Verifying Embedded Graphics Libraries leveraging Virtual Prototypes and Metamorphic Testing

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

Florian Stögmüller
Institute for Complex Systems
Johannes Kepler University
Linz, Austria
stoegmueller.f@gmail.com

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

***Abstract*—Embedded graphics libraries are part of the firmware of embedded systems and provide complex functionalities optimized for specific hardware. After unit testing of embedded graphics libraries, integration testing is a significant challenge, in particular since the hardware is needed to obtain the output image as well as the inherent difficulty in defining the reference result.**

In this paper, we present a novel approach focusing on integration testing of embedded graphic libraries. We leverage *Virtual Prototypes* (VPs) and integrate them with *Metamorphic Testing* (MT). MT is a software testing technique that uncovers faults or issues in a system by exploring how its outputs change under predefined input transformations, without relying on explicit oracles or predetermined results. In combination with virtualizing the displays in VPs, we even eliminate the need for physical hardware. This allows us to develop a MT framework automating the verification process. In our evaluation, we demonstrate the effectiveness of our MT framework. On an extended RISC-V VP for the GD32V platform we found 15 distinct bugs for the widely used TFT_eSPI embedded graphics library, confirming the strength our approach.

I. Introduction

In embedded systems, *Software* (SW) is closely tied to the *Hardware* (HW) it runs on. An important part of the embedded SW is the *Firmware* (FW) which provides low-level control for the device's specific HW. As many embedded systems include displays to visualize information as well as to enable easy interaction, FW libraries for these displays are very crucial components. Over the years, these FW libraries have become more and more powerful: One of the first implementations, where the term FW was already used, has been presented in [1]. This work interfaced different graphical terminals via the FW. Today, much more complex functionality is integrated in these *embedded graphics libraries*, extending the fundamental support of drawing simple geometric elements on different displays to advanced objects, fonts, and even features like sprites. Moreover, optimizations for different HW architectures are performed improving the rendering performance. Due to this increasing feature complexity, the importance of verification of embedded graphics libraries progressively amplifies.

To address the growing complexity, advancements in simulators and emulators are leveraged to enable the adaption of SW testing strategies for FW testing. The most fundamental strategy which is adopted is testing of individual components and functions in isolation, also referred to as unit testing [2]. While this approach is effective at uncovering numerous bugs, blind spots emerge because unit testing does not capture complex interactions among components, and overlooks integration challenges that can lead to functional and performance issues. *Integration testing* complements unit testing by focusing on these blind spots, capturing the interfaces and interactions among components [3], [4].

For successful integration testing on embedded devices, test inputs forcing the system into potential error cases have to be defined as well as comprehensive reference models are necessary to determine the test result. The latter may be very difficult to create as the effort to implement such models increases if more and more components (e.g. deeply layered functions) interact and if complex SW-to-HW stacks are involved. This challenge is well-known as *test oracle problem* [5] and in case of embedded graphics libraries it is even worse, as it typically means visual inspection of the results when executed on the HW. Altogether, this makes automating the verification very complicated.

Contribution: In this paper, we present a novel approach focusing on integration testing of embedded graphics libraries. As first component, *Virtual Prototypes* (VPs) are leveraged targeting the need of HW for visual inspection. VPs are predominantly modeled in SystemC, a standardized C++ library [6]; for a broader overview on SystemC we refer the reader to [7]–[9]. VPs enable the development and execution of SW production code as if the physical HW were present on the table [10].

The second essential component of our approach to face the test oracle problem is *Metamorphic Testing* (MT) [11]. MT circumvents the need of a reference model and found an impressive amount of bugs in different domains, like search engines, data engineering, compilers, machine learning programs, to just name a few [12]. MT is based on the core principle of using known relationships and properties among inputs and outputs to design effective test cases. These test cases enable the detection of potential faults or deviations in SW behavior through consistent transformations of input data. The relationships/properties have to be defined to form so-called *Metamorphic Relations* (MRs). Let us consider an example for an MR: Given an implementation of the sine function as the

program `impl_sin(x)`. As MR, we can use the periodicity of the sine function, i.e. we know $sin(x) = sin(180 - x)$. Now, instead of checking the expected output value for a concrete input `c`, we can check that `impl_sin(c)` equals `impl_sin(180-c)` so without having to know what the actual value of `impl_sin(c)` is.

For verification, we introduce multiple MRs tailored for embedded graphics libraries. For example, among the developed MRs we utilized the popular Eulerian path-finding problem "Haus vom Nikolaus" ("House of Santa Claus): the objective is to draw a house by connecting five points with exactly eight edges without traversing any edge twice. This can be done in multiple ways by varying the order in which the edges are drawn, but regardless of the method chosen, the resulting image of the house remains unchanged.

