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

Daniel Busch[1], Gerrit Nolte[1], Alexander Bainczyk[1], and Bernhard Steffen[1]

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

Abstract. This paper presents an approach to no-code development based on the interplay of *formally defined* (graphical) Domain-Specific Languages and *informal, intuitive* Natural Language which is enriched with contextual information to enable referencing of formally defined entities. The paper focuses on the use and automated integration of these enriched intuitive languages via ChatGPT-based code generation to exploit the best of both language paradigms for domain-specific application development. To compensate for the lack of control over the intuitive languages we apply automated system-level validation via automata learning and subsequent model checking. All this is illustrated using the development of point-and-click adventures as a minimal viable example.

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

1 Motivation and Introduction

GitHub Copilot [16] and ChatGPT [28] have opened a totally new programming experience: Code for diverse programming languages can be generated from just a few lines of natural language descriptions, and even simple programs can be generated fully automatically in a similar fashion. In particular, Large Language Model (LLM)-based programming exceeds what would have been expected in the past by far. On the other side, generated code of machine learning models may have surprisingly severe mistakes. Thus, LLM-based programming is bound to be supervised.

In this paper, we present an approach to software development based on the interplay of Domain-Specific Languages (DSLs) and Natural Language (NL) descriptions. Conceptually, our approach is an extension of Language-Driven Engineering (LDE)[15] to allow natural language specifications for process requirements. We call a natural language which is contextualized with additional information about the domain models a Domain-Specific Natural Language (DSNL).

This special kind of DSLs ensures that users do not need to specify implementation details, but all necessary information for LLM-based code generation is contained in the additional context. The role of these languages is sketched in Figure 1 which we explain along the description of our concept in Section 3. We illustrate this concept in Section 4 using the example of the river crossing puzzle[3]. In this puzzle a farmer is confronted with the problem to cross the river with a wolf, a goat, and a cabbage in a boat that is so small that it can only take one of the three items at a time. In order to avoid damage, the farmer must make sure that neither the wolf and the goat, nor the goat and the cabbage are on the same side of the river whilst the farmer is on the other side.

Our accompanying example concerns the automatic generation of a point-and-click adventure which is won exactly when all the items are safely transferred from one side of the river to the other. The landscape of the point-and-click adventure can easily be specified graphically. For our puzzle we only have to draw the two sides of the river and mutual connections for modeling the potential boat transfer as shown in Figure 2. This is enough to generate

- a Prompt Frame that provides contextual information to the NL description, so that it can be used as a DSNL, and
- the code that is meant to be extended by the LLM via code generation and code merging.

The example used in this paper focuses on the specification of the puzzle's constraints in natural language, enriched to talk about the objects of the graphical model. These specifications are fed into the Prompt Frame to generate code that extends the code generated from the graphical model. The resulting (merged) code can then be deployed to a fully running web application.

Our approach relies heavily on natural language descriptions with all its ambiguities. This reduces the number of properties which can be validated by the inherent properties of DSLs alone. As a result, we verify them via black box checking [6] of the overall system. Concretely, we combine active automata learning [2, 4, 5] to infer behavioral models of generated applications via testing and model checking to verify runtime properties formulated in temporal logic. To achieve a full no-code solution, our generators are implemented to generate instrumented web applications that are *learnable by design* [21]. Only the model checker requires manual input in terms of the property to be verified. In our example, we consider the property that the farmer can succeed to cross the river with all three 'items' while assuring that no damage occurs.

It should be noted that our solution is fully no-code and only requires domain experts to:

1. graphically model the architectural information,
2. to specify the process logic in natural language, and
3. to observe the feedback of automated model verification.

All of this can be done via an accessible web application which we will make publicly available for experimentation.

