Санкт-Петербургский государственный университет

Кафедра системного программирования

Зимин Григорий Александрович

Синтез моделей внешних вызовов для символьного исполнения

Выпускная квалификационная работа

Научный руководитель:
к.т.н., доц. каф. СП Литвинов Ю.В.

Рецензент:
ООО «ИнтеллиДжей Лабс»
разработчик ПО Мордвинов Д.А.

Санкт-Петербург
2018

SAINT PETERSBURG STATE UNIVERSITY

Software Engineering

Grigorii Zimin

External call models synthesis for symbolic execution

Graduation Thesis

Scientific supervisor:
Associate Professor Yurii Litvinov

Reviewer:
"IntelliJ Labs Co.Ltd"
Software Developer Dmitrii Mordvinov

Saint Petersburg
2018

Оглавление

1. Введение

Создание программного обеспечения (ПО) требует значительных ресурсов и является сложным процессом. Корректность и надежность разработанного ПО достигается различными способами. Общепринятым подходом для обеспечения качества ПО является тестирование. Тестирование предполагает, что о правильности программ можно судить по их поведению на конечном наборе примеров (тестовых сценариях), что не дает гарантий корректной работы на других примерах. Как писал Э.В. Дейкстра: «Тестирование программы может весьма эффективно продемонстрировать наличие ошибок, но безнадежно неадекватно для демонстрации их отсутствия».

Альтернативный подход к доказательству того, что ошибок в программе нет, — использование формальных методов. К примеру, *метод проверки моделей* (model checking) [1] предполагает проверку модели программы на удовлетворение заданной спецификации. Однако, несмотря на то, что формальные методы успешно применяются в отдельных областях разработки программного обеспечения [2, 3, 4], существует множество ограничений [5, 6, 7, 8, 9], из-за которых индустрия в целом не готова к их широкому применению.

Существует подход *символьного исполнения* (symbolic execution) [10] программ, впервые предложенный в 1976 году и продолжающий активно развиваться [11]. Данный метод позволяет исследовать все возможные пути исполнения программы одновременно, путем замены конкретных входных данных на «символьные». Такой подход позволяет получить более строгие гарантии на проверяемые свойства программы и находить ошибки. Также этот подход позволяет автоматически генерировать тесты, выдавая значения, на которых программа повела себя некорректно [12].

Символьное исполнение программ не лишено недостатков: современные сложные программные продукты используют множество сторонних библиотек, вызывают различные утилиты. Такие вызовы для символьных исполнителей являются «черными ящиками». Один из спосо-

бов анализа в таких ситуациях для символьного исполнителя — исполнение таких участков кода конкретно на каком-то ограниченном числе примеров, что, однако, не дает гарантий того, как поведет себя программа на других наборах данных. Другой способ предполагает разработку моделей используемых библиотек/утилит вручную. Например, инструмент для проверки Java программ Java PathFinder [13] содержит в себе модель фреймворка Java Swing [14]. Один из самых новых подходов к решению этой проблемы предлагает автоматически синтезировать такие модели фреймворков для символьных исполнителей [15].

Синтезирование программного кода как задача автоматического поиска программы на заданном языке программирования, которая удовлетворит намерениям пользователя, к примеру, заданным в виде полной логической спецификации [16], в виде примеров сопоставления входов программы ее выходам [17], естественным языком [18], частичными программами [19], — является еще одним способом создания надежных и безопасных программ [20]. Классическая формулировка задачи синтеза программы состоит в поиске программы `P`, которая соответствует спецификации ϕ, заданной в виде логической формулы. Решение таких задач изначально рассматривалось как решение задачи доказательства теорем [21]. Такие программы заведомо корректны и надежны.

Синтаксически-направленный синтез (syntax-guided synthesis) — это более современный подход, который предполагает, что вместе с семантическими ограничениями, выражающими намерение пользователя и заданными в виде спецификаций, пользователь дополнительно задаст синтаксические ограничения на пространство поиска синтезируемой программы [22]. Формулировка задачи синтаксически-направленного синтеза (SyGuS) выделяет и обобщает основную вычислительную задачу, решаемую в различных работах, относящихся к синтезу программного кода [23, 24, 25, 26, 27, 28]. Стандартизованный формат описания этой задачи [29] лег в основу соревнования *syntax-guided synthesis competition (SyGuS-Comp)* [30, 31], в рамках которого решатели, например `CVC4` [32, 33], и `EUSolver` [34], соревнуются в скорости и количестве решенных задач.

2. Постановка задачи

Целью работы является создание прототипа инструмента для автоматического синтеза моделей внешних вызовов на основании трасс программ, которые эти вызовы содержат.

Для достижения этой цели были поставлены следующие задачи:

- разработать метод сведения задачи синтеза моделей внешних вызовов по трассам к проблеме синтаксически-направленного синтеза;
- создать прототип инструмента для автоматического синтеза символьных моделей внешних вызовов;
- провести апробацию инструмента с помощью решателей `CVC4` и `EUSolver`.

3. Обзор

3.1. Символьное исполнение

Символьное исполнение — это подход, позволяющий анализировать программу, показывая какие входные параметры кода вызывают выполнение каких частей программы. Во время символьного исполнения каждый вход программы заменяется на символьное значение, например (λ, β, ...) вместо конкретных значений, к примеру (42, "$str1$", ...). При исполнении все присваивания, встретившиеся на пути исполнения, обновляются с использованием символьных значений.

```
void foo(int a , int b) {
    int x = 1, y = 0;
    if (a != 0) {
        y = 3 + x;
        if (b == 0)
            x = 2 * (a + b);
    }
    assert (x - y != 0);
}
```

Листинг 1: Фрагмет кода для анализа.

На Листинге 1 представлен код функции, цель которой — найти все входы, на которых проверка `assert` не выполняется. Так как каждый входной параметр имеет 2^{32} различных значений, то полный перебор даже такого простого фрагмента кода неэффективен. При символьном исполнении каждое значение, что не может быть определено за счет статического анализа кода (например, значения входных параметров функций или результат системных вызовов, что читают данные с потоков), представляется как α_i. В любой момент времени символьное исполнение описывается состоянием ($stmt$, σ, π), где:

- $stmt$ — следующий оператор;
- σ — символьный буфер, содержащий информацию о символьных выражениях над α_i;

- π — описывает ограничения пути исполнения.

Интерпретируя программу, мы получаем дерево ветвей исполнения (см. Рис. 1).

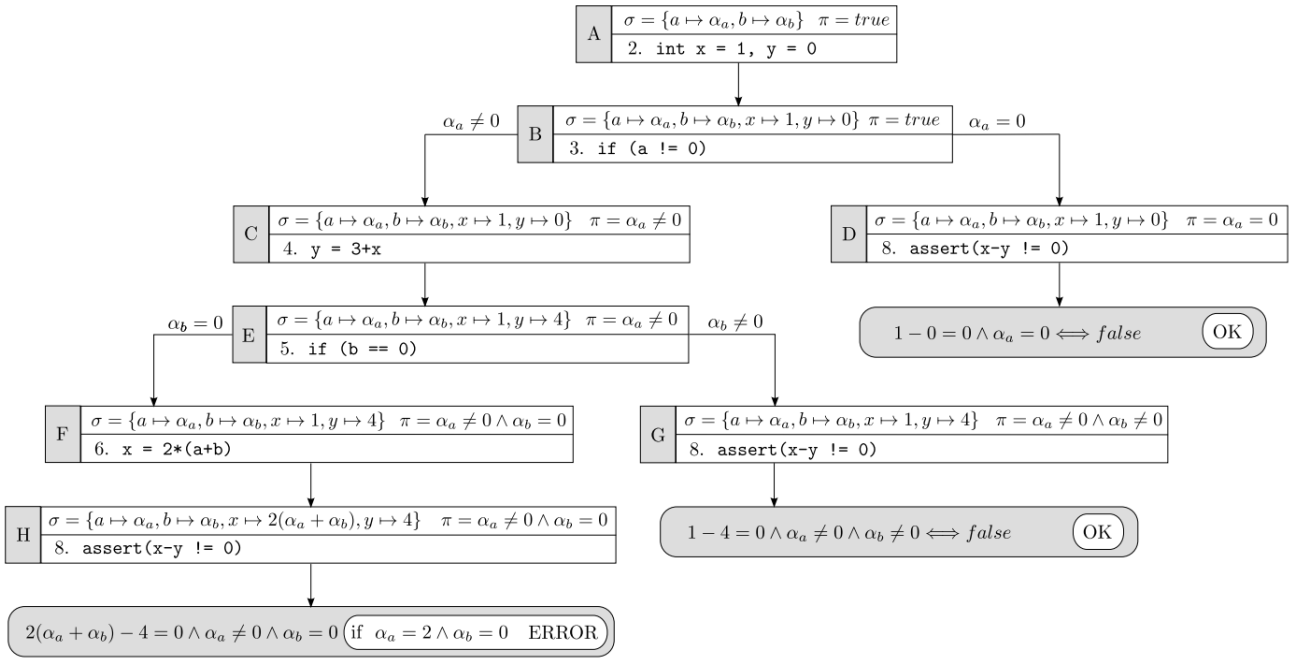


Рис. 1: Символьное исполнение. Рисунок взят из [11].

Вычисление происходит следующим образом.

1. Вычисление присваивания $x = e$ обновляет символьный буфер σ, ассоциируя с x новое символьное выражение e_s. Такая ассоциация записывается как $x \to e_s$, где e_s получено вычислением выражения в контексте текущего оператора.

2. Вычисление условного перехода `if` e `then` s_{true} `else` s_{false} влияет на ограничения пути π. Символьное исполнение создает два независимых символьных состояния с ограничениями π_{true} и π_{false} и продолжает работу независимо с двумя путями $\pi \wedge e_s$ и $\pi \wedge \neg e_s$ соответственно.

Теперь можно получить значения и сгенерировать по ним тест, который фрагмент кода в примере не пройдет. Для этого можно использовать решатели для задачи *выполнимости формул в теориях*

(satisfiability-modulo theory, SMT) [35, 36], отправив им запрос $2 * (\alpha_a + \alpha_b) - 4 = 0 \wedge \alpha_a \neq 0 \wedge \alpha_b = 0$.

Как было показано на примере Листинга 1, символьное исполнение программ может исследовать все возможные пути исполнения программы, по которым может пойти исполнение программы при конкретном исполнении от конкретных входных значений. Однако на практике это, как правило, невозможно: сложные приложения часто представляют собой набор взаимодействующих, различных программ. Реализация символьного исполнителя, который смог бы проанализировать весь стек технологий, была бы очень сложна, особенно учитывая различные внешние эффекты, возникающие при работе программ (работа с файлами, сетью и прочее). Символьное исполнение сталкивается со следующими известными сложностями.

1. Полное исследование всех инструкций может привести к экспоненциальному взрыву состояний, что не даст анализу дойти до каких-то участков кода.

2. Символьные исполнители постоянно взаимодействуют с решателями SMT формул во время исполнения. SMT-решатели являются «бутылочным горлышком», так как могут очень долго обрабатывать сложные запросы.

3. Вызовы внешних программ и компонент могут быть не трассируемыми для символьного исполнителя (будут для него «черным ящиком»).

Первая проблема решается с помощью различных эвристик, направляющих исполнение по путям, содержащим нерассмотренные инструкции [37, 38, 39, 40] или с помощью пользователя [41], вторая — с помощью оптимизации и адаптации решателей для символьного исполнения [42, 43, 44] или подхода *смешанного исполнения (concolic execution)* [45, 46].

Во всех случаях, когда программа взаимодействует с окружением (проблема 3), например, с файловой системой, переменными окруже-

ния, сетью — символьное исполнение сталкивается с необходимостью рассматривать весь стек технологий, окружающий их, включая библиотеки, ядра операционных систем, драйвера и прочее. Ранние работы [47, 45, 39] обрабатывали внешние вызовы, вызывая их с конкретными значениями, что сильно ограничивало возможности анализа.

Одним из способов обхода является написание абстрактных моделей, которые будут охватывать это взаимодействие. Например, в символьном интерпретаторе KLEE [38] поддержаны символьные файлы и символьная файловая система, чьи размеры и содержание задает пользователь. Так как количество функций в стандартной библиотеке очень велико, то написание моделей для них очень затратно и подвержено ошибкам [48], поэтому модели обычно реализованы на уровне системных, а не библиотечных вызовов. Однако данные модели были описаны вручную и могут также содержать в себе ошибки и неточности. Более того, системный код может «эволюционировать» со временем, что потребует обновления соответствующих моделей [15].

Другая ветвь исследований, появившаяся совсем недавно, заключается в попытке генерировать модели автоматически, так как только такой способ может быть жизнеспособным при взаимодействии с закрытым исходным кодом фреймворков, нативными методами `Java`, неуправляемым кодом платформы `.Net`. Например, в работе [49] авторы использовали срезку кода, чтобы извлечь участки кода, отвечающие за взаимодействие с определенным набором полей, относящихся к анализу, и на их основе строили абстрактные модели. В работе [15] был предложен способ синтеза моделей фреймворков Java по трассам и логам программ, взаимодействующих с ними, на основе комбинирования паттернов проектирования.

3.2. Синтез программ

Синтез программ — это задача автоматического поиска программы на заданном языке программирования, которая удовлетворит намерениям пользователя. Решатели задачи синтеза выполняют поиск на про-

странстве программ с целью найти программу, которая удовлетворит различным ограничениям, к примеру ограничениям, заданными в виде сопоставления входам выходам, демонстрациям, ограничениям на естественном языке или в виде частично заданных программ, утверждений (assertions) и другими.

