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 graphics 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 $\text{impl}\_\sin(x)$. 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 $\text{impl}\_\sin(c)$ equals $\text{impl}\_\sin(180-c)$ so without having to know what the actual value of $\text{impl}\_\sin(c)$ 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 $m_0, m_1, \ldots, m_{n-1}$. 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:

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

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:

$$\text{compileAndRun}: F \rightarrow I$$

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 $\hat{F}$. We can formalize this by defining a transformation function:

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

By applying this transformation function to the source FW $F$, we generate the follow-up FW $\hat{F}$. $\psi$ may either preserve or break the semantics of $F$. Therefore, the resulting FW $\hat{F}$ 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 $\psi$ is semantics-preserving, calling `compileAndRun` with $F$ and $\hat{F}$ should yield the same image. The resulting MR is based on the equality relation and can be formalized as follows:

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

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

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

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:

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

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:

$$\langle drawPixel(x_0, y_0, c_0), drawPixel(x_1, y_1, c_1), \ldots, drawPixel(x_{n-1}, y_{n-1}, c_{n-1}) \rangle$$

$$\equiv \langle drawPixel(x_{n-1}, y_{n-1}, c_{n-1}), drawPixel(x_{n-2}, y_{n-2}, c_{n-2}), \ldots, drawPixel(x_0, y_0, c_0) \rangle.$$
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.

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
#include "TFT_eSPI.h"

int main() {
  TFT_eSPI tft = TFT_eSPI();
tft.init();
tft.drawPixel(34,66,47586);
tft.drawPixel(357,21,60570);
tft.drawPixel(203,326,54515);
tft.writecommand(0xFF);
tft.writedata16(1);
  return 0;
}

Listing 1: Generated source FW for a DrawPixel MTC

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

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 MTC 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 GD32 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 MR are executed. To capture multiple bugs for a single MR, we want to continue even if an MTC failed. We configured MR to have 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/fcs-jku/riscv-vp-plusplus
Table I: Evaluation results

<table>
<thead>
<tr>
<th>MR</th>
<th>MTCs</th>
<th>Failed</th>
<th>Error Rate</th>
<th>Bugs</th>
<th>Avg. RT</th>
</tr>
</thead>
<tbody>
<tr>
<td>DrawPixel</td>
<td>4,465</td>
<td>0</td>
<td>0.00</td>
<td>-</td>
<td>2.85</td>
</tr>
<tr>
<td>FillScreen</td>
<td>3,595</td>
<td>0</td>
<td>0.00</td>
<td>-</td>
<td>3.71</td>
</tr>
<tr>
<td>PrintIn</td>
<td>4,016</td>
<td>0</td>
<td>0.00</td>
<td>-</td>
<td>3.29</td>
</tr>
<tr>
<td>Rotation</td>
<td>3,502</td>
<td>0</td>
<td>0.00</td>
<td>-</td>
<td>3.80</td>
</tr>
<tr>
<td>WedgeLine</td>
<td>817</td>
<td>25</td>
<td>2.94</td>
<td>2</td>
<td>17.36</td>
</tr>
<tr>
<td>FrameViewport</td>
<td>4,599</td>
<td>349</td>
<td>7.59</td>
<td>2</td>
<td>2.84</td>
</tr>
<tr>
<td>ViewportUnequal</td>
<td>4,549</td>
<td>0</td>
<td>0.00</td>
<td>-</td>
<td>2.87</td>
</tr>
<tr>
<td>Nikolaus</td>
<td>4,562</td>
<td>0</td>
<td>0.00</td>
<td>-</td>
<td>2.86</td>
</tr>
<tr>
<td>NikolausShapes</td>
<td>4,563</td>
<td>0</td>
<td>0.00</td>
<td>-</td>
<td>2.86</td>
</tr>
<tr>
<td>NikolausMove</td>
<td>3,217</td>
<td>111</td>
<td>3.45</td>
<td>1</td>
<td>4.08</td>
</tr>
<tr>
<td>NikolausSprite</td>
<td>4,445</td>
<td>0</td>
<td>0.00</td>
<td>-</td>
<td>2.91</td>
</tr>
<tr>
<td>DrawRectangle</td>
<td>3,687</td>
<td>3,234</td>
<td>87.71</td>
<td>1</td>
<td>3.24</td>
</tr>
<tr>
<td>FillEllipse</td>
<td>4,386</td>
<td>1,874</td>
<td>42.73</td>
<td>1</td>
<td>2.96</td>
</tr>
<tr>
<td>DrawEllipse</td>
<td>4,552</td>
<td>31</td>
<td>0.68</td>
<td>1</td>
<td>2.86</td>
</tr>
<tr>
<td>FillEllipse</td>
<td>4,417</td>
<td>4,276</td>
<td>96.81</td>
<td>1</td>
<td>2.96</td>
</tr>
<tr>
<td>DrawCircle</td>
<td>4,558</td>
<td>4,506</td>
<td>98.86</td>
<td>1</td>
<td>2.98</td>
</tr>
<tr>
<td>FillCircle</td>
<td>4,392</td>
<td>4,371</td>
<td>99.52</td>
<td>1</td>
<td>2.98</td>
</tr>
<tr>
<td>DrawsmoothCircle</td>
<td>4,372</td>
<td>0</td>
<td>0.00</td>
<td>-</td>
<td>2.96</td>
</tr>
<tr>
<td>FillsmoothCircle</td>
<td>4,210</td>
<td>3,559</td>
<td>84.54</td>
<td>1</td>
<td>3.05</td>
</tr>
<tr>
<td>DrawArc</td>
<td>3,997</td>
<td>2,065</td>
<td>51.66</td>
<td>3</td>
<td>3.24</td>
</tr>
<tr>
<td>DrawArcSegments</td>
<td>4,240</td>
<td>0</td>
<td>0.00</td>
<td>-</td>
<td>2.95</td>
</tr>
</tbody>
</table>

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.

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 or 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 MT. Our approach complements unit testing and focuses in particular on integration testing. We came up with a basic MT. Our approach complements unit testing and focuses in particular on integration testing.

ACKNOWLEDGMENTS

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

REFERENCES