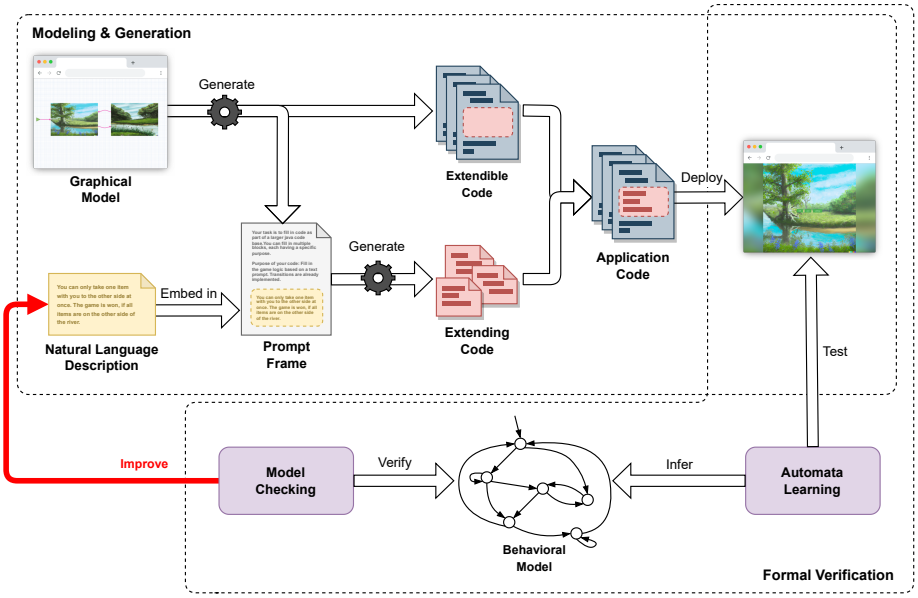

Fig. 1. Concept overview.

Outline. In Section 2, we introduce the three fundamental topics of this paper, LDE, LLMs in the context of programming, and learning-based testing. From there, we introduce the concept of utilizing natural language in a domain-specific environment in Section 3, and present implementation details in the context of an exemplary implementation in Section 4. Section 5 discusses the potential and drawbacks of the presented approach and also compares our contribution to related work. Finally, Section 6 concludes the paper, giving an overview, reflecting on the presented approach, and providing ideas for future work.

2 Preliminaries

DSLs and Natural Language Processing (NLP) are two very different approaches to interacting with machines. Each of these approaches has advantages and disadvantages. However, both their advantages and their disadvantages seem to be complementary. This is one of the main reasons why our approach attempts to merge the two paradigms to create one that benefits from the advantages of both while trying to overcome their disadvantages.

This section provides a brief overview of DSLs and NLP and describes their advantages and disadvantages. In addition, learning-based testing is presented as a means to enable a validation feedback loop for users using DSNLs in a no-code environment.

2.1 Language-driven Engineering

LDE is a paradigm that brings DSLs into focus for solving problems in specific domains [15, 17, 24]. It aims to provide multiple DSLs for each subproblem or stakeholder within a domain. This is done by decomposing potential solutions into independent DSLs that still have a high degree of interplay with each other to provide powerful tools that can be used to solve problems with ease.

The approach in this paper also follows an LDE approach. But instead of decomposing a single DSL into multiple DSLs, this paper's approach uses natural language within a domain-infused context. This concept of decomposition is further elaborated in Section 3.2

2.2 LLMs & Programming

Perhaps most famously brought to mainstream attention by ChatGPT, LLMs such as LLaMa [32], GPT1 through GPT4 [25], PaLM [18] etc. have led to rapid progress in natural language processing. Most commonly, LLMs are used as generative models, taking in a natural language description and returning some form of output, such as text (ChatGPT), images (Stable Diffusion [19]), audio (MusicLM [20]) or, most importantly for this work, code.

Modern LLMs have shown increasingly promising performances with regards to code generation, with model performance steadily increasing. As an example, GPT-4 [28] reached a score of 67% on the Human-Eval dataset [16] where GPT-3.5 only reached 48.1%.

2.3 Learning-Based Testing

Learning-based testing [7] is an approach to fully automated testing of black-box systems using *Active Automata Learning* [2]. In this context, *active learning* refers to a process in which a so-called *learner* interacts directly with a System Under Learning (SUL) via its public interfaces. By posing automatically constructed test queries over some input alphabet and recording the reactions of the SUL, automata models representing the system behavior can be inferred. Often, learning is combined with model checking to automatically verify system properties on inferred models. The practice of testing web applications via learning-based testing also has a long history in research [8, 9, 11, 12, 17], where Mealy machines proved to be an adequate representation for verifying user-level specific system properties.

Learning web applications[1] typically requires users to specify an input alphabet and implement system-specific *mappers* [10] that provide an interface for learning algorithms to interact with the target SUL. To address this problem, [21] introduces the *learnability-by-design* framework, which includes the instrumentation DSL iHTML. The language enables developers to instrument HTML

[1] In this work, the term *learning* refers to the use of active automata learning and is not related to the field of deep learning or other AI-based approaches.

code in a way that allows learning algorithms to incrementally mine behavioral models simply by analyzing the Document Object Model (DOM) of the website, without explicitly specifying an input alphabet.