Синтез программ — активно развивающая область на стыке таких направлений как машинное обучение, языки программирования, формальная верификация, логики различных порядков. Идея построения алгоритмов с доказательством их корректности путем рассмотрения подпроблем появилась еще в 1932 году в работе по конструктивной математике [50]. После развития в 70-х годах систем автоматического доказательства теорем [51] появились первые работы по их применению к задаче синтеза [52, 53]; при этом основной идеей было использование систем доказательства теорем, чтобы сначала доказать что-либо для спецификации, заданной пользователем, а потом на основе доказательства извлечь соответствующую логику синтезируемой программы. Следующий распространенный подход основывался на синтезе программ путем неоднократной трансформации полной спецификации, пока она не удовлетворит желаемой низкоуровневой программе [54].

Эти подходы представляли собой *дедуктивный синтез*. Дедуктивный синтез — это подход, при котором необходима полная формальная спецификация желаемой программы [55]. В большинстве случаев написание спецификаций для таких программ оказывается настолько же затратным, насколько написание самой программы. Это привело к новому подходу — индуктивному синтезу. Индуктивный синтез основан на индуктивно заданных спецификациях, например, примерах входных и выходных данных [56, 57], демонстрациях [58].

Синтез обычно включает в себя три компоненты: способ задания ограничений, выражающих намерение пользователя, пространство программ, на котором будет выполняться поиск, и саму процедуру поиска. Намерение пользователя может быть выражено в форме отношений между входами и выходами программы, примерами входа и выхода, демонстрациями, естественным языком, уже существующими неэффек-

тивными программами, логами (трассами) программ. Пространство поиска может быть описано над функциональными программами (с возможными ограничениями на структурные операторы) или задаваться над ограниченными моделями вычислений, например регулярными или контекстно-свободными грамматиками, или в виде сжатых логических представлений. Процедура поиска может быть основана на полном переборе, алгебре пространства версий [59], техниках машинного обучения [60], таких как «распространение доверия» (belief-propagation) и генетическое программирование, или с помощью логических методов, которые обычно состоят из двух шагов: генерация ограничений (constraint generation) и решение их (constraint solving) [61], зачастую с использованием SAT и SMT-решателей.

3.3. Синтаксически-направленный синтез

Современные подходы к синтезу накладывают синтаксические ограничения на пространство поиска искомой функции. Такой подход не только ускоряет процедуру поиска, но и позволяет получать более осмысленные результаты, например, фреймворк Sketch [19] позволяет пользователям написать частичную структуру программы, оставляя «дырки», которые будут заполнены в соответствие с заданными утверждениями и ограничениями, другими словами, с заданной спецификацией.

Формализация задачи синтеза с заданными синтаксическими ограничениями на пространство поиска получила название синтаксически-направленный синтез (SyGuS) [22].

Задача, сформулированная в таком виде, включает в себя три вида ограничений.

- Спецификация ϕ искомой функции f задается в виде логической формулы, в которой все переменные универсально квантифицированы;

- логические символы и их интерпретации ограничиваются лежащими в их основе теориями T;

- пространство возможных реализаций для искомых функций описывается в виде построенного на основе правил грамматики G множества выражений L, интерпретация выражений в котором подчиняется теории T.

Задача синтаксически-направленного синтеза заключается в поиске выражения $f_{implementation} \in L$, такого, что $\phi[f/f_{implementation}]$ станет выполнимой по модулю теории T, где $\phi[f/f_{implementation}]$ получается заменой каждого вхождения функции f на $f_{implementation}$ в спецификации ϕ.

Приведем пример. Рассмотрим логику первого порядка над теорией линейной арифметики (LIA), функцию f (см. 1a) и спецификацию ϕ (см. 1b).

$$f(int\ x,\ int\ y) : int \tag{1a}$$

$$(x \leq f(x,\ y)) \wedge (y \leq f(x,\ y)) \wedge (f(x,\ y) = x \vee f(x,\ y) = y) \tag{1b}$$

Задача в формате SyGuS представлена в Листинге 2.

```
1  (set-logic LIA)
2
3  (synth-fun f ((x Int) (y Int)) Int
4
5  ((Start Int (x y 0 1
6      (+ Start Start)
7      (- Start Start)
8      (ite StartBool Start Start)))
9
10  (StartBool Bool ((and StartBool StartBool)
11     (or StartBool StartBool)
12     (not StartBool)
13     (<= Start Start)
14     (= Start Start)
15     (>= Start Start)))))
16
17 (declare-var x Int)
18 (declare-var y Int)
19
20 (constraint (>= (f x y) x))
21 (constraint (>= (f x y) y))
22 (constraint (or (= x (f x y)) (= y (f x y))))
23
```

```
24 (check-synth)
```

Листинг 2: Задача синтеза в формате SyGuS.

Здесь первая строчка задает теорию T (теория задает множество символов, используемых для построения формул, набор значений для каждого типа и интерпретацию предикатных и функциональных символов, подробное описание представлено, например, в книге [35]). Формат поддерживает теорию линейной арифметики, теорию бит-векторов, теорию вещественных чисел, массивов, и теорию строк (начиная с 2016 года). Третья строка ставит задачу синтеза функции с именем f и определяет типы переменных и возвращаемого значения. Строки 5-16 описывают грамматику допустимых выражений G для поиска функции $f_{implementation}$. Нетерминал `Start` (строка 5) является обязательным и его тип должен совпадать с типом синтезируемой функции f. Строки 17-18 вводят универсально квантифицированные переменные `x`, `y`. Строки 20-22 описывают спецификацию ϕ. Наконец, строка 24 начинает процесс синтеза. Полная спецификация входного формата представлена в [29, 62], синтаксис формата основан на входном формате для SMT-решателей `SMT-LIB2` [63].

3.4. Синтез программ по трассам

Существуют различные способы задать ограничения: полная формальная спецификация (заданная в какой либо логике), примеры, логи (трассы) программ. Ограничения, заданные трассами, предоставляют более детальное описание того, как программа должна себя вести на входных примерах, так как описывают, *как* различные конкретные входы программы трансформируются в выходные (в отличие от спецификаций заданных лишь примерами входных и выходных данных).

Синтез программ по трассам показывает неплохой результат, один из первых фреймворков для синтеза [64] строил по входным трассам высокоуровневую спецификацию с последующей генерацией кода.

Современные подходы объединяют различные техники для получения информации о программах по их трассам. К примеру, используя

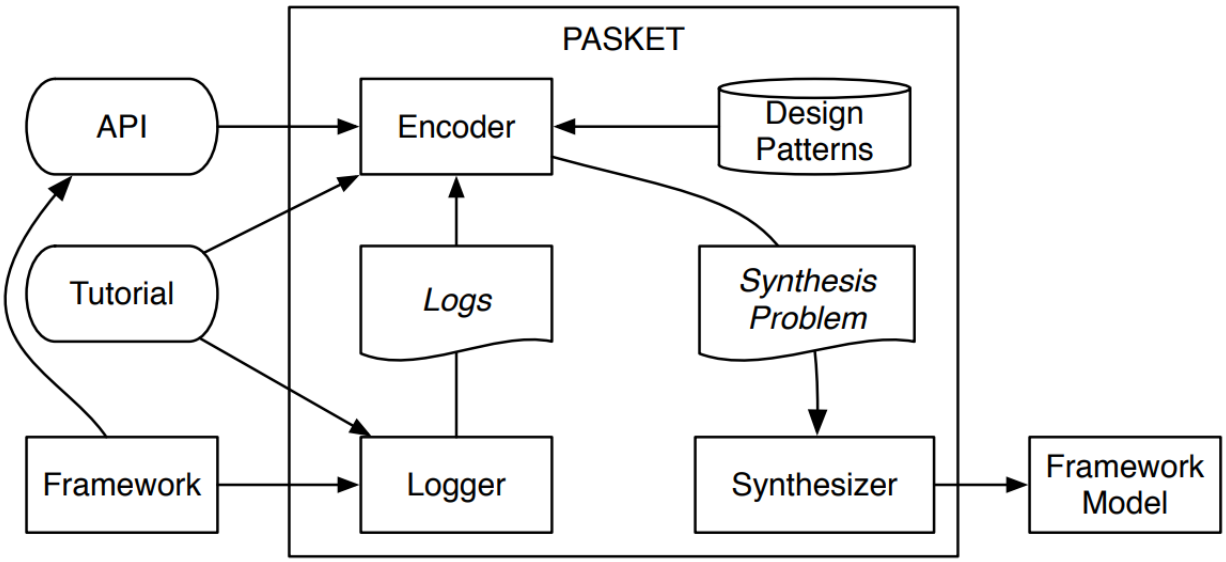


Рис. 2: Архитектура инструмента Pasket. Рисунок взят из [15].

алгебру пространства версий и трассы программ для конкретных входных параметров и машинное обучение, в статье [65] авторы создали фреймворк, который обучается по императивному языку (подмножество языка Python).

Одна из последних работ [15], относящаяся к синтезу программ по трассам, описывает инструмент Pasket. Pasket позволяет автоматически генерировать модель фреймворков для Java, чтобы улучшить возможности символьного исполнения, так как фреймворки являются «черным ящиком» для символьных исполнителей (к примеру, Java PathFinder (JPF) имеет собственную модель фреймворка Java Swing). Процесс синтеза происходит на основе предположения, что фреймворки проектируются с использованием большого количества шаблонов проектирования [66], с использованием трасс программ, информации об программном инферфейсе фреймворка и инструмента для синтеза фрагментов кода Sketch (см. рис. 2).

4. Метод синтеза моделей императивных функций

Предлагаемый в работе метод синтеза моделей императивных программ по трассам их исполнения путем решения проблемы SyGuS (см. 3.3) разбивается на последовательность из пяти основных шагов.

1. Сбор трасс программ и построение по ним дерева вызовов. Анализ дерева вызовов.
2. Восстановление потока управления функций. Восстановление выполняется для каждой функции, внутри которой присутствуют другие вызовы.
3. Построение спецификаций, задающих семантические ограничения на каждую функцию, и формирование задач для SyGuS-решателей.
4. Решение сформированных задач SyGuS-решателем.
5. Генерация по решенным задачам кода на целевом языке.

В этой главе будут подробно рассматриваться первые три шага метода.

4.1. Императивные программы

Функции в императивной парадигме программирования — это набор изменяющих состояние программы различных инструкций, связанных потоком управления. В общем случае функции могут принимать набор аргументов, изменять состояние программы (память), возвращать значения.

В статически типизируемых языках программирования функциям сопоставляется описание типов аргументов и тип ее возвращаемого значения. Сопоставим функциям дополнительно типы состояния, от которого она может зависеть:

$$\texttt{function} : \overline{\texttt{arguments}} \times \overline{\texttt{state}} \to \texttt{result} \times \overline{\texttt{state}} \tag{2}$$

Здесь `result` — тип возвращаемого значения, $\overline{\texttt{arguments}}$ — типы аргументов функции, $\overline{\texttt{state}}$ — типы внешнего (общего) состояния программы, от которого зависит результат выполнения функции, и на которое ее выполнение может влиять.

Выполнение императивной программы, состоящей из таких функций, можно рассматривать как последовательность применения функций к некоторому начальному состоянию. Рассмотрим пример. Пусть в программе есть две функции — `write`, `read`, где функция `write` дописывает строки в буфер, а `read` — возвращает текущее значение буфера. Рассмотрим следующий фрагмент кода (см. Листинг 3).

```
1 write("hello␣");
2 read();           // "hello "
3 write("world");
4 read();           // "hello world"
```

Листинг 3: Фрагмент кода императивной программы.

Состоянием программы, на которое влияет функция `write`, и от которого зависит функция `read`, является строка-буфер, назовем ее `stateS`. Большинство формализмов, на которых построены современные синтезаторы, не позволяет рассуждать о функциях, изменяющих состояние. Зная состояние программы и явно расширяя им функции, можно записать ту же последовательность в эквивалентном виде (см. Листинг 4).

```
1 write("hello␣", stateS); //  void         , stateS = "hello "
2 read(stateS);            // "hello "      , stateS = "hello "
3 write("world", stateS);  //  void         , stateS = "hello world"
4 read(stateS);            // "hello world" , stateS = "hello world"
```

Листинг 4: Фрагмент кода императивной программы.

Такая запись позволяет абстрагироваться от неявных состояний программы, так как они все могут быть записаны явно в сигнатурах функций, тем самым позволяя рассматривать побочные эффекты от императивных функций явно, не предполагая зависимость от внешнего окружения.

4.2. Преобразование состояний

Способа, продемонстрированного в Листинге 4, уже достаточно для сведения проблемы синтеза императивных функций к задаче SyGuS, однако такой подход потребует синтезировать все функции одновременно: то, как влияют функции друг на друга и их возвращаемое значение, будет задаваться в одной задаче с большим количеством ограничений. Более того, так как у каждой функции состояние может отличаться от других, придется рассматривать очень большое состояние, являющееся объединением состояний всех функций. Наличие таких неявных взаимных ограничений на состояние большого размера делает процедуру синтеза крайне непрактичной.

В рамках работы предлагается способ преобразования вида императивных функций в формат, который можно передать решателям задачи SyGuS, и позволяющий разбить процесс синтеза на отдельные подзадачи.

