On Programming Variability with Large Language Model-based Assistant

Mathieu Acher, José Galindo Duarte, Jean-Marc Jézéquel

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:
Mathieu Acher, José Galindo Duarte, and Jean-Marc Jézéquel. 2023. On Programming Variability with Large Language Model-based Assistant. In *Proceedings of 27th ACM International Systems and Software Product Lines Conference (SPLC'23).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

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

SPLC'23, August 28 – September 1, 2023, Tokyo, Japan

specification in natural languages (prompts); (2) realize a Python generator to synthesize and observe variants on top of the featured C program; (3) augment the feature set and variability leveraging domain knowledge of LLM and integrated into the generator.

- **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 seamlessly realized in (1) different programming languages (2) while choosing the suited mechanisms (conditional compilation, templates, feature toggling, command-line parameters).
- **Transforming an unfamiliar code with an end-user, Web-based customization tool (Section 4)** We report on the use of LLM for locating and implementing features, and for realizing a template code generator out of a low-level and non-variable TikZ code.

For each case, we summarize and describe interactive sessions with LLM-based assistants, including prompts and generated programs. The generated code examples provided in the paper are the result of refining prompts and interactively combining several queries to the LMM. We demonstrate the transformative potential of LLMs when implementing variability. In a sense, new ways of programming 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, ChatGPT-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() {
    // Declare and initialize variables
    long double x = 1.0;
    long double y = 0x1p-53 + 0x1p-64;
    long double z = x + y;

    // Print the value of z
    printf("z = %a\n", z);
    return 0;
}
```

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

> (LLM) 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;
#elif defined(USE_FLOAT)
    typedef float real;
#else
    typedef long double real;
#endif
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) = %La\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.

> Write a Python script that compiles and executes the three variants above with and without the -mfpmath=387 of gcc... Compile also with clang. Synthesize all results in a CSV containing the configuration (flag, with or without -mfpmath=387, compiler) and the different outputs. Please use pandas for writing the CSV... also the fieldnames should be compiler, flags, type, and result. The C file to compile is assoc.c

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.

```
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:
        subprocess.run(cmd, shell=True, check=True)
        binary = cmd.split()[-1]
        output = subprocess.check_output(f'./{binary}',
            shell=True).decode().strip()
        results.append(output)
    return results

def main():
    configurations = [
        ('gcc', ''), ('gcc', '-mfpmath=387'),
        ('gcc', '-ffast-math'),
        ('gcc', '-fno-signed-zeros'),
        ('gcc', '-fno-rounding-math'),
        ('clang', ''), ('clang', '-ffast-math'),
        ('clang', '-fno-signed-zeros'),
        ('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]},
            {'compiler': compiler, 'flags': flags, 'type'
                : 'double', 'result': results[1]},
            {'compiler': compiler, 'flags': flags, 'type'
                : 'float', 'result': results[2]},
        ])
    df = pd.DataFrame(data)
    df.to_csv('results.csv', index=False)
if __name__ == '__main__':
    main()
```

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 well be 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.

> 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.

> (LLM) [...] you can use the 'cfg' attribute to enable or disable the features at compile-time [...] in your 'Cargo.toml' file [...].

```
fn main() {
    #[cfg(feature = "hello")]
```

```
    print!("Hello ");
    #[cfg(feature = "beautiful")]
    print!("Beautiful ");
    #[cfg(feature = "wonderful")]
    print!("Wonderful ");
    #[cfg(feature = "world")]
    print!("World");
    println!();
}
```

Listing 4: Generated Rust code (conditional compilation)

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)**

(LLM) 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:...

```
  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**

```
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)

We have implemented the "Hello world" through *command-line parameters* in different programming languages (Julia, Python, etc.). A few instructions' changes at the prompt level allow developers to implement the same variability (1) in different programming languages (C++, Rust, Java, JavaScript, Python) (2) while choosing the suited variability mechanisms (conditional compilation, templating, feature toggles, command-line parameters). Overall, LLMs bring flexibility and automation: "features as prompts" can be seamlessly realized in different technological spaces.

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

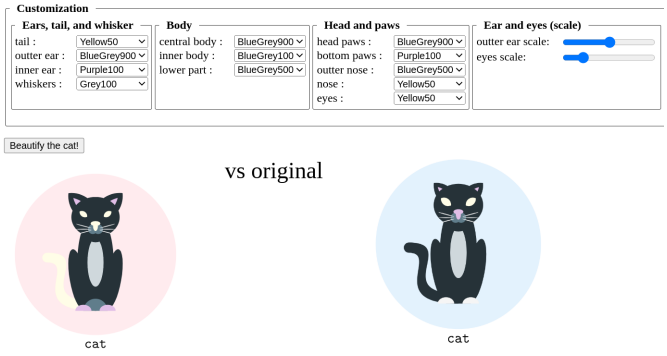

Figure 1: End-user customization of Cat TikZ code

final result is Figure 1 where users can play with different features and customizing the cat. We now detail how LLMs assisted us in implementing such a system.

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

