала, кроме проверки корректности.

<sup>6</sup> Сайт проекта Java String Analyzer (посещён: 30.05.2015): <http://www.brics.dk/JSA/>

<sup>7</sup> Сайт проекта Php String Analyzer (посещён: 30.05.2015): <http://www.score.cs.tsukuba.ac.jp/~minamide/phpsa/>

### 3.1.6. Особенности существующих решений

Стоит отметить, что первые четыре решения из представленных выше ориентированы именно на поддержку встроенных языков в средах разработки. Следовательно, для них немаловажным фактором является скорость работы, для достижения которой они жертвуют качеством статического анализа, делают его менее точным. Кроме всего прочего, особенностью всех перечисленных решений является то, что каждое из них работает с каким-то одним конкретным основным языком.

## 3.2. Поиск строк со встроенным кодом

Рассмотрим алгоритм, предназначенный для определения того, какие строки в программе содержат код на встроенном языке. Этот алгоритм используется в одном из инструментов платформы YaccConstructor<sup>8</sup> [6], в рамках которой разрабатываются средства анализа встроенных языков.

Алгоритм основывается на использовании некоторого заранее заданного набора функций или методов. Будем называть их *конечными*. Заранее известно, что они ожидают строку с кодом в качестве входного аргумента. В исходном коде выполняется поиск вызовов конечных функций. Входные аргументы найденных вызовов принимаются за строки со встроенным кодом. Логика такого подхода заключается в том, что программы, формирующие код на встроенном языке, зачастую используют в конечном итоге какую-либо подсистему для его исполнения. Примером может служить метод `execute` интерфейса `java.sql.Statement`, который используется в программах на Java для исполнения SQL через технологию JDBC. Другой пример – метод `exec`

---

<sup>8</sup> Страница проекта YaccConstructor на [github.com](https://github.com) (посещён: 30.05.2015): <https://github.com/YaccConstructor>

класса PDO в языке Php, который также используется для выполнения SQL-запросов.

В целом можно отметить, что использованный подход поиска строк со встроенным кодом справляется со своей задачей далеко не всегда, однако сравнительная простота реализации является его несомненным преимуществом.

### 3.3. Построение регулярной аппроксимации сверху

В данном разделе даётся обзор алгоритма построения регулярной аппроксимации, описанного в статье [7]. Она посвящена анализу строковых выражений, формируемых кодом на Php, и выявления среди них таких, которые могут использоваться для вредоносных целей. Описанный алгоритм особенно интересен, так как гарантирует построение регулярной аппроксимации сверху, при этом умеет обрабатывать многие конструкции основного языка.

Для начала введём несколько базовых понятий, которые будут использоваться при рассмотрении алгоритма.

*Граф потока управления (control flow graph, CFG)* [8] – множество всех возможных путей исполнения программы, представленное в виде связного ориентированного графа. В его вершинах находятся выражения из исходного кода программы, а рёбра соединяют вершины так, чтобы показывать порядок, в котором будут вычисляться выражения в процессе работы программы. Условным операторам, задающим альтернативные пути исполнения, соответствуют разветвления в графе потока управления, а циклам в коде – циклы в графе. У графа есть одна *входная* и ноль или более *выходных вершин*. *Входная* имеет только исходящие рёбра, выходные –

только входящие. Упрощённый пример такого графа для фрагмента кода на Листинге 1 показан на Рисунке 5.

*Целевое выражение* – строковое выражение в исходном коде программы, множество значений которого мы хотим построить.

*Целевая вершина* – вершина в CFG, содержащая целевое выражение.

*Граф зависимостей данных (data-dependency graph, DDG)* – ориентированный граф, представляющий зависимости между инструкциями в исходном коде. Вершины графа соответствуют выражениям из исходного кода. Наличие направленного ребра из вершины X в вершину Y означает, что для вычисления выражения, которому соответствует вершина Y, необходимо сначала вычислить выражение в вершине X. На Рисунке 6 показан пример графа зависимостей для ссылки на переменную sql в аргументе метода Db.execute.

```
1 | Entries getEntries(boolean cond) {  
2 |     String logMsg = "Selected";  
3 |     String sql = "SELECT * FROM ";  
4 |     if(cond)  
5 |         sql = sql + "Table1";  
6 |     else  
7 |         sql = sql + "Table2";  
8 |     Console.WriteLine(logMsg);  
9 |     return Db.execute(sql);  
10| }
```

Листинг 1. Фрагмент кода на C#

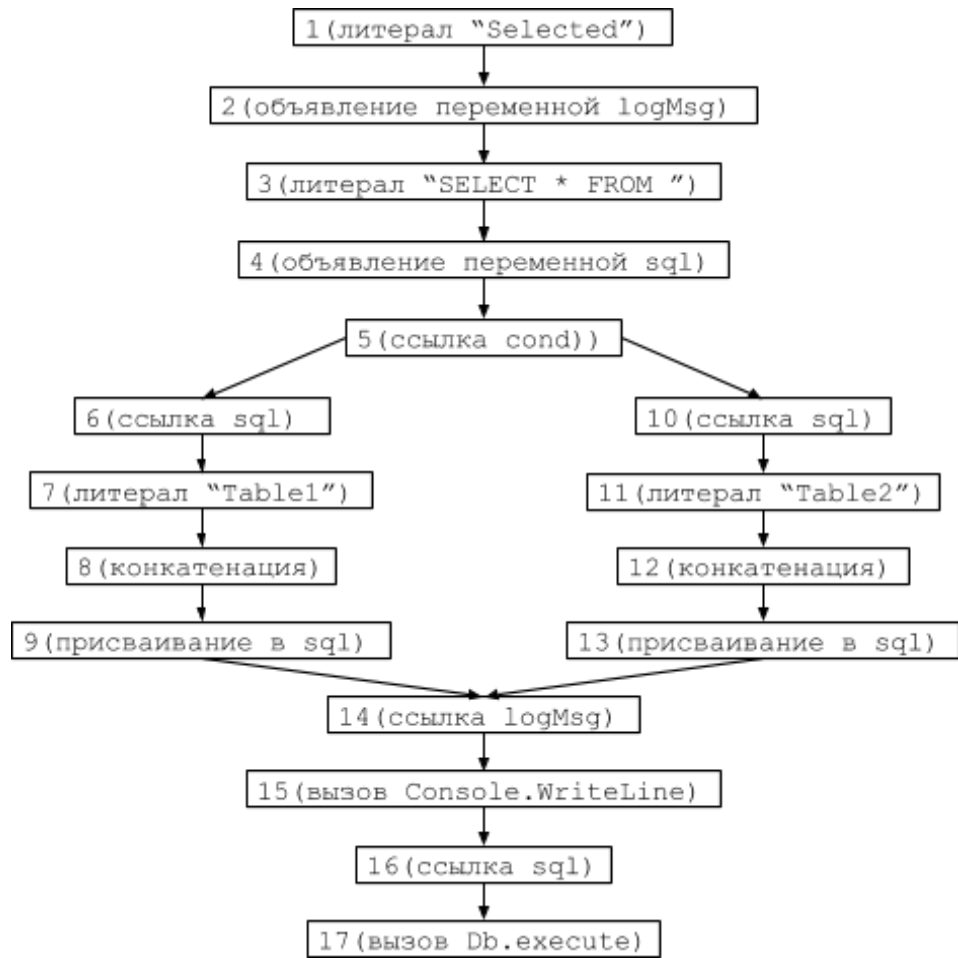


Рисунок 5. Граф потока управления для фрагмента кода на Листинге 1



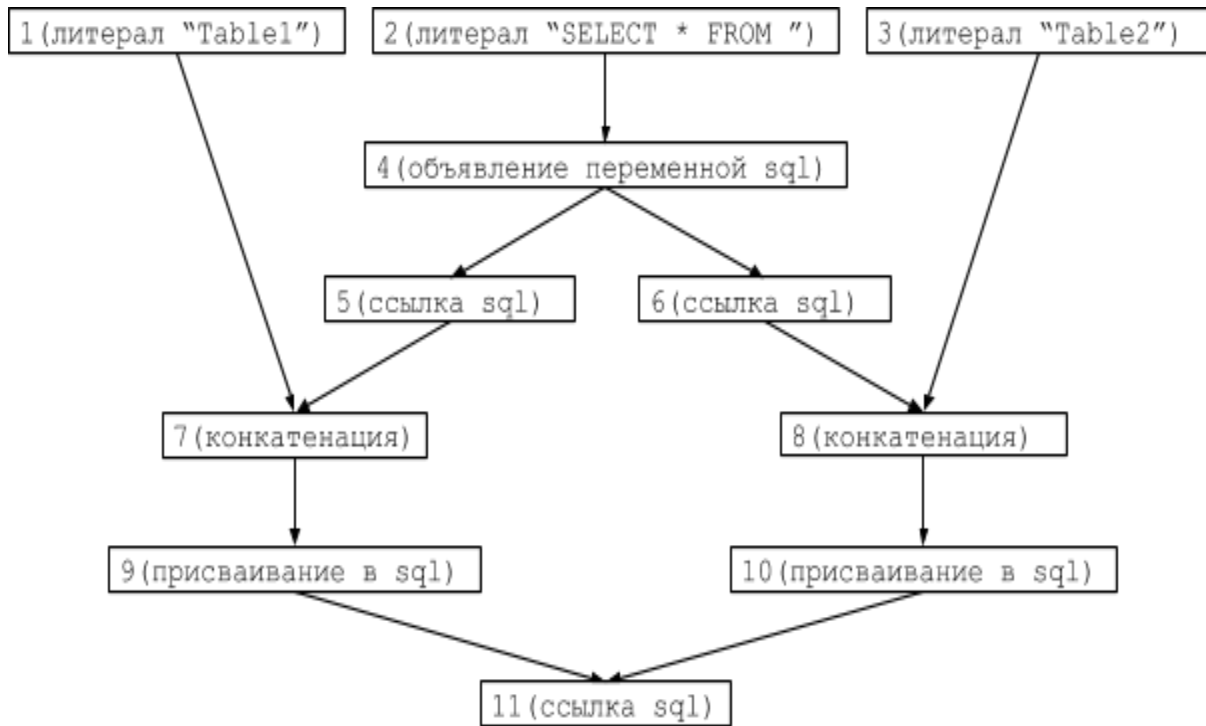


Рисунок 6. Граф зависимостей данных построенный для вершины 16 графа потока управления, изображённого на Рисунке 5.