Рассмотрим выполнение любой функции иначе. Разделим ее выполнение на две части: вычисление возвращаемого значения и изменение состояния программы. Введем для каждой функции набор *функций-эффектов*, которые будут описывать ее влияние на состояние всех функций, на которые она может как-то влиять (вид состояния каждой функции и список функций, на которые функция влияет будет рассмотрен позже в разделе 4.5). Вернемся к примеру с функциями `write` и `read` (см. Раздел 4.1). Функция `write` не имеет возвращаемого значения (тип `void`), но влияет на состояние программы, функция-эффект функции `write` на функцию `read` изменяет состояние `stateS`, конкатенируя буфер-строку с аргументом функции `write`.

Опишем сигнатуры функций-эффектов. Допустим, что требуется синтезировать `N` функций $\texttt{function}_i$, которые разделяют состояние программы и тем самым влияют на выполнение друг друга. Свяжем с каждой функцией дополнительно `N` функций $\texttt{functionEffect}_i^j$. Эти функции описывают влияние от исполнения i-й функции на состояние, от которого зависит функция $\texttt{function}_j$. Введем общий вид функций-эффектов:

$$\texttt{functionEffect}_i^j : \overline{\texttt{arguments}_i} \times \texttt{result}_i \times \overline{\texttt{state}_j} \to \overline{\texttt{state}_j} \quad (3)$$

Здесь $\overline{\texttt{arguments}_i}$ — это типы аргументов функции $\texttt{function}_i$, $\texttt{result}_i$ — тип ее результата, $\overline{\texttt{state}_j}$ — типы состояния, от которого зависит функция $\texttt{function}_j$. Таким образом, функции-эффекты — это функции, зависящие от аргументов и результата вычисления основной функции, которые отражают изменение состояния других функций, в результате своего выполнения. Разделив выполнение функций на вычисление результата и эффекты, вместо синтеза всех функций вместе, можно синтезировать каждую функцию отдельно с учетом функций-эффектов других функций на нее. Очевидно, что количество функций синтезируемых одновременно при таком подходе не уменьшается, но благодаря тому, что состояние рассматривается только для одной функции, общая задача становится легче.

Введя такие функции и разделив функцию на вычисление результата и функции-эффекты, запишем применение каждой функции в любой момент времени в виде вложенных применений функций-эффектов к состоянию:

$$\texttt{function}_j(\overline{\texttt{arguments}_j},\ \texttt{F}_k^j(..\texttt{F}_1^j(\overline{\texttt{state}})..)) = \texttt{result} \quad (4)$$

Здесь $\texttt{F}_i^j$ — сокращенная запись для

$$\texttt{functionEffect}_i^j(\overline{\texttt{arguments}_i},\ \texttt{result}_i,\ \overline{\texttt{state}_j})$$

4.2.1. Порядок применения функций-эффектов

Рассмотрим дерево вызовов, в котором присутствуют функции, содержащие внутренние вызовы (см. Рис 3).

Допустим, что все функции влияют друг на друга. Тогда запись применения функции будет включать в себя все применения функций-эффектов от соответствующих функций, которые завершились до начала ее исполнения.

Отметим, что рассмотрение всех функций-эффектов от внутренних

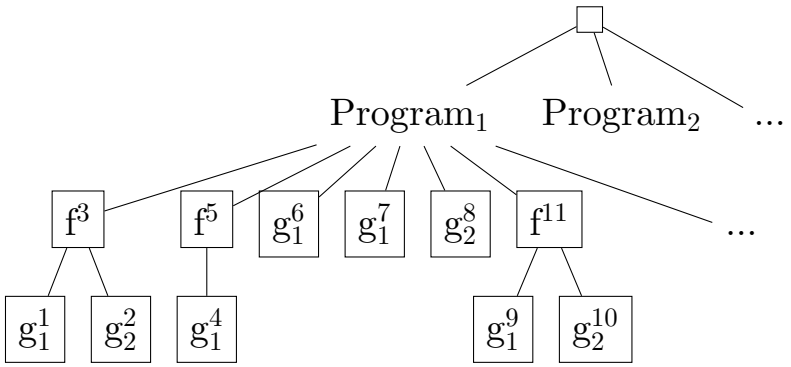


Рис. 3: Дерево вызовов для построения ограничений (верхний индекс — номер окончания вызова).

функций сильно увеличивает количество ограничений. Можно было бы рассматривать внутренние вызовы функций как «чистые» (без побочных эффектов, не меняющие состояния) вызовы с предположением, что эффект от них будет отражен в эффекте функции «родителя». Но такой подход привел бы к необходимости неоднократного синтезирования одной и той же функции, если бы ее вызов встречался в разных функциях.

4.3. Кодирование функций-эффектов в формате SyGuS

Синтаксис SyGuS позволяет задавать спецификацию в виде ограничений, представляющих собой логические формулы. Рассмотрим функцию `f`, имеющуя сигнатуру `f : int` $\rightarrow$ `int` (состояние отсутствует, функция принимает один аргумент). Пусть известно, что функции вызывалась два раза с аргументами `3, 4` и возвращала `1, 2` соответственно. Тогда ограничения на функцию `f` можно записать в виде набора из двух ограничений следующим образом:

```
(constraint (= (f 3) 1))
(constraint (= (f 4) 2))
```

Однако такая запись не подходит для функций-эффектов, так как достаточно сложно, используя синтаксис задания проблемы SyGuS, задать тип возвращаемого значения функции в виде кортежа значений. Для решения этой проблемы запишем сигнатуру для функций-эффектов

(см. (3)) в виде отношений:

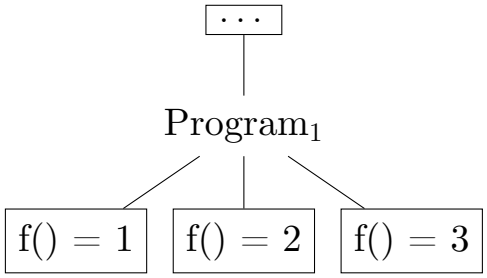
$$\texttt{functionEffect}_i^j : \overline{\texttt{arguments}_i} \times \texttt{result}_i \times \overline{\texttt{state}_j} \times \overline{\texttt{state}_j} \to \texttt{Bool} \quad (5)$$

Рассмотрим, как выглядит кодировка для вложенных эффектов на примере влияния функции на себя.

Допустим, что у нас есть ровно одна, неизвестная нам функция `f`, зависящая от глобального состояния (см. Листинг 5).

```
int f()
{
    int result = global_N;
    global_N = global_N + 1;
    return result;
}
```

Листинг 5: Пример зависящей от состояния простой функции.



Пусть известно, что было три ее вызова подряд, с результатами `0`, `1`, `2` соответственно, и что функция зависит от состояния, состоящего из одного целочисленного значения, проинициализированного нулем. На Листинге 6) приведена кодировка в формате SyGuS (описание кодировки подробно разобрано в Разделе 3.3), описывающая синтезируемые функции вместе с ограничениями на них, построенными для функции `f` (в кодировке для краткости намеренно уменьшены грамматики, описывающие пространство поиска функций).

```
(set-logic LIA)

(synth-fun functionEffect_f^f ((result Int) (st Int) (nSt Int)) Bool
 (
   (Start Bool ((= nSt NtInt)))
   (NtInt Int (st result (Constant Int) (+ NtInt NtInt) (- NtInt NtInt)))
 )
)

(synth-fun f ((st Int)) Int
 (
   (Start Int (NtInt))
   (NtInt Int (st (Constant Int) (+ NtInt NtInt) (- NtInt NtInt)))
 )
)
```

```
(declare-var st0 Int)
(declare-var st1 Int)
(declare-var st2 Int)

(constraint
 (
  and
  (=> (= st0 0) (= (f st0) 1))
  (=> (and (= st0 0) (functionEffect_f^f 1 st0 st1)) (= (f st1) 1))
  (=> (and (= st0 0) (functionEffect_f^f 1 st0 st1) (functionEffect_f^f 2 st1 st2))
      (= (f st2) 3))
 )
)

(check-synth)
```

Листинг 6: Задача SyGuS для функции `f` с заданными ограничениями.

4.3.1. Синтаксические ограничения

В Листинге 6 синтаксические ограничения, задаваемые грамматикой, намеренно были сокращены для простоты демонстрации.

В общем случае, в работе для каждой функции рассматривается грамматика из трех основных нетерминалов, задающих правила для построения выражений типа `int, string, bool`. Их общий вид представлен в Листинге 7.

```
(NtString String (
 ""
 " "
 (Variable String)
 (str.++ NtString NtString)
 (str.replace NtString NtString NtString)
 (str.at NtString NtInt)
 (int.to.str NtInt)
 (str.substr NtString NtInt NtInt)
 (ite NtBool NtString NtString)
))

(NtInt Int (
 (Constant Int)
 (Variable Int)
 (+ NtInt NtInt)
 (- NtInt NtInt)
 (str.len NtString)
 (str.to.int NtString)
 (str.indexof NtString NtString NtInt)
 (ite NtBool NtInt NtInt)
))

(NtBool Bool (
 true
 false
 (Variable Bool)
 (str.prefixof NtString NtString)
 (str.suffixof NtString NtString)
 (str.contains NtString NtString)
 (= NtString NtString)
 (< NtInt NtInt)
```

```
 (= NtInt NtInt)
 (<= NtInt NtInt)
 (>= NtInt NtInt)
 (> NtInt NtInt)
 (and NtBool NtBool)
 (or NtBool NtBool)
 (not NtBool)
))
```

Листинг 7: Грамматика, задающая синтаксические ограничения.

В зависимости от того, содержит ли сигнатура функции или функции-эффекта аргументы каждого типа, грамматики могут быть уменьшены. К примеру, если в теле функции не встречаются строковые значения (будь то аргумент, результат или часть состояния) все функциональные и предикатные символы работающие со строками, как и сам нетерминал, задающий правила их построения, заданы не будут. Несмотря на кажущуюся простоту такой эвристики, ее влияние весьма критично, так как ввиду несовершенства решателей задачи SyGuS даже добавление одного терминального символа может увеличить время поиска реализации функции в два раза.

4.4. Поток управления

Дерево (см. Рис. 4), построенное по трассам, для каждого вызова задает частичную структуру потока управления искомой реализации функции.

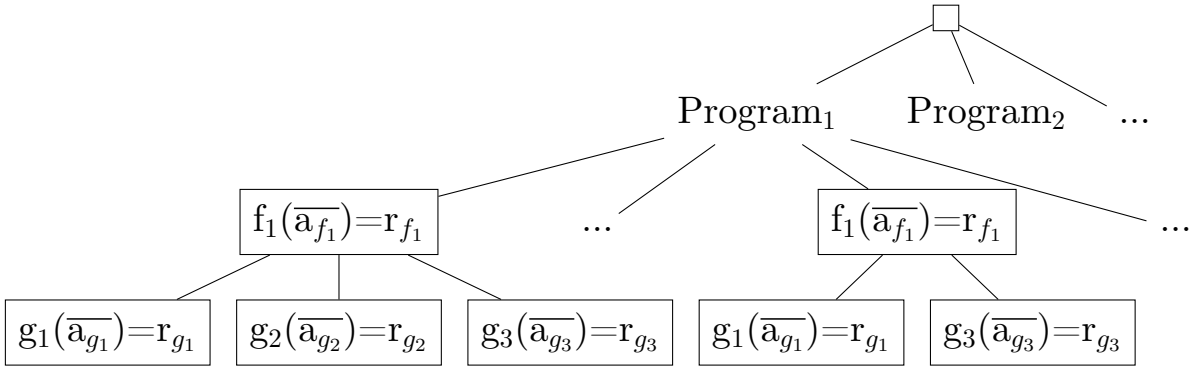


Рис. 4: Дерево вызовов, построенное по трассам (в скобках содержится список аргументов, r_f — результат функции `f`).

Так, например, зная, что для одного и того же вызова функции `f`($\overline{\texttt{arguments}}$) вызывались частично совпадающие наборы вызовов функ-

ций, допустим g_1, g_2, g_4 в первом случае, и g_1, g_3, g_4 во втором, g_1, g_2, g_3, g_4 в третьем соответственно, можно построить частичную программу в виде следующей последовательности операторов (см. Листинг 8).

```
1  Type f(arguments, state) {
2    Type localResult_1;
3    Type localResult_2;
4    Type localResult_3;
5    Type localResult_4;
6
7    localResult_1 = g_1([| g_1ArgHole_1 |], ...);
8    if ([| controlFlowHole_1 |])
9    {
10     localResult_2 = g_2([| g_2ArgHole_1 |], ...);
11   }
12   if ([| controlFlowHole_2 |])
13   {
14     localResult_3 = g_3([| g_3ArgHole_1 |], ...);
15   }
16   localResult_4 = g_4([| g_4ArgHole_1 |], ...);
17
18   return [| f(arguments, results, state) |];
19 }
```

Листинг 8: Пример частичной программы.

В истории вызовов есть информация о значении всех переменных $\texttt{localResult}_i$, о значении аргументов, которые были переданы в функции g_i. Однако условия внутри конструкций ветвления, накладывающие ограничения на вызовы функций g_i, и то, как зависят аргументы этих функций от аргументов искомой функции, внутри которой они были вызваны — неизвестны.

