Automated Display Kernel Validation

Major Project Report

Submitted in partial fulfillment of the requirements
for the degree of

Master of Technology
in
Electronics & Communication Engineering
(Embedded Systems)

By

Harsh Kotak
(13MECE08)

Electronics & Communication Engineering Branch
Electrical Engineering Department
Institute of Technology
Nirma University
Ahmedabad-382 481

May 2015

Automated Display Kernel Validation

Major Project Report

Submitted in partial fulfillment of the requirements
for the degree of

Master of Technology
in
Electronics & Communication Engineering
(Embedded Systems)

By

Harsh Kotak
(13MECE08)

Under the guidance of

External Project Guide:

Mr. Avinash Reddy Palleti
Graphics Software Engineer,
Intel Technology India Pvt. Ltd.,
Bangalore.

Internal Project Guide:

Dr. N. P. Gajjar
Professor,
EC Department,
Institute of Technology,
Nirma University, Ahmedabad.

Electronics & Communication Engineering Branch
Electrical Engineering Department
Institute of Technology
Nirma University
Ahmedabad-382 481

May 2015

Declaration

This is to certify that

a. The thesis comprises my original work towards the degree of Master of Technology in Embedded Systems at Nirma University and has not been submitted elsewhere for a degree.

b. Due acknowledgment has been made in the text to all other material used.

- Harsh Kotak

Disclaimer

"The content of this project report does not represent the technology, opinions, beliefs or positions of Intel Technology India Pvt. Ltd., its employees, vendors, customers, or associates."

Certificate

This is to certify that the Major Project entitled **"Automated Display Kernel Validation "** submitted by **Harsh Kotak (13MECE08)**, towards the partial fulfillment of the requirements for the degree of Master of Technology in Embedded Systems, Nirma University, Ahmedabad is the record of work carried out by him under our supervision and guidance. In our opinion, the submitted work has reached a level required for being accepted for examination.The results embodied in this major project, to the best of our knowledge, haven't been submitted to any other university or institution for award of any degree or diploma.

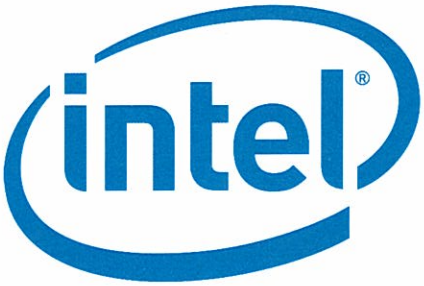
Date: Place: Ahmedabad

Dr. N. P. Gajjar
Internal Guide &
Program Coordnator

Dr. D. K. Kothari
Section Head, EC

Dr. P. N. Tekwani
Head of EE Dept.

Dr. K. Kotecha
Director, IT-NU



Intel Technology India Pvt. Ltd.

Certificate

This is to certify that the Major Project entitled **"Automated Display Kernel Validation"** submitted by **Harsh Kotak (13MECE08)**, towards the partial fulfilment of the requirements for the degree of Master of Technology in Embedded Systems, Nirma University, Ahmedabad is the record of work carried out by him under my supervision and guidance. In my opinion, the submitted work has reached a level required for being accepted for examination. The results embodied in this major project, to the best of my knowledge, haven't been submitted to any other university or institution for award of any degree or diploma.

Date: Place: Bangalore

Mr. Avinash Reddy Palleti,
Graphics Software Engineer,
Intel Technology India Pvt. Ltd.,
Bangalore.

Mr. Pratyush Dutta,
Engineering Manager,
Intel Technology India Pvt. Ltd.,
Bangalore.

Acknowledgements

I would like to express my gratitude and sincere thanks to **Dr. P. N. Tekwani**, Head of Electrical Engineering Department, and **Dr. N. P. Gajjar**, PG Coordinator of M.Tech Embedded Systems program for allowing me to undertake this thesis work.

I grateful to **Dr. N. P. Gajjar**, not only my internal project guide, but also a great motivator. I am thankful for his guidance, encouragement, motivation and constant monitoring throughout the course of this thesis. The blessings and support given by him time to time shall carry me a long way in the journey of life on which I am about to embark.

I am solely thankful to Android Display Validation Architect **Mr. Manikandan Pillai**, and my mentor **Mr. Avinash Reddy Palleti**, Graphics Software Engineer with Android Display Team, Intel Technology India Pvt. Ltd., for their cordial support, constant supervision as well as for valuable information regarding the project and guidance, which helped me in completing this task through various stages.

I am grateful to all the members of Android Display Team at Intel Technology India Pvt. Ltd. for their constant support and guidance.