В дальнейших рассуждениях касающихся конечных автоматов принято следующее соглашение. Предполагается, что любой автомат имеет некоторое дополнительное состояние, которое не указано явно. Назовём его *sink-состоянием*. Его особенность заключается в том, что оно не является конечным и зациклено на себе по всем символам алфавита. При упоминании конечных автоматов в тексте работы и на иллюстрациях, для их состояний будут указываться переходы лишь по некоторым символам алфавита. Если у состояния автомата, обозначенного явно, не указан переход по какому-либо символу алфавита, то предполагается, что переход по данному символу осуществляется в *sink-состояние* (см. Рис. 7). Sink-состояние и переходы в него опускаются для упрощения описаний и рисунков.

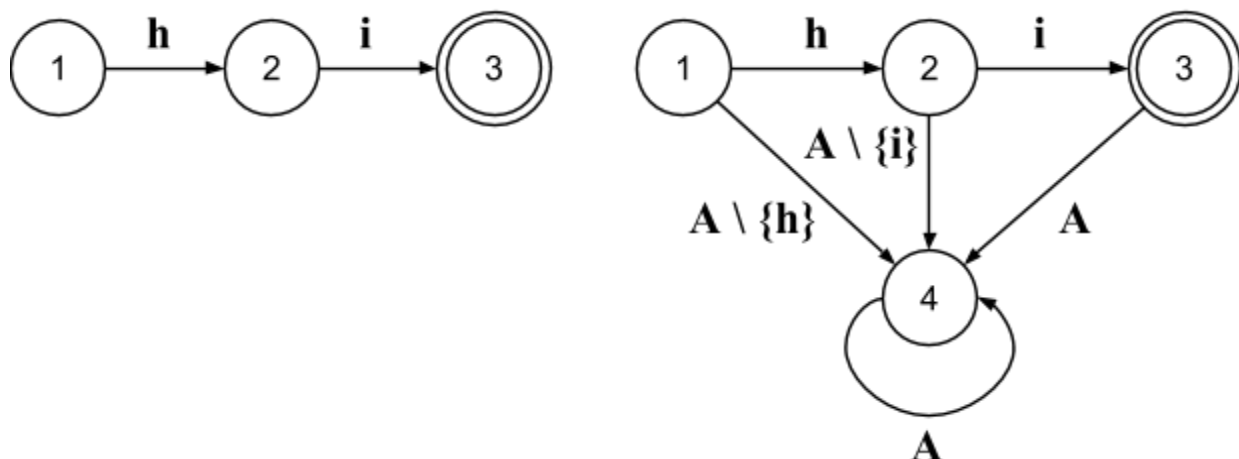


Рисунок 7. Упрощённый конечный автомат (слева) и соответствующий ему полностью определённый с дополнительным sink-состоянием (справа) над алфавитом  $A$ .

Перейдём к описанию алгоритма построения регулярной аппроксимации, описанного в статье [7]. Он состоит из следующих шагов:

1. Для исходного кода строится граф потока управления;
2. По графу потока управления и некоторой целевой вершине в нём строится граф зависимостей данных;
3. По графу зависимостей данных строятся конечные автоматы для выражений в вершинах, которые аппроксимируют множество значений этих выражений сверху. Автомат, полученный для целевой вершины, является результатом работы алгоритма;

Остановимся подробнее на третьем шаге и опишем, как именно в процессе обхода графа выражениям в вершинах сопоставляются конечные автоматы.

1. Строковые литералы. Конечный автомат для строкового литерала – это автомат, который принимает одно слово. Пример автомата для строки “hi” изображён на Рисунке 7.

2. Конкатенация строк. Для выполнения конкатенации берутся два автомата, соответствующие аргументам операции. На их основе строится конечный автомат, принимающий строки, состоящие из двух частей: префикса и суффикса. Префиксом является любое слово, которое принимает первый автомат-аргумент, суффиксом – любое слово, которое принимает второй автомат-аргумент (см. Рис. 8).

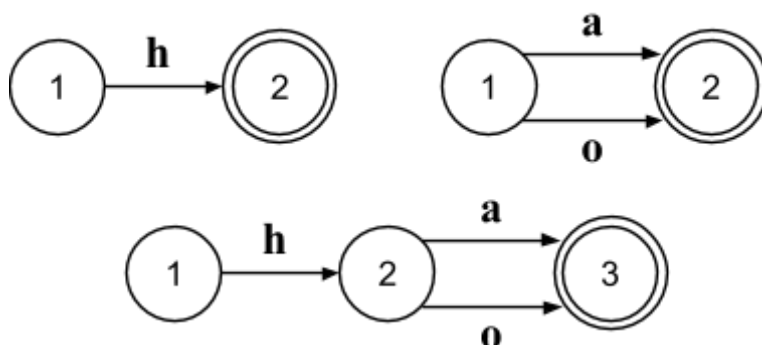


Рисунок 8. Два конечных автомата (сверху) и результат их конкатенации (снизу)

3. Замена в строках (replace). Операция замены принимает три автомата-аргумента (см. пример на Рис. 9). Первый содержит строки, в которых нужно произвести замену (левый верхний автомат на рисунке); второй – строки, каждую из которых нужно попытаться выделить в качестве подстроки в любой из строк первого автомата (средний верхний автомат на рисунке); третий – множество строк, на которое нужно заменить каждое найденное вхождение подстрок (правый верхний автомат на рисунке).

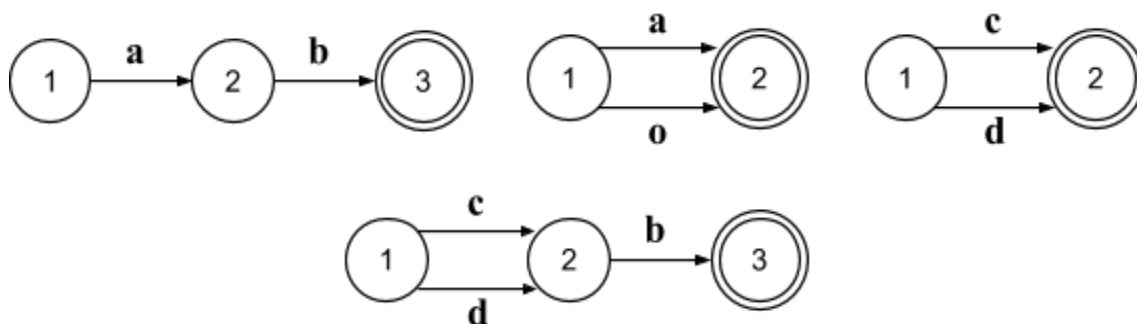


Рисунок 9. Три автомата (вверху) и результат применения операции замены к ним (внизу)

4. Оператор условия (if, if-else). Пусть в графе зависимостей данных есть вершина A. Если выражение в вершине A зависит от некоторого выражения, которое может формироваться в разных блоках оператора условия, то в графе есть несколько путей, ведущих в вершину A. В каждом из них формируется выражение одного и того же типа. Для того чтобы учесть каждое из них как одно из возможных, выполняется объединение автоматов, получающихся в конце каждого пути, ведущего в A.

5. Циклы. Обработка циклов сопряжена с рядом проблем. Во-первых, как уже было упомянуто ранее, на этапе статического анализа далеко не всегда можно определить, сколько раз будет выполняться тело цикла, поэтому в общем случае он порождает бесконечный язык (бесконечное множество строк). Во-вторых, этот язык может быть нерегулярным, что демонстрируется на Листинге 2. Данный фрагмент кода генерирует строки из множества  $\{a^n b^n \mid n = 1, 2, \dots\}$ , которое является классическим примером нерегулярного языка.

```
1 | String str = "ab";  
2 | for(...) {  
3 |     str = str.Replace ("ab", "aabb");  
4 | }  
5 | Db.Execute(str);
```

Листинг 2. Цикл с заменой на C# (упрощённо)

Для преодоления этих двух сложностей используется оператор расширения. Оператор принимает в качестве входных аргументов два автомата и на объединённом множестве состояний этих автоматов задаёт некоторое отношение эквивалентности. Классы эквивалентности

относительно данного отношения используются как состояния нового автомата, переходы строятся на основе переходов в исходных автоматах. Полученный автомат является результатом работы оператора. Особенностью такого расширенного автомата является то, что он принимает все слова исходных автоматов и, возможно, некоторые другие, в некотором смысле обобщая при этом исходные автоматы.

