LLM-assisted Metamorphic Testing of Embedded Graphics Libraries

Christoph Hazott Daniel Große
Institute for Complex Systems, Johannes Kepler University Linz, Austria
christoph.hazott@jku.at daniel.grosse@jku.at

***Abstract*—Modern applications increasingly rely on embedded systems that incorporate visual interfaces developed utilizing so-called embedded graphics libraries. Verifying these embedded graphics libraries is challenging due to hardware dependencies and the lack of reference outputs. The lack of reference outputs is tackled in *Metamorphic Testing* (MT) by constructing two *Firmware* (FW) versions with distinct implementations that maintain the same input-output relationships. These relations are known as *Metamorphic Relation*s (MRs). However, the development of these MRs remains a tedious and challenging task.**

In this paper, we present a novel approach for generating MRs for MT of embedded graphics libraries using *Large Language Model*s (LLMs). Because directly creating MRs with simple prompts is too complex for the LLM, we employ proven prompting strategies to develop our LLM-assisted MR pipeline. Strategies include role prompting, least-to-most prompting, zero-shot prompting, constraint-based prompting, and style prompting. In our experiments, we verify a widely used embedded graphics library. We compare our results with an existing manual approach and demonstrate that LLM-assisted MRs nearly doubles coverage and identifies additional bugs.

I. Introduction

Embedded devices have become ubiquitous in todays applications, ranging from consumer electronics to industrial control systems. To facilitate intuitive and efficient user access, visual interfaces are provided. Embedded graphics libraries are fundamental to the development of such visual interfaces. These *Software* (SW) libraries enable developers to efficiently render graphics within the limited computational and memory resources of embedded environments [1]. Their performance and capabilities are inherently dependent on the underlying *Hardware* (HW), requiring tight integration with specific processors, accelerators, and display technologies; therefore, embedded graphics libraries are typically considered part of the *Firmware* (FW). In addition, as the functionality of embedded graphics libraries rapidly increases, so does their complexity. As a consequence, the effort required for verification increases considerably, necessitating early HW/SW integration.

To enable verification of embedded graphics libraries early in the development process, the HW down to the display is typically modeled using abstractions, such as emulators or simulators. The latter replicate the behavior of the actual HW and therefore enable the parallel SW and HW development [2].

An industry-proven approach for HW simulation are *Virtual Prototype*s (VPs) which are high-level executable models of HW platforms capable of running unmodified production SW [2]. The predominant language for creating VPs is SystemC, a standardized C++ class library (IEEE 1666, [3]); for more details on SystemC we refer to [4]–[6]. Leveraging these VPs allows developers to test graphical functions without the need for physical HW.

Looking at the different test levels, unit testing is first employed to verify individual components or functions within embedded graphics libraries. This ensures that each component performs correctly in isolation. Then integration testing is used to verify that the different components of the graphics library interact correctly with each other and with the HW. In general, integration testing allows us to identify issues that may arise from component interplay or HW/SW integration [7], [8]. For integration testing of embedded graphics libraries, the major difficulty is establishing an effective and reliable method to determine whether the graphical outputs produced are correct. This is well known as *oracle problem* [9]. Therefore, in practice, often manual visual inspection of the display output is carried out.

Metamorphic Testing (MT), as presented in [10], [11], mitigates the oracle problem by leveraging so-called *Metamorphic Relation*s (MRs). These MRs express expected relationships between inputs and outputs rather than relying on reference inputs and outputs. In recent works [12] and [13], an open-source MT framework has been introduced for the purpose of verifying embedded graphics libraries. The authors manually developed 21 MRs. One of these MRs is called `DrawRectangle` MR and verifies whether two different methods, capable of drawing rectangles, render the same output when the parameters are set accordingly: `drawRect()` draws a rectangle with sharp corners whereas `drawRoundRect()` draws a rectangle with rounded corners; by disabling the rounded corners via a parameter, both methods are expected to render identical rectangles. The MT framework then generates two FW versions: the *source FW* which calls `drawRect()` and the *follow-up FW* which calls `drawRoundRect()`. The source and follow-up FW versions are executed separately on a RISC-V VP using a virtual HW display. This complete implementation of the SW-to-HW stack[1] enables the framework to capture screenshots of the resulting display output, which are then compared to determine whether the test has passed

[1]**SW** (Application→Library→Driver) **-to-HW** (CPU→Bus→SPI→Display)

or failed. As reported in [13], the MT approach has found 15 novel bugs in the widely used graphics library TFT_eSPI. However, manually developing MRs, which includes identifying non-obvious relationships – such as drawing a line by drawing multiple filled circles – can be a time-consuming and challenging process.

Contribution: Motivated by these observations, this paper introduces a novel approach that uses *Large Language Model*s (LLMs) to generate MRs for MT of embedded graphics libraries. Our MT approach extends [13] such that domain-specific and effective MRs are produced. While simple prompting techniques are sufficient to select an appropriate LLM model for MR generation, these types of prompts are insufficient to directly address the complexity of generating MRs for embedded graphics libraries. We solve this by employing well-established prompt engineering strategies, such as those surveyed in [14]. This includes `Role Prompting`, which assigns a developer persona to the LLM, and `Least-to-Most Prompting` to decompose the complex problem of generating an MR into simpler subproblems. Finally, the subproblems are integrated as steps into an LLM-assisted MR pipeline.

For evaluation, we apply the proposed LLM-assisted MT approach to verify the widely used TFT_eSPI library. We used Meta Llama 3.1 [15] via the Huggingface Transformers toolkit [16]. This local execution of the LLM, combined with setting a constant seed and disabling sampling, ensures the full reproducibility of the generated results. Executing the pipeline in this setup easily generated 160 unique MRs, an almost eightfold increase over 21 manually developed MRs of [13]. Additionally, the generated MRs uncover 14 novel bugs and they nearly double coverage.

The key contributions of this work include:

- Novel LLM-assisted MT approach tailored for embedded graphics libraries,
- Comprehensive LLM-assisted MR generation pipeline, systematically addressing subproblems employing established prompt engineering strategies,
- Extension to the existing open-source MT framework [17] available open-source on GitHub[2],
- Fully reproducible LLM generation and MT results, and
- Thorough evaluation of the effectiveness of our approach, including the analysis of structural coverage metrics and the detection of previously unknown bugs.