Lastly, I thank almighty, my parents and friends for their constant encouragement without which this thesis would not be possible.

- Harsh Kotak

Abstract

Number of portable devices had increased not in percentage but in multiples in last decade and Android is the most widely used operating system for handheld devices. Linux Kernel which is at the heart of the Android allows user to leverage the full potential of the hardware. Display is one of the most crucial components in these consumer electronic devices as it has a huge impact on user experience. Frame data seen on the display is received from processor and display drivers are needed for communication between the display and display engine. Display drivers for Android operating system running on Intel platforms are available in Linux kernel and are improved and developed everyday.

Validation of display kernel is different from other components as it is a visual entity and not just a computational entity. Manual validation would take hours and hours and even then it wouldn't be as accurate due to limitations of human eye. Even if some standard test cases are developed, it wouldn't be sufficient as there can be innumerable use case scenarios for a user to use the device which can vary from high end gaming or watching 4k movies on the same device to using it for social networking.

The automation framework developed here can be used to validate various display features over different types of displays. Different APIs are developed for validating various features irrespective to the display connected to the chipset. This framework is generic in the sense that as soon as a new platform comes up, few configuration files need to be added to it and framework is ready to run validation of display driver on the new chipset.

Python is used to automate this validation framework. To validate a feature, same API is used in different scenarios created by python to validate that feature in various possible configurations.

A set of all such test cases written in python make up a test suite that can itself be automated. With a single trigger, all test can start execution one after the other without any human intervention. The best part is target need not be at your desk. It can be any device from a pool of devices located at any point on the globe.

PAGE Tool is used to identify test cases to verify a patch submitted. For this being done manually so far, validation owner had to analyze the patch and plan out test cases to verify a patch. PAGE Tool on the other hand automatically analyzes a patch and identifies the most significant test cases by providing patch id as the input. It also recommends test cases to check regression.

GRAD Tool, on the contrary, grades validation framework. It tell validation engineer which part of driver code is not validated by any test cases. It will encourage validation engineer to develop stricter test cases. This can help in developing even better quality display driver.

Contents

List of Figures

Abbreviations

MIPI ..... Mobile Industry Processor Interface

DP ..... Display Port

eDP ..... embedded Display Port

HDMI ..... High-Definition Multimedia Interface

PAGE ..... Patch based Automated Guided Execution

GRAD ..... Gap analysis and Risk Assessment for Display driver

OpenGL ..... Open Graphics Library

OpenGL ES ..... OpenGL for Embedded Systems

EDID ..... Extended Display Identification Data

HWC ..... Hardware Composer

OEM ..... Original Equipment Manufacturer

ODM ..... Original Design Manufacturer

API ..... Application Program Interface

xml ..... Extensible Markup Language

Chapter 1

Introduction

1.1 Motivation

Display is one of the most important components in today's advanced consumer electronic devices, be it a tablet, phone, smartwatch or personal computer. Display is the closest component a user interacts with the device for a major part of the time he/she uses it. For a great user experience it is very important that display performs seamlessly which means display driver should be very well qualified. There are various use case scenarios of display in which user might want to use his/her device. It is important to validate the behaviour of the display for each use case scenario. Android uses Linux kernel, which means all the display drivers developed for Intel SoC will be upstreamed to open source Linux kernel.

1.2 Problem Statement

Manual validation of display kernel takes a lot of manpower and time. As display is a visual component, its validation would need a human interaction to make sure there are no bug in display driver. This would take hours and hours of man power and even then be prone to errors due to limitations of human eye. Also, we need to stress test the display driver to see there is no error while having a use case scenario hundreds of times in series, which again would need lot of manual work.

As new platform comes up, we would need to develop new test cases for each platform based on its configurations.

Wherever there is a patch submitted for a bug fix, we need to identify the most significant test case based on the patch details. To identify most significant test case, test planner needs to understand whole driver code, understand what changes are made and what other effects it might have other fixing the bug. Test planner also need to be aware of all the tests available and plan tests which would thoroughly test the display driver.

With such a large driver code, it is fairly possible that some part of code might not be validated by any of the test case. This increases the probability of gaps and escapes.

1.3 Thesis Organization

The rest of the thesis is organized is as follows:

Chapter 2 gives introduction on Android Graphics architecture and display architecture. It shows how graphics data flow in for of buffer from buffer creators to buffer consumers.

