Hybrid Code Generation: Synergies of Large Language Models and Language-driven Engineering

Dissertation

zur Erlangung des Grades eines

D o k t o r s d e r I n g e n i e u r w i s s e n s c h a f t e n

der Technischen Universität Dortmund
an der Fakultät für Informatik

von

Daniel Busch

Dortmund

2026

Tag der mündlichen Prüfung:
6. März 2026

Dekan:
Prof. Dr. Jens Teubner

Gutachter:
Prof. Dr. Bernhard Steffen
Prof. Dr. Edward A. Lee

Acknowledgements

Throughout my scientific journey, I have been fortunate enough to meet many amazing people. They have inspired me, supported me, and shaped my perspective on many things. I am very grateful to have met them and would like to take this opportunity to thank them.

First and foremost, I would like to thank my supervisor, *Bernhard Steffen*. The teams you have led were always exceptional, and I am very glad to have been part of them. Thank you for always listening to my new ideas, my problems, and everything in between. Without your guidance, none of this would have been possible. You have always encouraged me to pursue great goals and provided direction when needed.

I would also like to thank my co-supervisor, *Edward A. Lee*. Thank you for discussing my ideas and inspiring me with your presentations. Thank you, *Jakob Rehof* and *Falk Howar*, for completing my committee. I have attended many of your lectures and am now grateful to be able to present and discuss my very own work with you. The discussions with my dissertation committee were helpful in shaping ideas for my next steps.

I was also lucky to work with an exceptional research team. My colleagues were some of the best I could have asked for. *Alexander Bainczyk, Steve Boßelmann, Johannes Düsing, Markus Frohme, Anemone Kampkötter, Marco Krumrey, Michael Lybecait, Sami Mitwalli, Alnis Murtovi, Gerrit Nolte, Maximilian Schlüter, Jonas Schürmann, Steven Smyth, Tim Tegeler, Sebastian Teumert, and Dominic Wirkner*, thank you all so much! Working with you on every project we shared has always been a great experience. Without you, my time at the university would definitely have been a lot less fun.

I would like to express my deepest gratitude to my wife, *Raza Hadzic*, for always supporting me. You are always there for me, no matter the time or place, and that is invaluable.

Picking names for acknowledgments is always difficult. If we have worked together and your name is not listed here, please know that I am thankful to have met you all the same. It's almost impossible to name all the amazing people who have played a role in my scientific journey thus far.

Abstract

Language-Driven Engineering (LDE) aims to provide the most suitable modeling language for every purpose and stakeholder. As a concept for Low-Code/No-Code (LC/NC) environments, LDE aims to create an easy-to-use development environment for everyone. Besides textual modeling languages, graphical modeling languages are particularly well suited for this purpose. They are often easy for humans to understand and easy to learn. Large Language Models (LLMs) are similarly easy to use. Natural language provides a universal interface that humans are accustomed to using every day. Consequently, LLMs have become ubiquitous in recent times. They are used in various fields, including text, image, and video generation. They are also highly popular for code generation. LLMs offer a high degree of flexibility, not only because of their natural language input, but also because of their output. They can produce arbitrary outputs, making them more versatile in code generation tasks than conventional code generators. However, they have downsides regarding controllability and reliability. For example, they are not deterministic and can produce different outputs when given identical inputs. Additionally, they are difficult to control reliably. Their output may not meet user expectations, and it may contain errors.

This uncertainty is particularly undesirable in environments like LDE, where code is generated from formalized models. Nevertheless, LDE environments may still want to benefit from LLMs' flexibility for code generation. Furthermore, natural languages allow for the simplicity of expression that Domain-Specific Languages (DSLs) seek to provide. LC/NC approaches aim to provide the same code generation experience as LLMs. These approaches enable even non-expert users to easily articulate their needs and generate code.

Therefore, combining LDE and LLMs would enable great synergies. The flexibility of LLM code generation could be incorporated into LDE. At the same time, LDE could provide mechanisms to incorporate the control of conventional code generation from formalized models into LLM code generation.

This dissertation presents a hybrid code generation approach that combines LLM-supported and conventional code generation in the context of LDE. Specifically, it combines a two-step generation approach and an extension to Template-based Code Generation (TBCG) for LLMs. This two-step process leverages the flexibility of LLMs within LDE, while maintaining control. To achieve this, it intertwines DSLs and natural language into Domain-Specific Natural Languages (DSNLs). This enables synergies between conventional and LLM code generation. The extension to TBCG makes it easily usable as an LLM tool. Instead of outputting code without guidance, it constrains LLM code generation output. This makes LLM code generation more controllable.

The hybrid code generation approach takes advantage of the flexibility of LLMs. At the same time, it establishes three layers of control over the use of LLMs: 1. Contextualization: The presented approach generates contextualization for the LLM, so users can interact with it using a DSNL instead of natural language alone. DSNLs define the domain in which LLMs operate and enable referencing of other models in the LDE environment. DSNLs shift the responsibility of good prompting from users to LDE developers. This

makes it easier to achieve good code generation results. Thus, this layer controls the input for LLMs. 2. Validation by Design: System-level validation is applied to observe whether the generated output meets user expectations. Exploiting control over code generation ensures that Active Automata Learning (AAL) can infer behavioral automata for validation purposes. This enables quality control of the output at the semantic level. 3. Output Constraints: Extending TBCG for easy use with LLMs guides them regarding their generation capabilities. Rather than allowing the LLMs to output arbitrary and potentially unwanted code, the TBCG tooling constrains their output options. These constraints control the LLMs during code generation.

Attached Publications

The publications covered by this dissertation are listed below. They were written and published in collaboration with other authors. A summary of each publication, including my contributions, is given in Section 1.2.

I Daniel Busch, Gerrit Nolte, Alexander Bainczyk, and Bernhard Steffen. **ChatGPT in the Loop: A Natural Language Extension for Domain-Specific Modeling Languages**. In: *International Conference on Bridging the Gap between AI and Reality*. Springer. 2023. DOI: `https://doi.org/10.1007/978-3-031-46002-9_24`

II Daniel Busch, Alexander Bainczyk, and Bernhard Steffen. **Towards LLM-based System Migration in Language-Driven Engineering**. In: *International Conference on Engineering of Computer-Based Systems*. Springer. 2023. DOI: `https://doi.org/10.1007/978-3-031-49252-5_14`

III Daniel Busch, Alexander Bainczyk, Steven Smyth, and Bernhard Steffen. **LLM-based code generation and system migration in language-driven engineering**. In: *International Journal on Software Tools for Technology Transfer* 1 (2025). DOI: `https://doi.org/10.1007/s10009-025-00798-x`

IV Daniel Busch. **Towards Code-centric Code Generators**. In: *Electronic Communications of the EASST* (2023). DOI: `https://doi.org/10.14279/tuj.eceasst.82.1218`

V Daniel Busch, Steven Smyth, Tim Tegeler, and Bernhard Steffen. **Code-centric Code Generation**. In: *International Symposium on Leveraging Applications of Formal Methods*. Springer. 2024. DOI: `https://doi.org/10.1007/978-3-031-73709-1_21`

VI Alexander Bainczyk, Daniel Busch, Marco Krumrey, Daniel Sami Mitwalli, Jonas Schürmann, Joel Tagoukeng Dongmo, and Bernhard Steffen. **Cinco Cloud: A Holistic Approach for Web-Based Language-Driven Engineering**. In: *International Symposium on Leveraging Applications of Formal Methods*. Springer. 2022. DOI: `https://doi.org/10.1007/978-3-031-19756-7_23`

Contents

1 Introduction **1**
- 1.1 My Contributions 4
- 1.2 Context of Attached Publications 6

2 Background **9**
- 2.1 Language-driven Engineering 9
- 2.2 Code Generation 10
- 2.3 Cinco 13
- 2.4 Learning-based Testing and Evolution Control 14

3 Controlled LLM Code Generation **15**
- 3.1 Language Decomposition 16
- 3.2 Contextualization 17
- 3.3 Validation & Feedback 18
- 3.4 Migration Capabilities & Change Control 18

4 Code-centric Code Generation **21**
- 4.1 AST Operation Annotation 23
- 4.2 Template Generation 24
- 4.3 Evolution 26
- 4.4 Further Development/Properties 27

5 Code-centric LLM-powered Code Generation **29**
- 5.1 CCG Template Instantiation 30
- 5.2 Combined Architecture 31
- 5.3 Example 33
- 5.4 Synergies 39

6 Related Work **43**
- 6.1 LLM-powered Code Generation 43
- 6.2 Combining LLM-powered Code Generation and Formal Methods 44
- 6.3 Template-based Code Generation 44

7 Conclusion **47**
- 7.1 Future Work 48

References **51**

Online References **59**

1

Introduction

One of Language-Driven Engineering (LDE)'s key principles is to divide tasks into smaller ones and provide the most suitable tool for each. To achieve this, LDE developers aim to create a modeling language tailored to the needs of each purpose and stakeholder. As a concept for Low-Code/No-Code (LC/NC) environments, LDE wants to create an easy-to-use development environment for everyone. Textual Domain-Specific Languages (DSLs) are a common approach to this. They provide a syntax that feels natural to domain experts, making them easy to use. Graphical modeling languages are also well-suited for this purpose. They are often intuitive and easy to learn. Additionally, they often come with drag-and-drop editors that guide users. Large Language Models (LLMs) are similarly easy to use. Following the breakthrough of transformers [84], and the release of GPT, LLMs have grown in popularity for a variety of use cases. They stand out due to their high degree of flexibility. Using natural language as input provides a universal interface that humans are accustomed to using every day. Moreover, they can produce arbitrary output and are not limited to solving very specific, small tasks. Thus, LLMs are used to generate text, images, videos, music, and more. LLMs are also highly popular in software engineering contexts. They are used as programming assistants [80], code reviewers [49], or for full code generation [83]. Despite their widespread use, there may always be trust issues with code generated by LLMs. LLMs tend to perform subpar, especially for complex tasks with large contexts or very special problems. Additionally, their output may not match user expectations, even for simpler tasks. One reason for this is a problem with their input format. While natural language provides a lot of flexibility and makes LLMs attractive to most users, it also has a downside. Natural language is often ambiguous, which makes it difficult to use LLMs precisely and control them effectively. Further problems arise from the flexibility of their output. LLMs are not deterministic, therefore, when the same input is provided twice, two different outputs may be generated. In general, their output is hard to control. Users often rely on natural language descriptions of output constraints or special requirements. They cannot strictly enforce these constraints and must trust the LLM to comply with them. This hard-to-control behavior is especially unwanted in formal scenarios, such as in LDE. When generating code from formal models, users expect stable outputs. However, integrating the flexibility of LLM-based code generation would provide huge benefits for LDE. Rather than being limited by the expressiveness of provided DSLs and code generators, LLMs could expand domain capabilities. Furthermore, natural languages allow for the simplicity of expression that DSLs seek to provide. Overall, LC/NC approaches aim to provide the same code generation experience as LLMs. These approaches enable even non-expert users to easily articulate their needs and generate code.

In the context of Model-Driven Engineering (MDE), there is a lot of ongoing research around integrating LLM usage. In addition to research that categorizes LLM usage in MDE [31] or provides visions about its future use [29], there is also much research about its concrete use in different MDE workflows. Some create models of existing languages from natural language [56]. Others create new DSLs from natural language [23]. LLMs are also used to create models from code for reverse engineering [69, 70]. And LLMs are used to replace conventional code generators, to generate code from models [27, 90]. However, LLMs still generate code that may contain errors [50, 72] that users might not expect, especially in formalized environments like LDE.

This dissertation presents an approach that combines conventional and LLM-based code generation techniques in the context of full code generation and LDE, to mitigate these problems. The resulting hybrid code generation aims to bring the flexibility of LLM-based code generation to LDE. It also aims to provide better control over the generated code and more effective ways to validate that the LLM output aligns with users' expectations. To achieve these goals, the work presents two code generation approaches, which are covered in the attached publications, and combines them.

First, it covers an LLM extension to LDE. The approach establishes a two-step process for the code generation. In a first step, a conventional code generator produces parts of the output code. This step also generates a Prompt Frame from graphical models. The Prompt Frame provides the LLM with context about the domain, modeled information, and expected output form requirements. Since the included model information originates from a DSL model, the Prompt Frame intertwines the formal model and natural language, creating a Domain-Specific Natural Language (DSNL). This allows users to reference model entities using natural language and specifies the larger domain context, just as DSLs do. The DSNL is then used in a second generation step to enable guided and controlled usage of the LLM.

Second, Code-centric Code Generation (CCG), an extension to Template-based Code Generation (TBCG), is presented. It creates an abstraction layer over templates and generates them. Unlike the conventional development of string-based templates, code and transformation operations are managed separately in CCG. Prototypical applications provide the code basis and are parsed. Then, operations for the templates can be mapped onto the prototype's Abstract Syntax Tree (AST). Separating code and operations keeps the prototype parseable and runnable. This allows it to be used as a central artifact for the further development and maintenance of templates, and consequently, code generators.

When combining these two approaches, CCG will be extended to be usable in the second generation step of the LLM extension to LDE. Instead of allowing the LLM to freely generate arbitrary code, the template generated by CCG will be used to constrain the LLM's code generation. This provides control over the LLM's output capabilities.

Figure 1.1 illustrates the simplified architecture of the combined hybrid code generation. In summary, the approach comprises the following two-step process:

1. A DSL model, for example, a graphical model, is the basis for the entire generation process. This model serves as input for a manually implemented conventional code generator that outputs Base Code and a Prompt Frame. As the name suggests, the Base Code forms the basis of the overall resulting application's code. The Prompt Frame contextualizes the generation process for the LLM and allows for more control over it.

2. The LLM uses a Natural Language Description, the prompt of the user, embedded in the generated Prompt Frame from the first step as input. In the combined approach

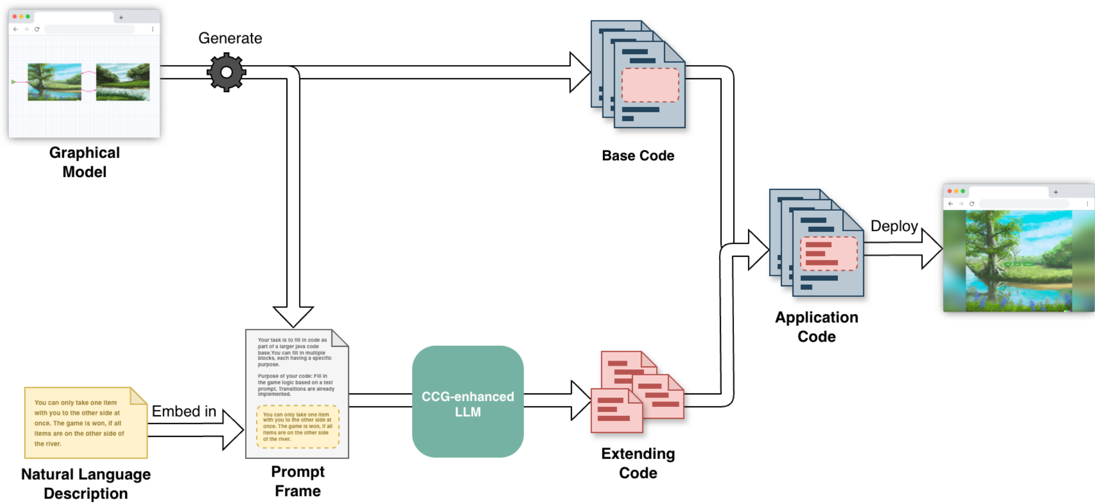

Figure 1.1: Minimal Overview of the Hybrid Code Generation Architecture in LDE

presented in this work, the LLM also has access to templates, generated with CCG, that it can use as tools. Details about this are provided in Chapter 5. With the Prompt Frame and the control introduced by the CCG-generated templates, the LLM outputs code that extends the Base Code from the first step. The outputs of both steps combined form the code of the resulting application.

The proposed hybrid code generation in LDE takes advantage of the flexibility of LLM-based code generation. It also introduces three layers of control over LLM usage:

- **Contextualization.** This layer controls the input for the LLM. Integration of LLM-supported code generation into LDE creates a new synergy of DSLs and natural language. It allows to easily express requirements that are difficult to formalize. This enables LDE to be applied to a broader variety of scenarios. Contextualization for the LLM is generated in the form of Prompt Frames. By intertwining natural language with information of the existing DSL models and general implications of the respective domain, DSNLs are formed. This enables users to reference model entities with natural language. DSNLs shift the responsibility for a good prompt from the user to the language developer that is responsible for the generator of the Prompt Frame. This makes it easier to achieve good generation results with the LLM.

- **Validation by design.** This layer enables system-level validation to control the quality of the output. It observes whether the generated output meets user expectations. Exploiting control over code generation ensures that Active Automata Learning (AAL) can infer behavioral automata for validation purposes. This dissertation presents an extension to CCG that allows LLMs to easily use CCG-generated templates. Using templates as tools for LLM-supported code generation also allows for guaranteed output structures. This enables the systematic integration of code instrumentation into code generators. The instrumentation can then be used for learnability-by-design [5]. Learnability-by-design allows to infer behavioral automata through AAL. With these automata, validation can be done visually or automatically using model checking.

- **Output Constraints.** This layer controls the LLM during code generation. Extending CCG allows for the easy integration of TBCG as LLM tools. Using this tool puts LLMs into a mode in which they cannot generate arbitrary output or potentially unwanted code. Instead, LLMs can only use the provided TBCG tool, and thus can only output within the limitations of the provided template. This constrains the LLM output during code generation.

This dissertation is structured as follows. The remainder of this chapter covers my contributions to research and my work at the university of the last years. Chapter 2 provides basic background knowledge needed to comprehend this dissertation. Next, Chapter 3 presents the LLM extension to LDE as in the attached publications [19, 20, 21]. CCG is also covered in attached publications and introduced in Chapter 4 [18, 22]. Chapter 5 presents the combined approach of the LLM extension in LDE and the extended CCG. Related work is covered in Chapter 6. Lastly, Chapter 7 closes this work with a conclusion, some words about limitations of this approach, and ideas for future work.

1.1 My Contributions

During my time at the Chair for Programming Systems, first as a student and then as a research assistant, I contributed to almost all of the active projects. I was initially part of the DIME team. DIME is a low-code development environment which provides several graphical modeling languages for creating web applications [16]. While on this team, I helped develop several DIME products, i.e. web applications created with DIME. DIME and its products always were subject to numerous changing requirements, especially since DIME was continuously developed alongside the work on the products.

During this time, I first considered enhancing TBCG to avoid unnecessary roundtrips. While working with DIME, we often tended to identify bugs or other issues in the generated code and fixed them there first. We then located the affected code in the templates from which it was generated and fixed it there as well. Typically, we generated code from the fixed templates again to ensure that the problems were no longer present in the generated products. In my work, I aimed to eliminate these unnecessary and cumbersome roundtrips.