The paper is structured as follows: Section II discusses related work. Section III provides background on the targeted embedded graphics library and the VP used for simulation. In Section IV, we present our LLM-assisted MT approach. Section V illustrates each step of the LLM-assisted MR generation pipeline, including example prompts and LLM responses. Section VI presents the experimental evaluation. Finally, a general discussion of using LLMs and the introduced LLM-assisted MR pipeline is done in Section VII. Section VIII concludes the paper.

[2]https://github.com/ics-jku/llm-assisted-mt

II. Related Work

Recently, a systematic study on the application of LLMs for *Electronic Design Automation* (EDA) was presented in [18]. The different tasks along the EDA flow are considered, including the application of LLMs for verification and analysis. For example, [19] incorporates feedback from commercial-grade EDA tools into LLMs to enhance testbench generation. [20] splits the task of generating and evaluating HW verification assertions into three steps, which are solved by different LLMs.

Parallel to these works, the application of LLMs for MT has been investigated. In an exhaustive case study on nine software systems, [21] asks whether ChatGPT can be used for SW tests with a particular focus on MT. The MRs in this work are generated using the zero-shot prompting strategy. The resulting MRs are then verified by domain experts who found that ChatGPT proposed MR candidates that can be adopted for implementing SW tests. The work [22] also explores the use of ChatGPT to automatically generate MRs. The idea is to use few-shot prompting to provide API specifications and requirements to the LLM. The study generated MRs for four software applications and for a web application. The quality of the resulting MRs was evaluated through a questionnaire-based survey and showed that this approach was able to generate comprehensible and pertinent MRs for testing purposes. The third study, which is relevant for this paper, is [23]. It generates MRs for autonomous driving systems. This study was also conducted using ChatGPT and the zero-shot prompting strategy. The evaluation of the generated MRs was also carried out by domain experts, who found ChatGPT to be a promising tool for MR generation in SW testing.

In contrast to these studies, our approach focuses on embedded graphics libraries, meaning our MRs target the full SW-to-HW stack. In addition, the evaluated studies conclude with a manual evaluation of the generated MRs by MT specialists without executing the tests. Within this paper, we also execute the generated MRs and evaluate the test results according to the identified bugs and the achieved structural coverage. Finally, our findings can be reproduced, as we make the necessary artifacts (including the implementation) open-source on GitHub. Furthermore, using the Huggingface Transformers toolkit and the Meta Llama 3.1 model, our approach runs locally with deterministic results.

III. Background

Before we introduce the LLM-assisted MT approach, we provide some background information on the TFT_eSPI library and the utilized VP.

A. TFT_eSPI Library

A simple example of the embedded graphics librarys drawing capabilities is presented in Fig. 1. The library is ranging within the 98th percentile of stars on GitHub (tracked by [24]), has about $1,200$ forks, and has the seventh rank out of $7,597$ libraries in the Arduino Library List of the most starred libraries [25].

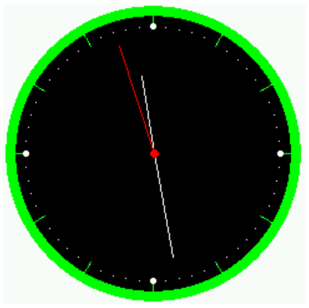
Fig. 1: Analog clock rendered using the TFT_eSPI library.

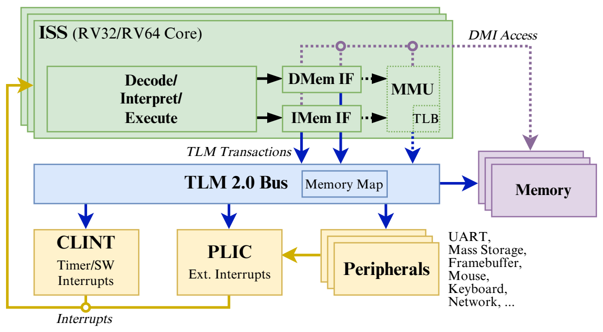

Fig. 2: RISC-V VP++ architecture showing ISS, the TLM-2.0 bus, memories, interrupts units and peripherals.

The main part of the library contains about $5,115$ lines of code contained within 163 functions, whereas about $1,380$ are responsible for drawing outputs on the display (the rest of the lines contains e.g. touch input, *Serial Peripheral Interface* (SPI) interactions, etc...). As one can imagine, filtering the drawing capabilities to generate meaningful MRs poses a large manual effort.

B. RISC-V VP++

In general VPs are commonly used for advanced verification approaches like [26]–[33]. The RISC-V VP++ [34] used in this paper is an open-source SystemC/TLM-2.0-based VP supporting 32- and 64-bit RISC-V cores (RV32/RV64), as shown in Fig. 2. The *Instruction Set Simulator* (ISS) includes Decode, Interpret, Execute units, *Instruction Set Architecture* (ISA)-defined registers (incl. *Control and Status Registers* (CSRs)), and an *Memory Management Unit* (MMU) for virtual memory support. Instruction and data memory access is solved via *Direct Memory Interface* (DMI) or the TLM-2.0 bus, which also connects peripherals and interrupt controllers (*Core Local Interruptor* (CLINT)/*Platform-Level Interrupt Controller* (PLIC)). Peripherals like *Universal Asynchronous Receiver-Transmitter* (UART), storage, or input devices connected via *Memory-Mapped Input/Output* (MMIO), allowing seamless ISS access via standard load/store instructions. The VP supports configurations from bare-metal microcontrollers to Linux-capable systems complemented with interactive graphical applications [35]. Recently the RISC-V vector extension has been added to RISC-V VP++ [36], [37] and significant ISS optimizations have been devised to increase the simulation performance [38].

IV. LLM-ASSISTED MT

Developing good MRs is a key element for the effectiveness of MT [10], [39]. In this work, we assist the developer by generating MRs using LLMs. We start with the necessary terminology and notation (Section IV-A) before we assess the current knowledge and capabilities of LLMs using simple prompts (Section IV-B). The assessment determines whether there is an LLM that is able to generate and implement MRs for embedded graphics libraries. Furthermore, having such an LLM eliminates the need for costly, complex, and potentially unnecessary fine-tuning. Despite having an LLM with the appropriate knowledge and capabilities, generating MRs for embedded graphics libraries is a complex task that cannot be accomplished merely by using simple prompts. The reasons are twofold: First, simple prompts may not provide LLMs with enough context or specificity to generate meaningful and effective MRs. Second, simple prompts often produce superficial outputs, inadequate for generating MRs which demand a detailed understanding and precise articulation of FW behaviors and properties. Therefore, we employ well-established prompt engineering strategies [14]. First, we set the context for the LLM to our problem domain (Section IV-C). Then, we want to break down the MR generation problem into individual steps using the LLM (Section IV-D). Remarkably, the LLM proposes 7 steps which we can directly use to structure an MR pipeline for MR generation. We implemented each of these steps (Section IV-E) and again used the LLM to find domain-specific graphical relations among API methods of the embedded graphics library.

