САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Математико-механический факультет

Кафедра системного программирования

Байцерова Юлия Сергеевна

Анализ тестового покрытия компиляторов

Дипломная работа

Допущена к защите.

Зав. кафедрой:

д.ф.-м.н., профессор **Терехов А.Н.**

Научный руководитель:

ст. преп. **Вояковская Н.Н.**

Рецензент:

ст. преп. **Луцив Д.В.**

Санкт-Петербург

2014

SAINT-PETERSBURG STATE UNIVERSITY

Mathematics&Mechanics Faculty

Software Engineering Chair

Baytserova Yulia

Test coverage analysis of compilers

Graduation Thesis

Admitted for defence.

Head of the chair:

Professor **A. N. Terekhov**

Scientific supervisor:

Assistant professor **N. N. Voyakovskaya**

Reviewer:

Assistant professor **D. V. Luciv**

Saint-Petersburg

2014

Оглавление

Введение

Языки высокого уровня давно стали основным средством разработки программного обеспечения. Это явилось причиной широкого распространения программ, обеспечивающих процесс такой разработки и, в частности, компиляторов.

Компиляторы решают задачу перевода языка высокого уровня в представление, которое может быть выполнено ЭВМ. Ошибки в компиляторах приводят к тому, что программы на исходном языке транслируются в исполняемые модули, поведение которых отличается от поведения, определяемого семантикой исходной программы. Такого рода ошибки очень сложно выявлять и исправлять, их наличие ставит под сомнение качество компонент, для генерации которых использовался компилятор. Можно с уверенностью утверждать, что корректность работы компилятора языка программирования является необходимым условием надежной работы любого программного обеспечения, реализованного с его помощью, а контроль правильности работы компиляторов - одной из важнейших мер по обеспечению надежности программного обеспечения.

Для компилятора, как и для любого другого вида программного обеспечения, одним из важнейших методов обеспечения надежности является тестирование. Процесс тестирования включает в себя три основные задачи:

- Генерация (написание) тестов.
- Вынесение вердикта о прохождении теста (тестовый оракул).
- Оценка качества тестов (критерий покрытия).

Для решения этих задач применяются различные методы, которые можно разделить на две группы — методы «черного» и «белого» ящика. В первом случае, единственным источником информации для создания тестов является описание функциональности тестируемого продукта, также называемое спецификацией. Во втором, для создания тестов также используется информация о реализации, исходный код тестируемого продукта.

В случае компиляторов, метод черного ящика состоит в получении тестовых наборов по описанию синтаксиса и семантики языка. Такие тестовые наборы предназначены для проверки того, что компилятор работает в соответствии с синтаксисом и семантикой поддерживаемого им языка. Метод белого ящика позволяет учитывать особенности конкретной реализации компилятора. Созданные этим методом тесты

позволяют более тщательно проверять части компилятора, специфичные для тестируемой реализации компилятора.

Для тестирования компиляторов важны как методы «черного ящика», так и методы «белого ящика». Эти методы взаимно дополняют друг друга и могут использоваться совместно. Дипломная работа посвящена тестированию компиляторов на основе исходного кода. [1]

1. Постановка задачи

Целью данной дипломной работы является разработка инструмента для анализа тестового покрытия компиляторов и автоматической генерации тестов на основе этого анализа. Для достижения этой цели были поставлены следующие задачи:

- Изучение существующих метрик и методов анализа тестового покрытия, их сравнение
- Разработка инструмента для построения графа потока управления с заданной глубиной раскрытия функций
- Разработка генератора тестов и анализатора тестового покрытия

2. Используемые термины

Граф потока управления (Control Flow Graph, CFG) — множество всех возможных путей исполнения программы, представленное в виде ориентированного графа. В графе потока управления каждый узел графа соответствует базовому блоку — прямолинейному участку кода, не содержащего в себе ни операций передачи управления, ни точек, на которое управление передается из других частей программы. В графе потока управления есть 2 выделенных узла: start и stop, обозначающих входной и выходной узлы графа.

Узел M доминирует над узлом N, если любой путь от входного узла к узлу N проходит через узел M. Входной узел доминирует над всеми остальными узлами графа.

Узел M постдоминирует над узлом N, если любой путь от узла N к выходному узлу проходит через узел M. Выходной узел постдоминирует над всеми остальными узлами графа.

Каждый узел N имеет единственный непосредственный доминатор M, который представляет собой последний доминатор N на любом пути от входного узла к N.

Каждый узел N имеет единственный непосредственный постдоминатор M, который представляет собой первый постдоминатор N на любом пути от узла N к выходному узлу.

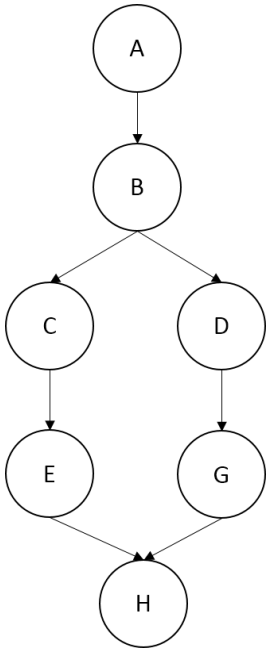
Дерево доминаторов — вспомогательная структура данных, содержащая информацию об отношениях доминирования. Дуга от узла M к узлу N идет тогда и только тогда, если M является непосредственным доминатором N.

Дерево постдоминаторов — вспомогательная структура данных, содержащая информацию об отношениях постдоминирования. Дуга от узла M к узлу N идет тогда и только тогда, если M является непосредственным постдоминатором N.

Ориентированный граф называется сильно связным, если любые две его вершины сильно связаны. Две вершины s и t любого графа сильно связаны, если существует ориентированный путь из s в t и ориентированный путь из t в s. Компонентами сильной связности ориентированного графа называются его максимальные по включению сильно связные подграфы.