Later, I joined the Cinco team and, with it, the team working on the foundations of DIME. Cinco is a development environment for domain-specific graphical modeling tools [51, 55]. It provides textual DSLs to meta-model graphical languages and generates tooling for these languages. When some colleagues presented the first prototypes of the Cinco Cloud version that was no longer based on Eclipse RCP and was fully available as a web application, I contributed significantly to the planning and partially to implementing the full feature set of Cinco into Cinco Cloud. Eventually, we dropped the “Cloud” name and made Cinco Cloud the main Cinco project.

At the end of 2022, ChatGPT was published and turned parts of the code generation community, and surely almost every computer science community, upside down. Like most, my colleagues and I investigated this new technology intensively and explored its implications for our code generation approaches, especially in the context of LDE. As a result, Gerrit Nolte and I, together with Alexander Bainczyk, came up with the ChatGPT in the Loop paper [21]. At that time, the use of the new LLMs for code generation was often criticized for its uncertainty and poor performance, regarding bad output quality. It was clear to us that we wanted to overcome these negative aspects by creating synergies when used together with our established approaches regarding LDE and validation using AAL. At the same time, I continued working with the Cinco Cloud team and later revisited ideas around code generation without LLMs.

In addition to always working on the tools of the Chair for Programming Systems, my main research contributions as a doctoral student can be sorted into the following two categories:

Enhancing Template-Based Code Generation

Due to the roundtrip that my colleagues and I experienced while working with DIME, I developed ideas to eliminate the extra step of transferring fixes from running applications to the templates they were generated from. In this context, I came up with CCG for the first time [18]. Shortly before this, my colleagues discussed how amazed they were by how MPS and its projectional editing essentially works directly on the AST [12, 85]. Consequently, I then thought about mapping transformational operations directly onto the AST of programs. The core idea of mapping operations onto the AST to generate templates for code generation, without making the original code unparsable as string-based templates do, emerged.

Through this central idea, CCG enables the development of code generators based on runnable prototype applications. Furthermore, it encourages keeping this prototype as a central artifact for the further development and maintenance of the code generators. If problems are discovered in the prototype, they can be fixed right at place. Depending on the position of the mapped operations, changes can be translated into the resulting templates without adding an additional development step beyond fixing the error in the prototype code. As a result, my contributed idea can mitigate or even eliminate the round trip problem entirely in some cases.

Creating Synergies of LLMs and LDE

While experimenting with ChatGPT in late 2022 or early 2023, Gerrit Nolte and I had the idea to incorporate LLM usage into the chair's go-to example of LDE and Cinco, the WebStory. This example is a graphical language to create simple HTML-based point-and-click games. It is the same example Cinco language used in this dissertation. Eventually, we developed the concept of generating code partially from our graphical models and completing it with LLM code generation. We achieved this by generating detailed prompts from the graphical models using a manually implemented code generator. This enabled us to use natural language to generate additions beyond what could easily be generated from simple graphical models. Combining LLMs and LDE made it possible to define crucial application aspects in a structured, formalized way using our graphical DSL and conventional code generation. LLMs allow for flexible code generation, such as for game or business logic. In this way, we added value to LDE in terms of flexibility and to LLM code generation in terms of control.

In discussions with Alexander Bainczyk and Bernhard Steffen, we came up with the idea of adding AAL to our new approach. To do so, we instrumented the code in the conventional code generation part so that we could infer behavioral automata from the application, regardless of what the LLM generates. The basis for this was the Malwa tool that Bainczyk presented in his dissertation [5, 46].

Furthermore, we used the inferred behavioral automaton for visual validation and also for model checking properties on it. This enabled us to establish a feedback loop through which LLMs can fix their own mistakes when they are pointed out to them. By combining these approaches to integrate LLM usage into LDE appropriately, I was able to add new possibilities for LDE users regarding flexible code generation based on natural language. I

also contributed to making LLM code generation more controllable and comprehensible in the context of LDE.

1.2 Context of Attached Publications

This section briefly summarizes the attached publications. It also provides context on how the publications contribute to the overall ideas presented in this dissertation. Additionally, it breaks down my contributions to each publication.

ChatGPT in the Loop: A Natural Language Extension for Domain-Specific Modeling Languages

The initial idea of extending LDE with LLM code generation is presented in [21]. It is the first paper to present the two-step process for full code generation: the first step uses classical modeling and conventional code generation, while the second step uses natural language as input and LLMs for code generation. To achieve this, it introduces the concept of generating a Prompt Frame from a domain model. The paper also establishes the term DSNL as a contextualized form of natural language that implicitly integrates the user prompt into a domain. This allows referencing domain knowledge contained in the model of the first generation step without requiring the user to describe any of it themselves. The resulting, full code-generated application is formed from code generated by both steps, conventional and LLM generation.

Additionally, the feedback loop for the LLM is established. During the conventional code generation step, code instrumentation is introduced that later allows to infer behavioral automata from the resulting application using AAL. These automata can be used visually or with model checking to provide feedback to the LLM. The LLM can then use this feedback to correct itself, creating a self-repairing loop mechanism.

The presented approach has been discussed and worked out by all authors. I am the main author of Sections 1, 2.1, 3.1 to 3.3, the entire Section 4, and Section 6. For Section 5, I was a co-author. Implementation work has been done by me, in collaboration with Gerrit Nolte. Validation-related work has been done by Alexander Bainczyk.

Towards LLM-Based System Migration in Language-Driven Engineering

This paper was awarded Best Short Paper at the 8th International Conference on Engineering of Computer-based Systems (ECBS) in 2023 [20]. It is based on “ChatGPT in the Loop” [21] and takes advantage of its LLM extension to LDE. The paper aims to migrate the conventional code generator used in the first step of the two-step generation in a semi-automated way using only LLMs for the migration process. The migration should result in the generated application being in another programming language. The idea behind this aim is that LLMs tend to perform better with fewer tokens, and migrating only the code generator instead of an entire application tends to be a smaller task.

The paper uses AAL as a control mechanism for a successful migration. To do so, the two-step generation process is executed before and after the migration. Using AAL, the behavioral automaton for both generated applications is inferred. Malwa, by Alexander Bainczyk [5], creates a difference automaton that highlights the behavioral differences of the learned applications. If any transition leads to different application states, the migration is deemed unsuccessful. This is because, when migrating an application, one would expect identical semantic behavior before and after.

Therefore, this paper highlights another beneficial property of the LLM extension to LDE.

The presented approach has been discussed and worked out by all authors. I am the main author of all sections except Section 2.2. The AAL parts of our experiment have been executed by Alexander Bainczyk, while I executed the LLM-powered migration.

LLM-based code generation and system migration in language-driven engineering

This paper [19] is the journal version of "Towards LLM-Based System Migration in LDE". Essentially, it presents the same concept, but with corrections and a more detailed presentation.

The further development of the original approach has been discussed and worked out by all authors. I am the main author of Sections 2.1, 2.2, 3 excluding for 3.1.4 and the validation part of Section 3.2, as well as Section 4 in its entirety. The other authors and I collaboratively wrote Sections 1, 5, 6, and 7.

Towards Code-centric Code Generators

The introduction of CCG and its fundamental ideas are presented in this paper [18]. It describes the separation of prototypical code and the transformational operations applied to it. To accomplish this, it establishes a mapping of these operations onto the prototype's AST. The paper also describes generating a template for code generation from the prototype and its mapped operations without going into detail.

I am the sole author of this paper and thus developed the entire approach on my own.

Code-Centric Code Generation

This paper [22] extends "Towards Code-centric Code Generators" [18] and further develops its ideas. It is the first publication to present an implementation of CCG, along with an example. Thus, this paper provides the first proof of concept for the CCG approach.

The further development of the original approach has been discussed by all authors. I am the main author of Section 3. All other sections were written collaboratively by all authors.

Cinco Cloud: A Holistic Approach for Web-Based Language-Driven Engineering

Cinco is the primary LDE tool used by the Chair for Programming Systems. This paper [7] presents the first fully browser-based implementation of Cinco. Previous implementations were based on the Eclipse Rich Client Platform. This contribution is based on Philip Zweihoff's dissertation [93]. The paper describes the implementation as a web application offering a holistic approach to LDE. The application provides textual languages for modeling graphical languages and features the use of the modeled graphical languages. It also provides capabilities to implement code generators for these graphical environments to produce output from this abstraction layer. The paper also presents the WebStory example in Cinco Cloud.

Cinco Cloud is important in the context of this dissertation because it is the central tool that my colleagues and I used for LDE. In all of my LDE-related work, Cinco or Cinco Cloud have been used. The ideas presented in this dissertation and my other work have been substantially influenced by Cinco and Cinco Cloud.

The presented approach has been discussed and worked out by all authors. I am the main author of Section 1, and I co-authored Sections 2, 4, 5, and 6. After publishing this paper, I joined the Cinco Cloud development team and made significant contributions.

2

Background

This chapter provides the background knowledge necessary to understand the ideas presented in this work and the attached publications. First, Language-Driven Engineering (LDE), the main principle on which this work is based, is explained briefly. Next, code generation in template-based and Large Language Model (LLM)-powered nuances is presented. After that, Cinco, the LDE workbench developed at TU Dortmund University, is described. Finally, a brief overview of Active Automata Learning (AAL) is provided.

2.1 Language-driven Engineering

LDE is a low-code development paradigm. It can be considered an extension of Model-Driven Engineering (MDE). MDE uses formal models as central artifacts during development [30, 35]. Rather than being a mere byproduct, models are used to gather, store, and process information [14, 68]. Depending on the need, they can be used to generate code or transformed into other representations. Models are usually either textual or graphical, acting as an abstraction of a given domain. Textual languages that abstract a domain are referred to as Domain-Specific Languages (DSLs) [34, 54]. They are distinguished from General Purpose Languages (GPLs) by focusing their use on only one domain, for which they aim to deliver a suitable representation in which domain knowledge can easily be expressed. This makes it easier for domain experts to handle such languages. Graphical languages, also called Domain-Specific Visual Languages (DSVLs) [73], aim to make use even easier. They lower the cost of learning a new textual language. Rather than writing the textual syntax manually, graphical languages often come with drag-and-drop editors. Furthermore, graphical languages can provide an easier overview through visualization than textual DSLs can. These simple solutions, tailored to the domain, can give MDE an advantage over conventional development with GPLs [40, 41].

Low-Code/No-Code (LC/NC) approaches adopt many MDE ideas and are based on them [24, 32, 67]. They focus heavily on code generation, while MDE is often too, but not necessarily. In the case of no-code or zero-code approaches, they rely on full code generation. This means, they are expected to generate full applications without requiring users to contribute any code.

LDE is a special paradigm to approach LC/NC development. It shifts the focus of the development process to very specific and individual textual and graphical DSLs [6, 15, 74]. In LDE, each subproblem and stakeholder receives the most suitable DSL for solving the overall problem, if necessary or beneficial [38]. The key to this approach is to decompose problems into smaller subproblems that can be adequately described with a DSL. This resembles the common divide-and-conquer strategy. In LDE, the DSLs are

strongly intertwined by referencing each other. Together, they contribute to a larger central artifact, the “one thing”, meaning they contribute to aggregating information within the same domain while still contributing individually [52]. This leads to XMDD, different stakeholder contributing to a central model with languages tailored to their needs [53].

2.2 Code Generation

Code generation is prevalent in MDE environments, and thus, in LDE environments as well. It enables the generation of code solutions based on source models. These solutions can be partial, such as code stubs from Unified Modeling Language (UML) models, or fully functioning programs, as in LC/NC contexts. Full code generation, in particular, can be complex and extensive. This requires good scaling and the ability to easily maintain and further develop existing code generators. As with other software, code generator output may be subject to changing requirements, either due to the need for new features or security implications.

This section presents two common approaches to code generation. First is Template-based Code Generation (TBCG), a common and structured approach in which string manipulations of existing code templates are performed based on input models. Next is LLM-based code generation. This approach, which has emerged in recent years, utilizes a variety of techniques to use LLMs to output code instead of natural language. This leads to a very versatile and flexible approach, though it also comes with some downsides regarding determinism and reliability.

Template-based Code Generation

TBCG is a target-driven approach to generating code [76]. In MDE, this approach is ubiquitous for implementing code generators. These generators are considered the de facto standard in this context because of their target-driven nature, which provides good flexibility.

One of the most notable advantages of TBCG is their effortless implementation. Template engines are source-agnostic and target-agnostic with regard to the implementation language used. The easy implementation is mainly characterized by the close resemblance to their output. Essentially, templates are large strings with the desired output, enriched with some DSL syntax that allows for certain degrees of variation. Thus, developers can use existing code and transfer it into templates with minimal manual work.

Templates consist of static and dynamic parts. Static parts can be considered hard-coded and are unaffected by input. Dynamic parts depend on the input source and produce custom outputs based on the source. These parts either introduce new data extracted from the source or modify the output structure through repetitions, conditional inclusions, and similar. Static and dynamic parts are intertwined by placeholders, statements, and expressions. The template parts of template-based code generators are typically implemented using template engines. These engines provide a DSL to incorporate the statements and expressions for the template’s dynamic parts. They also provide additional code generators that generate functions to instantiate templates and use their output in code generators. Prominent examples of template engines include Xtend String Templates [13], Free-Marker [36], Velocity [63], and Handlebars [44]. These engines all provide their own DSLs, which allow to allow for the integration of dynamic parts and statements within the static string parts of templates.

However, Syriani et al. criticize that migrating existing source code into template structures and syntax can still be complex and error-prone [76]. One aspect of this problem is the string representation of templates. While this leads to flexibility and target agnosticism, it also has drawbacks. As soon as the string is enriched with custom template DSL syntax, it becomes unparseable and therefore unexecutable by the language support of the original target language. This can cause problems when developing or maintaining existing templates. Tools commonly used in programming environments, such as linters, refactoring tools, and checkers, are no longer usable because the code cannot be parsed by the target language tooling. This can result in a roundtrip in the development process. Rather than editing the code directly within the templates, developers may be inclined to modify the generated code, test it, and then manually transfer the changes back into the strings of their templates to affect other generated code from this generator. This round-trip effect may be even worse when maintaining code or fixing bugs, which are sometimes “trial and error” tasks.

During my time working on DIME, we had experienced such roundtrips. Code generated from our DIME models, for example, contained security flaws that were detected by applying tools to the generated code. Although this made it easy to detect the flaws, we still had to fix them in the code generator that produced the malicious code. Thus, we first fixed it in our generated environment, validated that the fix solved the problem, identified the related position in the templates, fixed the problem again in the templates, and then we were able to retrigger code generation with this fixed template to propagate the fixes to all generated instances.

These roundtrip problems, which are related to a lack of tooling support, led to the research on Code-centric Code Generation (CCG), which is presented in Chapter 4. Thymeleaf [79] is a similar approach that maintains template code parsable and circumvents string representation problems. It instruments the target code with directives on how to modify the code during the code generation process. This allows for parsable code by deriving the dynamic parts of the code just when code generation takes place. One downside is that Thymeleaf is not target-agnostic because it relies on code instrumentation specific to each supported language. Thymeleaf uses Java as its generator language and can serve HTML, JS, CSS, XML, and text targets.

The templates mentioned in this subsection are often only part of the whole TBCG code generator. Additionally, a component is needed to convert the data from the source into the form required to instantiate the templates. Figure 2.1 illustrates the entire workflow of using a TBCG code generator and its artifacts. From a user’s perspective, the generator is a single unit. The input is the source, and the output is received by applying the generator to the input. In the case of MDE, the input is often a model, and the output is code. In the case of full code generation, the output is the complete code for an application.

From a developer’s perspective, there are two core components: the template and a component for template instantiation. As described earlier, templates are target-driven and responsible for the generator’s output, thus, they are usually developed by experts of the target domain. The second component is responsible for the template instantiation. It transforms the input model into the format needed by the template. Therefore, this component is necessary to make the template usable with sources from the domain in which the generator is used. Bringing the source into the required format involves a model-to-model transformation. In this process, relevant data is extracted, and some data is potentially computed based on the source. Since this component relies more on the source, it is typically developed by model domain experts. However, it could also be

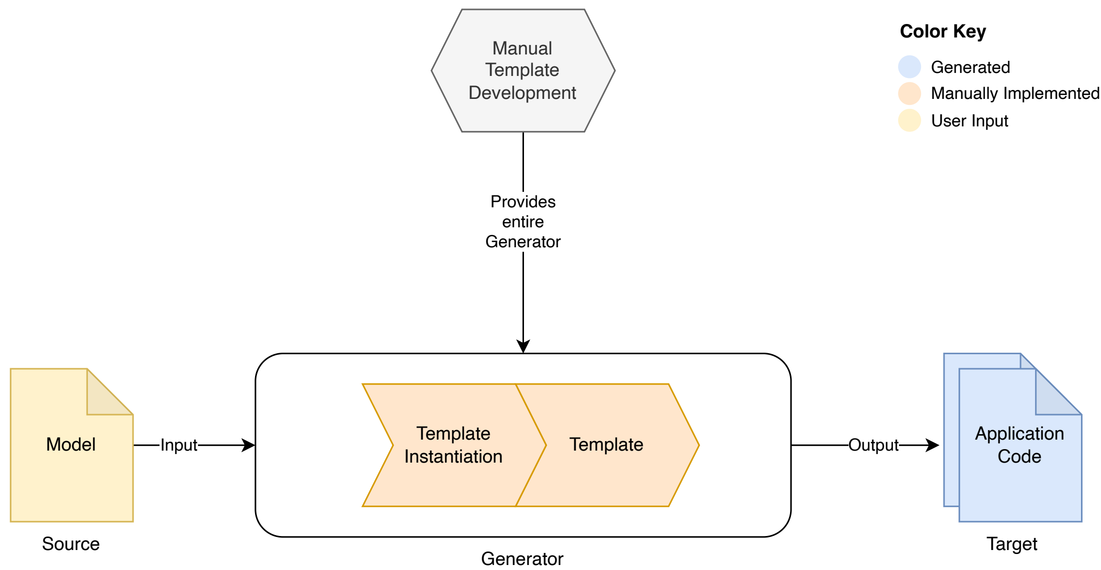

Figure 2.1: Workflow of Template-based Code Generation

developed by the same developers as the template because knowledge of the data required by the template is also necessary.

LLM-based Code Generation

In recent years, Artificial Intelligence (AI)-based code generation, more precisely, LLM-based code generation, has become increasingly popular. This was made possible by powerful new models such as LLaMa [82], OpenAI's GPT models [17, 60], PaLM [28], and many more. The rise in LLM usage led to its application in a wide variety of coding tasks [89]. LLMs are used as programming assistants [80], reviewers [49], or for full code generation [83]. Additionally, other generative AI models are used to generate images, such as Stable Diffusion [66], or music, for example, MusicLM [1]. These models demonstrate tremendous developments with increasingly better performance, for example, when benchmarked with the human-eval dataset [25].

