Санкт-Петербургский государственный университет

Кафедра системного программирования

Танков Владислав Дмитриевич

Основанный на данных синтез кода в IntelliJ IDEA

Выпускная квалификационная работа

Научный руководитель:
к. т.н., доцент Брыксин Т. А.

Рецензент:
ст. преп. Шпильман А. А.

Санкт-Петербург
2018

SAINT-PETERSBURG STATE UNIVERSITY

Chair of Software Engineering

Vladislav Tankov

Data-based synthesis of code in IntelliJ IDEA

Graduation Thesis

Scientific supervisor:
assistant professor Timofey Bryksin

Reviewer:
senior lecturer Aleksey Shpilman

Saint-Petersburg
2018

Оглавление

Введение

Задача автоматического синтеза кода — предмет множества исследований в области информатики и программирования на протяжении десятилетий. Под задачей автоматического синтеза кода понимается задача генерации описания некоторого алгоритма на определенном языке программирования из неполной спецификации данного алгоритма. В зависимости от подхода спецификацией может быть описание алгоритма на некотором предметно-ориентированном языке (domain specific language, DSL), описание на естественном языке, набор входов и выходов алгоритма, набор системных вызовов, происходящих при выполнении алгоритма и т.д.

В последние годы все больше исследователей обращаются к данной проблеме, а на рынке появляются первые промышленные образцы автоматических синтезаторов кода [6]. Однако, большинство современных инструментов синтеза кода слабо применимы в промышленности, так как они требуют от программиста освоения совершенно нового и практически всегда оторванного от основной области знаний программиста формализма. К примеру, DSL-подход требует изучения отдельного языка спецификаций.

В этом плане коренным образом отличается подход Bayesian Sketch Learning (BSL-подход), предложенный группой исследователей из Университета Райса в их статье [10]. BSL-подход позволяет использовать в качестве такого предметно-ориентированного языка идентификаторы языка и библиотек, используемых программистом. Это позволяет сократить разрыв между языком спецификации, используемым синтезатором кода, и кодом, синтезируемым системой и используемым в дальнейшем программистом.

У такого подхода есть весомый плюс: так как синтезатор, использующий данный подход, работает с библиотеками и языком, используемым программистом, и более ничем, то его возможно внедрить в интегрированную среду разработки (IDE) программиста, не изменяя при этом обычных паттернов использования этой IDE. Используя такой

синтезатор, IDE может предлагать более интеллектуальные контекстные подсказки (Auto-Completion) и даже в некоторых случаях синтезировать части разрабатываемой системы.

Постановка задачи

Исходя из вышесказанного была поставлена цель интегрировать подход BSL-синтезаторов как один из наиболее интересных современных подходов к синтезу кода с IntelliJ IDEA — ведущей IDE для JVM платформы. Для достижения цели потребовалось:

- разработать архитектуру конфигурируемого BSL-синтезатора;
- реализовать BSL-синтезатор на JVM платформе;
- создать интерфейс для взаимодействия пользователя и синтезатора;
- воспроизвести результаты эталонной реализации Bayou (см. [10]), таким образом продемонстрировав эквивалентность полученного синтезатора эталонной реализации.

1. Обзор

1.1. Обзор концепции BSL-синтезатора

BSL-синтезатор — это синтезатор программного кода, основанный на байесовском подходе (Bayesian Learning) [1]. Суть байесовского подхода в данном случае сводится к следующему: синтезатор обучается на готовом корпусе программ и так называемых "свидетельств" — некоторых значений, связанных с выполняемой программой задачей. После чего, получая некоторый набор свидетельств о программе, пытается синтезировать программный код, наиболее вероятно удовлетворяющий данным свидетельствам. То есть в процессе обучения строится априорное распределение программ по наборам свидетельств, а в процессе синтеза строится апостериорное распределение для конкретных переданных свидетельств.

Свидетельствами для BSL-синтезатора выступают как правило вызовы методов и классы, используемые в синтезируемой функции. При этом заметим, что эксперименты в работе [10] показали высокую эффективность BSL-синтезаторов для кода с большим числом API-вызовов. Заметим, что синтез такого типа кода — актуальная задача, решающая некоторые из проблем, описанных в [12] (к примеру, недостаток примеров, представляющих общие подходы к работе с API, недостаток примеров, представляющих взаимодействие нескольких API-методов и т.д.). В то же время, эффективность применения BSL-синтезатора для синтеза произвольного кода на данный момент не исследована.

В качестве внутреннего представления программ BSL-синтезатор использует скетчи [13]. Скетч — это упрощенное представление программы, содержащее лишь основные конструкции языка (такие как конструкции управления потоком исполнения и, в случае Bayou, API вызовы библиотек). Скетчи не сохраняют семантику программы, но зато они позволяют представлять программы со схожими задачами единообразно (семантику, тем не менее, можно восстановить с помощью вероятностных методов, см. [10]).

Скетчи связываются со свидетельствами с помощью техники байесовских кодировщиков-декодировщиков (довольно близких к VAE (variational autoencoders) [8]). Передаваемые свидетельства преобразуются в элемент пространства скрытой переменной ("кодируются"), а скрытая переменная преобразуется ("декодируется") в скетч, соответствующий данным свидетельствам.

Таким образом, с теоретико-вероятностной точки зрения происходят следующие преобразования.

1. Пусть X — набор свидетельств, Z — скрытая переменная [1], соответствующая X, Y — скетч, соответствующий Z.
2. Вычислим $P(Z|X)$ — распределение скрытой переменной при X.
3. Сэмплируем[2] Z в соответствии с полученным распределением.
4. Вычислим $P(Y|Z)$ — распределение скетчей при Z.
5. Сэмплируем Y в соответствии с полученным распределением.

Сочетая байесовский подход с представлением программ в виде скетчей, BSL-синтезатор может корректно синтезировать достаточно сложные методы работы с библиотекой и при этом учитывать распространенные практики использования конкретной библиотеки.

1.2. Обзор альтернативных подходов

На данный момент уже существует некоторый набор технологий, не основанных на BSL-подходе и позволяющих программисту получить методы работы с библиотекой по своего рода "свидетельствам". Рассмотрим эти альтернативы.

[1] Wikipedia, URL: `https://en.wikipedia.org/wiki/Latent_variable`

[2] То есть, в данном случае, выберем некоторое значение случайной величины на пространстве скрытой переменной в соответствии с ее распределением на нем.

