On Programming Variability with Large Language Model-based Assistant
Mathieu Acher, José Galindo Duarte, Jean-Marc Jézéquel

To cite this version:

HAL Id: hal-04153310
https://inria.hal.science/hal-04153310
Submitted on 6 Jul 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License
On Programming Variability with Large Language Model-based Assistant

Mathieu Acher  
University of Rennes, INSA, IRISA,  
Inria, IUF  
Rennes, France  
mathieu.acher@irisa.fr

José Galindo Duarte  
University of Sevilla  
Sevilla, Spain  
jagalindo@us.es

Jean-Marc Jézéquel  
University of Rennes, IRISA, Inria, IUF  
Rennes, France  
jean-marc.jezequel@irisa.fr

ABSTRACT

Programming variability is central to the design and implementation of software systems that can adapt to a variety of contexts and requirements, providing increased flexibility and customization. Managing the complexity that arises from having multiple features, variations, and possible configurations is known to be highly challenging for software developers. In this paper, we explore how large language model (LLM)-based assistants can support the programming of variability. We report on new approaches made possible with LLM-based assistants, like: features and variations can be implemented as prompts; augmentation of variability out of LLM-based domain knowledge; seamless implementation of variability in different kinds of artefacts, programming languages, and frameworks, at different binding times (compile-time or run-time). We are sharing our data (prompts, sessions, generated code, etc.) to support the assessment of the effectiveness and robustness of LLMs for variability-related tasks.

KEYWORDS

variability, programming, software product lines, generative AI, large language model

ACM Reference Format:

1 INTRODUCTION

Synthesizing variants with software is a holy grail in many application domains and fields. With an effective mastering of variability, organizations can explore different designs, ideas, and hypotheses while offering custom solutions that fit to a variety of contexts, requirements, and users. One of the toughest challenges is to manage the complexity, being essential or accidental, of modern software [8]. Variability further increases this software complexity since developers should program, maintain, and test multiple features, code variations, and an exponential number of possible variants [4, 26, 32, 37, 47]. During the last decades, numerous languages, paradigms, and technologies have been developed to support systematic transformation of problem-level abstractions to software implementations. From the early days of generative programming [16] and software product line (SPL) engineering [4, 37, 47], the goal has been to automatically generate variants from a specification written in one or more textual or graphical domain-specific languages.

In this short and exploratory paper, we defend the idea that large language models (LLMs) can be leveraged to support the programming of variability and realize the early ambition of generative programming and SPL engineering. As experimented and reported in this paper, an emerging pattern is that LLMs act as a new variability compiler capable of transforming a high-level specification (prompt) into variable code, features, generators, configurable systems, or SPLs written in a given technological space.

LLMs are gaining momentum and are capable of tackling more and more problems from linguistics, maths, commonsense reasoning, biology, physics, etc. BERT [18], GPT-3 [9], PaLM [15], to name a few, are scaling to support a variety of tasks such as text generation, question-answering, text classification, arithmetic on numbers, and many others [23, 52]. In software engineering, code assistants based on LLMs have been proposed like Alphacode, CodeParrot or Codex [14, 21, 35, 36] and are now deployed at scale for supporting programmers, such as GitHub Copilot [24] or ChatGPT [46]. Based on prompts composed of an informal instruction written in natural language of a task that may include existing code (LLM) synthesizes programs and possibly explanations.

Our contribution is to show how LLMs can be concretely and originally used for programming software variability. To the best of our knowledge, LLMs have not been considered in this context. Statistical machine learning has been applied in SPL and variability engineering, with different use cases [3, 25, 30, 33, 34, 38, 43, 44, 48, 53, 54, 59–63]: performance prediction, optimization, specialization, debugging, finding of configuration-related bugs, etc. In this paper, we consider LLMs that are based on transformers, a deep learning architecture originally invented in the context of natural language processing (NLP). The use of NLP within SPL engineering has caught attention, but mostly for domain analysis and requirements engineering [2, 6, 11–13, 17, 22, 27, 29, 42, 45, 50, 51], not for realizing variability. We report on three cases where variability has been programmed with LLM-based assistant:

- **Variability for reproducibility and floating-points (Section 2):** We revisit an issue raised in [7] to illustrate how developers can (1) implement features through high-level
We first write a prompt to vary the types of the variables $x$, $z$. A numerical program with floating-point computations may give different answers depending on the execution hardware and the compiler. We further show how this variability can be seamlessly realized in (1) different programming languages (2) while choosing the suited mechanisms (conditional compilation, templates, feature toggling, command-line parameters).