Оператор используется для двух автоматов, являющихся результатами соседних итераций цикла, следующим образом:

$$A'_i = \nabla(A_i \cup A_{i-1}, A_{i-1}),$$

где  $A_i$ ,  $A_{i-1}$  – результирующие автоматы текущей и предыдущей итераций,  $\nabla$  – оператор расширения. Автомат  $A'_i$  принимается за новый результат текущей итерации. Важной особенностью является то, что последовательность  $A'_i$ ,  $i = 0, 1, \dots$  гарантированно сходится [7], то есть начиная с некоторого  $m$ :

$$\forall i > m : A'_i = A'_{i+1} = R.$$

Запись  $A'_i = A'_{i+1}$  означает, что автоматы эквивалентны, то есть задают один и тот же регулярный язык. Автомат  $R$  является результатом аппроксимации цикла.

Рассмотрим пример использования оператора расширения для цикла на Листинге 3. Процесс построения автоматов показан на Рисунке 10. Для каждой итерации показаны следующие автоматы: исходный – получается в результате выполнения итерации цикла; объединение – получается объединением исходного и того, что был получен на предыдущей итерации; расширение – получается применением оператора расширения к результату объединения и автомату с прошлой итерации. Если на рисунке присутствует пунктирная стрелка из автомата  $A$  в  $C$  и из  $B$  в  $C$ , то  $C$  получен из  $A$  и  $B$

путем применения той операции, которая указана в строке где находится С (к примеру, объединением).

Из примера видно, что последовательность расширенных автоматов сходится уже на третьей итерации. Таким образом, расширенный автомат третьей итерации, а именно, принимающий множество строк  $\{ab^n \mid n = 0, 1, 2, \dots\}$ , является аппроксимацией цикла на Листинге 3.

```

1 | String str = "a";
2 | for(...) {
3 |     str += "b";
4 | }
5 | Db.Execute(str);

```

Листинг 3. Цикл с конкатенацией на C# (упрощённо)

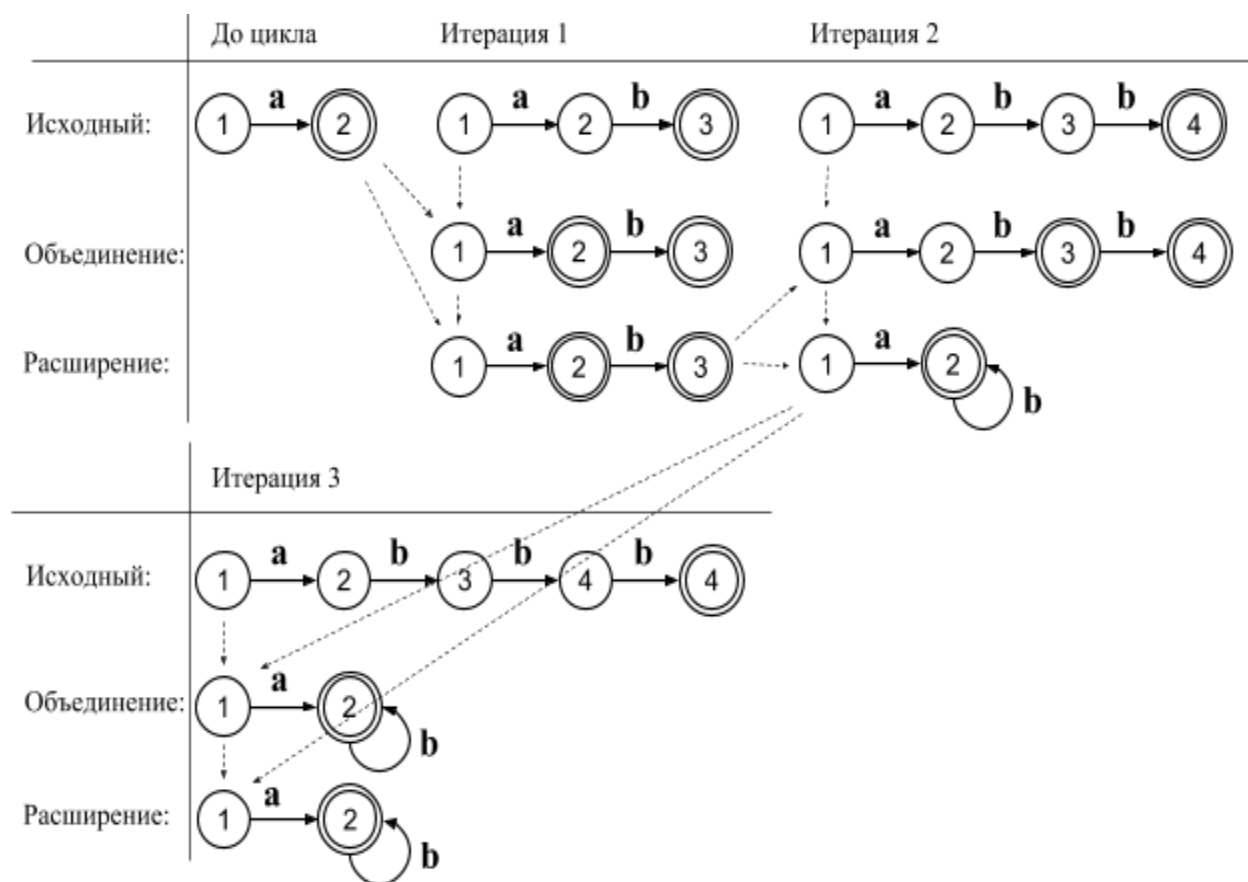


Рисунок 10. Построение аппроксимации для цикла на Листинге 3.

6. Функции ввода. Функции ввода данных (к примеру, чтение из консоли или из файла) могут возвращать любые строки и на этапе статического анализа эти значения предсказать невозможно. Поэтому считается, что конечный автомат для функций ввода данных – это автомат, который принимает произвольные строки.

Формальные определения операций конкатенации, замены и расширения для конечных автоматов даны в статье [7].

### 3.4. Платформа для реализации и вспомогательные технологии

Описанный алгоритм построения аппроксимации используется в качестве основы для инструмента, разрабатываемого в рамках данной работы. Для его адаптации используются следующие технологии.

*YaccConstructor* – платформа для исследований в области анализа языков и построения соответствующих инструментов. Статический анализатор встроенного кода разрабатывается именно в рамках данной платформы, поэтому часть, связанная с построением аппроксимации, должна быть внедрена в неё. В качестве основного языка реализации используется F#, на нём написана большая часть *YaccConstructor*.

*ReSharper SDK* – набор средств разработки для плагина ReSharper<sup>9</sup>. ReSharper используется для повышения производительности при разработке приложений в среде Microsoft Visual Studio. Немалая часть функционала в нём основана на статическом анализе. В данной работе с помощью ReSharper SDK строится абстрактное синтаксическое дерево (abstract syntax tree, AST) и граф потока управления для исходного кода на основном языке. Они являются исходными данными для инструмента построения: в частности,

---

<sup>9</sup> Сайт плагина ReSharper (посещён: 30.05.2015): <https://www.jetbrains.com/resharper/>

графы потока управления нужны для реализации описанного выше алгоритма.

*QuickGraph*<sup>10</sup> – библиотека для работы с графами на платформе .NET. Все графы, которые используются в работе, реализованы на основе так называемого двунаправленного графа (BidirectionalGraph) из QuickGraph. Двунаправленный граф является ориентированным. В каждой его вершине хранятся как входящие, так и исходящие рёбра. Это очень полезно, так как нам требуется выполнять как прямой, так и обратный обход графов.

*Библиотека для работы с конечными автоматами из YaccConstructor* – библиотека для представления конечных автоматов и выполнения операций над ними, которая входит в состав YaccConstructor.

---

<sup>10</sup> Страница проекта Quickgraph на [www.codeplex.com](http://www.codeplex.com) (посещён: 30.05.2015): <https://quickgraph.codeplex.com/>



## 4. Реализация

### 4.1. Общая схема работы инструмента

Реализованный в рамках данной работы инструмент состоит из front-end'а и back-end'а (см. Рис. 11).

Back-end реализует алгоритм построения аппроксимации, не зависящий от основного языка, адаптируя описанные ранее идеи алгоритма из статьи [7]. Входные данные для back-end'а – обобщённый граф потока управления с выделенной целевой вершиной. Результат работы – конечный автомат, который является регулярной аппроксимацией целевого выражения.

Front-end на основе конкретного основного языка подготавливает входные данные для back-end'а. Входные данные front-end'а – файл с исходным кодом на конкретном языке, в котором содержится встроенный код. Результат работы – обобщённый граф потока управления с выделенной целевой вершиной. В данной работе front-end реализован для языков C# и JavaScript. В реализации используется ReSharper SDK, в котором поддерживаются указанные языки.