DOT — язык описания графов. Граф, описанный на языке DOT, обычно представляет собой текстовый файл с расширением .gv или .dot в понятном для человека и обрабатывающей программы формате. Пример графа и DOT-файла для него:

```
digraph "" {
  __N1 [label="A"];
  __N2 [label="B"];
  __N3 [label="C"];
  __N4 [label="D"];
  __N5 [label="E"];
  __N6 [label="G"];
  __N7 [label="H"];
  __N1 -> __N2;
  __N2 -> __N3 [color="green", label="yes"];
  __N2 -> __N4 [color="red", label="no"];
  __N3 -> __N5;
  __N4 -> __N6;
  __N5 -> __N7;
  __N6 -> __N7;
}
```



Матрица смежности графа — квадратная матрица размерности NxN, где N — число узлов графа, элементы которой задаются следующей формулой:

$$A[i, j] = \begin{cases} 1, & \text{существует ребро, соединяющее } i \text{ и } j \text{ вершины} \\ 0, & \text{иначе} \end{cases}$$

Матрица достижимости графа — квадратная матрица размерности NxN, где N — число узлов графа, элементы которой задаются следующей формулой:

$$T[i, j] = \begin{cases} 1, & \text{вершина } j \text{ достижима из вершины } i \\ 0, & \text{иначе} \end{cases}$$

Матрица сильной связности графа — квадратная матрица размерности NxN, где N — число узлов графа, элементы которой задаются следующей формулой:

$$S[i, j] = \begin{cases} 1, & \text{вершина } i \text{ достижима из вершины } j \text{ и} \\ & \text{вершина } j \text{ достижима из вершины } i \\ 0, & \text{иначе} \end{cases}$$

Матрица доминаторов — матрица, отражающая отношение доминирования для узлов. Элементы матрицы доминаторов задаются следующей формулой:

$$DM[i, j] = \begin{cases} 1, & \text{вершина } j \text{ доминирует вершину } i \\ 0, & \text{иначе} \end{cases}$$

Функция sign от матрицы вычисляется как функция sign от всех ее элементов. Функция sign от числа вычисляется по следующей формуле:

$$\text{sign}(R) = \begin{cases} 1, & R > 0 \\ 0, & \text{иначе} \end{cases}$$

3. Обзор

3.1. Контекст работы

3.1.1. RulesLanguage

RulesLanguage – язык разработки бизнес-приложений.

Пример кода, написанного на RulesLanguage.

```
dcl
        i integer;
        c char(10);
        w(1) view contains c;
        v like w;
enddcl

proc w: like w
        proc return(v)
endproc

proc g
        if w(v) = w
        return
endif

w
endproc
```

3.1.2. Codegen

В качестве инструментального компилятора в рамках данной дипломной работы использовался Codegen — продукт, разработанный сотрудниками ЗАО «Ланит-Терком» и представляющий собой многоплатформенный компилятор языка RulesLanguage в языки C, Java, Cobol, C# и некоторые другие.

3.2. Критерии и метрики тестового покрытия

Набор тестов, используемый при тестировании, всегда конечен и, более того, ограничен соображениями экономической эффективности распределения ресурсов между разными видами деятельности при разработке ПО. Поэтому крайне важно строить его так,

чтобы используемые тесты проверяли как можно больше разных аспектов функциональности системы в как можно большем разнообразии ситуаций. Чтобы систематическим образом перебирать существенно отличающиеся друг от друга ситуации, используют критерии полноты тестирования или критерии адекватности тестирования. Тестовый набор, удовлетворяющий заданному критерию полноты, называют полным по этому критерию.

Чаще всего для определения критерия полноты некоторые из возможных тестовых ситуаций рассматривают как эквивалентные и определяют количество классов неэквивалентных тестовых ситуаций, встретившихся или «покрытых» во время тестирования. Такие критерии полноты называются критериями тестового покрытия. При этом определяется и числовая метрика тестового покрытия — доля покрытых классов ситуаций среди всех возможных. Критерий полноты может использовать различные значения метрики, например, он может требовать, чтобы полный тестовый набор всегда покрывал 100% выделенных классов ситуаций, или же считать достаточным покрытие 85% классов ситуаций.

Полноту тестирования можно определять по-разному, но в основе любого критерия полноты лежит представление о возможных ошибках в тестируемой системе. Различные способы классификации ситуаций, отражающие их разнообразие с точки зрения тестирования, перечислены ниже. Каждый из них и любое их подмножество совместно могут использоваться для определения метрик тестового покрытия. Классифицировать ситуации можно следующим образом.

- На основе структурных элементов тестируемой системы, которые выполняются или задействуются в ходе тестирования.
- На основе структуры входных данных, используемых во время тестирования.
- На основе элементов требований, проверяемых при выполнении тестов.
- На основе явно сформулированных предположений об ошибках, выявление которых должны обеспечить тесты.
- На основе произвольных моделей устройства или функционирования тестируемой системы.

В основе структурных критериев полноты лежит простая идея: если ошибка находится в какой-то конструкции кода, в каком-то компоненте тестируемой системы, то выполнив эту конструкцию или заставив работать этот компонент, мы, скорее всего, сможем ее обнаружить. Соответственно, если в двух ситуациях выполняются одни и те же

элементы кода, такая ошибка будет либо проявляться в обеих ситуациях, либо не проявляться ни в одной, поэтому их можно объявить эквивалентными и проверять всегда только одну из таких ситуаций.

Структурные метрики покрытия различаются в зависимости от размера элементов системы, используемых при их определении. Можно выделить три уровня структурных метрик — уровень отдельной функции или отдельного метода класса, уровень компонента или класса, включающего несколько операций, и уровень подсистемы или системы в целом, в составе которых может быть много компонентов.