- **Variability implementations of a classical Hello world (Section 3):** We first show how developers can implement the whole application and different features out of a single prompt. We further show how this variability can be envisioned. Section 5 concludes with the challenge of carefully assessing the robustness of current and next-generation LLMs for variability-related tasks.

  **Data and companion Web page [57].** In the rest, we exclusively report on sessions with ChatGPT-4 [46], using the Web chat interface of OpenAI. We also used other LLMs (ChatGPT-3, NellGPT-3.5, Claude [58]) as part of variability programming sessions from January 2023 to April 2023. Owing to the space limitation and scope of the paper, we do not comment on the relative merits or limits of the different LLMs. Yet, we are sharing all data (prompts, sessions, generated code, etc.) to support further inquiry: https://github.com/acherm/progvary-withgpt/.

2 EXPLORING VARIABILITY IN REPRODUCIBLE SCIENCE WITH LLM

A numerical program with floating-point computations may give different answers depending on the execution hardware and the compilation. We revisit such an example (see Listing 1) documented in [7]. We automatically explore how the variability within the code and the compiler affects the reproducibility of results.

### 2.1 Features as prompts

We first write a prompt to vary the types of the variables $x$, $y$, and $z$. We leverage LLM to generate three variations, implemented as conditional compilation within the C code.

I have the following program. I’d like to create variations over this program using conditional compilation. Specifically, I’d like to create a first variation that consists in replacing the type long double by double... and another alternative variation that consists in replacing long double by float.

```c
#include <stdio.h>
int main() {
    real x = 1.0;
    long double y = 0x1p-53 + 0x1p-64;
    long double z = x + y;
    printf("z = %a\n", z); // Print the value of z
    return 0;
}
```

Listing 1: Original C program (based on [7])

(LLL) You can use the preprocessor directives `#ifdef`, `#elif`, and `#endif` to create conditional compilation. Here’s an example that shows how to create the variations you mentioned.

```c
#include <stdio.h>

// Define the type to use as an alias
#ifdef USE_DOUBLE
    typedef double real;
#else
    typedef float real;
#endif

typedef long double real;

int main() {
    // Declare and initialize variables
    real x = 1.0;
    real y = 0x1p-53 + 0x1p-64;
    real z = x + y;
    // Print the value of z
    #ifdef USE_DOUBLE
        printf("z (using double) = %a\n", z);
    #elif defined(USE_FLOAT)
        printf("z (using float) = %a\n", z);
    #else
        printf("z (using long double) = %a\n", z);
    #endif
    return 0;
}
```

Listing 2: Generation of a configurable C program

The obtained code (Listing 2) is syntactically correct, can be compiled and executed, and constitutes a readable, well-designed, and extensible implementation. Specifically, there is a correct use of C preprocessor (CPP) directives inside the C program. The generalization of the variation point (type) using ‘typedef’ is a good practice and remarkable, since not straightforward when looking at the prompt. The instructions are not explicitly stating how to implement this variation, yet, the LLM-based assistant finds how to properly factor out the code. Furthermore, there is a customization of the behavior depending on options (print statements).

Beyond the technicalities, this session shows remarkable abilities of an LLM-based assistant. There is a translation of an informal specification of variations (features) into executable and configurable code units. A programmer can focus on the problem space, describing what are expected variations, whereas the LLM synthesizes a working program into the solution space. Hence, we can envision to follow the motto features as prompts. Though the C program is arguably short and simple, LLM-based assistant has identified where to apply variations and which mechanisms of the targeted language should be used. LLMs can adapt to an existing code base and apply variations. There is an automated detection of
which technology (here: CPP) to use. It is also worth noticing that directives are correctly mixed within another language (C).

