University of Kent

Advanced Computer Science MsC: end of year thesis

Generating Code To Benchmark Performance of Huge Systems

Author
Octave Larose

Supervisor
Dr. Stefan Marr

Total word count: 10,000.

Acknowledgements

I would like to thank my supervisor, Stefan Marr, for all the advice and help he has provided me during my work on this project. As I was unaccustomed to doing research due to my more practical education at Epitech, his input and insight have been of immense help, and his availability made it simple for me to progress consistently.

I would also like to thank Sophie Kaleba, PhD student at UKC and member of the Programming Languages and Systems Group, for giving me general advice alongside Stefan and advising me with regards to my work. The weekly meetings I've had with both of them since the start of this project were a tremendous help.

Finally, I'd like to thank the University of Kent and their teaching staff for their support throughout my year there, always being available for any queries I may have about various matters.

Abstract

Benchmarking is a vital part of programming language implementation, necessary to assess performance adequately and learn where to focus efforts to gain better results. As of now, most benchmarks used in academia are not representative of real applications. The latter ones would be ideal benchmarks, yielding the most insight into how a language implementation should behave to fit real world needs ; and actual codebases tend to be huge in size, being projects maintained over many years by many developers. Being often long-running, they are also the applications most likely to benefit from complex optimizations from their language of choice. However, those programs are most often proprietary and so inaccessible to researchers who could benefit from them.

We propose a solution to that problem: automatically generating our own large benchmarks. To achieve this, we aim to collect data on the structure of many sizeable codebases, and then to generate code that resembles them by having it abide to the extracted metrics. Through this, we could generate an arbitrary number of large benchmarks that would behave similarly to real programs, which we could use to improve language implementations to deal with massive codebases. While this goal has not been entirely realized, my work represents a good first step towards achieving this in its totality by analysing data from real codebases and managing to generate executable code from real programs.

Contents

1 Introduction

1.1 Why generate benchmarks?

Benchmarks used in academia are rarely representative of the applications used in practice. This is partly due to the great amount of variety in the latter, which also leads to the need for a vast amount of benchmarks: many different fields have many different needs, hence could make use of entirely different benchmarks. These fields also naturally evolve as time goes on, which may shift their needs: therefore the benchmarks tailored to their specific requirements must also evolve with them to make up for this.

Realistic synthetic benchmarks could potentially fix this problem. Being able to generate programs with behaviours resembling those of real codebases would mean getting access to a potentially infinite amount of benchmarks that would yield insightful data about the real world needs of applications.

Moreover, the most complex programs to optimize for (therefore the most insightful benchmarks) are massive in size, often exceeding hundreds of thousands of lines of code ; but these large codebases tend to be proprietary, as applications with this much work put into them often exist to be lucrative. This means we don't have access to them and can't analyse them to see how language implementations should be modified to account for what they require. Therefore, if we could generate our own, there would be no need to get access to these proprietary programs.

1.2 Objectives

The end goal is to manage to generate large benchmarks automatically. These need to be true to life, which is to say they need to simulate the behaviour of real applications accurately.

To achieve this, we need to focus on two different tasks. The first is fetching metrics from real codebases to understand how they tend to be structured and to gain information needed to know how to recreate similar ones. The second is actually generating the code, taking the acquired metrics into account to output source code that can be ran and has a similar behaviour to a real codebase. This is too great a task to achieve fully in a few months of work, though, and my actual goals are more realistic: acquiring data from very large codebases, but focusing on recreating smaller real benchmark programs.

I first fetched static metrics from a large number of real codebases (over 130) that I collected from GitHub, and to extract insightful data from them. I determined 200,000 lines of code to be the minimum for what we consider to be a large codebase. From these, we expect to gather static source code information (i.e data acquired without executing the program, from analysing the source code itself) about the structure large codebases such as them tend to have, such as the distribution of the number of methods per class or the number of variables per method.

Focusing solely on static properties with a purely statistical approach in mind is doable (finding a realistic amount of methods in a class from them, an adequate number of fields in comparison, etc.), but can only go so far to create realistic programs, as a lot of dead code will be present if these functions are not each called at some point. Program execution also has to be represented, and therefore dynamic metrics (i.e data acquired from analysing programs during their runtime) have to be acquired as well.

For the code generation aspect, I fetched dynamic metrics from real benchmarks, most notably in the form of calltraces. I aimed to generate fully executable code from this data, as well as simulating its method's contents and operations to a basic degree, attempting to recreate the control flow of the original.

My approach was Java-centric: only Java codebases were cloned, and my code generator and its output were all in this language. This is because I chose to focus on a single language because of time constraints, that Java's popularity meant I had more data to work with, and that its object-oriented design made code generation depend entirely on distinct, easy to generate classes. However, it should be entirely possible to port my approach to other programming languages.

2 Literature review

2.1 Repository mining

Repository mining, i.e analysing data from software repositories, is a widely used technique to acquire data about codebases. It can be utilized to achieve various goals, such as to analyse the lifespan of code smells [1], to assess developer contribution within a project [2] or to look for the relationship between production and test code [3]. Improvements with regards to it are still being achieved today, such as with CodeDJ [4], which allows for more complex queries than by simply relying on the GitHub public API.

Being used by so many papers, there has also been research pertaining to how to mine them correctly and how it can go wrong [5] [6] by fetching repositories unrepresentative of what the miner is looking for. Therefore, rules must be enforced to query only projects and data that are relevant to specific use cases.

2.2 Synthetic benchmarks

The idea of creating artificial programs matching characteristics of real ones, i.e generating synthetic benchmarks, is a fairly old one [7]. Benchmarks created in this fashion get a mixed reception overall, with some succeeding at reflecting program behaviour to a decent degree and others which do not. Many different approaches to achieve this goal exist, with various degrees of success.

Machine learning is an obvious approach to synthetic benchmark generation, because of it being suited to automatic generation of data similar to given inputs. This has been pursued by *CLgen* [8], which aims to generate OpenCL compute kernels (GPU routines). The authors also relied on repository mining to acquire inputs to train their system on, as well as an LLVM based toolchain for rejection filtering ; however, they later re-evaluated it [9] and found that their ML approach yielded no advantage over simply using their raw input data directly. Therefore, it seems synthetic benchmarks generated from machine learning don't tend to yield good results, but this sector being so active means they may improve drastically given time.

Program generation is also possible without inputs, and one of the most successful projects aimed at testing language implementations is *Csmith* [10], a compiler fuzzer (i.e generates invalid or random data as input for a compiler) which generates random C programs with no undefined behaviour. The focus, however, is not on size: I've generated 100 programs with it using its default settings, and got an average of 1422 lines of code (from 31 to 4,040, hence varying a lot). None of its input parameters are about increasing the size of the output, although many are about limiting it in various ways.

Moreover, all its output is a single consecutive code block, without dividing the functions into discrete modules or source files : the concern isn't realism, because of its nature as a fuzzer.