Вне зависимости от уровня структурные метрики могут быть основаны на информации двух видов — на информации о передаче управления между разными исполняемыми элементами системы или на информации об использовании и записи данных. Метрики первого типа называются основанными на потоке управления, второго типа — основанными на потоках данных.

Важным достоинством структурных метрик покрытия является возможность их автоматизированного вычисления при наличии доступа к коду или схемам архитектуры тестируемой системы. Существенным недостатком является отсутствие учета требований — мы можем покрыть все элементы структуры, но не обнаружим, что какое-то требование просто забыли реализовать. Но в рамках моей работы этот факт не является недостатком, т.к. ее цель – оценка тестового покрытия по набору тестов.

Структурные метрики покрытия для одной функции или метода на основе потока управления базируются на исполняемых в ходе теста элементах кода этой функции или этого метода.

Метрики покрытия на основе потока управления бывают следующими:

- Метрика покрытия инструкций (statement coverage)
- Метрика покрытия ветвей (branch coverage или decision coverage)
- Метрика покрытия комбинаций условий (multiple condition coverage)
- Метрика покрытия условий и ветвей (condition/decision coverage или condition/branch coverage).
- Метрика модифицированного покрытия условий и ветвей (modified condition/decision coverage, MC/DC).

В своей работе я использую несколько измененную метрику покрытия путей, а именно: вычисление тестового покрытия производится мной по дереву доминаторов, построенному по графу потока управления.

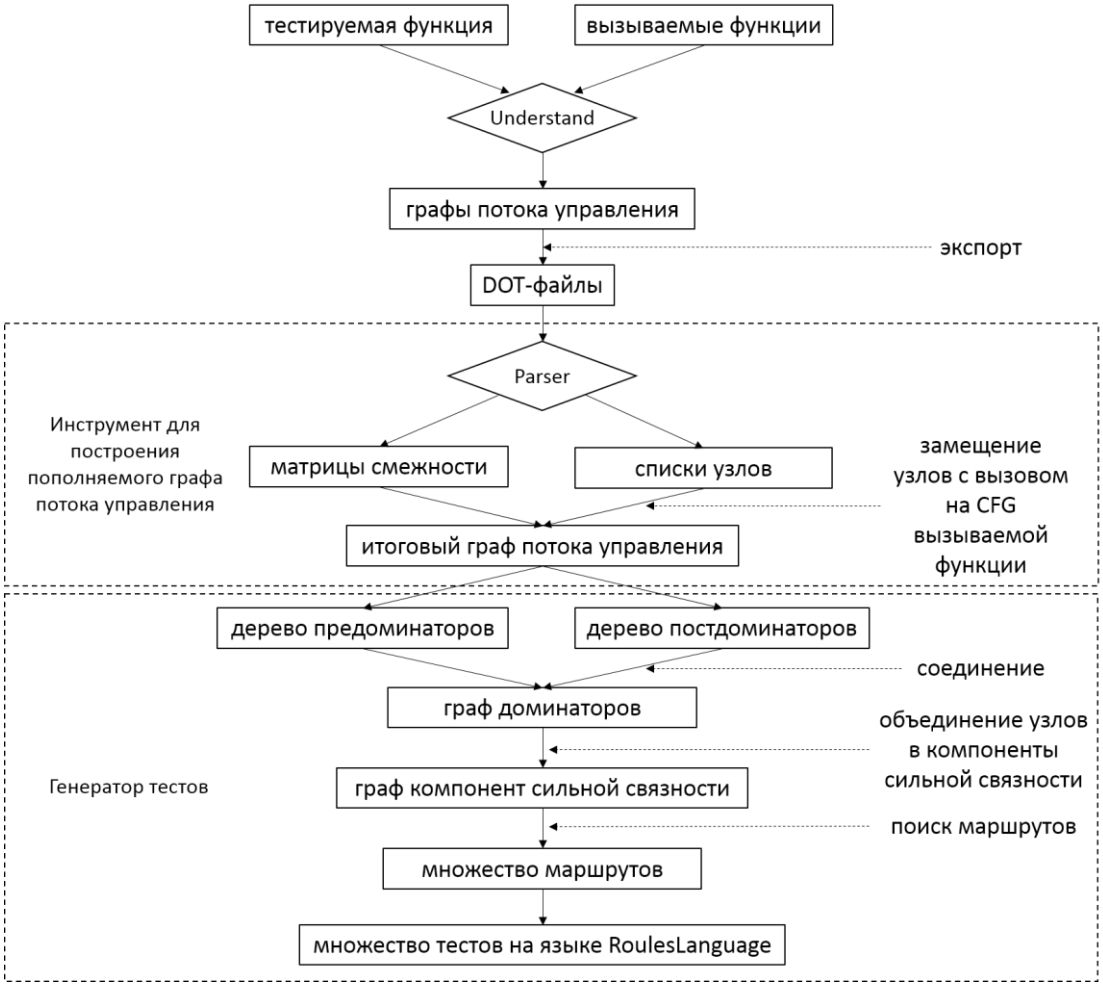
Подробнее о метриках можно прочитать в работе [2].

3.3. Используемые технологии

В своей работе я использовала инструмент под названием Understand. Это инструмент статического анализа кода проектов на Ada, Cobol, C#, Fortran, Java, Delphi/Pascal, VHDL, Python, C/C++, Web Languages. Позволяет анализировать код на соответствие стандартам, выводить отчеты об используемых функциях, его метрические данные, строить разнообразные диаграммы, в т.ч. Control Flow Graph, UML Class Diagramm, Hierarchy Graph, Dependency Graph. Информация может быть представлена в форме графиков и экспортирована в различные форматы. Understand содержит удобную систему навигации по коду. Встроенный скриптовый язык позволяет писать расширения для специализированного анализа, используя возможности инструмента.