2.2 Generating and augmenting variants

The prompt leads to a first version of the Python generator.

```python
import subprocess
import pandas as pd

def compile_and_run ( compiler , flags ) :
    commands = [  
        f"{ compiler } { flags } -o original approx .c",
        f"{ compiler } { flags } -DUSE_DOUBLE -o double_variation approx .c",
        f"{ compiler } { flags } -DUSE_FLOAT -o float_variation approx .c",
    ]
    results = []
    for cmd in commands:
        output = subprocess.check_output(f"{cmd}" , shell=True).decode().strip()
        results . extend (output)
    return results

def main () :
    configurations = [  
        ( 'gcc' , '') , ( 'gcc' , '-mfpmath=387' ),
        ( 'gcc' , '-ffast-math' ),
        ( 'gcc' , '-fno-signed-zeroes' ),
        ( 'gcc' , '-fno-rounding-math' ),
        ( 'clang' , '') , ( 'clang' , '-ffast-math' ),
        ( 'clang' , '-fno-signed-zeroes' ),
        ( 'clang' , '-fno-rounding-math' ),
    ]
    data = []
    for compiler , flags in configurations:
        results = compile_and_run ( compiler , flags )
        data . extend (
            [{ 'compiler' : compiler , 'flags' : flags , 'type' : 'original' , 'result' : results[0] }],
        )
    df = pd.DataFrame ( data )
    df . to_csv ( 'results . csv ' , index=False)
```

We then refine and ask:

Can you think about gcc flags or clang flags that could be combined/used and have an influence on floating point operations? Please implement them in the Python code.

The resulting program is depicted in Listing 3.

Listing 3: Generated Python code (generator that compiles the previous C-code using different directives, compilers, and compiler flags; synthesize results in a CSV)

The execution of the Python program does produce a correct CSV file that can be analyzed. We found that the use of gcc and -mfpmath=387 together with double (a specific variant) leads to a result different from other variants (as originally shown in [7]). Overall, it is possible to explore a variant space in a few prompts.

This session provides interesting insights. Compilation instructions (CPP directives, C compiler flags, C compiler used) are correctly assembled: The combination of Python code and C/CPP code is effective. The Python program acts as a generator to automatically synthesize C variants. Furthermore, it is possible to obtain an automated execution and organization of results’ variants (here: in a dataframe). Besides, domain knowledge is leveraged to guide and synthesize new variability (here: compiler flags). What is worth noticing is that the knowledge is actionable and directly integrated into the Python generator. The programmer workflow is simplified since she has only to review proposed changes at the code level. Finally, there is an extensible organization of the Python code for adding new features and configurations. In particular, a feature model can be well elaborated to further scope or augment the configuration space and eventually sample some valid configurations fed to the Python generator. Overall, developers can interact with LLMs to ask for new variability knowledge and then manually or directly extend the Python generator.

3 A FAMILY OF CONFIGURABLE HELLO WORLDS WITH LLM

We revisit a configurable “Hello world” application traditionally used in SPL courses [1] and documented, e.g., in Chapter 9 of FeatureIDE book [40].

3.1 Flexibility in programming languages

Out of a single prompt, LLMs can synthesize a complete and working C++ solution [57], retrieving the original implementation proposed in [40]. For the sake of illustration, we choose to show a configurable “Hello world” in another language, Rust, that has built-in support for conditional compilation. Remarkably, we use the same prompt as with the C++ except we replace C++ by Rust (in bold). Listing 4 correctly encodes the 16 possible variants that can be compiled and executed. The session below shows the ability of LLMs to synthesize variability in different programming languages with slight modifications of the original instructions.

```
fn main() {
    #[cfg(feature = "hello")]
    // write a configurable Rust program that implements four features and their selections, namely, Hello, Beautiful, Wonderful, and World... If the feature is selected, then we print the name of the feature. For instance, we could well print "Hello Beautiful World", "Hello World", "Hello", "World" etc. The four features should be implemented as compile-time features.
```

(LLL) [...] you can use the 'cfg' attribute to enable or disable the features at compile-time [...] in your 'Cargo.toml' file [...].
As an additional illustration, we implemented the "Hello World" application in JavaScript, relying this time on feature toggles [31, 41, 55]. A simple adaptation of the original prompt was sufficient [57]:

Write a configurable JavaScript program that implements […] The four features should be implemented as feature toggles. I’d like to use Unleash, a feature toggle service, integrated with Gitlab

```javascript
const { initialize } = require('unleash-client');
const unleash = initialize({
  app: 'my-app-name',
  instanceId: 'my-instance-id',
  url: 'https://unleash.example.com/api/',
});

function printFeatures() {
  let output = '';
  if (unleash.isEnabled('hello')) {
    output += 'Hello
';
  }
  if (unleash.isEnabled('beautiful')) {
    output += (output ? ' ' : '') + 'Beautiful
';
  }
  if (unleash.isEnabled('wonderful')) {
    output += (output ? ' ' : '') + 'Wonderful
';
  }
  if (unleash.isEnabled('world')) {
    output += (output ? ' ' : '') + 'World
';
  }
  console.log(output);
}
unleash.on('ready', printFeatures);
unleash.on('update', printFeatures);
```