Based on the developed MRs, we devise an automated MT framework available on GitHub[1]. For each MR, we create a pair of FW (for example two different drawing sequences of the House of Santa Claus, one in each FW). Then, we leverage the VP to run each FW which allows determining whether the MR is satisfied or not utilizing a virtual display which can take screenshots.

To improve automation and effectiveness, we additionally implemented a generator that automates the creation of multiple *Metamorphic Test Cases* (MTCs) by randomizing the test case parameters. To validate our approach on a real system, we have chosen the popular GD32V platform from GigaDevice, including a display controller. As embedded graphics library we took the widely used *TFT_eSPI* and added appropriate support for the display controller. Our extensive evaluation shows the effectiveness of our MT framework. In total, we were able to unveil 15 bugs in the TFT_eSPI embedded graphics library. All bugs found, showed the exact same faulty behavior on both, the virtual and real GD32V system.

II. Related Work

MT has been extensively studied by researchers [12], [13] and successfully applied in various domains, e.g., graph algorithms [13], web services [14], simulation and modeling [15], machine learning [16], and compilers [17], [18]. Furthermore, MT has been utilized in industry by companies such as Facebook [19] and Adobe [20] to test their software. Related to our work, Donaldson et. al [21] used MT for finding bugs in compilers for graphics shading languages. Their work resulted in the tool *GraphicsFuzz* which has been acquired by Google and is used within the Android ecosystem for the graphic shader compiler.

There also have been several contributions to MT in the embedded domain (e.g., [22]–[26]) and in the graphics domain (e.g., [27]–[29]). In [30] the authors applied MT to the processor verification domain and conducted a case study using a RISC-V instruction set simulator.

For FW verification/testing several approaches have been presented, e.g. [31], [32] and [33]. The verification results are gathered either by collecting CPU states as well as transferred messages on the peripheral bus. However, these approaches do not cover graphical displays.

Recently, the concept of *Instruction-Level Abstraction* (ILA) has been introduced [34]. The idea is to abstract from HW details as much as possible and by this to further improve verification effectiveness. Although this approach allows to target the verification of HW/SW interaction, embedded graphic libraries producing complex visual output have not been considered and the challenge of reference models still remains.

III. Basic Formalizations

In general, MT relies on MRs which are used to (1) generate test cases and (2) to decide whether a test fails or passes. In this section, we start with some basic formalizations. This includes the formalization of FW in terms of method calls, the compilation and execution of FW, and fundamental principles for FW MRs.

1) FW: Let F be a FW consisting of a series of $n \geq 1$ method calls to an embedded graphics library (e.g. TFT_eSPI), denoted as $m_0, m_1, ..., m_{n-1}$. Furthermore, for a method call m_i, the tuple of its parameters is denoted as p_i. A tuple p_i contains $k \geq 1$ elements, where each element is an input parameter for the corresponding method call. Formally, we write:

$$F := \langle m_0(p_0), m_1(p_1), \ldots, m_{n-1}(p_{n-1}) \rangle$$

It is important to note that our FW definition only facilitates the definition of MRs and therefore does not provide executable code. How a runnable FW is generated is discussed in Section V-B.

2) FW Compilation and Execution: The process of compiling a FW F and running it can be viewed as a function:

$$compileAndRun : F \mapsto I$$

Given an input FW F, this function performs compilation, execution, and generates the output I, which, in the case of an embedded graphics library, corresponds to an image.

3) FW MR Principles: MT relies on executing two FW (or a FW pair) and compares both execution results (here images) via an MR. Throughout the rest of this paper, such a **FW pair** will consist of a *source FW* and *follow-up FW*. To test a particular functionality of an embedded graphics library, we need to define a relation between source FW and follow-up FW and their corresponding output images. Although it is possible to do this by transforming both, the input and the output, in this paper we focus on transforming the input. That is, we define MRs by transforming only the FW, not the resulting images.

A FW F can be transformed using any combination of the following four modifications:

- Removing method calls
- Adding method calls
- Reordering method calls
- Manipulating the parameters

[1]https://github.com/ics-jku/mt-graphlib-framework

Transforming a FW F results in a new FW $\widehat{F}$. We can formalize this by defining a transformation function:

$$\psi(F) = \widehat{F}$$

By applying this transformation function to the source FW F, we generate the follow-up FW $\widehat{F}$. ψ may either preserve or break the semantics of F. Therefore, the resulting FW $\widehat{F}$ either shows equivalent behavior or non-equivalent behavior compared to the original FW F.