Before we further detail our approach, we begin with introducing the terminology and notation used.

A. Terminology and Notation

As already outlined in the introduction, MT for embedded graphics libraries requires the separate execution of two FW versions – the source FW and the follow-up FW – as well as the subsequent comparison of their graphical outputs. Moreover, both FW versions together with the expected relation of the graphical results produced (here visual equivalence) constitute an MR.

In the following, we approach the MR generation problem starting from an abstract perspective, i.e. we consider MRs on different levels:

- **Abstract:** The terms *source test* and *follow-up test* are used to reflect that they contain variables, i.e. the input to the methods used may depend on symbolic values. The assignment of concrete values yields the *source test **case*** and *follow-up test **case***, collectively referred to as a *Metamorphic Test Case* (MTC).
- **Executable:** Finally, the executable versions of the source test case and the follow-up test case correspond to our source FW and follow-up FW, respectively.

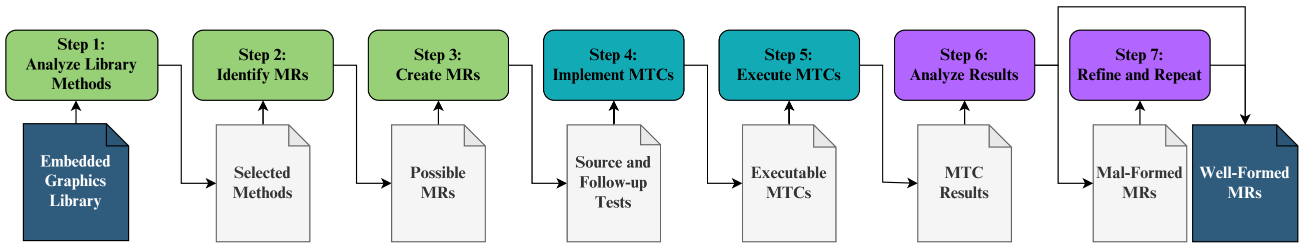

Fig. 3: LLM-assisted MR pipeline. Dark blue boxes indicate input and output to and from the pipeline. Green and light blue boxes indicate that these steps are fully automated. Green boxes indicate additionally that these steps are executed using the LLM. Light blue boxes additionally indicate that these steps are executed by the testing framework, independently of the LLM. Purple boxes indicate manually executed steps.

Prompt 1: Assigning a role

You are the developer of the `[X]` embedded graphics library.

Prompt 2: Decompose MR generation

For your library, Metamorphic Testing will be implemented. Give me a step-by-step manual to do this without implementing the steps.

B. Assessing LLMs via Simple Prompts

We evaluated the current knowledge and capabilities of LLMs, by performing an assessment on a variety of pre-trained LLMs, including OpenAI's GPT-4o [40], Mistral AI's Mistral v0.3 [41], and Meta's Llama 3.1 [15]. Each of the mentioned LLMs had knowledge of multiple graphics libraries (including the one used in Section V). When prompting for example implementations, the top results came from GPT-4o and Llama 3.1. Moreover, all of the mentioned LLMs demonstrated an understanding of MT and were able to propose alternative implementations for specific functionalities, once again with Llama 3.1 and GPT-4o delivering the best responses.

Based on the assessment, Llama 3.1 was chosen due to its open-source availability, advanced coding capabilities, and strong library skills, which made additional fine-tuning unnecessary and allowed us to focus on solving the task with advanced prompting strategies.

C. Assigning a Role to the LLM

To further improve the results of Llama 3.1, the first prompt engineering strategy we employ is called `Role Prompting` and assigns a persona to the LLM. Each interaction with the LLM is preceded by Prompt 1. In this prompt, `[X]` should be replaced by the name of the embedded graphics library. The assignment of the role of *library developer* helped to exclude the cases where the LLM generated responses that targeted other graphics libraries.

D. Decomposing the Problem: LLM-assisted MR Pipeline

To break down the complexity of the MR generation, we decompose the problem with the assistance of the LLM. The `Least-to-Most Prompting` strategy contained within the pool of `Decomposition` strategies [14] is applicable in our case. The idea is to ask the LLM to decompose the problem into subproblems without solving them, as seen in Prompt 2. The LLM response to this prompt suggests splitting the problem into 7 subproblems [3] as can be seen in Response 1,

Response 1: LLM steps in response to Prompt 2

1. Understand the Library
2. Identify Metamorphic Relations
3. Create Test Cases
4. Implement Test Cases
5. Run Test Cases
6. Analyze Results
7. Repeat and Refine

which we found remarkable. Furthermore, the LLM suggested using a testing framework and visualization tools.

Building on the "7 steps answer" provided by the LLM, we designed and assembled the MR pipeline illustrated in Fig. 3. The figure is organized in two rows. The top row shows the 7 pipeline steps identical to the LLM response with two minor exceptions: As can be seen, the LLM uses the term `Test Cases` in Steps 3-5. To fit the abstraction levels introduced in Section IV-A, we labeled Step 3 `Create MRs` instead of `Create Test Cases`. This is done because the values are kept symbolic until Step 4. Starting from Step 4, the values become concrete. For this reason, we add the prefix `Metamorphic` to the term `Test Cases` of Step 4 and Step 5 (leading to the abbreviation MTC).

The LLM-assisted pipeline steps (green) are designed to generate LLM prompts, and subsequently post-process the LLM response. The colored cyan boxes indicate that these steps do not interact with the LLM. The lower row shows the input and output files for each step. The dark blue files represent the input and output of the pipeline. In the following section, we detail each step.

E. LLM-assisted MR Pipeline Steps

Step 1 - Analyze Library Methods: The goal of our MT approach is to verify the library's capabilities to generate graphic objects on an embedded display. To do this, first, all library methods need to be analyzed to identify the methods with such capabilities. Asking the LLM to output a list of

[3]We shortened the response by removing the textual explanation per step.