В общем случае, каждое выражение, обособленное в виде `[|name|]`, представляет собой отдельную задачу синтеза, целью которой является поиск взаимосвязей между всей доступной информацией, полученной ранее, и информацией, которая используется в данный момент. Например, процесс вычисления аргументов $g_1\texttt{ArgHole}_i$ (строка 7) может зависеть только от аргументов и состояния, на которых была вызвана функция `f`. Условие $\texttt{controlFlowHole}_1$ (строка 8) для вызова функции g_2 — от аргументов функции `f`, от результата $\texttt{localResult}_1$ и от состояния функции `f`, однако в этой точке вычисления состояние функции `f` могло быть изменено функцией g_1, и это должно быть учтено в ограничениях. Для синтеза выражений вычисления аргументов $g_3\texttt{ArgHole}_i$ (строка 14) доступно больше информации, к примеру, следует дополнительно учесть факт наличия или отсутствия вычислений $\texttt{localResult}_2$. Похожим образом ставится задача синтеза для строки 18, где требует-

ся найти выражение для вычисления возвращаемого значения искомой функции: предполагается, что возвращаемое значение зависит только от аргументов функции, от набора пар вида результат × факт вычисления, несущего информацию о внутренних вычислениях, и от состояния функции, полученного после вычисления всех вложенных функций. Важно отметить, что задачи синтеза для аргументов и флагов условных конструкций могут решаться независимо.

Рассмотрим в качестве примера функцию `f` (см. Листинг 9) с соответствующим ей деревом вызовов (см. Рис. 5).

```
int f(int arg)
{
  int result = 0;
  result += g1();
  if (arg > 0)
    result += g2(arg * 2);
  return result;
}
```

Листинг 9: Пример функции с условным переходом.

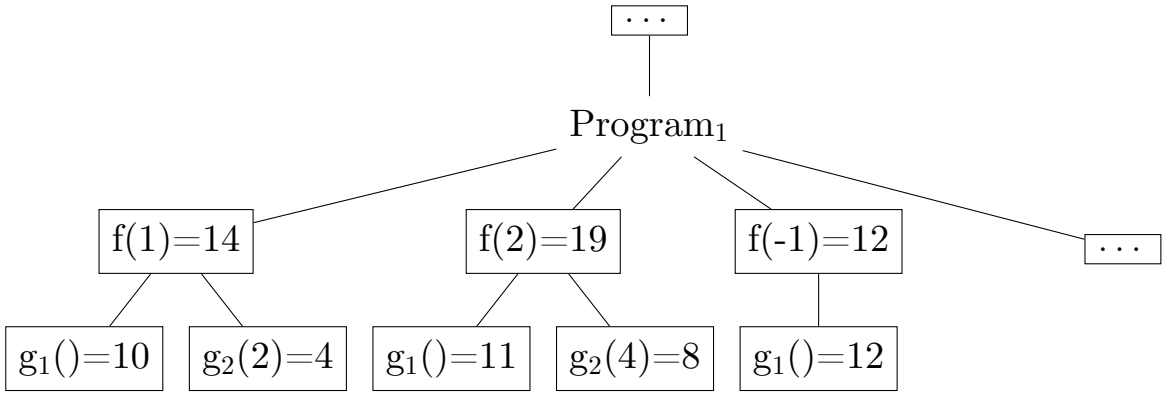


Рис. 5: Дерево вызовов для функции `f`.

Версия потока управления в данном случае совпадет с оригинальной функцией (см. Листинг 10).

```
int f(int arg, state_f)
{
  int localResult1, localResult2;
  localResult1 = g1(state_g1);
  if ([| controlFlowHole |])
    localResult2 = g2([| g2ArgHole |], state_g2);

  int result = [| f(arg, localResult1, localResult2, state_f) |];
  return result;
```

```
}
```

Листинг 10: Восстановленный поток управления для функции `f`.

Одно из решений задачи синтеза для `[| g2ArgHole |]`, полученное от решателей, — `arg + arg`. Для задачи `[| controlFlowHole |]` — `arg != -1`. Отметим, что хотя функция не соответствует изначальной «неизвестной» реализации, для ограничений, построенных по примерам, задача решена верно.

4.5. Оракулы зависимостей и состояния

Подход, описанный выше, выделяет общие шаги, необходимые для синтеза моделей функций по трассам их исполнения. Предложенный подход построен на том, что известно, как зависят функции друг от друга и о том, что является их состоянием. Из трасс программ нельзя «точно» заключить что-либо о зависимостях и состояниях функций.

За разрешение этих неопределенностей в предлагаемом подходе отвечают специальные подсистемы — *оракулы* поиска зависимостей и состояний. В качестве оракулов могут выступать различные системы анализа данных, пользователь или различные эвристики.

4.5.1. Поиск зависимостей

Как говорилось ранее (см. раздел 4.1), функции могут разделять общее состояние и влиять на выполнение друг друга. В общем случае, можно считать, что все функции влияют друг на друга. Однако, такое предположение сильно увеличивает время работы решателей задач синтеза, так как количество функций-эффектов, которое необходимо будет синтезировать решателю, для каждой функции будет равно общему количеству всех искомых функций.

В случае, когда не все функции влияют друг на друга, решатели все еще могут искать все функции-эффекты от других функций на каждую функцию (тривиальный оракул). Когда решение будет найдено, часть реализаций не влияющих функций будет содержать тривиальные (пустые) реализации. Проблема состоит в том, что в таком случае

сильно увеличивается время работы решателей в связи с тем, что решателям заранее неизвестно, что реализация будет пустой, при этом каждая функция-эффект сильно увеличивает пространство поиска, добавляя большое количество новых аргументов и требующая перебора всех взаимосвязей, указанных во вложенных ограничениях. Поэтому отсечение лишних зависимостей между функциями является довольно важным шагом.

Извлечение таких зависимостей из трасс программ — трудная задача. Однако для библиотек, предоставляющих функции для работы, например, с файлами, сокетами, графическими окнами, такая задача может быть решена проще. Функции, разделяющие внутреннее состояние в таких библиотеках, чаще всего содержат в себе *аргументы-идентификаторы*: например, файловые дескрипторы, идентификаторы окон или адрес объекта и так далее. С большой долей вероятности зависящие друг от друга функции будут манипулировать одной и той же сущностью в рамках трассы одной программы, поэтому часть их аргументов в истории вызовов совпадет.

Исходя из этого, имея построенное по трассам дерево вызовов, можно алгоритмически, без вмешательства пользователя, отсечь для каждой конкретной функции заведомо не влияющие на ее выполнение функции. К примеру, проследив зависимость по совпадающим аргументам или результатам применения функций (см. Рис. 6), можно предположить, что не все функции влияют друг на друга, отбросив часть лишних зависимостей и тем самым упрощая построение семантических ограничений (см. Раздел 4.2). Подход с легкостью допускает введение других стратегий поиска зависимостей.

4.5.2. Состояние функции

Весь описанный ранее алгоритм построен в предположении того, что состояние каждой функции известно. Очевидно, что трассы программ не содержат информации о его виде, поэтому одной из важнейших частей алгоритма является оракул, делающий предположение о состоянии функций. Задача облегчается тем, что процедура синтеза требует нали-

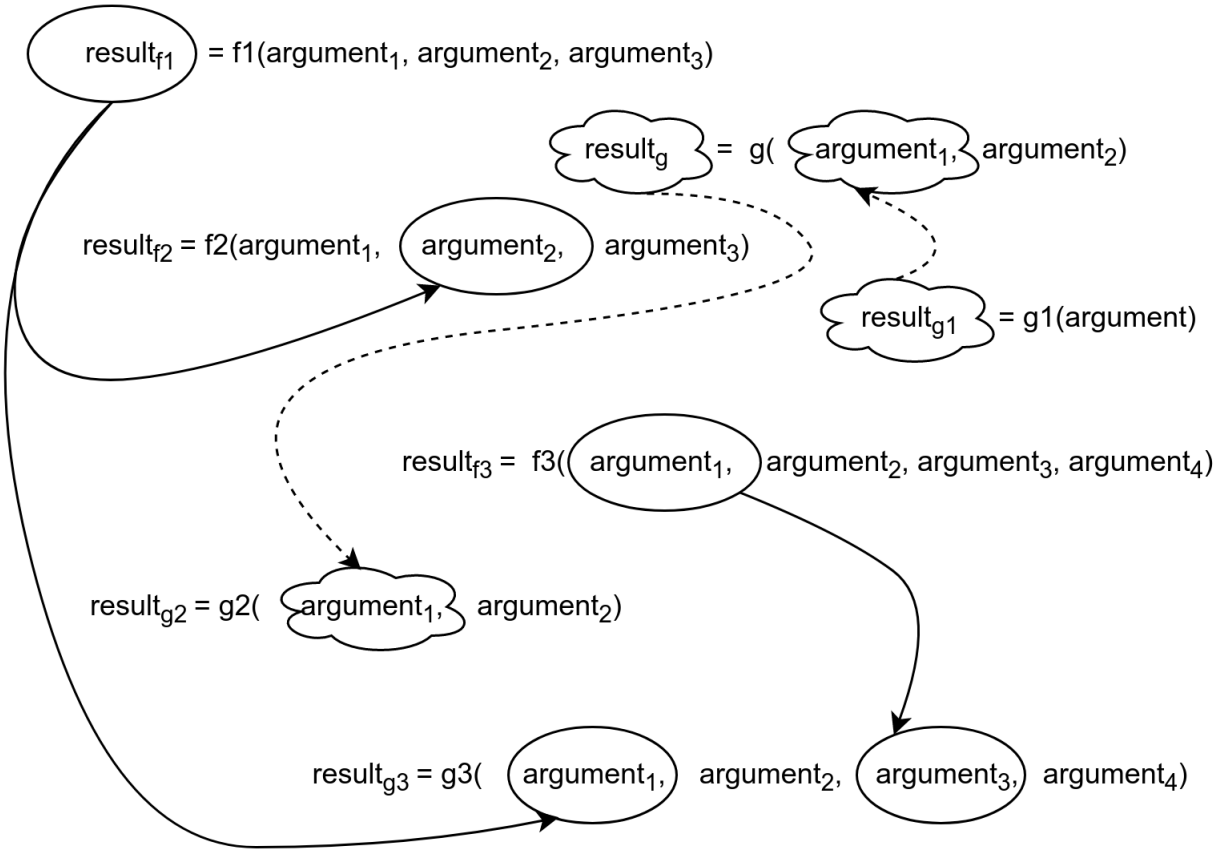


Рис. 6: Поиск зависимых функций.

чия лишь типов элементов состояния. Различные варианты начальных значений и изменений состояний будут перебираться решателями задач синтеза.

На основании информации о сигнатуре функции, полученной из трасс, и ее зависимостях можно построить множество оракулов, предсказывающих кортеж из типов переменных, от которых может зависеть ее исполнение. За поиск начальных значений, которыми будут проинициализированы состояния, будет отвечать SyGuS-решатель при синтезе каждой конкретной функции. В работе предложено пять типов оракулов.

Первый, самый простой оракул, возвращает кортеж из типов, полученных из сигнатуры функции, и никак не учитывает информацию о зависимостях между функциями.

Второй оракул для каждой функции пересекает кортеж ее типов, полученный от первого оракула, с кортежами всех функций, которые найдены для нее в качестве зависимостей, и выбирает из пересечений

максимальный по длине. Отметим, что общем случае, функция может зависеть сама от себя, поэтому этот оракул не рассматривает пересечение функции с собой, иначе результат работы второго оракула совпадет с первым.

Третий отличается от второго тем, что не ищет максимальный по длине кортеж из попарных пересечений, рассмотренных во втором оракуле, а объединяет попарные пересечения. Под объединением понимается выбор максимального количества элементов каждого типа. К примеру, если кортеж первого пересечения состоит из {`bool, int`}, второго — из {`int, int`}, то объединением будет кортеж {`bool, int, int`}.

Четвертый оракул предоставляет самые большие кортежи типов, объединяя все кортежи зависимых функций.

Наконец, пятый оракул считает количество вхождений каждого примитивного типа в объединение сигнатур всех зависимостей и возвращает состояние, количество элементов которого равно количеству аргументов функции плюс один (под результат), распределяя каждый примитивный тип пропорционально его доле в общем состоянии.

Для более точных предсказаний оракулы могут быть заменены. Например, в качестве оракула может выступать система анализаторов исходного кода библиотек, для которых синтезируются символьные модели. В данной работе мы ограничимся рассмотрением вышеописанных оракулов.

4.6. Общий вид синтезируемых функций

Предъявив способ задания состояния функций, разделив функцию на функцию вычисляющую возвращаемое значения и на набор функций-эффектов и предложив способ восстановления потока управления для них, опишем общий вид кода модели функции (см. Листинг 11).

```
1 // state_CF = { CurrentFunctionState_1, ..., CurrentFunctionState_n }
2 Type CurrentFunctionState_1 = [|CurrentFunctionInitializer_1|];
3 ...
4 Type CurrentFunctionState_n = [|CurrentFunctionInitializer_n|];
5
6 // state_Function_i = {FunctionState_1^i, ..., FunctionState_j^i}
7 ...
```

```
8  Type FunctionState_n^N = Function_N Initializer_n;
9
10 void ApplyEffectOnFunction_1(arguments, localResults, result)
11 { [|ApplyEffectHole_1|] }
12 ...
13 void ApplyEffectOnFunction_N(arguments, localResults, result)
14 { [|ApplyEffectHole_N|] }
15
16
17 Type CurrentFunction(arguments) {
18   // results = {res1, ..., resN}
19   Type localResult_1;
20   ...
21   Type localResult_N;
22
23   if ([| FunctionControlFlowHole_1 |])
24   {
25     localResult_1 = F_1([| F_1ArgHole_1 |], ...);
26     ...
27     localResult_K = F_K([| F_KArgHole_1 |], ...);
28   }
29   localResult_L = F_L([| F_LArgHole_1 |], ...);
30   ...
31   localResult_M = F_M([| F_MArgHole_1 |], ...);
32   if ([| FunctionControlFlowHole_1 |])
33   {
34     localResult_N = F_N([| F_NArgHole_1 |], ...);
35   }
36
37   Type Result = [|Function(arguments, results, state_CF)|];
38
39   ApplyEffectOnFunction_1(arguments, results, Result);
40   ...
41   ApplyEffectOnFunction_N(arguments, results, Result);
42
43   return Result;
44 }
```

Листинг 11: Шаблон синтезируемых функций.