3 Concept

In this section, we introduce a method for code generation that produces code from combined DSL and natural language descriptions. The core concept of this approach is twofold:

1. providing additional domain information to the LLM to generate code that satisfies specific requirements so that it can be used contextually, and
2. extending code that has been generated by traditional code generation.

With reference to Figure 1, the user creates the graphical model and automatically generates the extensible code and the prompt frame from it. Next, they can use NL to describe additional program logic that is embedded in the prompt frame so that the LLM can generate code from it. This code is then automatically merged into the previously generated extensible code of the graphical model, which together form the resulting application code. Automatic validation is performed on the running instance of the application, which outputs an automaton of accessible system states and can be used to provide feedback to the user using model checking on this automaton.

3.1 Goals

Code generation from formally defined DSLs is well-understood and scalable, in contrast to code generation with LLMs which, at least today, lacks

Control. In general it is hard to predict whether the generated code will be syntactically correct, let alone that it solves the intended task.

Scalability. Feasible problem descriptions are strongly limited in size and conceptual complexity.

To deal with these weaknesses, our approach is designed to only require the generation of small, clearly defined code blocks from natural language that are as independent as possible from the rest of the code-base (which is generated in a structured, rigorous fashion from a DSL model). Of course, we cannot guarantee semantic correctness even for these small code blocks, but we can assure that their aspect-oriented integration maintains executability. This allows us to validate runtime properties of the resulting applications using black-box checking.

3.2 Language Integration

Our approach is characterized by decomposing the overall specification into a formal and an intuitive part, both supported by dedicated DSLs. Such a decomposition is typical for LDE, but the required language integration is new.

It requires special care to let ChatGPT generate code that is ready for aspect-oriented merging with code generated from the graphical model. It is the role of the *Prompt Frame* to guarantee that the code generated by the LLM fits the code generated by the graphical DSLs. In particular, it must use the same programming language and utilize the desired functions and global variables.

3.3 Contextualization

To actually use the formal properties of DSLs in an LLM environment, the LLM needs further information about the context for which it should generate code. Thus, the actual prompt must be primed with information about its expected behavior, constraints and details about the graphically modeled instances, and about the expected output code. We call this priming information, into which the user prompt will later be embedded, the Prompt Frame.

Since the LLM's output code is expected to extend the code generated by the DSLs, the LLM is also provided with code stubs and descriptive comments. As a consequence, the Prompt Frame also contains this information about the expected output structure. We call this sub-frame, into which the LLM fills its code, the Generation Frame. The example in Section 4 takes advantage of aspect-oriented extensions by using this ensured LLM output structure.

The following paragraphs describe the Prompt Frame and its Generation Frame part in more detail, including their goals and overall purpose.

Prompt Frame. The Prompt Frame should serve two purposes: (1) to provide the LLM with information about the concrete model instances of the DSLs, and (2) to communicate requirements for the generated code that go beyond the general structure provided by the Generation Frame.

The information provided to the LLM should include, for example, information about available states and global variables, as exemplified in Listing 1.1. Any entity that has been modeled in an accompanying DSL and that should be able to interact with the logic prompts fed into the LLM should be mentioned in the prompt frame. Other information may include the desired programming language, or that the LLM should output only code and no supporting text.

Generation Frame. The Generation Frame is a part of the Prompt Frame and consists of code parts that the LLM should fill-in, as seen in Listing 1.2. It is used to ensure that the LLM outputs exactly the functions with the correct signatures that are needed by the DSLs' generated code, which enables the code generated by the DSLs to safely call the functions implemented by the LLM. This allows for easy extensions of the underlying program in an aspect-orientated fashion.

Information about which code is expected in each function is provided through comments or as additional descriptions in the outer Prompt Frame. Besides, LLMs are also able to extract semantic information from the names and signatures of functions and fill in code accordingly.

3.4 Validation & Feedback

In the context of this work, we leverage iHTML in our approach by embedding learnability-by-design practices into our manually implemented code generator, which generates the code frame for the LLM-based generator from our DSL. As a result, arbitrarily generated applications are automatically learnable, allowing us to infer behavioral models that represent user-level interaction processes. For verification purposes, desired system properties can be formulated in temporal logic, Computation Tree Logic (CTL) [1] in our case, to automatically verify learned models with a model checker. Feedback from the model checker is used to refine the natural language prompt, as discussed in Section 4.5.