Depending on this equivalence, we can distinguish between two general types of MRs for our embedded graphics library verification problem:

- If ψ is semantics-preserving, calling $compileAndRun$ with F and $\widehat{F}$ should yield the same image. The resulting MR is based on the equality relation and can be formalized as follows:

  $$F \equiv \widehat{F} \Rightarrow compileAndRun(F) = compileAndRun(\widehat{F}).$$

- On the other hand, if ψ breaks the semantics, calling $compileAndRun$ with F and $\widehat{F}$ should yield different images. Therefore, the resulting MR is based on the inequality relation and can be defined as follows:

  $$F \not\equiv \widehat{F} \Rightarrow compileAndRun(F) \neq compileAndRun(\widehat{F}).$$

In summary, we can define an MR for an embedded graphics library by defining a pair of either equivalent or non-equivalent FW, where a FW is simply a series of method calls.

IV. Definition of MRs

Embedded graphics libraries such as TFT_eSPI offer functionalities to create various visual elements ranging from basic geometric shapes to more complex objects, fonts, and sprites. To illustrate the transformations stated in Section III, we discuss 4 representative MRs out of 21 we developed in the following. Typically, the creation of MRs is guided by complexity considerations, where the number of nested functional calls and the number of involved components within the SW-to-HW stack incrementally grows. This allows to use the first and easier-to-develop MRs rapidly in our MT framework. We introduce each of the 4 MRs by first explaining which library function it targets from a generic perspective. Then, we show the function signature as provided by the *Application Programming Interface* (API) specification of the TFT_eSPI library. Based on both, we define the MR which relates two FW using the basic formalizations from Section III while describing the idea behind.

A. MR: DrawPixel

To draw a single pixel on a display, an embedded graphics library provides a respective method. In case of TFT_eSPI, its API contains:

```
1 void drawPixel(int32_t x, int32_t y, uint32_t
      color);
```

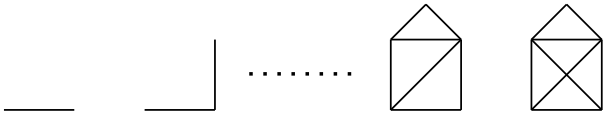


Fig. 1: "Haus vom Nikolaus (Haus of Santa Clause)".

The idea of our **DrawPixel** MR is to test whether individual pixels are drawn correctly regardless of the order in which they are drawn (assuming that no two pixels are drawn on the exact same positions). A straight forward transformation of a given source FW into a follow-up FW is to reverse the method calls of the source FW. As MR we define:

$$\begin{gathered}\langle drawPixel(x_0, y_0, c_0), drawPixel(x_1, y_1, c_1), \ldots, \\ drawPixel(x_{n-1}, y_{n-1}, c_{n-1})\rangle \\ \equiv \\ \langle drawPixel(x_{n-1}, y_{n-1}, c_{n-1}), drawPixel(x_{n-2}, y_{n-2}, c_{n-2}), \ldots, \\ drawPixel(x_0, y_0, c_0)\rangle.\end{gathered}$$

B. MR: WedgeLine

The next MR we define is the **WedgeLine**. A line is drawn starting from coordinates a and ending at coordinates b. A wedge line is a special case where we can also define the width aw of the line at the start point and the width bw of the line at the end point.

In case of the TFT_eSPI library, the API provides:

```
1 void drawWedgeLine(float ax, float ay, float bx,
      float by, float aw, float bw, uint32_t
      fg_color, uint32_t bg_color = 0x00FFFFFF);
```

The idea of the WedgeLine MR is to change the order of parameters[2]. In the case of a line we can do this by switching start and end points a and b. For the wedge line we additionally have to switch the parameters aw and bw.

$$\begin{gathered}\langle drawWedgeLine(ax, ay, bx, by, aw, bw, c_{fg}, c_{bg})\rangle \\ \equiv \\ \langle drawWedgeLine(bx, by, ax, ay, bw, aw, c_{fg}, c_{bg})\rangle\end{gathered}$$

C. MR: Nikolaus

We now discuss a representative of more complex MRs. The purpose behind the **Nikolaus** MR is to check the considerably intricate behavior exhibited by an embedded graphics library. Our idea was to create an algorithm that solves the famous Eulerian path-finding problem "Haus vom Nikolaus" ("House of Santa Claus"): as shown in Figure 1, the goal is to draw a simple house by connecting five points with exactly eight edges. The challenge is to draw it in one continuous line without traversing any edge twice. In total, there are 44 solutions, i.e., 44 different ways to draw the house using a continuous line.

For the MR, we used eight method calls and permuted them according to the 44 solutions which we computed in advance.

[2]The careful reader may have noticed that the coordinates are defined as floats. According to the library specification this is done to support sub-pixel calculation which is supported for some displays. Sub-pixels allow to address the color channels of a pixel and so visually make the pixel appear as having an offset which is smaller than one pixel. The technology used in our evaluation also supports color channels, meaning we also cover this feature.