LLMs used for coding tasks usually use high-performing models as a basis and are then fine-tuned for programming tasks using existing code repositories. This common approach applies to popular LLM-powered coding tools such as Claude Code, Copilot, and Codex. Many of these tools have direct Integrated Development Environment (IDE) integration, for example, in VS Code or in their own environments, like Cursor. These tools typically employ additional techniques to improve their performance further:

- **Retrieval-augmented Generation (RAG).** This technique provides additional relevant context to LLMs [11, 48, 92]. For RAG, the code base of a project, for example, the workspace opened in the IDE, is analyzed. LLMs then create so-called embeddings for the files, which are vectors representing the semantics of the code. These embeddings are indexed and later used to retrieve semantically relevant code in response to user prompts. When a user instructs the LLM to generate something, it looks up and automatically retrieves related existing code. This enables the LLM to integrate new code into the existing code base and also to follow the same coding styles. Additionally, it eliminates the need to provide more context than necessary.

Fewer tokens are needed, and there is less noise, which could have adverse effects on the performance of the LLM.

- **Self-repairing loops.** IDE-integrated, LLM-powered coding tools can execute the same tools used by regular developers. This includes compilers, linters, and testing frameworks. LLMs can then interpret the output of these tools to identify problems with the code they have generated. The LLMs can then use this information to iteratively fix or enhance their code, creating a self-repairing loop [26, 47]. This loop can greatly improve the performance of LLMs. Project-specific coding rules can also be enforced by the incorporation of tools like linters, resulting in code that fits into the project they work in.

- **Humans in the loop.** Although LLM-based code generation is becoming more effective, humans are still kept in the loop of creating code with LLMs. IDE integrations often provide tools, such as diff views, for easy supervision and review of code generated by LLMs. This resembles the common development process in which developers experienced in a project usually check the changes of other developers in the context of pull or merge requests. Additionally, most LLM integrations create several rollback points, allowing for easy reverting if the output does not meet the user's expectations.

The usage of existing development tools by LLMs to enhance the validation and trustworthiness of code generated by LLMs is a development that occurred in parallel with the research presented in this dissertation. Another central aspect of the presented approaches is keeping humans in the loop and maintaining control. This makes the approaches presented in this work consistent with most of the current popular developments around LLM-supported code generation.

2.3 Cinco

Cinco is a holistic solution for LDE that allows users to easily create and utilize graphical modeling languages [51, 55]. It is a meta-tooling suite, that was originally based on the Eclipse Rich Client Platform (Eclipse RCP). Cinco Cloud, its successor, aims to bring both the meta and the modeling layers into browser environments [7]. This allows it to exploit the benefits of browser-based applications over native clients, including zero setup time, device and OS independence, easier update distribution, and easier collaboration through centralization. The following paragraphs briefly describe the capabilities of the meta layer and the modeling layer.

Meta Layer. To be a suitable development environment for LDE, the meta layer must provide easy ways to develop and integrate new languages. For this Cinco provides two DSLs on the meta layer to allow for easy graphical language creation:

1. **Meta Graph Language (MGL):** The MGL is used to define the *abstract syntax* of a graphical language. This DSL is used to define available nodes, edges, and types, and their behavior and interplay.

2. **Meta Style Language (MSL):** Complementing the MGL, the MSL defines the *concrete syntax* of graphical languages. It adds style to the model elements defined in the MGL. Cinco provides a utility that makes it easy to apply different styles to node and edge elements.

Modeling Layer. Cinco aims to provide a holistic approach that offers editing capabilities on all abstraction layers during the LDE process. Therefore, the modeling layer provides graphical editors to display and edit models of graphical languages designed in the meta layer. These editors are complemented by additional tools aimed at supporting domain experts who lack coding knowledge or experience with typical IDEs. These tools include a validation view for semantic checks and inspectors for properties attached to model elements. They also have the capability to easily trigger code generation from models.

Cinco has proven to be a viable meta-tooling suite for different domains. With DIME [16], Cinco has been used to create a low-code modeling environment for web applications. Graphical languages for defining data schemas, processing data, and designing frontends enable users without coding experience to create complex web applications. Pyrus [94] is a Cinco product that provides an environment for modeling data analysis workflows and data flows. These workflows can then be delegated to and executed by underlying Jupyter notebooks.

2.4 Learning-based Testing and Evolution Control

Testing is a widespread and common field in software engineering with many aspects and approaches. Approaches for black-box testing include AAL[2] and learning-based testing [39], a fully automated approach that extends AAL with model checking.

In active learning, a learner interacts with a System Under Learning (SUL) via its public interfaces. By executing automatically constructed test queries over an input alphabet, the learner records the SUL's behavior and constructs an automaton that reflects the observed behavior.

AAL is often combined with model checking to verify system properties of the inferred behavioral model.

Web applications have a long history of being tested with learning-based approaches [6, 8, 57, 64, 65]. For this, Mealy machines have proven to be an adequate representation of user-level properties of the learned systems[1]. Typically, to learn web applications, an input alphabet and a system-specific mapper [42] that adapts interaction with the SUL are required. The learnability-by-design framework [5] eliminates this necessity by introducing `iHTML`. `iHTML` is a set of special HyperText Markup Language (HTML) attributes that enable learning frameworks to automatically infer the input alphabet from these attributes, eliminating the need to define the alphabet manually before learning time.

Previous research [57, 88] has shown that providing a stable alphabet abstraction is necessary to allow for the structural comparison of learned behavioral models. In this context, Bainczyk [9] introduces difference automata that can be inferred by learning multiple SULs simultaneously. These difference automata can be used for evolution control. To do so, two systems are learned, with one system being the evolution of the other. The result is a comparing difference automaton that can immediately detect and visualize divergent behavior during learning.

[1] As in the attached publications, the term *learning* refers to the application of AAL and is not related to deep learning or other recent AI-based approaches.

3

Controlled LLM Code Generation

Large Language Models (LLMs) are ubiquitous when it comes to programming tasks. They are used for support, as independent agents, reviewers, and system architects. While they lacked performance and reliability in the past, recent approaches like reasoning and agent orchestration led to a boost in almost every aspect. Nevertheless, LLMs can still hallucinate or make errors for any reason, just like human programmers. With supervised use of LLMs in software engineering, this may be a minor issue. However, as the autonomy of these systems regarding unsupervised work rises continuously, it becomes increasingly important to validate that the output LLMs create aligns with expected outcomes. Simply allowing LLMs to write their own tests is insufficient because these tests inherit the same limitations as the reliability of the code, as previously mentioned.

To mitigate these issues, we have researched combining LLM-supported code generation with Language-Driven Engineering (LDE) and learning-based testing and verification approaches. This combination decouples the creation and testing of LLM-created code and software. Integrating LLMs into LDE adheres to its principle that each subproblem should be solved using the most suitable tool or language. Then, LDE orchestrates each subproblem solution into a single solution for the full problem, similar to how LLM agents do.

The following sections cover the central aspects of integrating LLMs into LDE:

1. **Language Decomposition:** Using the most suitable tool or language for each subproblem.
2. **Contextualization:** Enriching and simultaneously restricting natural language to create a Domain-Specific Natural Language (DSNL).
3. **Validation & Feedback:** Exploiting LDE's orchestration to achieve learnability-by-design and taking advantage of this.
4. **Migration Capabilities & Change Control:** Extending the advantages of learnability-by-design to supervise changes in requirements.

This approach has been developed and presented in several attached publications [19, 20, 21]. Chapter 5 builds on this approach to make better use of the controllability of LLM-generated code introduced earlier.

This approach can be seen as an extension of LDE that adds natural language as another usable tool. This is achieved by contextualizing inputs for LLMs. The context adds information and limits the scope of work. It is similar to how Domain-Specific Languages (DSLs) hide context from users and limit what can be done, as opposed to a General Purpose Language (GPL), which makes it easier for users to express their goals in

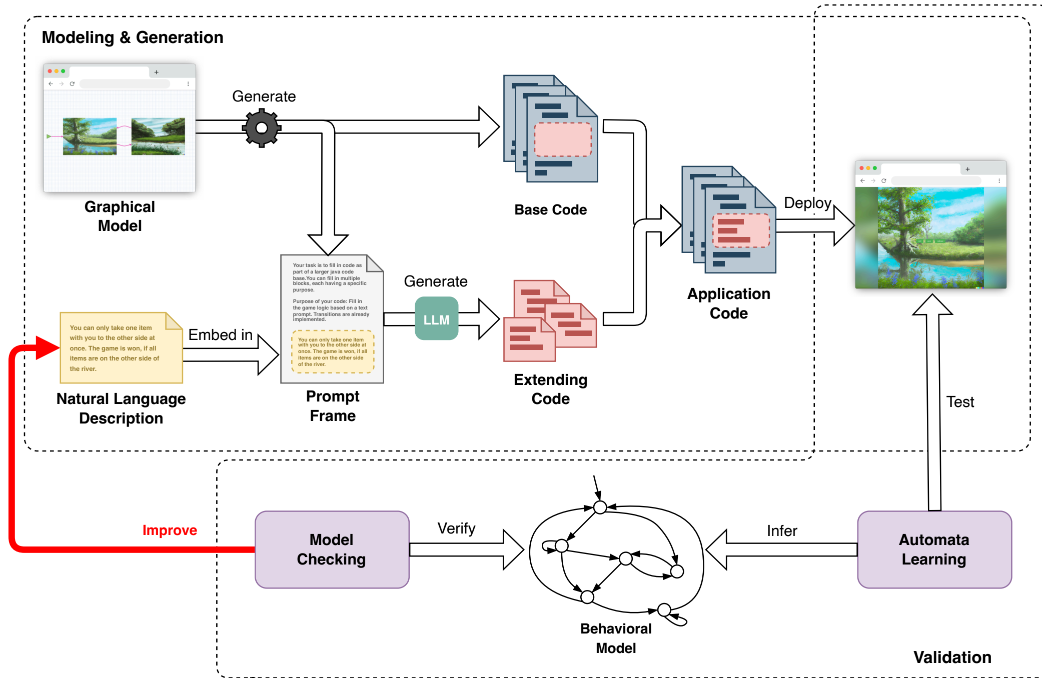

Figure 3.1: Concept of the Controlled LLM Code Generation in LDE; Reprint from [19, 21]

the DSL. Therefore, the contextualized natural language prompts can be considered as DSNLs. Contextualization is generated by other LDE tools and is usually presented in the form of a Prompt Frame. Figure 3.1 shows all the artifacts of the presented workflow, which are explained throughout this chapter. Generating context ensures the interplay of LDE tools by adding information about the code requirements for LLMs to be usable in the larger LDE environment.

While introducing natural language to LDE provides benefits, it also makes the environment more heterogeneous and less structured due to the lack of rule-based generation from an underlying model, as would the the case without natural language. This lack of formalization results in weaker validation capabilities for the resulting code. However, this can be mitigated by using learnability-by-design [5]. This has been presented in the attached publications [19, 20, 21]. Black-box testing, specifically Active Automata Learning (AAL) and formal verification techniques on inferred automata, can mitigate this issue. This approach allows users to validate the code generated by the LDE approach, including that produced by the LLM. Moreover, these validation capabilities also enable supervision of code evolution and migration processes by design. For instance, one can compare the generated code before and after migration, infer automata of the application at runtime, and create a difference automaton between the two. Given the same input, one would expect the same behavior, resulting in an empty difference automaton.

3.1 Language Decomposition

One of the central aspects of LDE is providing multiple languages and tools for solving a common problem. Each subproblem should be solved with the most suitable tool, and each partial solution will then be merged into the single final solution. For example, this

could be code for an application that solves the modeled problem. The presented LDE extension consists of tools, such as formal DSLs, and intuitive tools covered by DSNLs. These intuitive DSNL tools provide maximum flexibility, with only slight limitations in scope due to the contextualization.

Other LDE tools provide context. This context enables natural language to extend LDE with DSNL, and enable outsourcing subproblems to natural language (and thus LLMs). Since only parts that are typically solved with DSLs are outsourced, this process of splitting the domain is called *Language Decomposition*.

In the presented approach, the context is provided by generating the following with conventional code generation:

1. **Base Code/Code Frame:** The groundwork for the expected code resulting from the LDE approach is generated by a code generator that processes DSL models. This code provides basic functionality and extension points to vary the behavior of the generated code. Providing extension points instead of combining with arbitrary code maintains control over the overall functionality, and limits the resulting domain.

2. **Prompt Frame:** This is a natural language contextualization that is always provided to LLMs to utilize DSNLs. It informs the LLM about the overall environment, the available extension points in the base code, and additional requirements and limitations. It also includes basic information about the expected programming language and which properties it has to comply with.

LDE is extended by providing extension points through the Base Code which are then made usable by describing to LLMs how to use them via the Prompt Frame.

3.2 Contextualization

Introducing the formal aspects of a DSL into a LLM environment requires the LLM to retrieve information about the context of the domain in which the generated code should operate in, the properties needed, and the form of output expected. The Prompt Frame provides this “priming” information. It contextualizes the field in which the LLM will be used, thereby defining limitations and options with which the LLM must work. Using the Prompt Frame to narrow the domain makes user prompts with so called DSNLs instead of non-contextualized natural language.

In addition to limiting the context, the Prompt Frame is also used to enable the LLM to work with the base code. Over time, we have used different approaches to achieve this:

1. **Prompting:** Empowering the LLM by telling it which functions of the Base Code it can use and which properties are available. This is done using natural language. The code generated by the LLM can then make use of the described Base Code.

2. **Structured Outputs and Function Calling Modes:** Some LLMs are able to consume markup objects containing information about functions they can call or structures they are expected to output. The information includes function and parameter names and types, as well as semantic descriptions of the available functions in natural language. This is generally similar to the Prompting approach, but is more integrated into LLM usage. These LLMs’ responses are guaranteed to use functions syntactically correctly because the output markup is checked against the provided function calling information. However, function calling is primarily used by LLMs to invoke functions directly. We “mock up” the function calls and record

them to build the desired functionality. Additionally, this approach cannot provide properties like available variables of the Prompt Frame.

3. **Model Context Protocol (MCP):** This approach is more sophisticated than Function Calling. While also providing functions to call, it can also provide information retrieval (e.g., variables of the Base Code). Furthermore, this approach is standardized across different LLMs, unlike Function Calling.

In short, the Prompt Frame fulfills two main purposes:

1. Providing LLMs with information about the more formal DSL models of the LDE approach, and

2. Communicating the requirements of the output and usable extension points to the LLMs. This includes the expected programming language, general form, and more. Extension points can be communicated in one of the three ways mentioned earlier. Each would still be considered part of the Prompt Frame.

3.3 Validation & Feedback

Another advantage of decomposing the problem into parts that are solved using a conventional code generator and code generation by LLMs is having control over the properties of the code. This may include guaranteeing the inclusion of code instrumentation. With this, one can use AAL with code instrumentation to infer the input alphabet for the learning process. Bainczyk introduced this concept in his "lifelong learning framework" with his `iHTML` DSL for instrumentation [5]. The result is an automatically learned automaton that reflects the possible interactions on the user level. In a simple setup, this allows manual checking to ensure that the application's desired behavior is met. More sophisticated approaches could also check whether the learned automata satisfy Computation Tree Logic (CTL) formulae, for example. This makes it possible to automatically check for behaviors. For instance, one could check if web applications can always return to their starting page or if payment processes can always result in successful transactions. Feedback from model checkers could then be used to fix problems with the LLM code generation by providing it, along with the previously used prompts, to trigger the code generation again. This feedback loop is a self-repairing approach to code generation.

Past papers [19, 20, 21] required users to define their CTL formulae manually, but there is ongoing work trying to build feasible CTL formulae from natural language descriptions. With this, users could simply describe the expected behavior naturally, just as they use the code generation aspects embedded in LDE presented in this chapter.

3.4 Migration Capabilities & Change Control

In addition to its validation capabilities, the approach can be used to control changes, such as when migrating the target language of the code generation to another language. As depicted in Figure 3.2, this is achieved by using the same natural language description and input model for LLM-based code generation and conventional code generators, both before and after migration. The overall generation workflow remains unchanged, the top part of Figure 3.2 shows the same concept as in Figure 3.1. The lower part uses the same workflow and input, but different generators. The blue cog depicts a new, conventional code generator that outputs code for the migration's new *target language*. This generator also uses an altered Prompt Frame that changes the expected output format information

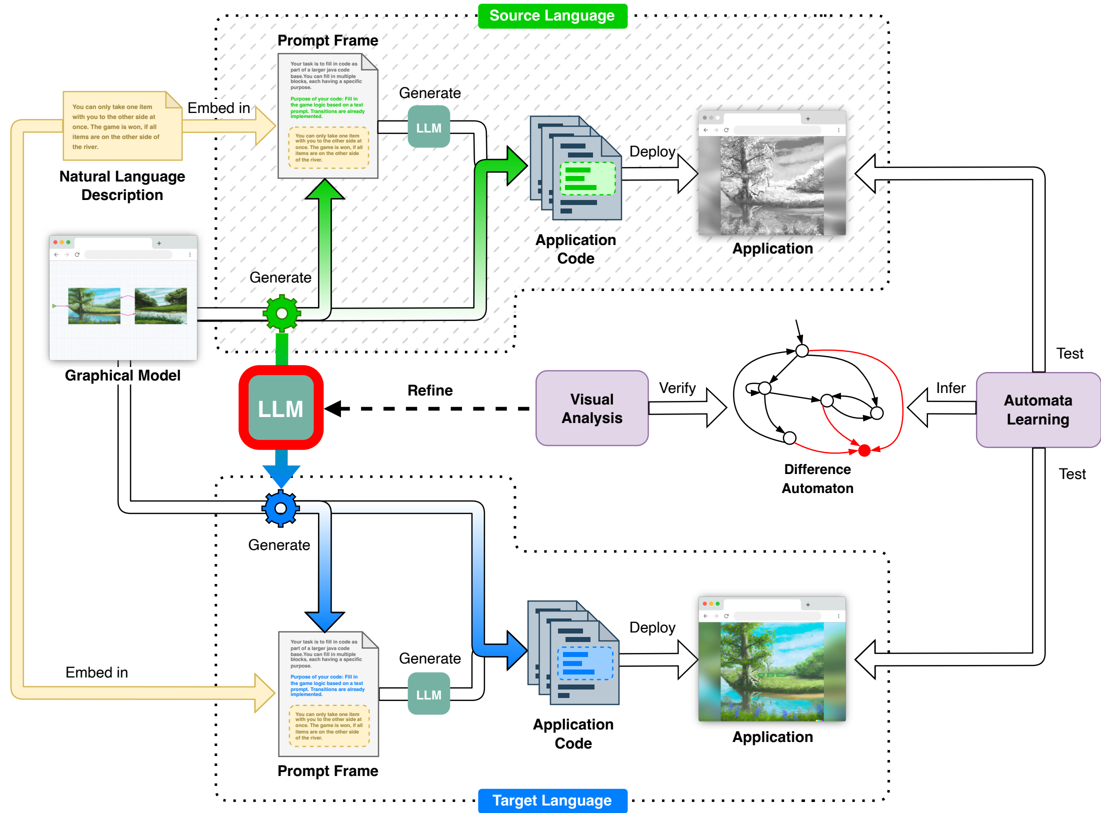

Figure 3.2: Concept of Evolution Control with the LLM Extension in LDE; Reprint from [19, 20]

to match the new target. Consequently, the entire application has been migrated to a different *target language* by modifying only these two components of the generation process.