Строки 2-4 описывают состояние, от которого зависит синтезируемая функция. Начальные значения элементов состояния ищутся решателями совместно с поиском реализации функции. Сама функция также может изменять состояние (строки 6-8), от которого зависит работа других функций. Эффект изменения состояния для этих функций от работы функции `CurrentFunction` в общем случае задается набором из `N` функций-эффектов (строки 10-14), где `N` — число всех синтезируемых функций. Все функции-эффекты вызываются в конце исполнения синтезируемой функции (строки 39-41).

В конечном итоге все синтезированные реализации (модели) функций объединяются в один файл, содержащий все глобальные перемен-

ные, описывающие состояния, $\mathtt{N} \times \mathtt{N}$ функций-эффектов в общем случае и $\mathtt{N}$ целевых функций.

5. Инструмент

Общий вид реализованного в работе прототипа инструмента представлен на рисунке 7. Весь процесс синтеза делится на два шага.

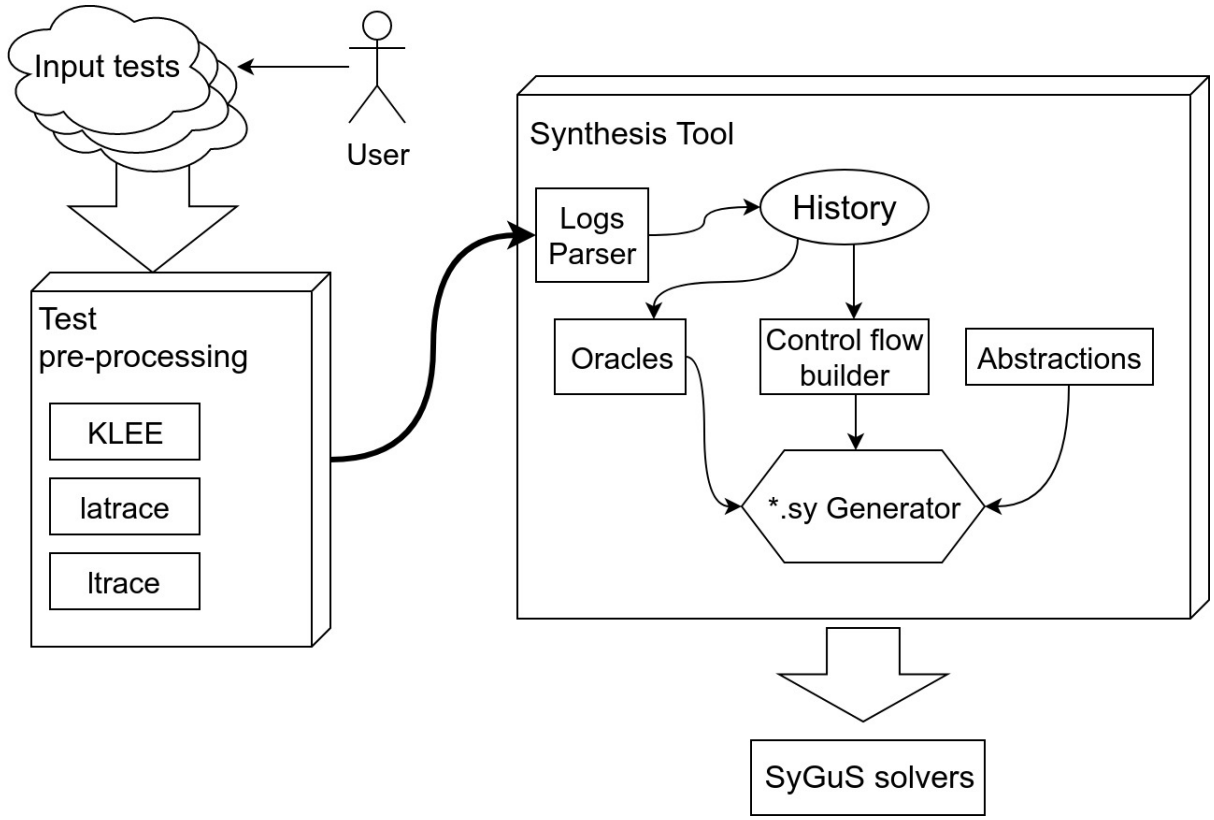


Рис. 7: Общая архитектура.

1. Сбор пользователем примеров, содержащих внешние вызовы, которые не могут быть исполнены символьной виртуальной машиной KLEE, с последующей трассировкой примеров с помощью трассирующих системных утилит.

2. Построение по трассам задач для синтаксически-направленного синтеза с последующим запуском решателей синтеза по ним.

5.1. Сборка трасс программ

На первом этапе собирается набор примеров на языке `C`, в которых происходят различные системные вызовы (например, работа с файловой системой с помощью системных вызовов `open`, `write` и других) или

вызовы внешних функций из подключаемых заголовочных файлов, таких как `stdio.h`. Пример тестового файла представлен в Листинге 12.

```
#include <stdio.h>
#include <fcntl.h>

void main()
{
  int fd = open("foo.txt", O_WRONLY | O_CREAT, 0777);
  write(fd, "hello", strlen("hello"));
  close(fd);
}
```

Листинг 12: Пример кода для сбора трасс.

Собранный набор тестов компилируются в llvm-файлы и исполняется с помощью символьной виртуальной машины KLEE. KLEE выдает список внешних, недостижимых для символьного исполнения функций (см. 13).

```
WARNING: undefined reference to function: open
WARNING: undefined reference to function: write
WARNING: undefined reference to function: close
KLEE: WARNING ONCE: calling external: open(24862928, 65, 511)
KLEE: WARNING ONCE: calling external: write(8, 24850048, 5)
KLEE: WARNING ONCE: calling external: close(8)
```

Листинг 13: Внешние вызовы, возвращаемые KLEE.

Для получения информации о системных вызовах, аргументах и результатах функций исполняемые файлы трассируются с помощью системных утилит `ltrace` и `latrace`. К примеру, для Листинга 12 часть трассы программы, полученной с помощью утилиты `ltrace`, отображена на Листинге 14.

```
__libc_start_main(0x4005c0,1,0x7ffcbdd36488,0x400600 <unfinished ...>
open("foo.txt", 65, 0777 <unfinished ...>
SYS_open("foo.txt", 65, 0777)                    = 3
<... open resumed> )                             = 3
write(3, "hello", 5 <unfinished ...>
SYS_write(3, "hello", 5)                         = 5
<... write resumed> )                            = 5
close(3 <unfinished ...>
SYS_close(3)                                     = 0
<... close resumed> )
+++ exited (status 0) +++
```

Листинг 14: Частичная трасса программы.

В конечном итоге, собранные трассы группируются в общих файлах и передаются на вход инструменту генерации задач для синтеза.

5.2. Формирование и запуск задач синтеза

После исполнения программ их трассы, сгруппированные по файлам, передаются инструменту, отвечающему за формирование задач в формате SyGuS. Здесь по трассам строятся деревья вызовов, к примеру, для функций из заголовочного файла `stdio.h` узлами верхнего уровня будут являться внешние библиотечные вызовы, а листьями — — различные системные вызовы (см. Рис 8).

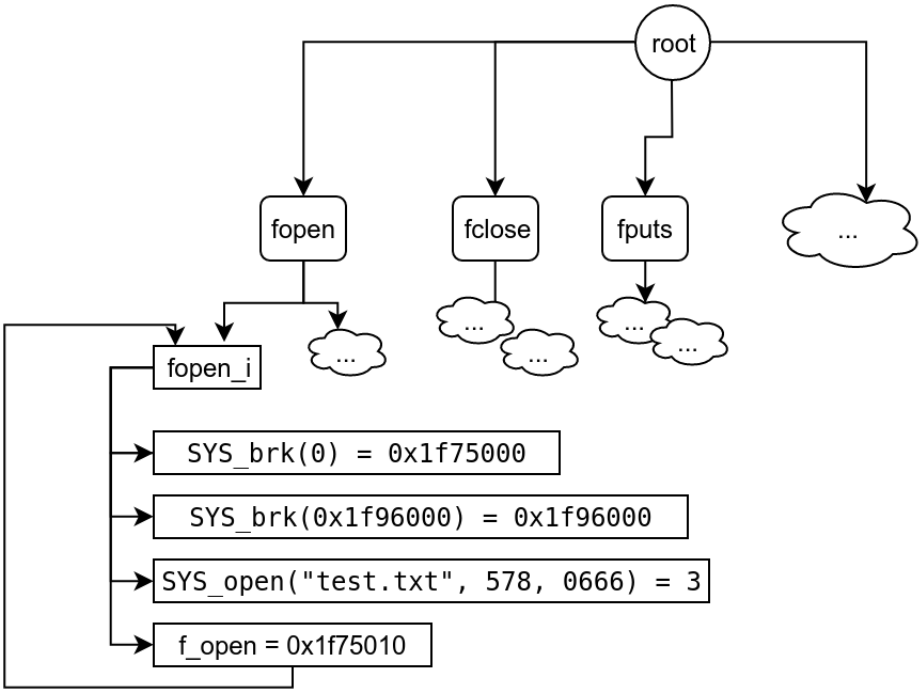


Рис. 8: Пример дерева вызовов для функций из `stdio.h`.

На основании описывающего историю вызовов дерева формируются оракулы состояний и зависимостей (если они не были заданы пользователем), производится «адаптация» сигнатур функций к формату SyGuS (см. 5.3), а также восстанавливается поток управления (см. 5.4). При восстановлении потока управления возникает множество задач синтеза для аргументов внутренних функций и условий для конструкций ветвления. Эти задачи добавляются к основной и зависят от результата решения задачи синтеза для нее. Как только будет решена основная задача, решателям независимо друг от друга отправляются запросы на поиск решения для зависимых задач. Отметим, что в описанном ранее алгоритме (см. 4.4), не предполагается вложенных зависимостей,

другими словами, вложенные задачи будут возникать только при восстановлении потока управления для функций, и следовательно глубина вложенности всегда будет равна единице. Стоит также отметить, что на этом этапе пользователь может «подсказать» решение для основной задачи, если ее не найдут решатели за заданное пользователем время.

Информацию о том, как зависят задачи синтеза друг от друга и о том, как выглядит шаблон синтезируемой функции, сохраняется на диск, чтобы в дальнейшем иметь возможность оттранслировать результаты синтеза в формате `SyGuS-IF` в код на целевом языке (к примеру, на языке `C`).

5.3. Адаптация сигнатур синтезируемых функций

В целях упрощения процесса синтеза сигнатуры искомых функций частично изменяются. Во-первых, в связи с нехваткой информации из дерева вызовов о том, на что ссылается указатель, полученный из трасс, на данном этапе работы их рассмотрение опущено. Указатели учитываются только оракулами для поиска зависимостей между функциями. Во-вторых, если среди аргументов функций есть какие-либо флаги или режимы, задаваемые битовыми масками в виде чисел в восьмеричной системе счисления, то в задаче формата SyGuS такие аргументы представляются в виде наборов булевых значений. При этом в целях уменьшения количества неиспользуемой информации, количество булевых флагов выбирается либо как наименьшее число необходимое для кодирования информации (к примеру, если примеры содержат только два значения `0666` и `0555`, достаточно одного флага для кодирования этой информации), либо как число позиций, в которых не совпадали все флаги в их двоичной записи (позиции, где биты всегда совпадают, рассматриваться не будут).

5.4. Поток управления

Задача восстановления потока управления, описанная в 4.4, сводится к решению NP-полной задачи поиска наименьшей общей надстроки,

решение которой весьма трудоемко [67]. Процесс сведения заключается в сопоставлении каждой функции уникального символа. Тогда, после нахождения общей надстроки все группы функций, выполнение которых зависит от общего условия, получаются тривиальным образом.

Рассмотрим пример. Пусть у нас есть три уникальных набора вызовов в истории ($\{g_4, g_5, g_1, g_2\}$, $\{g_1, g_2, g_4, g_1, g_2\}$,$\{g_3, g_4, g_5, g_1, g_2\}$), вызванные внутри функции `f`. Процесс выравнивания и группировки функций представлен на рисунке 9, здесь первая строка содержит имена функций, вторая — выравненные, сопоставленные им символы, следующие три строки описывают выравненные наборы функций, полученные из историй, пятая — результат сгруппированных функций (в квадратных скобках указаны функции, вызовы которых находятся внутри конструкций ветвления).

g_1	g_2	g_3	g_4	g_5	g_1	g_2
$\boldsymbol{a}$	$\boldsymbol{b}$	$\boldsymbol{c}$	$\boldsymbol{d}$	$\boldsymbol{e}$	$\boldsymbol{a}$	$\boldsymbol{b}$
			d	e	a	b
a	b		d		a	b
		c	d	e	a	b
$[g_1$	$g_2]$	$[g_3]$	g_4	$[g_5]$	g_1	g_2

Рис. 9: Восстановление потока управления для набора функций.