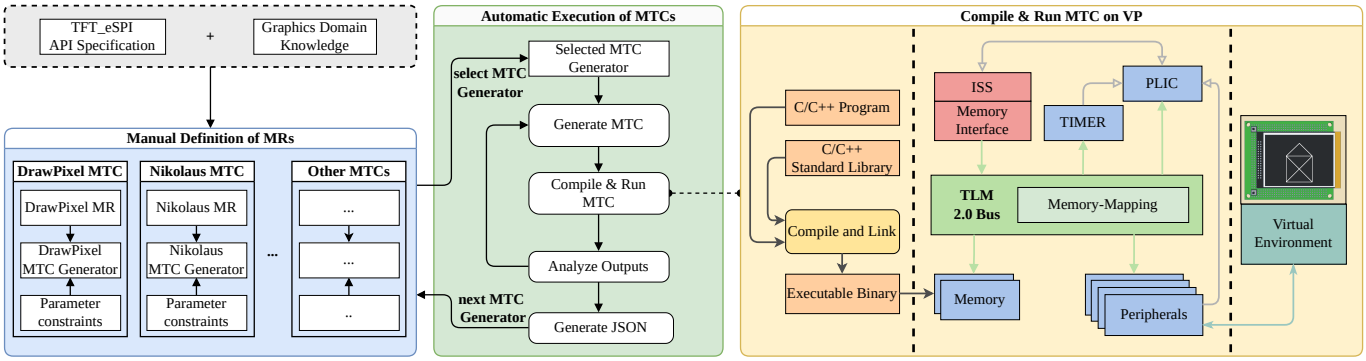


Fig. 2: Automated MT Framework

D. MR: NikolausMove

This MR is a sprite-based variation of the *Nikolaus* MR. The eight lines that make up the house are drawn onto a single sprite (a two-dimensional bitmap), which is then pushed to the screen. Again, the order in which the lines are drawn is different for the source FW and the follow-up FW. Additionally, in the follow-up FW, the sprite is moved along a rectangular path (e.g., left, down, right, and up), eventually returning to its original position. This sequence of movements and the order in which the edges of the house are drawn has no effect on the final output and forms the **NikolausMove** MR.

V. Automated MT Framework

In this section, we present our automated MT framework based on the developed MRs. We start with an overview in Section V-A. Thereafter, we give details about how we implemented the functionality to generate MTCs. We conclude this section by presenting how MTC execution is automated within our automated MT framework.

A. Overview

Fig. 2 provides an overview of our automated MT framework. The framework consists of several parts. In the upper left part (gray box), we consult the API specification of an embedded graphics library, in our case TFT_eSPI, and combine this information with general knowledge from the graphics domain. The result serves as basis to define the MRs (cf. blue box). As can be seen in Fig. 2, we list some MRs already defined in the previous section. In addition, we need to implement a generator for each MR within our MT framework. This step is essential due to the fact that the MRs we have defined thus far are not directly executable. More precisely, according to a given MR the generator will produce two FW, i.e. a source FW and a follow-up FW which becomes executable by setting concrete values for the (method) parameters. Thereby, also constraints on parameters (e.g. value ranges) are taken into account. These two executable FW constitute a *Metamorphic Test Case* (MTC). Therefore, the generator is denoted overall as *MTC Generator* and will be described in more detail in the following subsection. Moreover, since the generation is automated, multiple MTCs per MR can be generated.

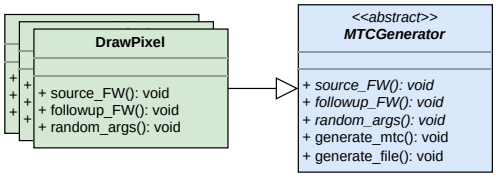


Fig. 3: MTC Generator: Blue colored box shows abstract `MTCGenerator` class. Green colored boxes indicate concrete implementations

The main part of our MT framework (green box in the center), which is fully automated, begins with selecting and starting an MTC Generator. The generated source FW and follow-up FW of an MTC are then compiled and executed on the VP. In Fig. 2, the VP is depicted within the yellow box on the right. The VP consists of three pieces, separated by dashed lines, covering the full SW to HW stack. The leftmost piece, summarizes the main steps for compiling FW. The center shows the main components of the VP which are used to execute FW during VP simulation. On the right side of the yellow box, the virtual display can be found as part of the Virtual Environment. Recall, that we have extended the VP and created the virtual display such that we can capture an image, i.e. the output produced by the FW.