Chapter 3 gives introduction to automation framework and how features of object oriented programming are leveraged to make up a generic framework. It also gives an idea on how Python is used for automation and how python talks to device with the use of python wrapper.
At the end of this chapter, Automation of Automated Tests is introduced which is an environment, where number of test cases can be executed sequential with a single trigger irrespective to where the target device is present and results are sent automatically to the concerned authorities.

Chapter 4 introduces two tools. First is PAGE Tool which is used to identify most significant test cases required to verify a patch. It also keeps check on regression testing. Another one is a GRAD Tool which tell how robust is the validation framework that what part of driver code does it validates and what part does it misses.

Chapter 5 brings us to conclusion on how Automation in Display Kernel Validation can make a difference and how we can get a better quality display driver by using it.

Chapter 2

Literature Survey

2.1 Android Graphics

Application developers draw images to the screen in two ways: with Canvas or OpenGL. android.graphics.Canvas is a 2D graphics API and is the most popular graphics API among developers. Canvas operations draw all the stock and custom android.view.Views in Android. In Android, hardware acceleration for Canvas APIs is accomplished with a drawing library called OpenGLRenderer that translates Canvas operations to OpenGL operations so they can execute on the GPU.[1]

In addition to Canvas, the other main way that developers render graphics is by using OpenGL ES to directly render to a surface. Android provides OpenGL ES interfaces in the android.opengl package that developers can use to call into their GL implementations with the SDK or with native APIs provided in the Android NDK. [1]

2.2 Android Graphics Components

No matter what rendering API developers use, everything is rendered onto a "surface." The surface represents the producer side of a buffer queue that is often consumed by SurfaceFlinger. Every window that is created on the Android platform is backed by a surface. All of the visible surfaces rendered are composited onto the display by SurfaceFlinger.[1]

