Data-based code synthesis in IntelliJ IDEA

Vladislav Tankov
Saint-Petersburg State University
JetBrains
Email: vdtankov@gmail.com

Timofey Bryksin
Saint-Petersburg State University
JetBrains Research
Email: t.bryksin@spbu.ru

***Abstract*—Automatic code synthesis has been attracting more attention lately. Some recent papers in this traditionally academic field even present results that could be applicable for industrial programmers. This paper provides an overview of Bayesian Sketch Learning (BSL) approach, describes basic concepts and workflow of a BSL synthesizer. Based on this we discuss an architecture of a configurable BSL synthesizer that could work as a part of an integrated development environment. We describe the implementation of such synthesizer for JVM platform and its integration with IntelliJ IDEA as a plugin. Two approaches to implement user interaction in a plugin like this are presented: method annotations and a domain-specific language. The paper concludes with an evaluation and a discussion on limitations of selected approach for industrial programmers.**

Introduction

Automatic code synthesis has been a field of interest of computer science and software engineering for decades. It is defined as a task of synthesizing an algorithm in certain programming language based on incomplete specifications. Depending on the approach, this specification can be a definition of an algorithm in some domain specific language (DSL), a set of inputs and outputs of the algorithm, a set of system calls that occur when the algorithm is executed, etc.

Recently, more and more researchers have been addressing this task, and even first industrial code synthesizers begin to appear on the market [1]. However, most modern code synthesis tools are poorly applicable, since they usually require programmers to master some new formalism, almost always alienated from programmers' main field of knowledge. For example, study a separate specification language.

In this regard, Bayesian Sketch Learning approach (BSL approach), proposed by a group of researchers from the University of Rice in [2], is fundamentally different. BSL approach allows you to use identifiers of a programming language and its libraries as domain specific language. It narrows the gap between a specification language used by a code synthesizer and a synthesized code used by programmers. This approach has a significant impact: since the synthesizer works with nothing else than the programming language and its libraries, it can be used within an integrated development environment (IDE) and fit in the usual patterns of work with this IDE. Moreover, using such a synthesizer, an IDE could offer more intelligent code completion and even try to synthesize parts of developed systems.

This paper describes the implementation of a BSL synthesizer for JVM platform and its integration with IntelliJ IDEA. The reminder of this paper is structured as follows. Section I provides an overview of BSL synthesizer's concepts, describes several alternative approaches to code synthesis and examines the architecture of Bayou which is the baseline implementation of the BSL approach. Section II presents an architecture of a configurable BSL synthesizer that could work as a part of an IDE. Section III describes implementation of such a synthesizer and its compatibility with Bayou models. Section IV explores challenges arising while implementing the synthesizer as an IntelliJ IDEA's plugin. Section V provides evaluation of suggested implementation.

I. Overview

A. Overview of BSL synthesizer's concepts

BSL synthesizer is a code synthesizer based on Bayesian Learning approach [3]. The essence of Bayesian approach in this case boils down to the following: a synthesizer is trained on a corpus of programs and so called "evidence" (some values associated with every program). After that, getting some set of evidence, it tries to synthesize code that most likely satisfies this evidence in context of the whole corpus. So, on the learning step an a priori distribution of programs is calculated for a given set of evidence, and on the synthesis step an a posteriori distribution is calculated for a specific evidence.

Evidence for a BSL synthesizer is usually method calls and classes used in the synthesized function. Experiments in [2] showed high effectiveness of BSL synthesizers for API-heavy code (code with a large number of API calls). Efficiency of BSL synthesizers in arbitrary code synthesis tasks has not been investigated yet.

BSL synthesizer uses sketches [4] as internal representation of programs. A sketch is a simplified representation of a program that consists only of basic language constructs (such as control flow and, in case of BSL synthesizers, API calls). Sketches do not preserve semantics of the program (which, nevertheless, can be recovered using probabilistic methods, see [2] for details), but they represent programs with similar intent in a uniform manner. Sketches are associated with evidence using Bayesian encoder-decoder (BED) techniques [2] (fairly close to variational autoencoders [5]). An evidence is converted into an element of intent latent variable space (“encoded”), and this element is converted (“decoded”) into a sketch corresponding to the given evidence.