1.2.1. Bing Developer Assistant (BDA)

Bing Developer Assistant [2] — это система поиска примеров функций и даже целых проектов по запросам на естественном языке. BDA включает в себя плагин к Visual Studio, предоставляющий пользовательский интерфейс (frontend), и облачную платформу, обеспечивающую поиск (backend).

Frontend часть предоставляет несколько способов коммуникации с пользователем — это естественный язык (к примеру, фраза "how to save png image") и запрос об ошибке компиляции (BDA соберет данные о текущих ошибках). Получив ответ от backend части, BDA может вставить код в текущий проект пользователя, открыть отдельное окно с примерами кода или даже предложить целые проекты с GitHub, подходящие под текущий запрос.

Кроме того, frontend автоматически извлекает данные о контексте запроса и может обогощать запрос этими данными (к примеру, преобразуя "how to save png image" в случае C# проекта в "how to save png image in C#")

Для поиска кода по естественному запросу backend часть использует Bing[3], ограничиваясь, впрочем, жестко заданным набором сайтов для поиска. Специальный фреймворк, предложенный в [2], извлекает код из найденных Bing интернет страниц и ранжирует их, после чего подготовленные примеры кода передаются frontend части.

Для поиска решения ошибок компиляции опять же используется Bing, однако запрос составляется автоматически из самой ошибки и данных контекста (таких как язык проекта, данные о текущем коде проекта и подключенных к нему библиотеках). Полученные результаты опять же отображаются пользователю в frontend части.

BDA — удачная система поиска кода, признанная пользователями (670 тысяч скачиваний по данным [2]). Однако, в отличие от BSL-синтезаторов — это лишь система поиска кода. BDA не способна к обобщению и синтезу кода, не известного ей, однако выводимого из

[3]Поисковый движок компании Microsoft, URL: `https://www.bing.com`.

уже известных данных о коде.

1.2.2. Synthesizing API usage examples (SAU)

Другая интересная альтернатива предложена в статье [3] — это алгоритм синтеза примеров кода для API библиотек Java.

Данный алгоритм принимает на вход тип (т.н. целевой тип), для которого требуется синтезировать примеры использования, и корпус кода, в котором данный тип каким-либо образом используется. Алгоритм не требует предобучения и может работать онлайн (однако, индексация корпуса кода может ускорить его работу). Сам синтез производится в несколько шагов.

На первых двух шагах производится перебор существующих в корпусе методов, использующих целевой тип, и при помощи символьного исполнения [14] строится графовая модель, представляющая различные способы работы с целевым типом. На третьем шаге производится кластеризация полученных конкретных использований целевого типа. Кластеризация производится методом k-medoids [7] на основе метрики, введенной в [3]. Для каждого из полученных кластеров синтезируется абстрактный элемент — представитель всего кластера. Наконец, на четвертом шаге производится конкретизация абстрактных представителей кластеров в итоговый код. При этом код может не быть компилируемым — обработка проверяемых исключений, инициализация переменных могут опускаться, если в действительности в корпусе обработка исключений производится выше по стеку, а инициализация различна и не важна для синтезируемого примера.

SAU показывает отличные результаты в генерации примеров (82% опрошенных не увидели разницы или предпочли бы примеры SAU примерам, написанным человеком) и обладает некоторой обобщающей способностью, однако, исходя из представленного описания, видно, что этот алгоритм чрезвычайно специализирован для синтеза примеров. В отличие от BSL-синтезаторов SAU не способен синтезировать код, использующий несколько типов, а такая поддержка потребовала бы существенной переработки всего алгоритма.

1.3. Bayou

Исследователи из группы Университета Райса также представили свою реализацию BSL-синтезатора — Bayou. Bayou является эталонной реализацией статьи [10].

Bayou — это сложная система машинного обучения, состоящая из множества алгоритмов. Она может работать в двух режимах: обучения, при котором изменяются статистические модели системы, подстраиваясь под обучающую выборку, и синтеза, при котором с помощью имеющейся статистической модели производится синтез программного кода.

На вход подается набор множеств свидетельств: по множеству (возможно, пустому) на каждый тип свидетельств.

При синтезе этот набор проходит через следующие слои обработки:

1. Evidence Embedding Layer: множество свидетельств (каждого типа по отдельности) преобразуется в числовой вектор заданной для данного типа размерности;

2. Evidence Encoder Layer: множество векторов кодируется во внутреннее представление системы: вектор пространства скрытой переменной;

3. Intent Decoder Layer: вектор из пространства скрытой переменной преобразуется в скетч посредством специального синтезатора;

4. Combinatorial Concretization Layer: скетч преобразуется в программный код с помощью случайных блужданий[4].

Заметим, что представленные слои — лишь общее описание архитектуры синтезатора. В зависимости от задачи, к которой применяется синтезатор, могут использоваться разные алгоритмы в слоях Evidence Embedding и Combinatorial Concretization, и разное множество допустимых типов свидетельств. Совокупность алгоритмов и допустимых типов свидетельств мы будем называть метамоделью BSL-синтезатора, а результат обучения конкретной метамодели — моделью BSL-синтезатора.

[4]Wikipedia, URL: `https://en.wikipedia.org/wiki/Random_walk`

На данный момент авторами Bayou представлены две метамодели, Android SDK и Java STDlib, и довольно большое число моделей для каждой из них.

Метамодель Android SDK использует три типа свидетельств: множество API вызовов, множество типов, на которых должны вызываться API вызовы, и множество типов контекста, которые должны использоваться для типов аргументов API вызовов в итоговом коде. Метамодель оптимизирована для синтеза кода, взаимодействующего с Android SDK[5], и подробно описана в [10].

Метамодель Java STDlib использует два типа свидетельств: множество API вызовов и множество типов, на которых должны вызываться API вызовы. Метамодель оптимизирована для синтеза кода, взимодействующего с Java STDlib[6] и разрабатывается исследователями в данный момент.

Теперь рассмотрим более подробно слои обработки данных Bayou.

1.3.1. Evidence Embedding Layer

Задачей Evidence Embedding Layer является преобразование множества свидетельств в числовой вектор, то есть так называемый embedding[7].

Метамодель Android SDK сначала преобразует множество свидетельств в вектор с помощью TF-IDF (term frequency–inverse document frequency) [11], затем к полученному вектору применяется LDA (Latent-Direchlet Allocation) [5]. Результирующий вектор передается на вход Evidence Encoder Layer.

