Санкт-Петербургский государственный университет

Зиновьева Анна Геннадьевна

Выпускная квалификационная работа

Генерация структурных текстовых программ в TRIK Studio

Уровень образования: бакалавриат

Направление *02.03.03 «Математическое обеспечение и администрирование информационных систем»*

Основная образовательная программа *СВ.5006.2017 «Математическое обеспечение и администрирование информационных систем»*

Профиль *Параллельное программирование*

Научный руководитель:
к.т.н., доцент кафедры системного программирования Ю.В. Литвинов

Консультант:
ст.преп. кафедры системного программирования Я.А. Кириленко

Рецензент:
разработчик ООО «Яндекс Беспилотные технологии» Г.А. Зимин

Санкт-Петербург
2021

Saint Petersburg State University

Anna Zinovyeva

Bachelor's Thesis

Generation of structural text prorgams in TRIK Studio

Education level: bachelor

Speciality *02.03.03 "Software and Administration of Information Systems"*

Programme *CB.5006.2017 "Software and Administration of Information Systems"*

Profile *Parallel Programming*

Scientific supervisor:
C.Sc., Docent Yurii Litvinov

Consultant:
Senior Lecturer Iakov Kirilenko

Reviewer:
Developer at "Yandex Self Driving Group" Grigorii Zimin

Saint Petersburg
2021

Оглавление

Введение

В современном мире робототехника всё чаще начинает изучаться в школах. Ученики младших классов ещё до первого знакомства с текстовыми языками осваивают основы программирования с помощью роботов. Одним из инструментов для обучения робототехнике является TRIK Studio.

TRIK Studio — это среда визуального программирования роботов, которая позволяет представить программу в виде диаграммы из блоков и стрелок, где блоки являются операторами, а стрелки задают порядок выполнения этих операторов. Как только программа для робота написана, можно отладить её во встроенном симуляторе. Особенно последний год имитационное моделирование активно используется как для домашних заданий, так и для проведения соревнований. Симулятор позволяет сразу увидеть результат, а при отсутствии реального робота, является возможностью узнать, как программа будет выполняться.

После того как программа отлажена в симуляторе, в TRIK Studio есть возможность преобразовать её в код на одном из текстовых языков. Когда ребенок понимает, как работает алгоритм, он может увидеть, как написанная им программа выглядит на языке программирования. Такой плавный переход от визуального программирования к текстовому является очень важной частью процесса обучения.

В то же время, на роботах отсутствуют интерпретаторы диаграмм, и поэтому для того, чтобы загрузить программу на реального робота или, например, получить формат, подходящий для внешнего симулятора, её нужно преобразовать в текст. Увлекаясь решением задачи, дети не задумываются, что не каждая программа может быть преобразована TRIK Studio в код. Случаются ситуации, когда после написания программы и отладки её в симуляторе, ребенок сталкивается с проблемой, что его программа не может быть сгенерирована.

Причиной этого является отсутствие безусловных переходов в некоторых языках программирования. В частности, для контроллера ТРИК поддерживается генерация программ на языки Python и JavaScript. Оба

языка не позволяют использовать безусловные переходы. Текущая реализация генерации текста для языков программирования старается из относительно простых по структуре программ породить читаемый понятный код, в котором легко проглядывается структура оригинальной программы. Однако не каждую программу легко преобразовать. Поэтому в некоторых случаях при генерации текстовой программы для роботов ТРИК возникают ошибки и существует необходимость расширить множество программ, которое может быть преобразовано TRIK Studio в код без безусловных переходов.

1. Постановка задачи

Целью данной работы является усовершенствование алгоритма генерации структурных текстовых программ в TRIK Studio.

Были поставлены следующие задачи:

- определить, в каких случаях текущий алгоритм не может получить структурный код;
- найти или предложить новый алгоритм, который будет работать для всех или значительного большинства программ;
- реализовать алгоритм в TRIK Studio;
- провести апробацию.

2. Обзор

В этом разделе описываются как огранизована генерация кода в TRIK Studio, алгоритм, использующийся для этого на данный момент, а также производится обзор нескольких других существующих алгоритмов для преобразования программ.

2.1. Генерация кода в TRIK Studio

Генерация кода в TRIK Studio состоит из двух шагов: генерация семантического дерева, некого промежуточного состояния программы, и преобразование этого дерева в текст на языке программирования.

Семантическое дерево — это упорядоченное, подвешенное дерево, описывающее структуру программы. Узлами являются операторы и базовые управляющие конструкции, такие как if, $while$ и другие. Вершины-конструкции могут иметь потомков, которые соответствуют вложенным в эти конструкции операторам и другим конструкциям (например, ветви if или тело цикла), которые, в свою очередь, также могут содержать потомков. Порядок вершин в дереве задает последовательность выполнения операторов. Семантическое дерево не зависит напрямую от контроллера или языка и только отображает структуру самой программы. Одно и то же семантическое дерево может быть преобразовано в текст для разных языков.

На данный момент в TRIK Studio существует два генератора семантического дерева, Structural и Goto. В зависимости от особенностей языка, в который происходит генерация, использоваться могут только один или оба. Goto генератор может создать семантическое дерево всегда, а Structural на данный момент нет. В случае контроллера ТРИК из-за отсутствия в языках программирования оператора goto применяется только Structural генератор.

Генераторы работают с графом потока управления программы, то есть графом, который отображает все возможные пути исполнения программы. Общая схема генерации текстовой программы на рис. 1

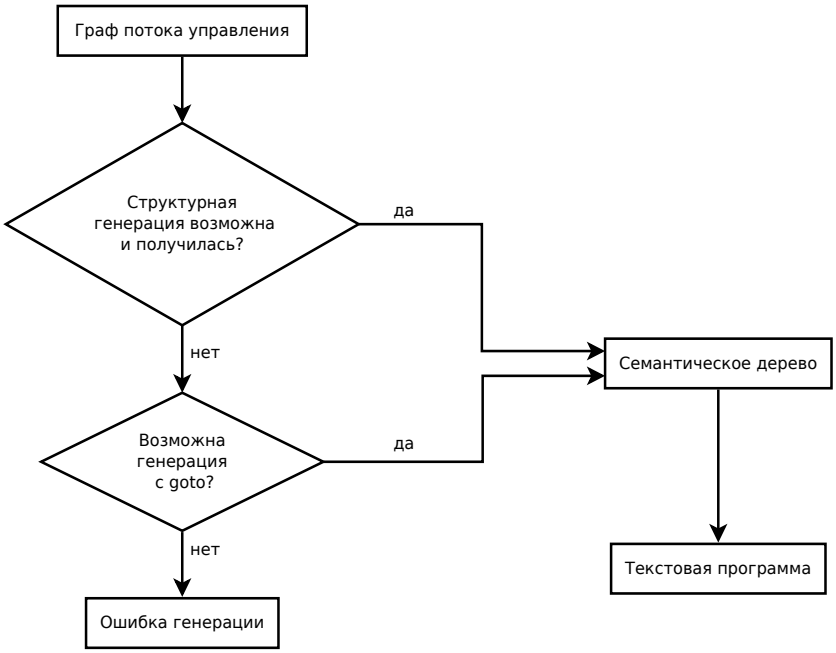


Рис. 1: Схема генерации текстовой программы в TRIK Studio.