4 Example Implementation

This section shows a sample implementation of the concept presented in this paper. For this, the point-and-click adventure mentioned before is implemented. In our scenario it consists of a puzzle with two locations: the left and the right side of a river. At the beginning there are three objects on the left side of the river: a wolf, a goat and a cabbage. The game is *won* when all three items have been moved to the right side in a fashion satisfying the following constraints: The player can only take one item at a time and at no time the wolf and the goat or the goat and the cabbage are on the same side of the river while the player is on the other side. To implement this puzzle, we chose the graphical DSL Webstory[13], created with the IME workbench Cinco[14].

As the title of this paper suggests, the LLM used for this example is ChatGPT in its API version "gpt-3.5-turbo". Conceptually, this choice is not important, but the use of other LLMs may well lead to different results.

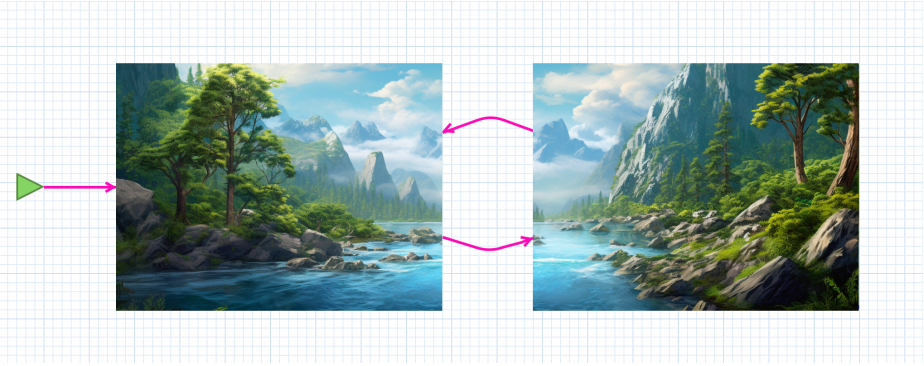

Fig. 2. Examplary WebstoryGM model on Cinco product level.

4.1 Model Decomposition

When creating a point-and-click adventure, two aspects are most important: (1) the design of each game screen and its reachability, and (2) the game logic about when the game is won or lost, as well as the existence and behavior of game objects. Following our approach, we decompose our example into *WebstoryGM*, a graphical DSL to model game screens and transitions between them and *WebstoryNL*, the domain-specific natural language to express the game logic:

- WebstoryGM is suitable to visually specify a high level overview of the game with game screens as images that are actually used in the game, and arrows to model their connections.
- WebstoryNL allows to describe the win and lose conditions using natural language without requiring any form of formalization.

4.2 Graphical Modelling

In its original version, Webstory includes three basic elements for modeling point-and-click adventures: screens, click areas, and transitions. Using these three types of elements, users can define images for game screens and where to click to trigger transitions to different screens. In addition, the original version of Webstory allows users to graphically model game logic. This can be done using graphical representations for variables and adding guards to transitions to change the behavior of screen transitions depending on the overall state of the game. However, implementing game logic using the graphical method is rather tedious, and its semantics cannot be immediately deduced by just looking at the model.

As described in Section 4.1, WebstoryGM should only be used to create a sitemap-like overview of all available game screens and their general reachability for this example. Thus, the original Webstory has been modified to contain only screens and transitions, resulting in the desired WebstoryGM. These two elements allow the user to create such a sitemap from which all the information needed to generate the base code, the Prompt Frame and the Generation Frame can be derived. In addition, game screens for a losing and a winning state have been added. These screens don't need to be modeled by the user, as they are implicitly accessible when the player reaches a winning or losing state, as specified by the natural language description of the game logic.

Figure 2 shows an example model for the river crossing puzzle. It consists only of the left and right sides of the river, starting on the left side.

4.3 Prompt Frame & Generation Frame

The game's logic should be described and generated using natural language and ChatGPT as an LLM. To be able to extend the underlying game modelled using the WebstoryNL as described earlier in Section 3 the Prompt Frame, as seen in Listing 1.1, contains the following information:

- names of the available game screens

```
Your task is to fill in code as part of a larger JavaScript code base.

You can fill in multiple blocks, each having a specific purpose.

Larger context for code: a point-and-click adventure with state
    transitions. All code that you write is part of a game with
    attributes states=["leftRiver", "rightRiver"] and currentState
    which holds the currently visited river side and is therefore one
    of either values of states. These attributes are already written
    and you can safely assume that the rest of the code works as
    intended.

Purpose of your code: Fill in the game logic based on a text prompt.
    Game objects beside states and currentState should be objects
    having a name, and potentially multiple transition objects that
    contain a screen property which is the name the transition can be
    triggered on, and a function property which is the transition
    function for this screen. Game objects are considered present in a
    state if they possess the currentScreen property of the state.

The code blocks for you to implement:

// [...], see Listing 2

Prompt: // [...], see Listing 3

Answer as follows: Write down ONLY the filled in code blocks with the
    code that you seem fit. Add comments if you want but do NOT explain
    anything about the code, your answer should ONLY contain javascript
    code.
```

Listing 1.1. Excerpt of the generated Prompt Frame.