Synthetic benchmarks may also be created from a limited number of insightful microarchitecture-independent program characteristics. Ajay M. Joshi et al. highlight many interesting ones in their work [11], such as instruction mix (i.e frequency of various operations) or instruction level parallelism (i.e number of instructions between a memory location's initialization and when it's read from) ; and using only these few metrics, they can recreate synthetic benchmarks similar in behaviour to their real counterparts (in terms of instructions per cycle and branch prediction rate, with the SPEC CPU2000 suite). This gives us an idea of the characteristics by which we can measure our system's performance. But for our purposes, their approach was more about generating assembly instructions directly than source code ; and their system's scalability and efficiency with very large codebases was not addressed other than by mentioning the performance impact of increasing the code size.

Another approach is the "record & replay" one used by G. Richards et al. to create automated JavaScript benchmarks [12] using real applications. They first get a trace of JS operations from recording execution of a real website's code ; then that call trace is modified to fit certain requirements (such as browser-independence or determinism), and can be replayed to get an executable benchmark with a similar behaviour to that of a real application. Generating benchmarks in a similar manner is what I aimed to achieve with the code generation aspect of my project, as I will expand upon in Section 4.2.

3 Analyzing the structure of huge codebases

To understand how to recreate large codebases, we first need to analyse their structure. This requires fetching a large number of them to acquire this data from : however, as previously mentioned, most are proprietary applications that we have no access to. Therefore, we need to turn to open-source repositories from sites such as GitHub, and mine them to fetch some that correspond to our criteria ; from these, we acquire insightful information which can later be used to generate code similar to the fetched programs.

3.1 Data fetching

Fetching data about large codebases was divided into two major parts: finding large, relevant repositories ; then extracting information from their source code.

3.1.1 Methodology

Fetching repositories. All my repositories were acquired from GitHub, relying on its public REST API to clone relevant projects onto my machine. Precautions were taken to avoid cloning non-relevant projects : for instance, in 2014, 71.6% of GitHub projects were entirely personal [5] and most often simply existed for storage purposes. As such, projects with less than three contributors were ignored entirely.

Fetching static metrics. I relied on many metrics, most notably CK [13], the latter being chosen because of the insights they yielded at both the class level and method level. Examples of CK metrics are "Coupling Between Objects" (the number of dependencies a class has), or "Weight Method Class" (the number of branch instructions in a class). Simpler metrics such as a method's number of parameters or number of local variables were also relied upon.

3.1.2 Fetching repositories

Repositories were cloned using a custom script I wrote, available on GitHub [1] and that relies on the latter's public REST API.

The minimum number of lines of code was set at 200,000. This is a somewhat arbitrary number as the concept of what is and isn't a "large" codebase is debatable.

Only Java repositories were considered. Projects that had under 3 contributors were ignored to avoid cloning personal projects, which could be possibly experimental or only used for code storage, instead of codebases representative of real world needs.

[1] `https://github.com/OctaveLarose/github-fat-cloner`

Class name	CBO	WMC	DIT	RFC	LCOM	...
WalletUtils	21	43	2	41	76	...

Table 1: Example of various CK metrics.

Projects under 20 commits were also ignored as a precaution for the same reason, as an average of over 1000 lines of code per commit is indicative of the project getting imported from somewhere else, hence likely to exist for storage purposes since it's not actively maintained enough to have many contributions. However, the previous rules were enough to stop these repositories the overwhelming majority of the time.

In total, 131 codebases were fetched using these criteria. However, information was only acquired from 63 of them, as my static metrics tool wouldn't work on half of the codebases.

3.1.3 Fetching static metrics

I relied on M. Aniche's CK tool [14], which returns CK metrics at the class and method level. Additional information such as local variables' respective number of usages is also returned, giving us a lot of data about the structure of the program at every level.

Table 1 shows an example of the values CK metrics can have. As an overview , we get useful information about the class' relationship to the rest of the codebase, with the CBO ("Coupling Between Objects", i.e number of external non-default dependencies) or DIT ("Depth Inheritance Tree", i.e number of classes in the class' inheritance tree until reaching the root class `java.lang.Object`) metrics.

Metrics about the class' design independently of the program's architecture include WMC ("Weight Method Class", number of branch instructions in a class) or RFC ("Response For a Class", number of unique method invocations in a class) or LCOM ("Lack of Cohesion of Methods", based on how many pairs of methods aren't related through both accessing/modifying the same field).

Finally at the method level, we also have access to non-CK metrics such as the number of return statements, the number of local variables, how many try/catch blocks are present or how many string literals appear in the method body.

Hence we have access to static metrics about every notable level of the source code (insights about its architecture, class composition and method structure).

3.2 Data interpretation

I wrote Python scripts [2] to interpret the data I fetched, generating graphs and CSV files to visualize and compare different attributes. Figure 1 shows an example of a graph generated by my program from the data directly, here counting the number of public methods per class.

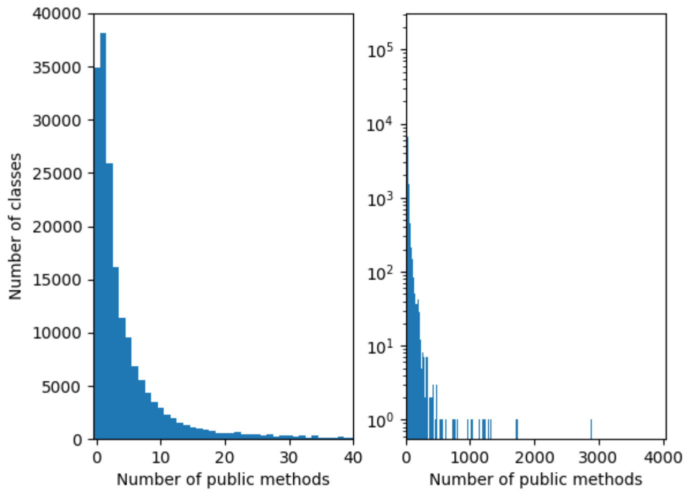

Figure 1: One of my graphs generated from the acquired static metrics.

To look for interesting correlations between attributes, I relied on Spearman's rank-order correlation coefficient to look for monotonic relationships (i.e where a variable increasing or decreasing is tied to another doing the same along with it) between pairs of class-level attributes. The class level was specifically chosen because it contained many insightful metrics about the architecture of the codebases, and I focused solely on it because of time constraints: however, there is a lot of data available for the method level, for instance, which could yield insightful information about their behaviour.

Importantly, classes with the word "test" in their name or file path were not considered, in an attempt to not take test suites into account, which would skew the results.

[2] `https://github.com/OctaveLarose/msc_thesis_2021/tree/main/data_interpretation/static_metrics`

3.3 Results and Insights

3.3.1 Graphs showcase

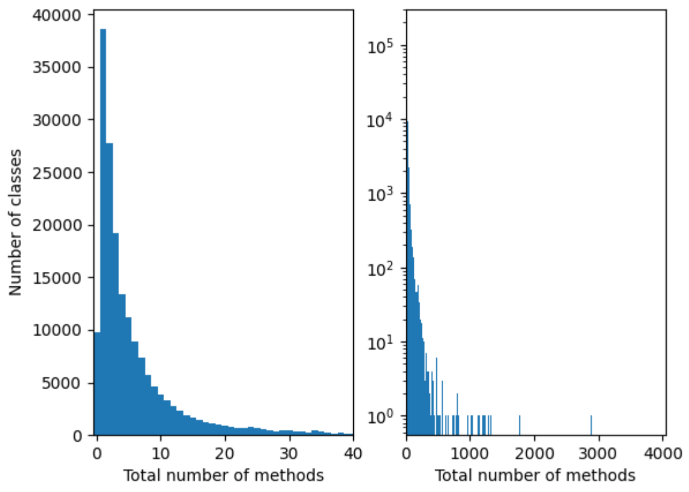

Figure 2: Distribution of every class in the fetched codebases w.r.t their total number of methods.