2.2. Ограничения текущего алгоритма

В структурном генераторе генерация семантического дерева не всегда завершается успешно. Приведем несколько примеров реальных программ, когда текущий алгоритм не может получить структурную программу.

На рис. 2 изображена программа для робота, который едет по линии и считает перекрестки. В зависимости от номера встреченного перекрестка, выполняются различные действия, описанные в подпрограммах.

На рис. 3 программа для обхода роботом лабиринта. Если свободно впереди, но не слева, то едем вперед. Если свободно и впереди, и слева или только слева, то едем налево. Если же нельзя поехать ни налево, ни вперед, то разворачиваемся на 180 градусов. Обе диаграммы корректны, но на данный момент для них нельзя получить структурную

текстовую программу.

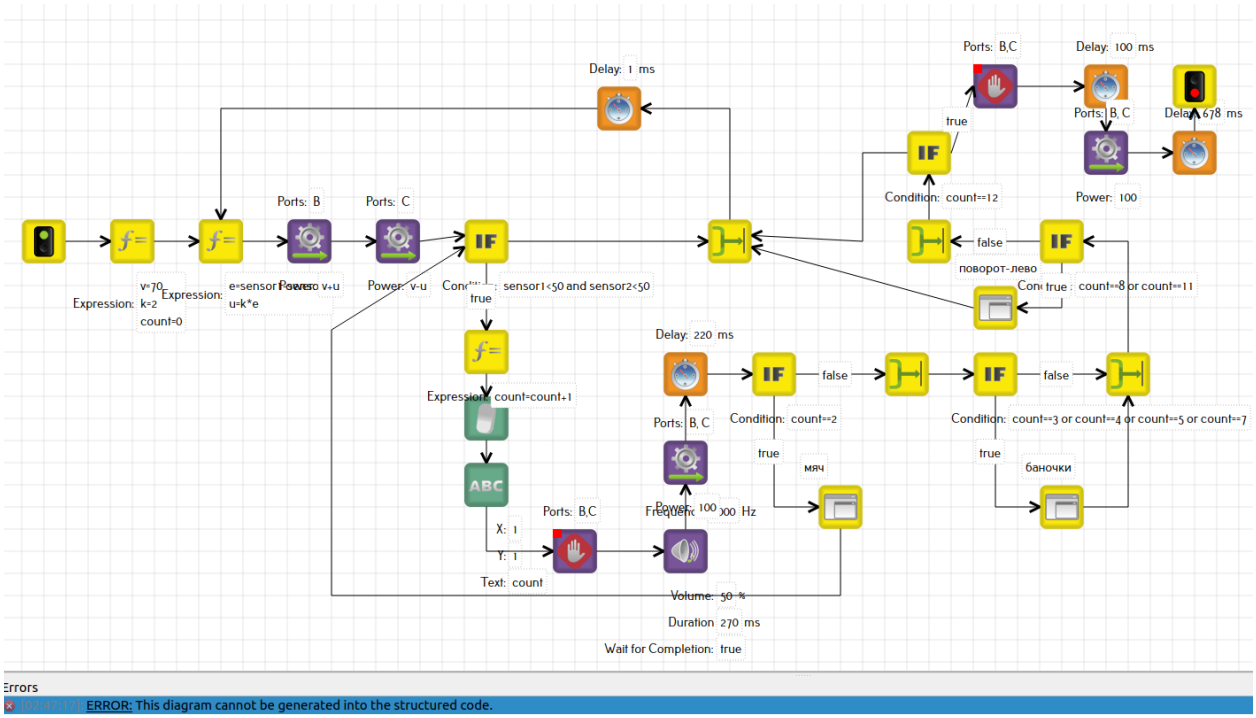


Рис. 2: Пример негенерирующейся диаграммы. Задача с подсчетом перекрестков.

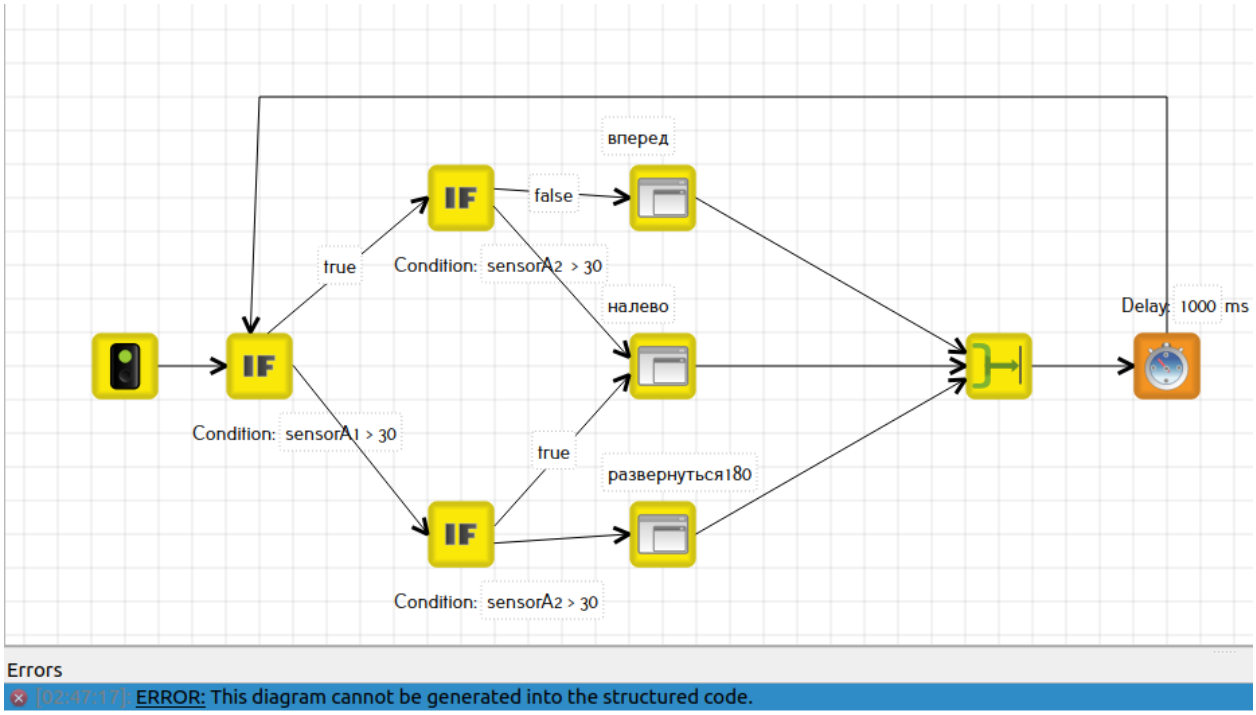


Рис. 3: Пример негенерирующейся диаграммы. Обход лабиринта.

Текущий алгоритм основан на структурном анализе [13] графа по-

тока управления. Пока возможно найти подграф текущего графа, соответствующий одному из заданных шаблонов (то есть некой структуре), этот подграф заменяется на одну вершину, и анализ продолжается. Если в результате анализа в графе осталась одна вершина, анализ завершился успешно, этот граф является структурируемым, и можно восстановить ответ. Примеры шаблонов, которые использует структурный анализ изображены на рис. 4.