Once both FW of an MTC have been compiled and executed, the MR is checked by analyzing the outputs, i.e., comparing the images (see *Analyze Outputs* in the middle of the green box in Fig. 2). If the current MTC has passed, a new MTC is generated if a given timeout limit has not been reached. If the current MTC failed, we either continue to generate additional MTCs for the same MR, or we continue with another MR. Throughout MTC execution, our framework keeps track of various metrics and stores them in a JSON file once all MTCs for an MR are complete. The following subsections provide a detailed description for the MTC generators and the automatic MTC execution.

B. MTC Generators

To produce MTCs from our MRs we use so-called MTC generators. Fig. 3 provides an architectural overview of the MTC generators. The `generate_mtc()` method is the main entry point, which gets called by the framework. The

```
1  #include "TFT_eSPI.h"
2  int main() {
3      TFT_eSPI tft = TFT_eSPI();
4      tft.init();
5      tft.drawPixel(34,66,47586);
6      tft.drawPixel(357,21,60570);
7      tft.drawPixel(203,326,54515);
8      tft.writecommand(0xFF);
9      tft.writedata16(1);
10     return 0;
11 }
```

Listing 1: Generated source FW for a DrawPixel MTC

```
1  #include "TFT_eSPI.h"
2  int main() {
3      TFT_eSPI tft = TFT_eSPI();
4      tft.init();
5      tft.drawPixel(203,326,54515);
6      tft.drawPixel(357,21,60570);
7      tft.drawPixel(34,66,47586);
8      tft.writecommand(0xFF);
9      tft.writedata16(2);
10     return 0;
11 }
```

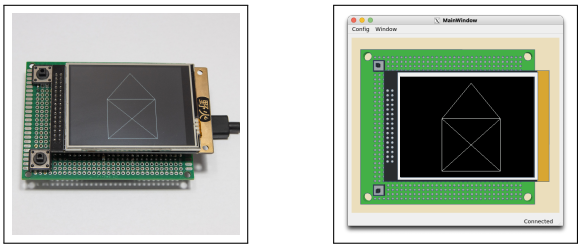
Listing 2: Generated follow-up FW for a DrawPixel MTC

flow to generate an MTC is the same for all MTC generators. Thus, this functionality is implemented in the abstract `MTCGenerator` class and is inherited by the concrete generator classes. The second method where the functionality stays the same across MTC generators is the `generate_file()` function, which writes the resulting source FW and follow-up FW code into an output file.

The `random_args()` method is implemented within the concrete classes. This function is responsible to generate random parameters for the source FW. These parameter values are limited by constraints which define ranges of valid values for each parameter. The `source_FW()` and `followup_FW()` methods contain the template to generate the corresponding code. The output of our MTC generator are two C/C++ files, for source FW and follow-up FW, which can be compiled and run on a VP.

C. Automatic MTC Execution

Based on the generators, we can create multiple MTCs, run them on the VP and check whether an MTC passes or fails. The core of the automation builds a so-called **MTCRunner**. This runner starts by selecting an MTC generator and calling `generate_mtc()`. As already said, the output of the generator are two FW code files. Listing 1 shows the source FW code and Listing 2 show the follow-up FW code for a **DrawPixel** MTC. Both figures show on Lines 3-4 the initialization of the TFT_eSPI library. This is common for generated code. In Lines 5-7 of both figures, we see the method calls for the **DrawPixel** MR as defined in Section IV-A. Listing 1 calls three times the `drawPixel` method with randomized parameters. Listing 2, the follow-up FW, calls the same method with the same parameters but in reversed order. Lines 8-9 in both figures are commands we added to the SW-to-HW stack to instruct the virtual environment to save the output.



(a) Real Environment (b) Virtual Environment

Fig. 4: Real and Virtual Environments

We have selected $0xFF$ for the built-in `writecommand()` method. According to the API specification, this command has no functionality for real hardware. The parameter given to the `writedata16()` method is indicating if the current execution is done by a source FW (1) or a follow-up FW (2).

Both files are compiled separately into executable FW binaries ready to be executed on the VP (and unmodified on real HW). To reduce compile time, the TFT_eSPI library is only compiled once and linked to all compiled FW. After the execution of an MTC, we get a source image and a follow-up image. To determine whether the two images are identical, they are compared pixel by pixel.

VI. Evaluation

To demonstrate the effectiveness of our developed MT framework and to assess its performance, we conducted a comprehensive evaluation in this section. We start by describing the steps we have taken to set up our environment and afterwards discuss the results of our evaluation.

A. Experimental Setup

The TFT_eSPI library is built upon the Arduino framework and uses an SPI protocol to communicate with a display. Since the TFT_eSPI library does not natively support the RISC-V GD32V board, we opted to port the current version 2.5.22 to the GD32V environment and switch to the faster 16-bit 8080 parallel interface.