Prompt 3: Step 1 - Analyze library methods

There are multiple methods within your library. Take a closer look at the definition of the `[SIGNATURE]` method. For the following questions answer only with 'Yes' or 'No'.
Q: Calling the `[SIGNATURE]` method generates a graphical object on the display?
A:

Prompt 4: Step 2 - Identify MRs

For all possible parameters, can a working alternative of the following source code be generated: `[SOURCE]`
Follow the hard requirements:
- the functionality should be implemented using the `[FOLLOWUP]` API method
- the same geometrical properties and the colors need to be kept for the generated display output
- make sure not only the outline of the objects matches, but also the infill
Answer only with 'Yes' or 'No'

such methods was not successful. So we designed this pipeline step LLM-assisted. The idea is to use a code parser and extract all methods with a public interface from the library. After this, each method is classified by the LLM with `Yes`, if the method can generate graphic objects or `No`, if not. The appropriate prompt employs the `Zero-Shot Prompting` strategy, which means that only one prompt is necessary to solve the task. Informing the LLM that it is participating in a question-and-answer exchange enhanced the results of this prompt. To do this, we used **Q:** to explicitly indicate our question and **A:** to explicitly tell the LLM to answer the question. Furthermore, we discovered that encouraging the LLM to thoroughly analyze the definition of methods led to an improvement in the number of accurate classifications. [4] With this, we devised Prompt 3 (`[SIGNATURE]` denotes method signature).

Step 2 - Identify MRs: In this step, the method signatures, classified with *'Yes'*, are used to identify potential MRs. Since prompting the LLM to directly generate MRs for each method was not successful, we came up with a different approach. The idea is to identify an alternative realization for a given method. Specifically, for a method classified with *'Yes'*, our objective is to find an alternative method, from the pool of methods classified with *'Yes'*, that can produce an identical graphical output [5]. For all possible alternatives, Prompt 4 is executed, with `[SOURCE]` and `[FOLLOWUP]` replaced by the corresponding method signatures. In addition to `Zero-Shot Prompting`, the `Constrained-Based Prompting` strategy was applied by adding requirements for the LLM such as: `the same geometric properties and the colors need to be kept for the generated display output`. This was done to improve classification reliability.

[4]Using 'specification' reduced the number of correct classifications.

[5]Alternative here also includes to use the same method, but switching/adjusting parameters.

Prompt 5: Step 3 - Create MRs

Generate an alternative implementation of the source code: [SOURCE]
Follow the hard requirement:
- use the [FOLLOWUP] method for your implementation
- make sure the same geometrical properties are kept
- the image created on the display stays the same
- output only the C++ code

Step 3 - Create MRs: Having a list of possible pairs of methods, the pipeline continues to create the MRs. For the source tests, the call to the method is taken directly. To create the code for the follow-up tests, we reformulate Prompt 4 into Prompt 5 so that the LLM generates the alternative implementation using the `Style Prompting` strategy to generate C++ code. Please note that although at this step we already have C++ code for the source and follow-up tests, the implementation is still abstract, as it still contains variables for the method calls.

Step 4 - Implement MTCs: This is the first step that works independently of LLMs and will be executed by the suggested testing framework from Section IV-D. The step assigns concrete values to the variables while satisfying the relation and translates the previously created MRs into executable MTCs.

Step 5 - Run MTCs: The purpose of Step 5 is to execute each MTC at least once. This is done (1) to find out which MTCs are genuinely executable and (2) to generate display outputs for further analysis.

Step 6 - Analyze Results: The results of Step 5 are manually inspected for correctness by analyzing the display outputs. An execution of an MTC that results in accurate display outputs, where both FW versions yield identical outputs, is considered as a `Well-Formed` MR. Otherwise, the MR is labeled as `Mal-Formed`.

Step 7 - Refine and Repeat: This step is again a manual step where the `Mal-Formed MRs` are evaluated. The goal is to modify additional MRs to be `Well-Formed`.

In the following section, we outline the implementation aspects and present our approach using examples.

V. Demonstration of LLM-assisted MT

In this section, we demonstrate our LLM-assisted MT approach for embedded graphics libraries. Following the suggestions as given by the LLM in Section IV-D, we need a testing framework and visualization tools. The MT framework for embedded graphics libraries from [13], available on GitHub [17], fulfills this request. Moreover, the framework verifies the drawing capabilities of the embedded graphics library TFT_eSPI by manually creating the MRs. Hence, we also consider the verification of the TFT_eSPI library to compare our approach against manually creating MRs.

To execute the steps of the MR pipeline (see Fig. 3 and Section IV-E), we use Python and Huggingface Transformers. The latter serves as a LLM toolkit, linking to the PyTorch library [42]. To guarantee reproducibility of the results, the

TABLE I: LLM-assisted MR pipeline results showing possible, rejected and selected candidates for each step (cf. Fig. 3) in rows. The columns contain the single steps according to the pipeline. The second row header indicates the type of the candidates, used for the corresponding steps.

Candidates	Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Step 7
	Methods	MRs		MTC generators		Resulting MRs	
Possible	163	3,481	2,357	2,245	1,199	908	847
Rejected	104	1,124	112	1,046	291	847	722
Selected	**59**	**2,357**	**2,245**	**1,199**	**908**	**61**	**125**

PyTorch seed was set to a constant value of 42 and the Transformers toolkit sampling was turned off. Table I shows the results for each step. Each step starts with a list of candidates as indicated by the column headers (e.g. Step 1 starts with 163 method candidates) from which a subset is selected to proceed to the next step while the remaining candidates are rejected.

Step 1 - Analyzing the Library: Our goal here is to cover the main drawing capabilities provided via the TFT_eSPI interface. Therefore, a code parser extracts possible method candidates, i.e. public methods. This extraction led to 163 methods[6]. For each of these methods, the LLM was asked to classify if the method is capable of rendering content on the display. Of these 163 method candidates, the LLM successfully identified 59 drawing candidates.

Step 2 - Identify MRs: The LLM pipeline continues in this step with 59 drawing candidate methods. Considering that each of the 59 selected drawing candidate methods can be used as a substitute for any other method, there are a total of $59 59 = 3,481$ possible combinations. Our implementation evaluates all these combinations via the LLM Prompt 4. $2,357$ combinations received the answer *'Yes'*; therefore, they proceed to the next step as possible MR candidates.