Метамодель Java STDlib использует более простой Embedding — k-hot vector[8].

[5]Список входящих в Android SDK библиотек, URL: `https://developer.android.com/reference/packages.html`; список поддерживаемых метамоделью методов, URL: `https://www.dropbox.com/s/yfivipttekwdoc6/config.json`.

[6]Под Java STDlib в данном случае понимается довольно большой список пакетов вида "java.*" и "javax.*"; список поддерживаемых метамоделью методов: `https://www.dropbox.com/s/1fi1oztmv1un16t/config.json`.

[7]Embedding — вложение пространства большой размерности в меньшее с помощью специальной функции.

[8]Представление подмножества конечного множества, имеющего биективное отображение в натуральные числа от 1 до мощности множества, как вектора состоящего из нулей и единиц в индексах, соответствующих присутствующим в подмножестве элементам.

1.3.2. Evidence Encoder Layer

Полученное из Evidence Embedding Layer представление свидетельств передается в кодирующую сеть BED (Bayesian Encoders-Decoders).

Задача BED при кодировании — закодировать переданные свидетельства, представив их в виде элементов пространства скрытой переменной. Каждый тип свидетельств кодируется отдельной функцией, представляющей из себя нейронную сеть со скрытым слоем. По полученному набору свидетельств вычисляется распределение $P(Z|X)$ (распределение скрытой переменной по множеству закодированных свидетельств). Из этого распределение сэмплируется Z — скрытая переменная.

1.3.3. Evidence Decoder Layer

Задача BED при декодировании — построить по переданному элементу пространства скрытой переменной абстрактное синтаксическое дерево (AST) скетча. Для этого используются два декодера, реализованные как реккурентные нейронные сети, а именно, LSTM [15].

Вводится понятие продуцирующего пути (production path) — это путь, сходный с путем зависимостей (dependency path) в [15], но несколько упрощенный для грамматики скетчей. Продуцирующий путь — это последовательность вида $(v_1, e_1), (v_2, e_2), ...$ где v — конструкция языка (управляющая конструкция или API-вызов), а e — вид соединения со следующей вершиной (либо C — ребенок (child), либо S — брат (sibling)).

Начиная с $(root, c)$ синтезируется продуцирующий путь. На i-ом шаге сэмплируется v_{i+1} из $P(v_{i+1}, Y_i, Z)$ (где Y_i уже порожденный продуцирующий путь). Тип вершины v_{i+1} выбирается в зависимости от самой вершины. Если v_{i+1} терминальная вершина, то она будет подставлена в скетч как S. Если нетерминальная, то вершина будет подставлена и как C, и как S, и поиск будет рекурсивно продолжен в обоих направлениях.

В результате работы Evidence Decoder Layer из элемента пространства скрытой переменной получается AST скетча. Это еще не семанти-

чески полная программа, однако она уже содержит основные вызовы API и управляющие конструкции.

1.3.4. Combinatorial Concretization Layer

Получив скетч от Evidence Decoder Layer, необходимо синтезировать итоговую программу на AML (процедурное подмножество языка Java, см. [10]).

Заметим, что программа синтезируется с учетом локального контекста, а именно, имен и типов переменных, доступных в локальной области видимости. Такой синтез существенно расширяет границы применимости Bayou, позволяя применять его для автодополнения кода в IDE и синтеза частей методов.

Синтез итоговой программы из скетча проводится с помощью случайных блужданий в пространстве частично уточненных скетчей. Частично уточненный скетч — это скетч, в котором часть конструкций языка скетча заменена на конструкции языка AML.

Пусть H_i — состояние процедуры синтеза на i-ом шаге, а Env — изначальный контекст синтеза. Тогда $Next(H_i, Env)$ — это новое состояние процедуры, полученное объявлением новых переменных или использованием уже объявленных в соответствии с контекстом и типами API-вызов. Определим $P(H_{i+1}|H_i, Env)$ — некоторое (возможно даже эвристически выбранное) распределение, присуждающее ненулевую вероятность H_{i+1} тогда и только тогда, когда $H_{i+1} \in Next(H_i, Env)$. Блуждание оканчивается, когда состояние H^*, полученное в процессе синтеза — полностью конкретизированный скетч (то есть программа на AML).

Заметим, что приведенный алгоритм синтеза не является строго формальным и может быть реализован по-разному. И действительно, алгоритм конкретизации разнится в зависимости от метамодели. К примеру, Java STDlib алгоритм может заменять некоторые аргументы функций на *null* значение, в случае, если значения данного типа невозможно синтезировать в текущем контексте.

1.3.5. Технологические особенности реализации

Рассмотрев общую архитектуру Bayou и его метамоделей, уделим внимание и технологической реализации.

Bayou реализован как два web-сервера: назовем их Java-сервер и ML-сервер. Пользовательский интерфейс — это HTML-форма, в которую вносится код, в контексте которого необходимо произвести синтез. В этом же коде помечается место, в которое нужно добавить синтезированные код и описываются свидетельства. К примеру, может использоваться такой формат:

```
import java.io.*;
import org.tanvd.Evidence;
import java.util.*;

public class TestIO {
    void read(File file) {
        Evidence.calls(''readLine'');
        Evidence.types(''BufferedReader'');
        Evidence.types(''FileReader'');
    }
}
```

В данном случае будет сгенерирована тело метода read с учетом свидетельств: API-вызова ”call”, типов “FileReader” и “BufferedReader”.

Данные передаются Java-серверу (сервлет в Tomcat сервере). На нем с помощью Eclipse JDT[9] разбирается код и извлекаются свидетельства, которые далее передаются ML-серверу.

ML-сервер реализован на Python. На нем выполняется Embedding (с помощью scikit[10]), Encoding и Decoding (нейронная сеть реализована с помощью Tensorflow[11]), полученный скетч сериализуется и передается Java-серверу.

[9]Eclipse Java Development Toolkit — средства разбора и обработки языка Java, URL: `https://www.eclipse.org/jdt/`.

[10]scikit-learn — библиотека машинного обучения для Python, URL: `http://scikit-learn.org`.

[11]Tensorflow — библиотека машинного обучения для Python, URL: `https://www.tensorflow.org/`.