To compile and run our MTCs we enhanced the RISC-V VP from [35], [36] to support the GD32V and added a virtual display performing parallel communication as mentioned above. The enhancements are available in the open-source *RISC-V VP++* on GitHub[3] together with general improvements [37]. Fig. 4a shows a photo of the real system (the RISC-V GD32 cannot be seen as it is below the circuit board). Fig. 4b shows the interface of the virtual display which connects to the RISC-V VP. Additionally, we had to set the criteria for our framework which indicates if all MTCs for an MR are executed. To capture multiple bugs for a single MR, we want to continue even if an MTC failed. We configured this and also set a timeout limit of 4 hours per MR.

All evaluations were carried out on an Intel Core i7-10700 CPU @ 2.90GHz (8 cores) machine with 64 GB of main memory running Ubuntu 20.04.5 LTS.

[3]https://github.com/ics-jku/riscv-vp-plusplus

TABLE I: Evaluation results

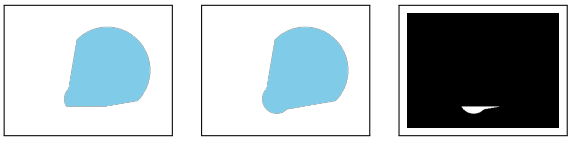
MR	MTCs	Failed	Error Rate	Bugs	Avg. RT
DrawPixel	4,465	0	0.00	-	2.85
FillScreen	3,595	0	0.00	-	3.71
Println	4,016	0	0.00	-	3.29
Rotation	3,502	0	0.00	-	3.80
WedgeLine	817	24	2.94	2	17.36
FrameViewport	4,599	349	7.59	2	2.84
ViewportUnequal	4,549	0	0.00	-	2.87
Nikolaus	4,562	0	0.00	-	2.86
NikolausShapes	4,563	0	0.00	-	2.86
NikolausMove	3,217	111	3.45	1	4.08
NikolausSprite	4,445	0	0.00	-	2.91
DrawRectangle	3,687	3,234	87.71	1	3.24
FillRectangle	4,386	1,874	42.73	1	2.96
DrawEllipse	4,552	31	0.68	1	2.86
FillEllipse	4,417	4,276	96.81	1	2.96
DrawCircle	4,558	4,506	98.86	1	2.86
FillCircle	4,392	4,371	99.52	1	2.98
DrawSmoothCircle	4,372	0	0.00	-	2.96
FillSmoothCircle	4,210	3,559	84.54	1	3.05
DrawArc	3,997	2,065	51.66	3	3.24
DrawArcSegments	4,240	0	0.00	-	2.95

B. Result Summary

Table I summarizes the results obtained with our MT framework for the TFT_eSPI embedded graphics library. For every MR, the table lists the name of the MR (column *MR*), the total number of MTCs that were executed in (column *MTCs*) and the corresponding number of failed MTCs (column *Failed*). To facilitate the comparison of failed tests across different MTCs, we included a column showing the relative error rate (column *Error Rate*). Furthermore, we list the number of distinct bugs a given MR was able to detect. The last column *Avg. RT* provides information about the average runtime per MR in seconds.

As can be seen, the number of MTCs executed per MR correlates with the average run times. Typically, around 4,300 MTCs were generated for most MRs. However, due to the complexity of the `drawWedgeLine` library implementation, the *WedgeLine* MR is an outlier, with only 817 MTCs generated within the 4h time limit.

Combined with the absolute and relative number of failed MTCs, the number of distinct bugs found by a MR is an important metric. It serves as an indicator of how effective an MR is at detecting bugs in the TFT_eSPI library and how easily bugs can be triggered. However, we have to be careful when looking at the error rates as they are dependent on the test case parameter constraints we defined for the MR. For example, if a particular bug is only triggered when a certain method parameter is negative, and the allowed range for that parameter contains more negative values than positive ones, we will get a high error rate. Also, note that due to the random generation of the method parameters and the constraints including invalid values, some MTCs were generated where the shapes were drawn outside the visible area or not drawn at all. Since this resulted in two completely empty and identical images, such MTCs were considered to have passed. Therefore, all MRs contain at least some successful tests.



(a) Source FW (b) Follow-up FW (c) Difference

Fig. 5: `drawWedgeLine` Bug

We ran MTCs for 21 different MRs and found that for 10 of them all MTCs passed. In these cases, we have a high degree of confidence that the library satisfies the particular property being tested. However, for the remaining 11 MRs some MTCs failed, revealing bugs in the library with some of these MTCs exposing even multiple bugs. In total, our framework discovered 15 unique bugs. We selected a failing MTC for each bug and ran both FW on the GD32V hardware board. This allowed us to confirm that all the bugs also occurred on the real hardware.