Step 3 - Create MRs: Source test and a follow-up test versions are created for each of the $2,357$ MR candidates. Furthermore, due to the optimization of the LLM for contextual reasoning and comprehensive response generation, it additionally provides an explanation of how the follow-up test is derived from the source test, even if not explicitly requested in the prompt. An example is shown in Listing 1. We can see the source test call in Line 4–5 of Listing 1. In this example, it is `drawFastHLine()` with parameters x and y for the origin and w for the width of the horizontal line. The parameter *color* sets the color of the line. In Line 10–18 of Listing 1, we can see the follow-up test. As an alternative implementation to `drawFastHLine()`, the LLM generated code to draw the line using a rectangle. Furthermore, the LLM generated an explanation as can be seen in Line 22–26 of Listing 1. Transforming a one-dimensional line into a two-dimensional rectangle with a height of 1, by recognizing its attributes, highlights the advanced inference capabilities of modern LLMs to handle geometric concepts.

Such transformations require a high amount of memory on the graphics card. Due to the 16 GB video memory capacity in our setup, the LLM was not able to carry out this step for

[6]We excluded the methods for touch and sprites.

````
 1  Source
 2  ----------
 3  ```cpp
 4  drawFastHLine(int32_t x, int32_t y,
 5                int32_t w, uint32_t color)
 6  ```
 7  Follow-up
 8  ----------
 9  ```cpp
10  void drawFastHLine(int32_t x, int32_t y, {
11                     int32_t w, uint32_t color)
12      // Calculate the height of the line
13      // which is 1 pixel in this case
14      int32_t h = 1;
15
16      // Call the drawRect method to draw the line
17      drawRect(x, y, w, h, color);
18  }
19  ```
20  MR
21  ----------
22  This alternative implementation uses the `drawRect`
23  method to draw a horizontal line. The height of the
24  rectangle is set to 1 pixel, effectively drawing a line.
25  The width of the rectangle is set to the width of the
26  line, and the color is set to the specified color. ...
````

Listing 1: Excerpt of markdown result for generated code pairs containing the C++ code for the generated source and follow-up implementations. Additionally the result contains the MR description as generated by the LLM.

112 MR candidates. This leads to a final number of $2,245$ MR candidates (see column Step 3 in Table I).

Step 4 - Implement MTCs: This is the first step performed independently of the LLM. The used MT framework is built on so-called MTC generators between the MR and executable MTC. An MTC generator contains templates of a source test and the corresponding follow-up test where the symbolic values are replaced with concrete values when the MTC generator is executed. If the MTC generator is executed several times per MR, different concrete values are determined for the symbolic values. To fit this design, the implementation within this step parses the MR candidates selected in Step 3 and translates them into MTC generators. To gain further insight into the implementation of MTC generators and their role in producing the source and follow-up FW versions, please see Appendix A.

From the original $2,245$ MR candidates, $1,199$ MTC generators could be translated and compiled successfully. The $1,049$ rejected MTC generators contained tests that failed compilation or missed additional behavior that would have to be implemented. For example, the `pushImage()` method pushes an array of pixels (the image) onto the display. This array is given as an input parameter to the source test case. An example follow-up test case required that the parameters given to the `drawCircle()` method generate the same image as the one encoded within the array of pixels. This highlights the limitations of current LLMs as none of the MTC generators contained the appropriate code [7].

Step 5 - Execute MTCs: Next, the MT framework executed all MTC generators at least once, meaning that at least one MTC is generated per MTC generator. The MT framework builds on the RISC-V VP++ introduced in Section III,

[7]Even for humans, this problem is non-trivial due to its complexity and the cognitive effort involved.

extending it with a virtual display to visualize the output of both source and follow-up test cases. The framework is implemented in Python and handles MTC generator selection, MTC generation, test execution, and result comparison.

Out of the 1,199 MTC generators, 291 needed to be discarded since executing the source and follow-up test cases caused either an infinite loop or a memory allocation error. Because of this, only the results of the 908 MTC generators were selected to proceed to the next step.

Step 6 - Analyze Results: For the remaining 908 MTC generators, 61 of them generated reasonable MTC results. Thus, 61 are well-formed MRs and the remaining 847 MTC generators are moved to Step 7 for refinement.

Step 7 - Refine and Repeat: At this step, we manually review each of the 847 MTC generators that were previously dismissed, in order to identify mistakes in the related MR. We have identified two common cases for such errors:

- Dimensional offsets; e.g. for rectangles, the width is specified, whereas for lines, the start and end coordinates are used. This discrepancy leads to a one-pixel offset.
- Missing method calls; e.g. `pushColor()` requires an additional call to `setAddr()` to set the coordinates. The LLM did not consider this requirement.

Modifying these MRs and executing the corresponding MTC generators again gave another 125 well-formed MRs. This brings up the final result to 186 MRs generated by the proposed LLM-assisted approach.

It is worth noting that while the last two steps – analyzing and refining the results – were performed manually, they required only a few hours of effort. In contrast, without this approach, the process of generating MRs, implementing and testing them, and subsequently analyzing and refining the results, typically spans several days to weeks.

VI. LLM-assisted MT Results

In this section, we evaluate the MRs generated by our LLM-assisted MT approach for embedded graphics libraries in comparison to the manually crafted MRs for the MT framework from [13]. First, we compare the bug detection capabilities. Then, we compare the structural coverage achieved.

A. Comparing Bug Detection Capabilities

Table II provides an overview of the detected bugs. The methods in which the bugs occurred are given in column *Method*. The number of bugs identified for the corresponding method is given in column *Bugs*. The following two columns show whether the bug has been detected (✓) or not (×) with the *MT framework* from [13] and our proposed *LLM-assisted MT* approach. As can be seen in the table, 12 of the bugs have been identified by the MT framework from [13] and our approach. The first difference are two bugs in the `drawWedgeLine()` method that are not identified by our approach. The reason for this is that the LLM was not able to recognize that changing the starting and end points leads to a valuable MR when drawing a wedge line. As `drawWedgeLine()` is a highly specific functionality of the TFT_eSPI library, an additional interaction with the LLM showed that only limited "knowledge" is available on wedge lines. The next difference is for the `fillSprite()` method. The reason for this is that we did not generate MRs for extensions as we focus only on the core functionality of the TFT_eSPI library.

TABLE II: Comparison of detected bugs. The first column contains the method name as defined in the TFT_eSPI library. The second column indicates the number of bugs identified for each method. The last two columns indicate if the single bugs were identified (✓) or not (×) for the MT framework and the LLM-assisted MT.