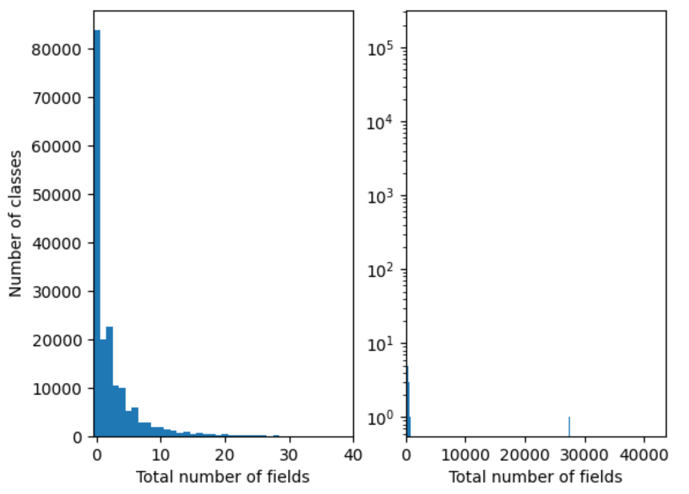

Figure 3: Distribution of every class in the fetched codebases w.r.t their total number of fields.

As shown in Figure 2, the majority of classes (38,567) only had a single method. 15,011 of them were anonymous classes (i.e unnamed local classes), as although 68% of projects that relied on them had less than 200 of them, some could rarely exceed 1,000 (with the outlier *osmandapp/OsmAnd* reaching 2,370).

For the 9,792 who have none, 33% are enums, which are primarily designed to hold constants, hence do not always need methods. Inner classes come third with 22%, but regular classes are second with 23%. While the first tend to be as simple data structures, the second having so many instances with no methods whatsoever can come off as unexpected : regular classes tend to be used with private attributes modified via methods. It's fair to assume inheritance was relied upon to initialize methods in parent classes and allow many classes to create none themselves.

Other than for 0 methods, the distribution of methods is mostly always a descending curve until around 22 methods per class, where it becomes more irregular with values sometimes exceeding the ones preceding them. It seems classes past this point don't seem to be easily categorized, and are being uncommon for having so many methods in the first place.

The two outliers with the most methods per class, at 2,897 and 4,045 methods respectively, are both JNI classes (*Java Native Interface*, used to call/be called by native applications as opposed to relying on the JVM) which seem to handle a vast amount of physics operations. Considering their root package is named *badlogic*, however, it's likely the developers would have preferred having these classes be smaller in scope. The third largest, with 1,760 methods, seems to be a library "entry point" to instantiate various interfaces from it ; none of these classes seems to have been automatically generated.

The distribution of fields (Figure 3) differs a lot. The vast majority of classes have no fields, then around 20,042 have only one, 22,603 have two, the values descending from there. Other than the huge gap from 0 fields to 1 field, the biggest gap seems to be from 2 to 3 fields, being a difference of 500.

Interestingly and unlike the distribution of methods, it seems the curve doesn't descend as smoothly : there are more classes with 2 fields than with 1 field, for instance, which is unexpected as it differs from the gradual descent of the total methods curve.

While there tends to be more methods per class than fields per class overall according to our data (with the method distribution reaching a higher peak and descending more slowly than its fields counterpart), the fields have some major outliers in this case. Indeed, one class in particular at the very edge of the graph reached 43,588 fields : this is a generated class containing a large amount of IDs stored as public fields, and so are the other 3 classes with the most fields (at 41,970, 27,454 and 24,050 fields respectively), hence the massive numbers we observe.

Figure 4 shows the distribution of the number of classes among all fetched projects. No project exceeded 7361 classes, and the lowest only had 252 : a surprising amount of variation in their number showing how much large codebases can differ in this regard. The majority of them, 27.5%, circled

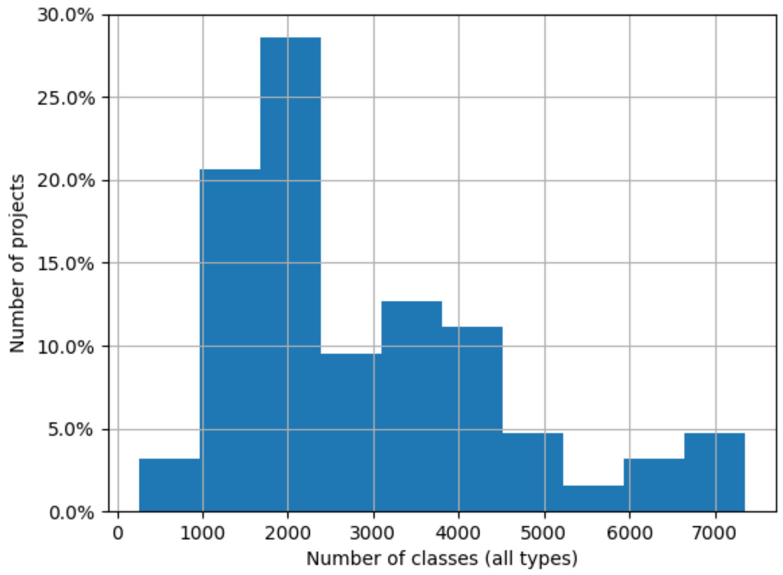

Figure 4: Distribution of the number of classes per project.

around a number of 2,000 classes ; and 20.6% had around 1,300. It seems about half of all codebases had between 1,000-2,300 classes, with around 92% of the remaining half exceeding these numbers. Therefore, while this distribution can vary a lot, most projects tend to hover around a specific amount of total classes, with codebases having less than this amount being a stark minority.

3.3.2 Correlation data

Attr. 1	Attr. 2	Sp. Correlation factor
wmc	totalMethodsQty	0.904
wmc	abstractMethodsQty	0.858
wmc	loc	0.912
rfc	loc	0.801
totalMethodsQty	publicMethodsQty	0.808
totalMethodsQty	abstractMethodsQty	0.970
totalMethodsQty	loc	0.812
publicMethodsQty	abstractMethodsQty	0.826
totalFieldsQty	staticFieldsQty	0.911
totalFieldsQty	protectedFieldsQty	0.845
loc	assignmentsQty	0.842
loc	variablesQty	0.824
assignmentsQty	variablesQty	0.960

Table 2: Correlation factors for class attributes where the Spearman factor either exceeded 0.8 or was inferior to -0.5.

Table 2 shows attributes which showed particularly high or low correlation factors (exceeding or being below two thresholds I defined) when compared.

WMC, or Weight Method Class, counts the number of branch instruction in a class. We can see it being highly correlated to the total number of methods and LoC (lines of code), which makes complete sense as more methods or code increases the likeliness of that code having branch instructions.