C. Discussion of Bugs

In the following we discuss two particularly interesting bugs found by our MT framework in TFT_eSPI library.

WedgeLine: Executing this MR revealed two bugs in the `drawWedgeLine` method. One of those bugs is shown in Fig. 5 and is the most severe bug found by our approach. When calling the method with a specific combination of valid arguments, the resulting wedge line is incomplete. We found this issue to be infrequent, occurring only twice out of 817 MTCs, which suggests that the bug is only triggered by very specific arguments.

NikolausMove: This complex MR revealed a bug in the sprite functionality. For the follow-up FW, the house, which was drawn using the sprite's `drawRect` method, was moved left, down, right, and up, ultimately returning it to its original position. While moving, the sprite leaves a trail that should have the same color as the background and thus be invisible. However, if the color parameter of `drawRect` exceeds the 16-bit maximum of the display, the colors of the trail and the background differ slightly. The constraint for the color parameter was set just slightly above 65535, therefore, most tests were generated with a valid color parameter value and the bug was triggered in only 3.45% of all MRs.

VII. Discussion

Our experiments demonstrated the effectiveness of our MT framework for finding bugs in an embedded graphics library. The proposed framework focuses on integration testing and complements unit testing. As such our approach targets in particular the verification of the interactions of components which includes deeply nested functions spanning the full HW/SW stack. Our MT framework leverages VPs and MT which allows to automate the verification process. For this, our framework uses multiple MRs. These MRs enable automatic generation of powerful tests without having to create a reference model (or logic which decides the correctness of the result). Instead, our generator produces two FW, i.e. the source FW and a follow-up FW forming a metamorphic test case, and by comparing

the execution results of both FW with respect to the MR we determine whether the test passed of failed.

When considering the concrete verification results obtained, we see that we have found 15 new bugs in TFT_eSPI, an embedded graphics library which is in the field for years. The automation of our framework which employs randomization during MTC generation (controlled by constraints) allows (1) to generate diverse tests in comparison to manually created function calls and (2) to check the results easily as the oracle becomes obsolete via the MR comparison. As becomes evident from the found bugs, MT is very viable here as a manual inspection of images on a display is too costly, and for non-trivial API functions, as targeted by the presented MRs, the creation of a reference model is virtually impossible.

VIII. Conclusions

In this paper, we have presented a novel approach of verifying embedded graphics libraries by leveraging VPs and MT. Our approach complements unit testing and focuses in particular on integration testing. We came up with a basic formalization to define MRs and tailored them for embedded graphic libraries. To generate and execute MTCs based on our MRs, we developed an automated MT framework. Our framework leverages an enhanced RISC-V VP modeling the GD32V platform for execution of the source FW and follow-up FW. With our approach, we were able to find 15 distinct bugs in the widely used TFT_eSPI library and confirmed all bugs on real HW.

In future work, we plan to integrate a detailed test analysis based on dynamic instrumentation techniques as presented in [38].

Acknowledgments

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

References

[1] G. A. Rose, ""intergraphic," a microprogrammed graphical-interface computer," *IEEE Trans. on Electronic Comp.*, vol. EC-16, no. 6, pp. 773–784, 1967.

[2] P. Liggesmeyer and M. Trapp, "Trends in embedded software engineering," *IEEE Software*, vol. 26, no. 3, pp. 19–25, 2009.

[3] A. Marrero Perez and S. Kaiser, "Integrating test levels for embedded systems," in *TAICPART*, 2009, pp. 184–193.

[4] C. Kaner and R. L. Fiedler, *Foundations of Software Testing*. Cengage Learning, 2013.

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

[6] *IEEE Standard SystemC Language Reference Manual*, IEEE Std. 1666, 2011.

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

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

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

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

[11] S. Segura and Z. Q. Zhou, "Metamorphic testing 20 years later: A hands-on introduction," in *ICSE*, 2018, pp. 538–539.

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

[13] T. Y. Chen, F.-C. Kuo, H. Liu, P.-L. Poon, D. Towey, T. H. Tse, and Z. Q. Zhou, "Metamorphic testing: A review of challenges and opportunities," *ACM Comput. Surv.*, vol. 51, no. 1, 2019.

[14] Z. Q. Zhou, S. Xiang, and T. Y. Chen, "Metamorphic testing for software quality assessment: A study of search engines," *TSE*, vol. 42, no. 3, pp. 264–284, 2016.

[15] C. Murphy, M. S. Raunak, A. King, S. Chen, C. Imbriano, G. Kaiser, I. Lee, O. Sokolsky, L. Clarke, and L. Osterweil, "On effective testing of health care simulation software," in *SEHC*. ACM, 2011, p. 40–47.