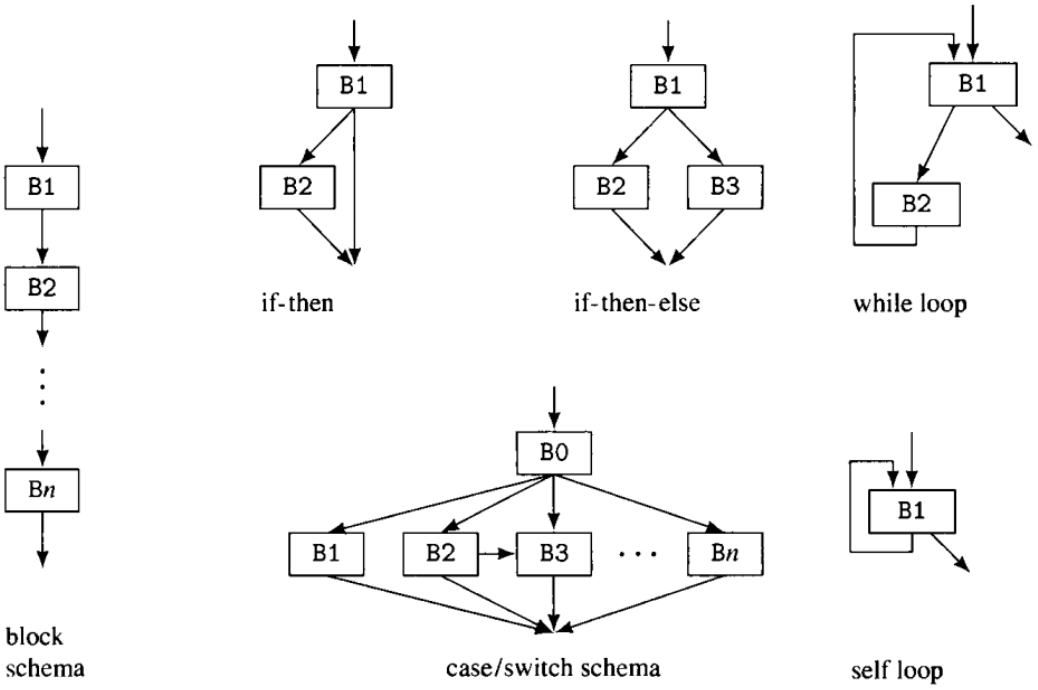


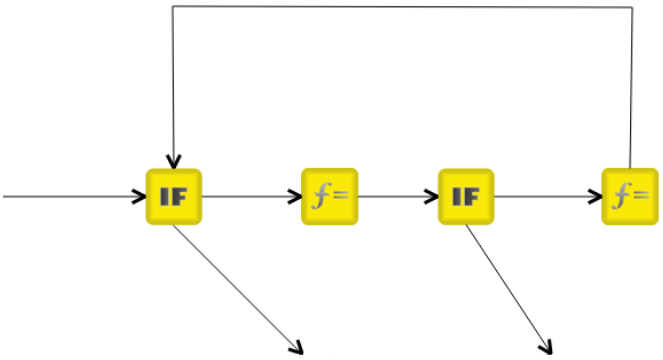
Рис. 4: Шаблоны из [13], которые ищет структурный анализ.

Также, стоит отметить, что в текущем алгоритме ещё обрабатывается дополнительный случай, не покрытый шаблонами из [13]. Если существует цикл с несколькими выходными вершинами (то есть вершинами, имеющими переход к другой вершине, не принадлежащей данному циклу), и все выходные ветви имеют общего потомка. Такой шаблон может быть представлен как цикл с операторами *break*.

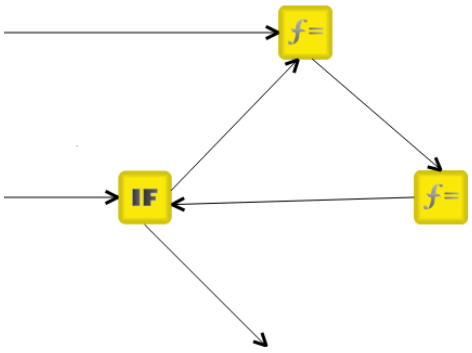
Таким образом, данный алгоритм для структурируемой программы находит её структурный вид. Но не все программы структурируемы. В [12] описано условие неструктурируемости.

Минимальные подграфы в графе потока управления, наличие которых делает программу неструктурируемой:

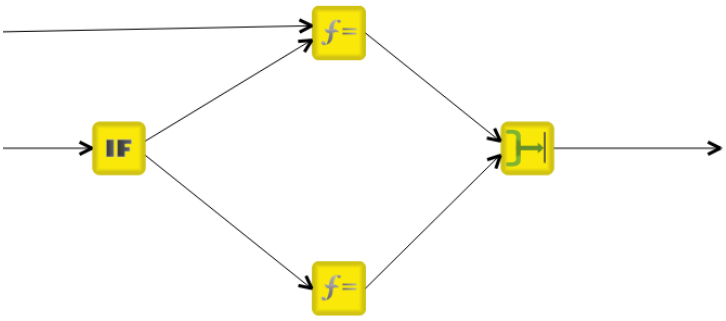
- подграф, являющийся циклом с несколькими выходными вершинами;



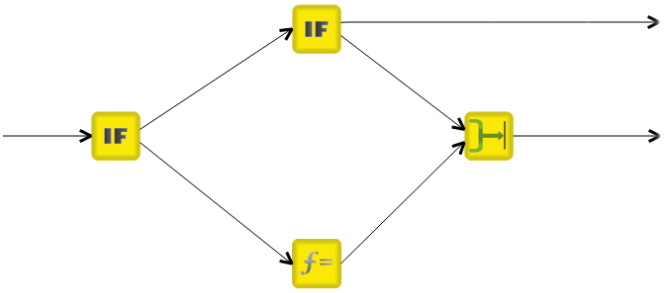
- подграф, являющийся циклом с несколькими входными вершинами;



- подграф, являющийся условным оператором и содержащий другой вход внутри ветки этого оператора;



- подграф, являющийся условным оператором, ветка которого содержит несколько выходов (из условного оператора).



Однако *Теорема Бёма-Якопини* [4] говорит, что любой исполняемый алгоритм может быть преобразован к виду, когда ход его выполнения определяется только при помощи трёх структур управления: последовательной, ветвлений и циклов; то есть к структурному виду.

Значит, можно получить функционально эквивалентную программу для неструктурируемых случаев. Но доказательство теоремы не содержит способ преобразования алгоритмов к этому виду.

2.3. Преобразование потока управления

Задача преобразования программ к более «хорошему» виду достаточно актуальна. Структурные программы легче понимать, анализировать, изменять и оптимизировать, например, автоматически распараллеливать.

После введения понятия «структурное программирование» [5], было предложено немало вариантов трансформаций программ. Двумя основными методами являются копирование вершин и использование дополнительных переменных.

2.3.1. Копирование вершин

Копирование вершин, или *node splitting*, — это техника, впервые представленная в [15], для преобразования графа G_1 в эквивалентный граф G_2.

Пусть все вершины в первоначальном графе имеют «метки». Определим, что при копировании вершины, новая вершина имеет ту же мет-

ку, что и её оригинал.