Thus, from the probability theory’s point of view, following calculations take place.

1) Let X be a collection of evidence, Z is an intent latent variable, corresponding to X, Y is a sketch corresponding to Z.
2) Calculate $P(Z|X)$ — a distribution of the hidden variable by X.
3) Sample Z in accordance with the distribution obtained.
4) Calculate $P(Y|Z)$ — a sketch distribution by Z.
5) Sample Y in accordance with the distribution obtained.

By combining Bayesian approach with representation of programs as sketches, BSL synthesizers can correctly synthesize fairly complex API-heavy methods.

B. Related works

Currently there are several tools that are capable of synthesizing API-heavy methods provided with evidence.

1) Bing Developer Assistant (BDA): Bing Developer Assistant [6] is a system for searching code samples or even entire projects corresponding to natural language queries. BDA consists of a Visual Studio plugin that consists of a user interface (frontend), and a cloud-based platform that provides search capabilities (backend). The frontend part uses natural language as a way of communicating with its users (for example, queries like “how to save png image”). After receiving response from the backend, BDA is able to insert code into current user’s project, open a separate window with a code sample, or even offer GitHub projects matching with the current request.

To search code using natural language queries the backend part uses Bing[1], restricting itself to a limited set of sites. A framework proposed in [6] extracts code pieces from Bing search result pages and ranks them. Then code samples are passed to the frontend.

BDA is a successful tool to search code with 670 thousand downloads according to [6]. However BSL synthesizers are not just code search systems. BDA is not capable of generalizing and synthesizing code not known to it, but deducable from already known data.

2) Synthesizing API usage examples (SAU): Another interesting alternative is proposed in [7] — an algorithm for synthesizing code samples for Java Standard Library API. This algorithm takes an input type T (so called target type) for which it is required to synthesize usage examples, and code corpus where this type is used in some way. Synthesis is performed in several steps.

[1]Bing: a web search engine by Microsoft, URL: https://www.bing.com

Initially existing methods using type T are enumerated, and via symbolic execution [8] a graph model is constructed, representing different ways of working with type T. Then, obtained use cases of type T are clustered. Clustering is performed using k-medoids method [9] based on the metric proposed in [7]. For each of the clusters obtained, an abstract element is synthesized — a representative of this entire cluster. Finally, all these abstract representatives are concretized into final code. This code might not be compilable: catching of checked exceptions or initialization of variables may be omitted, if exception handling in code corpus is usually performed outside of functions using type T or initialization is not important for the sake of synthesized examples.

SAU shows excellent results of synthesizing examples (82% of respondents did not see any difference between SAU generated code and human written code or even would prefer SAU examples) and has a decent generalizing capability. However, it is obvious that this algorithm is highly specialized for code examples synthesis. Unlike BSL synthesizers, SAU is not able to synthesize code that uses several types, and such support would require fundamental rework of the entire algorithm.

C. Bayou

Researchers from the Rice University also proposed their implementation of the BSL synthesizer, Bayou [2]. Bayou is a complex machine learning system employing several algorithms. It work in two modes: learning mode, when a statistical model is obtained, adjusting to a training sample, and synthesis mode, when code synthesis is performed in accordance with the statistical model.

Input is a set of sets of evidence — a set (possibly empty) for each type of evidence. The implementation supports three types of evidence:

- ApiCall — call of some API method (for example, “readLine”);
- ApiType — usage of some library class (for example, “BufferedReader”);
- ContextType — usage of some class as method argument.

During synthesis evidence is being passed through the following transformation layers:

1) *Evidence Embedding Layer*: evidence sets (for each evidence type separately) are converted into a numeric vector;
2) *Evidence Encoder Layer*: obtained set of vectors is encoded into an element of intent latent variable space (also a numeric vector);
3) *Intent Decoder Layer*: obtained element is converted into a sketch;
4) *Combinatorial Concretization Layer*: the sketch is turned into code using random walk technique.

Evidence Encoder and Intent Decoder layers together form Bayesian Encoder-Decoder (BED).