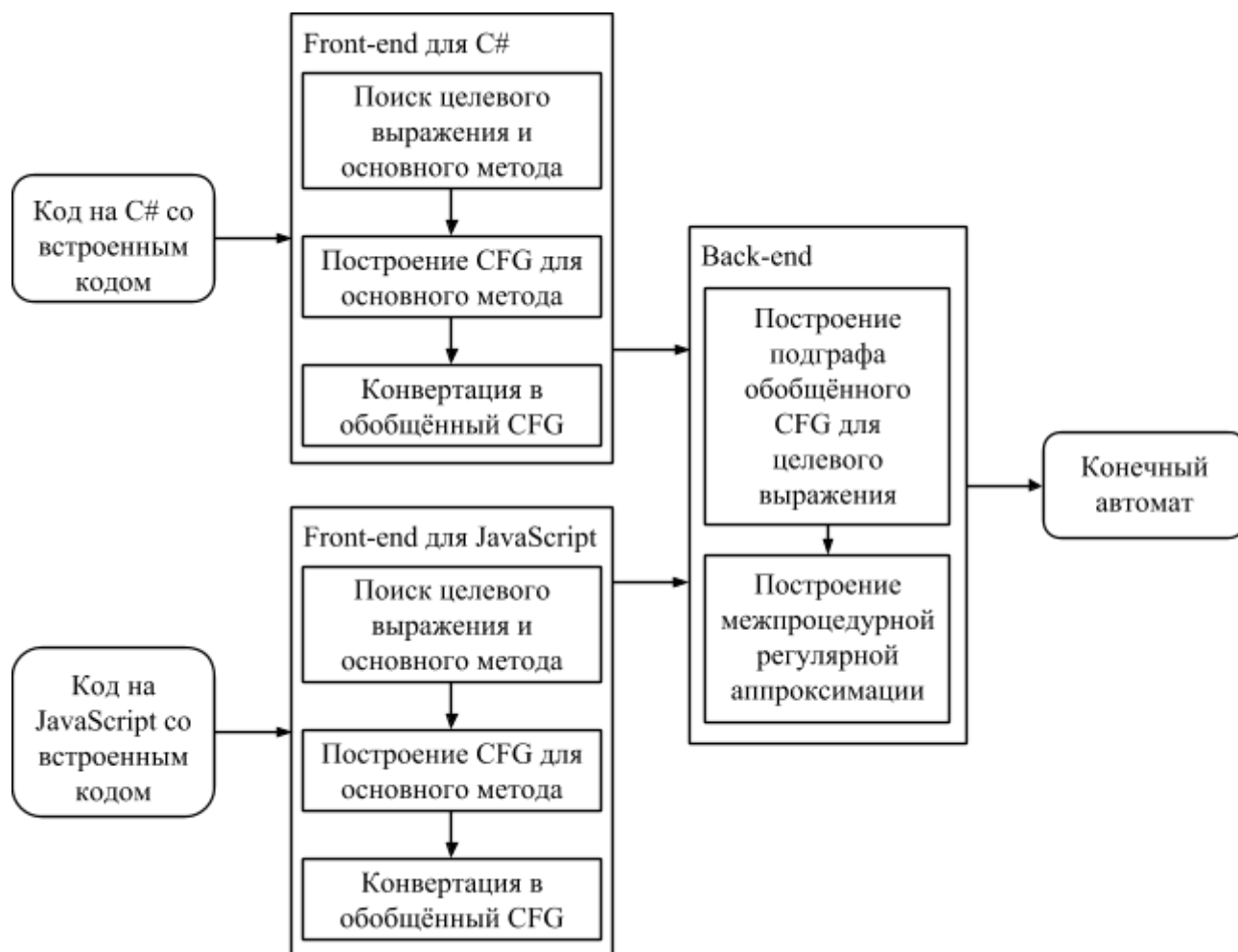


Рисунок 11. Архитектура разработанного инструмента построения регулярной аппроксимации встроенных языков

Рисунок 12 иллюстрирует место реализованного инструмента в общей схеме статического анализатора встроенных языков, разрабатываемого в YaccConstructor.

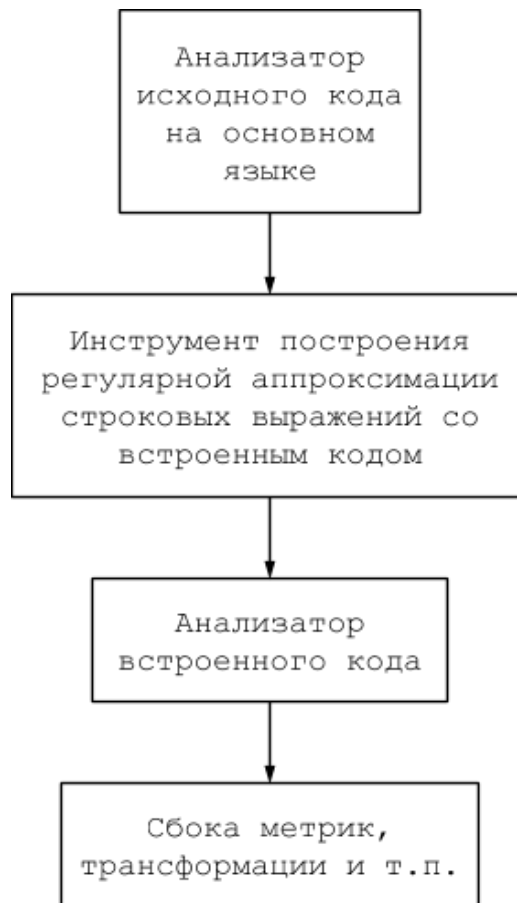


Рисунок 12. Место инструмента построения аппроксимации в общей схеме статического анализатора

В текущей реализации инструмента предполагается, что файлы с исходным кодом, которые поступают на обработку, организованы следующим образом. В файле имеется метод (или функция), в пределах которого начинается и заканчивается построение кода на встроенном языке, то есть он в своей работе не опирается на входные параметры. Такой метод (или функцию) мы будем называть *основным*. При этом основной метод может использовать вызовы других методов. Они будут корректно обработаны, и их участие в формировании встроенного кода будет учтено. Организация программы, при которой встроенный код начинает формироваться в одном методе, а заканчивает – в другом, не поддерживается

инструментом. Обработка такой организации может быть реализована практически тем же способом, которым обрабатываются вызовы методов из основного. Однако это является уже скорее задачей построения конечного плагина, тогда как данный проект ориентирован в основном на реализацию базового алгоритма аппроксимации.

В дальнейшем построение аппроксимации не только в пределах основного метода, но и с обработкой вызываемых методов, будет называться *межпроцедурной аппроксимацией*.

Общую схему работы инструмента можно разбить на следующие этапы:

1. Обработка всего файла:
  - 1.1. Построение AST для исходного кода;
  - 1.2. Поиск строк со встроенным кодом по AST и определение основного метода (или функции);
2. Обработка основного метода:
  - 2.1. Построение CFG для основного метода;
  - 2.2. Конвертация специфичного для языка CFG в обобщённый CFG;
  - 2.3. Построение подграфа обобщённого CFG для выражений, найденных на этапе 1.2;
  - 2.4. Построение конечного автомата по подграфу обобщённого CFG;
3. Обработка методов, вызываемых из основного:
  - 3.1. Построение CFG для вызываемого метода;
  - 3.2. Конвертация специфичного для языка CFG в обобщённый CFG;
  - 3.3. Построение подграфа обобщённого CFG для выражения, возвращаемого вызываемым методом;
  - 3.4. Постобработка подграфа обобщённого CFG для борьбы с хвостовой рекурсией;
  - 3.5. Построение конечного автомата по подграфу обобщённого CFG.

Рассмотрим указанные этапы подробнее.

#### 4.2. Поиск строк со встроенным кодом по AST

Реализация поиска строк на основе алгоритма, описанного ранее, уже имела в составе YaccConstructor для языка C#. В рамках данной работы были проведены незначительные технические улучшения реализации. Кроме того, написана реализация для языка JavaScript.

В реализации множество конечных функций и методов описывается в конфигурационном xml файле. В нём хранятся описания сигнатур (полное имя, возвращаемое значение), а также указание, какой именно из аргументов должен содержать строку с кодом. В AST, построенном с помощью ReSharper SDK, выполняется поиск вершин, соответствующих вызовам методов или функций. Каждый вызов проверяется на соответствие одной из записей в конфигурационном файле. Если запись найдена, то вызываемый метод считается конечным, соответствующий аргумент – целевым выражением, а метод, в пределах которого вызывается конечный, – основным. Для дальнейшего построения CFG из AST также извлекается определение основного метода.

Как отмечалось ранее, данный алгоритм справляется со своей задачей не всегда и выбран из-за простоты реализации. Однако поиск реализован в виде отдельной функции, которая предоставляет AST найденного основного метода и целевое выражение в нём как результат своего выполнения другим частям инструмента. Поэтому замена текущего алгоритма поиска на более совершенный, при необходимости, не составит труда.

Поиск реализован как часть front-end'a, то есть отдельно для каждого языка, и в рамках работы не предпринималась попытка обобщить его. Это связано с тем, что для обобщения потребуется обобщённое взаимодействие

со специфичными для разных языков AST, организация которого является непростой задачей. С другой стороны, back-end работает с обобщённым CFG. На его основе можно организовать обобщённый поиск. Однако это потребует построения специфичных для языка CFG и конвертации в обобщённый для всего исходного кода, а не только для участков, принимающих участие в формировании встроенного кода. Это плохо скажется на производительности работы инструмента.

#### 4.3. Обобщённый CFG

С помощью ReSharper SDK на основе AST основного метода и целевого выражения в нём строится CFG, в котором отмечается целевая вершина. ReSharper SDK строит разные CFG для C# и JavaScript. Они отличаются способом хранения информации в вершинах, а также незначительно отличаются структурой. Это же может быть верно и для других языков, тем более, если строить CFG с помощью других инструментов.