Java-сервер выполняет этап комбинаторной конкретизации с помощью Eclipse JDT, после чего возвращает полученный код пользователю. Пример сгенерированного кода для примера выше:

```
import java.io.*;
import java.util.*;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.FileReader;
import java.io.BufferedReader;

public class TestIO {
  void read(File file) {
    {
      FileReader fr1;
      String s1;
      BufferedReader br1;
      try {
        fr1 = new FileReader(file);
        br1 = new BufferedReader(fr1);
        while ((s1 = br1.readLine()) != null) {}
        br1.close();
      } catch (FileNotFoundException _e) {
      } catch (IOException _e) {
      }
      return;
    }
  }
}
```

1.3.6. Выводы из обзора

Эталонная реализация BSL-синтезатора Bayou представляет собой готовый сервис для синтеза кода, однако не может быть внедрена в IntelliJ IDEA как плагин без существенной переработки. IntelliJ IDEA не требует установки Python-интерпретатора и, соответственно, у большого числа пользователей интерпретатор не установлен. Таким образом слои, реализованные на Python, должны быть перереализованны на JVM-платформу.

Кроме того, текущая архитектура Bayou не поддерживает представление метамоделей как конфигураций и каждая метамодель является отдельным приложением. Разумно предоставить возможность конфигурирования метамоделей на основе единой технологической платформы, предоставляющей все необходимые алгоритмы.

Наконец, учитывая то, что модели для существующих метамоделей продолжают улучшаться коллективом исследователей из Университета Райса, разумно предусмотреть совместимость моделей Bayou с реализованным для JVM-платформы BSL-синтезатором.

2. Архитектура конфигурируемого BSL-синтезатора

Как было отмечено выше, BSL-синтезатор может быть представлен как набор слоев трансформаций, преобразующих входные свидетельства в итоговый код. В соответствии с подходом, предложенным в [16], были выделение следующие слои:

- Evidence Embedding Layer,
- Evidence Encoder Layer,
- Intent Decoder Layer,
- Combinatorial Concretization Layer.

Кроме того, мы определили понятие метамодели — некоторой параметризации набора слоев конкретными алгоритмами и видами входов.

В отличие от реализации Bayou, мы хотим построить конфигурируемый BSL-синтезатора — то есть BSL-синтезатор, способный работать в соответствии с разными метамоделями. Используя введенные ранее абстракции, это несложно сделать: отобразим существующие слои абстрактного BSL-синтезатора на слои реализации. Каждый слой реализации может конфигурироваться различными алгоритмами, а набор параметризаций слоев и образует метамодель.

Полная Data Flow диаграмма конфигурируемого BSL-синтезатора представлена на Рис. 1.

2.1. Input Layer

Задачей Input Layer является обработка входа (фрагментов кода со свидетельствами), то есть извлечение свидетельств из кода и создание некоторого промежуточного представления для передачи Concretization Layer для вставки итогового кода.

Данный слой отсутствует в описании абстрактного BSL-синтезатора и выполняет скорее технологические функции — выделив обработку

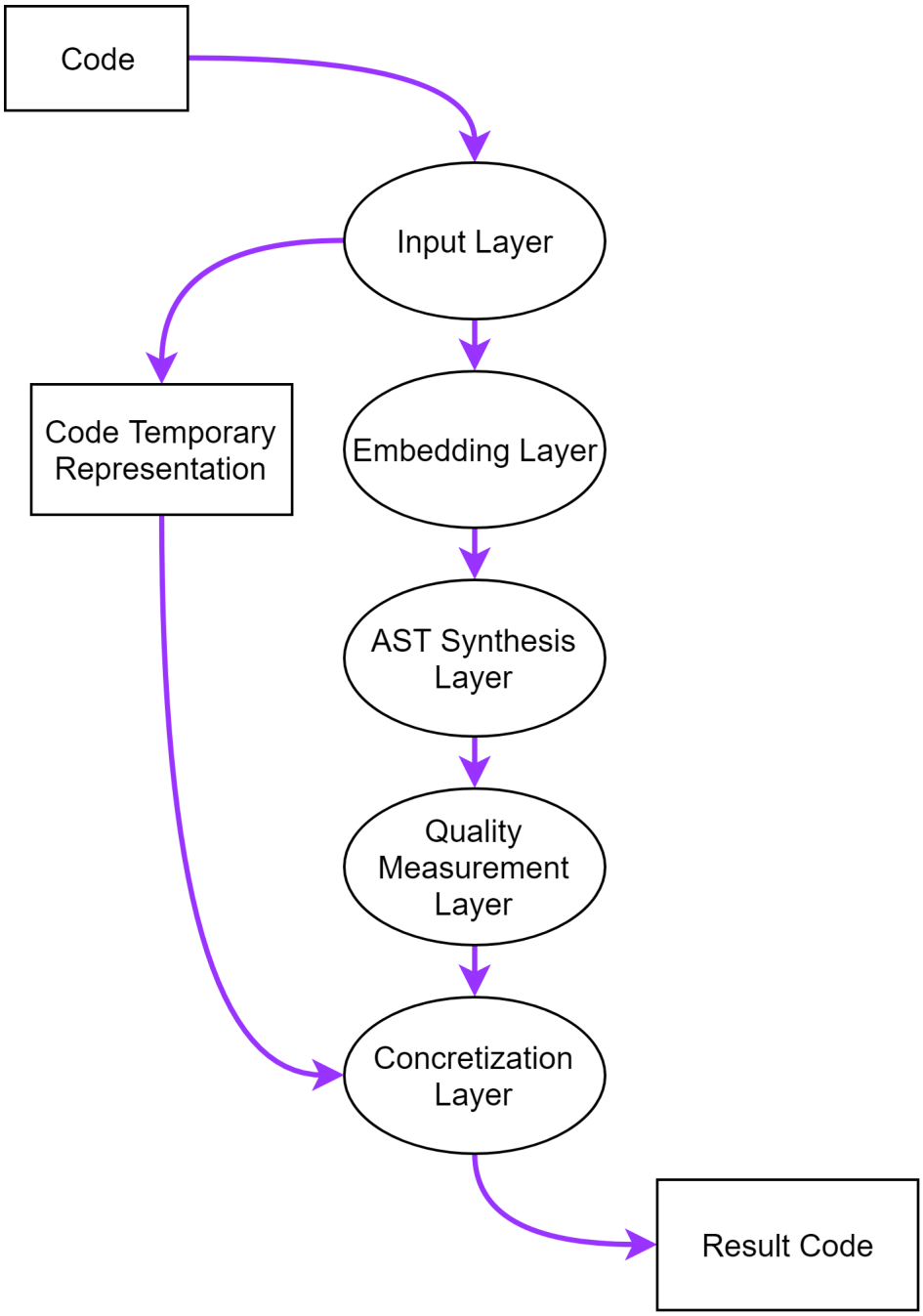

Рис. 1: Data Flow диаграмма конфигурируемого BSL-синтезатора

входа в отдельный слой, мы можем использовать разные представления свидетельств и даже не внедрять их в код, а передавать вместе с кодом и номером строки для вставки синтезированного кода.