These layers are only a general description of the synthesizer's architecture. Depending on the task and its domain, different algorithms can be used within Evidence Embedding and Combinatorial Concretization layers, acceptable evidence types also might differ.

We will call a set of algorithms and acceptable evidence types a *metamodel* of the BSL synthesizer, and the result of learning a particular metamodel is a *model* of the BSL synthesizer. Currently there are two metamodels, for Android SDK and for Java STDlib, and a fairly large number of models for each of them.

Android SDK metamodel uses three types of evidence: ApiCall, ApiType and ContextType. This metamodel is optimized for models trained on Android SDK library and is described in detail in [2]. Java STDlib metamodel uses two types of evidence: ApiCall and ApiType. This metamodel is optimized for models trained on Java STDlib and is under development by researchers at this moment.

Now let us consider Bayou's implementation. Bayou is implemented as two web servers: a Java server and a machine learning (ML) server. Input code is passed to the Java server (a servlet within a Tomcat web server), where it is parsed using Eclipse JDT[2] and a set of evidence is retrieved. Then this set is passed to the ML server, which is implemented in Python. It runs embedding (using Scikit-learn[3]), encoding and decoding steps (neural networks are implemented with Tensorflow[4]). The resulting sketch is serialized and passed back to the Java server, where Combinatorial Concretization via Eclipse JDT is performed, and synthesized code is returned to the user.

Thus, reference implementation of Bayou's BSL synthesizer is a ready-to-use web service for code synthesis, but it can not be integrated into IntelliJ IDEA as a plugin without significant rework. IntelliJ IDEA does not require installation of a Python interpreter, so most users might not have it installed. All layers implemented in Python must be re-implemented on the JVM platform. Moreover, current Bayou architecture does not support configuration of metamodels, each metamodel is implemented as a separate application.

II. Architecture of the configurable BSL synthesizer

Unlike Bayou's implementation, we want to build a configurable BSL synthesizer — a BSL synthesizer capable of running synthesis according to different metamodels. We provide a mapping of existing abstract BSL synthesizer's layers to layers of our implementation. Each layer of our implementation can be parameterized with different algorithms, and such set of parameterizations of all layers, as we defined above, forms a metamodel.

[2]Eclipse Java Development Toolkit: a tool for parsing and processing Java code, URL: https://www.eclipse.org/jdt/.

[3]Scikit-learn: a machine learning library for Python, URL: http://scikit-learn.org

[4]Tensorflow: a machine learning library for Python, URL: https://www.tensorflow.org/

Complete Data Flow diagram of the suggested configurable BSL synthesizer is shown in Fig. 1.