- the desired programming language; in this case JavaScript
- a general directive that should be concise and produce nothing more than the desired code, so it can be easily merged with the code of the graphical models
- expected general properties like the already implemented transitions that are derived from the game objects the LLM should provide, or the general scenario of a point-and-click adventure

As another part of the Prompt Frame, the LLM is provided with code stubs, as illustrated in Listing 1.2, it has to fill-in as part of the Generator Frame. This Generator Frame comprises

- the function `initVariables()` for game objects, `checkWin()` and `checkLoss()` for implementations of the winning and losing conditions
- Comments with further information about the expected game objects or to mark the positions where the LLM-generated code has to be inserted.

The Prompt Frame (including the Generation Frame) is sufficient to prime ChatGPT in a way that it generates code that is ready to be merged into the code generated from the WebstoryGM model.

```
function initVariables() {
  // state objects should be of the following form
  // this.gameObjects = [
  //   {
  //     name: 'someName',
  //     currentScreen: 'someState',
  //     transitions: [
  //       {
  //         screen: 'someState',
  //         function: () => ()
  //       }
  //     ],
  //   }
  // ]
  // they can possess multiple transitions and are only
  // rendered on screens they have transitions for

  this.gameObjects = []; // fill in
}

function checkWin() { // fill in }

function checkLoss() { // fill in }
```

Listing 1.2. The example Generation Frame.

```
On the left side of the river there are a wolf, a goat, and a cabbage.
The game is won if every object has been brought to the right side of
    the river.
The game is lost if the wolf and the goat are on the same side of the
    river, while the player is on the other side, or if the cabbage and
    the goat are on the same side of the river while the player is on
    the other side.
```

Listing 1.3. Example for a natural language description.

4.4 Resulting Web Application

Once the user has modeled the WebstoryGM model as in Figure 2 and generated the extensible code and prompt frame from it, the DSNL prompt can be added to state the desired game logic.

An example prompt to let ChatGPT implement the game logic of the river crossing example can be seen in Listing 1.3. Once the prompt has been inserted into the prompt frame, it can be sent to ChatGPT either through its API or through the web interface. ChatGPT will only respond with the desired JavaScript code which is then merged into the base code generated from the graphical model. Two screens of the resulting example game can be seen in Figure 3.

After automatic deployment, the desired point-and-click adventure is ready to use. Initially, the player sees the screen of the left side of the river where the wolf, the goat, and the cabbage are represented as buttons which, when clicked,

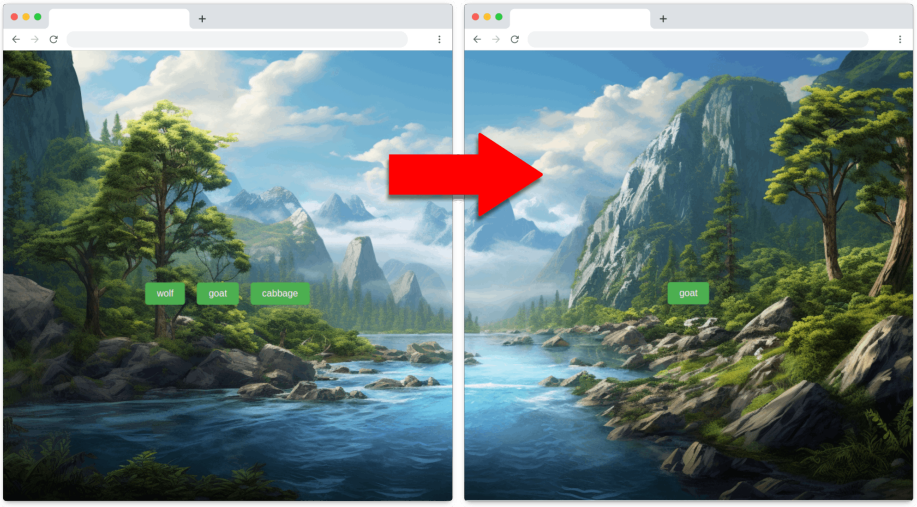

Fig. 3. The initial game screen (left) after full code generation and the screen after taking the goat to the right riverside (right).

transfer the corresponding object and the player to the other side of the river. In addition to the buttons, ChatGPT has also implemented the `checkWin()` and `checkLoss()` functions that check after each step whether the game is won or lost. If one of these conditions holds, the corresponding win or loss screen is displayed.