The LLM extension of the LDE approach enables the automatic validation of the migration process. This is done by generating applications using the same input before and after the migration process. Benefits of instrumentation are expected to be exploited for generators before and after migration. This enables the automated inferring of behavioral automata, as described in Section 3.3. To check the differences between these automata, a *difference automaton* can be formed. Figure 3.3 shows an example difference automaton. It contains different reachable states in the highlighted area. In one case, a winning state of the underlying game application is reachable in the other, it is not.

These difference automata can detect potential problems in the LDE generator migration process. The information gained can be used to refine the generators and resolve the identified issues. In a fashion that tries to automate such migration processes as much as possible, one could outsource the migration task of the generators to an LLM, provide feedback on the success or failure for different example inputs, and then enter a feedback loop to let the LLM correct itself.

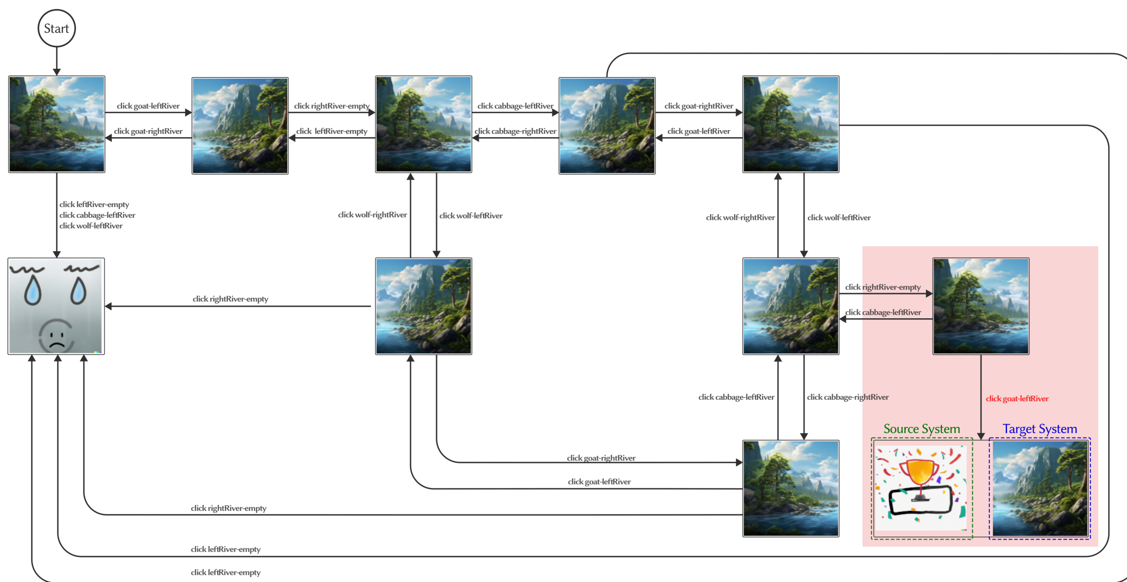

Figure 3.3: Inferred Difference Automaton; Reprint from [19]

4

Code-centric Code Generation

Code generation is a central aspect of Model-Driven Engineering (MDE) and, thus, of Language-Driven Engineering (LDE). It is used to create applications or structured data from (graphical) models. String-based template generators are a common approach for many low-code/no-code tools that require full code generation. These generators are simple to develop because they are relatively close to their output targets and only introduce transformational operations where variation is necessary. However, generators for full code generation scenarios can become large and complex, at the cost of maintainability and extendibility during further development. One reason for this is that they manipulate only strings with their transformations. The underlying code of the string-based templates is not parseable, so tools that would normally be used to maintain and develop code are not applicable.

Like any piece of modern software, code generators may also underly changing requirements. This could be due to needed modernization, changed dependencies, or to close security risks. During my time as a student in the working group at the chair, I also worked on the team that developed Dime, a low-code platform for modeling and developing web applications. Lack of evolution capabilities and tool support for string-based template generators frequently became apparent in this project. Most solutions resulted in a round trip: First, issues were identified in instances output by our generators. Next, the issues were fixed in these instances, and it was checked whether they were working properly. If so, the affected origins in the string templates were identified and fixed. These roundtrips were time-consuming, tedious, and sometimes led to further mistakes.

Code-centric Code Generation (CCG) aims to mitigate these issues by focusing on maintainability to eliminate the roundtrip problem and enable easy development and evolution of code generators. This is mainly achieved by making the prototype a central artifact of the code generator. Rather than translating the target output into a string representation, CCG works directly on output instances and transforms them by mapping operations onto the prototype's Abstract Syntax Tree (AST). This allows common tools to be used for development, just like with any other code. The result is a more "native" development and evolution experience for code generator development. This may be especially helpful in dynamic environments where requirements for code generators may require frequent adaptations.

The result of the CCG workflow is a template-based generator that closely resembles the underlying prototype (see Figure 4.1). The only difference when using CCG is how the template is developed and evolved. Instead of implementing the template manually, CCG is used to generate it based on the prototype application code and the transformational operations mapped onto the prototype. Users of the generator do

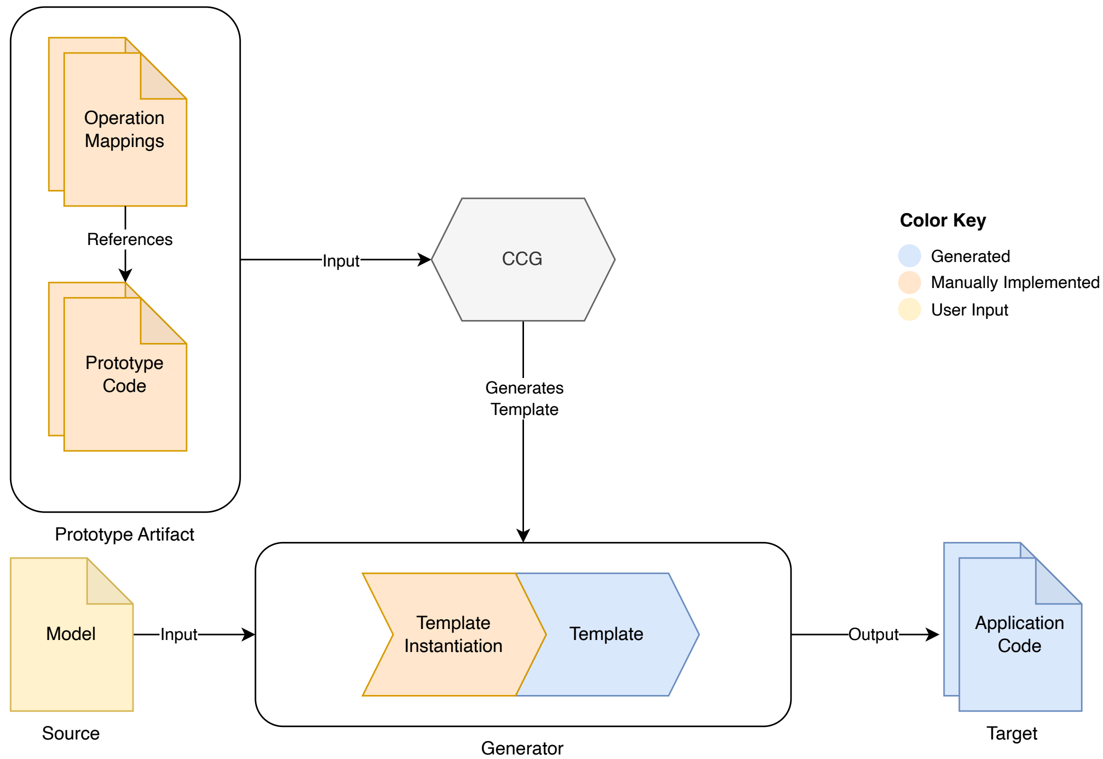

Figure 4.1: Workflow of CCG Code Generation

not face any differences compared to regular Template-based Code Generation (TBCG) code generators, as described in Section 2.2 and illustrated in Figure 2.1.

CCG's focus on prototype code leads to the following workflow for developing code generators:

1. Parse the prototype and construct its AST
2. Enrich the AST with additional properties, like IDs, resulting in a CCG tree
3. Map transformational operations onto CCG tree nodes
4. Generate the resulting template from the CCG tree
5. Use the generated template in a code generator

Based on this workflow, the generated template can also be easily further developed or maintained. For this purpose, the Prototype Code and the Operation Mappings remain central artifacts and are not omitted. Changes to static parts of the template can be made by simply adjusting the Prototype Code. While adjusting the code, it can still be executed or used in other ways to check the changes or develop with conventional tools and workflows. Changes to dynamic parts are made by adjusting the Operation Mappings. After applying all changes, the new template can be generated by invoking CCG again. Unlike conventional template development, manual translation into template structures or direct fixing within the template is not necessary.

The following sections cover the development workflow in detail. Chapter 5 then picks up CCG again and leverages it into the Large Language Model (LLM) extension to LDE described in Chapter 3.

4.1 AST Operation Annotation

CCG uses the prototype's code as its basic structure. This prototype acts as an example of a target instance that the code generator could produce. Any alternative output instance is achieved by applying transformation operations to the prototype's AST. In its simplest form, a CCG generator would output the prototype code unaltered. This is comparable to a string-template that hasn't been enriched with any operations yet.

In order to map operations onto the prototype's AST, CCG first constructs the AST and enriches it with additional properties. The remainder of this section describes which information is added and why. Since constructing ASTs requires a parser, CCG relies on tool support to map operations onto the prototype and generate the resulting code generator. The CCG can then select a parser that matches the underlying prototype language.

To construct the CCG tree, the tooling enriches the AST with the following information:

- **Unique IDs:** Each AST node gets two IDs. These IDs are used to precisely map operations and persist or reconstruct mappings from a persisted CCG file. This renders it unnecessary to persist the entire AST. Only the operations and their target IDs need to be persisted. The first ID, the top-down ID, is computed from the AST rule name and the path from the AST root to the node. The second ID, the bottom-up ID, uses the path from the leftmost leaf of the tree to the node. These two redundant IDs aim to make CCG more robust to changes to the prototype. Section 4.3 further elaborates on the effects of the redundant IDs and prototype changes.
- **Annotated Operations:** These are used to visualize and persist operations, as well as to restore operations from persisted files. The available operations and how they are used to generate the code generator are described below.
- **Custom Names:** Optional, user-defined names facilitate the identification of nodes among code generator developers. They can provide semantic information, simplify the process of assigning parameter values into the generated code generator (see Section 4.2), and be used to restore lost operation mapping information, as described in Section 4.3.

This CCG tree is then used to map transformational operations onto the prototype. Mapped operations are always applied to the entire subtree that the mapped node is the root of. The following four operations are available:

1. **Substitution:** Replaces the annotated node with arbitrary contents.
2. **Deletion:** Deletes the annotated node.
3. **Condition:** Wraps the node in a conditional operation.
4. **Repetition:** The node will be wrapped in a repeat operation. Depending on the context, this operation may lead to two different outcomes, see Section 4.2 for details.

Based on the mappings of these operations, versatile CCG generators can be created that produce transformed instances of the underlying prototype. Section 4.2 elaborates on the creation of these generators.

Mapped operations are persisted in CCG files. These files contain the path to the prototype, as well as mapped operations and user-assigned node names. Information

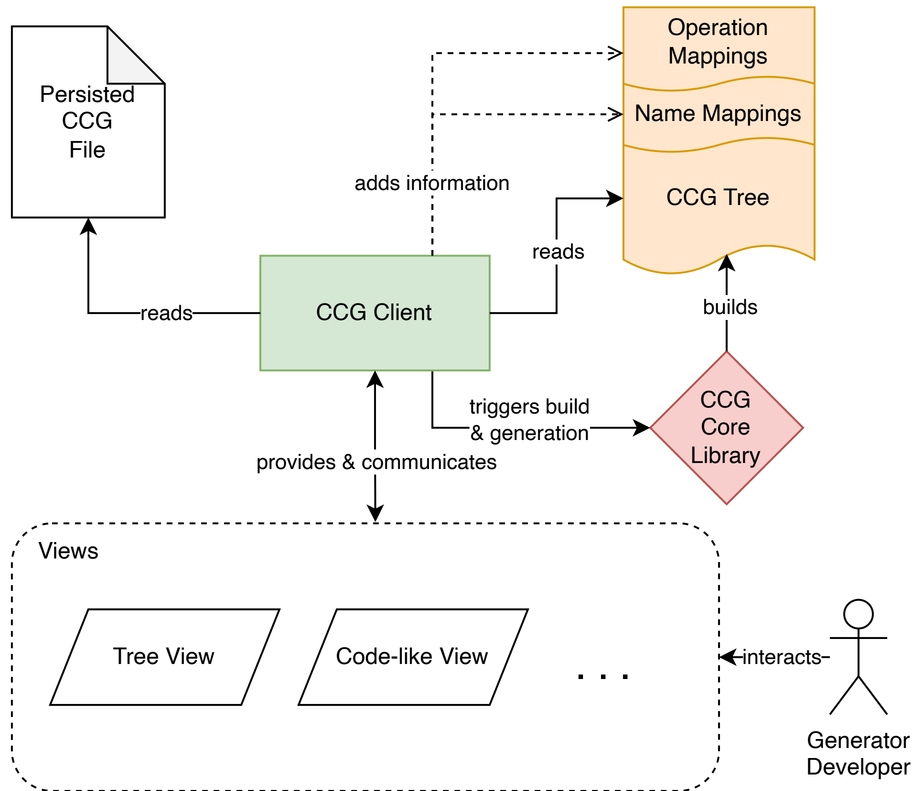

Figure 4.2: Overview of the CCG Client Architecture; Reprint from [22]

about mapped operations and custom names is stored alongside the redundant, unique IDs. ASTs are not stored and must be reconstructed from the referenced prototype, if necessary. CCG files are used to store, share, and retrieve CCG tree information.

Code generator developers use CCG tooling to add information to the CCG tree. The visualization is not predefined and can take different forms (see Figure 4.2). In [22] a proof-of-concept implementation is presented that provides two different views via a VS Code extension. One view closely resembles the typical hierarchical representation of trees, where nodes are colored to depict mapped operations. The other view resembles textual code editors, framing code blocks with different colors to depict mapped operations.

Figure 4.3 shows all artifacts involved in the CCG workflow. It highlights that, unlike in string-based template development, the prototype is not merely a byproduct that is discarded during development, instead, it remains a relevant and central artifact on which many other artifacts and workflow steps rely. The figure also illustrates the importance of CCG tooling (here, the CCG client). The CCG tooling builds the AST from the underlying prototype and constructs the CCG tree with its additional properties. CCG files store and reconstruct CCG trees. The tooling triggers the creation of the resulting generator from the CCG tree, as described in the next section.

4.2 Template Generation

CCG generates template-based code generators from CCG files to output code. The CCG generator parses the referenced AST and builds the CCG tree from it and the input CCG file to create the actual code generator. The annotated operations are translated into transformational operations for the generator, as follows:

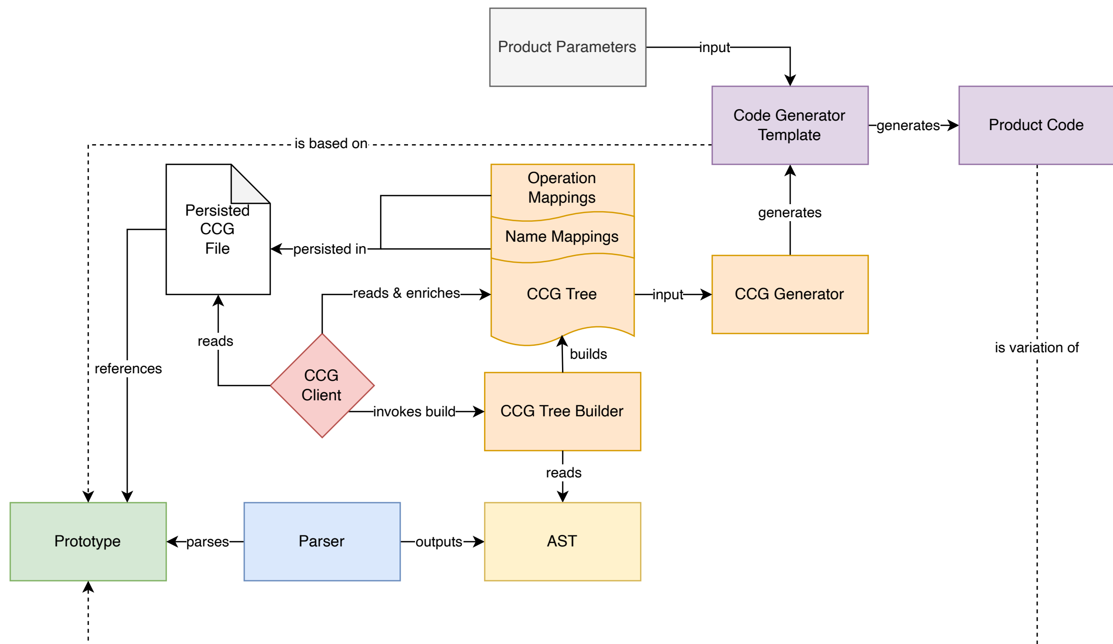

Figure 4.3: Overview of the CCG Architecture; Modified reprint from [22]

1. **Substitution:** Expects any string input for replacement. The substitution of contents can be checked for syntactic validity (see Section 4.4).
2. **Deletion:** This operation prevents the subtree's contents from being included in the generator and does not result in an operation inside the generator.
3. **Condition:** The conditional operation accepts a boolean parameter that determines whether the CCG generator includes the node in its output.
4. **Repetition:** Depending on the context, this operation may lead to two different outcomes: 1. If no operations (except for deletions) are mapped to any nodes within the repetition node's subtree, the repetition results in a repeat operation that expects a numerical parameter. The node will be repeated according to the value of its parameter. 2. If the subtree contains other operations, the repeat operation will expect an array of objects containing the subtree's operation parameters. The node will be repeated according to the amount of valid parameter input objects. These inputs will be fed into the subtree operations. Nested repetitions are also valid and may lead to hierarchical repetitions.

Each operation creates new parameters for the resulting generator. These parameter names are derived from either the top-down ID of the annotated node or user-defined names mapped to the node, which simplifies the identification of nodes and input parameters. The parameter types match the needs of the described operations. Nested parameters may also occur if the second case of repetition operations applies. With repetitions inside other repetitions, this can lead to nested parameters of arbitrary depth.

Although CCG mitigates the issues around string-based template generators, it outputs string-based template generators itself. The mitigation of issues is achieved by adding the prototype as a central artifact and working directly on it to generate such template generators. Because string-based template generators are familiar to developers, they can use generators created with the CCG approach as they normally would. This includes

calling the generators in other program contexts or CI/CD, orchestrating them to generate larger outputs (e.g., for full code generation scenarios), and extracting parameters from sources, such as Domain-Specific Language (DSL) models, and feeding them into the generators. As in most scenarios where models are first class citizens (here, the prototype is considered a central model), generated code should never be modified manually. Thus, any necessary changes to CCG generators should be applied to the prototype's code or the mapped operations, but never to the output generator itself. Otherwise, this could lead to a discrepancy between the modeled generator behavior and the generator's output.