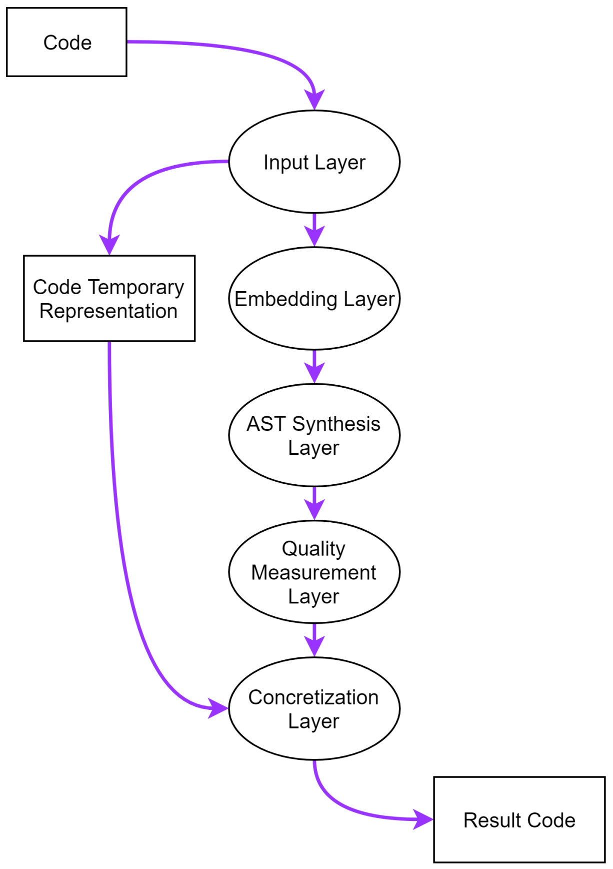

Figure 1. Data flow diagram of the configurable BSL synthesizer

A. Input Layer

The purpose of Input Layer is to handle input data (some code with evidence): extract evidence from the code and create code's intermediate representation for Concretization Layer. This intermediate representation will be used as context for synthesized code. There is no such layer in the abstract BSL synthesizer, its role here is strictly technical. Having input data processed in a separate layer, we can pass evidence into the synthesizer in different ways: embed it into code, or pass along with code and a position to insert synthesized code into.

B. Embedding Layer

Embedding Layer handles embedding the evidence received from Input Layer. Embedding Layer supports several algorithms required by different metamodels: TF-IDF and LDA for Android SDK and k-hot encoding for Java STDlib. This layer corresponds to Evidence Embedding Layer in the abstract BSL synthesizer's architecture. It is parameterized by evidence types accepted by the metamodel and embedding algorithms for each evidence type. In case embedding requires some additional data (an LDA model, a k-hot encoding dictionary, etc.) they are also added to layer's configuration.

C. AST Synthesis Layer

AST Synthesis Layer performs synthesis of sketches' ASTs by the embedded evidence. This layer can be parameterized with a number of hidden layers of encoder and decoder, however this is not used currently for existing metamodels.

This layer corresponds to both Evidence Encoder Layer and Intent Decoder Layer. Within the AST Synthesis Layer evidence is encoded into an intent latent variable element and than decoded from this element into a sketch. Two layers are combined into one because the change of parameters or models of one layer is impossible without corresponding changes to the other layer. AST Synthesis Layer is parameterized by the BED model.

D. Quality Measurement Layer

Quality Measurement Layer measures quality of synthesized sketches, deduplicates and ranks them. There is also no such layer in the abstract BSL synthesizer, here it is used to isolate quality measuring algorithms into individual abstractions. Thus, we can use different ranking algorithms for different metamodels, or even let our users choose which quality metrics to use.

E. Concretization Layer

Concretization Layer turns sketches obtained from Quality Measurement Layer and code's intermediate representation received from Input Layer into Java code. This layer corresponds to Combinatorial Concretization Layer in the abstract BSL synthesizer's architecture. Concretization algorithm has several parameters (for example, recursive search depth while synthesizing method arguments), however all existing metamodels use default settings.

It is worth noting that concretization is performed taking surrounding code in account, so Concretization Layer accepts not only an AST, but also an intermediate representation of the input source code, and uses this representation as context of synthesized code.

The result of Concretization Layer is a source code fragment with synthesized code block inserted into it.

III. Implementation of the configurable BSL synthesizer

Now let us consider implementation of the architecture presented above. We start with the way models for the configurable BSL-synthesizer were obtained.

A. Exporting the models

As mentioned earlier, researchers from the Rice University continue to develop Bayou and release improved models regularly. Therefore, it seems reasonable to re-use existing Bayou models and maintain compatibility with the new ones.

Depending on the metamodel, a model contains different data. Bayou's source code was instrumented to export all this data in runtime. For Android SDK version of the Embedding Layer it was required to export LDA and TF-IDF models from Scikit-learch objects for each type of evidence. In case of TF-IDF, we export the dictionary and IDF matrix [10]. In case of LDA, it is α, η values and ϕ matrix [11]. For Java STDlib version of the Embedding Layer only k-hot vector dictionary was exported.

For the AST Synthesis Layer both metamodels required export of Tensorflow models. To achieve this, we name all variables and output tensors of the Tensorflow model (otherwise it will be difficult to access them when executing the model) and save the model using Tensorflow's export capabilities.

B. Implementation of layers

Now we discuss implementation of each synthesizer's layer.

1) Input Layer: This layer currently has two implementations. Both implementations use Eclipse JDT to parse input code and create its intermediate representation (as JDT's CompilationUnits).