However, it's interesting that the correlation between WMC and abstract methods specifically is so high, when its corr. factor with public/private methods was below our threshold. This highlights that many classes contain nothing but abstract methods, such as interfaces, as is confirmed by the factor of 0.970 between total methods and abstract methods ; while no high correlation between total methods and private methods appeared, for instance, as classes with solely private methods tend to be rare.

We do see a factor of 0.808 between the total methods and public methods, showing classes with only public methods are fairly common. The high correlation between public and abstract is related to their properties of both often being the only type of methods in a given class, and doesn't necessarily imply they tend to often coexist within a class context.

We can brush over the high correlation between lines of code and the total number of methods, RFC (number of unique method invocations), number of variables and number of variable assignments: it's self-explanatory that a larger number of lines of code is more likely to be present in a context with more methods (hence more code) or more instructions of a given type (more likely to appear the more code is written).

The amount of fields in total seems to be highly correlated with static fields (0.911), and a little less with protected fields (0.845). This is a similar situation than with abstract and public methods, in that it shows it's recurrent for a given class to only have static fields (such as in an class keeping track of various constants) or only protected fields (such as in a parent class which subclasses need to inherit from). Interestingly, the first case seems to be slightly more common than the second.

Strong negative correlations seem to be absent from these results. While there were negative ones, the lowest only reached -0.249, corresponding to public methods and default methods (i.e interface methods that have implementations). I personally can't find a satisfying explanation for this correlation, which may need to be disregarded altogether because of its rather low value.

Finally, we can note interesting CK metrics about the class w.r.t the rest of its codebase - such as DIT ("Depth Inheritance Tree") or CBO ("Coupling Between Objects") - were not present, indicating the absence of strong correlations between them and other metrics. LCOM ("Lack of Cohesion of Methods") was also absent despite being about the class' content itself like other metrics that appeared, which means it isn't associated to how many methods of a given access type are in a class, for instance.

3.4 Conclusion

We can see patterns emerge in the number of methods or fields that a class tends to have, as well as in the number of classes per codebase. The outliers from our data set highlight interesting information as well, such as the fact that classes with enormous amounts of fields tend to be automatically generated, while the same can't be said about classes with a large amount of methods.

Insightful information could be yielded from the observed data in general, such as the fact that there are many regular classes with no methods whatsoever, or that the distribution of the number of fields doesn't descend as smoothly as the distribution of the number of methods, despite their resemblance.

Correlations between various properties of the codebase highlight characteristics classes tend to have, such as only relying on static fields or having no method that isn't public. None of the metrics I compare seem to be strongly negatively correlated, while the opposite isn't true.

This information was all pertaining to the metrics I fetched regarding the codebases I mined. Note that no dynamic metrics were utilized, as executing all these codebases would have been complex and very time consuming. However, dynamic metrics also need to be relied upon to generate realistic code, and describing how I fetched them will be done in the next section.

4 Generating benchmark code

As a more appropriate first step before attempting to automatically create codebases with several thousands lines of code, I made the decision to start by focusing on smaller programs. The AWFY benchmark suite [15] was chosen to be generated programs from, containing both micro and macro benchmarks, so that my system may learn to recreate the behaviour of real ones. This allowed me to see what language features needed to be generated to simulate a good benchmark in general.

It is provided limited information about their structure, and attempts to emulate their behaviour based on it. The reason for this data being restricted is to strive to create programs with similar executions, but without copying them exactly, to end up creating new benchmarks from them.

My system was designed to be scalable with larger input programs, though, and its aim is to be reused and improved in the future to actually generate huge benchmarks by itself. Creating code that can be executed given limited input information turned out a more complex task than expected, and so most of this heavy lifting has already been achieved in my work.

4.1 Dynamic metrics fetching

I relied on DiSL [16], a DSL language for bytecode instrumentation, to fetch the needed dynamic metrics.

Its main role was providing calltraces, by logging method entries/exits as well as information about them ; like the name of the method, of the parent class, number and type of parameters, or the time spent between entry and exit. Therefore our calltraces are essentially dynamic callgraphs, i.e recordings of an individual execution of a program, and do not attempt to depict every possible run of it.

I also used it to get information about method bytecode, most notably arithmetic operations, as they can be recreated easily and have an impact on program behaviour by taking up many CPU cycles to calculate the result. By logging arithmetic instructions in order for each method, this created lists of operations that the generated methods could simulate.

4.2 The code generator

My code generator, named *bootleg*, relied entirely on Abstract Syntax Trees (i.e tree representations of source code structure). This is a standard technique in meta-programming [17] which ensures static correctness in the generated code, since the trees have to be well-formed. This allowed me not to operate at a level too low for our purposes (such as bytecode). The main library used for code generation was JavaParser [18], providing me with these data structures ; and the final project ended with 158 commits and 2,519 lines of code.

The way it works is that it takes a calltrace as an input, corresponding to the execution of an actual program, and from this it generates a similar program that can be ran. The two main overarching challenges were getting them to build without issues (i.e generating code with valid syntax, referencing symbols that exist within the referrer scope, etc.) and getting them to run without errors (most notably by preventing them from ever accessing null pointers, which means objects needed to be instantiated instead of relying on these null references).

4.2.1 Execution

Each call in the calltrace contains various information about the call:

- the callee and caller methods' names;
- the callee method's type signature (i.e input parameters and return type);
- the parent classes of both methods and parent classes;
- a timestamp in the case of a method exit, representing the amount of time elapsed between entry and exit.

We iterate over all of them in order and generate source code progressively: we keep track of all known classes as well as the methods within them, and create new ones when non registered names appear in a call.

If a call is a method exit, then we add a return statement to the end of the method in question, as long as it isn't a procedure or doesn't already have a final return statement. Failing to do so would prevent compilation, as functions would be generated without their expected return values.

However, if it's a method entry, then a call to the current method needs to be added to the previous one in the calltrace. This is where the most complex logic in the project lies, and there are several hurdles to simulating a realistic program related to it. These issues will be expanded upon in the next section.

The generator can also take an operations file as additional input, which logs all arithmetic operators called in the original bytecode, the format of which can be seen in Figure 5. For each of them in order and for each method individually, arithmetic operations are added to the method body using the recorded type and operator.

Once the calltrace has been entirely processed, every class is exported to a file. Their package declaration is taken into account so that they can be exported to adequate subfolders depending on it: the result is a collection of classes which form a program executable from an entry point in the same class as in the input program.

```
Mandelbrot.mandelbrot
DMUL DDIV DSUB DMUL DDIV DSUB DADD DMUL DMUL IADD ISUB ISUB
Run.measure
LSUB LDIV LADD
Run.reportBenchmark
LDIV
```

Figure 5: An example of an operations file, logging arithmetic operations that were detected during runtime.

No post-processing is necessary and each call is only interpreted once, since it was designed with possibly huge codebases (hence, possibly huge calltraces) in mind. However, it would be possible to implement and it may be relevant in the future to do so: for instance, iterating through each generated function to find unused variables, and modifying the method calls to make use of them (or putting them in a field).

4.2.2 Method generation

The generator can easily generate empty methods from calltraces, seeing as they define both their names and input/output types. However, their bodies need to be filled as well with instructions, which can be of various types: object instantiations, primitive type variables instantiations, arithmetic operations, return statements, and of course method calls.