4.3 Evolution

Changes to the generator output requirements may require changes to the underlying prototype's code. However, modifying the code typically results in changes to its AST as well, which can cause problems for mapped operations. One of CCG's central aims is to prevent loss of information when the underlying prototype has been modified. Ideally, code generator developers could modify the prototype, regenerate the generators with CCG, and generate output with the same inputs, propagating changes to all instances of generator output. This would ensure that all changes made to the prototype are transitively applied to all generator output instances. To make this achievable, CCG introduces techniques that mitigate possible loss of information due to changes to the prototype.

Changes that are orthogonal to the mappings are trivial to handle. These changes do not affect the node IDs of the orthogonal parts of the AST, so all mappings remain intact. However, three other types of changes could potentially cause a loss of mappings:

1. **Changes between the root node and an annotated node.** Problem 1 directly results in a change to the annotated node ID, because they are derived from the tree path. However, the enriched CCG tree contains another, redundant, bottom-up ID derived from the path of the leftmost leaf up to the node. If this second ID remains valid, the mapping can still be matched to a node. This allows lost mappings to be restored, and the top-down ID in the CCG file can be updated to the new value after changes to the prototype are made.

2. **Changes to the annotated node itself.** The second way of potential mapping loss could be resolved by looking at the parent ID and the relative position of the changed node. CCG can ask the code generator developers whether the matching should still be applied to the n-th child of the original parent. However, changes to the annotated node itself may imply changes to its semantics, thus automatic repair of the mapping is not provided in this case.

3. **Removal of an annotated node.** The third scenario is the removal of an annotated node in the prototype's code. This is an intentional "loss" of information but may result in a dead reference in the CCG file. Therefore, the code generator developers should also remove the mapped operation. CCG will still try to match the existing mapping even if it cannot find the related node. It should then notify code generator developers of this problem so they can remove the mapping.

In general, if the repair mechanism through the redundant IDs does not work, CCG should point out these faulty mappings to developers. Potentially assigned custom names to the annotated nodes may provide semantic information about the mapping, making it easier to manually restore lost mappings or decide to remove them.

4.4 Further Development/Properties

While the main aim of CCG is to create code generators that are easy to develop and maintain by making the prototype a central artifact, there are additional benefits that can be enabled by prototype usage.

Automatic Analysis. One benefit is the potential for automatic analysis of the prototype. Since the prototype is a parsable code instance, analysis tools, such as Software Composition Analysis (SCA) tools, can be invoked on it. This is not possible with string representations, as they are used in string-based template generators. Since CCG code generator outputs are only a transformation of the prototype base instance, flaws identified by SCA tools may also appear in these outputs. The transitive property of code changes being applied to the prototype and reflected in the outputs of CCG-generated transformations simplifies the process of fixing flaws and may even make automated fixes possible in some cases. Flaws can be fixed in the prototype instance and carried over to the outputs after re-generation.

Syntactic checked substitutions. The substitution operation is perhaps the most radical as it allows an arbitrary string to be pasted into the AST. This applies to CCG as well as to string-based template generators. In CCG, however, the syntactic context is known because the AST context of substitution operations is also known. Thus, CCG can generate checks to validate whether the provided substitution parameter values satisfy the syntactic requirements of the underlying grammar. Conventional string-based template generators would require manual, potentially cumbersome implementations to perform such checks. While these checks are most significant for repetitions, other operations could also lead to invalid AST configurations. Nevertheless, the same contextual check mechanisms can be applied to make CCG code generation syntax safe.

5

Code-centric LLM-powered Code Generation

Language-Driven Engineering (LDE) is based on a modular principle that allows multiple, suitable tools to be combined to best utilize their individual strengths. This chapter shows that the two presented code generation approaches, i.e., Code-centric Code Generation (CCG) and the Large Language Model (LLM) extension to LDE, are well combinable. Together, they provide further advantages over individual use in LDE. The main goal of this combined approach is to continue to build trust in LLM-generated code through control and validating the outputs, as presented in Chapter 3.

Ideally, both the control and validation capabilities should come without adding additional steps to the usual workflow. The approach presented in this chapter adheres to this principle. To do so, it retains the validation capabilities introduced by the LLM extension to LDE, as described in Chapter 3. Embedding code instrumentation in code generators allows to infer behavioral automata of generated applications. The properties of these applications can be checked by analyzing the automata visually or by applying model checking to see if given properties hold. Control is given by allowing the LLM to implement only parts of the resulting application, instead of the full application code. For example, as described in 3, in the LLM extension to LDE, LLMs were only used to generate functions that checked for win and lose conditions, as well as functions that triggered transitions in the example game domain. Code instrumentation is only generated by the conventional generator of the first step. This ensures that the necessary code instrumentation is present to learn the behavioral automaton via Active Automata Learning (AAL). CCG can further refine this control. Creating a CCG-generated template and providing a schema to instantiate it provides LLMs with limited access to transformations of the underlying CCG prototype code. The schema is then used to put the LLM into a mode where its output can only be used for template instantiation. This prevents LLMs from generating arbitrary code.

Enabling CCG for use with LLMs requires only a small extension, which is presented in this chapter. Additionally, the benefits of CCG remain: Easy code generator development by being prototype-driven and attaching only transformational operations to it. It is also easy to evolve the generator by simply editing the prototype instead of, for instance, editing string-based templates directly.

As already mentioned, the most suitable tools should be used to solve each subtask in LDE. Consequently, LDE can be used with the LLM extension alone, the CCG extension alone, both extensions together, or any combination of approaches. This chapter focuses on the combined use of these two approaches in LDE and describes the resulting synergies.

5.1 CCG Template Instantiation

In order for LLMs to be able to use CCG, it needs to be given context about the operations that have been mapped onto the Abstract Syntax Tree (AST). It needs information about which parameters and which types it is expected to instantiate the CCG template with. Additionally, it needs information about the semantics of each parameter so that it can fill it with useful input. The LLM extension to LDE provides context through the Prompt Frame. This frame contains information about the functions that the LLM will implement, as well as the available Base Code variables and functions, target requirements, and serialized information of the graphical model. While this contextualization through the Prompt Frame would also be possible for CCG, a more direct and stricter method of contextualization could be used. For this purpose, CCG has to be complemented with the following two artifacts: 1. An option for generator developers to provide operation semantics. 2. A schema that describes the expected instantiation parameters and their types.

Operation Semantics. Information about the semantics of the mapped operations can be attached alongside the operations, mapped to AST nodes, similar to the custom names described in Chapter 4. This method of providing semantic context is integrated into the development of CCG generators and does not require additional steps to provide context. The context provides additional semantics to custom names that can already be mapped. Although, suitable names can also help LLMs to get the purpose of operations. The context should be provided in natural language and state the purpose of the mapped operation or, more specifically, the purpose of the output instance when applying the transformational operation. This is similar to the natural language descriptions of tool calls or skills in common LLMs or Model Context Protocol (MCP) tools.

Parameter Schema. In addition to the context of the mapped operations, the LLM needs to be provided with information about the parameters required to call the CCG-generated template. For this, CCG generates a supplementary (JSON) schema, that describes the parameters derived from the operation mappings. This schema contains the names and structure of the parameters, their data types, and their aforementioned semantic natural language descriptions. Figure 5.1 illustrates the mentioned artifacts of the CCG workflow with the presented LLM extension. With the schema and the included semantic parameter descriptions, the LLM is able to instantiate the template, making it essentially a part of the code generator. Compared to the standard Template-based Code Generation (TBCG) workflow illustrated in Figure 2.1 and the CCG workflow in Figure 4.1, the LLM replaces the component responsible for transforming the source into a format suitable for template instantiation.

CCG's template generation makes it easy to use templates with LLMs, with a small extension of natural language context to existing operation mappings and a schema describing the template's parameter structure. These capabilities can be provided to LLMs in different ways, for example, through structured output modes or tool calling modes.

The aim of the presented hybrid approach is to maintain and fortify control with this CCG extension over solely using the LLM extension for LDE. LLMs can only transform nodes to which operations have been mapped, and are not able to freely implement arbitrary code.

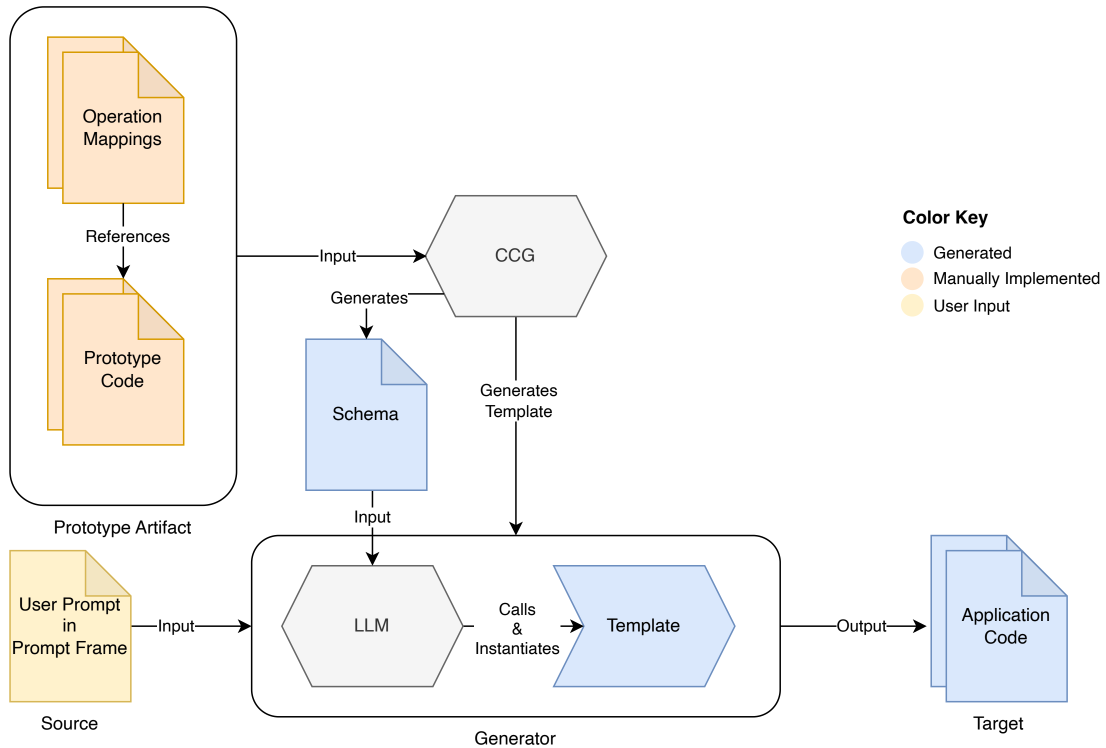

Figure 5.1: Workflow of CCG Code Generation with the LLM Extension to LDE

5.2 Combined Architecture

Figure 5.2 illustrates the concept of combining CCG and the LLM extension in LDE. While some parts are similar to the workflow described in Chapter 3, the LLM usage and the way context is provided differ. These changes result in the following workflow for the combined approach:

1. A (graphical) model provides the basis again. It is used to formally describe the fundamentals of the resulting application. For example, it can be used to describe valid states and transitions between them.

2. The model serves as input for a conventional code generator that provides the Base Code and the Prompt Frame. The Base Code contains the fundamental application structure and the necessary instrumentation to automatically infer the behavioral automaton later on. The Prompt Frame provides context about the graphical model. Unlike to Chapter 3, the Prompt Frame does not contain detailed information about the code generation instructions for the LLM. This is accomplished through CCG, which is described later.

3. As in Chapter 3, the Prompt Frame is merged with the User Prompt. Together, they form the resulting prompt for the LLM.

4. Additionally, the LLM is provided with a CCG schema. This schema describes the parameters with which the CCG generator is called. It contains the JSON schema to tell the LLM which object structure is expected to instantiate the template generated with CCG. Additionally, it contains the Operation Semantics, i.e., the description of what the template parameters are expected to do in natural language.

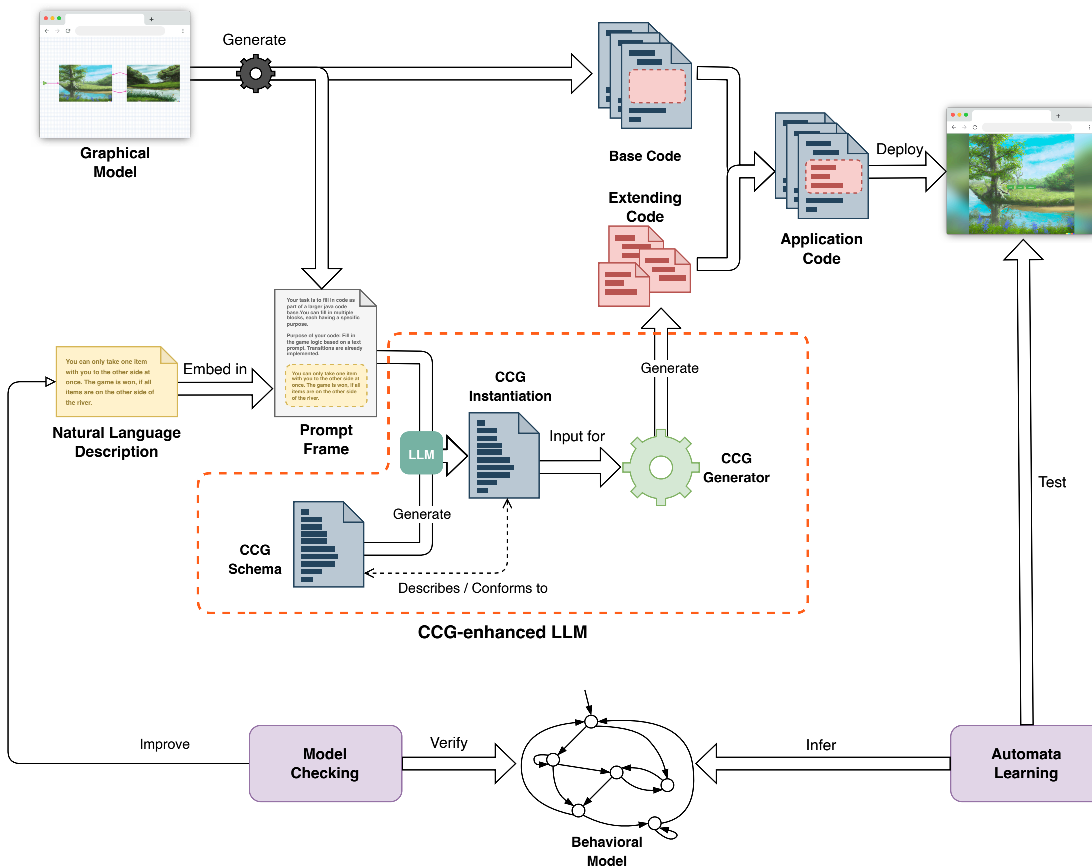

Figure 5.2: Combined Architecture of CCG and the LLM Extension to LDE

5. The LLM is then expected to produce a valid output based on the provided CCG schema. This can be achieved by using LLM structured output modes or tool calling modes, for example. The resulting output is then a valid object that adheres to the input schema, guaranteed by LLMs that support such modes. This output can be used to instantiate the CCG-generated template.

6. The LLM's output object is the input for the CCG generator. It uses the LLM output to call the template provided by CCG. The CCG generator then outputs a new transformed instance based on its prototype and the input operation parameter values.

7. This instance represents the extending code, which, when combined with the Base Code generated from the (graphical) model, results in the full application code.

8. The remaining pipeline is analogous to the workflow of the LLM extension for LDE, presented in Chapter 3. Code instrumentation is ensured by relying only on the Base Code, or the fixed parts of the CCG-generated code. This instrumentation allows to infer a behavioral automaton using AAL. The resulting automaton can then be used to validate the application's behavior.

With this combined approach, the graphical model remains a central artifact, and the surrounding workflow remains the same. The Base Code is generated from the model and provides the foundation for the resulting application. Additionally, the Prompt Frame

continues to contextualize LLM usage. This allows to make references to modeled entities and restricts the domain in which the LLM should operate. This means the approach uses a Domain-Specific Natural Language (DSNL) to narrow the domain and provide domain-specific capabilities, without the user knowing about it. However, unlike the LLM extension for LDE, the introduction of CCG into the workflow requires less context about the expected LLM output. Especially, the description of the structure can be omitted, e.g., the expected programming language and functions to implement, as this is handled by CCG instead.

The validation part of the workflow is also very similar to using only the LLM extension. A behavioral automaton is inferred from the running application using AAL. More precisely, learnability-by-design allows the automaton to be learned automatically, eliminating the need for manually created input alphabets. This is achieved by instrumenting the code and deriving the alphabet and other necessary properties for AAL from the instrumentation. To be certain that the instrumentation is present, this approach does not rely on the LLM to implement the correct instrumentation. Instead, instrumentation is solely done in the base code, as in the approach of Chapter 3, as well as in the fixed parts of the CCG-generated template in this combined approach. The CCG approach only allows transformations mapped onto the AST by the generator developer. The instrumentation can be included if it is not marked for transformation in a way that allows the LLM to modify it with any input. This allows for more flexible instrumentation that can be intertwined with LLM-generated parts, without risking to lose the instrumentation information. This is achieved by incorporating CCG into the workflow to provide the LLM output constraints.

As with the LLM extension, the learned automaton can be analyzed to validate the behavior of the learned application. This can be done by inspecting the automaton or performing model checking. Model checking can automate the validation process and establish a feedback loop. If problems occur, the feedback can be sent to the LLM to start a new iteration of LLM-supported code generation. The LLM can address semantic issues detected by the validation loop and resolve them when provided with the feedback.

5.3 Example

This section demonstrates the combined code-centric LLM-powered code generation approach for LDE, using a customized version of the WebStory Domain-Specific Language (DSL). The WebStory DSL is frequently used to demonstrate the Cinco workbench. In its basic form, the WebStory is a graphical language used to model browser-based point-and-click games. Figure 5.3 shows an exemplary WebStory model. It consists of screens and click areas that trigger transitions to different screens. The generated application is implemented using HTML and JavaScript. Several variations of the WebStory DSL exist. For this demonstration, only the most basic graphical parts of the original WebStory DSL are used.

This example follows best practices for splitting the domain to best-fit tool usage in LDE, as described earlier. Consequently, the basis of the solution is modeled using the graphical WebStory variant. It is used to express reachable game screens and to define valid transitions between game screens, as well as the starting screen. Game logic that would be difficult to express in a formal modeling language is described using natural language. Thus, the example uses an LLM to implement the described logic. For more control and to adhere to the combined approach, this step uses CCG to let the LLM output code.

```
[...]
function renderButtons() {
    const buttonsDiv = document.getElementById("buttons");
    buttonsDiv.innerHTML = '';
    buttons.forEach(button => {
        if (button.screen === currentScreen) {
            const buttonElem = document.createElement('button');
            buttonElem.innerHTML = button.text;
            buttonElem.onclick = createTransition(button.target,
                button.callback);
            buttonElem.setAttribute('data-lbd-action', 'Click');
            buttonElem.setAttribute('data-lbd-name', screen + '-' + button.id);
            buttonsDiv.appendChild(buttonElem);
        }
    });
}
[...]
```

Listing 5.1: Excerpt of the Base Code

First, the example covers the perspective of the language developers, who develop the WebStory language. Then, it covers the perspective of the users, who model the WebStory.