4.5 Model Learning & Verification

After deploying the generated web application, we can learn the instance using Malwa, the tool introduced in [21] to learn instrumented web applications just by providing the URL to the server where the application is deployed to the tool. Note that, because of the instrumentation, we do not need to specify an input alphabet for the learning algorithm, as it is build incrementally just by analyzing and observing the DOM of the website. The learning process results in the eleven-state automaton displayed in Figure 4. It represents the user-level interaction graph that results from the interactions with the user interface of the Webstory product by clicking on the generated buttons. We represent models using Moore automata[2] to simplify model checking and visual analysis, since each state is linked to exactly one screenshot, thus accurately reflecting user-level interactions with the website.

Upon visual inspection, one can see that the generation resulted in an application that adheres to the constraints formulated in our natural language

[2] Moore automata are semantically equivalent to Mealy automata which are the standard target automaton type for active learning of web applications.

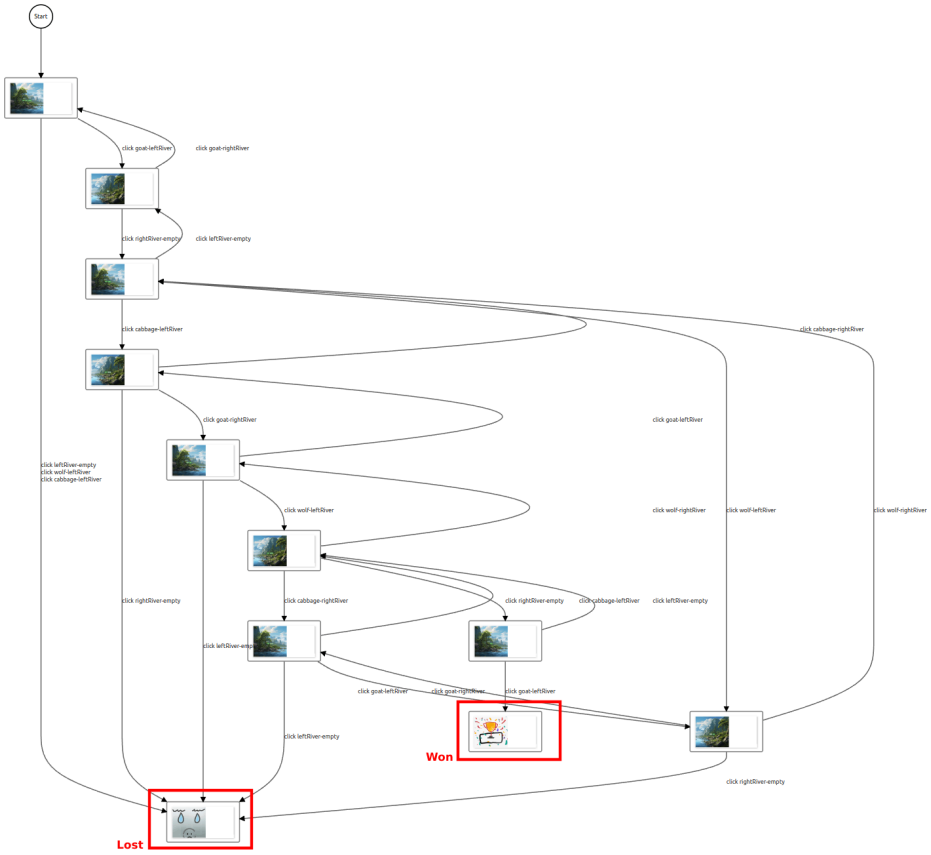

Fig. 4. Learned model of the generated Webstory.

prompt and that the game can be played as intended. The property that it is possible to win the game can also easily be seen as the *won* screen is reachable in the learned model. For larger models, model checking is an ideal means to verify such properties. Our solution supports CTL model checking using our model checker GEAR for this purpose. Currently, this requires the manual specification of the intended formula which, in particular, may still require to identify dynamically generated atomic propositions, a task that we are aiming to automatically support in the future.

5 Discussion and Related Work

Unsurprisingly, the novel ability to both understand and generate code from natural language descriptions (although still limited) has attracted attention from the software engineering community.