Handling method calls differs depending on the concerned type. Invoking static methods implies calling them from the class' name (`ParentClassName-.staticMethod(...)`), and calls to method within the same class scope need to be detected to be called from a simple `this` keyword (`this.-localMethod(...)`). These two cases are relatively simple compared to the final option, non-static calls to methods from other classes, which require an instance of the callee class.

In a simple system, we can simply create a new instance of the class needed to invoke certain methods and ignore all information available in the method : its input parameters, local variables or the returned values of the calls it executes. We may also pass `null` pointers to methods as arguments instead of creating new object instances, since we know they would never get accessed anyway as their parameters are always ignored. This very simple internal logic is easier to implement, but is obviously non-desirable, as this would make the generated program only similar in terms of architecture and not in terms of actual behaviour.

Instead, interesting behaviour would be a continuity in the values accessed akin to how a real codebase operates : for instance, if a method takes certain arguments, it will almost assuredly require their values in some of the operations it executes. If a local variable is instantiated, it's highly likely it will be modified and accessed. Detecting and ignoring unused values like this is trivial for any modern compiler as tracking variable usage is extremely common for them : therefore, our system strives to make full use of the variables available in each context, instead of needlessly instantiating new ones or relying on random or default values.

For instance, if a method call takes as parameters a `JsonValue` object and a boolean, a `null` pointer or a new instance of this class can be fed for the first and a random boolean for the second. However, if the caller method takes a `JsonValue` instance as an input parameter (therefore already has an instance of it available for the method call) and a previous method call returned a boolean, then our system will detect them and give them to the method in question as input.

```
public boolean innerBenchmarkLoop(final int innerIterations) {
  return verifyResult(mandelbrot(innerIterations), innerIterations);
}
```

```
> pub (I)Z Mandelbrot.innerBenchmarkLoop
> pri (I)I Mandelbrot.mandelbrot
< pri (I)I Mandelbrot.mandelbrot (231262ns)
> pri (II)Z Mandelbrot.verifyResult
< pri (II)Z Mandelbrot.verifyResult (39034ns)
< pub (I)Z Mandelbrot.innerBenchmarkLoop (494058ns)
```

```
public boolean innerBenchmarkLoop(int iypou) {
  int sujed = this.mandelbrot(iypou);
  boolean pdzlq = this.verifyResult(sujed, iypou);
  return pdzlq;
}
```

Figure 6: From top to bottom: the real `innerBenchmarkLoop` method in the `Mandelbrot` benchmark, the part of the calltrace which describes this method, and finally the method *bootleg* generated from it.

Figure 6 highlights this generation process. Calls to methods will be added to its body in order of appearance. Each method call's return value, if it has one, is stored in a local variable: the generator aims to find a use for it in a future operation, such as giving it as input to a method or use it as its return value. Input parameters are also local variables that can be considered in the same way.

Both happen in this example: first, the `mandelbrot` method takes an integer as an argument, which the generator infers can be accessed from the input parameter. This function returns another integer, stored in a local variable. The next method call, this time to `verifyResult`, takes in two integers: and since there are two integers available in this context, it feeds

```
private void measure(final Benchmark bench) {
  long startTime = System.nanoTime();
  if (!bench.innerBenchmarkLoop(innerIterations)) {
    throw new RuntimeException("...");
  }
  long endTime = System.nanoTime();
  long runTime = (endTime - startTime) / 1000;

  printResult(runTime);
  this.total += runTime; // "this.total" type: long
}
```

```
private void measure(Benchmark kcynl) {
  long seqla = 3364;
  seqla -= 9757;
  seqla /= 5693;
  seqla += 9022;
  Mandelbrot bmaxj = new Mandelbrot();
  boolean twrpg = bmaxj.innerBenchmarkLoop(8908);
  this.printResult(seqla);
}
```

Figure 7: Arithmetic operations example in a generated method.

them to this function. With no more method calls, the `innerBenchmarkLoop` needs to end after adding a return statement with a boolean value. If there weren't any, it would be forced to put a random one, but there is one returned from the second method call: hence, this local variable is returned. In this simple example, this ends up creating a method with the same behaviour as the real one.

Arithmetic operations can also be provided and taken into account in the generation of method bodies. As seen in Figure 7, the original method had a subtraction, a division and an addition between two `long` values: therefore the opcodes `LSUB`, `LDIV` and `LADD` appear in this order in the original code. Those were detected and logged during the input program's runtime so that they could be recreated, which is why the generated method contains these arithmetic operations in order. However, since calls to standard library Java function aren't taken into account in the calltrace, it has to instantiate its own `long` variable to operate on, hence the behaviour turns out different than the real ones which relied on the return values of calls to `System.nanoTime()`.

4.2.3 Classes

Classes are simply a collection of methods with a package declaration, in the code generator's current state. Whenever a method is encountered, its associated class is either fetched from the existing ones or created.

Class constructors are considered to be special methods, and are present in the input calltrace along with their type signature. They are not treated as any different than regular methods in terms of how their bodies are filled and how they're created.

Importantly, this means we do not create our own constructors, and can only instantiate a class if it has a public one: this can turn out to be an issue in rare cases where no instance of a class exists in the local context, and that no constructors are available. For instance, using the singleton design pattern forces class instances to be fetched from a specific static method instead of a constructor. But this also forces the system to be realistic and stick to the actual program's structure, instead of bypassing potential issues such as this one.

4.3 Results

The generator was tested for 10 of the 14 Java benchmarks in the AWFY suite, which exhibit a variety of behaviours, and successfully builds programs that can be executed without errors for each of them. The four that were ignored were because they made use of language features not implemented (see section 4.4.1), or in one case (`Havlak`), having a calltrace of 4.8G which made it unpractical to test alongside the others.

We'll analyse the generated program from one of the benchmarks in the suite with one of its simplest calltraces, `Mandelbrot`.

4.3.1 Output

Figure 8 shows the output of the generated program created from the `Mandelbrot` AWFY benchmarks, from the calltrace in Figure 9. Every method from the calltrace appears in the output, and the execution order is respected as well. By default, each method contains a statement that prints its name and class, so that we get visual feedback of the program performing basic operations (in this case, highlighting the generated architecture) ; but this behaviour can be turned off, and was for all the code examples in this thesis.

```
Current method: Harness.main
Current method: Harness.processArguments
Current method: Run.<init>
Current method: Run.getSuiteFromName
Current method: Run.setNumIterations
Current method: Run.runBenchmark
Current method: Mandelbrot.<init>
Current method: Benchmark.<init>
Current method: Benchmark.<init>
Current method: Run.doRuns
Current method: Run.measure
Current method: Mandelbrot.<init>
Current method: Benchmark.<init>
Current method: Mandelbrot.innerBenchmarkLoop
Current method: Mandelbrot.mandelbrot
Current method: Mandelbrot.verifyResult
Current method: Run.printResult
Current method: Run.reportBenchmark
Current method: Run.printTotal
```