Language Developer View

This subsection describes the workflow for developing in LDE with combined code-centric LLM-powered code generation. It explains how to decompose generation subtasks into model-driven and LLM-powered parts, supported by CCG.

Graphical Language. As previously mentioned, the graphical model serves as the basis for the overall modeling. The WebStory variant is stripped down to provide only the available screens, their images, the initial screen, and the allowed transitions. This can be compared to a sitemap, although the screens do not necessarily correspond to specific states. Since game logic will be implemented using an LLM, it is possible to reach multiple game states on the same screen. For instance, players could collect certain items and return to a previous screen with them. Therefore, the game state is always a combination of the game logic state and the current screen. The differences between states and screens are highlighted later when presenting the game logic capabilities and validation through AAL.

Base Code Generation. From models of the graphical language, the Base Code and a Prompt Frame are generated. The Base Code manages rendering of game screens and provides functions for adding buttons to enable user interactions. An excerpt of the Base Code for button rendering can be found in Listing 5.1. The LLM is expected to add buttons to a global object using a separate function. When rendering the buttons, the Base Code sets up the buttons with the button object information that the LLM provided. The button rendering function expects a game screen name on which to render, the text to render, an ID, the name of the game screen to transition to, and an optional function that returns a boolean value to determine whether to trigger the transition and execute game logic. Setting up buttons in the global button object is made available to be called by code from the LLM and is not done by the Base Code itself.

```
## Quick Summary

- **Total Screens:** 7
- **Starting Screen:** house
- **Available Screens:** house,forest1,forest2,river,treasure,market,castle

---

## Context

You are contributing to a **browser-based adventure game** where players
    navigate through different screens, make choices via buttons, and
    experience win/loss outcomes based on game conditions.

The game framework is already built. Your task is to implement the missing
    game logic and content to make the game playable.
[...]
```

Listing 5.2: Excerpt of the Prompt Frame

Prompt Frame Generation. In addition to the Base Code, the Prompt Frame is generated from WebStory models. It serves two purposes: First, it provides context for the LLM, enabling users to implicitly reference entities of the graphical language in their natural language prompt. Second, it restricts the overall domain, making the LLM aware of the expected domain in which it should generate code. In this example, the provided context only contains the available game screens. The LLM only needs to know about these entities when generating code because the Base Code is responsible for checking for valid transitions. The Prompt Frame also includes the overall domain of the point-and-click game and a request to write creative and fun text for the buttons. Listing 5.2 shows an excerpt of the Prompt Frame generated for this example. Users can later use the provided Prompt Frame to describe their desired game logic with the DSNL. The available screen names are derived from name attributes of the screens, made available through the WebStory language, and defined by the users in the model.

Combined CCG and LLM Usage. In this presented combined approach, CCG also comes into play. Rather than letting the LLM generate code directly, as in Chapter 3, the combined approach uses a CCG generator for controlled code generation. To implement the CCG generator for game logic, language developers create an underlying prototype that implements a function to check win and lose conditions and statements that register example buttons using the available functions of the Base Code. The win and lose functions are called by the Base Code each time a valid transition is triggered. To make the relevant parts of the prototype transformable for the LLM, the language developers map suitable transformation operations on those parts. For instance, they mark the function to add buttons as repeatable, as well as each parameter, so that they can be replaced with custom content. The developers also provide names for the operations and semantic descriptions so the LLM knows how to fill in the transformation inputs. Listing 5.3 shows an excerpt of the CCG schema, adapted for use with the OpenAI Structured Outputs mode. It shows how the LLM is told to instantiate the template for button objects that are rendered with the Base Code, shown in Listing 5.1. Making code generation available through CCG provides the advantage of making language developers less reliant on a good prompt frame to produce good code. Instead, they provide the prototype and semantic context, along

```
{
    "type": "json_schema",
    "name": "generate_webstory",
    "strict": true,
    "schema": {
        "type": "object",
        "properties": {
            "buttons": {
                "type": "array",
                "items": {
                    "type": "object",
                    "properties": {
                        "text": {
                            "type": "string",
                            "description": "Text to be rendered inside the button."
                        },
                        [...]
                    },
                    "required": ["text", [...]],
                    "additionalProperties": false
                },
                "description": "Buttons to add interaction options to game
                    screens."
            },
            [...]
        },
        "required": ["buttons", [...]],
        "additionalProperties": false
    }
}
```

Listing 5.3: Excerpt of the Generated JSON Schema for OpenAI Structured Outputs Mode

with the template parameters. This is more aligned with the traditional development workflow, in which developers write comments or name variables to provide context to other developers.

Validation Loop. Language developers are not required to take additional steps for the validation loop. Automated validation is ensured by providing the right instrumentation. To maintain control and avoid reliance on the LLM, this instrumentation is provided through the Base Code and fixed parts of the CCG generator. In this example, the AAL tool Malwa is used to automatically run automata learning on instrumented code. The instrumentation is `iHTML`, which are special attributes added to HTML tags. Parts of the Base Code instrumentation can be seen in Listing 5.1. The instrumentation adds the `data-lbd-action` property, indicating that the learner can interact with the element by clicking on it. The second property, `data-lbd-name`, gives the button a unique name for identification purposes. No additional implementation work is required besides instrumentation, as this is sufficient to run Malwa on the generated code and infer a behavioral automaton from it. For the sake of simplicity, this example only relies on a visual analysis of the automaton. Therefore, language developers do not need to implement any model checking capabilities.

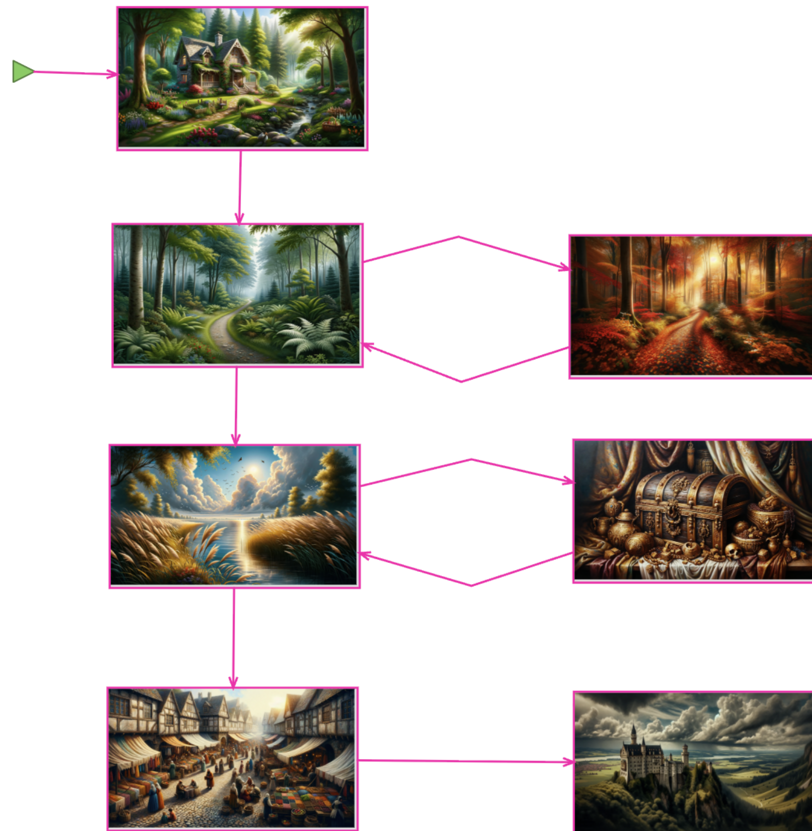

Figure 5.3: Example Graphical WebStory Model

User View

This subsection covers the use of the combined code-centric LLM-powered code generation approach for LDE. The workflow for users is the same as it is for the LLM extension in LDE alone.

Graphical Modeling. Users start by creating their graphical model to lay the groundwork. In this example, they use the WebStory language to graphically model the available game screens, the starting screen, and valid transitions between them. Figure 5.3 depicts an example WebStory model. In this example, the first screen is depicting a house. The starting screen is marked as the successor of the green triangle. From there, a forest consisting of two game screens can be accessed and navigated back and forth. Beyond the forest is a small river where players can discover a treasure. Finally, players can reach a market, and eventually, a castle. In addition to the modeled screens and transitions, users can give screens names to reference them more easily. This is not shown in Figure 5.3, as the example uses Cinco for modeling, which handles attributes of model elements in a different view.

First Generation Step. Users can use the generator that the language developers implemented to create the Base Code and Prompt Frame from the WebStory model. Cinco provides a Visual Studio Code (VS Code) extension that makes use of LLM code

```
Starting Point: The House
The player begins their journey at home.

The Forest (Stage 1)

From the house, the player travels to forest1.
The player becomes trapped in the forest area and can only move between
    forest1 and forest2.
This restriction continues until the player has visited forest1 at least five
    times.
After the fifth visit, the path to the river opens.

The River (Stage 2)

At the river, the player faces a choice:
 - Approach the stranger: The player receives a treasure and then proceeds to
    the market
 - Avoid the stranger: The player goes directly to the market without the
    treasure

The Market (Stage 3)

The player can purchase a present from a merchant.
Important: The present can only be bought if the player obtained the treasure
    at the river.

The Castle (Final Stage)

The player proceeds from the market to the castle and approaches the king:

Victory condition: Bringing a present to the king
Defeat condition: Arriving without a present
```

Listing 5.4: Example User Prompt

generation with the generated Prompt Frame and the provided CCG generator. When opening the extension, users can enter a natural language prompt to generate additional code contributing to the final application. In this example, the user enters the text in Listing 5.4 to describe the desired game logic. The user prompt includes a description of win and lose conditions, as well as constraints on how to reach several game states. Note that users can reference the game screens modeled in the WebStory model shown in Figure 5.3. The referenced game screens are highlighted in the listing. Users can reference the screens using the names assigned to them in the graphical model. This is achieved by the generated Prompt Frame, that provides context to the LLM. The VS Code extension automatically uses this context to provide the DSNL to users, setting the domain and enabling references to existing model entities.

Second Generation Step. When the user clicks "generate" in the provided VS Code extension, the LLM receives the full prompt and the CCG schema. The LLM then outputs a valid object that serves as input for the CCG generator. The CCG generator's result is then merged with the Base Code to form the final application. An example of the initial state of the running application is depicted in Figure 5.4. The VS Code extension automatically handles all generating, communication, and merging and does not require

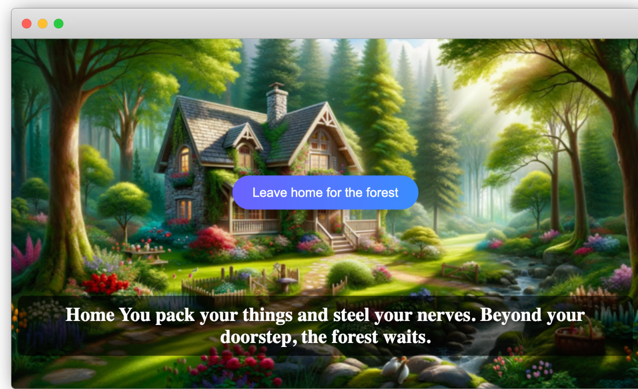

Figure 5.4: Starting Screen of the Generated Example WebStory

additional action by users. Since this extension is universal to the presented approach, language developers do not need to create it. Instead, the LDE engineers can provide the extension to language developers for them to provide to the final user.

Validation. After generating the application, users can provide a URL to the running application to Malwa. Since the instrumentation has been assured, Malwa can infer the behavioral automaton without requiring additional steps. An example of a learned automaton can be seen in Figure 5.5. It observes that the generated application begins with the expected `home` screen and allows users to move to `forest1`. From there, users can travel back and forth between `forest2` and `forest1`, until they can eventually reach the `river`. From the `river`, they can reach the `treasure` or the `market`, which leads to the `castle`. Taking a look at the automaton enables the user to comprehend whether the generated application implements the expected semantics, e.g., game logic, or not.

5.4 Synergies

The presented approach combines two powerful tools from the LDE landscape. This combination comes with several synergies, making it a viable approach. The most notable synergies include greater flexibility than conventional code generation and three layers of control unavailable in pure LLM code generation.

Flexibility. Compared to conventional code generation, the LLM support in the presented hybrid code generation approach offers more flexibility. Conventional code generation can only cover cases that the developers of the code generators have considered. Each case must be implemented separately and manually. LLMs, on the contrary, are less strict with their input and output.

Because the input is natural language, it is based on a universal interface that is natural to humans. However, natural language is ambiguous, which can be avoided in DSLs. Despite this ambiguity, LLMs always try to produce output that matches what they extracted from the input. Consequently, potential ambiguity could lead to a wrong interpretation and, thus, incorrect output.

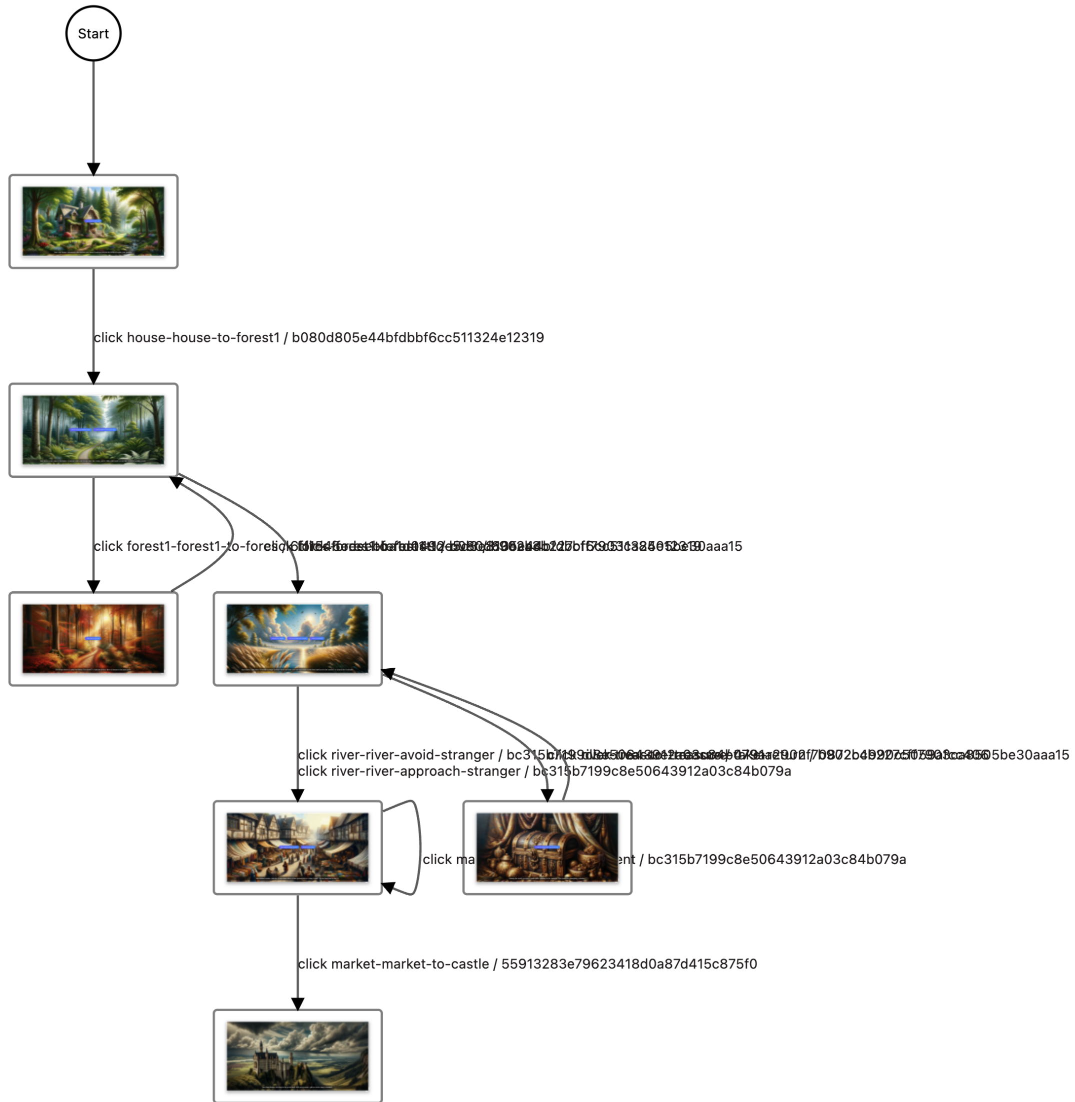

Figure 5.5: Learned Automaton of the Generated Example WebStory

The presented hybrid code generation takes advantage of the flexibility while minimizing the threat of ambiguity. This flexibility is achieved by integrating LLMs into the LDE workflow. This leads to flexible output that conventional code generators are not capable of. However, the full flexibility of generating arbitrary code is only available when using the substitution operation when using CCG-generated templates with the LLM. In domains where a high degree of control and certainty about the output is necessary, these operations and full flexibility should be used with caution.

Another dimension of flexibility is gained through template instantiation by LLMs. Rather than relying on information extraction from models by parts of the code generator written by developers, LLMs extract this information without rigid processing. This mitigates problems where information is contained in the input models that developers of conventional code generators would not expect, or where the model contains implicit information that conventional code generators would expect to be stated explicitly. With conventional instantiation, these problems could only be mitigated through complex analyses.

However, ambiguity remains a prevalent problem when relying on natural language, as LLMs do. To a certain degree, this problem can be mitigated with other synergies of the hybrid code generation, the three layers of control.

Three layers of control. The presented hybrid code generation approach establishes three layers of control that improve the handling of LLM code generation and increase trust in its outputs. The three layers are introduced as follows:

1. **DSNLs.** The first layer of control is established by shifting the responsibility of good prompting for the LLM from the users to the language developers. For this purpose, the Prompt Frame is generated in the first step of the two generation steps. The Prompt Frame provides context for the LLM, eliminating the need for users to describe it. Users can instead focus on their specific intent within the domain.

   The Prompt Frame has multiple parts that contribute to the context, making the natural language a DSNL:

   - **General domain description.** The Prompt Frame contains a general description of the domain. This provides a rough sketch of the context in which the LLM is expected to operate. For instance, the description indicates whether the context is less strict, as in the case of a game, or more strict, as in the case of a control system in an industrial setting. This general context includes implicit requirements that influence the LLM's output and behavior.
   - **Model context.** Providing information about the model from the first generation step is another crucial part of the Prompt Frame. It allows users to make references to things they have modeled, and the LLM can understand them. Therefore, all essential information contained in the model should be provided. For graphical models, this typically includes modeled nodes and edges, as well as the properties and relationships of the model elements. Information considered irrelevant to code generation, such as concrete syntax information (e.g., positional information in graphical models), should be omitted. This model context allows users to continue modeling with the flexibility of natural language. Moreover, it allows for an interplay between the Base Code generated in the first generation step and the Extending Code generated in the second step.