В качестве инструмента, ищущего общую надстроку, использовалась утилита множественного выравнивая строк `mafft` [68].

5.5. Ограничения

Текущая реализация содержит множество ограничений. На данном этапе, восстановление потока управления работает только с операторами ветвления; к примеру, циклы из `n` итераций будут представлены как

`n` конструкций ветвления. В будущих исследованиях, планируется расширить анализ дерева вызовов путем выделения шаблонов для поиска итераций циклов.

Еще одним ограничением, связанным с недостатком информации в трассах программ, является отсутствие работы с указателями: из трасс можно лишь получить факт наличия указателя, но не содержимое памяти, на которую он указывает.

6. Апробация подхода и эксперименты

В качестве апробации метода сведения были собраны различные наборы примеров: для тестирования обеих фаз метода были собраны примеры, покрывающие работу с файловой системой с помощью системных вызовов (см. Разделы 6.2, 6.3); для отдельного тестирования метода сведения вручную были написаны трассы программ (см. Раздел 6.1). Все запросы к SyGuS-решателям проводились на ноутбуке с ОС `Ubuntu 16.04`, с процессором `Intel Core i7-7500U` и 8 Гб оперативной памяти. Рассматривались решатели `EUSolver`[1] 2017 года и версии `CVC4`[2] 2017 и 2018 года.

6.1. Синтетические примеры в виде трасс

В качестве синтетических примеров было решено рассмотреть набор искомых функций, разделенных на следующие группы.

1. Функции без внутренних вызовов и без зависимостей от глобальных состояний.
2. Функции с внутренними вызовами и без зависимостей от глобальных состояний.
3. Функции без внутренних вызовов с зависимостями от глобальных состояний.
4. Функции с внутренними вызовами с зависимостями от глобальных состояний.

6.1.1. Группа 1

В группе рассматривалось четыре функции: минимум из двух чисел (`min2`), линейная функция от двух числовых аргументов (`linearF`: `-2 * a1 + 2 * a2 + 3`), функция конкатенации двух строк (`strConcat`),

[1] https://bitbucket.org/abhishekudupa/eusolver/ Дата обращения: 23 июля 2018 г.
[2] http://cvc4.cs.stanford.edu/web/ Дата обращения 25 апреля 2018 г.

функция от двух числовых аргументов в виде тернарного оператора (`f_if: a1 > a2 ? a1 - a2 : a1 + 2`).

Функция `min2` по примерам была найдена обоими всеми решателями за 1-2 секунды. Для функции `strConcat` версии `CVC4` также находят решение за секунды, `EUSolver` не поддерживает работу со строками, когда ограничения заданы в виде, описанном в главе 4. Поиск линейной функции от двух аргументов занял у решателя `CVC4` 2017 года приблизительно 15 минут, 2018 года — ∼5 минут, `EUSolver` смог найти функцию спустя 40 минут. Поиск реализации для функции с ветвлением `f_if` занял у `CVC4` ∼2 минуты и ∼15 секунд для версий 2017 и 2018 года соответственно, `EUSolver` нашел реализацию искомой функции за полчаса.

Стоит отметить, что `EUSolver` находит решения быстрее, чем `CVC4`, если ограничения в формате SyGuS задаются в виде набора равенств (утверждений), а не конъюнкции отношений, но такой подход не позволяет работать с состояниями функций, а значит и зависимостями между ними.

6.1.2. Группа 2

В группе рассматривались следующие функции: функция, вычисляющая максимум из трех элементов (`max3`), вычисляемая путем вызова трех функций максимума из двух (`max2`); функция `f2` с двумя внутренними вызовами, защищенными условиями (см. Листинг 15);

```
int g(int a)
{
  return a * 2;
}

int g1(int a)
{
  return a * 3;
}

int f2(int a)
{
  int r = 0;
  if (a > 0)
    r = g(a-3);
  if (r > 4)
```

```
    r = g1(r);
  return r;
}
```

Листинг 15: Функция с внутренним вызовов и условием (без зависимостей между функциями и без глобального состояния).

Синтезированная версия функции (`max3`) совпадает с ее описанием (три вызова функции (`max2`)). Вместе с синтезированной версией `f2` были синтезированы функции `g, g1`, их реализации совпали с исходными (умножение было записано как сложение), но `f2` уже отличается от искомой (см. Листинг 16). Не смотря на отличия, данная реализация функции `f2` остается корректной реализацией на заданных трассах. Версия решателя `CVC4` 2017 года завершила решение всех задач за ∼3 минуты, 2018 — быстрее в 4 раза; `EUSolver` — ∼1 час.

```
1  int f2(int a)
2  {
3    int r1 = maxInt, r2 = maxInt;
4    if (0 < a)
5      r1 = g(a - (1 + (1 + 1)));
6
7    if (0 < (r1 < a ? 0 : a))
8      r2 = g1(r1);
9
10   return r1 < a
11     ? r1
12     : (r1 < r2 ? r2 : 0);
13 }
```

Листинг 16: Синтезированная версия функции `f2`.

В реализации функции `f2` в строке 5 ожидаемое выражение `a - 3` намеренно оставлено в виде, полученном от решателей, чтобы отметить, что возможности решателей сильно ограничены (увеличение высоты синтаксического дерева сильно увеличивает время поиска искомой функции).

6.1.3. Группа 3

В этой группе были рассмотрены две функции: функция, возвращающая следующую степень двойки (см. Листинг 17); функция, возвращающая значение глобального состояния в зависимости от знака аргумента (см. Листинг 18).

```
int previous = 1;

int next()
{
  previous *= 2;
  return previous;
}
```

Листинг 17: Степень двойки.

```
int state_pos = 0;
int state_neg = 0;

int twoState(a)
{
  if (a > 0)
    return state_pos++;
  else
    return state_neg--;
}
```

Листинг 18: Функция двух состояний.

Функция `next` при изначальном значении состояния `previous = 1` была найдена обоими решателями меньше чем за минуту (`CVC4`: ∼15(4) с., `EUSolver`: ∼50 с.). Как упоминалось ранее, увеличение высоты синтаксического дерева искомых функций может сильно увеличить время поиска, к примеру, при замене начального состояния на `previous = 2` решателям потребовалось в среднем в 20 раз больше времени.

Задача для функции `twoState` без упрощения пользователем пространства перебора всеми решателями за 12 часов завершена не была. При внесении пользователем информации о виде функции-эффекта, изменяющей состояние, `CVC4` 2018 года нашел реализацию функции вместе с функцией-эффектом и начальными значениями состояния за ∼20 минут, 2017 года — за ∼2 часа, `EUSolver` предложил реализации функций спустя ∼6 часов.

6.1.4. Группа 4

Здесь рассматривались две функции, зависящие от одного состояния (см. Листинг 19).

```
int counter = 0;

int g(int a)
{
  int r = counter + a;
  ++counter;
  return r + 2;
}

int f(int a1, int a2)
{
  int r = 0;
```

```
  if (a1 > 0)
    r = g(a1 + a2);
  ++counter;
  return r;
}
```

Листинг 19: Влияющие друг на друга функции.

Весь процесс поиска всех функций, с ограничениями, построенными по деревьям вызовов (см. Рис. 10), в сумме был закончен решателем `CVC4` 2018 года за ∼`23` часа, `EUSolver` и `CVC4` 2017 года были остановлены спустя сутки — решение найдено не было.

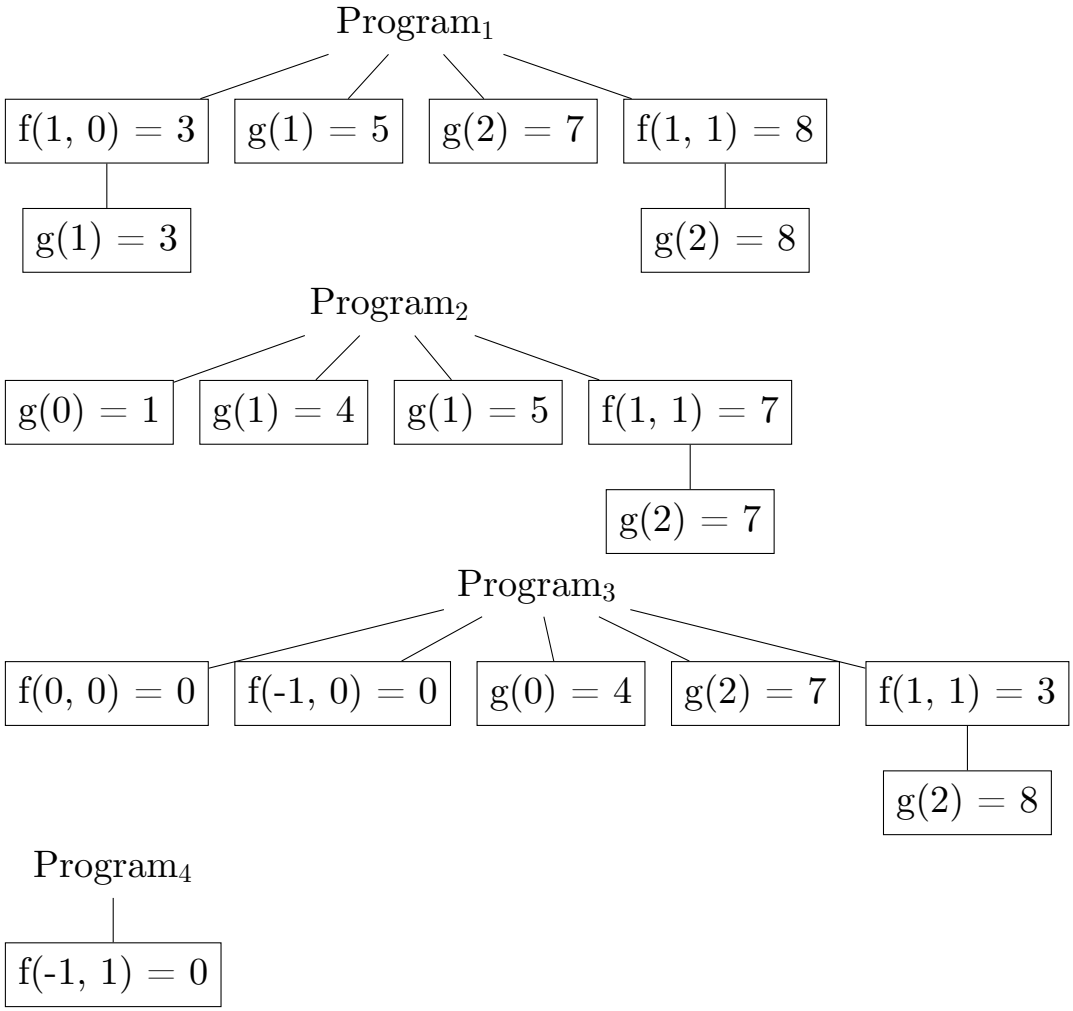


Рис. 10: Деревья вызовов, построенные по трассам.

Результат синтеза, представлен в Листинге 20. Результат отличается от «оригинальных» (неизвестных) функций, но удовлетворяет ограничениям, которые были получены из трасс.

```
int state_F = 0;
int state_G = 0;
```

```
void g_effect_f(int a1, int result)
{
  state_F = state_F + 1;
}

void g_effect_g(int a1, int result)
{
  state_G = state_G + 1;
}

void f_effect_g(int a1, int a2, int innerResult, int result)
{
  state_G = a1;
}

void f_effect_f(int a1, int a2, int innerResult, int result)
{
  state_F = a1;
}

int g(int a1)
{
  int result = ((1 + 1) + a1) + state_G;
  g_effect_g(a1, result);
  g_effect_f(a1, result);
  return r + 2;
}

int f(int a1, int a2)
{
  int r1 = 0;
  if (0 < a1)
    r1 = g(a1 + a2);

  int result = state_F == 0 ? state_F : r1;
  f_effect_f(a1, a2, r1, result);
  f_effect_g(a1, a2, r1, result);
  return result;
}
```

Листинг 20: Синтезированные версии функций `f` и `g`.

6.2. Простые версии реальных функций

В качестве эксперимента рассматривались группы примеров, содержащие вызовы функций `open, close, write, read`. Для простоты демонстрации, в этом разделе будут описаны пары «противоположных» функций.

6.2.1. Функции `open` и `close`

Функция open в простом случае, при успешном открытии файла, возвращает первый свободный файловый дескриптор. Простая программа, иллюстрирующая это, представлена в Листинге 21.

```
#include<stdio.h>
#include<fcntl.h>

int main()
{
    int fd = open("foo.txt", O_CREAT, 0777);
        fd = open("foo.txt", O_CREAT, 0777);
        fd = open("foo.txt", O_CREAT, 0777);
    return 0;
}
```

Листинг 21: Программа, описывающая поведение `open`.

В виду того, что в данном примере все флаги совпадали, они не дошли до задачи синтеза, в которой рассматривались «адаптированные» сигнатуры.

Вызов `open` включает в себя системный вызова `SYS_open` (см. Рис 11).