Figure 8: The output of *bootleg* for the `Mandelbrot` benchmark.

```
> pub/sta ([Ljava/lang/String;)V Harness.main
> pri/sta ([Ljava/lang/String;)LRun; Harness.processArguments
> pub/con (Ljava/lang/String;)V Run.<init>
> pri/sta (Ljava/lang/String;)Ljava/lang/Class; Run.getSuiteFromName
< pri/sta (Ljava/lang/String;)Ljava/lang/Class; Run.getSuiteFromName (139715517ns)
< pub/con (Ljava/lang/String;)V Run.<init> (141135378ns)
> pub (I)V Run.setNumIterations
< pub (I)V Run.setNumIterations (39866ns)
< pri/sta ([Ljava/lang/String;)LRun; Harness.processArguments (277671449ns)
> pub ()V Run.runBenchmark
> con ()V Mandelbrot.<init>
> pub/con ()V Benchmark.<init>
< pub/con ()V Benchmark.<init> (88194ns)
< con ()V Mandelbrot.<init> (1218161ns)
> pri (LBenchmark;)V Run.doRuns
> pri (LBenchmark;)V Run.measure
> pub (I)Z Mandelbrot.innerBenchmarkLoop
> pri (I)I Mandelbrot.mandelbrot
< pri (I)I Mandelbrot.mandelbrot (231262ns)
> pri (II)Z Mandelbrot.verifyResult
< pri (II)Z Mandelbrot.verifyResult (39034ns)
< pub (I)Z Mandelbrot.innerBenchmarkLoop (494058ns)
> pri (J)V Run.printResult
< pri (J)V Run.printResult (1061280ns)
< pri (LBenchmark;)V Run.measure (1968001ns)
< pri (LBenchmark;)V Run.doRuns (3503820ns)
> pri ()V Run.reportBenchmark
< pri ()V Run.reportBenchmark (4621214ns)
< pub ()V Run.runBenchmark (15559286ns)
> pub ()V Run.printTotal
< pub ()V Run.printTotal (651724ns)
< pub/sta ([Ljava/lang/String;)V Harness.main (295037647ns)
```

Figure 9: The input calltrace generated from the real `Mandelbrot` benchmark. Note: the third value, such as `(II)Z`, describes the method's type signature.

4.3.2 Package structure

For each class, their position in the package tree is taken into account. Package names are detected and directories are created from them, and the classes in each package are generated in the correct context.

Figure 10 shows how the code class package trees differ for both real and generated code. They are very close, but Figure 10b is lacking one class from the real benchmark: the `ParseException` class. This is because exceptions are not taken into account, as they don't appear in our input calltraces which do not throw or go through `catch` blocks ; therefore, the generator has no knowledge of them.

Moreover, the generated benchmark introduces a new class, `JsonObjectSHashindexTable`. It's absent from the original tree as it's an inner class in this first one, therefore not being present in a separate file ; in the generated code, inner classes are interpreted as regular public classes, hence appear in files different from where they do in the input benchmark.

Other than these, the generator manages to recreate the same classes as the original program without fail.

```
src
|__ json
|   |__ JsonArray
|   |__ JsonLiteral
|   |__ JsonNumber
|   |__ JsonObject
|   |__ JsonPureStringParser
|   |__ JsonString
|   |__ JsonValue
|   |__ ParseException
|__ som
|   |__ Vector
|__ Benchmark
|__ Harness
|__ Json
|__ Run
```

(a) Actual code tree for the `Json` benchmark.

```
src
|__ json
|   |__ JsonArray
|   |__ JsonLiteral
|   |__ JsonNumber
|   |__ JsonObject
|   |__ JsonObjectSHashindexTable
|   |__ JsonPureStringParser
|   |__ JsonString
|   |__ JsonValue
|__ som
|   |__ Vector
|__ Benchmark
|__ Harness
|__ Json
|__ Run
```

(b) Code tree of the generated code from the `Json` benchmark.

Figure 10: Comparing the generated code to the actual program class structure. Each file is a Java `.class` file.

4.3.3 Code

TODO: The only missing section. I need to compare both real and generated benchmark in a relevant fashion, most likely with CK metrics. Figure 11 is a starting point, showing how the system can succeed. Also, [20] by A. Phansalkar et al. (uses metrics like instruction mix and techniques like PCA)

```java
public boolean innerBenchmarkLoop(final int innerIterations) {
  NBodySystem system = new NBodySystem();
  for (int i = 0; i < innerIterations; i++) {
    system.advance(0.01);
  }
  return verifyResult(system.energy(), innerIterations);
}
```

```java
public boolean innerBenchmarkLoop(int xznbt) {
  NBodySystem nbodysystem = new NBodySystem();
  nbodysystem.advance(0.54304755f);
  nbodysystem.energy();
  this.verifyResult(0.030931115f, 1355);
  return true;
}
```

```java
public boolean innerBenchmarkLoop(int ocucg) {
  NBodySystem vscku = new NBodySystem();
  vscku.advance(0.51389205f);
  double jtthj = vscku.energy();
  boolean pgwxs = this.verifyResult(jtthj, ocucg);
  return pgwxs;
}
```

Figure 11: A real method from the AWFY suite and two methods generated by two different versions of *bootleg*. From top to bottom: the real method, a non context-aware recreation using random values, and a recreation that takes parameter and method return values into account.

4.4 Limitations

While it was written with generality and portability in mind, my code generator doesn't account for certain possibilities which can prevent correct generation.

4.4.1 Language specifics

Generating code is time intensive, therefore many aspects of the Java language weren't taken into account because of time constraints.

Inheritance is the best example, and was entirely ignored: as shown in Figure 12, our system chooses to instantiate a subclass instead of using an already available instance of its parent class. This is because of two things:

- Inheritance being ignored means no class extends from any other, hence the generator does not know of the relationship that's supposed to exist between the two classes in question.

- Our calltraces use method declarations based on the class' internal name during runtime, which return the actual subclass instance's name even if the method in question is defined in a parent class. Hence, we have `subclass.parentMethod()` and not `parentclass.parentMethod()` in our calltraces.

This makes it so that the parent class will be created by our system if it's present in the calltrace, but since its methods are never queried from an actual instance of it, the class will be empty. In our case, the `Benchmark` class only contains an empty constructor (called from `NBody`'s constructor), because of an implicit `super()` call added by Java in the original program, hence present in the calltrace. This class goes completely unused otherwise.

```
private void measure(Benchmark mxupn) {
  NBody nbody = new NBody();
  nbody.innerBenchmarkLoop(56);
  this.printResult(8635);
}
```

Figure 12: An example of inheritance not being taken into account in this generated code. In the real program, `NBody` is a subclass of `Benchmark` and `innerBenchmarkLoop()` is defined in `Benchmark`, hence instantiating an `NBody` class should be unnecessary.

Instead, information about class hierarchies should be fed to the generator so that it may realize when it can substitute one class for its parent. This could be in the form of a static call graph, depicting every relationship between every class/subroutine, which could be generated with an algorithm such as RTA [19].

Inner classes are also not implemented. While it's clear which class is inner because they appear in our calltraces with the format `parentClass-$innerClass` (example: `List$Element`), they are instead just interpreted as regular public classes, which causes no building or runtime issues.