2.2. Embedding Layer

Задачей Embedding Layer является embedding свидетельств, переданных Input Layer. Embedding Layer поддерживает алгоритмы, требуемые разными метамоделями (TF-IDF и LDA для Android SDK и k-hot для Java STDlib).

Данный слой соответствует Evidence Embedding Layer в описании абстрактного BSL-синтезатора. Он параметризуется типами свидетельств, принимаемых на входе, и типами embedding для каждого из типов свидетельств. В случае, если embedding требует некоторые дополнительные данные (модель для LDA, словарь для k-hot), они также передаются в конфигурации.

2.3. AST Synthesis Layer

Задачей AST Synthesis Layer является синтез AST скетчей по векторам, представляющим свидетельства. AST Synthesis Layer может параметризовываться количеством скрытых слоев кодировщика и декодировщика, однако это не используется в существующих метамоделях.

Данный слой соответствует слоям Evidence Encoder Layer и Intent Decoder Layer, то есть в рамках AST Synthesis Layer производится кодировка свидетельств в элемент пространства скрытой переменной и декодировка из элемента пространства скрытой переменной в скетч. Два слоя объединены в один, так как изменение параметров или модели одного слоя невозможно без соответствующих изменений другого слоя. Параметризуется слой моделью BED.

2.4. Quality Measurement Layer

Задачей Quality Measurement Layer является оценка качества синтезированных скетчей, их дедупликация и ранжирование.

Данный слой также отсутствует в описании абстрактного BSL-синтезатора и требуется для выделения алгоритмов оценки качества в отдельную абстракцию. Таким образом мы можем использовать разные алгоритмы ранжирования для разных метамоделей и моделей, или даже для одних и тех же, предоставляя возможность выбора метрик качества пользователю.

2.5. Concretization Layer

Задачей Concretization Layer является синтез Java кода по полученным от Quality Measurement Layer скетчам и представлению контекста синтезируемого кода, полученного от Input Layer.

Данный слой соответствует Combinatorial Concretization Layer в описании абстрактного BSL-синтезатора. Он может параметризовываться алгоритмом конкретизации (к примеру, глубиной рекурсивного поиска при порождении аргументов методов), однако существующие метамодели используют настройки конкретизатора по умолчанию.

Заметим, что конкретизация производится с учетом контекста окружающего кода, и именно поэтому Concretization Layer принимает на вход не только AST, но и промежуточное представление исходного кода, в контексте которого и производится синтез.

Результатом Concretization Layer является исходный код с добавленным в него блоком синтезированного кода.

Предложенная архитектура BSL-синтезатора видится достаточно гибкой для того, чтобы описать две существующие на данный момент метамодели. Кроме того, были выделены в отдельные абстракции алгоритмы извлечения свидетельств и измерения качества, что позволит изменять их, не затрагивая слои, непосредственно относящиеся к синтезу.

3. Реализация конфигурируемого BSL-синтезатора

Рассмотрим реализацию приведенной архитектуры. Начнем с описания того, как для конфигурируемого BSL-синтезатора были получены модели.

3.1. Экспорт моделей

Как уже упоминалось ранее, исследователи из Университета Райса продолжают развитие Bayou и регулярно выпускают улучшенные модели. Соответственно, разумно переиспользовать уже существующие модели Bayou и поддержать совместимость с новыми моделями.

Экспорт моделей производится из runtime Bayou, путем внедрения функций экспорта моделей в исходный код Bayou. Кроме как из runtime выгрузить модели невозможно, так как многие из них (к примеру, модели TF-IDF) извлекаются из объектов с помощью рефлексии. Модели экспортируются в независящем от языка программирования формате и после используются нашим BSL-синтезатором.

Для Embedding Layer в Android SDK требовалось экспортировать модель LDA и модель TF-IDF из Scikit-learn объектов для каждого типа свидетельств. В случае TF-IDF мы экспортируем словарь и IDF матрицу [11]. В случае LDA экспортируются α, η коэффициенты и матрица ϕ [5].

Для Embedding Layer в Java STDlib потребовалось экспортировать только словарь k-hot vector.

Для AST Synthesis Layer в обеих метамоделях требовалось экспортировать Tensorflow модель. Для этого мы в runtime именуем переменные и выходные тензоры Tensorflow модели (иначе к ним будет сложно получить доступ при загрузке модели) и сохраняем модель с помощью SaveModel функциональности Tensorflow.

3.2. Реализация слоев

Рассмотрим реализацию каждого из слоев синтезатора.

3.2.1. Input Layer

Данный слой имеет две реализации. Обе реализации используют Eclipse JDT для разбора кода и промежуточного его представления (в виде CompilationUnit).

Первая реализация аналогична предложенной в [10]. В ней “свидетельства” передаются напрямую в коде в виде вызовов функций библиотеки. К примеру:

```
import java.io.*;
import org.tanvd.Evidence;
import java.util.*;

public class TestIO {
    void read(File file) {
        Evidence.calls(''readLine'');
        Evidence.types(''FileReader'');
    }
}
```

С помощью Eclipse JDT код разбирается, извлекаются свидетельства, а место, где они находились, помечается для последующей вставки кода. Сами свидетельства из кода удаляются.

Вторая реализация принимает строку с кодом (без свидетельств), сами свидетельства и позицию для вставки синтезированного кода. Этот интерфейс удобен для программного взаимодействия и несколько более эффективен, так как не требуется извлекать свидетельства из кода.

3.2.2. Embedding Layer

Данный слой имеет две реализации — k-hot для Java STDlib метамодели и TF-IDF + LDA для Android SDK. Вообще говоря, для каждого

типа свидетельств может использоваться разный тип embedding, однако для реализации текущих метамоделей такого не требовалось.