4. Предлагаемое решение

Схематично предлагаемое решение можно изобразить следующим образом:



Предложенный алгоритм можно описать следующим образом:

1. Построение пополняемого графа потока управления тестируемой функции
2. Построение деревьев доминаторов и постдоминаторов по пополняемому графу потока управления
3. Построение графа доминаторов
4. Построение графа компонент сильной связности путем выделения компонент сильной связности в графе доминаторов
5. Генерация множества путей в графе компонент сильной связности от корня к листьям с целью дальнейшего использования их в качестве тестовых сценариев

6. Генерация тестов на языке RilesLanguage
7. Сравнение с заданными тестами, если таковые имеются, с помощью инструмента, разработанного в рамках курсовой работы [3], и выделение набора уникальных сгенерированных тестов
8. Получение числового представления покрытия при наличии заданного набора тестов

В основе алгоритма лежит построение графа потока управления, что позволяет учитывать все возможные варианты работы программы и обеспечить конечность процесса генерации тестовых сценариев.

Идея использования деревьев доминаторов и постдоминаторов заключается в следующем: тестовый сценарий, тестирующий какой-то узел в графе потока управления, автоматически тестирует все доминаторы этого узла. Эта идея позволяет значительно сократить набор тестов, требуемых для достижения наилучшего покрытия. Объединение в компоненты сильной связности позволяет еще больше сократить этот набор. Подробнее можно прочитать в работе [4].

5. Инструмент для построения пополняемого графа потока управления

5.1. Взаимодействие с Understand

Пополняемым графом потока управления будем называть граф потока управления, рекурсивно полученный из графа потока тестируемой функции или процедуры путем последовательного замещения вызываемых функций и процедур на их граф потока управления. Для его построения необходимо получить граф потока управления для тестируемой функции или процедуры, а также для всех функций и процедур, которые могут быть задействованы в анализе тестируемой. В данной работе для достижения этой цели использовался инструмент Understand. Полученные с помощью него графы потока управления экспортируются в .dot-файлы, которые впоследствии должны быть обработаны синтаксическим анализатором для получения матриц смежности. Матрицы смежности в дальнейшем используются для построения пополняемого графа потока управления для тестируемой функции или процедуры.

5.2. Алгоритм построения пополняемого графа потока управления

Построение пополняемого графа потока управления можно разделить на следующие этапы:

1. Сгенерировать графы потока управления для всех функций и процедур

2. В узлах графа потока управления тестируемой функции найти и заместить вызовы функций и процедур на их графы потока управления

3. Повторять шаги 1 и 2, пока не будет достигнут желаемый уровень раскрытия функций

В ходе работы было разработано следующее:

1. Синтаксический анализатор для обработки .dot-файлов и построения матриц смежности

2. Инструмент построения пополняемого графа потока управления с заданным уровнем раскрытия функций

Инструмент построения пополняемого графа потока управления оперирует деревьями, полученными из матриц смежности. В каждом узле графа инструмент ищет

вызов функции, затем ищет матрицу смежности для графа вызываемой функции, строит по ней дерево, после чего замещает узел с вызовом функции на это дерево. Пример работы алгоритма представлен на рисунке 1.