Для того, чтобы алгоритм аппроксимации не зависел от того, с каким основным языком он работает, был реализован обобщённый CFG, в который конвертируются специфичные для языков CFG. Обобщённый CFG имеет ту же самую структуру, что и специфичный для языка, то есть то же множество вершин и соединяющих их рёбер, за исключением того, что вершины заменяются на обобщённые. В Таблице 1 приведены все типы обобщённых вершин и те выражения исходного кода (а, следовательно, и вершины специфичных для языков CFG), которым они соответствуют.

Таблица 1. Типы вершин обобщённого CFG

Тип обобщённой вершины	Какие выражения исходного кода обобщает	Пояснения
Declaration (Объявление)	Объявления переменных	Объявление любой переменной в коде представляется вершиной этого типа
Updater (Обновление)	Присваивание, присваивание с конкатенацией (+=)	Для присваивания и присваивания с конкатенацией (+=) используются разные разновидности вершины этого типа
Operation (Операция)	Вызовы функций и операторов	Функции, имеющие семантику замены в строках, представляются специальным подтипом Replce, функции и операторы конкатенации (+) - подтипом Concat, все остальные функции и операторы - подтипом Arbitrary
Literal (Литерал)	Литералы	Представляет литералы, в частности, строковые
VarRef	Ссылка на переменную	Представляет любые ссылки на переменные, которые используются в коде
LoopNode (Вершина цикла)	Первое выражение цикла	Первое выражение, которое встречается на пути выполнения программы и которое принадлежит какому-либо циклу. Чаще всего им является выражение, которое соответствует условию цикла, так как это первое, с чего он начинается
LoopEnter (Вход в цикл)	-	Тип для служебных вершин-меток
LoopExit (Выход)	-	Тип для служебных вершин-меток

из цикла)		
LoopBodyBeg (Вход в тело цикла)	-	Тип для служебных вершин-меток
LoopBodyEnd (Выход из тела цикла)	-	Тип для служебных вершин-меток
StartNode (Стартовая вершина)	-	Входная вершина графа, не имеющая входных рёбер.
ExitNode (Конечная вершина)	-	Выходная вершина графа, не имеющая исходящих рёбер
OtherNode (Другая вершина)	Любое выражение	Используется для выражений, не попадающих под описанные выше категории

Обобщённые вершины содержат специфичную для их типа информацию, необходимую для дальнейшего построения аппроксимации. В целом, типы выбраны таким образом, чтобы представлять основные манипуляции со строками, которые могут происходить в программах. К примеру, имеется отдельный тип для операции плюс-равно ( $+=$ ), но нет типа для минус-равно ( $-=$ ), так как последняя является довольно нетипичной при работе со строками, тогда как первая используется часто. Тем не менее, если язык имеет нетипичные строковые операции, их можно представить, используя подтип Arbitrary типа Operation.

Вершины-метки, указанные в таблице, используются для упрощения работы с циклами. Если в процессе обхода графа встречается вершина типа LoopNode, необходимо иметь возможность выбирать, в каком направлении



обходить граф дальше: посетить тело цикла или выйти из него. Для этого нужно знать, какие из рёбер вершины LoopNode ведут в тело цикла, а какие – нет. Вершины-метки используются для кодирования именно этой информации.

Рисунок 13 иллюстрирует, как это происходит. Слева изображён граф потока управления для цикла без меток, справа – с метками. Если обход достиг вершины LoopNode в правом графе, необходимо лишь проверить типы смежных с ней вершин для того, чтобы выбрать дальнейшее направление обхода. Метки LoopExit и LoopBodyBeg используются при прямом обходе графа, LoopEnter и LoopBodyEnd – при обратном.

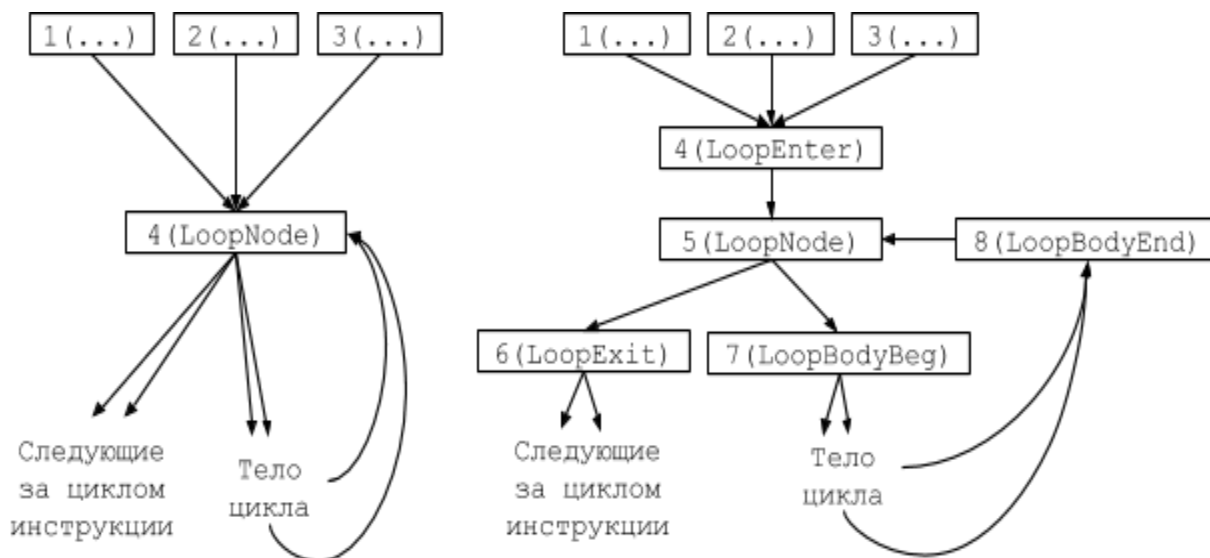


Рисунок 13. Цикл в графе потока управления (с метками и без)

Добавление меток происходит при конвертации специфичных для языков CFG в обобщённый. Конвертация выполняется с помощью обхода графа в глубину. Специфичной для конкретного языка по сути является лишь функция, которая конвертирует очередную посещённую вершину в обобщённую.

В заключение стоит отметить, что реализованный обобщённый CFG не претендует на абсолютную универсальность, то есть возможность

представлять программы на совершенно произвольных языках. Задача построения такого CFG сама по себе является весьма содержательной и нетривиальной. Однако реализованный вариант вполне пригоден для работы с императивными языками, что было продемонстрировано удачной реализацией конвертации для C# и JavaScript.

#### 4.4. Особенности обработки графов

Алгоритм построения аппроксимации из статьи [7] предполагает построение конечных автоматов по DDG, который строится на основе CFG. DDG строится для того, чтобы исключить из рассмотрения те выражения, которые не принимают участия в формировании значений целевого. Авторы статьи при реализации инструмента, основанного на предложенном ими алгоритме, опирались на готовую библиотеку для построения DDG. В нашем случае для обобщённого CFG такой библиотеки нет, а значит построение DDG нужно реализовать самостоятельно.

DDG и CFG имеют разную структуру, поэтому для построения DDG необходимо уметь конвертировать одну структуру графа в другую, что вызывает ряд неудобств при реализации. С другой стороны, зависимости выражений друг от друга присутствуют в CFG по определению, поэтому можно строить конечные автоматы прямо по нему, избежав этапа конвертации. Однако в этом случае будут обрабатываться и те выражения, которые не влияют на формирование целевого.

Для того, чтобы избежать необходимости конвертации графа одной структуры в граф другой структуры, сохранив при этом преимущество использования DDG, в данной работе выделяется подграф CFG, состоящий из всех возможных путей, по которым можно из входной вершины CFG добраться до целевой вершины. При этом из путей удаляются те участки,

которые соответствуют выражениям, не принимающим участия в формировании целевого. Назовём его *подграфом потока управления для целевого выражения* (*targeted control-flow subgraph, TCFS*).

В процессе построения TCFS по CFG потребовался *обход графа в порядке топологической сортировки со специальной обработкой циклов*. Опишем его идею. Известно, что топологическая сортировка возможна лишь для ациклических графов. В CFG же могут быть циклы. Однако, наиболее распространённые виды циклов, используемые в программах, имеют одну точку входа (см. Рис. 14). При этом, из рассмотрения исключаются программы, в которых используются операторы перехода типа *goto*.