First implementation is quite similar to the one proposed in [2]. Evidence is passed directly within the code as library functions calls. Using Eclipse JDT, the code is parsed, evidence is extracted, and its position is marked for subsequent code insertion. The evidence itself is removed from the code.

Second implementation separately takes input code, an evidence and a position to insert synthesized code into. It has a slightly more convenient API, and this implementation is a little bit more efficient, since it does not need to extract evidence from input code.

2) Embedding Layer: This layer also has two implementations: k-hot encoding for Java STDlib metamodel and TF-IDF + LDA for Android SDK metamodel. The layer even allows to use different embedding algorithms for each type of evidence, but existing metamodels don't use this at the moment. All embeddings are parameterized with Bayou's exported models and satisfy specifications of corresponding Scikit-learn algorithms.

The number of output dimensions for each embedding is defined by embedding's model (for example, k-hot vector's length will be equal to the dictionary's length).

3) AST Synthesis Layer: This layer has a single implementation. It is parameterized with the exported Tensorflow model. The model itself is loaded and executed using Tensorflow for Java.

At the encoding step, evidence vectors received from the Embedding Layer are passed to the corresponding inputs of BED encoder and the Tensorflow model is executed. The result is the element of intent latent variable space (ψ).

At the decoding step, this ψ element is passed to the BED decoder (which is again a Tensorflow model). As a response, the decoder returns a vertex — start of a production path, which is described in [2] in detail. Then, in accordance with the algorithm, an entire production path is constructed. On each step of this algorithm, depending on the given node, the construction of the production path can branch out in several directions. For example, “DBranch” node (the “if” node) will build paths for a predicate, “then” and “else” branches. At the moment all models support only “if”, “while” and “try ... catch” control flow constructs.

AST Synthesis Layer’s result is a set of sketches’ ASTs, their desired number can be specified in layer’s configuration.

4) Quality Measurement Layer: This layer has several implementations and their number continues to grow.

Currently following algorithms are used for all metamodels: presence verification for all evidence, deduplication and ranking ASTs based on their occurrence frequencies. Evidence presence verification walks through an AST and collects API calls, types and contexts (types of method arguments) present in the current sketch candidate. The result is compared with the original evidence set. If some evidence is missing, this candidate sketch is removed from the result. Deduplication removes duplicate ASTs and counts the number of occurrences for every individual AST. Occurrences frequency is used for ranking: more often an AST is met, more likely it will fit a given query.

However, this naive algorithm is not the only option. We can also rank ASTs using standard code metrics, such as *LOC* (lines of code), cyclomatic complexity [12], or more complex ones. Depending on task’s domain, these metrics can be selected to provide more relevant results (for example, *LOC* could be used for generating examples, which are preferred to be short according to [7]).

5) Concretization Layer: Finally, Concretization Layer performs synthesis of the program. This layer has a single implementation based on Eclipse JDT.

Concretization Layer takes an AST and code’s intermediate representation as input. In our case the intermediate representation is an Eclipse JDT CompilationUnit object. The concretization algorithm described in [2] is executed on this CompilationUnit object. Then, combinatorial search is performed guided by several heuristics. For example, functions without arguments are examined earlier than functions with arguments, since functions with arguments produce further search.

It is worth noting that current implementation is able to synthesize a method, some arguments of which can not be synthesized within current context. In such cases, a variable of an appropriate type is created and initialized with a *null* value. It is assumed that the programmer will replace it with an appropriate initialization.

Finally, unreachable code or code that does not affect the result of the function is removed (i.e. Dead Code Elimination is performed), the code is formatted, fully qualified class names are converted into simple class names and import expressions. Resulting code fragments are returned according to corresponding rank order of their ASTs.

IV. Implementation of the IntelliJ IDEA plugin

This section describes an IntelliJ IDEA plugin providing user interface to the implemented BSL synthesizer. Firstly, we mention several issues arising from such integration that need to be resolved to make the synthesizer work in IDEA’s environment.

A. Integrating BSL synthesizer with IntelliJ IDEA

There were two main challenges integrating implemented BSL synthesizer into an IntelliJ IDEA’s plugin: obtaining models and progress indication.