At present, applications of LLMs in software engineering are in very early stages and most work on LLMs and their role in software development focuses on supportive applications in traditional software engineering contexts [22]. In most of these works, LLMs are not tasked with full code generation but predominantly with tasks such as: Explaining code to a human, providing feedback to a human, finding potential bugs in human-written code or generating small-scale helper functions. Moreover, most of these works focus on applications of ChatGPT as it is one of the strongest and most widely used LLMs at present.

The authors of [31] apply ChatGPT to a range of code generation benchmark tasks as well as supporting tasks such as code summarization, bug fixing and program repair. They note that, while ChatGPT is neither able to repair, explain or generate programs reliably on its own, it can often provide a range of possible solutions from which an expert programmer can choose an adequate one and propose ChatGPT as "programming support" in software engineering. In this context, [26] propose a continuous improvement cycle where code generated by the LLM is model checked and counterexamples are fed back to the LLM to incrementally improve the generated code until all properties are satisfied.

Quite similarly, the authors of [30] evaluate ChatGPT's performance on bug fixing and program repair and, again, find empirically that performance is only adequate if a human programmer works in tandem with ChatGPT.

This point of view is also reflected in current industrial applications. Commercial tools such as GitHub Copilot [27] show that this is currently the predominant application of LLMs in software engineering.

An approach to using LLMs for the entirety of software development has been investigated by [23]. They investigate the potential of using LLMs for design, code generation, and testing processes. From their findings and their case study, they derive challenges to be addressed and possible future scenarios for LLM-based software engineering.

Larger-scale code generation has been attempted by AutoGPT [29]. Using specific prompts, AutoGPT forces ChatGPT to divide a larger task into multiple subtasks and iterate upon its earlier results to solve more complex problems in a divide and conquer fashion.

The idea to decompose a complex programming task into small pieces is similar to our approach, but the means are different: Whereas AutoGPT uses ChatGPT for handling the decomposition also, we use a formally defined DSL which provides us with better control and scalability. In particular, we can guarantee the executability of the overall generated application, a precondition to apply black-box checking.

We are not aware of any approach that aims at a similarly holistic integration of natural language-based specifications into a (no/low code) development framework.

6 Conclusion

We have presented an approach to no-code development based on the interplay of formally defined (graphical) Domain-Specific Languages and informal, intuitive Natural Language which is enriched with contextual information to enable referencing of formally defined entities. Our implementation within the LDE ecosystem which is designed to support application development using multiple DSLs has illustrated how one can exploit the best of the two language paradigms:

- the control provided by the graphical DSL to ensure executability of the developed application, a precondition to apply black-box checking for validation, and
- the ease of prompting ChatGPT with natural language which enables people without any computational knowledge to specify process requirements.

Technically, our approach depends on enhancing the code generator for the graphical modeling language to also generate an appropriate prompt frame that guarantees that the code generated from the natural language specification can be merged into the code generated from the graphical model while guaranteeing that the resulting application is executable.

Our approach has been illustrated in detail using the development of point-and-click adventures as minimal viable application scenario. More concretely, we stepwise developed a point-and-click adventure that is inspired by the river crossing puzzle.

Guaranteed executability is essential to provide non-technical users also with no-code validation, but it is not sufficient as the required automata learning typically requires to provide a dedicated learning alphabet. In our application scenario, the code generator guarantees that the generated applications are learnable by design: the required learning alphabet can be incrementally deduced from the Document Object Models (DOM) as part of the learning process. In contrast, the subsequent model checking requires the manual specification of the intended formula which, in particular, may still require to identify dynamically generated atomic propositions. We are currently investigating how far this identification can be automated.

The enormous potential of intuitive specifications became evident with the recent development of LLMs, and we are only at the beginning of understanding what their combined power with formal specifications might be. In this paper we have shown how to gain some control over LLM-generated code by embedding it into 'traditionally' generated code. This has allowed us to guarantee the executability of the generated overall applications, a precondition for applying validation techniques like black box checking. We are convinced that this kind of embedding LLM-based code generation into formal methods-based application development is a promising way to exploit the power of natural language specification while taming its shortcomings.

References

[1] E. M. Clarke, E. A. Emerson, and A. P. Sistla. "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications". In: *ACM Trans. Program. Lang. Syst.* 8.2 (Apr. 1986), pp. 244–263. DOI: 10.1145/5397.5399.

[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.

[3] Peter Burkholder. "Alcuin of York's Propositiones ad acuendos juvenes: Introduction, Commentary & Translation". In: *History of Science & Technology Bulletin* 1.2 (1993).