Все Embedding параметризуются экспортированными ранее моделями и соответствуют спецификациям алгоритмов Scikit-learn, используемых в Bayou.

TF-IDF реализован простейшим образом — подсчет TF, умножение на IDF и нормализация. LDA реализован несколько сложнее, в соответствии с [5]. Реализация K-hot опять же тривиальна — проход по всем свидетельствам и отметка присутствующих в соответствующих элементах массива.

Размерность выхода каждого из Embedding определяется используемой моделью (к примеру, размерность k-hot будет соответствовать размерности вектора словаря).

3.2.3. AST Synthesis Layer

Данный слой имеет единственную реализацию. Он параметризуется экспортированной Tensorflow моделью. Сама модель загружается и исполняется с помощью Tensorflow for Java.

На стадии кодирования полученные от Embedding Layer вектора свидетельств передаются на соответствующие входы кодировщика BED, после чего Tensorflow модель запускается на исполнение. Результатом является элемент пространства скрытой переменной ψ.

На стадии декодирование полученный ранее ψ передается декодировщику BED (опять же модели Tensorflow). В ответ декодировщик отдает вершину — начало продуцирующего пути, обсуждавшегося в обзоре. Далее в соответствии с описанным алгоритмом строится весь продуцирующий путь. При этом, в зависимости от порожденного узла, построение продуцирующего пути может разветвляться в нескольких направлениях. К примеру, узел ”DBranch” (то есть узел конструкции ”if”) приведет к построению путей для предиката, ветвей ”then” и ”else”. Всего же существующие модели поддерживают ”if”, ”while” и ”try ... catch” конструкции управления скетчей.

Результатом работы AST Synthesis Layer является множество AST скетчей, их количество можно задавать в конфигурации.

3.2.4. Quality Measurement Layer

Данный слой имеет несколько реализаций и их количество может продолжать расти. На данный момент для всех метамоделей используются следующие алгоритмы — верификация присутствия всех свидетельств, дедупликация и ранжирование по частоте повторения AST.

Верификация присутствия всех свидетельств обходит AST и собирает информацию о производящихся в нем API-вызовах, типах и контексте (типах аргументов). Полученные множества пересекаются с изначальными свидетельствами, и если какое-то свидетельство отсутствует в AST, то AST удаляется из результирующего множества.

Дедупликация удаляет повторяющиеся AST, а оставшемуся ставит в соответствие количество его повторений в изначальном множестве. Количество повторений используется при ранжировании: чем чаще AST встречалось, тем вероятнее, что оно подходит под заданный запрос.

Однако, этот наивный алгоритм ранжирования не единственен. Ранжировать также можно, используя стандартные метрики кода, такие как *LOC* (lines of code) или цикломатическую сложность [9]. В зависимости от ситуации эти метрики могут дать более релевантный результат (например, в случае генерации примеров, исходя из исследования в [3], предпочтительны более короткие примеры)

3.2.5. Concretization Layer

Наконец, рассмотрим Concretization Layer — слой итогового синтеза программы. Данный слой имеет единственную реализацию, основанную на Eclipse JDT.

Concretization Layer принимает на вход не только AST, но и промежуточное представление окружающего кода, в нашем случае в виде CompilationUnit. В контексте данного CompilationUnit и выполняется алгоритм, подробно описанный в обзоре. Перебор выполняется в по-

рядке, определяемом эвристикой. К примеру, функции без аргументов рассматриваются раньше, чем функции с аргументами, так как функции с аргументами влекут дальнейший поиск.

Отдельно заметим, что реализация позволяет использовать методы, некоторые аргументы которых не могут быть порождены в текущем контексте. В таких случаях, создается переменная данного типа и ей присваивается значение *null*. Предполагается, что программист заменит её на соответствующую инициализацию.

Кроме того, из результирующего кода удаляется недостижимый и не влияющий на результат функции код (т.н. Dead Code Elimination), код форматируется, полные имена классов преобразуются в import выражения.

Результирующий код возвращается синтезатором в порядке релевантности соответствующих AST.

Полученная реализация конфигурируемого BSL-синтезатора полностью исполняется на JVM-платформе и способна исполнять обе существующие на данный момент метамодели. Решение поддержать совместимость моделей с Bayou позволило обновлять модели одновременно с ним и таким образом поддерживать качественные характеристики на уровне с эталонной реализацией.

4. Реализация плагина к IntelliJ IDEA

Рассмотрим создание плагина к IntelliJ IDEA с пользовательским интерфейсом для BSL-синтезатора. Рассмотрение данного вопроса нужно начать с описания интеграции самого BSL-синтезатора с IntelliJ IDEA, то есть обеспечения его работоспособности в рамках платформы IntelliJ IDEA.

4.1. Интеграция BSL-синтезатора с IntelliJ IDEA

При интеграции требовалось решить две основных проблемы — получение моделей для синтезатора и отслеживание прогресса выполнения.

Как уже упоминалось ранее, BSL-синтезатору для работы требуются модели. Распространять их вместе с плагином нельзя, так как они крайне объемны (Android SDK — 100 мегабайт, Java STDlib — 200 мегабайт), а пользователю может потребоваться только одна модель. Соответственно, потребовалось реализовать сетевой репозиторий для моделей. В данный сетевой репозиторий (на данный момент все файлы выложены на Amazon S3[12]) выкладывается файл-дескриптор репозитория и сами файлы моделей. В дескрипторе репозитория перечисляются все существующие модели, пути до отдельных файлов каждой модели в сетевом репозитории и MD5 хэши этих файлов.

На стороне пользователя создается локальный репозиторий — директория с моделями и дескриптором локального репозитория. При запросе модели проверяется, присутствует ли данная модель в локальном репозитории, и совпадают ли хэши файлов существующей модели с хэшами, указанными в сетевом репозитории. В случае, если модель отсутствует или файлы повреждены, нужные файлы скачиваются с сетевого репозитория.

Кроме того, сам синтез — довольно длительная процедура, занимающая от 1 до 10 секунд, а загрузка файлов моделей может занимать

[12]Amazon S3 — объектное хранилище от Amazon, URL: `https://aws.amazon.com/ru/s3`.

и ещё больше времени. Отсутствие возможности отслеживать прогресс длительных процессов может существенно негативно повлиять на пользовательский опыт работы с плагином. Поэтому была внедрена система отслеживания прогресса.