Listing 6: generated JavaScript code (feature toggles)

3.2 Flexibility in variability mechanisms

Obviously, the instructions for C++ and Rust will not work for programming languages that do not support conditional compilation like Java or JavaScript. In this case, more effort should be done at the prompt level and the instructions should be more specific (in bold). Instead of using the Munge or Antenna preprocessors (as done in [40]), we ask to leverage an off-the-shelf templating engine. Out of the prompt below, LLMs can synthesize valid Java variants using a template-based code generator.

Write a configurable Java program that implements […] The four features should be implemented as compile-time features. As there is no preprocessor and built-in support for conditional compilation in Java, it is needed to leverage a templating engine for customization, acting as a preprocessor for Java. You can use Jinja (Python)

(LLL) To create a configurable Java program that implements the four features as compile-time features using Jinja (Python) templating engine, you need to follow these steps: ... Create a Jinja template (main_template.java.j2) for your Java program [...] ... Create a Python script (generate_java.py) that will generate the Java source file based on the provided features:

```java
public class Main {
    public static void main(String[] args) {
        // if (features.Hello)
        System.out.print("Hello ");
        // endif
        // if (features.Beautiful)
        System.out.print("Beautiful ");
        // endif
        // if (features.Wonderful)
        System.out.print("Wonderful ");
        // endif
        // if (features.World)
        System.out.print("World");
        // endif
        System.out.println();
    }
}
```

Listing 5: Generated Java code (template)

Listing 5 is again correct, and the Python program (not showed due to space limitations) can correctly synthesize the 16 variants. As an additional illustration, we implemented the "Hello World" application in JavaScript, relying this time on feature toggles [31, 41, 55]. A simple adaptation of the original prompt was sufficient [57]:

Write a configurable JavaScript program that implements […] The four features should be implemented as feature toggles. I’d like to use Unleash, a feature toggle service, integrated with Gitlab

```javascript
const { initialize } = require('unleash-client');
const unleash = initialize({
  appName: 'my-app-name',
  instanceId: 'my-instance-id',
  url: 'https://unleash.example.com/api/',
});

function printFeatures() {
  let output = '';
  if (unleash.isEnabled('hello')) {
    output += 'Hello
';
  }
  if (unleash.isEnabled('beautiful')) {
    output += (output ? ' ' : '') + 'Beautiful
';
  }
  if (unleash.isEnabled('wonderful')) {
    output += (output ? ' ' : '') + 'Wonderful
';
  }
  if (unleash.isEnabled('world')) {
    output += (output ? ' ' : '') + 'World
';
  }
  console.log(output);
}
unleash.on('ready', printFeatures);
unleash.on('update', printFeatures);
```

Listing 6: generated JavaScript code (feature toggles)

4 FROM UNFAMILIAR CODE TO END-USER CUSTOMIZATION WITH LLM

We face the challenge of customizing a cat realized in TikZ, a widely used package of \LaTeX{} for drawing images. A visual rendering of the cat is the image in the bottom right of Figure 1. Listing 7 shows an excerpt of the code. The original TikZ code is not subject to variation and here to produce a unique cat. The 40 lines of code are arguably hard to understand: there is no comment for describing the different parts of the cat while TikZ code combines different low-level primitives like fill, ellipse, etc. The code was found in a public forum [56], created by a TikZ expert, without many explanations. For all these reasons, we consider that the TikZ code is unfamiliar – something that may happen when onboarding on a complex project, when experimenting with new languages, frameworks, or libraries.

Hence, the challenge is as follows: transforming the unfamiliar, non-variable code into something configurable by end-users. The
4.1 Feature location and implementation

As a line-by-line technical explanation of the TikZ code was unhelpful, we directly ask what kinds of variations we want – features as prompts.