Working of the key components is as follows

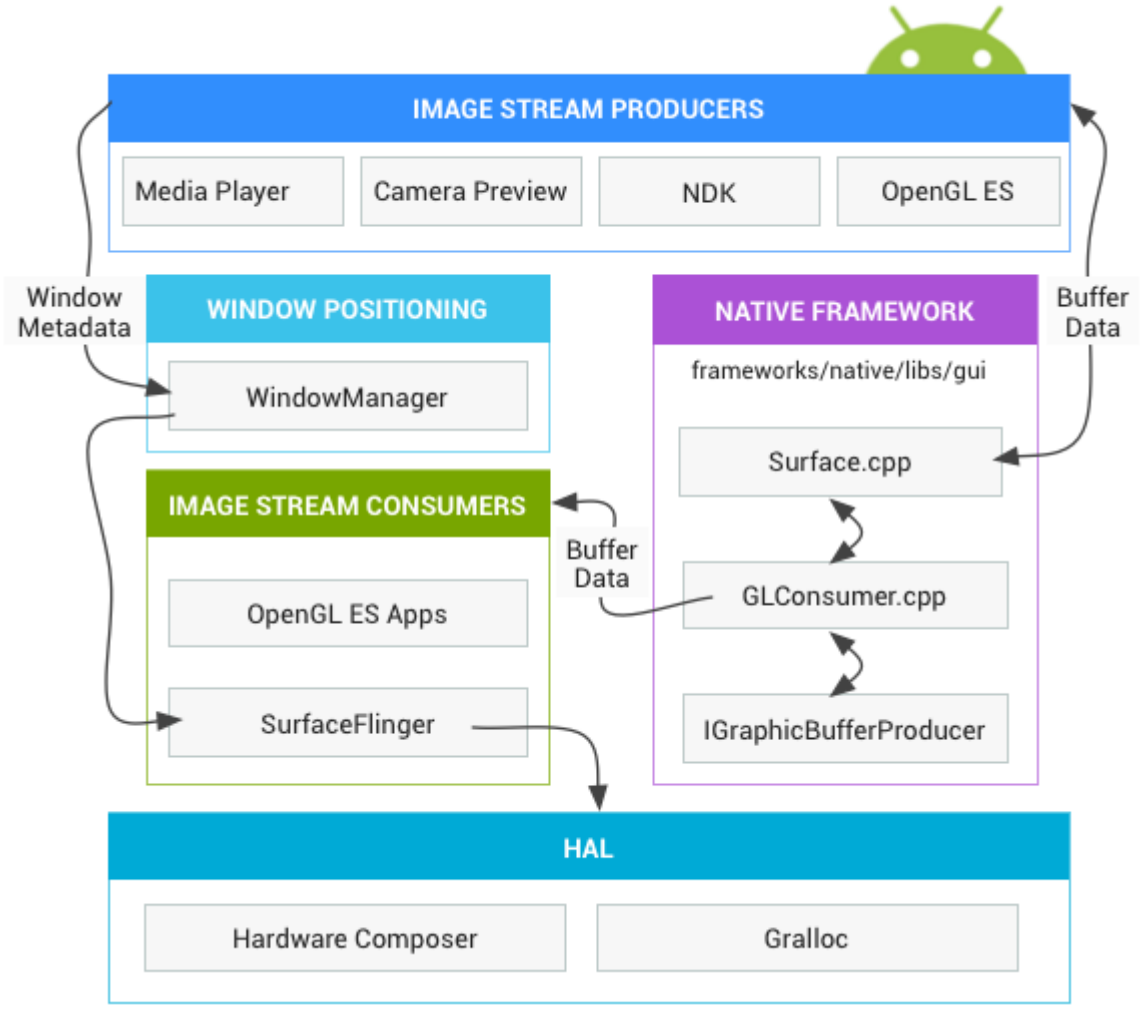


Figure 2.1: How surfaces are rendered [1]

Image Stream Producers:
An image stream producer can be anything that produces graphic buffers for consumption. Examples include OpenGL ES, Canvas 2D, and mediaserver video decoders.[1]

Image Stream Consumers:
The most common consumer of image streams is SurfaceFlinger, the system service that consumes the currently visible surfaces and composites them onto the display using information provided by the Window Manager. SurfaceFlinger is the only service that can modify the content of the display. SurfaceFlinger uses OpenGL and the Hardware Composer to compose a group of surfaces.[1]

Other OpenGL ES apps can consume image streams as well, such as the camera app consuming a camera preview image stream. Non-GL applications can be consumers too, o, for example the ImageReader class.[1]

Window Manager:
The Android system service that controls a window, which is a container for views. A window is always backed by a surface. This service oversees life cycles, input and focus events, screen orientation, transitions, animations, position, transforms, z-order, and many other aspects of a window. The Window Manager sends all of the window metadata to SurfaceFlinger so SurfaceFlinger can use that data to composite surfaces on the display.[1]

Hardware Composer:
The hardware abstraction for the display subsystem. SurfaceFlinger can delegate certain composition work to the Hardware Composer to offload work from OpenGL and the GPU. SurfaceFlinger acts as just another OpenGL ES client. So when SurfaceFlinger is actively compositing one buffer or two into a third, for instance, it is using OpenGL ES. This makes compositing lower power than having the GPU conduct all computation.[1]

The Hardware Composer HAL conducts the other half of the work. This HAL is the central point for all Android graphics rendering. Hardware Composer must support events, one of which is VSYNC. Another is hotplug for plug-and-play HDMI support.[1]

Gralloc: The graphics memory allocator is needed to allocate memory that is requested by image producers. See the Gralloc HAL section for more information.[1]

2.3 Data Flow

Following diagram depicts Android graphics pipeline:

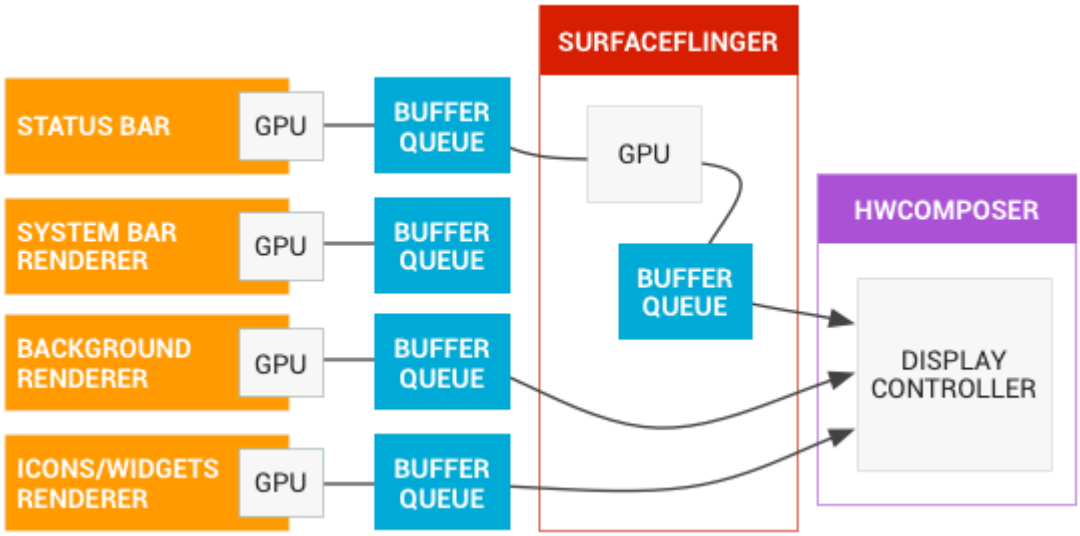


Figure 2.2: Graphics data flow through Android [1]

