Towards LLM-based System Migration in Language-Driven Engineering

Daniel Busch, Alexander Bainczyk, and Bernhard Steffen

TU Dortmund University, Department of Computer Science, Chair for Programming Systems, 44227 Dortmund, Germany
daniel2.busch@tu-dortmund.de, alexander.bainczyk@tu-dortmund.de, bernhard.steffen@tu-dortmund.de

Abstract. In this paper we show how our approach of extending Language Driven Engineering (LDE) with natural language-based code generation supports system migration: The characteristic decomposition of LDE into tasks that are solved with dedicated domain-specific languages divides the migration tasks into portions adequate to apply LLM-based code generation. We illustrate this effect by migrating a low-code/no-code generator for point-and-click adventures from JavaScript to TypeScript in a way that maintains an important property: generated web applications can automatically be validated via automata learning and model analysis by design. In particular, this allows to easily test the correctness of migration by learning the difference automaton for the generated products of the source and the target system of the migration.

Keywords: Software Engineering, Low-Code/No-Code, Language-driven Engineering, Large Language Models, Migration, Transformation, Automata Learning, Verification, Web Application

1 Motivation and Introduction

Many Large Language Models (LLMs) can be used for coding tasks [12]. They are used as programming assistants [16], reviewers [10], or even full-blown code generation tools [11]. In particular for small problems this works very well, but the quality and reliability of the code drastically degrades with growing context size and structural or conceptual complexity of the software projects. In [14], we have illustrated how this problem of scalability can be mitigated within a heterogeneous approach that comprises model-based and LLM-based code generation: We extended our Language-Driven Engineering (LDE) environment [7] for low-code/no-code development via dedicated Domain-Specific Languages (DSLs) to also support specification in natural language with the following benefits:

- The tasks to be solved via LLM-based code generation can be tailored in size and conceptual complexity and
- the overall heterogeneously constructed system can be directly validated at system level using automata learning and model analysis.

Our approach has been illustrated via a system to generate fully running, web-based point-and-click adventures from two specifications, (1) an easy graphical specification for the 'landscape' of the adventure, and (2) a natural language specification for the game logic. The corresponding system structure is depicted in the upper half of Figure 1. In [14], we concluded that this approach is unique by placing this concept in the context of existing research.

In this paper, we illustrate an additional benefit of our approach to LDE-based natural language integration: The heterogeneous LDE-based structure also supports the *automatic migration* of entire heterogeneous LDE-based systems. In particular, we sketch a migration process that allows one to migrate the entire system generators we constructed for point-and-click adventures to different programming languages just using a simple user prompt specifying the target language. For illustration, we migrated the system generator from JavaScript to TypeScript which essentially requires the migrator to automatically insert the missing type information into the system code.

Like in [14], our approach applies natural language based code generation only for very dedicated, small-scale tasks. Other tasks can be solved using textual or graphical DSLs. For each task we apply the paradigm that is most suitable to solve it. In our example these tasks are:

1. Generating base code that implements the 'landscape' of the adventure using graphical models.
2. Generating a prompt frame that provides contextual information using the same graphical models.
3. Introducing game logic into the generated base code (from task one) utilizing the generated prompt frame (from task two) using an LLM.

This separation of tasks does not only addresses the scalability problem, but it also allowed us to maintain the second benefit mentioned above: The migrated system generator automatically supports validation at system level via automata learning and model analysis. In particular, this allows us to easily test the correctness of migration simply by learning the difference automaton for the generated products of the source and the target system of the migration.

Figure 1 explains the reason for this benefits: Migrating the original JavaScript system generator sketched in the top to the TypeScript generator in the bottom only requires very local adaptations. In our example, this means that we only have to provide a descriptive natural language prompt for the the portion marked in red in Figure 1.

This paper is organized as follows: In Section 2, we outline our previous work [14] and introduce fundamentals of learning-based evolution control. Section 3 covers our concept of LLM-based code generator migration. Next, Section 4 demonstrates this concept with a running example, and following that, Section 5 concludes this paper with a discussion and an outlook on future work.