```
  cat/.pic={
  \tikzset{x=3cm/5,y=3cm/5,shift={(0,-1/3)}}
  \useasboundingbox (-1,-1) (1,2);
  \fill [BlueGrey900] (0,-2)
    .. controls ++(180:3) and ++(0:5/4) .. (-2,0)
    arc (270:90:1/5)
    .. controls ++(0:2) and ++(180:11/4) .. (0,-2+2/5);
  ...
  \fill [BlueGrey900] ellipse [x radius=3/4, y radius=2];
  \fill [BlueGrey100] ellipse [x radius=1/3, y radius=1];
  \fill [BlueGrey900]
    (0,15/8) ellipse [x radius=1, y radius=5/6]
  ...
}
```

Listing 7: Original TikZ code of cat

> (LLM) To change the color of the tail, you need to modify the fill color specified in the first 'fill' command, which draws the tail shape. In the provided code, the tail color is set to 'BlueGrey900'. You can replace this color with any other color you'd like. Here's an example with the tail color changed to 'red':

```
\tikzset{%
cat/.pic={
\tikzset{x=3cm/5,y=3cm/5,shift={(0,-1/3)}}
\useasboundingbox (-1,-1) (1,2);
\fill [red] (0,-2) % Change the color here
  .. controls ++(180:3) and ++(0:5/4) .. (-2,0)
  arc (270:90:1/5)
  .. controls ++(0:2) and ++(180:11/4) .. (0,-2+2/5);
% The rest of the code remains the same
```

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