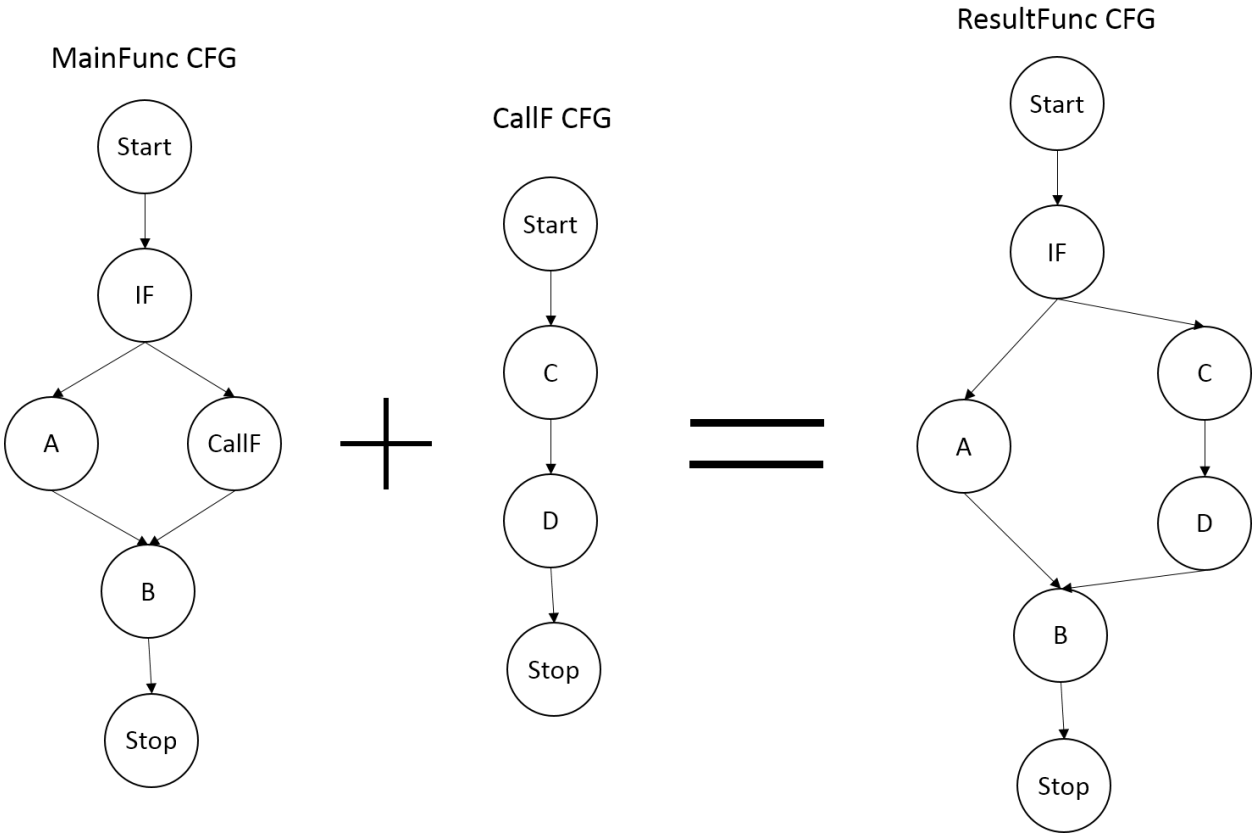


Рис. 1. Пример замещения узла с вызовом функции на ее граф потока управления

6. Генератор тестов

6.1. Пользовательский интерфейс

Генератор тестов представляет собой WinForms приложение, генерирующее набор тестов по заранее заданной тестируемой функции. Пользователь вводит в поле “Functions disclosure level” значение, обозначающее уровень раскрытия функций, нажимает на кнопку “Generate tests!” и получает список уникальных тестов. Тесты можно посмотреть с помощью встроенного в приложение визуализатора. Внешний вид генератора показан на рисунке 4.

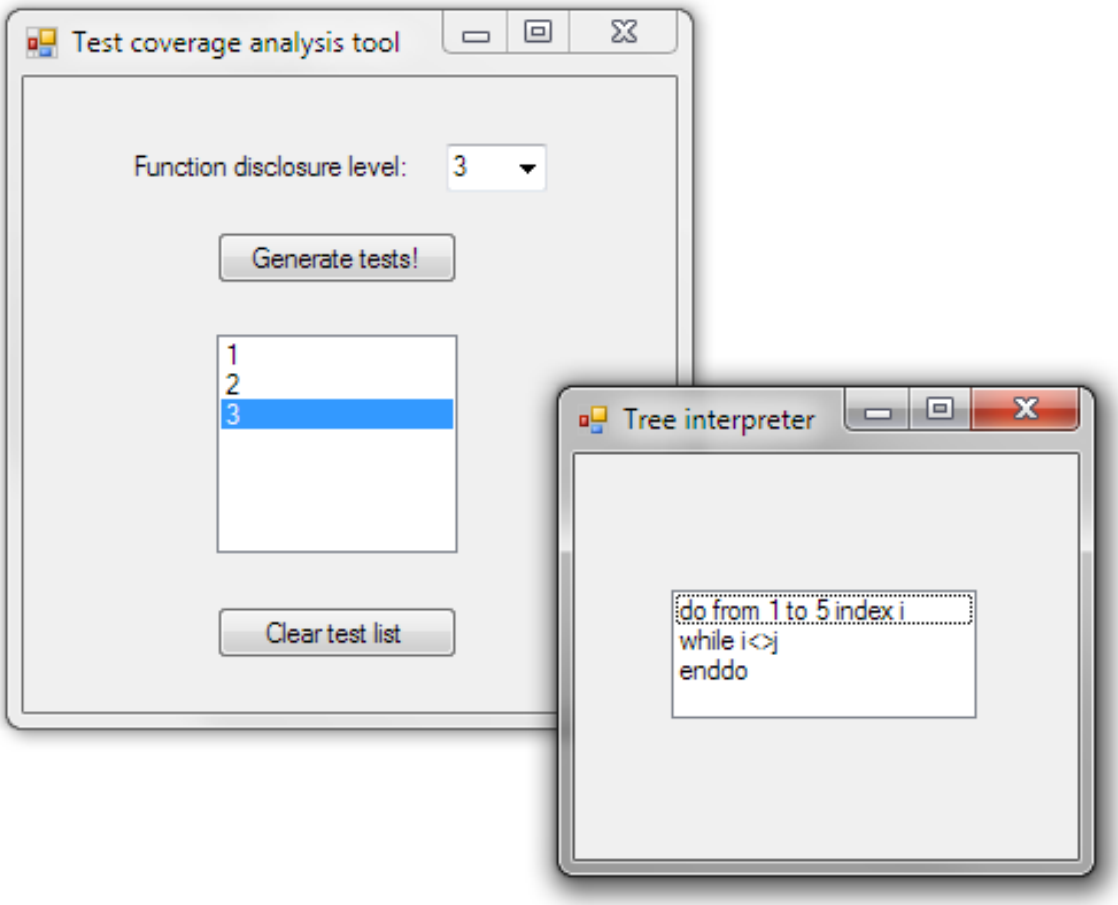


Рис. 4. Пользовательский интерфейс разработанного генератора тестов

6.2. Описание алгоритма генерации тестов

Алгоритм генерации тестов состоит из 4 шагов:

1) по пополняемому графу потока управления строятся дерево доминаторов и дерево постдоминаторов

2) путем слияния этих деревьев получается граф доминаторов

3) из графа доминаторов путем выделения компонент сильной связности получается граф компонент сильной связности

4) по графу компонент сильной связности генерируются тестовые сценарии

5) полученные тестовые сценарии с помощью инструмента, разработанного в качестве курсовой работы [3], сравниваются с имеющимися тестами и выделяется набор уникальных сгенерированных тестовых сценариев

6) по этому набору генерируются тесты на языке RulesLanguage

7) число, обозначающее тестовое покрытие, вычисляется по следующей формуле:

$$\frac{n}{m-n}$$

где n – количество уникальных сгенерированных тестов, m – общее количество сгенерированных тестов

6.3. Алгоритм построения деревьев доминаторов и постдоминаторов

Алгоритм построения дерева доминаторов можно представить следующим образом:

1. Строим матрицу доминаторов (DM). Изначально это матрица, в которой во всех строках стоят 1, кроме первой строки. В ней 1 только в первом столбце.