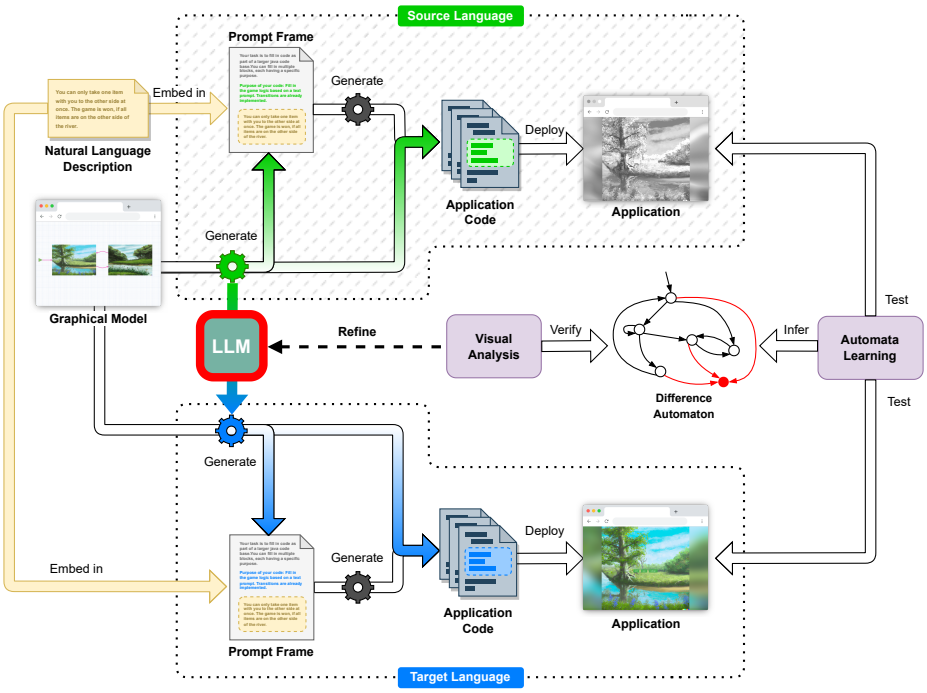

Fig. 1. LLM-based code transformation concept.

2 Preliminaries

In this section, we recapitulate the core ideas of our work from our previous paper [14] and outline the necessary basics of learning-based testing in the context of our approach.

2.1 LLM-based Code Generators and Language Decomposition

In [14] we presented a way to combine LDE with the generative power of LLMs. This approach enables users to use both graphical modeling and natural language descriptions, each applicable to whichever part of the domain is more suitable. This also reduces overall complexity by splitting the domain into smaller, more manageable aspects. We further demonstrated the approach on a web-based point-and-click adventure: A sitemap of game screens is modeled graphically, while the game logic is described in natural language.

Our approach generates code from graphical models and an accompanying Prompt Frame that provides context to the LLM. The Prompt Frame contains the expected target language, variables and functions that are available from the code generated from the graphical model, as well as code stubs that provide

the function signatures for which the LLM needs to generate code to obtain the resulting product. The combination of the code generated from the graphical model, which we refer to as the base code, and the code generated by the LLM then results in the final product.

This divide-and-conquer approach aims to solve some problems of LLM-based code generation, such as too large contexts or loose task descriptions. Moreover, it allows us to benefit from the formal aspects of the LDE paradigms and informal natural languages and LLMs.

Further, our approach employs code instrumentation for the resulting product code which allows the automated inference of behavioral models for verification purposes, see Section 2.2. The instrumentation is part of the manually implemented code generator that generates code from the graphical model.

2.2 Learning-Based Evolution Control

Active Automata Learning [1] has proven to be a viable solution for automated black-box testing of web applications in the past [2, 4, 9]. *Active learning* refers to the process in which a *learner* poses test queries over an input alphabet to a System Under Learning (SUL) in an automated fashion to infer a formal automaton model representing the SUL's behavior. Because web applications can be characterized as reactive systems, previous research relied on Mealy machines to capture their behavior.

To minimize manual effort, [13] introduces the iHTML DSL to instrument an application's HTML code in a way that enables the on-the-fly inference of system inputs by interacting with and analyzing the website's Document Object Model (DOM) automatically. We already exploited iHTML in [14] for our LLM-based code generation approach to generate instrumented, web-based point-and-click adventures that can be learned by simply providing their URLs.

Previous research [3, 4] established that a stable alphabet abstraction is required to enable structural comparisons between models to detect behavioral changes. In this context, [8] introduces the notion of *difference automata*, i.e. Mealy machines inferred by testing two systems simultaneously. The resulting automaton will then show all traces that lead to the occurrence of divergent behavior, see e.g. Figure 2. In this paper, we learn difference automata to detect and visualize behavioral differences between two software versions that are the result of LLM-based code migration.