Method	Bugs	MT framework [13]	LLM-assisted MT
drawWedgeLine	2	✓, ✓	×, ×
frameViewport	1	✓	✓
fillSprite	1	✓	×
drawRoundRect	1	✓	✓
fillRoundRect	1	✓	✓
drawEllipse	1	✓	✓
fillEllipse	1	✓	✓
drawCircle	1	✓	✓
fillCircle	1	✓	✓
drawArc	3	✓, ✓, ✓	✓, ✓, ✓
fillSmoothCircle	3	✓, ×, ×	✓, ✓, ✓
drawRect	2	✓, ×	✓, ✓
fillRect	1	×	✓
drawLine	1	×	✓
drawSmoothRoundRect	1	×	✓
drawWideLine	2	×, ×	✓, ✓
pushImage	1	×	✓
drawBitmap	1	×	✓
drawXBitmap	1	×	✓
drawSpot	3	×, ×, ×	✓, ✓, ✓

Next, for the `fillSmoothCircle()` method, our LLM-assisted MT approach found 2 new bugs. One of the newly identified bugs in `fillSmoothCircle()` appears when the circle size is set to 1 pixel. In this case, no circle is drawn. For `drawRect()`, we also found a new corner-case bug. When the line size is set to 0, two pixels are drawn instead of none.

Finally, our proposed approach identified 11 bugs in methods that the MT framework from [13] did not find, leading to a total of 14 previously unknown bugs. One of these bugs was found in the `drawBitmap()` method. When generating a filled rectangle using `drawBitmap()`, some segments of pixels were missing within the rectangle. Another bug was found in the `drawSpot()` method, which showed that when a small radius was given to the method, no spot was drawn.

The newly found bugs are occurring under certain conditions. The criticality is therefore context-dependent and requires further evaluation in consultation with the library developers.

B. Structural Coverage Results

To further evaluate the effectiveness of the proposed approach, the structural coverage was measured based on the 59 methods identified in Step 1. The full function coverage (100%) is therefore achieved when all 59 methods are covered. Note that our approach identified more methods with drawing capabilities than the MT framework from [13]. Therefore, in our measurement, the absolute number of functions as well as lines and branches, which can be covered, has increased. This leads to different coverage results in our measurement and those presented in [13]. Table III shows

TABLE III: The tables contains the structural coverage results. The first column indicates the coverage type. The second and third column indicate how much coverage for the single types was achieved for the MT framework and the LLM-assisted MT.

Coverage	MT framework [13]	LLM-assisted MT
Function	47.46%	89.83%
Line	44.93%	73.70%
Branch	45.74%	77.60%

the results for function, line, and branch coverage, comparing the 21 manually created MRs from the MT framework [13] with the 186 MRs generated by our LLM-assisted MT. After executing $1,000$ MTCs per MR, no further increase in coverage was observed. In terms of function coverage, the MT framework reached 47.46%, while the LLM-assisted MT achieved 89.83%, nearly doubling the coverage. This result can be attributed to the significant manual effort required to find all the methods with drawing capabilities of the library, which we automated in Step 1 using the LLM. Although manual evaluation identified 28 methods from 163 public methods, our approach automatically identified 59 methods capable of drawing objects on the display. However, the table also shows that our LLM-assisted approach does not reach 100% function coverage since our pipeline approach was unable to generate well-formed MRs for 6 methods. Here, a thorough analysis is left for future work. The increase from around 45% to around 75% for line coverage and branch coverage further shows the effectiveness of our approach. This is also strengthened by finding 14 new bugs as described in the previous section.

We carried out an additional analysis to identify MRs that achieve equivalent function, line, and branch coverages, but do not uncover additional bugs. This analysis identified 26 duplicated MRs. By removing these MRs, we found a total of 160 unique MRs generated with the help of our LLM-assisted approach. These are almost 8 times more MRs than the carefully handcrafted MRs defined in [13].

VII. Discussion

LLMs are transforming the way developers approach tasks, offering powerful capabilities out of the box. As a result, using off-the-shelf LLMs has emerged, simplifying adaptation but introducing trade-offs. Next to the resource demands, a trade-off lies in the lack of factual reliability (e.g. hallucinations).

Fine-tuning can address reliability issues but requires expertise in machine learning. Many verification engineers have yet to acquire this expertise, given the relatively short time since LLMs advanced to a stage where they can be applied across various domains. Furthermore, fine-tuning can be disadvantageous in this context because the LLM already entails an exhaustive understanding of the problem domain. For our case, fine-tuning the unique wedge line function of the TFT_eSPI library would require a large amount of specialized training data. Given that it would only enhance reliability for this specific case, the result does not justify the effort to further fine-tune the LLM.

An alternative approach to fine-tuning is the application of prompt engineering strategies combined with classical programming. This approach allowed us to use the unmodified Meta Llama 3.1 model, using its pre-existing data in graphics libraries, including TFT_eSPI. This generalization also makes our method applicable to other embedded graphics libraries. Moreover, combining prompt engineering with classical programming enables easy adaptation to alternative LLMs. If the model contains knowledge on the desired library and MT, minimal prompt adjustments, such as rephrasing terms, are sufficient. An example of such a rephrasing was given in Footnote 4 where replacing "specification" with "definition" improved the classifications for Llama 3.1.

Furthermore, we want to discuss the LLM-assisted MR pipeline. Within the pipeline, only the first three steps have been executed using the LLM for certain sub-tasks. Although these steps could potentially be condensed into a single step with the advancement of reasoning-capable models, we observed that current LLMs struggle to produce a large number of diverse MRs (output of Step 3). In practice, only around five unique MRs were generated before the models began to hallucinate exhaustively when tasked with generating more.

Step 4, which involves translating the generated MRs, was implemented purely with classical programming, as relying on LLMs for this step introduced a significant risk of mistranslation. Step 5, which involves parameter injection and executing the resulting MTCs on a VP-based framework, likewise does not require LLM support.

Despite the growing capabilities of LLMs, human oversight remains essential, as shown in Steps 6 and 7. While the model generated 61 MRs, manual review and refinement yielded an additional 125. One might argue that these steps could be automated using visual models, but further research is required to reliably distinguish between valid failures due to actual bugs and those caused by malformed MRs.