Определение. Два графа G_1 и G_2 эквивалентны тогда и только тогда, когда для каждого пути в G_1 существует путь в G_2 с теми же метками, и наоборот.

На рис. 5 изображены два эквивалентных графа, первый является неструктурируемым, второй — нет.

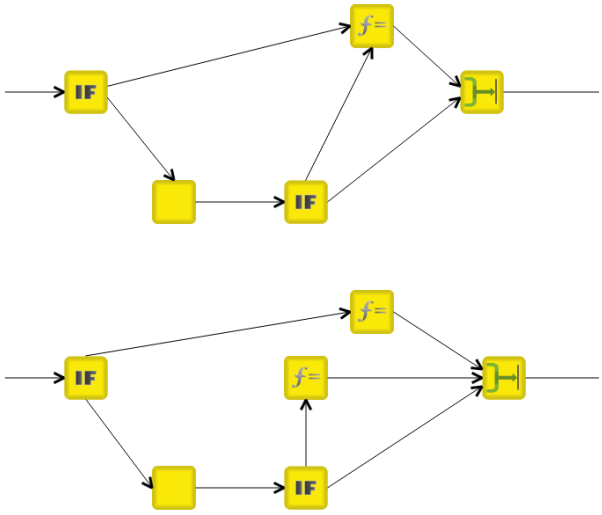


Рис. 5: Копирование вершин. Два эквивалентных графа.

Копировать можно любую вершину, которая имеет больше одного входящего в неё ребра. Копирование каждой вершины влечет за собой увеличение размера графа и программы.

2.3.2. Использование дополнительных переменных

Для того, чтобы запомнить, какой переход в определенный момент совершила программа, используют дополнительные переменные. Обычно, дополнительные переменные являются либо булевыми значениями условных выражений, либо числовыми значениями для выбора перехода.

Примеры использования можно увидеть на рис. 6.

```
pred1 = expr1;
if (pred1) {
    cmd1;
} else {
    cmd2;
}
if (pred1) {
    some actions;
} else {
    other actions;
}
```

```
if (expr1) {
    cmd1;
    selector = 1;
} else {
    cmd2;
    selector = 2;
}
if (selector == 1) {
    some actions;
}
if (selector == 2) {
    other actions;
}
```

Рис. 6: Два примера использования дополнительных переменных.

2.3.3. Существующие алгоритмы

Стоит отметить, что всегда можно преобразовать программу в структурную методом, описанным в [7]. Для этого создается один *while*, глобальный для всей программы, а также заводится «счетчик» состояний, который и определяет, какой шаг будет выполнен следующим.

Псевдокод такого решения:

```
p = 1;
while (p > 0) {
        if (p == 1) {
                perform step 1;
                p = next step from the flowchart;
        }
        if (p == 2) {
                perform step 2;
                p = next step from the flowchart;
        }
        ...
        if (p == n) {
                perform step n;
```

```
                p = next step from the flowchart;
            }
}
```

Однако в такой программе полностью теряется структура оригинальной, текст такой программы труден для анализа и изменений, и поэтому данный способ не подходит для применения.

При использовании метода копирований для преобразования программы, копирование не каждой вершины «улучшает» структуру графа. В [8] рассказывается про неконтролируемое копирование, как на рис. 7, и предлагается способ, как сделать граф *интервально сводимым* (но не структурируемым) за минимальное количество *копирований*.

Интервальная сводимость [1] — это более слабое условие, чем структурируемость. Это условие равносильно тому, что в графе нет циклов с несколькими входными вершинами, то есть только одному из условий структурируемости.

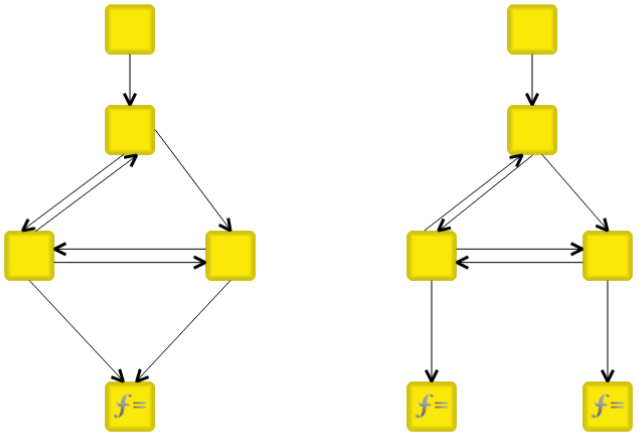


Рис. 7: Копирование вершин. После копирования структура не улучшилась, но размер графа увеличился.

Для контролируемого копирования вершин сначала строится граф из необратных ребер исходного. Если этот граф ацикличный и до любой вершины можно добраться из начальной, то исходный граф интервально сводим. Иначе алгоритм находит неразрешимые (с несколькими входами) циклы, считает для множеств вершин этих циклов множества

общих внешних доминаторов (SED) — вершин, имеющих один и тот же непосредственный доминатор[1], не принадлежащий циклу. И уже из элементов этих множеств с помощью эвристики выбираются те, что нужно копировать.

Программа, полученная из преобразованного таким способом графа остается близка к логике оригинальной программы и не усложняется для восприятия. Однако это условие не эквивалентно структурируемости, но может быть использовано, например, как часть решения.

Алгоритм, предложенный в [11], использует метод *копирования* вершин для разрешения неструктурируемых условных конструкций, а для преобразования циклов также *дополнительные переменные*.

Шаги алгоритма:

- найти все «плохие» конструкции циклов и разветвлений;
- все циклы начиная с самого вложенного преобразуются:
  - сначала избавляются от нескольких выходов с помощью дополнительных переменных как изображено на рис. 8;
  - затем избавляются от нескольких входов, поместив скопированное тело цикла до начала этого цикла, а все ребра графа, ранее ведущие в цикл, теперь указывают на вершины до цикла;
- все разветвления разрешаются копированием вершин.

Данный алгоритм преобразует любую программу в структурированную, но до начала преобразований надо провести дополнительный анализ на структурируемые части.

Ещё один алгоритм можно найти в [2]. Он тоже гарантированно приводит любую программу к структурированному виду. При этом использует дополнительные переменные для хранения результатов выражений всех условных конструкций, а также копирует вершины, но только для разрешения *интервальной несводимости*.

[1]Вершина A доминатор вершины B, если любой путь из начальной вершины до B проходит через вершину A, [10]

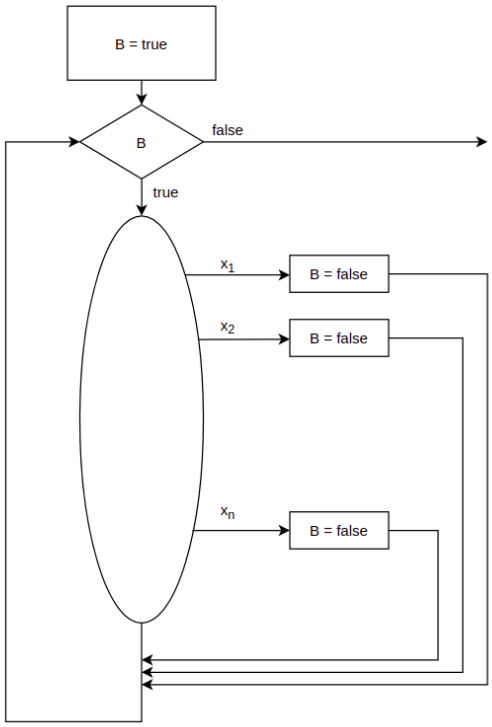


Рис. 8: Преобразование для цикла с несколькими выходами. Используется дополнительная переменная B. $x_1, x_2, .., x_n$ — условия выходов из исходного цикла

В основе алгоритма лежит решение системы уравнений методом Гаусса, где последовательно выражаются переменные. Уравнения описывают состояния программы как оператор (который выполняется на данном этапе) и *продолжения* — переходы в другие состояния программы. Все условные выражения для гарантии корректного результата при использовании более одного раза записываются в дополнительную переменную.