2. Определяем 2 операции над строками матрицы доминаторов:

- Объединение

  $i \cup j = \begin{bmatrix} i, & i = j \\ 1, & \text{иначе} \end{bmatrix}$, где i и j – строки матрицы доминаторов

- Пересечение

  $i \cap j = i * j$, где i и j – строки матрицы доминаторов

Пример.

1	0	1	0	1	0	1	∪
1	1	1	0	0	0	1	=
1	1	1	0	1	0	1	

1	0	1	0	1	0	1	∩
1	1	1	0	0	0	1	=
1	0	1	0	0	0	1	

3. Для каждого узла x, кроме root, определяем множество доминаторов следующим образом:

$$dom(x)=\{x\}\cup\{dom(i_1)\cap dom(i_2)\cap\ldots\cap dom(i_k)\},$$

где $i_1, i_2,\ldots,i_k$ – предки узла x.

4. Повторяем шаг 3, пока матрица доминаторов не перестанет изменяться после прохода всех узлов.

Рассмотрим алгоритм построения дерева доминаторов на примере. Пусть дан граф, показанный на рисунке 2.

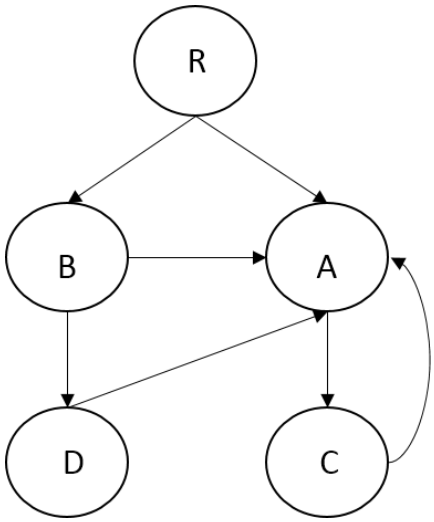


Рис. 2. Исходный граф

Матрица смежности для этого графа будет выглядеть следующим образом:

	R	A	B	C	D
R	0	1	1	0	0
A	0	0	0	1	0
B	0	1	0	0	1
C	0	1	0	0	0
D	0	1	0	0	0

Шаг 1. Строим матрицу доминаторов. В нашем примере она будет иметь следующий вид:

	R	A	B	C	D
R	1	0	0	0	0
A	1	1	1	1	1
B	1	1	1	1	1
C	1	1	1	1	1
D	1	1	1	1	1

Шаг 2. Распишем последовательно пункт 3 для нашего примера:

1. dom(A)={A}∪ {dom(R)∩dom(B)∩dom(C)∩dom(D)}

Подставив строки из матрицы доминирования в эту формулу, получаем:

$$\begin{aligned} dom(A) = {} & 0\ 1\ 0\ 0\ 0 \cup \{ 1\ 0\ 0\ 0\ 0 \cap \\ & 1\ 1\ 1\ 1\ 1 \cap \\ & 1\ 1\ 1\ 1\ 1 \cap \\ & 1\ 1\ 1\ 1\ 1 \} \end{aligned}$$

что эквивалентно:

$$\begin{aligned} dom(A) &= 0\ 1\ 0\ 0\ 0 \cup \{ 1\ 0\ 0\ 0\ 0 \} \\ &= \mathbf{1\ 1\ 0\ 0\ 0} \end{aligned}$$

Аналогично для других узлов.

2. dom(B)={B}∪ {dom(R)}

$$\begin{aligned} dom(B) &= 0\ 0\ 1\ 0\ 0 \cup \{ 1\ 0\ 0\ 0\ 0 \} \\ &= \mathbf{1\ 0\ 1\ 0\ 0} \end{aligned}$$

3. dom(C)={C}∪ {dom(A)}

$$\begin{aligned} dom(C) &= 0\ 0\ 0\ 1\ 0 \cup \{ 1\ 1\ 0\ 0\ 0 \} \\ &= \mathbf{1\ 1\ 0\ 1\ 0} \end{aligned}$$

4. dom(D)={D}∪ {dom(B)}

$$\begin{aligned} dom(D) &= 0\ 0\ 0\ 0\ 1 \cup \{ 1\ 0\ 1\ 0\ 0 \} \\ &= \mathbf{1\ 0\ 1\ 0\ 1} \end{aligned}$$

После этих процедур матрица доминаторов из нашего примера будет иметь вид:

DM	R	A	B	C	D
R	1	0	0	0	0
A	1	1	0	0	0
B	1	0	1	0	0
C	1	1	0	1	0
D	1	0	1	0	1

В нашем примере при повторном проходе узлов матрица доминаторов не изменится, значит, алгоритм завершен.

Шаг 3. Используя матрицу доминаторов, строим дерево доминаторов путем связывания узла x с его непосредственным доминатором.

В результате получаем дерево доминаторов, показанное на рисунке 3.

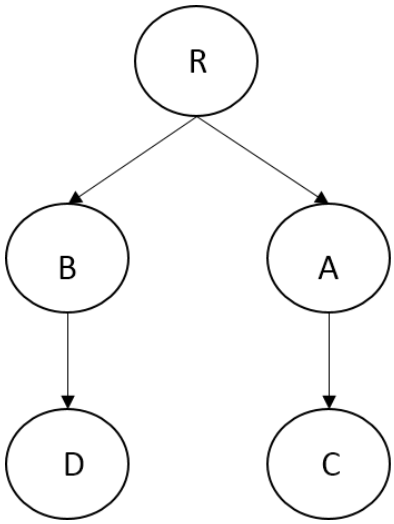


Рисунок 3. Результат работы алгоритма

Более подробное описание алгоритма можно прочитать в работе [5].