VIII. Conclusions

In this paper, we introduce a novel approach that uses LLMs to generate *Metamorphic Relation*s (MRs) for *Metamorphic Testing* (MT) of embedded graphics libraries. After assessing the LLM capabilities and assigning the developer role to the LLM, it was able to decompose the MR generation problem into 7 steps. Based on these steps, we structured and created an LLM-assisted MR pipeline, where we found that carefully selecting the right prompting strategy is crucial. For evaluation of our approach, we extended an existing MT framework with our LLM-assisted MR pipeline. We found that our approach was able to easily generate 8 times more unique MRs in comparison to the existing manual MT framework. Through a comprehensive assessment, we demonstrated that our MRs produced by the LLM identified 14 novel bugs and nearly doubled coverage.

Finally, we discussed that our prompt-engineering-based approach balances accessibility, adaptability, and generalization, tailoring it for verification engineers working on embedded graphics libraries. However, while our approach leverages

LLMs to generate effective MRs, it still requires human expertise. In future work, to further reduce the human effort necessary for Steps 6 and 7, an approach to include a visual model will be evaluated. Additionally, the methodologies presented in [43]–[46] will be incorporated to enable a more comprehensive system analysis.

Appendix

This appendix provides further implementation specifics concerning the MTC generators and the produced FW codes. The listings expand on the example presented in Listing 1.

A. MTC generator core

Listing 2 contains the Python script that generates the surrounding C++ code for the source and follow-up test cases. The setup code, including necessary libraries, the main function interface and initialization for the TFT_eSPI library, is defined in Line 3–10.

```
 1 def generate_file(self, src_path, testcase_fn):
 2     cpp = Generators.Util.file_gen.CppFile(src_path)
 3     cpp('#include "TFT_eSPI.h"')
 4     cpp('#include <cstdlib>')
 5     cpp('#include <algorithm>')
 6     cpp('using namespace std;')
 7 
 8     with cpp.block("int main()"):
 9         cpp("TFT_eSPI tft = TFT_eSPI();")
10         cpp("tft.init();")
11 
12         testcase_fn(cpp)
13 
14         cpp("tft.writecommand(0xFF);")
15         cpp(f"tft.writedata16({screenshot_num});")
16         cpp("std::exit(0);")
17 
18     cpp.close()
```

Listing 2: MTC generator core generating surrounding C++ code for test cases, including the library initialization and the screenshot generation.

The call to `testcase_fn` in Line 12 generates source or follow-up specific code. Line 14–16 include the necessary instructions to take a screenshot (using TFT write commands) and to shut down the VP.

B. MR specific MTC generator

Listing 3 contains the class responsible for generating the specific codes for the source (Line 3–6) and the follow-up (Line 9–14) test cases. The functions are called accordingly when Line 12 from Listing 2 is called. In Line 9, the code initializes `h` to 1 as specified by the MR from Listing 1. Line 10–14 then calls `drawRect`, replacing all parameters except `h` with dynamically generated Python variables during MTC execution.

C. Source FW

Listing 4 contains the generated source FW. Line 10 contains the call to `drawFastHLine` with filled-in parameters. Line 11 contains the initialization of parameter `h`, as well as the call to the `drawRect` function with, except `h`, the same parameters as Line 10 from Line 3.

```
 1 class drawFastHLinedrawRect(MTCGenerator):
 2     def source_testcase(self,cpp):
 3         cpp(f"tft.drawFastHLine({self.args['x']},
 4                                 {self.args['y']},
 5                                 {self.args['w']},
 6                                 {self.args['color']});")
 7 
 8     def follow_up_testcase(self, cpp):
 9         cpp(f"int32_t h = 1;")
10         cpp(f"tft.drawRect({self.args['x']},
11                            {self.args['y']},
12                            {self.args['w']},
13                            h,
14                            {self.args['color']});")
```

Listing 3: MTC generator class for DrawFastHLineDrawRect MR translated from LLM response into Python. The first function contains the code to generate the source testcase and the second function contains the code to generate the follow-up testcase.

```
 1 #include "TFT_eSPI.h"
 2 #include <cstdlib>
 3 #include <algorithm>
 4 using namespace std;
 5 int main()
 6 {
 7   TFT_eSPI tft = TFT_eSPI();
 8   tft.init();
 9 
10   tft.drawFastHLine(168, 156, 68, 10158409);
11 
12   tft.writecommand(0xFF);
13   tft.writedata16(1);
14   std::exit(0);
15 }
```

Listing 4: Generated source FW for DrawFastHLineDrawRect MR including library initialization and screenshot generation as well as the concrete source testcase.

D. Follow-up FW

The generated follow-up FW is provided in Listing 5. Parameter `h` is initialized in Line 10 and set to 1 as specified for this MR. The call to `drawRect` with the generated concrete values, matching the parameters from the source FW, is located in Line 11.

```
 1 #include "TFT_eSPI.h"
 2 #include <cstdlib>
 3 #include <algorithm>
 4 using namespace std;
 5 int main()
 6 {
 7   TFT_eSPI tft = TFT_eSPI();
 8   tft.init();
 9 
10   int32_t h = 1;
11   tft.drawRect(168, 156, 68, h, 10158409);
12 
13   tft.writecommand(0xFF);
14   tft.writedata16(2);
15   std::exit(0);
16 }
```

Listing 5: Generated follow-up FW for DrawFastHLineDrawRect MR including library initialization and screenshot generation as well as the concrete source testcase.

Acknowledgments

This work has partially been supported by the LIT Secure and Correct Systems Lab funded by the State of Upper Austria.

References

[1] J. Park, N. Baek, and H. Lee, "Design of a small footprint embedded graphics system," in *IEEE Global Conference on Consumer Electronics*, 2012, pp. 187–188.

[2] T. De Schutter, *Better Software. Faster!: Best Practices in Virtual Prototyping*. Synopsys Press, March 2014.

[3] *IEEE Standard for Standard SystemC Language Reference Manual*, IEEE Std. 1666 (Revision of IEEE Std 1666-2011), 2023.

[4] D. Große and R. Drechsler, *Quality-Driven SystemC Design*. Springer, 2010.

[5] V. Herdt, D. Große, and R. Drechsler, *Enhanced Virtual Prototyping: Featuring RISC-V Case Studies*. Springer, 2020.

[6] M. Hassan, D. Große, and R. Drechsler, *Enhanced Virtual Prototyping for Heterogeneous Systems*. Springer, 2022.

[7] P. C. Jorgensen, *Software testing: a craftsman's approach*. Auerbach Publ., 2013.

[8] A. Marrero Perez and S. Kaiser, "Integrating test levels for embedded systems," in *Testing: Academic and Industrial Conference - Practice and Research Techniques*, 2009, pp. 184–193.

[9] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 507–525, 2015.