Решение системы:

- если переменная является рекурсией, то преобразовать в цикл с постусловием;
- факторизовать все появления данной переменной в других уравнениях;
- заменить в остальной системе данную переменную на соответствующее выражение.

Решением системы получается структурная программа, содержащая много дополнительных переменных и состоящая только из конструкций *последовательность*, *разветвление* и *цикл с постусловием*. Также алгоритм не увеличивает значительно размер графа, так как копирует вершины только в случае интервальной несводимости.

Также стоит отметить алгоритм, который приводит граф к виду *гамака* [16]. Гамак — это такой граф потока управления $< N, E, n_0, n_e >$, имеющий одну начальную вершину n_0 и одну конечную вершину n_e, что его обратный граф $< N, E^{-1}, n_e, n_0 >$, где $E^{-1} = \{(a, b) \mid (b, a)\ \epsilon\ E\}$, также является графом потока управления.

Данный метод основывается на том, что преобразования внутри подграфа, который является гамаком, не влияют на остальную программу. Для каждой неструктрурируемой части находится минимальный гамак и используются три преобразования:

- срез (cut transformation) для циклов с несколькими выходами
  - в цикле вместо внешнего перехода остается break, а после цикла добавляется условие с дополнительной переменной и переходом
- копирование назад (backward copy) для ребер, ссылающихся назад
  - это цикл repeat until, который копированием тела преобразуется в while, чтобы избежать переходов внутрь тела цикла
- копирование вперед (forward copy) для неструктурируемых условных конструкций
  - общие части пересекающихся ветвей дублируются, тем самым разрешая неструктурируемость

Ещё один алгоритм описан в [6]. Он применяется к текстовым программам, написанным на *C* и избавляется от явных использований операторов goto:

- если goto находится до своей метки, то эта часть программы преобразуется в условный оператор
- если goto находится после метки, то эта часть преобразуется в цикл
- если goto и метка находятся не на одном уровне
  - для перемещения goto из цикла используется break
  - для перемещения goto в цикл добавляются условные операторы с дополнительными переменными внутри цикла

Существуют алгоритмы [3, 9], которые преобразуют программы в структурные, но кроме упомянутых ранее трех конструкций используют ещё одну — *multi-level break*. Большинство языков, в том числе языки программирования, генерация в которые поддерживается TRIK Studio, не имеют такой функциональности. Хотя можно реализовать *multi-level break* с помощью дополнительных переменных, вместе с другими преобразованиями, которые совершают данные алгоритмы, итоговый результат будет сложен для восприятия.

2.4. Требования к получаемому результату

Среда имеет обучающие цели, так что не каждый алгоритм подойдет для генерации текстовой программы. Результатом алгоритма должен быть код программы, имеющий следующие свойства:

- структура текстовой программы близка к структуре оригинальной программы;
- отсутствие избыточного копирования блоков диаграммы/кода;
- читаемость текстовой программы.

Процесс подбора метрик для качественного и количественного анализа читаемости кода является сложной задачей и не входит в рамки

данной работы. Приближенно, этот показатель рассчитывался из субъективных оценок разработчиков TRIK Studio.

Исходя из описанных критериев было решено использовать алгоритм, основанный на решении системы методом Гаусса из [2], но после получения результата произвести дополнительный анализ для уменьшения количества используемых дополнительных переменных.

Алгоритм из [6] применяется на уже текстовой программе, которой в нашем случае изначально нет, а применение на варианте программы, полученной Goto генератором, приведет к чрезмерному количеству преобразований. Алгоритм из [16], который использует свойства графа гамака, было решено не использовать, так как TRIK Studio позволяет любое количество блоков завершения программы, и преобразование оригинальных программ к программам с одним выходом может повлиять на структуру.

Излишнее копирование является плохой практикой программирования, так как ведет к большей вероятности совершения ошибок при изменении дублирующихся участков кода, поэтому было решено не использовать метод из [11].

При небольших изменениях алгоритма [2], описанных в разделе реализация, на множестве программ, которые уже могут быть сгенерированы, результат будет эквивалентен результату предыдущего алгоритма, и, значит, оригинальная структура будет так же ясно отражена.

Алгоритм копирует вершины только в случае интервальной несводимости, что достаточно редкий случай даже в неструктурируемых программах. Поэтому для упрощения реализации было решено не использовать метод из [8].

3. Реализация

3.1. Алгоритм

Опишем более подробно алгоритм из [2].

В основе алгоритма лежит решение системы уравнений типом Гаусса, где последовательно выражаются переменные.

Система составляется из уравнений для состояний программы. Общий вид уравнений такой:

$$x_i = cmd_i \ ; \ if \ expr_i \ then \ x_j \ else \ x_k$$

где x_j и x_k — другие переменные данной системы.

Все условные операторы разделяются на:

$$x_i = (pred_i := expr_i) \ ; \ if \ pred_i \ then \ x_j \ else \ x_k$$

Порядок выражений переменных влияет на количество итоговых копирований вершин. Для нахождения оптимального порядка выражений переменных граф топологически сортируется. В полученном порядке все вершины, являющиеся головой цикла, смещаются к концу списка на место, находящееся сразу после всех вершин, составляющих тело этого цикла.

Решение системы. Последовательное исключение переменных в почти-топологическом порядке:

- если выражение является рекурсией, то факторизовать[2] все присутствующие переменные, преобразовать выражение в цикл с постусловием;
- факторизовать все появления данной переменной в других уравнениях, где переменная встречается более одного раза;
- заменить во всей системе данную переменную на соответствующее

[2]В данном случае факторизация переменной — это разделение выражения на два таких, что первое не содержит переменную, а второе содержит только её и один раз; при этом последовательное выполнение этих выражений эквивалентно первоначальному выражению.

выражение.

3.2. Выражение переменных

Так как диаграмма — это блоки и стрелки, то всё, что нам известно на момент начала алгоритма, это только соседи каждой вершины.

Поэтому для графа размера n будет n неизвестных, по одной для каждого блока, и система уравнений, где уравнения вида

$$x_i = cmd_i \ ; \ x_j$$

для простых блоков, и

$$x_i = if \ cond_i \ then \ x_j \ else \ x_k$$

для условных блоков.

Важным условием применения алгоритма является определить уже структурируемые части программы до начала процесса нормализации.