The objects on the left are renderers producing graphics buffers, such as the home screen, status bar, and system UI. SurfaceFlinger is the compositor and Hardware Composer is the composer.[1]

2.3.1 BufferQueue

BufferQueues provide the glue between the Android graphics components. These are a pair of queues that mediate the constant cycle of buffers from the producer to the consumer. Once the producers hand off their buffers, SurfaceFlinger is responsible for compositing everything onto the display. [1]

BufferQueue communication process is shown by diagram below:

BufferQueue contains the logic that ties image stream producers and image stream consumers together. Some examples of image producers are the camera previews produced by the camera HAL or OpenGL ES games. Some examples of image consumers are SurfaceFlinger or another app that displays an OpenGL ES stream, such as the camera app displaying the camera viewfinder.[1]

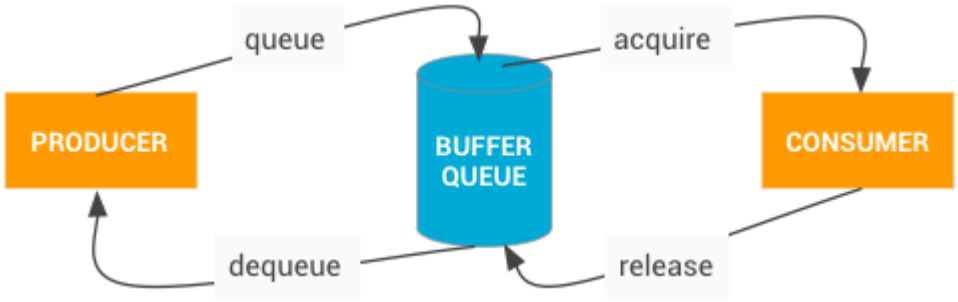


Figure 2.3: BufferQueue communication process [1]

BufferQueue is a data structure that combines a buffer pool with a queue and uses Binder IPC to pass buffers between processes. The producer interface, or what you pass to somebody who wants to generate graphic buffers, is IGraphicBufferProducer (part of SurfaceTexture). BufferQueue is often used to render to a Surface and consume with a GL Consumer, among other tasks.[1]

There are three modes of operation of BufferQueue

Synchronous-like mode - BufferQueue by default operates in a synchronous-like mode, in which every buffer that comes in from the producer goes out at the consumer. No buffer is ever discarded in this mode. And if the producer is too fast and creates buffers faster than they are being drained, it will block and wait for free buffers.[1]

Non-blocking mode - BufferQueue can also operate in a non-blocking mode where it generates an error rather than waiting for a buffer in those cases. No buffer is ever discarded in this mode either. This is useful for avoiding potential deadlocks in application software that may not understand the complex dependencies of the graphics framework.[1]

Discard mode - Finally, BufferQueue may be configured to discard old buffers rather than generate errors or wait. For instance, if conducting GL rendering to a texture view and drawing as quickly as possible, buffers must be dropped.[1]

To conduct most of this work, SurfaceFlinger acts as just another OpenGL ES client. So when SurfaceFlinger is actively compositing one buffer or two into a third, for instance, it is using OpenGL ES.[1]

The Hardware Composer HAL conducts the other half of the work. This HAL acts as the central point for all Android graphics rendering.[1]

2.3.2 Synchronization framework

Since Android graphics offer no explicit parallelism, vendors have long implemented their own implicit synchronization within their own drivers. This is no longer required with the Android graphics synchronization framework.[1]

The synchronization framework explicitly describes dependencies between different asynchronous operations in the system. The framework provides a simple API that lets components signal when buffers are released. It also allows synchronization primitives to be passed between drivers from the kernel to user space and between user space processes themselves.[1]

For example, an application may queue up work to be carried out in the GPU. The GPU then starts drawing that image. Although the image has not been drawn into memory yet, the buffer pointer can still be passed to the window compositor along with a fence that indicates when the GPU work will be finished. The window compositor may then start processing ahead of time and hand off the work to the display controller. In this manner, the CPU work can be done ahead of time. Once the GPU finishes, the display controller can immediately display the image.[1]

The synchronization framework also allows implementers to leverage synchronization resources in their own hardware components. Finally, the framework provides visibility into the graphics pipeline to aid in debugging.[1]