Интерфейс BSL-синтезатора принимает объект, позволяющий отслеживать прогресс (ProgressIndicator). Это обычный Data Class, полям которого можно присвоить имя текущего выполняемого процесса (к примеру, “Generating Sketches”, “Downloading TF-IDF Model” и т.п.) и прогресс это процесса от 0.0 до 1.0. Таким образом, можно создать этот объект в потоке взаимодействия с пользователем и передать его в поток синтеза, а дальше показывать все обновления данного объекта в потоке взаимодействия с пользователем.

Все задачи синтеза запускаются в IntelliJ IDEA в блокирующем режиме: пользовательский интерфейс отображает только текущее состояние индикатора прогресса, а все остальные элементы управления блокируются.

4.2. Реализация пользовательского интерфейса

Пользовательский интерфейс к плагину в IntelliJ IDEA можно реализовывать по разному: и как набор оконных форм, и как предметно-ориентированный язык, распознаваемый в комментариях, и как некоторые конструкции внутри кода. Было решено применить два подхода: аннотации метода, определяющие требуемые свидетельства (такой подход применяет, к примеру, известная Java-библиотека Project Lombok[13]) и отдельный предметно-ориентированный язык в комментариях.

4.2.1. Аннотирование метода

Была реализована Java библиотека “bayou-annotations” содержащая следующие аннотации:

[13]Project Lombok — библиотека синтезатора утилитарных методов для Java программ, URL: `https://projectlombok.org/`.

- BayouSynthesizer — аннотация, определяющая метамодель, которую нужно использовать при синтезе;
- ApiCall — аннотация, определяющая свидетельство API-вызова;
- ApiType — аннотация, определяющая свидетельство типа, на котором должен быть вызван метод ApiCall;
- ContextType — аннотация, определяющая свидетельство типа аргумента одного из ApiCall.

Рассмотрим запрос к синтезатору, передающий свидетельство типа "ApiCall" со значением "readLine" и свидетельство типа "ApiType" со значением "FileReader". Приведем описание данного запроса с помощью аннотаций:

```
import tanvd.annotations.*;

import java.io.File;
import java.io.FileReader;

public class TestIO {
    @BayouSynthesizer(type = SynthesizerType.StdLib)
    @ApiCall(name = ''readLine'')
    @ApiType(name = FileReader.class)
    void read(File file) {
    }
}
```

Заметим, что ApiType и ContextType принимают объект типа Class<T> — класс класса в Java и для них производится Auto-Completion в IntelliJ IDEA. А вот ApiCall в силу ограничений Java нельзя передать Function<T>. Соответственно, в ApiCall передается либо Enum, предварительно созданный и описывающий все возможные API-вызовы текущей модели, либо String. Для Java STDlib Enum создать не получится, так как количество доступных API-вызовов превышает максимальный размер Enum.

Еще одной проблемой такого подхода является отсутствие строгой типизации. К примеру, возможно указать BayouSynthesizer как StdLib (Java STDlib метамодель) и при этом использовать ContextType аннотацию, не поддерживаемую данной метамоделью.

Таким образом, аннотирование метода, несмотря на привычность для Java программистов, обладает существенными недостатками и должно использоваться лишь программистами, хорошо знакомыми с системой синтеза.

4.2.2. Предметно-ориентированный язык

С использованием Grammar Kit[14] был реализован предметно-ориентированный язык описания запросов к синтезатору. В нем присутствуют следующие идентификаторы:

- STDLIB или ANDROID — начальный идентификатор, определяет метамодель, которую нужно использовать при синтезе;
- API — идентификатор, определяющий свидетельство API-вызова;
- TYPE — идентификатор, определяющий свидетельство типа, на котором должен быть вызван метод ApiCall;
- CONTEXT — идентификатор, определяющий свидетельство типа аргумента одного из ApiCall.

Снова рассмотрим запрос к синтезатору, передающий свидетельство типа "ApiCall" со значением "readLine" и свидетельство типа "ApiType" со значением "FileReader. Теперь приведем описание данного запроса с помощью предметно-ориентированного языка:

```
import java.io.File;
import java.io.FileReader;

public class TestIO {
```

[14]Grammar Kit — инструментарий поддержки создания языков для IntelliJ IDEA, URL: https://github.com/JetBrains/Grammar-Kit.

```
    /*
    STDLIB
    API:=readLine
    TYPE:=FileReader
    */
    void read(File file) {
    }
}
```

Предметно-ориентированный язык хорошо интегрируется с IntelliJ IDEA. Присутствует строгая типизация (к примеру, CONTEXT нельзя использовать в случае, если выбрана метамодель Java STDlib), Auto-Completion для значений всех типов свидетельств и аннотирование ошибок (возможность определять сообщения для некоторых конкретных ошибок в грамматике).

Полученный язык достаточно прост, а строгая типизация и подсказки для ошибок позволяют освоиться с языком запросов без дополнительной документации.

5. Апробация

Для полученного плагина была проведена апробация на примерах, приведенных в [10] для Android SDK и примерах, аналогичных приведенным на сайте `http://www.askbayou.com/` для Java STDlib. Всего было взято 20 задач синтеза для Android SDK (к примеру, код работающий с “BluetoothSocket”, код использующий “File” и т.д.) и 20 задач синтеза для Java STDlib (к примеру, код работающий с классами коллекций, “File” и различными “Reader”). Все задачи синтеза запускались на последних сборках соответствующих Bayou и на предложенной системе. Во всех случаях синтезаторы выдали одинаковые результаты.