```
cat/.pic={
  \tikzset{x=3cm/5,y=3cm/5,shift={(0,-1/3)}}
  \useasboundingbox (-1,-1) (1,2);
  \fill [<< tail_color >>] (0,-2)
    .. controls ++(180:3) and ++(0:5/4) .. (-2,0)
    arc (270:90:1/5)
    .. controls ++(0:2) and ++(180:11/4) .. (0,-2+2/5);
  ...
  \fill [<< central_body_color >>] ellipse [x radius
      =3/4, y radius=2];
  \fill [<< inner_body_color >>] ellipse [x radius=1/3,
       y radius=1];
  \fill [<< head_paws_color >>]
  ...
```

Listing 9: Template-based generator of TikZ

The Python generator can then be used to synthesize a gallery of diverse cats (e.g., with random colors and scales) [57]. The generator can also be controlled through a Web application (see Figure 1). The client side offers to users the control to configure different features

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 number 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 different pieces of code and (2) call to combine different languages and technologies for realizing variability. Our work can be seen as a first step towards a *benchmark* for variability-related tasks that will serve to evaluate current and next-generation of LLMs.

Improving the robustness of LLMs. As prompts are central to the effectiveness of LLMs, a key question is how to properly formulate instructions w.r.t. 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

[1] Mathieu Acher, Roberto Erick Lopez-Herrejon, and Rick Rabiser. 2017. Teaching Software Product Lines: A Snapshot of Current Practices and Challenges. *ACM Transactions on Computing Education (TOCE)* (2017), 31.

[2] Vander Alves, Christa Schwanninger, Luciano Barbosa, Awais Rashid, Peter Sawyer, Paul Rayson, Christoph Pohl, and Andreas Rummler. 2008. An Exploratory Study of Information Retrieval Techniques in Domain Analysis. In *SPLC'08*. IEEE, 67–76.

[3] Juliana Alves Pereira, Hugo Martin, Mathieu Acher, Jean-Marc Jézéquel, Goetz Botterweck, and Anthony Ventresque. 2021. Learning Software Configuration Spaces: A Systematic Literature Review. *Journal of Systems and Software (JSS)* (June 2021). https://doi.org/10.1145/nnnnnnn.nnnnnnn

[4] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer-Verlag.

[5] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. *Program Synthesis with Large Language Models*. Technical Report arXiv:2108.07732. arXiv. https://doi.org/10.48550/arXiv.2108.07732 arXiv:2108.07732 [cs] type: article.

[6] Ebrahim Bagheri, Faezeh Ensan, and Dragan Gasevic. 2012. Decision support for the software product line domain engineering lifecycle. *Automated Software Engineering* 19, 3 (2012), 335–377.

[7] Sylvie Boldo and Thi Minh Tuyen Nguyen. 2011. Proofs of numerical programs when the compiler optimizes. *Innovations in Systems and Software Engineering* 7 (2011), 151–160. https://inria.hal.science/hal-00777639

[8] Frederik P Brooks and No Silver Bullet. 1987. Essence and accidents of software engineering. *IEEE computer* 20, 4 (1987), 10–19.

[9] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[10] Javier Cámara, Javier Troya, Lola Burgueño, and Antonio Vallecillo. 2023. On the Assessment of Generative AI in Modeling Tasks: An Experience Report with ChatGPT and UML. *Softw. Syst. Model.* 22, 3 (may 2023), 781–793. https://doi.org/10.1007/s10270-023-01105-5

[11] Jessie Carbonnel. 2018. *L'analyse formelle de concepts: un cadre structurel pour l'étude de la variabilité de familles de logiciels*. Ph. D. Dissertation. Université Montpellier.

[12] Jessie Carbonnel, Marianne Huchard, André Miralles, and Clémentine Nebut. 2017. Feature model composition assisted by formal concept analysis. In *ENASE: Evaluation of Novel Approaches to Software Engineering*. SciTePress, 27–37.

[13] Kun Chen, Wei Zhang, Haiyan Zhao, and Hong Mei. 2005. An approach to constructing feature models based on requirements clustering. In *RE'05*. 31–40. https://doi.org/10.1109/RE.2005.9

[14] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. *Evaluating Large Language Models Trained on Code*. Technical Report arXiv:2107.03374. arXiv. arXiv:2107.03374 [cs] type: article.

[15] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2022. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311* (2022).

[16] Krzysztof Czarnecki and Ulrich Eisenecker. 2000. *Generative Programming: Methods, Tools, and Applications*.

[17] Jean-Marc Davril, Edouard Delfosse, Negar Hariri, Mathieu Acher, Jane Cleland-Huang, and Patrick Heymans. 2013. Feature Model Extraction from Large Collections of Informal Product Descriptions. In *ESEC/FSE*.

[18] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[19] Natalia Díaz-Rodríguez, Javier Del Ser, Mark Coeckelbergh, Marcos López de Prado, Enrique Herrera-Viedma, and Francisco Herrera. 2023. Connecting the dots in trustworthy Artificial Intelligence: From AI principles, ethics, and key requirements to responsible AI systems and regulation. *Information Fusion* (2023).