2.4 Graphics Architecture

2.4.1 BufferQueue and gralloc

To understand how Android's graphics system works, we have to start behind the scenes. At the heart of everything graphical in Android is a class called BufferQueue. Its role is simple enough: connect something that generates buffers of graphical data (the "producer") to something that accepts the data for display or further processing (the "consumer"). The producer and consumer can live in different processes. Nearly everything that moves buffers of graphical data through the system relies on BufferQueue.[1]

The basic usage is straightforward. The producer requests a free buffer (dequeueBuffer()), specifying a set of characteristics including width, height, pixel format, and usage flags. The producer populates the buffer and returns it to the queue (queueBuffer()). Some time later, the consumer acquires the buffer (acquireBuffer()) and makes use of the buffer contents. When the consumer is done, it returns the buffer to the queue (releaseBuffer()).[1]

Most recent Android devices support the "sync framework". This allows the system to do some nifty thing when combined with hardware components that can manipulate graphics data asynchronously. For example, a producer can submit a series of OpenGL ES drawing commands and then enqueue the output buffer before rendering completes. The buffer is accompanied by a fence that signals when the contents are ready. A second fence accompanies the buffer when it is returned to the free list, so that the consumer can release the buffer while the contents are still in use. This approach improves latency and throughput as the buffers move through the system.[1]

Some characteristics of the queue, such as the maximum number of buffers it can hold, are determined jointly by the producer and the consumer.[1]

The BufferQueue is responsible for allocating buffers as it needs them. Buffers are retained unless the characteristics change; for example, if the producer starts requesting buffers with a different size, the old buffers will be freed and new buffers will be allocated on demand.[1]

Buffer contents are never copied by BufferQueue. Moving that much data around would be very inefficient. Instead, buffers are always passed by handle.[1]

2.4.2 gralloc HAL

The actual buffer allocations are performed through a memory allocator called "gralloc", which is implemented through a vendor-specific HAL interface. The alloc()function takes the arguments you'd expect – width, height, pixel format – as well as a set of usage flags. Those flags merit closer attention.[1]

The gralloc allocator is not just another way to allocate memory on the native heap. In some situations, the allocated memory may not be cache-coherent, or could be totally inaccessible from user space. The nature of the allocation is determined by the usage flags, which include attributes like:[1]

- how often the memory will be accessed from software (CPU)[1]
- how often the memory will be accessed from hardware (GPU)[1]
- whether the memory will be used as an OpenGL ES ("GLES") texture[1]
- whether the memory will be used by a video encoder[1]

For example, if your format specifies RGBA 8888 pixels, and you indicate the buffer will be accessed from software – meaning your application will touch pixels directly – then the allocator needs to create a buffer with 4 bytes per pixel in R-G-B-A order. If instead you say the buffer will only be accessed from hardware and as a GLES texture, the allocator can do anything the GLES driver wants – BGRA ordering, non-linear "swizzled" layouts, alternative color formats, etc. Allowing the hardware to use its preferred format can improve performance.[1]

Some values cannot be combined on certain platforms. For example, the "video encoder" flag may require YUV pixels, so adding "software access" and specifying RGBA 8888 would fail.[1]

The handle returned by the gralloc allocator can be passed between processes through Binder.[1]

2.5 SurfaceFlinger and Hardware Composer

Having buffers of graphical data is wonderful, but life is even better when you get to see them on your device's screen. That's where SurfaceFlinger and the Hardware Composer HAL come in.[1]

SurfaceFlinger's role is to accept buffers of data from multiple sources, composite them, and send them to the display. When an app comes to the foreground, the WindowManager service asks SurfaceFlinger for a drawing surface. SurfaceFlinger creates a "layer" - the primary component of which is a BufferQueue - for which SurfaceFlinger acts as the consumer. A Binder object for the producer side is passed through the WindowManager to the app, which can then start sending frames directly to SurfaceFlinger. (Note: The WindowManager uses the term "window" instead of "layer" for this and uses "layer" to mean something else. We're going to use the SurfaceFlinger terminology. It can be argued that SurfaceFlinger should really be called LayerFlinger.)[1]

For most apps, there will be three layers on screen at any time: the "status bar" at the top of the screen, the "navigation bar" at the bottom or side, and the application's UI. Some apps will have more or less, e.g. the default home app has a separate layer for the wallpaper, while a full-screen game might hide the status bar. Each layer can be updated independently. The status and navigation bars are rendered by a system process, while the app layers are rendered by the app, with no coordination between the two.[1]