3 Concept

Our goal is to use LLMs for code generator migration tasks. While in some cases this could be done for any code generator, we want to apply additional principles to be able to more easily handle the outcome of the code generated by the LLMs. The principles are based on the approach presented in [14] and are as follows:

1. Split the problem domain to minimize individual generation contexts and make code generation for LLMs easier to solve.

2. Instrument the generated code so that products can be verified which provides additional trust in the LLM-generated code.
3. Validate the product using automata learning and provide feedback to the user. Mismatches introduced by the migration can be detected using difference automata.

Splitting We split the problem domain according to our approach of [14], as described in Section 2.1. The existing system generator (see the colored cogs in Figure 1) consists of two sub-generators, one for the application code and another for the Prompt Frame (see Source Language in Figure 1). Each sub-generator is migrated separately by being passed to an LLM, together with a supporting description to instruct the LLM with the migration task.

Instrumentation Only the LLM-based migration of the application code generator may lead to violations of the iHTML syntax. However, such violations are automatically detected by the iHTML syntax checker and can be corrected manually by refining the prompt for the LLM-based migration.

Validation Syntactically correct instrumented code can automatically be validated via difference automata provide via automata learning (see Figure 2): Whenever there is a path ending in a behavioural discrepancy (see area marked in red) we can conclude that the LLM-based migration is erroneous. This information is then passed to a human expert for updating the prompt for the LLM-based migration in a similar fashion as before for eliminating iHTML syntax violations.

Figure 1 summarizes our setup. The upper half of the figure shows the approach as presented in [14]. In the middle, it is visualized that the generator used to generate the application code and the Prompt Frame is fed into an LLM (e.g. ChatGPT) to instruct it to migrate the sub-generators separately into the desired target language. The bottom half of the figure shows the same workflow as the top half, but using the migrated generator instead. Having two application instances, automata learning is used to create the difference automaton and feedback is passed to the user who refines the LLM-based migrator.

4 Example

To evaluate our concept described in Section 3, we have applied it to the example of the river crossing puzzle [14]. In this example, we developed a web-based point-and-click adventure using the Webstory DSL [5] of the graphical modeling suite Cinco [6]. Webstory has been modified so that graphical modeling is only used to model the available game screens and their reachability in a sitemap-like manner. From these graphical models a point-and-click adventure base code as well as a Prompt Frame with contextual information is generated (see Figure 1). All game logic, such as win/loss conditions, is modeled using natural language descriptions that are embedded in the generated Prompt Frame.

Migration of the source generators that generate the base code and the Prompt Frame was done using ChatGPT in its GPT-4 version [15]. The source generators use JavaScript in the case of the base code and the code stubs in the Prompt Frame, or natural language referencing JavaScript and JavaScript objects in the case of the natural language part of the Prompt Frame. In this example, our goal is to migrate these generators to TypeScript, a typed scripting language.

Migration Listing 1.1 shows the initial prompt that prepares ChatGPT to migrate the Prompt Frame generator. An excerpt of the target Prompt Frame generated by ChatGPT can be seen in Listing 1.2. Note that ChatGPT successfully migrated both the natural language contextualization and the code stubs to be implemented. All necessary functions were present and properly typed after the migration.

The base code generator was migrated using a separate conversation and prompts. Listing 1.3 is an excerpt of the target base code generator. Two things are noteworthy about this successful migration. First, the overall migration and typing was done correctly and quite extensively. Second, the instrumentation that is introduced with this base code is preserved. This second aspect is critical to the validation of the migration proposed in this paper.

```
You are provided with prompt frames. The prompt frame is
    wrapped into "BEGIN PROMPT FRAME" and "END PROMPT
    FRAME". The prompt frame includes ALL text AND code.
These prompt frames should be used for yourself to
    provide you with information to get a desired code
    output for an input scenario.

Your overall task will be to modify the given prompt
    frame so that you output a modified prompt frame for
    another programming language instead of the given
    prompt frame.

Answer only as follows in two interactions:
1. First, output only the programming language for which
    the given prompt frame seems to be made, and ask the
    user which programming language you should migrate the
    prompt frame to.
2. After receiving the user's answer, display only the
    migrated prompt frame and no additional text.
```

Listing 1.1. Priming prompt for Prompt Frame migration.

```
BEGIN PROMPT FRAME
Your task is to fill in code as part of a larger
    TypeScript code base.
[...]
The code blocks for you to implement:

function initVariables(): void {[...]}
function checkWin(): void {[...]}
function checkLoss(): void {[...]}
```