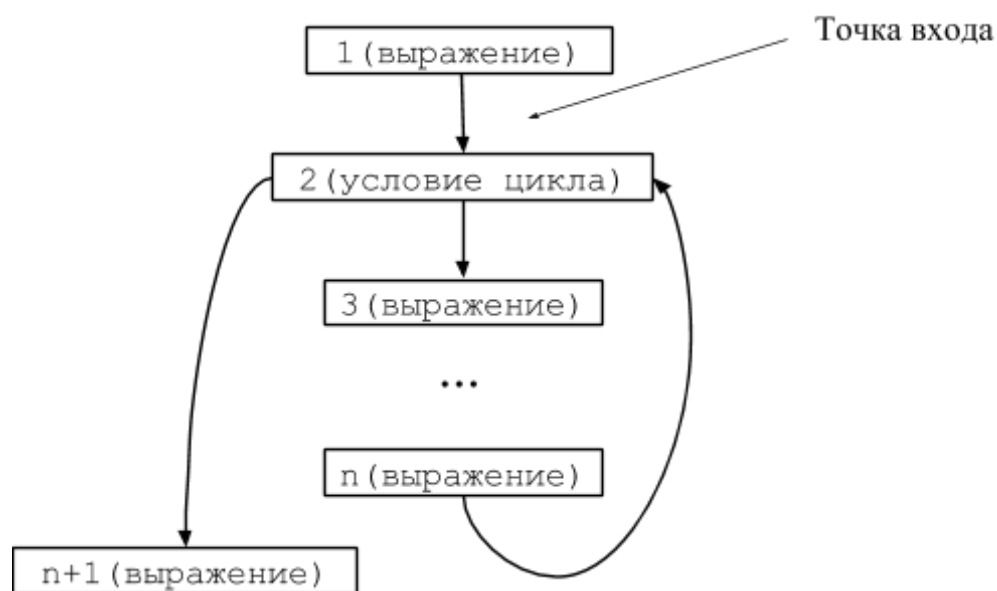


Рисунок 14. Структура графа потока управления для цикла

В связи с этим граф для фрагмента кода программы с одним циклом можно разбить на несколько частей. Первая – ациклический граф, состоящий из всех путей из начальной вершины в вершину, соответствующую точке входа в цикл. Вторая – тело цикла, то есть ациклический граф состоящий из всех путей, которые начинаются в вершинах, соответствующих первым выражениям в теле, и заканчиваются в вершинах, соответствующих

последним выражениям теле. Третья – ациклический граф, состоящий из путей, которые начинаются в теле цикла и заканчиваются в выходных вершинах графа. Каждая часть обходится в порядке топологической сортировки, а в точке входа в цикл принимается решение о том, какую часть следует обойти дальше. Ясно, что если какая-либо из частей содержит цикл, то её можно также разбить на части, сведя таким образом задачу её обхода к описанной только что.

Приведённый алгоритм реализован как простейшая модификация алгоритма Кана [9]. Алгоритм Кана предназначен для выполнения топологической сортировки. Он достаточно прост в реализации за счёт того, что требует хранения в вершинах графа дополнительной информации, а именно: количества входящих рёбер вершины. Однако, как было упомянуто ранее, для представления графов в данной работе используются двунаправленные графы из библиотеки QuickGraph, в которых эта информация имеется. Кроме всего прочего, этот же самый способ обхода удачно применён при обходе TCFS в процессе построения конечных автоматов.

#### 4.5. Построение TCFS

TCFS строится для того, чтобы исключить из дальнейшего рассмотрения часть CFG, не принимающую участия в формировании множества значений выражения выделенной вершины.

Для построения TCFS выполняется обход обобщённого CFG в порядке топологической сортировки со специальным учётом циклов, описанный ранее, начиная с выделенной вершины. Обход выполняется в обратную сторону, то есть против направления рёбер. Для каждой посещённой вершины принимается решение, вносит ли она вклад в формирование

целевого выражения, и, если вносит, то она включается в TCFS. Решение принимается для вершин в зависимости от их типа и дополнительной информации, которая накапливается в процессе обхода. Дополнительная информация – это список переменных, которые участвуют в формировании значений целевого выражения (обозначим его VL, variables list) и список вершин, от которых зависят уже включённые в TCFS вершины (обозначим его NL, nodes list). Для того, чтобы понять, как работает алгоритм, рассмотрим пример.

```
1 | Entries getEntries() {  
2 |     ...  
3 |     String logMsg = "Selected";  
4 |     String sql = "SELECT * FROM " + table;  
5 |     Console.WriteLine(logMsg);  
6 |     return Db.execute(sql);  
7 | }
```

Листинг 4. Фрагмент кода на C#

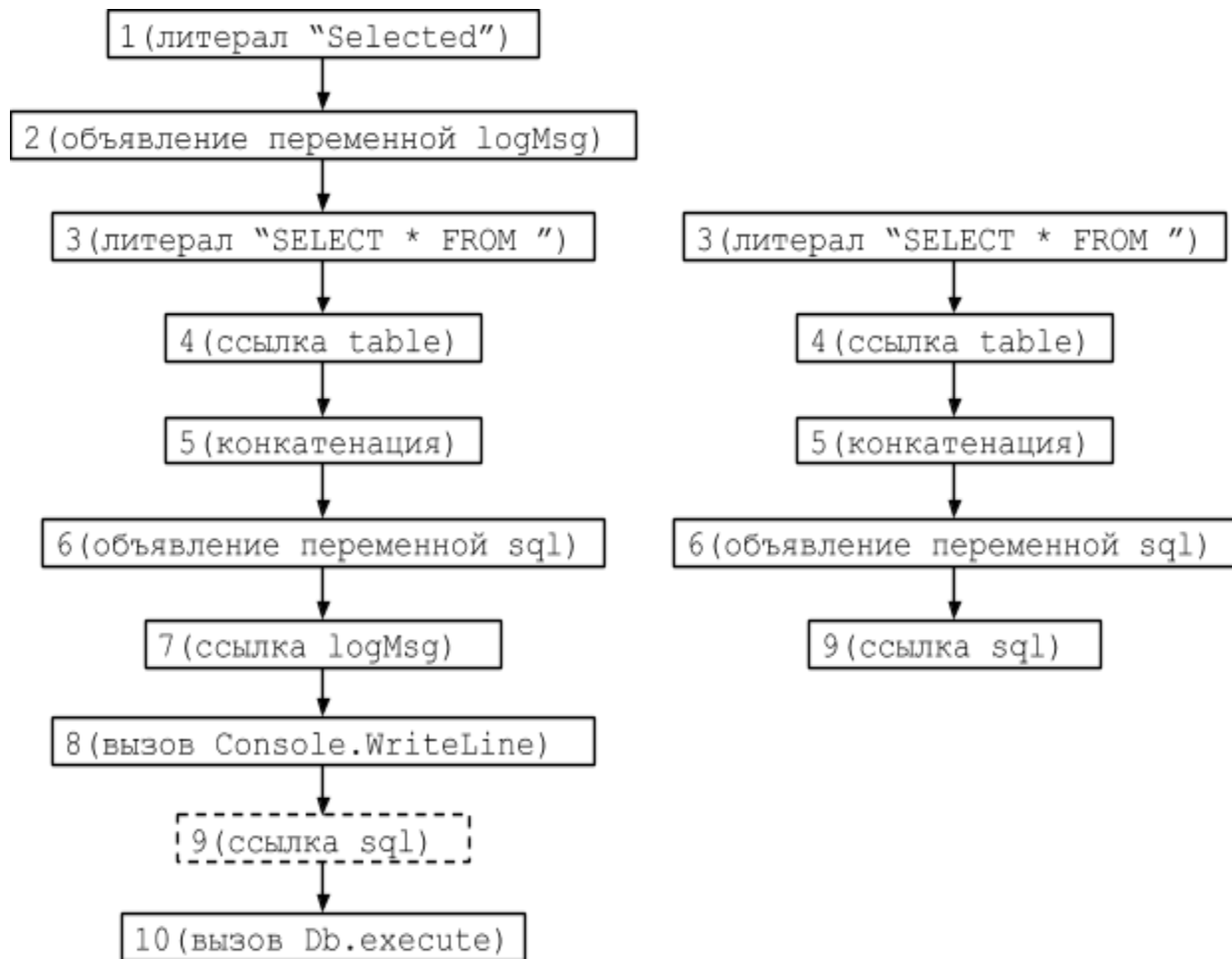


Рисунок 15. CFG с выделенной вершиной (слева) и соответствующий ему TCFS (справа)

Слева на Рисунке 15 представлен обобщённый CFG с выделенной вершиной, построенный для фрагмента кода на Листинге 4. Алгоритм будет обрабатывать каждую из вершин следующим образом.

*Шаг 1.* Вершина 9 – ссылка на переменную sql. Это первая вершина обхода. Она содержит целевое выражение, и поэтому гарантировано включается в TCFS. Имя переменной sql добавляется в VL.

*Шаг 2.* Вершина 8 – вызов метода. Вершина такого типа напрямую не влияет на формирование переменных из VL, поэтому данная вершина не включается в TCFS.

*Шаг 3.* Вершина 7 – ссылка на переменную logMsg. Данная переменная не влияет на формирование переменных из VL, и вершина не включается в TCFS.

*Шаг 4.* Вершина 6 – объявление переменной, имя которой имеется в VL. Данная вершина включается в TCFS, так как представляет манипуляцию с интересующей нас переменной. Кроме того, вершина типа “объявление” хранит номер вершины, соответствующей инициализатору при объявлении. В данном случае – это конкатенация в вершине 5. Число 5 добавляется в NL.

*Шаг 5.* Вершина 5 – конкатенация. Вершина данного типа напрямую не влияет на переменные из VL, однако её номер есть в NL, то есть от неё зависит какая-то вершина, уже включённая в TCFS. Поэтому данная вершина также включается в TCFS. Вершина типа “конкатенация” хранит номера вершин-аргументов. В данном случае – это номера 4 и 3. Эти числа добавляются в NL.

*Шаг 6.* Вершина 4 – ссылка на переменную table. Данная переменная напрямую не влияет на переменные в VL. Но номер вершины имеется в NL, так что данная вершина включается в TCFS. Кроме того, имя переменной добавляется в VL, так как она влияет на формирование переменной из VL, а значит манипуляции с ней нас тоже интересуют.

*Шаг 7.* Вершина 3 – литерал. Литералы напрямую не влияют на формирование переменных из VL, но номер вершины есть в NL, так что она включается в TCFS.

*Шаг 8 и 9.* Вершины 2 и 1 напрямую не влияют на вершины из VL, и их номеров нету в списке NL, так что они не включаются в TCFS.