[4] Michael J. Kearns and Umesh V. Vazirani. *An Introduction to Computational Learning Theory*. Cambridge, MA, USA, 1994. ISBN: 0-262-11193-4.

[5] José L. Balcázar, Josep Dıéaz, and Ricard Gavaldà. "Algorithms for Learning Finite Automata from Queries: A Unified View". In: *Advances in Algorithms, Languages, and Complexity*. 1997, pp. 53–72.

[6] Doron Peled, Moshe Y Vardi, and Mihalis Yannakakis. "Black box checking". In: *International Conference on Protocol Specification, Testing and Verification*. Springer. 1999, pp. 225–240.

[7] H. Hungar, T. Margaria, and B. Steffen. "Test-based model generation for legacy systems". In: *Test Conference, 2003. Proceedings. ITC 2003. International*. Vol. 1. Oct. 2003, pp. 971–980. DOI: 10.1109/TEST.2003.1271205.

[8] Harald Raffelt, Bernhard Steffen, and Tiziana Margaria. "Dynamic Testing Via Automata Learning". In: *Hardware and Software: Verification and Testing*. Ed. by Karen Yorav. Vol. 4899. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2008, pp. 136–152. DOI: 10.1007/978-3-540-77966-7_13.

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

[10] Bengt Jonsson. "Learning of Automata Models Extended with Data". In: *Formal Methods for Eternal Networked Software Systems: 11th International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM 2011, Bertinoro, Italy, June 13-18, 2011. Advanced Lectures*. Ed. by Marco Bernardo and Valérie Issarny. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 327–349. ISBN: 978-3-642-21455-4. DOI: 10.1007/978-3-642-21455-4_10. URL: https://doi.org/10.1007/978-3-642-21455-4_10.

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

[12] Alexander Bainczyk et al. “Model-Based Testing Without Models: The TodoMVC Case Study”. In: *ModelEd, TestEd, TrustEd: Essays Dedicated to Ed Brinksma on the Occasion of His 60th Birthday*. Ed. by Joost-Pieter Katoen, Rom Langerak, and Arend Rensink. Cham: Springer International Publishing, 2017, pp. 125–144. DOI: 10.1007/978-3-319-68270-9_7.

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

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

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

[16] Mark Chen et al. “Evaluating large language models trained on code”. In: *arXiv preprint arXiv:2107.03374* (2021).

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

[18] Aakanksha Chowdhery et al. “Palm: Scaling language modeling with pathways”. In: *arXiv preprint arXiv:2204.02311* (2022).

[19] Robin Rombach et al. “High-resolution image synthesis with latent diffusion models”. In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2022, pp. 10684–10695.

[20] Andrea Agostinelli et al. “Musiclm: Generating music from text”. In: *arXiv preprint arXiv:2301.11325* (2023).

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

[22] Adna Beganovic, Muna Abu Jaber, and Ali Abd Almisreb. “Methods and Applications of ChatGPT in Software Development: A Literature Review”. In: *Southeast Europe Journal of Soft Computing* 12.1 (2023), pp. 08–12.

[23] Lenz Belzner, Thomas Gabor, and Martin Wirsing. “Large Language Model Assisted Software Engineering: Prospects, Challenges, and a Case Study”. In: *This Volume* (2023).

[24] Steve Boßelmann. “Evolution of Ecosystems for Language-Driven Engineering”. PhD thesis. Dortmund, Germany: TU Dortmund University, 2023. DOI: 10.17877/DE290R-23218.

[25] Sébastien Bubeck et al. “Sparks of artificial general intelligence: Early experiments with gpt-4”. In: *arXiv preprint arXiv:2303.12712* (2023).

[26] Yiannis Charalambous et al. *A New Era in Software Security: Towards Self-Healing Software via Large Language Models and Formal Verification*. May 2023. DOI: 10.48550/arXiv.2305.14752.

[27] GitHub. *GitHub Copilot*. https://copilot.github.com/. Accessed: July 21, 2023. 2023.

[28] OpenAI. "GPT-4 Technical Report". In: *ArXiv* abs/2303.08774 (2023).

[29] Toran Bruce Richards. *Auto-GPT: An Autonomous GPT-4 Experiment*. Accessed: 21/07/2023. 2023. URL: https://github.com/Significant-Gravitas/Auto-GPT.

[30] Dominik Sobania et al. "An analysis of the automatic bug fixing performance of chatgpt". In: *arXiv preprint arXiv:2301.08653* (2023).

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

[32] Hugo Touvron et al. "Llama: Open and efficient foundation language models". In: *arXiv preprint arXiv:2302.13971* (2023).