Listing 1.2. Excerpt of migrated Prompt Frame generator.

```
interface GameObject {
    name: string;
    currentScreen: string;
    transitions: Array<{ screen: string, function: () =>
        void }>;
}

function init(): void {
    this.currentState = states.first;
    this.states = states;
    this.gameObjects = [] as GameObject[];
    [...]
}
[...]
function addCustomClickAreas(): void {
    [...]
    items.forEach((item: GameObject) => {
    const itemElement: HTMLButtonElement =
        document.createElement('button');
    itemElement.classList.add('flex-item',
        'interaction-item');
    itemElement.setAttribute('data-lbd-action', 'Click');
    itemElement.setAttribute('data-lbd-name', item.name +
        '-' + this.currentState);
    itemElement.innerText = item.name;
    [...]
}
[...]
```

Listing 1.3. Excerpt of migrated base code generator.

Verifying the Migration For illustrative purposes, we demonstrate how automata learning can be used to detect behavioral differences between two system iterations. The means for this are *difference automata* [8] (see Figure 2), which

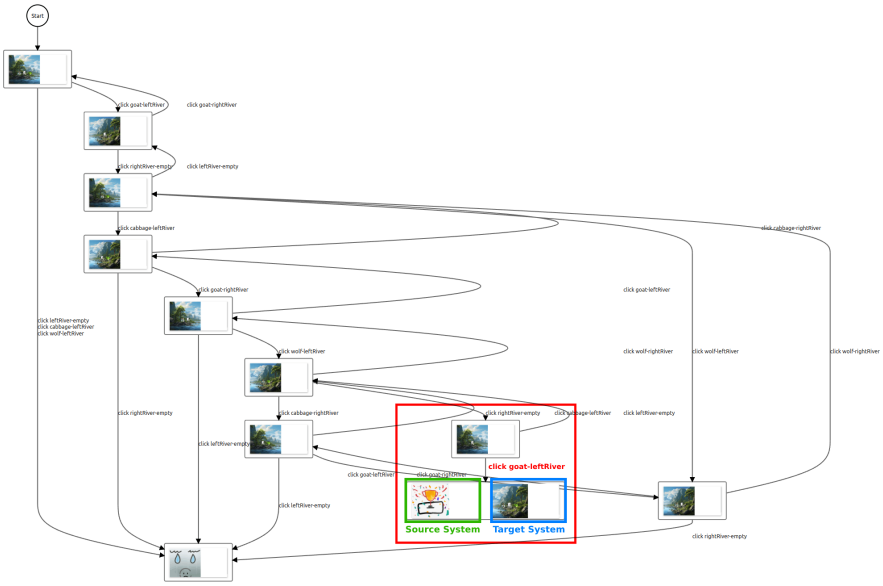

Fig. 2. Difference automaton for two learned WebStories.

contain all observed traces that lead to a different input-output behavior of the two systems in question. By default, difference automata are constructed as Mealy machines, but in this paper we convert them to Moore machines to reflect user-level interactions more accurately [14].

To demonstrate the benefits of our migration approach, we manually introduced a bug into the generated code to simulate a possible flaw in the LLM when translating the user specifications into TypeScript code. The changes affect the part of the code responsible for checking the game's win condition. More specifically, it affects a function that returns `true` when the win condition is met, i.e. when all items are on the right side of the river. For our example, however, we have modified the function to return `false` in this case, resulting in the game never reaching the winning screen.

We first learned the automaton of the original JavaScript application, transformed it to TypeScript using our presented approach, then manually introduced the bug, and finally learned the automaton of the now erroneous application to infer the difference automaton seen in Figure 2. The behavioral difference is highlighted in red: If the farmer is on the left side of the river with the goat, while the cabbage and wolf are on the right side of the river, the game would have been won as soon as the user clicked on the goat, resulting in the display of the winning screen in the source system. However, in our modified target system, the game enters a state where instead of the winning screen, all three items are displayed on the right side of the river, and therefore the game is never actually won. This information is graphically displayed and can be used to fix the bug.

5 Conclusion

In this paper we have shown how our approach of extending Language-Driven Engineering (LDE) with natural language-based code generation presented in [14] supports system migration. Central to this extension is the LDE-characteristic decomposition into tasks that are solved with dedicated domain-specific languages, be they textual, graphical, or natural. This decomposition allows the division of the migration tasks into portions adequate to apply LLM-based code generation. We have illustrated the impact of our approach by migrating a low-code/no-code generator for web-based point-and-click adventures from JavaScript to TypeScript, showing that

- the LLM-based migration correctly introduces the types required for TypeScript and that
- also the point-and-click adventures generated with the migrated system can be validated via automata learning and model analysis by design. In particular, this allows to easily test the correctness of migration by learning the difference automaton for the generated products of the source and the target system of the migration.