Device displays refresh at a certain rate, typically 60 frames per second on phones and tablets. If the display contents are updated mid-refresh, "tearing" will be visible; so it's important to update the contents only between cycles. The system receives a signal from the display when it's safe to update the contents. For historical reasons we'll call this the VSYNC signal.[1]

The refresh rate may vary over time, e.g. some mobile devices will range from 58 to 62fps depending on current conditions. For an HDMI-attached television, this could theoretically dip to 24 or 48Hz to match a video. Because we can update the screen only once per refresh cycle, submitting buffers for display at 200fps would be a waste of effort as most of the frames would never be seen. Instead of taking action whenever an app submits a buffer, SurfaceFlinger wakes up when the display is ready for something new.[1]

When the VSYNC signal arrives, SurfaceFlinger walks through its list of layers looking for new buffers. If it finds a new one, it acquires it; if not, it continues to use the previously-acquired buffer. SurfaceFlinger always wants to have something

to display, so it will hang on to one buffer. If no buffers have ever been submitted on a layer, the layer is ignored.[1]

Once SurfaceFlinger has collected all of the buffers for visible layers, it asks the Hardware Composer how composition should be performed.[1]

2.5.1 Hardware Composer

The Hardware Composer HAL ("HWC") was first introduced in Android 3.0 ("Honeycomb") and has evolved steadily over the years. Its primary purpose is to determine the most efficient way to composite buffers with the available hardware. As a HAL, its implementation is device-specific and usually implemented by the display hardware OEM.[1]

The value of this approach is easy to recognize when you consider "overlay planes." The purpose of overlay planes is to composite multiple buffers together, but in the display hardware rather than the GPU. For example, suppose you have a typical Android phone in portrait orientation, with the status bar on top and navigation bar at the bottom, and app content everywhere else. The contents for each layer are in separate buffers. You could handle composition by rendering the app content into a scratch buffer, then rendering the status bar over it, then rendering the navigation bar on top of that, and finally passing the scratch buffer to the display hardware. Or, you could pass all three buffers to the display hardware, and tell it to read data from different buffers for different parts of the screen. The latter approach can be significantly more efficient.[1]

As you might expect, the capabilities of different display processors vary significantly. The number of overlays, whether layers can be rotated or blended, and restrictions on positioning and overlap can be difficult to express through an API. So, the HWC works like this:[1]

- SurfaceFlinger provides the HWC with a full list of layers, and asks, "how do you want to handle this?"[1]
- The HWC responds by marking each layer as "overlay" or "GLES composition."[1]
- SurfaceFlinger takes care of any GLES composition, passing the output buffer to HWC, and lets HWC handle the rest.[1]

Since the decision-making code can be custom tailored by the hardware vendor, it's possible to get the best performance out of every device.[1]

Overlay planes may be less efficient than GL composition when nothing on the screen is changing. This is particularly true when the overlay contents have transparent pixels, and overlapping layers are being blended together. In such cases, the HWC can choose to request GLES composition for some or all layers and retain the composited buffer. If SurfaceFlinger comes back again asking to composite the same set of buffers, the HWC can just continue to show the previously-composited scratch buffer. This can improve the battery life of an idle device.[1]

Devices shipping with Android 4.4 ("KitKat") typically support four overlay planes. Attempting to composite more layers than there are overlays will cause the system to use GLES composition for some of them; so the number of layers used by an application can have a measurable impact on power consumption and performance.[1]

The overlay planes have another important role: they're the only way to display DRM content. DRM-protected buffers cannot be accessed by SurfaceFlinger or the GLES driver, which means that your video will disappear if HWC switches to GLES composition.[1]

2.5.2 The Need for Triple-Buffering

To avoid tearing on the display, the system needs to be double-buffered: the front buffer is displayed while the back buffer is being prepared. At VSYNC, if the back buffer is ready, you quickly switch them. This works reasonably well in a system where you're drawing directly into the framebuffer, but there's a hitch in the flow when a composition step is added. Because of the way SurfaceFlinger is triggered, our double-buffered pipeline will have a bubble.[1]

Suppose frame N is being displayed, and frame N+1 has been acquired by SurfaceFlinger for display on the next VSYNC. (Assume frame N is composited with an overlay, so we can't alter the buffer contents until the display is done with it.) When VSYNC arrives, HWC flips the buffers. While the app is starting to render frame N+2 into the buffer that used to hold frame N, SurfaceFlinger is scanning the layer list, looking for updates. SurfaceFlinger won't find any new buffers, so it prepares to show frame N+1 again after the next VSYNC. A little while later, the app finishes rendering frame N+2 and queues it for SurfaceFlinger, but it's too late. This has effectively cut our maximum frame rate in half.[1]