[20] Jean-Baptiste Döderlein, Mathieu Acher, Djamel Eddine Khelladi, and Benoît Combemale. 2022. Piloting Copilot and Codex: Hot Temperature, Cold Prompts, or Black Magic? *CoRR* abs/2210.14699 (2022). https://doi.org/10.48550/arXiv.

2210.14699 arXiv:2210.14699
[21] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. https://doi.org/10.48550/arXiv.2002.08155 arXiv:2002.08155 [cs]
[22] Alessio Ferrari, Giorgio Oronzo Spagnolo, and Felice dell'Orletta. 2013. Mining commonalities and variabilities from natural language documents. In *SPLC*.
[23] Mor Geva, Ankit Gupta, and Jonathan Berant. 2020. Injecting Numerical Reasoning Skills into Language Models. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Online, 946–958. https://doi.org/10.18653/v1/2020.acl-main.89
[24] Github. 2021. *GitHub Copilot · Your AI pair programmer*. https://copilot.github.com
[25] Jianmei Guo, Krzysztof Czarnecki, Sven Apel, Norbert Siegmund, and Andrzej Wasowski. 2013. Variability-aware performance prediction: A statistical learning approach. In *ASE*.
[26] Axel Halin, Alexandre Nuttinck, Mathieu Acher, Xavier Devroey, Gilles Perrouin, and Benoit Baudry. 2019. Test them all, is it worth it? Assessing configuration sampling on the JHipster Web development stack. *Empirical Software Engineering (ESE)* 24, 2 (July 2019), 674–717. https://doi.org/10.07980
[27] Negar Hariri, Carlos Castro-Herrera, Mehdi Mirakhorli, Jane Cleland-Huang, and Bamshad Mobasher. 2013. Supporting Domain Analysis through Mining and Recommending Features from Online Product Listings. *IEEE Transactions on Software Engineering* 99, PrePrints (2013), 1. https://doi.org/10.1109/TSE.2013.39
[28] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring Coding Challenge Competence With APPS. arXiv:2105.09938 [cs]
[29] Nili Itzik and Iris Reinhartz-Berger. 2014. SOVA - A Tool for Semantic and Ontological Variability Analysis. In *Joint Proceedings of the CAiSE 2014 Forum and CAiSE 2014 Doctoral Consortium*. 177–184.
[30] Pooyan Jamshidi, Norbert Siegmund, Miguel Velez, Akshay Patel, and Yuvraj Agarwal. 2017. Transfer Learning for Performance Modeling of Configurable Systems: An Exploratory Analysis. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Press, 497–508.
[31] Jean-Marc Jézéquel, Jörg Kienzle, and Mathieu Acher. 2022. From feature models to feature toggles in practice. In *SPLC 2022 - 26th ACM International Systems and Software Product Line Conference*. ACM, Graz / Hybrid, Austria, 234–244.
[32] Dongpu Jin, Xiao Qu, Myra B. Cohen, and Brian Robinson. 2014. Configurations everywhere: implications for testing and debugging in practice. In *Companion Proceedings of the 36th International Conference on Software Engineering - ICSE Companion 2014*. ACM, 215–224.
[33] Christian Kaltenecker, Alexander Grebhahn, Norbert Siegmund, Jianmei Guo, and Sven Apel. 2019. Distance-Based Sampling of Software Configuration Spaces. In *Proceedings of the International Conference on Software Engineering (ICSE)*.
[34] Thomas Krismayer, Rick Rabiser, and Paul Grünbacher. 2017. Mining Constraints for Event-Based Monitoring in Systems of Systems. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Press, 826–831.
[35] Thomas Wolf Lewis Tunstall, Leandro von Werra. 2022. *Natural Language Processing with Transformers, Revised Edition [Book]*. https://www.oreilly.com/library/view/natural-language-processing/9781098136789/ ISBN: 9781098136796.
[36] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-Level Code Generation with AlphaCode. arXiv:2203.07814 [cs]
[37] Roberto E. Lopez-Herrejon, Jabier Martinez, Wesley Klewerton Guez Assunção, Tewfik Ziadi, Mathieu Acher, and Silvia Regina Vergilio (Eds.). 2023. *Handbook of Re-Engineering Software Intensive Systems into Software Product Lines*. Springer International Publishing. https://doi.org/10.1007/978-3-031-11686-5
[38] Hugo Martin, Mathieu Acher, Juliana Alves Pereira, Luc Lesoil, Jean-Marc Jézéquel, and Djamel Eddine Khelladi. 2022. Transfer Learning Across Variants and Versions: The Case of Linux Kernel Size. *IEEE Trans. Software Eng.* 48, 11 (2022), 4274–4290. https://doi.org/10.1109/TSE.2021.3116768
[39] Antonio Mastropaolo, Luca Pascarella, Emanuela Guglielmi, Matteo Ciniselli, Simone Scalabrino, Rocco Oliveto, and Gabriele Bavota. 2023. On the robustness of code generation techniques: An empirical study on github copilot. *arXiv preprint arXiv:2302.00438* (2023).
[40] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2017. *Mastering software variability with FeatureIDE*. Springer.
[41] Jens Meinicke, Chu-Pan Wong, Bogdan Vasilescu, and Christian Kästner. 2020. Exploring Differences and Commonalities between Feature Flags and Configuration Options. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice* (Seoul, South Korea) *(ICSE-SEIP '20)*. Association for Computing Machinery, New York, NY, USA, 233–242. https://doi.org/10.1145/3377813.3381366