В итоге мы получаем TCFS показанный на Рисунке 15 справа. Заметим, что TCFS не может содержать вершины типа OtherNode, так как они не содержат какой-либо информации и не могут вносить непосредственного

вклада в формирование переменных. Они также не используются в качестве вершин, от которых зависят какие-либо вершины CFG.

Отметим особенность обработки циклов в описываемом алгоритме. В некоторых ситуациях, обход тела цикла нужно выполнять несколько раз, чтобы корректно включить все нужные вершины. Ситуация, в которой это необходимо, иллюстрируется на Листинге 5 и Рисунке 16. Вершины-метки не изображаются для упрощения.

```
1 | for(...) {  
2 |     ...  
3 |     s += t;  
4 |     t += "b";  
5 |     ...  
6 | }  
7 | Db.Execute(s);
```

Листинг 5. Фрагмент кода на C# с циклом (упрощённо)





Рисунок 16. Граф потока управления для фрагмента кода на Листинге 6

В данном примере целевой является вершина 6. После её посещения в ходе обхода будет посещена вершина 1. Далее – тело цикла, то есть вершины 5, 4, 3, 2 и т.д. Вершины 5 и 4 формируют значение переменной *t*, а вершины 3 и 2 – переменной *s*. При первом обходе тела в TCFS будут включены лишь вершины 3 и 2, так как на момент первого обхода в VL будет лишь переменная *s*. Однако после посещения вершины 2 в VL добавится *t*, над которой производятся манипуляции в пропущенной нами части графа. Поэтому обход тела выполняется ещё раз, и вершины 4 и 5 добавятся в TCFS. Общее правило заключается в том, что если после обхода тела в VL появились новые переменные, то обход повторяется. Так происходит до тех пор, пока множество переменных в VL на одной итерации не будет совпадать с множеством на следующей.

TCFS, полученный на данном этапе, содержит всю необходимую информацию для построения регулярной аппроксимации.

#### 4.6. Построение конечного автомата по TCFS

Каждый путь из начальной вершины TCFS в конечную задаёт манипуляции, выполняемые программой для формирования одного из возможных значений строки со встроенным кодом. Чтобы получить все возможные значения, необходимо пройти по всем путям и выполнить описанные в них манипуляции. Под выполнением в данном случае понимается формирование соответствующих конечных автоматов для выражений в вершинах.

Построение автомата реализовано в виде интерпретатора, который обходит TCFS в порядке топологической сортировки со специальным учётом циклов. Интерпретатор выполняет выражения в вершинах, рассматривая их как манипуляции над конечными автоматами. Интерпретатор имеет стек, в который кладутся результаты операций, а также словарь переменных, в котором указывается, какой автомат соответствует каждой переменной, используемой в программе. После обработки очередной вершины текущее значение словаря переменных ассоциируется с каждым из исходящих рёбер обработанной вершины. Перед началом обработки очередной вершины все словари, ассоциированные с входящими рёбрами, объединяются, то есть над автоматами, соответствующими одним и тем же переменным, выполняется операция объединения.

Рассмотрим работу интерпретатора на примере. Слева на Рисунке 17 изображён TCFS для ссылки на переменную `sql` в аргументе вызова метода `Db.Execute` на Листинге 6. Справа на рисунке показаны состояния интерпретатора после обработки соответствующей вершины TCFS слева.

```

1 | String sql = "SELECT * FROM MyTable";
2 | if(cond) sql = sql + "WHERE id > 0";
3 | Db.Execute(sql);

```

Листинг 6. Фрагмент кода на C# с условием

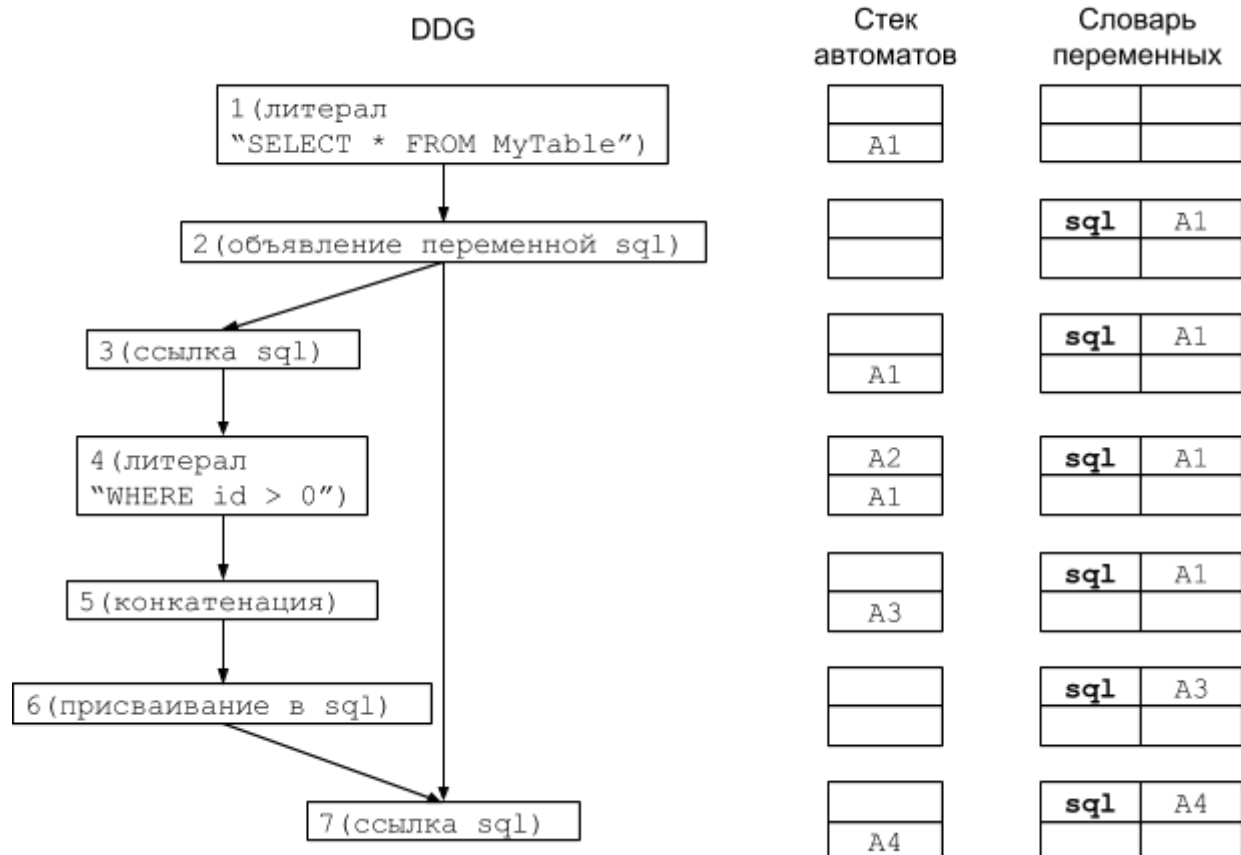


Рисунок 17. Пример построение конечного автомата по TCFS для фрагмента кода на Листинге 6.

Интерпретатор выполняет следующие шаги.

*Шаг 1.* Вершина 1 – литерал. У вершины нет входящих рёбер, а значит объединение словарей выполнять не нужно. Для литерала создаётся соответствующий конечный автомат A1 и кладётся в стек. Пустой словарь ассоциируется с единственным исходящим ребром.

*Шаг 2.* Вершина 2 – объявление переменной sql. С вершины стека извлекается автомат A1 и в пустой словарь переменных, взятый с

единственного входящего ребра, добавляется отображение из имени sql в этот автомат. Полученный словарь ассоциируется с двумя исходящими рёбрами.

*Шаг 3.* Вершина 3 – ссылка sql. Автомат, соответствующий имени sql, достаётся из словаря и загружается в стек.

*Шаг 4.* Вершина 4 – литерал. Создаётся автомат A2 и загружается в стек.

*Шаг 5.* Вершина 5 – конкатенация. Автоматы A1 и A2 извлекаются из стека, над ними выполняется конкатенация. Результирующий автомат A3 кладётся в стек.

*Шаг 6.* Вершина 6 – присваивание в sql. Автомат A3 снимается со стека и отображение в словаре для имени sql обновляется.

*Шаг 7.* Вершина 7 – ссылка на переменную sql. Вершина имеет 2 ребра, а значит нужно выполнить слияние словарей, ассоциированных с ними. В данном случае, это словарь с отображением из sql в A1 и словарь с отображением из sql в A3. Автоматы A1 и A3 объединяются в A4, создаётся новый словарь с отображением из sql в A4. A4 загружается в стек.

Автомат A4 является регулярной аппроксимацией множества значений выражения sql, а значит, результатом работы алгоритма.

В Таблице 2 показано, как меняется состояние интерпретатора при обработке всех типов вершин TCFS. Вершины типа StartNode, ExitNode и вершины-метки не указаны, так как они служат лишь для упрощения процесса обхода и не влияют на состояние интерпретатора.