Static initializers are detected but modified to not potentially prevent execution. Each one is simply replaced by a public static method which is invoked in the caller. In reality, they should be put in static initializer blocks that don't have to be invoked directly by any method ; but their behaviour is relatively well simulated with this adjustment.

Other possible improvements in this regard include using abstract classes and interfaces, or even generics.

4.4.2 Generation

In terms of more general and standard programming language features, some do not appear in the generated code for various reasons ; most notably, because the input data didn't contain enough information about these features to represent them realistically in the generated benchmark.

Conditionals are vital for allowing the management of control flow, but aren't implemented in this first version of my code generator. Since no information is provided about which instructions are conditional ones, how many statement blocks there are and what their content is, these cannot be handled with the current input data. Providing this information in an interpretable format would yield a lot of information about the real codebase, which my project voluntarily tried to avoid for this first iteration to see how much a recreation of a program being given very limited inputs could achieve.

With the generator's current version, since we only rely on a single execution trace for a program, the behaviour is set and doesn't need branching paths no matter what. If conditionals are to be added, the system must also be aware of every possible trace the code can have, such as by relying on a static call trace.

Loops are a highly important feature of programming in general and are present in nearly every program: in the AWFY benchmarks, 12% of all methods contained at least one loop, and for the fetched codebases, that number was 5.8%. They have a great impact on program behaviour by causing certain parts of the code to be executed several times, which can easily increase the time complexity of the program by orders of magnitude.

Therefore if we want to strive for realism, they should be simulated in our generated code, but they currently are not. In fact, multiple calls to the same method are ignored after the first one has been taken into account, as shown in Figure 13: the reason for this is specifically because of loops, to avoid writing hundreds of method calls if they happen in a simple loop body that is called hundreds of times in a row in the input program.

With four calls to the same method in a row in this example, one could assume this takes place in a simple loop in the original program. However, this is in fact four fields being initiated, each relying on a different random value.

```
> con (Lsom/Random;)V Bounce$Ball.<init>
> pub ()I som/Random.next
< pub ()I som/Random.next (52262ns)
> pub ()I som/Random.next
< pub ()I som/Random.next (22503ns)
> pub ()I som/Random.next
< pub ()I som/Random.next (19378ns)
> pub ()I som/Random.next
< pub ()I som/Random.next (19456ns)
< con (Lsom/Random;)V Bounce$Ball.<init> (896799ns)
```

```
Bounce$Ball(Random poslx) {
  int golxv = poslx.next();
}
```

Figure 13: Multiple calls to the same method in the same scope are ignored after the first one is interpreted. At the top, the input calltrace, at the bottom, the generated method.

This shows how detecting loops from a simple calltrace is not trivial. The number of loops in a given method could also be provided as additional input, but detecting where loop blocks begin and end could still be a problem. For instance, in Figure 14, even the knowledge of the existence of a loop in the concerned method will not make it easy to extract a pattern simply by looking for a repeated code block ; unless the generator detects they're all instances of the same class, no pattern can be extracted, and even with that knowledge inferring what the input program did exactly from its limited information is complex.

The logic by which each subclass is chosen here is hard to deduce. Apart from a field, these benchmark subclasses can only be fetched from the input array, but the index by which they're accessed isn't easily inferable. It could be one of the two input integers, or the return of the `nextInt()` method : if it's the former, then it cannot know if they were used as is or if operations were applied to them, and if so, which they were. Additionally, if it's the latter, that `nextInt()` function took an integer as a parameter to which the same problem applies: it does not know what its input was. Overall, in a case like this where an array access is present, it's likely to make wrong decisions and cause an exception by attempting to access a value out of its

```
void runBenchmarks(Benchmark[] benchmarks, int benchmarksNbr, int val) {
  Random rn = new Random();
  for (int i = 0; i < benchmarksNbr; i++) {
    int randBenchmarkId = rand.nextInt(benchmarksNbr + 1);
    benchmarks[randBenchmarkId].overridenMethod(val);
  }
}
```

```
> pri/sta ([LBenchmark;II)V Harness.runBenchmarks
> pub/con ()V java/util/Random.<init>
< pub/con ()V java/util/Random.<init> (82698ns)
// Iteration start:
> pub (I)I java/util/Random.nextInt
< pub (I)I java/util/Random.nextInt (29504ns)
> pub (I)Z NBody.overridenMethod
...
< pub (I)Z NBody.overridenMethod (1288011ns)
// Iteration start:
> pub (I)I java/util/Random.nextInt
< pub (I)I java/util/Random.nextInt (24833ns)
> pub (I)Z Mandelbrot.overridenMethod
...
< pub (I)Z Mandelbrot.overridenMethod (366524ns)
// Iteration start:
...
< pri/sta ([LBenchmark;II)V Harness.runBenchmarks (11648335ns)
```

Figure 14: A custom method (not present in the AWFY suite) and its associated calltrace. `NBody` and `Mandelbrot` extend `Benchmark`.

bounds.

Hence loops are difficult to implement without causing many issues ; not to mention the possibility of infinite loops causing the program to never halt, or the question of how to decide on what condition the loop should be based on. Compare this to them being ignored, in which case we would simply instantiate each subclass consecutively and not run into any issues. However, because of their importance, they're a vital part of any plausible program recreation, and the generator should be improved to take them into account in some way.

Fields are also an important part of Java programs, and OOP in general, as shown in Figure 3. As with conditionals, loops and inheritance, a simple calltrace is not enough information for the generator to make informed decisions ; without additional data, it would not know which variables need to end up in fields and which should simply stay local variables.

A simple solution would be to specify the type of all the fields present in each class individually, and feed it as additional input to the generator. When it would normally instantiate a new variable of a given type needed for an operation, it would instead look up the class' fields and see whether or not there is one of the required type. This is already close to the local variable / method parameters lookups done by our system.

As to when to initialize the fields, we could keep track of whether they're initialized or not, and not fetch them in the aforementioned fashion if they're not. If that's the case, then we'd instantiate a new local variable instead as usual, which could then be set to the adequate field. This would avoid run-time errors, as this would guarantee no null pointers would ever be queried from in the case of field objects.

More control over field usage could be acquired if information regarding the number of field reads and writes were provided for each method, as two separate pieces of information. The former would let us only look up fields when necessary, saving time, and the latter would help us know when fields are initialized or operated on.

5 Future works

My work was designed to be reused in the future as a stepping stone, and was always meant to be improved in several ways ; most importantly, by being combined with other systems to achieve automatic benchmark generation, tailored to huge codebases.

5.1 Improvements to the generation

This was described in section 4.4 in detail: my code generator is functional on most of the AWFY benchmarks, but may not work with other input programs that rely heavily on features not implemented in my work (generics, for instance). Not being able to generate programs that compile correctly from any input makes it non *guaranteed-legal* [17], but enforcing this property is hard to achieve. Getting it to run with every single benchmark in the AWFY suite could be a good starting point: for example, the `DeltaBlue` benchmark has an issue related to inheritance since one of its method takes an interface as a parameter, which is never called hence is unknown to the generator.

Past getting it to run with different inputs, I've also described ways to improve the realism in its generation (loops and field usage, for example), and a lot of improvements could be achieved in this regard ; especially with the large variety in real codebases, making use of possibly obscure language features or utilizing some I've mentioned in an unconventional way.