[42] Alexandr Murashkin, Michal Antkiewicz, Derek Rayside, and Krzysztof Czarnecki. 2013. Visualization and exploration of optimal variants in product line engineering. In *17th International Software Product Line Conference, SPLC 2013, Tokyo, Japan - August 26 - 30, 2013*, Tomoji Kishi, Stan Jarzabek, and Stefania Gnesi (Eds.). ACM, 111–115. https://doi.org/10.1145/2491627.2491647
[43] I Made Murwantara, Behzad Bordbar, and Leandro L. Minku. 2014. Measuring Energy Consumption for Web Service Product Configuration. In *Proceedings of the 16th International Conference on Information Integration and Web-based Applications & Services* (Hanoi, Viet Nam). ACM, 224–228.
[44] Vivek Nair, Zhe Yu, Tim Menzies, Norbert Siegmund, and Sven Apel. 2018. Finding Faster Configurations Using Flash. *IEEE Transact. on Software Engineering* (2018).
[45] Nan Niu and Steve M. Easterbrook. 2009. Concept analysis for product line requirements. In *AOSD'09*, Kevin J. Sullivan, Ana Moreira, Christa Schwanninger, and Jeff Gray (Eds.). ACM, 137–148.
[46] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]
[47] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag.
[48] Adam Porter, Cemal Yilmaz, Atif M Memon, Douglas C Schmidt, and Bala Natarajan. 2007. Skoll: A Process and Infrastructure for Distributed Continuous Quality Assurance. *IEEE Transactions on Software Engineering* 33, 8 (2007), 510–525.
[49] Ruchir Puri, David S. Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss. 2021. CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks. https://doi.org/10.48550/arXiv.2105.12655 arXiv:2105.12655 [cs]
[50] Iris Reinhartz-Berger. 2014. Can domain modeling be automated? Levels of automation in domain modeling. In *SPLC '14*. 359.
[51] Iris Reinhartz-Berger, Arnon Sturm, and Yair Wand. 2013. Comparing functionality of software systems: An ontological approach. *Data Knowl. Eng.* 87 (2013), 320–338. https://doi.org/10.1016/j.datak.2012.09.005
[52] Aarohi Srivastava, Abhinav Rastogi, Abhishek Rao, Abu Awal Md Shoeb, Abubakar Abid, Adam Fisch, Adam R Brown, Adam Santoro, Aditya Gupta, Adrià Garriga-Alonso, et al. 2022. Beyond the Imitation Game: Quantifying and extrapolating the capabilities of language models. *arXiv preprint arXiv:2206.04615* (2022).
[53] Shaghayegh Tavassoli, Carlos Diego Nascimento Damasceno, Ramtin Khosravi, and Mohammad Reza Mousavi. 2022. Adaptive behavioral model learning for software product lines. In *SPLC '22: 26th ACM International Systems and Software Product Line Conference, Graz, Austria, September 12 - 16, 2022, Volume A*, Alexander Felfernig, Lidia Fuentes, Jane Cleland-Huang, Wesley K. G. Assunção, Andreas A. Falkner, Maider Azanza, Miguel Á. Rodríguez Luaces, Megha Bhushan, Laura Semini, Xavier Devroey, Cláudia Maria Lima Werner, Christoph Seidl, Viet-Man Le, and José Miguel Horcas (Eds.). ACM, 142–153.
[54] Paul Temple, Mathieu Acher, Jean-Marc Jézéquel, and Olivier Barais. 2017. Learning-Contextual Variability Models. *IEEE Software* 34, 6 (Nov. 2017), 64–70. https://hal.inria.fr/hal-01659137
[55] Xhevahire Tërnava, Luc Lesoil, Georges Aaron Randrianaina, Djamel Eddine Khelladi, and Mathieu Acher. 2022. On the Interaction of Feature Toggles. In *VaMoS 2022 - 16th International Working Conference on Variability Modelling of Software-Intensive Systems*. Florence, Italy.
[56] Stackexchange thread about TikZ. 2023. https://tex.stackexchange.com/questions/387047/the-duck-pond-showcase-of-tikz-drawn-animals-ducks.
[57] https://github.com/acherm/progvary-withgpt/. 2023. Companion web page (prompts, sessions, generated codes).
[58] https://www.anthropic.com/index/introducing-claude. 2023. Introducing Claude.
[59] Pavel Valov, Jean-Christophe Petkovich, Jianmei Guo, Sebastian Fischmeister, and Krzysztof Czarnecki. 2017. Transferring Performance Prediction Models Across Different Hardware Platforms. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. ACM, 39–50.
[60] Dennis Westermann, Jens Happe, Rouven Krebs, and Roozbeh Farahbod. 2012. Automated Inference of Goal-Oriented Performance Prediction Functions. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM, 190–199.
[61] Cemal Yilmaz, Myra B Cohen, and Adam A Porter. 2006. Covering Arrays for Efficient Fault Characterization in Complex Configuration Spaces. *IEEE Transactions on Software Engineering* 32, 1 (2006), 20–34.
[62] Yi Zhang, Jianmei Guo, Eric Blais, Krzysztof Czarnecki, and Huiqun Yu. 2016. A Mathematical Model of Performance-Relevant Feature Interactions. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 25–34.
[63] Wei Zheng, Ricardo Bianchini, and Thu D Nguyen. 2007. Automatic Configuration of Internet Services. *ACM SIGOPS Operating Systems Review* 41, 3 (2007), 219–229.