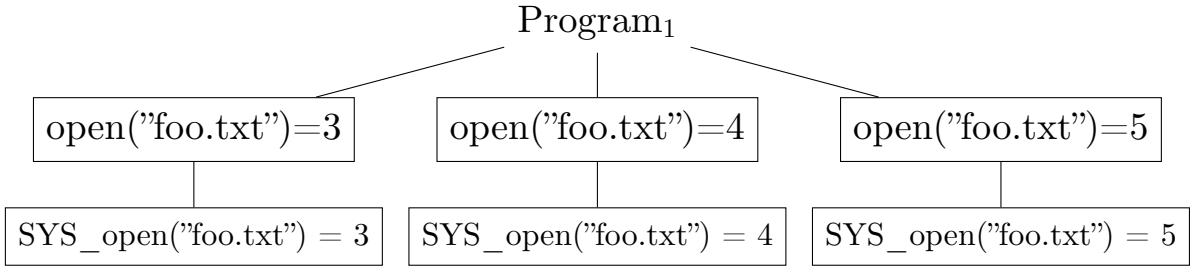


Рис. 11: Дерево вызовов, для примера из Листинга 21.

По этому дереву оракулы выделили в качестве кортежа состояний, от которого может зависеть функция `open`, строковую и числовую переменные (аналогично для `SYS_open`). Решателем `CVC4` была найдена реализация, которая на языке `C` выглядит так (см. Листинг 22).

```
char *state_string_SYS_OPEN; // unused
char *state_string_open; // unused

int state_int_open = 0; // unused
int state_int_SYS_OPEN = 1 + 1;

int SYS_open(char *file, ...)
{
    int r = SYS_open(file);
    // effects are empty
    return state_int_SYS_OPEN + 1;
```

```
}

int open(char *file, ...)
{
    int r = SYS_open("foo.txt");
    // effect on SYS_open inlined
    state_int_SYS_OPEN = r;
    return r;
}
```

Листинг 22: Реализация функций, записанная на языке `C`.

Функция `close` пытается закрыть файл с ассоциированным ему файловым дескриптором. Деревья вызовов, представленные на рисунке 12, описывают простое взаимодействие функций открытия и закрытия файлов.

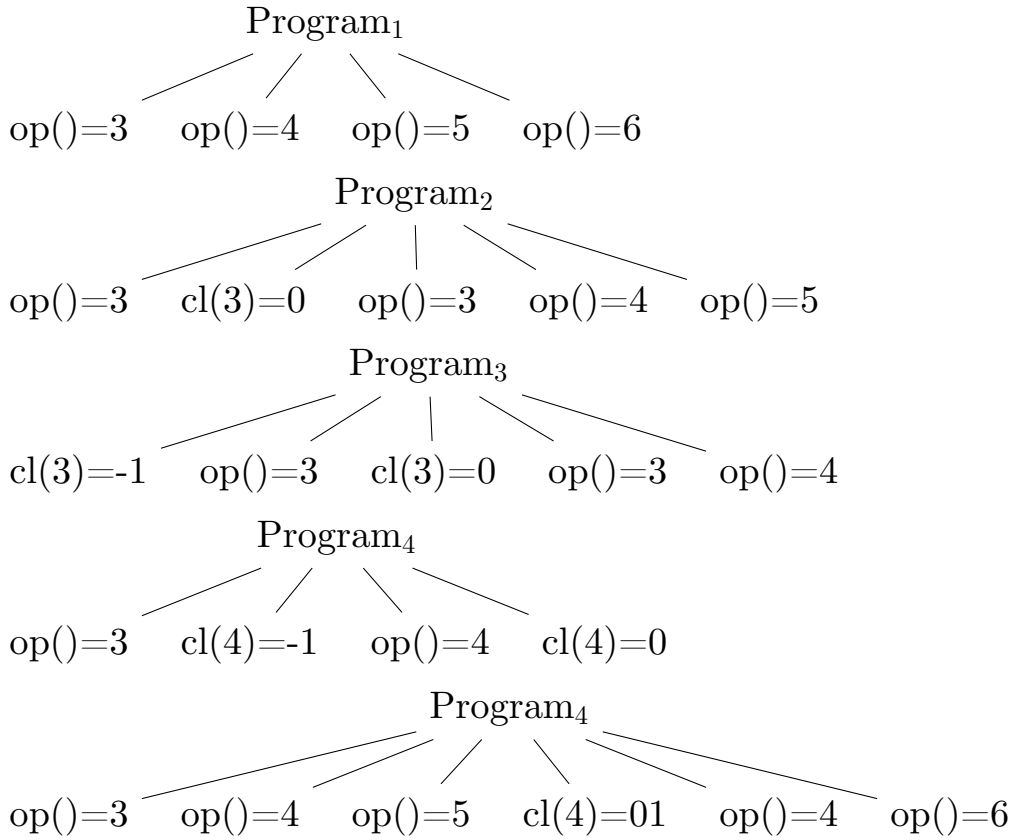


Рис. 12: Деревья вызовов, для взаимодействия `op (open)` и `cl (close)`.

Для таких деревьев вызовов, решатель задачи синтеза `cvc4` нашел самую простую реализацию, ее запись на языке `C` представлена в Листинге 23.

```
int state_close_0 = 0;
int state_close_1 = -1;
```

```
int effect_open_on_close(int open_res)
{
    state_close_0 = open_res;
}

int close(int fd)
{
    int res;
    if (state_close_0 < fd)
        res = state_close_1;
    else
        res = state_close_0;

    // effect close on close
    state_close_0 = fd;
    state_close_1 = fd;

    return res;
}
```

Листинг 23: Синтезированная версия функции `close`.

6.2.2. Функции `write` и `read`

Рассмотрим простые версии функций `write` и `read`. Пусть функция `write` записывает строку в указанный дескриптор и возвращает длину записанной строки, а функция `read` возвращает все записанные в него данные в виде строки. Для этих функций были построены следующие деревья вызовов (см. Рис 13).

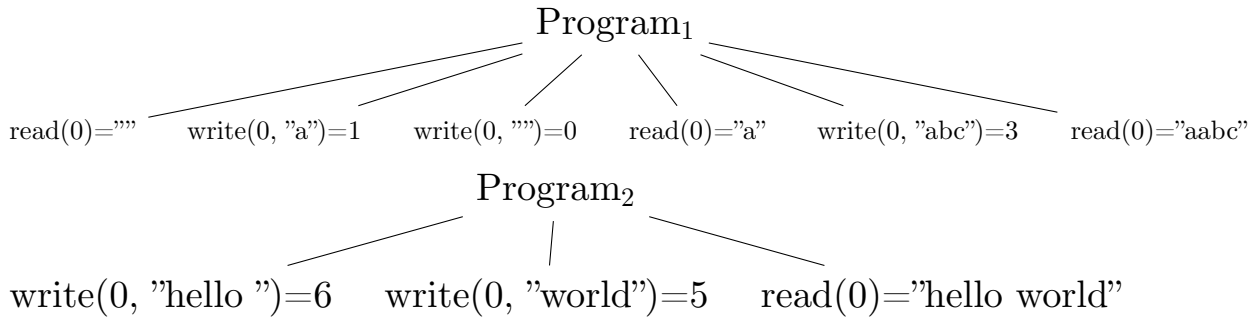


Рис. 13: Деревья вызовов, для взаимодействия функций `write` и `read`.

Функции влияют друг на друга. Для состояния, состоящего из строковой и числовой переменной, решатели задачи синтеза за сутки смогли найти реализации обеих функций. Результат, представленный в укороченном виде, полученный от решателей, для обеих функций представлен в Листинге 24. Код синтезированных функций на языке `C` с эффектами, записанными в телах функций — в Листинге 25.

```
; write
(define-fun state_0 ((var String)) Bool (= var ""))

(define-fun state_1 ((var Int)) Bool (= var 0))

(define-fun write
 ((arg_0 Int) (arg_1 String) (result Int) (state_0 String) (state_1 Int)) Bool
(= result (+ (str.len arg_1) arg_0)))

(define-fun write_EffectOn_write
 ((arg_0 Int) (arg_1 String) (result Int) (state_0 String)
 (state_1 Int) (state_0_Upd String) (state_1_Upd Int)) Bool
(and (= state_0_Upd arg_1) (= state_1_Upd arg_0)))

(define-fun mread_EffectOn_write
 ((arg_0 Int) (result String) (state_0 String) (state_1 Int)
 (state_0_Upd String) (state_1_Upd Int)) Bool
(and (= state_0_Upd state_0) (= state_1_Upd arg_0)))

; read

(define-fun state_0 ((var String)) Bool (= var ""))

(define-fun state_1 ((var Int)) Bool (= var 0))

(define-fun read
 ((arg_0 Int) (result String) (state_0 String) (state_1 Int)) Bool
(= result state_0))

(define-fun mwrite_EffectOn_read
 ((arg_0 Int) (arg_1 String) (result Int) (state_0 String)
 (state_1 Int) (state_0_Upd String) (state_1_Upd Int)) Bool
(and (= state_0_Upd (str.++ state_0 arg_1)) (= state_1_Upd arg_0)))

(define-fun read_EffectOn_read
 ((arg_0 Int) (result String) (state_0 String) (state_1 Int)
 (state_0_Upd String) (state_1_Upd Int)) Bool
(and (= state_0_Upd state_0) (= state_1_Upd arg_0)))
```

Листинг 24: Синтезированные версии функций `write` и `read`, полученные от решателей.

```
#include <string.h>

char writeStateS[1024];
char readStateS[1024];
int writeStateI;
int readStateI;

int write(int fd, char *str)
{
 int res = strlen(str);

 // effect on write
 strncpy(writeStateS, str, sizeof(str));
 writeStateS[strlen(str)] = '\0';
 writeStateI = fd;
 // effect on read
 strcat(readStateS, str);
 readStateI = fd;

 return res;
}
```

```
char* read(int fd)
{
 char* res = &readStateS;

 // effect on read
 readStateI = fd;
 // effect on write
 writeStateI = fd;

 return res;
}
```

Листинг 25: Синтезированные версии функций `write` и `read` на языке `C`.

6.3. Примеры, покрывающие `stdio.h`

Был проведен эксперимент формирования и запуска задач синтеза по примерам для функций, присутствующих в подключаемом заголовочном файле `stdio.h`. Примеры собирались из разных источников: из обучающих ресурсов[3,4], из общеизвестных ресурсов, посвященным языку `C` и `C++`[5], из тестов библиотеки `libc`[6,7], часть примеров была написана вручную.

Всего инструменту было подано на вход `43` файла примеров на языке `C`, покрывающих работу функций `remove, rename, fopen, fclose, fflush, freopen, setbuf, fgetc, fgets, fputc, fputs, feof, fread, fwrite, fseek, rewind` (в некоторых файлах встречалась функция `strlen`). Размер каждого примера не превышал `30` строк кода.

По примерам было сформировано `30` файлов в формате `SyGuS-IF`, содержащих задачи для поиска всех перечисленных выше функций и системных вызовов, которые они используют, и функций-эффектов, и `86` файлов, содержащих задачи для вычисления выражений для аргументов внутренних функций и условий для конструкций ветвления восстановленного потока управления.

Так как задачи для аргументов внутренних функций и условий ветв-

[3]https://www.geeksforgeeks.org/input-output-system-calls-c-create-open-close-read-write/ Дата обращения: 23 июля 2018 г.

[4]https://www.tutorialspoint.com/cprogramming/c_file_io.htm Дата обращения: 23 июля 2018 г.

[5]http://www.cplusplus.com/reference/cstdio/ Дата обращения: 23 июля 2018 г.

[6]http://www.etalabs.net/src/libc-bench/ Дата обращения: 23 июля 2018 г.

[7]http://nsz.repo.hu/git/?p=libc-test;a=tree;hb=HEAD Дата обращения: 23 июля 2018 г.

ления зависят от результатов синтеза для содержащих их функций, то сначала производился запуск решателей на первых 30 файлах. Запуск производился с решателем CVC4 с ограничением по времени в 12 часов. За отведенное время решатель CVC4 синтезировал функции strlen, fseek и rewind, остальные задачи решены не были. Стоит отметить, что хотя задачи синтеза для этих функций и были завершены, в данном случае имеется в виду частичное решение задачи синтеза для них, так как их эффекты синтезируются в других подзадачах (совместно с функциями, на которые они влияют).

6.4. Выводы

Эксперименты показали работоспособность сведения задачи автоматического синтеза императивных функций к проблеме синтаксически-направленного синтеза. Был сделан вывод, что для «точных» реализаций искомых функций необходимо предъявить большое количество примеров, количество которых сильно влияет на время поиска. С простыми функциями решатели задачи синтаксически-направленного синтеза справляются за разумное время. Решения для более сложных задач на текущий момент развития области требует доработки решателей задачи SyGuS.

Обнадеживающим фактом является то, что решатели задач синтеза SyGuS активно развиваются. За время работы над диссертацией вышло несколько версий решателя CVC4, последние версии которого решают отдельные задачи быстрее чем версия, представленная на соревновании SyGuS-Comp 2017, в 3–15 раз.

7. Заключение

В рамках работы были достигнуты следующие результаты.

- Был разработан метод синтеза моделей внешних вызовов по трассам программ. Представленный метод позволяет свести задачу синтеза моделей императивных функций к проблеме синтаксически-направленного синтеза (SyGuS). Описанный в работе метод включает в себя анализ трасс программ, содержащих внешние вызовы; формализацию воздействия искомых функций друг на друга и их зависимость от глобального состояния программы; восстановление по трассам потока управления для построения моделей искомых внешних вызовов (функций); формирование на основе трасс семантических ограничений;

- был создан прототип инструмента, который автоматизированно выполняет сбор трасс из набора примеров, представленных пользователем, и реализует описанный метод путем генерации множества задач в формате SyGuS с последующим запуском решателей для этих задач;

- прототип инструмента был апробирован на наборе синтетических и реальных примеров. Были поставлены эксперименты с передовыми решателями (победителями соревнований SyGuS-Comp 2017) `CVC4` и `EUSolver`. На основании проведенных экспериментов был сделан вывод, что подход в целом можно считать успешным (было синтезировано множество небольших функций), однако для масштабирования подхода решатели задачи SyGuS требуют улучшений.