- **Output requirements.** The third important part of the Prompt Frame is the description of output requirements. These requirements include the necessary output language and functions to implement, among other things. They also contain additional information about the Base Code and the functions and data it provides that can be extended or used. However, these requirements, are considered "soft" because they are only described in natural language and are not strictly enforced. More strict enforcement is achieved by providing CCG-generated templates as a tool for the LLM, as described in the next layer of control.

2. **AAL-based validation.** The second layer of control is established by enabling the simple inference of behavioral automata of the generated code using AAL. Using a conventional code generator in the first generation step ensures the necessary code instrumentation is present. Additionally, using CCG-generated templates as LLM tools can lead to guaranteed code instrumentation, as previously described.

   This instrumentation can then be used to infer behavioral automata with AAL, as described in Section 3.3 and the attached publications [19, 21]. The inferred automata form the basis for validating whether the generated application semantically satisfies the user's expectations. For a more sophisticated validation, model checking can be used to automatically verify that the given properties are satisfied.

   The result of the model checking can then be used to establish a feedback loop. Within this loop, feedback about failed property checks can be provided back to the LLM. This can establish an iterative, self-repairing approach by incorporating knowledge of prior code generation failures into the prompt. Subsequent iterations of code generation are then more likely to avoid making the same mistakes.

3. **CCG as an LLM tool.** The third layer of control is established by letting the LLM operate within boundaries only. The boundary is that the LLM is only allowed to interact with CCG-generated templates as a tool, instead of freely outputting anything. This results in the LLM only instantiating the template instead of responding with arbitrary text. Unlike restrictions based on natural language input, this output-focused restriction is stricter and thus provides a greater degree of control.

   Restricting the LLM's output capabilities through tooling means that it can only output arbitrary text when instantiating templates that accept arbitrary strings in their parameters. This is only the case for substitution operations in CCG. Even then, the arbitrary output remains embedded in the template's surrounding context. This provides strong control over the degree of flexibility given to the LLM, even within the template.

   Code that is out of reach for manipulation by the LLM is guaranteed to be part of the output. This is the case for code that is not in an AST subtree to which CCG substitution operations have been mapped. This property can be exploited to include guaranteed code instrumentation in addition to the first generation step. This instrumentation allows for applying validation through AAL leading to the third layer of control.

6

Related Work

To the best of my knowledge, I am not aware of any holistic, Large Language Model (LLM)-powered approach to combining code generation and formal methods in Model-Driven Engineering (MDE), or rather, Language-Driven Engineering (LDE), but LLM-powered code generation itself is a very popular and vibrant topic. Due to its novelty, there are many papers and rapid developments in the field. This chapter covers research related to the ideas presented in this work. First, the chapter presents research on general LLM support and usage in code generation domains. Next, it covers approaches to combining LLM code generation and formal methods. Lastly, it covers work related to Code-centric Code Generation (CCG), partially about Template-based Code Generation (TBCG), and combining TBCG and LLMs.

6.1 LLM-powered Code Generation

In recent years, code generation with LLMs has become ubiquitous. Popular and widely used tools include Github Copilot [37], Cursor [77], Codex [59], and Claude Code [3]. While these tools offer more and more full automation, they still keep human users in the loop by providing code diffs and letting users accept changes individually. Sobania et al. [71] extensively used LLM support for bug fixing and repairing programs. They underline empirically that the performance of LLM-supported code generation is only appropriate when used as a programming partner, therefore, the human user should remain involved in the loop. Belzner et al. [10] used LLM support throughout the entire software development cycle of designing, generating code, and testing. They also discuss future ways of LLM usage in software development. Possible ways also include to cutting humans entirely from this loop. Tools like AutoGPT, on the other hand, aim to make LLMs increasingly autonomous [86]. These tools split complex tasks into smaller ones and use specialized prompts. This resembles the "divide-and-conquer" approach that the LDE LLM extension also uses. However, AutoGPT decomposes the problem space using natural language, while the LDE LLM extension does this via formal models, such as the defined Domain-Specific Languages (DSLs), and adds natural language on top.

Integrating LLMs into LDE establishes a feedback loop that provides feedback from formal tools to the LLM, enabling it to potentially correct itself and improve the performance and reliability of the code generation. More common approaches use a self-correcting loop in which LLMs review their own intermediate results. With Chain of Thought (CoT), Wei et al. [87] present a method for splitting tasks into intermediate steps and enable reviewing these steps. This is done solely on a natural language level, which increases LLMs' performance for complex tasks [33]. Tree of Thought (ToT) [91] extends this

approach. It adds additional intermediate steps to CoT and organizes them in a tree structure. Backtracking is used to find optimal solutions if one branch does not produce to the expected results. Skeleton of Thought (SoT) [58] first generates a rough structure of the target code, then refines the remaining parts step by step.

6.2 Combining LLM-powered Code Generation and Formal Methods

Formal methods are widely used in computer science to add trust in programs across different domains, particularly those with critical safety requirements, such as automotive, aviation, and medical domains. Formal methods are integrated into the development process during specification, verification or during development. Therefore, integrating formal methods into an LLM-supported code generation process is a logical next step. Many researchers have contributed to this in recent years.

Tihanyi et al. [81] use a loop with formal verification for continuous improvement. Counterexamples are fed back into the LLM until all checked properties are met.

Speculyzer, the specification synthesizer, is another approach to integrating formal methods into the development workflow [45]. The authors use LLMs to generate a set of candidate programs and candidate specifications to verify the generated programs. Each candidate program is tested against each candidate specification to calculate a “probability of correctness” score. Programs can then be suggested by their score, while those below a certain threshold are discarded.

The ideas of Clover [75], for Closed-Loop Verifiable Code Generation, are somewhat similar to those behind Speculyzer. Clover generates not only code and formal specifications but also natural language descriptions in the form of docstring. During the validation phase, they check if each artifact can be reconstructed from the other two artifacts, respectively. Only if this is the case is the result generated by the LLM considered accepted.

6.3 Template-based Code Generation

The CCG approach is an improvement on TBCG. It introduces an abstraction layer above the templates, which provides several benefits. For example, it makes the prototype code a central artifact of template development and maintenance. It also enables easy usage with LLMs by generating a schema for the template as a byproduct. These features make CCG unique. However, there are other approaches to code generation that at least loosely share some aspects.

Genesys [43] is an approach to code generation in a service-oriented way that offers graphical modeling components. It separates output descriptions and generator logic. Generator logic can be provided through either a rule-based generator or a template. Additionally, Genesys provides graphical languages for this purpose. Genesys aims to be agnostic to most existing approaches and provides good testing and verification capabilities. However, Genesys is an entirely new tool with a new DSL that users may need to learn first. CCG, on the other hand, resembles the workflow that developers are already familiar with and is close to the output code.

The idea of letting CCG map transformational operations onto the Abstract Syntax Tree (AST) was inspired by MPS's projectional editing [61]. In MPS, users add nodes to the AST, rather than writing text and parsing it later. Through projectional editing, the AST can be visualized in different ways. As a basis, it uses the same concrete syntax of text that the parser would expect. However, rather than making each character editable,

code parts are added en bloc for each AST node. Other, more graphical visualizations, like tables and similar, are also possible.

Repleo, by Arnoldus et. al [4], is a template engine that, like CCG, tries to always produce syntactically correct output. It achieves this by extending existing grammars with expressions for transformational operations. These operations can then be used to become part of the new language's AST. The result is a new grammar resembling the original target language grammar that cannot be parsed using the original target language parser anymore. CCG, on the other hand, stores transformational operations separately and only maps them onto the original output language's AST.

The reason CCG separates the transformational operations from the original output syntax is that it wants to keep it parsable. Approaches like Thymeleaf [78] do this as well with natural templates. Natural templates instrument the code with attributes that don't interfere with the ability to parse code, but add information about transformational operations. These operations can then be used to generate new code on basis of the instrumented original. Thymeleaf is available for HTML, XML, JavaScript, CSS, and text.

7 Conclusion

This dissertation presented a hybrid code generation approach in the context of Language-Driven Engineering (LDE). The approach offers new ways to integrate Large Language Models (LLMs) as code generators in LDE and, thus, Model-Driven Engineering (MDE) in a controllable manner while benefiting from the flexibility of LLMs. Furthermore, the approach enables easy validation of outputs. This makes it possible to check whether the LLM output matches the users' semantic expectations. For this approach, Code-centric Code Generation (CCG) was extended so that templates generated with it could easily be used by LLMs. The extended CCG was then incorporated into the two-step generation process of the LLM extension to LDE to maximize its potential.

In this approach, template instantiation is handled by the LLM rather than a designated part of the code generator. This renders tedious data extraction from models and transformations to bring it into the correct format for templates or other code generators unnecessary. Despite the new flexibility that this hybrid approach provides over conventional code generators, it establishes three layers of control for LLM usage:

1. **DSNLs** The first layer of control shifts the responsibility of good prompting from the users to the language developers. For this purpose, a Prompt Frame is generated in the two-step generation process. The Prompt Frame provides the LLM with the context for generating code. This limits the domain in which the LLM should work, while allowing it to link information from the user's prompt to modeled information. We coined this a Domain-Specific Natural Language (DSNL) for users in [21]. This layer controls the input for LLMs.

2. **Validation by design.** The second layer of control is established by the ability to easily infer behavioral automata using Active Automata Learning (AAL), which allows for system-level validation. Either conventional code generators in the first step of the two-step generation process or LLM-controlled CCG templates can be used to ensure code instrumentation in the output. This instrumentation can then be used to execute AAL and infer behavioral automata. These automata can be used to validate whether the output satisfies user expectations. Validation can be done visually or automatically via model checking. This process acts as a form of quality control for LLM outputs on a semantic level.

3. **Constraining LLM outputs.** The third layer emerges from constraining the LLM's output capabilities. Instead of allowing the LLM to generate arbitrary code, it only instantiates templates generated with CCG. With an accompanying schema of the template parameters, that includes semantic descriptions of the purpose of each parameter, LLMs can use CCG-generated templates without additional effort.

Consequently, LLMs can be used in a mode where they can only interact with the provided template as a tool instead of outputting anything else. With this output constraint, the LLM can only produce variations that the template and its operations allow. Therefore, parts of the templates that are not encapsulated in operations that allow manipulation are guaranteed to be included in the output. This guarantees important code properties, such as expected code instrumentation. This feature can be considered partial determinism for LLM code generation. Constraining the LLM's output controls the LLM during code generation.

These layers of control can increase the trustworthiness and controllability of LLM usage, making it a viable solution for LDE.

However, there are limitations and pitfalls to consider when using this hybrid code generation.

As with most application prompts or scope limitations through natural language instructions, DSNLs may be vulnerable to direct prompt injection [62]. This attack involves ignoring the context in which the LLM operates to break out of its boundaries. This might be negligible, though, as users would only harm themselves by breaking out of the DSNL context. Still, it should be kept in mind.

CCG adds a new abstraction layer on top of templates. This requires language developers to learn how to use CCG instead of the template engines they are already familiar with. Some language developers might avoid CCG because of this. However, using conventional templates with LLMs the way CCG allows would require extra work from the language developers. They would still have to provide schemas describing the templates' parameters, including natural language descriptions of the parameters' semantics. Conversely, CCG can generate these alongside the templates without additional effort. Nevertheless, the general CCG limitations still apply. For now, CCG has only been implemented for a couple of target languages. Supporting new target languages requires additional implementation work because as the prototypes need to be parsed. Templating engines, on the other hand, are usually target-agnostic and applicable to any target language.

The AAL tool Malwa, which is used in the examples of the presented approaches, only allows one to infer behavioral automata for HTML-based UIs. While this is widely used nowadays, even for desktop applications, the presented instrumentation is not a universal solution. If language developers want to maintain the AAL validation control layer of the approach, they must select AAL tools that suit their needs and are usable with instrumentation in a similar manner.

Overall, the presented hybrid code generation is a unique and versatile approach. Due to the nature of LDE, parts of the approach can be used combined or separately. This allows language developers to decide which aspects of the approach suit their needs best.

7.1 Future Work

The presented approaches offer a variety of advantages. Further research should explore additional possibilities and the limits of hybrid code generation. The following paragraphs present ideas for future work.

Partial Generation

When code is generated from models using conventional code generators, the necessary information must be present in specific forms. However, it is possible that the model will

be incomplete. This could be due to user oversight or other mistakes. In this case, users must be made aware of this with model validation.

If a code generator's input model is missing crucial information, the generator might not be able to produce code. Conversely, LLMs use informal natural language input for code generation. The level of detail of the input may vary, but LLMs will still try to produce output that satisfies the user's request.

When integrating LLM code generation into LDE, LLMs can fill in the gaps in incomplete input models for code generation. If the extended CCG for LLM usage is used in the first generation step, the LLM can comply with the provided template schema, regardless of the input model's completeness. The LLM could be given the entire model, providing access to all modeled information. The user's prompt could contain additional information about their intent. Based on these two sources, the LLM can infer missing data that complements the information the user has already provided. Alternatively, a dialogue could be established between the LLM and the user, requesting additional information if needed.

Then, the LLM can instantiate the template with data from the model and the prompt, as well as additional data that it inferred. It would also be interesting to investigate whether this new information could be fed back into the model and stored in it.

Template Orchestration

Code generators may need to output code for various targets. For example, this could be to satisfy different target architectures or to comply with other structural requirements. Templates that match the output requirements are usually determined using rule-based approaches with input model information or by letting users decide. LLMs could simplify this process by deciding which templates to execute for different targets on their own.

For this purpose, language developers could register several templates for the different target requirements. With tool calling or Model Context Protocol (MCP) capabilities, LLMs could then choose which templates to instantiate.

This could lead to more modularized, smaller templates that are orchestrated. In complex scenarios, templates may have dependencies. They could require outputs or orchestration with other specific templates in a higher-order fashion. In this case, LLMs should be enabled to query and select the available templates. This creates additional flexibility while maintaining controllability of LLM usage in LDE.

References

[1] Andrea Agostinelli, Timo I. Denk, Zalán Borsos, Jesse Engel, Mauro Verzetti, Antoine Caillon, Qingqing Huang, Aren Jansen, Adam Roberts, Marco Tagliasacchi, Matt Sharifi, Neil Zeghidour, and Christian Frank. "MusicLM: Generating Music From Text." In: *arXiv preprint arXiv:2301.11325* (2023). DOI: https://doi.org/10.48550/arXiv.2301.11325.

[2] Dana Angluin. "Learning Regular Sets from Queries and Counterexamples." In: *Information and Computation* 75.2 (1987), pp. 87–106. DOI: https://doi.org/10.1016/0890-5401(87)90052-6.

[4] Jeroen Arnoldus, Jeanot Bijpost, and Mark van den Brand. "Repleo: a syntax-safe template engine." In: *Proceedings of the 6th international conference on Generative programming and component engineering.* 2007, pp. 25–32. DOI: https://doi.org/10.1145/1289971.1289977.

[5] Alexander Bainczyk. "Simplicity-Oriented Lifelong Learning of Web Applications." PhD thesis. TU Dortmund University, 2023. DOI: http://doi.org/10.17877/DE290R-24274.

[6] Alexander Bainczyk, Steve Boßelmann, Marvin Krause, Marco Krumrey, Dominic Wirkner, and Bernhard Steffen. "Towards Continuous Quality Control in the Context of Language-Driven Engineering." In: *International Symposium on Leveraging Applications of Formal Methods.* Springer. 2022, pp. 389–406. DOI: https://doi.org/10.1007/978-3-031-19756-7_22.

[7] Alexander Bainczyk, Daniel Busch, Marco Krumrey, Daniel Sami Mitwalli, Jonas Schürmann, Joel Tagoukeng Dongmo, and Bernhard Steffen. "Cinco Cloud: A Holistic Approach for Web-Based Language-Driven Engineering." In: *International Symposium on Leveraging Applications of Formal Methods.* Springer. 2022, pp. 407–425. DOI: https://doi.org/10.1007/978-3-031-19756-7_23.

[8] Alexander Bainczyk, Alexander Schieweck, Bernhard Steffen, and Falk Howar. "Model-Based Testing Without Models: The TodoMVC case study." In: *ModelEd, TestEd, TrustEd.* Springer, 2017, pp. 125–144. DOI: https://doi.org/10.1007/978-3-319-68270-9_7.

[9] 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.* Springer, 2022, pp. 38–53. DOI: https://doi.org/10.1007/978-3-031-08166-8_3.

[10] Lenz Belzner, Thomas Gabor, and Martin Wirsing. "Large Language Model Assisted Software Engineering: Prospects, Challenges, and a Case Study." In: *International Conference on Bridging the Gap between AI and Reality.* Springer. 2023, pp. 355–374. DOI: https://doi.org/10.1007/978-3-031-46002-9_23.

[11] Moez Ben Hajhmida and Edward A Lee. "RAG and Agentic Assistant: A Combined Approach." In: *International Conference on Bridging the Gap between AI and Reality*. Springer. 2025, pp. 47–62. DOI: https://doi.org/10.1007/978-3-032-07132-3_4.

[12] Thorsten Berger, Markus Völter, Hans Peter Jensen, Taweesap Dangprasert, and Janet Siegmund. "Efficiency of projectional editing: A controlled experiment." In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 2016, pp. 763–774. DOI: https://doi.org/10.1145/2950290.2950315.

[13] Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. 2nd Edition. Packt Publishing, 2016. ISBN: 978-1-78646-496-5.

[14] Jean Bézivin. "On the unification power of models." In: *Software & Systems Modeling* 4.2 (2005), pp. 171–188. DOI: https://doi.org/10.1007/s10270-005-0079-0.

[15] Steve Boßelmann. "Evolution of Ecosystems for Language-Driven Engineering." PhD thesis. TU Dortmund University, 2023. DOI: http://doi.org/10.17877/DE290R-23218.

[16] Steve Boßelmann, Markus Frohme, Dawid Kopetzki, Michael Lybecait, Stefan Naujokat, Johannes Neubauer, Dominic Wirkner, Philip Zweihoff, and Bernhard Steffen. "DIME: A Programming-Less Modeling Environment for Web Applications." In: *International Symposium on Leveraging Applications of Formal Methods*. Springer. 2016, pp. 809–832. DOI: https://doi.org/10.1007/978-3-319-47169-3_60.

[17] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, Harsha Nori, Hamid Palangi, Marco Tulio Ribeiro, and Yi Zhang. "Sparks of artificial general intelligence: Early experiments with gpt-4." In: *arXiv preprint arXiv:2303.12712* (2023). DOI: https://doi.org/10.48550/arXiv.2303.12712.

[18] Daniel Busch. "Towards Code-centric Code Generators." In: *Electronic Communications of the EASST* 82 (2023). DOI: https://doi.org/10.14279/tuj.eceasst.82.1218.