Если не сделать этого, то во время нормализации структура этих частей может измениться. Например, из обычного вложенного условия получится несколько последовательных if. Покажем на примере.

$$\begin{cases} x_1 = if \ cond_1 \ then \ x_2 \ else \ x_6 \\ x_2 = if \ cond_2 \ then \ x_3 \ else \ x_4 \\ x_3 = cmd_3 \ ; \ x_5 \\ x_4 = cdm_4 \ ; \ x_5 \\ x_5 = cmd_5 \ ; \ x_6 \\ x_6 = cmd_6 \end{cases}$$

Порядок выражений переменных будет следующим: 2, 3, 4, 5, 6.

$$\begin{cases} x_1 = if\ cond_1\ then\ (if\ cond_2\ then\ x_3\ else\ x_4)\ else\ x_6 \\ x_3 = cmd_3\ ;\ x_5 \\ x_4 = cdm_4\ ;\ x_5 \\ x_5 = cmd_5\ ;\ x_6 \\ x_6 = cmd_6 \end{cases}$$

$$\begin{cases} x_1 = if\ cond_1\ then\ (if\ cond_2\ then\ (cmd_3\ ;\ x_5)\ else\ (cdm_4\ ;\ x_5))\ else\ x_6 \\ x_5 = cmd_5\ ;\ x_6 \\ x_6 = cmd_6 \end{cases}$$

$$\begin{cases} x_1 = if\ cond_1\ then\ (if\ cond_2\ then\ cmd_3\ else\ cdm_4)\ else\ x_6; \\ \qquad if\ (cond_1\ and\ (cond_2\ or\ !cond_2))\ then\ (cmd_5\ ;\ x_6) \\ x_6 = cmd_6 \end{cases}$$

$$\begin{cases} x_1 = if\ cond_1\ then\ (if\ cond_2\ then\ cmd_3\ else\ cdm_4)\ ; \\ \qquad if\ (cond_1\ and\ (cond_2\ or\ !cond_2))\ then\ cmd_5\ ; \\ \qquad if\ (!cond_1\ or\ (cond_1\ and\ (cond_2\ or\ !cond_2)))\ then\ cmd_6 \end{cases}$$

И, если упростить условные выражения:

$$\begin{cases} x_1 = if\ cond_1\ then\ (if\ cond_2\ then\ cmd_3\ else\ cdm_4)\ ; \\ \qquad if\ cond_1\ then\ cmd_5\ ; \\ \qquad cmd_6 \end{cases}$$

На самом деле условные выражения в первом и втором if одинаковые, можно их объединить, и тогда результатом будет именно та конструкция, которую хотелось бы получить. В более сложных конструкциях выражения не получаются всегда равными. Пример с большим количеством вершин не приводится только из-за объема вычислений.

Однако этот пример уже показывает, что если у какой-то вершины больше одного входящего ребра, то переменная, соответствующая этой вершине, будет факторизована и вынесена вне конструкции, в которой ей следует находиться.

Для того, чтобы сохранить «естественные» *условные операторы*, будем перед каждой факторизацией проверять, не заканчиваются ли две ветви какого-то if одними и теми же конструкциями. Если заканчиваются, то значит, что if уже завершен, а общий «хвост» вынесем сразу после if. После такого преобразования переменная будет встречаться только один раз в уравнении и потому не будет факторизована.

Это будет сделано корректно для любого структурируемого if, так как вершины обходятся в топологическом порядке, а свойство этого порядка в том, что сначала будет выражена голова if, потом обе ветви и только потом общий потомок ветвей.

Для конструкции *последовательность* никаких дополнительных вычислений не требуется, так как в *последовательности* во вторую вершину ведет всего одно ребро, а значит переменная не будет факторизована до её замены.

Для конструкции *цикл* также не потребуется дополнительных действий, так как по выбранному алгоритмом порядку выражений переменных, все головы циклов будут выражены после тела цикла. Это значит, что в отличие от if, тело цикла будет точно сформировано до замены переменной, в которой находится этот цикл, и значит конструкция не будет разделена.

3.3. Добавление оператора break

Результатом выбранного алгоритма является программа только из трёх управляющих конструкций. Оператор break используется в некоторых других описанных ранее алгоритмах [6, 16], а также частично в предыдущей реализации генерации текстовой программы для циклов с несколькими выходами. Без использования break условие продолжения цикла в таких случаях содержит дополнительные переменные и

становится большого размера, а потому менее читаемым, так что стоит добавить его и в данной реализации.

Для этого изменим действия при преобразовании рекурсивных переменных.

- Если переменная является рекурсией, то факторизовать её появления, преобразовать в цикл; для всех других продолжений в теле цикла заменить продолжения на break, а факторизованное условие с продолжением поместить сразу после цикла.

Это изменение никак не меняет часть программы, находящуюся после цикла. Внутри цикла операторы break обеспечат завершение этого цикла в тех же случаях, что и составное условие, и потому составное условие более не требуется.

3.4. Уменьшение количества дополнительных переменных

Одним из первых шагов алгоритма является запись значений всех условных выражений в дополнительные переменные. Это делается для обеспечения корректных вычислений в синтетических условных выражениях, но совершенно необязательно в таких случаях:

```
cond1 = expression1;
if (cond1) {
        cond2 = expression2;
        if (cond2) {
                do something;
        } else {
                do something else;
        }
}
```

То есть, если дополнительная, или синтетическая, переменная используется только один раз сразу после создания, то её создание может быть исключено.

3.5. Преобразование циклов с постусловием

Результатом алгоритма является программа, содержащая циклы только с постусловием.

Тогда, например, для простого цикла с предусловием код на C будет выглядеть так:

```
do {
        condition = expression;
        if (condition) {
                do something;
        } else {
                do something else;
        }
} while (condition)
```

Поэтому для того, чтобы сохранить исходным циклам с предусловием их привычную форму, а также для того, чтобы уменьшить количество синтетических переменных (в данном случае, использования переменной condition можно избежать), добавим ещё одну проверку, если переменная оказалась рекурсивной и была преобразована в цикл.

- Если полученный цикл состоит только из одного условного оператора, условие которого эквивалентно/противоположно условию цикла, то преобразуем цикл с постусловием в цикл с предусловием, а операторы из ветви else/then поместим сразу после тела цикла

4. Архитектура

Новый алгоритм был реализован как часть существующего класса StructuralControlFlowGenerator, создающего семантическое дерево. Решение системы происходит в классе Structurizer, который принимает граф потока управления программы, а возвращает программу в виде StructurizerNode, которая является деревом.

StructurizerNode описывает «выражение». Это может быть простой оператор, условный оператор, цикл или же последовательность операторов — то есть, произвольная программа.

Каждая используемая конструкция и действия, совершаемые над ней, описаны в соответствующих классах.

На рис. 9 представлена новая архитектура генерации структурного семантического дерева.