[16] X. Xie, J. W. Ho, C. Murphy, G. Kaiser, B. Xu, and T. Y. Chen, "Testing and validating machine learning classifiers by metamorphic testing," *JSS*, vol. 84, no. 4, pp. 544–558, 2011.

[17] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," *SIGPLAN Not.*, vol. 49, no. 6, p. 216–226, 6 2014.

[18] Q. Tao, W. Wu, C. Zhao, and W. Shen, "An automatic testing approach for compiler based on metamorphic testing technique," in *ASPEC*, 2010, pp. 270–279.

[19] J. Ahlgren, M. Berezin, K. Bojarczuk, E. Dulskyte, I. Dvortsova, J. George, N. Gucevska, M. Harman, M. Lomeli, E. Meijer, S. Sapora, and J. Spahr-Summers, "Testing web enabled simulation at scale using metamorphic testing," in *IEEE/ACM ICSE-SEIP*, 2021, pp. 140–149.

[20] D. C. Jarman, Z. Q. Zhou, and T. Y. Chen, "Metamorphic testing for adobe data analytics software," in *MET*, 2017, pp. 21–27.

[21] A. F. Donaldson, H. Evrard, A. Lascu, and P. Thomson, "Automated testing of graphics shader compilers," *PACMPL*, vol. 1, no. OOPSLA, 10 2017.

[22] M. Hassan, D. Große, and R. Drechsler, "System-level verification of linear and non-linear behaviors of RF amplifiers using metamorphic relations," in *ASP-DAC*, 2021, pp. 761–766.

[23] ——, "System level verification of phase-locked loop using metamorphic relations," in *DATE*, 2021, pp. 1378–1381.

[24] T. Tse and S. Yau, "Testing context-sensitive middleware-based software applications," in *COMPSAC*, 2004, pp. 458–466 vol.1.

[25] F.-C. Kuo, T. Y. Chen, and W. K. Tam, "Testing embedded software by metamorphic testing: A wireless metering system case study," in *LCN*, 2011, pp. 291–294.

[26] M. Jiang, T. Y. Chen, F.-C. Kuo, and Z. Ding, "Testing central processing unit scheduling algorithms using metamorphic testing," in *ICSESS*, 2013, pp. 530–536.

[27] J. Mayer and R. Guderlei, "On random testing of image processing applications," in *QSIC*, 2006, pp. 85–92.

[28] W. K. Chan, J. C. F. Ho, and T. H. Tse, "Finding failures from passed test cases: improving the pattern classification approach to the testing of mesh simplification programs," *STVR*, vol. 20, no. 2, pp. 89–120, 2010.

[29] F.-C. Kuo, S. Liu, and T. Y. Chen, "Testing a binary space partitioning algorithm with metamorphic testing," in *SAC*. ACM, 2011, pp. 1482–1489.

[30] F. Riese, V. Herdt, D. Große, and R. Drechsler, "Metamorphic testing for processor verification: A RISC-V case study at the instruction level," in *VLSI-SoC*, 2021, pp. 1–6.

[31] L. McMinn and J. Butts, "A firmware verification tool for programmable logic controllers," in *Critical Infrastructure Protection VI*. Springer, 2012, pp. 59–69.

[32] S. Ahn and S. Malik, "Automated firmware testing using firmware-hardware interaction patterns," in *CODES+ISSS*. ACM, 2014.

[33] B. Feng, A. Mera, and L. Lu, "P2IM: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling," in *USENIX Security*. USENIX Association, Aug. 2020, pp. 1237–1254.

[34] B.-Y. Huang, H. Zhang, P. Subramanyan, Y. Vizel, A. Gupta, and S. Malik, "Instruction-level abstraction (ila): A uniform specification for system-on-chip (soc) verification," *TODAES*, vol. 24, no. 1, 2018.

[35] V. Herdt, D. Große, H. M. Le, and R. Drechsler, "Extensible and configurable RISC-V based virtual prototype," in *FDL*, 2018, pp. 5–16.

[36] V. Herdt, D. Große, P. Pieper, and R. Drechsler, "RISC-V based virtual prototype: An extensible and configurable platform for the system-level," *JSA*, vol. 109, p. 101756, 2020.

[37] M. Schlägl and D. Große, "GUI-VP Kit: A RISC-V VP meets Linux graphics - enabling interactive graphical application development," in *GLSVLSI*, 2023, pp. 599–605.

[38] C. Hazott and D. Große, "DSA monitoring framework for HW/SW partitioning of application kernels leveraging VPs," in *DVCon Europe*, 2023.