[19] Daniel Busch, Alexander Bainczyk, Steven Smyth, and Bernhard Steffen. "LLM-based code generation and system migration in language-driven engineering." In: *International Journal on Software Tools for Technology Transfer* 27.1 (2025), pp. 137–147. DOI: https://doi.org/10.1007/s10009-025-00798-x.

[20] Daniel Busch, Alexander Bainczyk, and Bernhard Steffen. "Towards LLM-based System Migration in Language-Driven Engineering." In: *International Conference on Engineering of Computer-Based Systems*. Springer. 2023, pp. 191–200. DOI: https://doi.org/10.1007/978-3-031-49252-5_14.

[21] Daniel Busch, Gerrit Nolte, Alexander Bainczyk, and Bernhard Steffen. "ChatGPT in the Loop: A Natural Language Extension for Domain-Specific Modeling Languages." In: *International Conference on Bridging the Gap between AI and Reality*. Springer. 2023, pp. 375–390. DOI: https://doi.org/10.1007/978-3-031-46002-9_24.

[22] Daniel Busch, Steven Smyth, Tim Tegeler, and Bernhard Steffen. "Code-centric Code Generation." In: *International Symposium on Leveraging Applications of Formal Methods*. Springer. 2024, pp. 340–355. DOI: https://doi.org/10.1007/978-3-031-73709-1_21.

[23] Jordi Cabot. “Exploring the use of large language models in domain-specific language development: an experience report.” In: *Joint Proceedings of the STAF 2025 Workshops*. CEUR. 2025.

[24] Jordi Cabot. “Positioning of the low-code movement within the field of model-driven engineering.” In: *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. 2020, pp. 1–3. DOI: `https://doi.org/10.1145/3417990.3420210`.

[25] 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. “Evaluating Large Language Models Trained on Code.” In: *arXiv preprint arXiv:2107.03374* (2021). DOI: `https://doi.org/10.48550/arXiv.2107.03374`.

[26] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. “Teaching Large Language Models to Self-Debug.” In: *arXiv preprint arXiv:2304.05128* (2023). DOI: `https://doi.org/10.48550/arXiv.2304.05128`.

[27] Yoonsik Cheon. “LLMs as Code Generators for Model-Driven Development.” In: *Proceedings of the 20th International Conference on Software Technologies ICSOFT*. Vol. 1. 2025, pp. 386–393. DOI: `https://doi.org/10.5220/0013580300003964`.

[28] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. “PaLM: Scaling Language Modeling with Pathways.” In: *Journal of Machine Learning Research* 24 (2023), pp. 1–113.

[29] Benoit Combemale, Jeff Gray, and Bernhard Rumpe. “Large language models as an “operating” system for software and systems modeling.” In: *Software and Systems Modeling* 22.5 (2023), pp. 1391–1392. DOI: `https://doi.org/10.1007/s10270-023-01126-0`.

[30] Alberto Rodrigues Da Silva. “Model-driven engineering: A survey supported by the unified conceptual model.” In: *Computer languages, systems & structures* 43 (2015), pp. 139–155. DOI: https://doi.org/10.1016/j.cl.2015.06.001.

[31] Juri Di Rocco, Davide Di Ruscio, Claudio Di Sipio, Phuong T Nguyen, and Riccardo Rubei. “On the use of large language models in model-driven engineering.” In: *Software and Systems Modeling* 24 (2025), pp. 923–948. DOI: https://doi.org/10.1007/s10270-025-01263-8.

[32] Davide Di Ruscio, Dimitris Kolovos, Juan de Lara, Alfonso Pierantonio, Massimo Tisi, and Manuel Wimmer. “Low-code development and model-driven engineering: Two sides of the same coin?” In: *Software and Systems Modeling* 21.2 (2022), pp. 437–446. DOI: https://doi.org/10.1007/s10270-021-00970-2.

[33] Guhao Feng, Bohang Zhang, Yuntian Gu, Haotian Ye, Di He, and Liwei Wang. “Towards Revealing the Mystery behind Chain of Thought: A Theoretical Perspective.” In: *Advances in Neural Information Processing Systems*. Vol. 36. 2023, pp. 70757–70798.

[34] Martin Fowler. *Domain-specific languages*. Pearson Education, 2010. ISBN: 0321712943.

[35] Robert France and Bernhard Rumpe. “Model-driven Development of Complex Software: A Research Roadmap.” In: *Future of Software Engineering (FOSE’07)*. IEEE. 2007, pp. 37–54. DOI: https://doi.org/10.1109/FOSE.2007.14.

[38] Frederik Gossen, Tiziana Margaria, Alnis Murtovi, Stefan Naujokat, and Bernhard Steffen. “DSLs for decision services: a tutorial introduction to language-driven engineering.” In: *International Symposium on Leveraging Applications of Formal Methods*. Springer. 2018, pp. 546–564. DOI: https://doi.org/10.1007/978-3-030-03418-4_33.

[39] Hardi Hungar, Tiziana Margaria, and Bernhard Steffen. “Test-based model generation for legacy systems.” In: *International Test Conference*. Vol. 2. IEEE. 2003, pp. 150–159. DOI: https://doi.org/10.1109/TEST.2003.1271205.

[40] John Hutchinson, Mark Rouncefield, and Jon Whittle. “Model-driven engineering practices in industry.” In: *Proceedings of the 33rd International Conference on Software Engineering*. 2011, pp. 633–642. DOI: https://doi.org/10.1145/1985793.1985882.

[41] John Hutchinson, Jon Whittle, and Mark Rouncefield. “Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure.” In: *Science of Computer Programming* 89 (2014), pp. 144–161. DOI: https://doi.org/10.1016/j.scico.2013.03.017.

[42] Bengt Jonsson. “Learning of Automata Models Extended with Data.” In: *International School on Formal Methods for the Design of Computer, Communication and Software Systems*. Springer, 2011, pp. 327–349. DOI: https://doi.org/10.1007/978-3-642-21455-4_10.

[43] Sven Jörges. *Construction and Evolution of Code Generators: A Model-Driven and Service-Oriented Approach*. Springer, 2013. DOI: https://doi.org/10.1007/978-3-642-36127-2.

[45] Darren Key, Wen-Ding Li, and Kevin Ellis. “Toward trustworthy neural program synthesis.” In: *arXiv preprint arXiv:2210.00848* (2023). DOI: https://doi.org/10.48550/arXiv.2210.00848.

[46] Marco Krumrey, Alexander Bainczyk, Falk Howar, and Bernhard Steffen. "Malwa: learnability by design." In: *Principles of Verification: Cycling the Probabilistic Landscape: Essays Dedicated to Joost-Pieter Katoen on the Occasion of His 60th Birthday, Part III*. Springer, 2024, pp. 66–88. DOI: https://doi.org/10.1007/978-3-031-75778-5_4.

[47] Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. "CodeRL: Mastering Code Generation through Pretrained Models and Deep Reinforcement Learning." In: *Advances in Neural Information Processing Systems* 35 (2022), pp. 21314–21328.

[48] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks." In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 9459–9474.

[49] Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, and Neel Sundaresan. "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*. Association for Computing Machinery, 2022, pp. 1035–1047. DOI: https://doi.org/10.1145/3540250.3549081.

[50] Zhijie Liu, Yutian Tang, Xiapu Luo, Yuming Zhou, and Liang Feng Zhang. "No Need to Lift a Finger Anymore? Assessing the Quality of Code Generation by ChatGPT." In: *IEEE Transactions on Software Engineering* 50.6 (2024), pp. 1548–1584. DOI: https://doi.org/10.1109/TSE.2024.3392499.

[51] Michael Lybecait, Dawid Kopetzki, Philip Zweihoff, Annika Fuhge, Stefan Naujokat, and Bernhard Steffen. "A Tutorial Introduction to Graphical Modeling and Metamodeling with CINCO." In: *International Symposium on Leveraging Applications of Formal Methods*. Springer. 2018, pp. 519–538. DOI: https://doi.org/10.1007/978-3-030-03418-4_31.

[52] Tiziana Margaria and Bernhard Steffen. "Business process modeling in the jABC: the one-thing approach." In: *Handbook of research on business process modeling*. IGI Global Scientific Publishing, 2009, pp. 1–26. DOI: https://doi.org/10.4018/978-1-60566-288-6.ch001.

[53] Tiziana Margaria and Bernhard Steffen. "Service-orientation: conquering complexity with XMDD." In: *Conquering complexity*. Springer, 2012, pp. 217–236. DOI: https://doi.org/10.1007/978-1-4471-2297-5_10.

[54] Marjan Mernik, Jan Heering, and Anthony M Sloane. "When and how to develop domain-specific languages." In: *ACM computing surveys (CSUR)* 37.4 (2005), pp. 316–344. DOI: https://doi.org/10.1145/1118890.1118892.

[55] Stefan Naujokat, Michael Lybecait, Dawid Kopetzki, and Bernhard Steffen. "CINCO: a simplicity-driven approach to full generation of domain-specific graphical modeling tools." In: *International Journal on Software Tools for Technology Transfer* 20.3 (2018), pp. 327–354. DOI: https://doi.org/10.1007/s10009-017-0453-6.

[56] Lukas Netz, Judith Michael, and Bernhard Rumpe. "From Natural Language to Web Applications: Using Large Language Models for Model-Driven Software Engineering." In: *Modellierung 2024*. Gesellschaft für Informatik. 2024, pp. 179–195.

[57] 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: `https://doi.org/10.1007/s10009-014-0321-6`.

[58] Xuefei Ning, Zinan Lin, Zixuan Zhou, Zifu Wang, Huazhong Yang, and Yu Wang. "Skeleton-of-thought: Prompting llms for efficient parallel generation." In: *arXiv preprint arXiv:2307.15337* (2023). DOI: `https://doi.org/10.48550/arXiv.2307.15337`.

[60] OpenAI et al. "GPT-4 Technical Report." In: *arXiv preprint arXiv:2303.08774* (2023). DOI: `https://doi.org/10.48550/arXiv.2303.08774`.

[61] Vaclav Pech, Alex Shatalin, and Markus Voelter. "JetBrains MPS as a tool for extending Java." In: *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. 2013, pp. 165–168. DOI: `https://doi.org/10.1145/2500828.2500846`.

[62] Fábio Perez and Ian Ribeiro. "Ignore Previous Prompt: Attack Techniques For Language Models." In: *arXiv preprint arXiv:2211.09527* (2022). DOI: `https://doi.org/10.48550/arXiv.2211.09527`.

[64] Harald Raffelt, Maik Merten, Bernhard Steffen, and Tiziana Margaria. "Dynamic testing via automata learning." In: *International Journal on Software Tools for Technology Transfe* 11 (2009), pp. 307–324. DOI: `https://doi.org/10.1007/s10009-009-0120-7`.

[65] Harald Raffelt, Bernhard Steffen, and Tiziana Margaria. "Dynamic Testing Via Automata Learning." In: *Haifa Verification Conference*. Springer. 2007, pp. 136–152. DOI: `https://doi.org/10.1007/978-3-540-77966-7_13`.

[66] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. "High-Resolution Image Synthesis With Latent Diffusion Models." In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2022, pp. 10684–10695.

[67] Apurvanand Sahay, Arsene Indamutsa, Davide Di Ruscio, and Alfonso Pierantonio. "Supporting the understanding and comparison of low-code development platforms." In: *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE. 2020, pp. 171–178. DOI: `https://doi.org/10.1109/SEAA51224.2020.00036`.

[68] Douglas C Schmidt. "Guest Editor's Introduction: Model-Driven Engineering." In: *Computer* 39.02 (2006), pp. 25–31. DOI: `https://doi.org/10.1109/MC.2006.58`.

[69] Hanan Siala and Kevin Lano. "A comparison of large language models and model-driven reverse engineering for reverse engineering." In: *Frontiers in Computer Science* 7 (2025). DOI: `https://doi.org/10.3389/fcomp.2025.1516410`.

[70] Hanan A Siala and Kevin Lano. "Towards Using LLMs in the Reverse Engineering of Software Systems to Object Constraint Language." In: *2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2025, pp. 1–6. DOI: `https://doi.org/10.1109/SANER64311.2025.00096`.

[71] Dominik Sobania, Martin Briesch, Carol Hanna, and Justyna Petke. "An Analysis of the Automatic Bug Fixing Performance of ChatGPT." In: *2023 IEEE/ACM International Workshop on Automated Program Repair (APR)*. IEEE. 2023, pp. 23–30. DOI: `https://doi.org/10.1109/APR59189.2023.00012`.

[72] Da Song, Zijie Zhou, Zhijie Wang, Yuheng Huang, Shengmai Chen, Bonan Kou, Lei Ma, and Tianyi Zhang. "An Empirical Study of Code Generation Errors made by Large Language Models." In: *7th Annual Symposium on Machine Programming*. 2023.

[73] Jonathan Sprinkle and Gabor Karsai. "A domain-specific visual language for domain model evolution." In: *Journal of Visual Languages & Computing* 15.3-4 (2004), pp. 291–307. DOI: https://doi.org/10.1016/j.jvlc.2004.01.006.

[74] Bernhard Steffen, Frederik Gossen, Stefan Naujokat, and Tiziana Margaria. "Language-Driven Engineering: From General-Purpose to Purpose-Specific Languages." In: *Computing and Software Science: State of the Art and Perspectives*. Springer, 2019, pp. 311–344. DOI: https://doi.org/10.1007/978-3-319-91908-9_17.

[75] Chuyue Sun, Ying Sheng, Oded Padon, and Clark Barrett. "Clover: Closed-Loop Verifiable Code Generation." In: *International Symposium on AI Verification*. Springer. 2024, pp. 134–155. DOI: https://doi.org/10.1007/978-3-031-65112-0_7.

[76] Eugene Syriani, Lechanceux Luhunu, and Houari Sahraoui. "Systematic mapping study of template-based code generation." In: *Computer Languages, Systems & Structures* 52 (2018), pp. 43–62. DOI: https://doi.org/10.1016/j.cl.2017.11.003.

[80] Haoye Tian, Weiqi Lu, Tsz On Li, Xunzhu Tang, Shing-Chi Cheung, Jacques Klein, and Tegawendé F Bissyandé. "Is ChatGPT the Ultimate Programming Assistant – How far is it?" In: *arXiv preprint arXiv:2304.11938* (2023). DOI: https://doi.org/10.48550/arXiv.2304.11938.

[81] Norbert Tihanyi, Yiannis Charalambous, Ridhi Jain, Mohamed Amine Ferrag, and Lucas C Cordeiro. "A New Era in Software Security: Towards Self-Healing Software via Large Language Models and Formal Verification." In: *2025 IEEE/ACM International Conference on Automation of Software Test (AST)*. IEEE. 2025, pp. 136–147. DOI: https://doi.org/10.1109/AST66626.2025.00020.

[82] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. "LLaMA: Open and Efficient Foundation Language Models." In: *arXiv preprint arXiv:2302.13971* (2023). DOI: https://doi.org/10.48550/arXiv.2302.13971.

[83] Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. "Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models." In: *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems*. Association for Computing Machinery, 2022, pp. 1–7. DOI: https://doi.org/10.1145/3491101.3519665.

[84] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. "Attention is all you need." In: *Advances in Neural Information Processing Systems*. Vol. 30. 2017.

[85] Markus Voelter, Janet Siegmund, Thorsten Berger, and Bernd Kolb. "Towards user-friendly projectional editors." In: *International Conference on Software Language Engineering*. Springer. 2014, pp. 41–61. DOI: https://doi.org/10.1007/978-3-319-11245-9_3.

[86] Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, Wayne Xin Zhao, Zhewei Wei, and Jirong Wen. "A survey on large language model based autonomous agents." In: *Frontiers of Computer Science* 18.6 (2024). DOI: https://doi.org/10.1007/s11704-024-40231-1.

[87] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter brian, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models." In: *Advances in Neural Information Processing Systems*. Vol. 35. 2022, pp. 24824–24837.

[88] Stephan Windmüller, Johannes Neubauer, Bernhard Steffen, Falk Howar, and Oliver Bauer. "Active continuous quality control." In: *Proceedings of the 16th International ACM Sigsoft symposium on Component-based software engineering*. 2013, pp. 111–120. DOI: https://doi.org/10.1145/2465449.2465469.

[89] Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. "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. DOI: https://doi.org/10.1145/3520312.3534862.

[90] Qiaomu Xue and Kevin Lano. "Comparing LLM-based and MDE-based code generation for agile MDE." In: *Joint Proceedings of the STAF 2025 Workshops*. CEUR. 2025.

[91] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. "Tree of Thoughts: Deliberate Problem Solving with Large Language Models." In: *Advances in Neural Information Processing Systems*. Vol. 36. 2023, pp. 11809–11822.

[92] Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. "RepoCoder: Repository-Level Code Completion Through Iterative Retrieval and Generation." In: *arXiv preprint arXiv:2303.12570* (2023). DOI: https://doi.org/10.48550/arXiv.2303.12570.

[93] Philip Zweihoff. "Aligned and collaborative language-driven engineering." PhD thesis. TU Dortmund University, 2022. DOI: http://doi.org/10.17877/DE290R-22594.

[94] Philip Zweihoff and Bernhard Steffen. "Pyrus: An Online Modeling Environment for No-Code Data-Analytics Service Composition." In: *International Symposium on Leveraging Applications of Formal Methods*. Springer. 2021, pp. 18–40. DOI: https://doi.org/10.1007/978-3-030-89159-6_2.

Online References

[3] Anthropic. *Claude 3.7 Sonnet and Claude Code*. Last Accessed: 2025-12-19. Anthropic Website, 2025. URL: https://www.anthropic.com/news/claude-3-7-sonnet.

[36] FreeMarker. *FreeMarker Website*. Last Accessed: 2026-01-09. The Apache Software Foundation, 2026. URL: https://freemarker.apache.org.

[37] Nat Friedman. *Introducing GitHub Copilot: your AI pair programmer*. Last Accessed: 2025-12-19. GitHub Blog, 2021. URL: https://github.blog/news-insights/product-news/introducing-github-copilot-ai-pair-programmer.

[44] Yehuda Katz. *Handlebars Website*. Last Accessed: 2026-01-09. 2026. URL: https://handlebarsjs.com.

[59] OpenAI. *Introducing Codex*. Last Accessed: 2025-12-19. OpenAI Website, 2025. URL: https://openai.com/index/introducing-codex.

[63] Velocity Project. *The Apache Velocity Project Website*. Last Accessed: 2026-01-09. The Apache Software Foundation, 2026. URL: https://velocity.apache.org.

[77] Cursor Team. *Introducing Cursor 2.0 and Composer*. Last Accessed: 2025-12-19. Cursor Website, 2025. URL: https://cursor.com/blog/2-0.

[78] Thymeleaf. *Thymeleaf 3.1*. Last Accessed: 2025-12-21. Thymeleaf Documentation, 2024. URL: https://www.thymeleaf.org/doc/tutorials/3.1/usingthymeleaf.html.

[79] Thymeleaf. *Thymeleaf Website*. Last Accessed: 2026-01-09. 2026. URL: https://www.thymeleaf.org/.