Алгоритм построения дерева постдоминаторов аналогичен построению дерева доминаторов, только в качестве исходного графа берется граф с инвертированными ребрами. Путем слияния дерева доминаторов и дерева постдоминаторов получается граф доминаторов.

6.4. Алгоритм построения графа доминаторов

Алгоритм построения графа доминаторов можно описать следующим образом:

1. Получаем матрицы смежности для дерева доминаторов и дерева постдоминаторов: A и B.

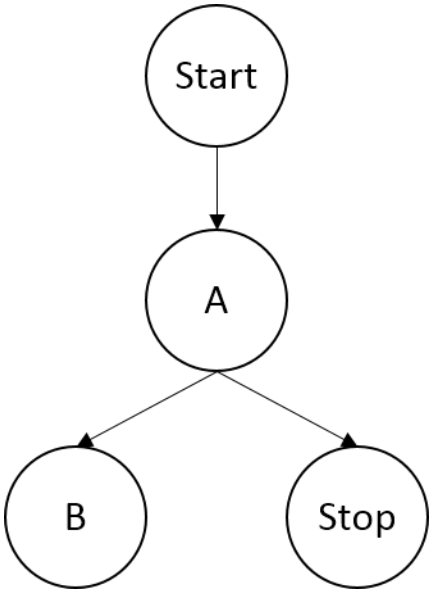
2. Элементы матрицы B симметрично отображаем относительно горизонтальной оси. Получаем матрицу B’.

3. Элементы матрицы B’ симметрично отображаем относительно вертикальной оси. Получаем матрицу B’’.

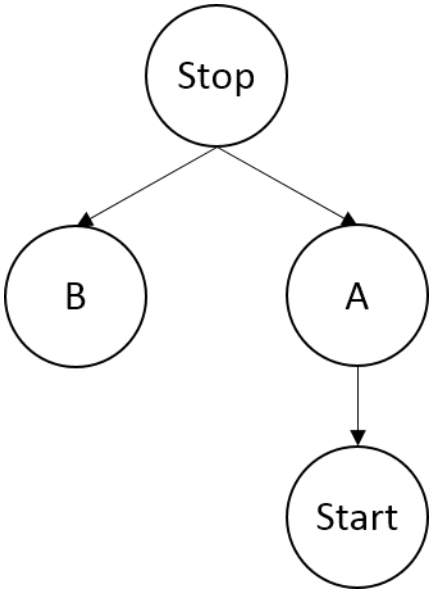
4. Матрицы A и B’’ поэлементно складываем. Получаем матрица C.

5. К матрице C применяем функция sign. Получаем граф доминаторов.

Рассмотрим работу алгоритма на примере. Пусть даны 2 дерева:



Дерево доминаторов



Дерево постдоминаторов

1. Получаем матрицы смежности для этих деревьев:

$$A = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \qquad B = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

2. Получаем матрицу B’:

$$B' = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{pmatrix}$$

3. Получаем матрицу B’’:

$$B'' = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{pmatrix}$$

4. Получаем матрицу C:

$$\begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{pmatrix}$$

5. Получаем граф доминаторов:

$$\text{sign} \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{pmatrix}$$

6.5. Алгоритм построения графа компонент сильной связности

Граф компонент связности получается из графа доминаторов путем нахождения компонент сильной связности. Алгоритм нахождения компонент сильной связности можно описать следующим образом:

1. Строим матрицу смежности графа A(D) (D — исходный ориентированный граф).
2. Находим матрицу достижимости T(D), используя алгоритм Уоршелла [6].
3. Строим матрицу сильной связности S(D), используя формулу из [6]:

$$S[i, j] = \begin{bmatrix} 1, & i = j \\ T[i,j] \wedge T[j,i], & \text{иначе} \end{bmatrix}$$

4. Присваиваем P = 1 (P — количество компонент связности), $S_1 = S(D)$.
5. Включаем в множество вершин V_p компоненты сильной связности D_p вершины, соответствующие единицам первой строки матрицы S_p. В качестве матрицы $A(D_p)$ возьмем подматрицу матрицы A(D), состоящую из элементов матрицы A, находящихся на пересечении строк и столбцов, соответствующих вершинам из V_p.
6. Вычеркиваем из S_p строки и столбцы, соответствующие вершинам из V_p. Если не остается ни одной строки (и столбца), то P — количество компонент сильной связности. В противном случае обозначим оставшуюся после вычеркивания срок и столбцов матрицу как S_{p+1}, присваиваем P = P + 1 и переходим к п. 5.

Разберем алгоритм на примере.

Выделим компоненты связности ориентированного графа, изображенного на рисунке 5. В данной задаче количество вершин N = 5.

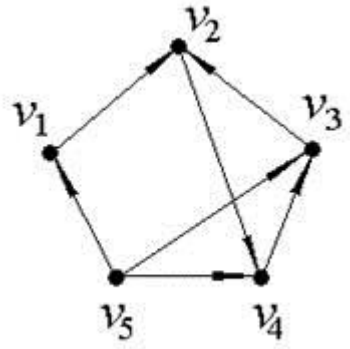


Рис. 5. Исходный граф

Значит, для данного ориентированного графа матрица смежности будет иметь размерность 5×5 и будет выглядеть следующим образом

$$A(D) = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \end{pmatrix}$$

Найдем матрицу достижимости для данного ориентированного графа, используя алгоритм Уоршелла из [6]:

$$T(D) = \begin{pmatrix} 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

Таким образом, матрица сильной связности, полученная по формуле:

$$\mathrm{S}\,[\mathrm{i},\,\mathrm{j}] = \begin{cases} 1, & i = j \\ T[i,j] \wedge T[j,i], & \text{иначе} \end{cases}$$

будет следующей:

$$S(D) = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}.$$

Присваиваем P = 1, S_1 = S(D) и составляем множество вершин первой компоненты сильной связности D1: это те вершины, которым соответствуют единицы в первой строке матрицы S(D). Таким образом, первая компонента сильной связности состоит из одной вершины $V_1 = \{v_1\}$.