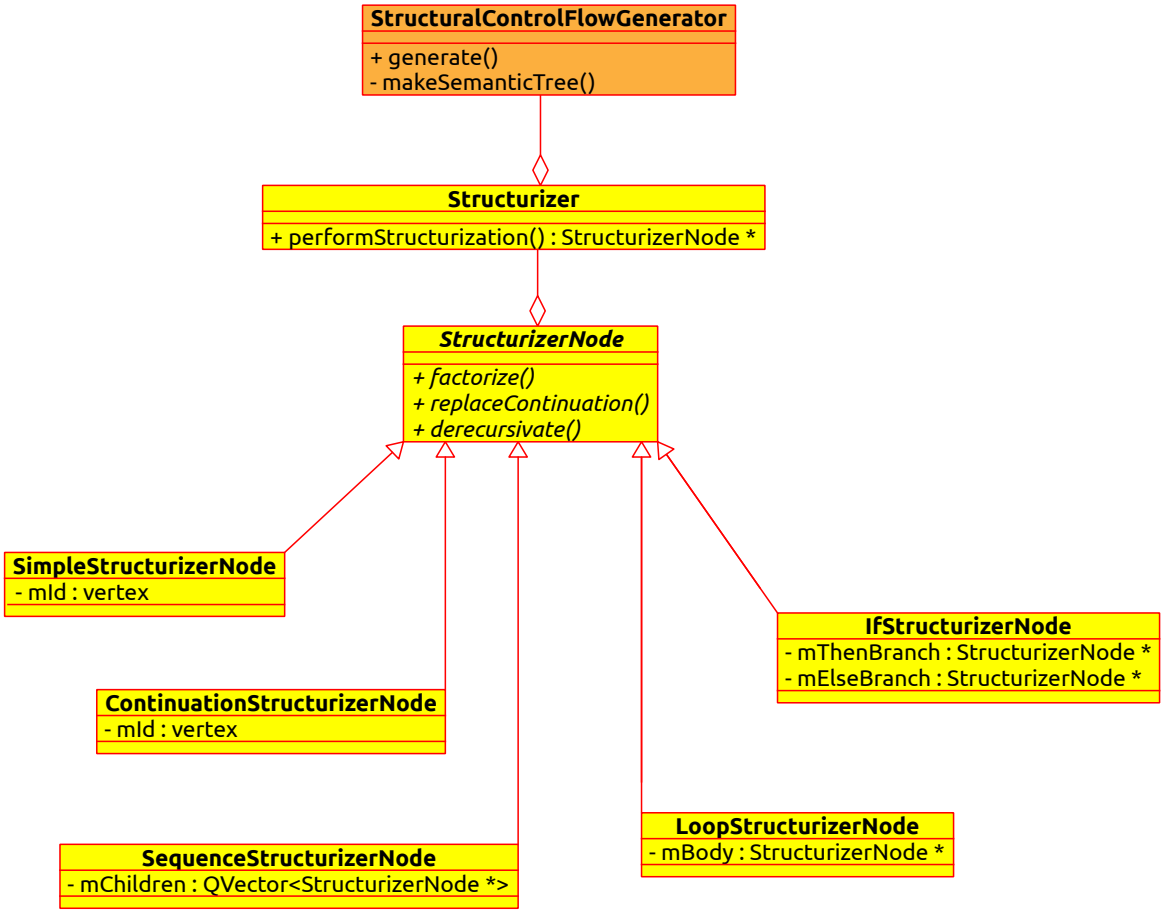


Рис. 9: Архитектура модуля TRIK Studio, отвечающего за построение семантического дерева.

Для части TRIK Studio, отвечающей за генерацию текстовой программы из семантического дерева, были добавлены генераторы для *синтетических условных конструкций* и *синтетических переменных*, которые используются для хранения информации о значении в ранее вычисленном условном выражении.

Синхронизация названий синтетических переменных с условиями происходит в добавленном классе ReadableLabelManager, который назначает и хранит уникальные названия переменных, имеющие удобные для чтения пользователем имена.

На рис. 10 представлена архитектура этого модуля. Желтым выделены классы, которые были добавлены.

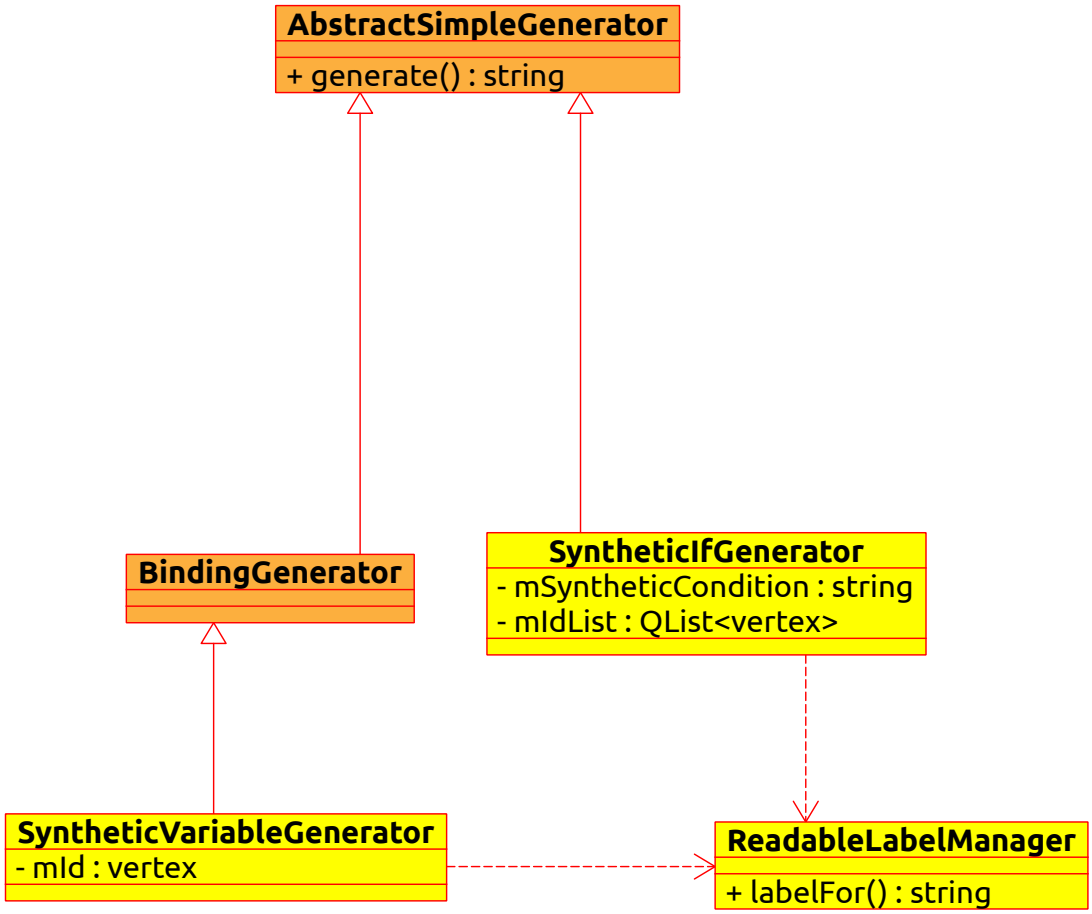


Рис. 10: Архитектура модуля TRIK Studio, отвечающего за генерацию текстовой программы добавленных синтетических блоков.

5. Тестирование

Новый алгоритм был протестирован на различных диаграммах, размером от 2 до 71 блоков, содержащих большое количество последовательных элементов, условий, циклов, в том числе пересекающихся и вложенных друг в друга.

Восемнадцать структурируемых программ, на которых работала предыдущая реализация генерации текста, были протестированы в первую очередь. Проверялось, генерируется ли программа, её корректность, а также присутствие в коде дополнительных переменных и копирования кода.