Рассмотрим несколько примеров.

```
public class TestIO {
    /*
    STDLIB
    API:=readLine
    TYPE:=FileReader
    TYPE:=BufferedReader
    */
    void read(File file) {
    }
}
```

```
import java.io.File;
import java.io.FileReader;

public class TestIO {
    void read(File file) {
        FileReader fr1;
        String s1;
        BufferedReader br1;
        try {
            fr1 = new FileReader(file);
            br1 = new BufferedReader(fr1);
            while ((s1 = br1.readLine()) != null) {}
            br1.close();
        } catch (FileNotFoundException _e) {
        } catch (IOException _e) {
        }
        return;
    }
}
```

Листинг 1: Шаблон (слева) и сгенерированная по нему программа, производящая буферизованное чтение файла (справа)

```
public class TestDelete {
    /*
    STDLIB
    API:=delete
    TYPE:=File
    */
    void delete(String file) {
    }
}
```

```
import java.io.File;

public class TestIO {
    void delete(String file) {
        File f1;
        f1 = new File(file);
        boolean b1;
        b1 = f1.delete();
        return;
    }
}
```

Листинг 2: Шаблон (слева) и сгенерированная по нему программа, производящая удаление файла (справа)

```
public class TestRemove {
    /*
    STDLIB
    API:=remove
    TYPE:=Iterator
    */
    void remove(List<String> list) {
    }
}
```

```
import java.util.Iterator;
import java.util.List;

public class TestRemove {
    void remove(List<String> list) {
        Iterator<String> i1;
        boolean b1;
        i1 = list.iterator();
        while ((b1 = i1.hasNext())) {
            i1.remove();
        }
        return;
    }
}
```

Листинг 3: Шаблон (слева) для и сгенерированная по нему программа, производящая удаление всех элементов списка (справа)

Как мы видим, синтезатор способен синтезировать достаточно сложные методы и может использоваться для синтеза утилитарных функций для работы с библиотеками.

5.1. Границы применимости

Разумеется, область применения BSL-синтезаторов ограничена.

Во-первых, BSL-синтезаторы не способны синтезировать код, являющийся порождением двух метамоделей одновременно: то есть нельзя синтезировать код, использующий часть методов, присутствующих только в Android SDK, и часть методов, присутствующих только в Java STDlib. Единственный вариант — построить новую метамодель и модель, охватывающие две предыдущие. Однако, на данный момент нет исследований зависимости качества синтеза от пространства синтезируемых программ. Исходя из здравого смысла, можно предположить, что качество будет падать.

Во-вторых, на данный момент BSL-синтезаторы подходят для синтеза только методов или даже отдельных их частей. С помощью них на данный момент нельзя синтезировать классы и, соответственно, нельзя производить полностью автоматизированный синтез программных проектов. Исследований относительно возможности синтеза классов (фактически, автоматизированного проектирования) с помощью BSL-синтезаторов на данный момент не существует.

Однако, BSL-синтезаторы отлично подходят для своей основной области применения — синтеза методов с большим количеством API-вызовов, что и было подтверждено в процессе апробации.

Заключение

В ходе выполнения данной работы были достигнуты следующие результаты:

- была предложена архитектура конфигурируемого BSL-синтезатора, позволяющего выполнять существующие метамодели и добавлять новые;
- конфигурируемый BSL-синтезатор был реализован на JVM-платформе, что позволило интегрировать его с IntelliJ IDEA;
- был создан плагин, предоставляющий интерфейс к BSL-синтезатору и предложены два пользовательских интерфейса: аннотирование методов и предметно-ориентированный язык;
- была проведена апробация, продемонстрирована эквивалентность полученного BSL-синтезатора эталонному, и определены границы применимости полученного BSL-синтезатора.

В результате данной работы был создан плагин для IntelliJ IDEA, предоставляющий принципиально новые возможности синтеза кода в данной IDE. Кроме того, конфигурируемый BSL-синтезатор, реализованный как часть данной работы, может использоваться в работе [4] как синтезатор из промежуточного представления цепочки API-вызовов в итоговый код, и таким образом представить инструмент, синтезирующий естественный язык в код.

Список литературы

[1] Barber David. Bayesian reasoning and machine learning. — Cambridge University Press, 2012. — ISBN: 978-0521518147.

[2] Bing Developer Assistant: Improving Developer Productivity by Recommending Sample Code / Hongyu Zhang, Anuj Jain, Gaurav Khandelwal et al. // Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. — FSE 2016. — New York, NY, USA : ACM, 2016. — P. 956–961.

[3] Buse Raymond P. L., Weimer Westley. Synthesizing API Usage Examples // Proceedings of the 34th International Conference on Software Engineering. — ICSE '12. — Piscataway, NJ, USA : IEEE Press, 2012. — P. 782–792.

[4] Deep API Learning / Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, Sunghun Kim // CoRR. — 2016. — Vol. abs/1605.08535. — 1605.08535.

[5] Hoffman Matthew, Bach Francis R., Blei David M. Online Learning for Latent Dirichlet Allocation // Advances in Neural Information Processing Systems 23 / Ed. by J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor et al. — Curran Associates, Inc., 2010. — P. 856–864.

[6] Inductive Programming Meets the Real World / Sumit Gulwani, José Hernández-Orallo, Emanuel Kitzelmann et al. // Communications of the ACM. — 2015. — oct. — Vol. 58, no. 11. — P. 90–99.

[7] Kaufman L., Rousseeuw P. Finding Groups in Data: An Introduction to Cluster Analysis. — Wiley Interscience, 2005. — P. 68–125. — ISBN: 9780471878766.

[8] Kingma Diederik P, Welling Max. Auto-Encoding Variational Bayes. — 2013. — arXiv:1312.6114.

[9] McCabe T. J. A Complexity Measure // IEEE Transactions on Software Engineering. — 1976. — Dec. — Vol. SE-2, no. 4. — P. 308–320.

[10] Murali Vijayaraghavan, Chaudhuri Swarat, Jermaine Chris. Bayesian Sketch Learning for Program Synthesis // CoRR. — 2017. — Vol. abs/1703.05698. — 1703.05698.

[11] Rajaraman A., Ullman J.D. Mining of Massive Datasets. — Cambridge University Press, 2012. — P. 1–17. — ISBN: 978-1-139-05845-2.

[12] Robillard Martin P. What makes APIs hard to learn? Answers from developers // IEEE software. — 2009. — Vol. 26, no. 6.

[13] Solar Lezama Armando. Program Synthesis By Sketching : Ph. D. thesis : UCB/EECS-2008-177 / Armando Solar Lezama ; EECS Department, University of California, Berkeley. — 2008. — Dec.

[14] Baldoni Roberto, Coppa Emilio, D'Elia Daniele Cono et al. A Survey of Symbolic Execution Techniques. — 2016. — arXiv:1610.00502.

[15] Xingxing Zhang Liang Lu Mirella Lapata. Top-down Tree Long Short-Term Memory Networks // Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies. — 2016. — June. — P. 310–320.

[16] Ксезнов Михаил. Рефакторинг архитектуры программного обеспечения: выделение слоев. — 2004. — URL: `http://citforum.ru/SE/project/refactor/` (online; accessed: 17.12.2016).