Таблица 2. Изменение состояния интерпретатора при обработке вершин TCFS

Тип вершины	Изменение стека автоматов	Изменение словаря переменных
Declaration	Удаляется вершина	Добавляется отображение из объявленного имени в автомат с вершины стека
Updater, подтип Assing	Удаляется вершина	Обновляется отображение из имени, в которое выполняется присваивание, в автомат с вершины стека
Updater, подтип PlusAssing	Удаляются 2 вершины	Обновляется отображение из имени, в которое выполняется присваивание, в автомат, являющийся результатом конкатенации двух автоматов с вершины стека
Operation	Удаляется число автоматов, равное арифности операции, добавляется автомат, являющийся результатом применения операции к удалённым автоматам	Не изменяется
Literal	Добавляется новый автомат	Не изменяется
VarRef	Добавляется автомат, ассоциированный с именем ссылки в словаре переменных	Не изменяется
LoopNode	Не изменяется	Все отображения обновляются на объединённые и расширенные с

		соответствующими отображениями словаря предыдущей итерации
--	--	---

Обратим внимание на поведение интерпретатора при обработке циклов. Если в процессе обхода TCFS встречается вершина типа LoopNode, то текущий словарь переменных запоминается в ней. После очередного обхода тела цикла получается новый словарь. Для каждого автомата из этого словаря выполняется объединение и расширения с соответствующим автоматом из словаря, хранящегося в вершине LoopNode, как было описано ранее. Полученный в итоге словарь снова запоминается в вершине LoopNode. Повторные обходы тела цикла заканчиваются, когда словари соседних итераций содержат эквивалентные автоматы для соответствующих переменных.

Стоит отметить, что оператор расширения, необходимый для обработки циклов, отсутствовал в использованной библиотеке для работы с конечными автоматами, поэтому был реализован согласно определению в статье [7] и добавлен в библиотеку. Также была добавлена возможность проверки эквивалентности конечных автоматов.

## 4.7. Межпроцедурная аппроксимация

### 4.7.1. Основные сведения о реализации

Как было отмечено ранее, инструмент строит межпроцедурную аппроксимацию, то есть обрабатывает не только основной метод (или функцию), но и все, которые из него вызываются.

Вызовам произвольных методов в CFG соответствует вершина типа Arbitrary, которая хранит входные данные для построения CFG вызываемого метода. Если на этапе построения автоматов по TCFS интерпретатор

встречает вершину типа Arbitrary, он запускает весь алгоритм построения аппроксимации рекурсивно: по информации в вершине строится CFG, потом TCFS для выражений, возвращаемых методом, а по TCFS генерируется конечный автомат, который представляет множество строк, которые метод может вернуть. Таким образом, алгоритм построения аппроксимации является рекурсивным. Глубина рекурсии ограничена некоторым числом, которое задаётся как управляющий параметр алгоритма. Если будет достигнута максимально возможная глубина, рекурсивные запуски больше не будут выполняться, а в качестве результата посещения вершин типа Arbitrary будут использоваться автоматы, принимающие любые строки. Будет считаться, что вызов метода возвращает всё, что угодно. То же самое происходит, если по какой-либо причине в вершине типа Arbitrary недостаточно данных для построения CFG вызываемого метода.

Заметим, что вызываемые методы могут принимать аргументы. К моменту, когда интерпретатор встречает вершину типа Arbitrary, автоматы, соответствующие этим аргументам, уже известны и лежат на стеке интерпретатора. Перед рекурсивным запуском аппроксимации из автоматов с вершины стека формируется начальный словарь переменных, то есть автоматы связываются с именами параметров метода, который передаётся на вход алгоритму аппроксимации и используется на этапе построения автоматов по TCFS.

#### 4.7.2. Обработка рекурсивных методов и функций

В процессе построения межпроцедурной аппроксимации возникает проблема при обработке вызовов рекурсивных методов, так как в общем случае на этапе статического анализа нельзя определить, когда будет достигнута база рекурсии. Это значит, что алгоритм построения аппроксимации гарантированно достигнет максимальной глубины рекурсии,

заданной управляющим параметром, и аппроксимация множества возвращаемых значений метода будет неточной.

Чтобы бороться с хвостовой рекурсией, используется предобработка TCFS. Перед тем, как строить автоматы по TCFS, он проверяется на наличие рекурсивных вызовов в конце, то есть имеется ли среди выходных вершин графа вершины типа Arbitrary, хранящие информацию для метода с таким же именем, как и имя метода, для которого построен TCFS. Если это так, то структура TCFS перестраивается в эквивалентную в виде цикла. Таким образом реализуется подобие алгоритма оптимизации хвостовой рекурсии.

Для нехвостовых рекурсивных вызовов было предложено использовать технику Continuation-Passing Style (CPS), которая позволяет любую рекурсию представить в виде хвостовой, то есть позволяет свести задачу к предыдущему случаю. К сожалению, данный подход не был реализован и остался лишь в теории, поэтому нехвостовая рекурсия обрабатывается инструментом неточно.

#### 4.7.3. Особенности реализации межпроцедурной аппроксимации для JavaScript

Необходимо отметить, что межпроцедурная аппроксимация фактически не строится для языка JavaScript. В процессе работы с ReSharper SDK возникли проблемы с извлечением из CFG информации для вершин типа Arbitrary, которая необходима для построения CFG вызываемых методов. Дело в том, что ReSharper SDK имеет достаточно бедную документацию, в частности, по работе с JavaScript. Её объёма оказалось недостаточно, чтобы понять, как извлечь нужные данные и возможно ли это в принципе. Поэтому вершины типа Arbitrary не хранят никакую информацию и обрабатываются в процессе межпроцедурной аппроксимации с использованием автоматов, которые принимают любые строки, как было описано ранее.



## 5. Заключение

В данной работе реализован инструмент для построения регулярной аппроксимации встроенных языков. В процессе решения поставленных задач были получены следующие результаты.

1. Реализован алгоритм построения аппроксимации строковых выражений с поддержкой основных строковых операций и конструкций языка, использующихся при манипуляциях со строками: циклы, условные операторы, вызовы функций (в том числе с хвостовой рекурсией), объявления переменных, присваивание, конкатенацию, присваивание с конкатенацией, замену в строках и строковые литералы.

2. Решение сделано независимым от основного языка. Оно имеет чёткое разделение на две части: первая реализует алгоритм построения регулярной аппроксимации без привязки к основному языку, а вторая преобразует специфичные для конкретного языка исходные данные в обобщённые для передачи на обработку первой части. Это позволяет полученному инструменту достаточно просто поддерживать новые основные языки, что невозможно реализовать с помощью существовавших ранее инструментов.

3. Результат работы успешно интегрирован в платформу YaccConstructor, в рамках которой разрабатываются другие части инструмента статического анализа встроенных языков. Этот инструмент нацелен на решение задач, с которыми слабо справляются существующие анализаторы.

4. Реализовано построение регулярной аппроксимации для языков C# и JavaScript, что подтвердило независимость алгоритма аппроксимации от основного языка. Специфичная для основного языка часть инструмента была реализована для двух указанных языков и успешно протестирована на наборе

тестовых файлов с исходным кодом, формирующим код на встроенном языке.

Дальнейшая работа может быть связана с добавлением поддержки других строковых операций, к примеру, извлечения подстроки, форматных строк, реализации поддержки нехвостовой рекурсии, для которой в данной работе было предложено лишь теоретическое решение. Ещё одно направление – это доработка и улучшение обобщённого графа потока управления с учётом особенностей других основных языков, в частности, широко использующих функциональную парадигму.

В целом, стоит отметить, что успешная реализация данного проекта является новым шагом в направлении получения наиболее универсального инструмента статического анализа встроенных языков, наличие которого позволит решать задачи анализа не только в рамках поддержки подсветки синтаксиса встроенных языков, автодополнений и подобных базовых функций сред разработки, но и в рамках любых других задач, связанных со встраиванием кода, в которых статический анализ может быть полезен.

## **6. Библиографический список**

- [1] T.J. McCabe. A Complexity Measure. IEEE Transactions on Software Engineering, vol.2, no. 4, pp. 308-320, December 1976
- [2] Annamaa A., Breslav A., Kabanov J. e.a. An Interactive Tool for Analyzing Embedded SQL Queries. Programming Languages and Systems. LNCS, vol. 6461. Springer: Berlin; Heidelberg, p. 131–138, 2010.
- [3] H. Nguyen, C. Kästner, and T. Nguyen. Varis: IDE Support for Embedded Client Code in PHP Web Applications. In Proceedings of the 37th International Conference on Software Engineering (ICSE), May 2015
- [4] Aske Simon Christensen, Møller A., Michael I. Schwartzbach. Precise analysis of string expressions, Proc. 10th International Static Analysis Symposium (SAS), Vol. 2694 of LNCS. Springer-Verlag: Berlin; Heidelberg, June, p. 1 – 18, 2003.
- [5] Yasuhiko Minamide. Static approximation of dynamically generated web pages. In Proceedings of the 14th International Conference on World Wide Web, WWW '05, p. 432–441, New York, NY, USA, 2005. ACM.
- [6] Кириленко Я.А., Григорьев С.В., Авдюхин Д.А. Разработка синтаксических анализаторов в проектах по автоматизированному реинжинирингу информационных систем. ИТВ СПбГПУ. Информатика. Телекоммуникации. Управление. Выпуск 3(174)/2013.
- [7] Fang Yu, Muath Alkhalaf, Tefvik Bultan, and Oscar H. Ibarra. Automata-based symbolic string analysis for vulnerability detection. Formal Methods in System Design 44(1): 44-70 (2014)
- [8] Frances E. Allen. Control Flow Analysis. In Proceedings of a Symposium on Compiler Optimization (1970), pp. 1-19
- [9] Kahn, Arthur B. Topological sorting of large networks, Communications of the ACM 5 (11): 558–562 (1962)