5.2 Extending it to other languages

While that was one of my original goals, I chose to focus on Java specifically because of time constraints and the reasons described in section 1.2. The code generation doesn't make use of advanced Java-specific features, however, therefore it can be ported to any object oriented language I'm aware of.

Imperative languages can also be considered: since fields are currently not used, classes are nothing more than a collection of methods that can therefore potentially be done with and replaced with simple modules/source files to contain these functions. The same may apply to functional languages.

5.3 Usage of the fetched static metrics

Because of the switch to focusing on the AWFY benchmark suite instead of large codebases, the data I acquired on mined large codebases went unused for now. Since my generator works with an input calltrace, the structure of the program that has to be generated is inferable from the provided data, and information about the general organization of various programs can not be used in a relevant fashion. The relevancy of these metrics should appear when combined with a bigger next step: not having to take in a calltrace at all.

5.4 Removing the calltrace input

As mentioned in section 3.4, the static metrics I acquired were originally meant to be used in a statistical approach, that was not relied upon for this project to focus solely on a calltrace approach instead. This information is still relevant because of its insights into real large codebases, and therefore needs to be fused with our current code generator in some way.

Perhaps by finding ways of generating our own calltraces from a variety of real ones, as well as relying on the static metrics to define structural information such as each class' number of methods/fields. Data mining techniques could potentially be used to interpret the large amount of data, or a machine learning approach could be attempted ; this is a complex question that could warrant another thesis by itself, hence why I refer to it as a possible next step.

6 Conclusion

I've looked into ways of generating benchmarks from large codebases automatically, acquired insightful metrics about real large programs' structure, did a significant amount of the code generation research and implementation, as well as provided theories for where to go next to achieve this overarching goal. While generating code with valid syntax that also executes correctly turned out to be more complex than anticipated, I've still put in enough work to get a first version I'm satisfied with.

The metrics I've acquired regarding large codebases yielded information about their structure that can be used to generate more realistic code in the future. My program generator is functional with many different inputs, and much of the heavy lifting of code generation has been achieved through it ; the next step is combining those two in an insightful and relevant way, to actually generate benchmarks from large codebases which will hopefully simulate real ones in a satisfactory manner.

It's uncertain whether benchmarks generated in this unusual fashion could end up being useful for language implementations, as opposed to classic man-made ones, but I believe it's an idea worth exploring ; and this theory can only be proven or denied if more work is put towards this objective, using my work as a stepping stone.

References

[1] Ralph Peters and Andy Zaidman. Evaluating the lifespan of code smells using software repository mining. In Tom Mens, Anthony Cleve, and Rudolf Ferenc, editors, *CSMR*, pages 411–416. IEEE Computer Society, 2012.

[2] Jalerson Lima, Christoph Treude, Fernando Marques Figueira Filho, and Uirá Kulesza. Assessing developer contribution with repository mining-based metrics. In Rainer Koschke, Jens Krinke, and Martin P. Robillard, editors, *ICSME*, pages 536–540. IEEE Computer Society, 2015.

[3] Andy Zaidman, Bart Van Rompaey, Arie van Deursen, and Serge Demeyer. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. Technical Report TUD-SERG-2010-035, Software Engineering Research Group, Delft University of Technology, 2010.

[4] Petr Maj, Konrad Siek, Alexander Kovalenko, and Jan Vitek. CodeDJ: Reproducible Queries over Large-Scale Software Repositories. In Anders Møller and Manu Sridharan, editors, *35th European Conference on Object-Oriented Programming (ECOOP 2021)*, volume 194 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 6:1–6:24, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[5] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. Germán, and Daniela E. Damian. The promises and perils of mining github. In Premkumar T. Devanbu, Sung Kim, and Martin Pinzger, editors, *MSR*, pages 92–101. ACM, 2014.

[6] Saad Razzaq, Fahad Maqbool, Bilal Anjum, Samreen Zafar, Umme Laila, and Faiza Noor. The challenges & case for mining software repositories. In Sio Iong Ao, Oscar Castillo, Craig Douglas, David Dagan Feng, and Jeong-A Lee, editors, *IMECS*, Lecture Notes in Engineering and Computer Science, pages 734–739. Newswood Limited, 2007.

[7] H. J. Curnow and Brian A. Wichmann. A synthetic benchmark. *Comput. J.*, 19(1):43–49, 1976.

[8] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. Synthesizing benchmarks for predictive modeling. In Vijay Janapa Reddi, Aaron Smith, and Lingjia Tang, editors, *CGO*, pages 86–99. ACM, 2017.

[9] Andrés Goens, Alexander Brauckmann, Sebastian Ertel, Chris Cummins, Hugh Leather, and Jeronimo Castrillon. A Case Study on Machine Learning for Synthesizing Benchmarks. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2019, page 38–46, New York, NY, USA, 2019. Association for Computing Machinery.

[10] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In Mary W. Hall and David A. Padua, editors, *PLDI*, pages 283–294. ACM, 2011.

[11] Joshi, Ajay and Eeckhout, Lieven and John, Lizy. The Return of Synthetic Benchmarks. pages 1–11, 2008.

[12] Gregor Richards, Andreas Gal, Brendan Eich, and Jan Vitek. Automated construction of javascript benchmarks. In Cristina Videira Lopes and Kathleen Fisher, editors, *OOPSLA*, pages 677–694. ACM, 2011.

[13] Shyam R. Chidamber and Chris F. Kemerer. Towards a metrics suite for object oriented design. *SIGPLAN Not.*, 26(11), 1991.

[14] Maurício Aniche. *Java code metrics calculator (CK)*, 2015. Available in https://github.com/mauricioaniche/ck/.

[15] Stefan Marr, Benoit Daloze, and Hanspeter Mössenböck. Cross-Language Compiler Benchmarking—Are We Fast Yet? In *Proceedings of the 12th Symposium on Dynamic Languages*, DLS'16, pages 120–131. ACM, November 2016. (acceptance rate 55

[16] Lukás Marek, Alex Villazón, Yudi Zheng, Danilo Ansaloni, Walter Binder, and Zhengwei Qi. Disl: a domain-specific language for bytecode instrumentation. In Robert Hirschfeld, Éric Tanter, Kevin J. Sullivan, and Richard P. Gabriel, editors, *AOSD*, pages 239–250. ACM, 2012.

[17] Yannis Smaragdakis. Program generators and the tools to make them. In Roberto Giacobazzi, editor, *SAS*, volume 3148 of *Lecture Notes in Computer Science*, pages 19–20. Springer, 2004.

[18] Roya Hosseini and Peter Brusilovsky. Javaparser; a fine-grain concept indexing tool for java problems. In Erin Walker and Chee-Kit Looi, editors, *AIED Workshops*, volume 1009 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2013.

[19] Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. In Mary Beth Rosson and Doug Lea, editors, *OOPSLA*, pages 281–293. ACM, 2000. SIGPLAN Notices 34.

[20] A. Phansalkar, A. Joshi, L. Eeckhout, and L. K. John. Measuring program similarity: Experiments with spec cpu benchmark suites. In *IEEE International Symposium on Performance Analysis of Systems and Software, 2005. ISPASS 2005.*, pages 10–20, March 2005.