Here is a TikZ code... I’d like to change the color of the tail

\begin{verbatim}
\tikzset{\}
\end{verbatim}

Listing 8: Suggested variation of TikZ code

The answer is remarkable for two reasons: (1) the LLM has been able to locate features out of the TikZ code. That is, the visual and high-level description (“tail”) has been mapped to a precise line of code; (2) the proposed variation makes sense and is syntactical correct, only focusing on the parts that need to be modified (the comment “The rest of the code remains the same” has been truly generated by the LLM).

We can locate and implement other features related to the central body of the cat, the paws, the eyes, etc. We can play with colors but also with the scale of visual elements, increasing or decreasing their size. Hence, a series of prompts can be fed to an LLM that then returns the piece of code to modify and the suggested modification corresponding to a feature. To speed up the process, it is possible to ask LLMs to discover and implement many features with, e.g.:

change as much as possible elements, applying different styles, colors, and transformations. Give me the new TikZ code and explain me the changes

The generated code is available [57]. The counterpart is to lose control over the identified features and the kinds of variations. Another drawback of letting LLM performing the location of features is that the mapping may be incorrect. Despite a positive experience here with the cat TikZ, replications over different TikZ code representing squid, chimpanzee, or dogs show that LLMs are in fact inaccurate (details in the companion web page). LLMs can be lucky to guess which portion of code is associated to a feature, but clearly not 100% of the time. It is problematic since then the mapping between features and code element should then be revised, after manual investigation.

4.2 Engineering a generator

Once individual features have been implemented, there is then the need to combine all variations together. As TikZ has no obvious built-in mechanisms to support variation, we implement a template-based code generator with LLM. Crucially, the prompt comes after the series of prompts for locating and implementing features (9 features were identified at the time of the session). Hence, there is a context that the LLM can leverage to synthesize a generator.

Listing 9 shows an excerpt of the template.

Can you write a Python program that would apply the 9 proposed changes? I recommend using a templating engine in such a way the 9 changes can be applied and configured for different elements, colors, and transformations.
of a cat, while the server side internally calls the template-based generator and renders the images in SVG.

5 CONCLUSION

In this exploratory paper, we showed how LLMs can assist developers in implementing variability in different programming languages (C, Rust, Java, TikZ, etc.) and mechanisms (conditional compilation, feature toggles, command-line parameters, template, etc.). With `features as prompts`, there is hope to raise the level of abstraction, increase automation, and bring more flexibility when synthesizing and exploring software variants. However, there is a major barrier: LLMs are by construction stochastic, non-determinist and highly sensitive to prompt variations — and so are corresponding implementations of features and variability.

**Future work.** In particular, LLMs can produce different programs out of the same exact prompt. Some variations of prompts, temperature, or targeted programming language can also significantly influence the outcome (see, e.g., [10, 20, 39]). Unfortunately, some of these programs can be incomplete or simply wrong, introducing serious variability errors. In any case, we are well-aware that current LLM-based technology cannot be used in a one-shot setting and without supervision. For instance, we have reported at the end of Section 4.1 some limits of LLMs. In general, we have collected preliminary evidence of these limitations through the *replication* of programming sessions with different LLMs. We are sharing our data [57] to support further inquiry, and we encourage the SPL community to contribute. There are additional aspects of LLMs to closely monitor such as intellectual property, hallucination, or explainability [19]. Overall, LLMs act as assistant that recommend some possible implementations of variability. It is the role of programmers to review the recommendation and either selects another recommendation or manually implement variability. We are seeing two major research directions.

**A variability benchmark for LLM-based assistant.** It is needed to characterize and quantify the ability of LLMs to realize variability programming tasks, for example, determining the quality of generated solutions until finding a satisfactory one (precision and recall) or the robustness of LLMs w.r.t. prompt variations. Most of the evaluations of LLM-based code assistant rely on algorithmic problems like HumanEval [14], APPS [28], Codenet [49], MBPP or MathQA-Python [5], with rather short prompts and programs written in a single language (e.g., Python). In our case, we target comprehensive programming scenarios that (1) require combining and possibly refining different features and variability. The generated programs are most of the time the result of combining and possibly refining different prompts to the LMM. The informality of the natural language is a strength, but can also backfire and lead to ambiguous interpretation and thus incorrect programs. A possible approach is to devise a dedicated variability language for capturing the effective prompts’ patterns. Another approach is to specialize LLMs (e.g., through fine-tuning) when performing variability-related tasks.

**REFERENCES**