Будущие исследования

Предложенный алгоритм содержит множество ограничений, решение которых может значительно улучшить представленный в работе результат.

Одним из возможных путей развития является улучшение алгоритма восстановления потока управления путем добавления поддержки циклов. Другим возможным улучшением является более мощный анализ дерева вызовов, построенного на основе трасс программ, к примеру, с помощью методов машинного обучения и статического анализа кода библиотек, с целью выявления зависимостей между функциями и состоянием, от которого зависит их выполнение. Расширение возможностей сбора информации, чтобы иметь возможность работать с указателями и памятью языка `C` является еще одной важной ветвью исследований.

Одной из наиболее важных ветвей развития работы является улучшение подходов к решению задачи SyGuS.

Список литературы

[1] Clarke Edmund M, Grumberg Orna, Long David E. Model checking and abstraction // ACM transactions on Programming Languages and Systems (TOPLAS). 1994. Т. 16, № 5. С. 1512–1542.

[2] Мордвинов ДА, Литвинов ЮВ. Обзор применения формальных методов в робототехнике // Научно-технические ведомости Санкт-Петербургского государственного политехнического университета. Информатика. Телекоммуникации. Управление. 2016. № 1 (236).

[3] Haxthausen Anne E. An introduction to formal methods for the development of safety-critical applications // DTU Informatics Technical University of Denmark. 2010.

[4] Rodhe Ioana, Karresand Martin. Overview of formal methods in software engineering. 2015. 12.

[5] Goguen Joseph A [и др.]. Formal methods: Promises and problems // IEEE Software. 1997. Т. 14, № 1. С. 73–85.

[6] Batra Mona. Formal methods: Benefits, challenges and future direction // International Journal of Global Research in Computer Science (UGC Approved Journal). 2013. Т. 4, № 5. С. 21–25.

[7] Formal Methods for Commercial Applications Issues vs. Solutions / Saiqa Bibi, Saira Mazhar, Nasir Mehmood Minhas [и др.] // Journal of Software Engineering and Applications. 2014. Т. 7, № 08. С. 679.

[8] Bjørner Dines, Havelund Klaus. 40 years of formal methods // International Symposium on Formal Methods / Springer. 2014. С. 42–61.

[9] Serna Edgar M, Serna Alexei A. Power and limitations of formal methods for software fabrication: Thirty years later // Informatica. 2017. Т. 41, № 3. С. 275–282.

[10] King James C. Symbolic execution and program testing // Communications of the ACM. 1976. Т. 19, № 7. С. 385–394.

[11] A survey of symbolic execution techniques / Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia [и др.] // arXiv preprint arXiv:1610.00502. 2016.

[12] Cadar Cristian, Sen Koushik. Symbolic execution for software testing: three decades later // Communications of the ACM. 2013. Т. 56, № 2. С. 82–90.

[13] Havelund Klaus, Pressburger Thomas. Model checking java programs using java pathfinder // International Journal on Software Tools for Technology Transfer. 2000. Т. 2, № 4. С. 366–381.

[14] Mehlitz Peter, Tkachuk Oksana, Ujma Mateusz. Jpf-awt: Model checking gui applications // Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering / IEEE Computer Society. 2011. С. 584–587.

[15] Synthesizing framework models for symbolic execution / Jinseong Jeon, Xiaokang Qiu, Jonathan Fetter-Degges [и др.] // Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on / IEEE. 2016. С. 156–167.

[16] Pnueli Amir, Rosner Roni. On the synthesis of a reactive module // Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages / ACM. 1989. С. 179–190.

[17] Gulwani Sumit, Esparza Javier, Grumberg Orna [и др.]. Programming by Examples-and its applications in Data Wrangling. 2016.

[18] Program synthesis using natural language / Aditya Desai, Sumit Gulwani, Vineet Hingorani [и др.] // Proceedings of the 38th International Conference on Software Engineering / ACM. 2016. С. 345–356.

[19] Solar-Lezama Armando. Program synthesis by sketching. University of California, Berkeley, 2008.

[20] Program synthesis / Sumit Gulwani, Oleksandr Polozov, Rishabh Singh [и др.] // Foundations and Trends® in Programming Languages. 2017. Т. 4, № 1-2. С. 1–119.

[21] Manna Zohar, Waldinger Richard. A deductive approach to program synthesis // Readings in artificial intelligence and software engineering. Elsevier, 1986. С. 3–34.

[22] Syntax-guided synthesis / Rajeev Alur, Rastislav Bodik, Garvit Juniwal [и др.] // Formal Methods in Computer-Aided Design (FMCAD), 2013 / IEEE. 2013. С. 1–8.

[23] Synthesis of loop-free programs / Sumit Gulwani, Susmit Jha, Ashish Tiwari [и др.] // ACM SIGPLAN Notices. 2011. Т. 46, № 6. С. 62–73.

[24] Gulwani Sumit, Harris William R, Singh Rishabh. Spreadsheet data manipulation using examples // Communications of the ACM. 2012. Т. 55, № 8. С. 97–105.

[25] Oracle-guided component-based program synthesis / Susmit Jha, Sumit Gulwani, Sanjit A Seshia [и др.] // Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1 / ACM. 2010. С. 215–224.

[26] TRANSIT: specifying protocols with concolic snippets / Abhishek Udupa, Arun Raghavan, Jyotirmoy V Deshmukh [и др.] // ACM SIGPLAN Notices. 2013. Т. 48, № 6. С. 287–296.

[27] Srivastava Saurabh, Gulwani Sumit, Foster Jeffrey S. From program verification to program synthesis // ACM Sigplan Notices / ACM. Т. 45. 2010. С. 313–326.

[28] Software synthesis procedures / Viktor Kuncak, Mikaël Mayer, Ruzica Piskac [и др.] // Communications of the ACM. 2012. Т. 55, № 2. С. 103–111.

[29] Raghothaman Mukund, Udupa Abhishek. Language to specify syntax-guided synthesis problems // arXiv preprint arXiv:1405.5590. 2014.

[30] Results and analysis of sygus-comp'15 / Rajeev Alur, Dana Fisman, Rishabh Singh [и др.] // arXiv preprint arXiv:1602.01170. 2016.

[31] SyGuS-Comp 2016: Results and Analysis / Rajeev Alur, Dana Fisman, Rishabh Singh [и др.] // arXiv preprint arXiv:1611.07627. 2016.

[32] Cvc4 / Clark Barrett, Christopher L Conway, Morgan Deters [и др.] // International Conference on Computer Aided Verification / Springer. 2011. С. 171–177.

[33] Counterexample-guided quantifier instantiation for synthesis in SMT / Andrew Reynolds, Morgan Deters, Viktor Kuncak [и др.] // International Conference on Computer Aided Verification / Springer. 2015. С. 198–216.

[34] Alur Rajeev, Radhakrishna Arjun, Udupa Abhishek. Scaling enumerative program synthesis via divide and conquer // International Conference on Tools and Algorithms for the Construction and Analysis of Systems / Springer. 2017. С. 319–336.

[35] Satisfiability Modulo Theories. / Clark W Barrett, Roberto Sebastiani, Sanjit A Seshia [и др.] // Handbook of satisfiability. 2009. Т. 185. С. 825–885.

[36] De Moura Leonardo, Bjørner Nikolaj. Satisfiability modulo theories: introduction and applications // Communications of the ACM. 2011. Т. 54, № 9. С. 69–77.

[37] Burnim Jacob, Sen Koushik. Heuristics for scalable dynamic test generation // Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on / IEEE. 2008. С. 443–446.

[38] KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. / Cristian Cadar, Daniel Dunbar, Dawson R Engler [и др.] // OSDI. Т. 8. 2008. С. 209–224.

[39] EXE: automatically generating inputs of death / Cristian Cadar, Vijay Ganesh, Peter M Pawlowski [и др.] // ACM Transactions on Information and System Security (TISSEC). 2008. Т. 12, № 2. С. 10.

[40] Fitness-guided path exploration in dynamic symbolic execution / Tao Xie, Nikolai Tillmann, Jonathan de Halleux [и др.] // Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on / IEEE. 2009. С. 359–368.

[41] Chopped Symbolic Execution / David Trabish, Andrea Mattavelli, Noam Rinetzky [и др.]. 2018.

[42] Erete Ikpeme, Orso Alessandro. Optimizing constraint solving to better support symbolic execution // Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on / IEEE. 2011. С. 310–315.

[43] Jia Xiangyang, Ghezzi Carlo, Ying Shi. Enhancing reuse of constraint solutions to improve symbolic execution // Proceedings of the 2015 International Symposium on Software Testing and Analysis / ACM. 2015. С. 177–187.

[44] Palikareva Hristina, Cadar Cristian. Multi-solver support in symbolic execution // International Conference on Computer Aided Verification / Springer. 2013. С. 53–68.

[45] Sen Koushik, Marinov Darko, Agha Gul. CUTE: a concolic unit testing engine for C // ACM SIGSOFT Software Engineering Notes / ACM. Т. 30. 2005. С. 263–272.

[46] Industrial application of concolic testing approach: A case study on libexif by using CREST-BV and KLEE / Yunho Kim, Moonzoo Kim, YoungJoo Kim [и др.] // Proceedings of the 34th International

Conference on Software Engineering / IEEE Press. 2012. С. 1143–1152.

[47] Godefroid Patrice, Klarlund Nils, Sen Koushik. DART: directed automated random testing // ACM Sigplan Notices / ACM. Т. 40. 2005. С. 213–223.

[48] Thorough static analysis of device drivers / Thomas Ball, Ella Bounimova, Byron Cook [и др.] // ACM SIGOPS Operating Systems Review. 2006. Т. 40, № 4. С. 73–85.

[49] Ceccarello Matteo, Tkachuk Oksana. Automated generation of model classes for Java PathFinder // ACM SIGSOFT Software Engineering Notes. 2014. Т. 39, № 1. С. 1–5.

[50] Kolmogoroff Andrej. Zur deutung der intuitionistischen logik // Mathematische Zeitschrift. 1932. Т. 35, № 1. С. 58–65.

[51] Green Cordell. Theorem proving by resolution as a basis for question-answering systems // Machine intelligence. 1969. Т. 4. С. 183–205.

[52] Waldinger Richard J, Lee Richard CT. PROW: A step toward automatic program writing // Proceedings of the 1st international joint conference on Artificial intelligence / Morgan Kaufmann Publishers Inc. 1969. С. 241–252.

[53] Green Cordell. Application of theorem proving to problem solving // Readings in Artificial Intelligence. Elsevier, 1981. С. 202–222.

[54] Manna Zohar, Waldinger Richard. Knowledge and reasoning in program synthesis // Artificial intelligence. 1975. Т. 6, № 2. С. 175–208.

[55] Manna Zohar, Waldinger Richard. Fundamentals of deductive program synthesis // IEEE Transactions on Software Engineering. 1992. Т. 18, № 8. С. 674–704.

[56] Shaw D, Wartout W, Green Cordell. Inferring LISP Programs From Examples. // IJCAI. T. 75. 1975. C. 260–267.

[57] Biermann Alan W. The inference of regular LISP programs from examples // IEEE transactions on Systems, Man, and Cybernetics. 1978. T. 8, № 8. C. 585–600.

[58] Smith David Canfield. Pygmalion: a creative programming environment: Tech. Rep.: : STANFORD UNIV CA DEPT OF COMPUTER SCIENCE, 1975.

[59] Lau Tessa A, Domingos Pedro M, Weld Daniel S. Version Space Algebra and its Application to Programming by Demonstration. // ICML. 2000. C. 527–534.

[60] Neuro-symbolic program synthesis / Emilio Parisotto, Abdelrahman Mohamed, Rishabh Singh [и др.] // arXiv preprint arXiv:1611.01855. 2016.

[61] Ghédira Khaled. Constraint satisfaction problems: csp formalisms and techniques. John Wiley & Sons, 2013.

[62] SyGuS Syntax for SyGuS-COMP'16: Tech. Rep.: / Rajeev Alur, Dana Fisman, P Madhusudan [и др.]: Technical report, University of Pennsylvania, 2016. Available at http://sygus. seas. upenn. edu/files/SyGuS-Syntax-SyGuSCOMP'16. pdf, 2015.

[63] The smt-lib standard: Version 2.0 / Clark Barrett, Aaron Stump, Cesare Tinelli [и др.] // Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England). T. 13. 2010. C. 14.

[64] Phillips Jorge V. Program Inference from Traces using Multiple Knowledge Sources. // IJCAI. 1977. C. 812.

[65] Lau Tessa, Domingos Pedro, Weld Daniel S. Learning programs from traces using version space algebra // Proceedings of the 2nd

international conference on Knowledge capture / ACM. 2003. С. 36–43.

[66] Gamma Erich. Design patterns: elements of reusable object-oriented software. Pearson Education India, 1995.

[67] Gallant John, Maier David, Astorer James. On finding minimal length superstrings // Journal of Computer and System Sciences. 1980. Т. 20, № 1. С. 50–58.

[68] Katoh Kazutaka, Standley Daron M. MAFFT multiple sequence alignment software version 7: improvements in performance and usability // Molecular biology and evolution. 2013. Т. 30, № 4. С. 772–780.