As mentioned before, a BSL synthesizer needs models to perform. Unfortunately these files are quite large (~100 Mb for the Android SDK model and ~200 Mb for the Java STDlib model), so we can’t distribute them with the plugin directly. Currently, these models are stored in an Amazon S3[5] repository along with a descriptor file listing all supported models, paths to corresponding files and MD5 hashes of these files. Each instance of the plugin creates a directory for local repository. If the requested model is missing on disk or corrupted it will be downloaded from the remote repository.

Another issue is that code synthesis can take up to 10 seconds, and downloading the models could take even longer, so the lack of progress indication during such tasks will result in quite negative user experience. To handle progress indication the synthesizer subsystem accepts a special object: it is a data class object which fields are used for storing currently executed process (e.g. “Generating Sketches”, “Downloading TF-IDF Model”, etc.) and its progress as a double from 0.0 to 1.0. Based on this object’s state the UI thread shows and updates a progress indicator when time-consuming operations take place.

B. User interface

User interface for IntelliJ IDEA’s plugins could be implemented in a number of ways: using special window dialogs, with a separate DSL to use in comments, or as some language elements within Java code itself. We decided to create two alternatives: an approach based on method annotations defining required evidence (the same approach is employed by numerous popular Java libraries, for example, Project Lombok[6]) and an approach based on a comment-based DSL.

[5]Amazon S3: an object storage by Amazon, https://aws.amazon.com/s3.

[6]Project Lombok: a library for generation of utility methods in Java classes. URL: https://projectlombok.org/

1) Method annotations: We have implemented a Java library containing following annotations:

- BayouSynthesizer — annotation defining a metamodel to use;
- ApiCall — annotation defining an API call evidence;
- ApiType — annotation defining an API type evidence;
- ContextType — annotation, defining a context type evidence (type of an API call argument).

Let us consider a request to the synthesizer, which contains an "ApiCall" evidence with a value "readLine" and an "ApiType" evidence with a value "FileReader". Using method annotations approach this query looks like this:

```
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) {
    }
}
```

ApiType and ContextType annotations accept an object of type Class<T> — class of a Java class and for them IntelliJ IDEA automatically performs code completion. But because of Java constraints, ApiCall annotation can not accept objects of type Function<T>. Hence, ApiCall should accept either a String or an enumeration, previously created and describing all possible API calls of the current model. For Java STDlib, this kind of enumeration can not be created because the number of available API calls exceeds maximum enumeration size. And obviously, there is no auto-completion for String values in IntelliJ IDEA.

Another issue with this approach is the lack of strict typing. For example, it is possible to specify BayouSynthesizer as StdLib (Java STDlib metamodel) and add ContextType annotation which is not supported by this metamodel.

Thus, method annotations approach common to Java programmers has significant drawbacks and should be used only by programmers who are very well acquainted with the synthesis system.

2) Domain Specific Language: The DSL was implemented using Grammar Kit[7]. It contains the following identifiers:

- STDLIB or ANDROID — initial identifier, defines a metamodel to use;
- API — identifier for an API call evidence;
- TYPE — identifier for an API type evidence;
- CONTEXT — identifier for a context type evidence (type of an API call argument).

The same query for a "readLine" call and a "FileReader" type evidence will look like this using DSL approach:

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

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

The domain specific language integrates well with IntelliJ IDEA. Strict typing is available (for example, CONTEXT can not be used when Java STDlib metamodel is chosen), and code completion for all of evidence types values is active. Moreover, IntelliJ IDEA allows to extend language's grammar to show custom messages for some specific grammar errors.

The resulting language is quite simple. Strict typing and error prompts make it easy to learn even without additional documentation.

V. Evaluation

Our plugin was tested on a number of generation tasks mentioned in [2] for Android SDK and on a set of examples similar to those that are presented on http://www.askbayou.com/ for Java STDlib. In total we have tried 20 generation tasks for Android SDK (e.g. code that works with "BluetoothSocket", code that works with "File", etc.), and 20 generation tasks for Java STDlib (code working with Collection classes, "File" and different "Readers"). All tasks were executed on latest builds of corresponding Bayou applications and on our system. The results show that our implementation is equivalent to Bayou's: in all cases both synthesizers produced the same result.