[10] S. Segura, G. Fraser, A. B. Sanchez, and A. Ruiz-Cortés, "A survey on metamorphic testing," *IEEE Transactions on Software Engineering*, vol. 42, no. 9, pp. 805–824, 2016.

[11] T. Y. Chen *et al.*, "Metamorphic testing: A review of challenges and opportunities," *ACM Comput. Surv.*, vol. 51, no. 1, 2019.

[12] C. Hazott, F. Stögmüller, and D. Große, "Verifying embedded graphics libraries leveraging virtual prototypes and metamorphic testing," in *Asia and South Pacific Design Automation Conference*, 2024, pp. 275–281.

[13] ——, "Using virtual prototypes and metamorphic testing to verify the hardware/software-stack of embedded graphics libraries," *Integr.*, vol. 101, 2025.

[14] S. Schulhoff *et al.*, "The prompt report: A systematic survey of prompting techniques," 2024. [Online]. Available: https://arxiv.org/abs/2406.06608

[15] https://huggingface.co/meta-llama/Meta-Llama-3.1-8B.

[16] https://huggingface.co/docs/transformers/index.

[17] https://github.com/ics-jku/mt-graphlib-framework, 2023.

[18] R. Zhong *et al.*, "LLM4EDA: Emerging progress in large language models for electronic design automation," 2023. [Online]. Available: https://arxiv.org/abs/2401.12224

[19] J. Bhandari, J. Knechtel, R. Narayanaswamy, S. Garg, and R. Karri, "LLM-aided testbench generation and bug detection for finite-state machines," 2024. [Online]. Available: https://arxiv.org/abs/2406.17132

[20] W. Fang *et al.*, "AssertLLM: Generating and evaluating hardware verification assertions from design specifications via multi-LLMs," 2024. [Online]. Available: https://arxiv.org/abs/2402.00386

[21] Q.-H. Luu *et al.*, "Can chatgpt advance software testing intelligence? an experience report on metamorphic testing," 2023. [Online]. Available: https://arxiv.org/abs/2310.19204

[22] S. Y. Shin *et al.*, "Towards generating executable metamorphic relations using large language models," 2024. [Online]. Available: https://arxiv.org/abs/2401.17019

[23] Y. Zhang, D. Towey, and M. Pike, "Automated metamorphic-relation generation with chatgpt: An experience report," in *IEEE International Conference on Computers, Software, and Applications*, 2023, pp. 1780–1785.

[24] RepositoryStats, https://repositorystats.com/bodmer/tft_espi, 2025.

[25] N. Humfrey, https://www.arduinolibraries.info/, 2025.

[26] W. Mueller, M. Becker, A. Elfeky, and A. DiPasquale, "Virtual prototyping of cyber-physical systems," in *Asia and South Pacific Design Automation Conference*, 2012, pp. 219–226.

[27] F. Pêcheux, C. Grimm, T. Maehne, M. Barnasconi, and K. Einwich, "SystemC AMS based frameworks for virtual prototyping of heterogeneous systems," in *IEEE International Symposium on Circuits and Systems*, 2018, pp. 1–4.

[28] C. B. Aoun, L. Andrade, T. Maehne, F. Pêcheux, M.-M. Louërat, and A. Vachoux, "Pre-simulation elaboration of heterogeneous systems: The SystemC multi-disciplinary virtual prototyping approach," in *2015 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, 2015, pp. 278–285.

[29] V. Herdt, H. M. Le, D. Große, and R. Drechsler, "Verifying SystemC using intermediate verification language and stateful symbolic simulation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 7, pp. 1359–1372, 2019.

[30] V. Herdt, D. Große, H. M. Le, and R. Drechsler, "Verifying instruction set simulators using coverage-guided fuzzing," in *Design, Automation and Test in Europe Conference*, 2019, pp. 360–365.

[31] ——, "Early concolic testing of embedded binaries with virtual prototypes: A RISC-V case study," in *Design Automation Conference*, 2019, pp. 188:1–188:6.

[32] P. Pieper, V. Herdt, D. Große, and R. Drechsler, "Dynamic information flow tracking for embedded binaries using SystemC-based virtual prototypes," in *Design Automation Conference*, 2020, pp. 1–6.

[33] ——, "Verifying SystemC TLM peripherals using modern C++ symbolic execution tools," in *Design Automation Conference*, 2022, pp. 1177–1182.

[34] M. Schlägl, C. Hazott, and D. Große, "RISC-V VP++: Next generation open-source virtual prototype," in *Workshop on Open-Source Design Automation*, 2024.

[35] M. Schlägl and D. Große, "GUI-VP Kit: A RISC-V VP meets Linux graphics - enabling interactive graphical application development," in *ACM Great Lakes Symposium on VLSI*, 2023, pp. 599–605.

[36] M. Schlägl, M. Stockinger, and D. Große, "A RISC-V "V" VP: Unlocking vector processing for evaluation at the system level," in *Design, Automation and Test in Europe Conference*, 2024, pp. 1–6.

[37] M. Schlägl and D. Große, "Single instruction isolation for RISC-V vector test failures," in *IEEE/ACM International Conference on Computer-Aided Design*, 2024, pp. 156:1–156:9.

[38] ——, "Fast interpreter-based instruction set simulation for virtual prototypes," in *Design, Automation and Test in Europe Conference*, 2025, pp. 1–7.

[39] T. Y. Chen, D. Huang, T. Tse, and Z. Q. Zhou, "Case studies on the selection of useful relations in metamorphic testing," in *The Ibero-American Conference on Software Engineering and Knowledge Engineering*, 2004, pp. 569–583.

[40] https://openai.com/index/hello-gpt-4o/.

[41] https://huggingface.co/mistralai/Mistral-7B-Instruct-v0.3.

[42] https://pytorch.org/.

[43] C. Hazott and D. Große, "DSA monitoring framework for HW/SW partitioning of application kernels leveraging VPs," in *IEEE Design and Verification Conference and Exhibition Europe*, 2023, pp. 34–41.

[44] ——, "Relation coverage: A new paradigm for hardware/software testing," in *IEEE European Test Symposium*, 2024, pp. 1–4.

[45] ——, "Boosting SW development efficiency with function lifetime diagrams," in *IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems*, 2025, pp. 99–104.

[46] L. Klemmer and D. Große, "WAVING goodbye to manual waveform analysis in HDL design with WAL," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 43, no. 10, pp. 3198–3211, 2024.