No. теста	Пред. алг.	Нов. алг.	Корр.	Доп. перем.	Коп. кода
1	+	+	+	−	−
2	+	+	+	−	−
3	+	+	+	−	−
4	+	+	+	−	−
5	+	+	+	−	−
6	+	+	+	−	−
7	+	+	+	−	−
8	+	+	+	−	−
9	+	+	+	−	−
10	+	+	+	−	−
11	+	+	+	−	−
12	+	+	+	−	−
13	+	+	+	−	−
14	+	+	+	−	−
15	+	+	+	−	−
16	+	+	+	−	−
17	+	+	+	−	−
18	+	+	+	−	−

Все программы были сгенерированы без синтетических переменных, а их код эквивалентен тому, что получал предыдущий алгоритм.

Также, алгоритм был проверен на шестнадцати неструктурируемых программах, которые не могли быть сгенерированы раньше. Проверялось наличие неструктурируемых условных операторов, циклов в диа-

грамме, а также возможность получить код, его коррекность, наличие дополнительных переменных и копирование кода.

No. тест	Нестр. усл.	Нестр. циклы	Пред. алг.	Нов. алг.	Корр.	Доп. перем.	Коп. кода
1	+	−	−	+	+	+	−
2	+	−	−	+	+	+	−
3	+	−	−	+	+	+	−
4	+	−	−	+	+	+	−
5	+	−	−	+	+	+	−
6	−	+	−	+	+	−	+
7	−	+	−	+	+	+	−
8	+	+	−	+	+	+	−
9	+	+	−	+	+	+	+
10	+	+	−	+	+	+	−
11	−	+	−	+	+	+	+
12	+	−	−	+	+	+	−
13	+	+	−	+	+	+	−
14	+	−	−	+	+	+	−
15	+	+	−	+	+	+	+
16	+	−	−	+	+	+	−

Для всех неструктурируемых диаграмм был получен текстовый результат. На рис. 11 код, сгенерированный для неструктурируемой программы на рис. 3.

```
49    while (true) {
50        temp_1 = brick.sensor(A1).read() > 30;
51        if (temp_1) {
52            temp_2 = brick.sensor(A2).read() > 30;
53            if (!temp_2) {
54                forward();
55            }
56        } else {
57            temp_3 = brick.sensor(A2).read() > 30;
58            if (!temp_3) {
59                turn180();
60            }
61        }
62        if (temp_1 && temp_2 || !temp_1 && temp_3) {
63            left();
64        }
65        script.wait(1000);
66
67    }
```

Рис. 11: Сгенерированный код для неструктурируемой программы на рис. 3 «Обход лабиринта»

6. Апробация

Для оценки читаемости кода, получаемого в случае неструктурируемых программ, был проведен опрос.

Участникам опроса было предложено решить 5 заданий на чтение, изменение и понимание кода, сгенерированного TRIK Studio. Задания для опроса были сделаны подобно заданиям, описанным в [14]. Также учитывались потребовавшееся время и сложность по мнению опрошенных.

В опросе участвовали 29 человек: 16 детей и 13 взрослых. Приведем таблицу с результатами опроса: средним временем в минутах, средней сложностью заданий по шкале от 1 до 10, а также общим количеством правильных ответов.

Возраст	Задание	Ср. время	Ср. слож.	Прав. отв.
≤ 18	1	2.81	4.13	14
	2	1.31	1.93	16
	3	1.63	2.81	13
	4	2.31	4.56	8
	5	1.25	2.5	11
> 18	1	2.46	4.46	13
	2	1.15	2.62	13
	3	1.38	2.62	11
	4	2.85	4.62	9
	5	1.77	3.69	11

Среди опрошенных школьников только трое были не знакомы с TRIK Studio. Два задания оказались значительно сложнее остальных, а также потребовали больше времени. В целом, правильные ответы были даны на 77.5% заданий.

Среди опрошенных взрослых с TRIK Studio были знакомы два человека. Больше всего ошибок было в тех же заданиях, которые вызвали сложность и у школьников. В целом, правильные ответы были даны на 87.7% заданий.

В обеих группах участники справились с большинством заданий, что позволяет предположить, что программы достаточно читаемые.

Заключение

В ходе данной работы были достигнуты следующие результаты:

- было изучено как работает предыдущий алгоритм, а также выявлены случаи, когда он не может получить результат;
- изучены существующие алгоритмы для решения данной задачи, один из которых был выбран и реализован с улучшениями;
- проведено тестирование как на случаях, с которыми справлялся предыдущий алгоритм, так и на неструктурируемых программах;
- проведена апробация для оценки читаемости получаемой текстовой программы.

Полученный новый алгоритм работает для любой произвольной диаграммы, для неструктурируемых программ порождает достаточно читаемый код, а для структурируемых программ дает такой же результат, что и предыдущий алгоритм.

Список литературы

[1] В.Н. Касьянов. Оптимизирующие преобразования программ. — Наука, 1988.

[2] Ammarguellat Zahira. A control-flow normalization algorithm and its complexity // Software Engineering, IEEE Transactions on. — 1992. — 04. — Vol. 18. — P. 237 – 251.

[3] Baker Brenda S. An Algorithm for Structuring Flowgraphs // J. ACM. — 1977. — Vol. 24, no. 1.

[4] Böhm Corrado, Jacopini Giuseppe. Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules // Commun. ACM. — 1966. — Vol. 9, no. 5.

[5] Dijkstra E. Goto statment considered harmful // Communications of The ACM - CACM. — 1968. — 01.

[6] Erosa Ana, Hendren Laurie. Taming Control Flow: A Structured Approach to Eliminating Goto Statements // IEEE International Conference on Computer Languages. — 2000. — 05.

[7] Harel David. On Folk Theorems // Commun. ACM. — 1980. — Vol. 23, no. 7.

[8] Janssen Johan, Corporaal Henk. Controlled Node Splitting. — 2006. — 01. — P. 44–58.

[9] Kozen Dexter, Tseng Wei-Lung. The Böhm–Jacopini Theorem Is False, Propositionally. — 2008. — 07. — P. 177–192.

[10] Lengauer Thomas, Tarjan Robert Endre. A Fast Algorithm for Finding Dominators in a Flowgraph // ACM Trans. Program. Lang. Syst. — 1979. — Vol. 1, no. 1. — P. 121–141.

[11] M. Howard Williams, Ossher H. L. Conversion of Unstructured Flow Diagrams to Structured Form // Comput. J. — 1978. — Vol. 21, no. 2. — P. 161–167.

[12] McCabe Thomas J. A Complexity Measure. ICSE '76. — Washington, DC, USA : IEEE Computer Society Press, 1976.

[13] Muchnick Steven S. Advanced Compiler Design and Implementation. — San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1998.

[14] Oliveira Delano, Bruno Reydne, Madeiral Fernanda. Evaluating Code Readability and Legibility: An Examination of Human-centric Studies // IEEE International Conference on Software Maintenance and Evolution. — 2020. — 09. — P. 348–359.

[15] Peterson W. W., Kasami T., Tokura N. On the Capabilities of While, Repeat, and Exit Statements // Commun. ACM. — 1973. — Vol. 16, no. 8.

[16] Zhang Fubo, H. D'Hollander Erik. Using Hammock Graphs to Structure Programs // IEEE Trans. Softw. Eng. — 2004. — Vol. 30, no. 4. — P. 231–245.