Вычеркиваем из матрицы $S_1(D)$ строку и столбец, соответствующие вершине $V1$, чтобы получить матрицу $S_2(D)$:

$$S_2(D) = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Присваиваем P = 2. Множество вершин второй компоненты связности составят те вершины, которым соответствуют единицы в первой строке матрицы $S_2(D)$, то есть $V_2 = \{v_2, v_3, v_4\}$. Составляем матрицу смежности для компоненты сильной связности D_2 исходного графа D − в ее качестве возьмем подматрицу матрицы A(D), состоящую из элементов матрицы A, находящихся на пересечении строк и столбцов, соответствующих вершинам из V_2:

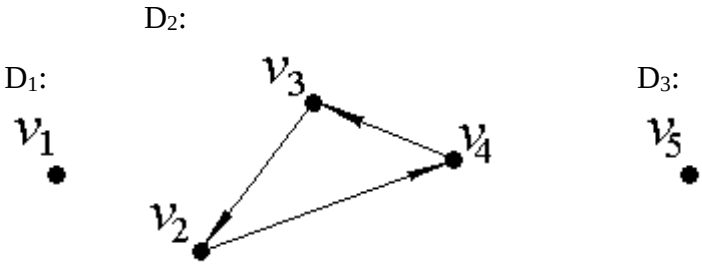
$$A(D_2) = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

Вычеркиваем из матрицы $S_2(D)$ строки и столбцы, соответствующие вершинам из V_2, чтобы получить матрицу $S_3(D)$, которая состоит из одного элемента:

$$S_3(D) = (1)$$

И присваиваем P = 3. Таким образом, третья компонента сильной связности исходного графа, как и первая, состоит из одной вершины $V_3 = \{v_5\}$.

Таким образом, выделены 3 компоненты сильной связности ориентированного графа D:



Данный алгоритм можно найти в работе [6].

6.6. Алгоритм генерации путей

Для генерации путей в графе компонент сильной связности используется алгоритм обхода графа в глубину. Алгоритм поиска описывается следующим образом: для каждой непройденной вершины необходимо найти все непройденные смежные вершины и повторить поиск для них. Подробное описание алгоритма можно найти в [7].

6.7. Алгоритм генерации тестов

Алгоритм генерации тестов на языке RulesLanguage можно описать следующим образом:

1. Получаем из сгенерированного пути список узлов, в которых встречается задание марки узла вида: MARK (<node>) = <mark_name>.

2. Заменяем узлы такого вида на соответствующие метки. Получаем общую структуру тестов.

3. Выделяем набор уникальных тестов.

4. В полученном наборе тестов заменяем метки узлов на конструкции на языке RulesLanguage.

6.8. Выделение набора уникальных тестов

Для выявления набора уникальных тестов используется инструмент, разработанный в рамках курсовой работы [3]. Этот инструмент использует в качестве представления тестов линейную скобочную запись парсерных деревьев, которые умеет строить Codegen, а в качестве алгоритма сравнения – подсчет схожести между деревьями по матрице различия. В рамках данной дипломной работы в качестве сравниваемых объектов выступают тесты, полученные на шаге 2 алгоритма генерации тестов, и линейная скобочная запись парсерных деревьев существующих тестов.

7. Сравнение алгоритмов

Реализованные инструменты были протестированы на функции синтаксического анализатора компилятора Codegen. В результате их работы было сгенерировано 528 тестов, в то время как при обходе в глубину исходного графа потока управления было сгенерировано 850 тестов.

Заключение

В ходе данной работы были достигнуты следующие результаты:

- Изучены и проанализированы метрики и методы анализа тестового покрытия.
- Разработан и реализован на языке C# инструмент построения пополняемого графа потока управления.
- Разработан и реализован на языке C# инструмент генерации тестов и анализатора тестового покрытия.

На сегодняшний день аналогов разработанного инструмента нет, есть отдельные инструменты для семантического и синтаксического анализа.

В качестве дальнейшего развития может выступать инструмента, позволяющего варьировать процент покрытия. Подробно об этом изложено в работе [4].

Список литературы

[1] М. А. Посыпкин. Применение формальных методов для тестирования компиляторов. — Москва: ИСП РАН, Препринт 2, 2004. — Стр. 1-14

[2] В. В. Кулямин. Тестирование на основе моделей. — Москва, ИСП РАН

[3] Ю. С. Байцерова. Проверка избыточности и минимизация множества тестов. — Санкт-Петербург: СПбГУ, http://se.math.spbu.ru/SE/YearlyProjects/2013/list

[4] Hiralal Agrawal. Dominators, Super Blocks, and Program Coverage. — New York, USA: POPL '94 Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 1994. — p. 25-34

[5] Aho, V., Sethi, R. & Ullman, J.D. (1986). Compilers principles techniques and tools. Addison-Wesley, Reading (MA).

[6] Род Хаггарти. Дискретная математика для программистов. — Москва: Техносфера, 2012. — Стр. 178, 282-289.

[7] Кормен Т., Лейзерсон Ч., Ривест Р. Глава 22. Элементарные алгоритмы для работы с графами // Алгоритмы: построение и анализ (второе издание). — М.: «Вильямс», 2005. — С. 622-632.