For instance, the following is a piece of code generated from the example above that used "readLine" and "FileReader" as evidence:

```
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);
```

[7]Grammar Kit: a tool for custom language support for IntelliJ IDEA, URL: https://github.com/JetBrains/Grammar-Kit

```
        while ((s1 = br1.readLine()) != null) {}
        br1.close();
      } catch (FileNotFoundException _e) {
      } catch (IOException _e) {
      }
      return;
    }
}
```

This example shows that the synthesizer is able to generate code with non-trivial logic and could be applied to generate supplementary functions working with libraries.

A. Known limitations

Surely, BSL synthesizers have their limitations. First of all, they are not capable of generating code using two metamodels simultaneously. For example, it is not possible to generate code using both methods existing only in Android SDK and methods that exist only in Java STDlib. The only way to do this is to create a new metamodel including the other two. As far as we understand, there is also no research available showing how synthesis' quality depends on the growth of generated programs' space. An educated guess is that the quality will drop.

Secondly, BSL synthesizers are currently applied to generate only parts of methods or whole methods. BSL synthesizers could not be used to generate a whole class or even a whole project. As far as we know, there is no published research on generating pieces of code larger than methods using BSL synthesizers. However, they are a suitable tool to generate API-heavy code, which is confirmed by evaluation in [2] and our experiments.

Conclusion

In this paper we present a configurable BSL synthesizer compatible with Bayou models. It was implemented as a plugin for IntelliJ IDEA providing two types of user interface: Java annotations and built-in DSL. Our evaluation shows that qualitatively and quantitatively our implementation complies with Bayou's experimental results presented in [2].

As future work we plan to improve plugin's user interface, prepare models for some other libraries (e.g. IntelliJ Platform SDK) and improve IDEA's code completion subsystem using synthesizer's results providing users with a tool to generate chains of API method calls. Furthermore, we plan to integrate our BSL-synthesizer with DeepAPI tool [13] to create a system capable of synthesizing code from natural language queries.

References

[1] S. Gulwani, J. Hernández-Orallo, E. Kitzelmann, S. H. Muggleton, U. Schmid, and B. Zorn, "Inductive programming meets the real world," *Communications of the ACM*, vol. 58, no. 11, pp. 90–99, oct 2015.

[2] V. Murali, S. Chaudhuri, and C. Jermaine, "Bayesian sketch learning for program synthesis," *CoRR*, vol. abs/1703.05698, 2017. [Online]. Available: http://arxiv.org/abs/1703.05698

[3] D. Barber, *Bayesian reasoning and machine learning.* Cambridge University Press, 2012.

[4] A. Solar Lezama, "Program synthesis by sketching," Ph.D. dissertation, EECS Department, University of California, Berkeley, Dec 2008.

[5] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," 2013. [Online]. Available: https://arxiv.org/abs/1312.6114

[6] H. Zhang, A. Jain, G. Khandelwal, C. Kaushik, S. Ge, and W. Hu, "Bing developer assistant: Improving developer productivity by recommending sample code," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: ACM, 2016, pp. 956–961.

[7] R. P. L. Buse and W. Weimer, "Synthesizing api usage examples," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 782–792.

[8] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," 2016. [Online]. Available: https://arxiv.org/abs/1610.00502

[9] L. Kaufman and P. Rousseeuw, *Finding Groups in Data: An Introduction to Cluster Analysis.* Wiley Interscience, 2005.

[10] A. Rajaraman and J. Ullman, *Mining of Massive Datasets.* Cambridge University Press, 2012.

[11] M. Hoffman, F. R. Bach, and D. M. Blei, "Online learning for latent dirichlet allocation," in *Advances in Neural Information Processing Systems 23*, J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta, Eds. Curran Associates, Inc., 2010, pp. 856–864.

[12] T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, Dec 1976.

[13] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep API learning," *CoRR*, vol. abs/1605.08535, 2016. [Online]. Available: http://arxiv.org/abs/1605.08535