Technically, we have used LLMs to automatically migrate all code generators involved in our presented example, those that follow classical model-driven approaches as well as those that were based on natural language descriptions. Currently, we are experimenting with more complex scenarios.

We are convinced that hybrid approaches as the one presented here are a good way to mitigate the weaknesses of LLM-based code generation: They provide means for decomposition-based scalability, and to safely position LLM-based code in an overall application.

References

[1] Dana Angluin. "Learning Regular Sets from Queries and Counterexamples". In: 75.2 (1987), pp. 87–106.

[2] Harald Raffelt et al. "Dynamic testing via automata learning". In: *International Journal on Software Tools for Technology Transfer (STTT)* 11.4 (2009), pp. 307–324. ISSN: 1433-2779. DOI: http://dx.doi.org/10.1007/s10009-009-0120-7.

[3] Stephan Windmüller et al. "Active Continuous Quality Control". In: *Proceedings of the 16th International ACM Sigsoft Symposium on Component-Based Software Engineering*. CBSE '13. Vancouver, British Columbia, Canada: Association for Computing Machinery, 2013, pp. 111–120. ISBN: 9781450321228. DOI: 10.1145/2465449.2465469. URL: https://doi.org/10.1145/2465449.2465469.

[4] Johannes Neubauer, Stephan Windmüller, and Bernhard Steffen. "Risk-Based Testing via Active Continuous Quality Control". In: *International Journal on Software Tools for Technology Transfer* 16.5 (2014), pp. 569–591. DOI: 10.1007/s10009-014-0321-6.

[5] Michael Lybecait et al. “A tutorial introduction to graphical modeling and metamodeling with CINCO”. In: *Leveraging Applications of Formal Methods, Verification and Validation. Modeling: 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part I 8*. Springer. 2018, pp. 519–538.

[6] Stefan Naujokat et al. “CINCO: a simplicity-driven approach to full generation of domain-specific graphical modeling tools”. In: *International Journal on Software Tools for Technology Transfer* 20 (2018), pp. 327–354.

[7] Bernhard Steffen et al. “Language-driven engineering: from general-purpose to purpose-specific languages”. In: *Computing and Software Science: State of the Art and Perspectives* (2019), pp. 311–344.

[8] Alexander Bainczyk, Bernhard Steffen, and Falk Howar. “Lifelong Learning of Reactive Systems in Practice”. In: *The Logic of Software. A Tasting Menu of Formal Methods: Essays Dedicated to Reiner Hähnle on the Occasion of His 60th Birthday*. Ed. by Wolfgang Ahrendt et al. Cham: Springer International Publishing, 2022, pp. 38–53. ISBN: 978-3-031-08166-8. DOI: 10.1007/978-3-031-08166-8_3. URL: https://doi.org/10.1007/978-3-031-08166-8_3.

[9] Alexander Bainczyk et al. “Towards Continuous Quality Control in the Context of Language-Driven Engineering”. In: *Leveraging Applications of Formal Methods, Verification and Validation. Software Engineering*. Ed. by Tiziana Margaria and Bernhard Steffen. Cham: Springer Nature Switzerland, 2022, pp. 389–406. ISBN: 978-3-031-19756-7.

[10] Zhiyu Li et al. “Automating code review activities by large-scale pre-training”. In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2022, pp. 1035–1047.

[11] Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. “Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models”. In: *Chi conference on human factors in computing systems extended abstracts*. 2022, pp. 1–7.

[12] Frank F Xu et al. “A systematic evaluation of large language models of code”. In: *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. 2022, pp. 1–10.

[13] Alexander Bainczyk. “Simplicity-Oriented Lifelong Learning of Web Applications”. [work in progress]. PhD thesis. Dortmund, Germany: TU Dortmund University, 2023.

[14] Daniel Busch et al. “ChatGPT in the Loop - A Natural Language Extension for Domain-Specific Modeling Languages”. In: *Lecture Notes of Computer Science*. Vol. 14380. Springer, 2023.

[15] OpenAI. “GPT-4 Technical Report”. In: *ArXiv* abs/2303.08774 (2023).

[16] Haoye Tian et al. “Is ChatGPT the Ultimate Programming Assistant–How far is it?” In: *arXiv preprint arXiv:2304.11938* (2023).