We can fix this with triple-buffering. Just before VSYNC, frame N is being displayed, frame N+1 has been composited (or scheduled for an overlay) and is ready

to be displayed, and frame N+2 is queued up and ready to be acquired by SurfaceFlinger. When the screen flips, the buffers rotate through the stages with no bubble. The app has just less than a full VSYNC period (16.7ms at 60fps) to do its rendering and queue the buffer. And SurfaceFlinger / HWC has a full VSYNC period to figure out the composition before the next flip. The downside is that it takes at least two VSYNC periods for anything that the app does to appear on the screen. As the latency increases, the device feels less responsive to touch input.[1]

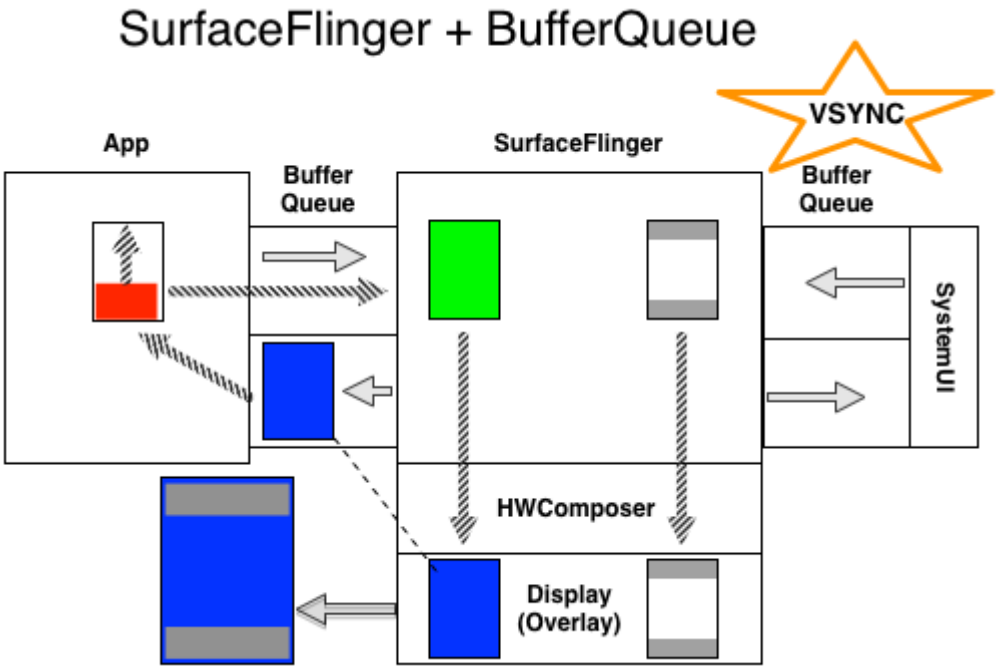


Figure 2.4: SurfaceFlinger + BufferQueue [1]

The diagram above depicts the flow of SurfaceFlinger and BufferQueue. During frame:[1]

- red buffer fills up, then slides into BufferQueue[1]
- after red buffer leaves app, blue buffer slides in, replacing it[1]
- green buffer and systemUI shadow-slide into HWC (showing that SurfaceFlinger still has the buffers, but now HWC has prepared them for display via overlay on the next VSYNC).[1]

The blue buffer is referenced by both the display and the BufferQueue. The app is not allowed to render to it until the associated sync fence signals.[1]

On VSYNC, all of these happen at once:

- red buffer leaps into SurfaceFlinger, replacing green buffer[1]
- green buffer leaps into Display, replacing blue buffer, and a dotted-line green twin appears in the BufferQueue[1]
- the blue buffers fence is signaled, and the blue buffer in App empties[1]
- display rect changes from (blue + SystemUI) to (green + SystemUI)[1]

(The buffer doesnt actually empty; if you submit it without drawing on it youll get that same blue again. The emptying is the result of clearing the buffer contents, which the app should do before it starts drawing)[1]

We can reduce the latency by noting layer composition should not require a full VSYNC period. If composition is performed by overlays, it takes essentially zero CPU and GPU time. But we can't count on that, so we need to allow a little time. If the app starts rendering halfway between VSYNC signals, and SurfaceFlinger defers the HWC setup until a few milliseconds before the signal is due to arrive, we can cut the latency from 2 frames to perhaps 1.5. In theory you could render and composite in a single period, allowing a return to double-buffering; but getting it down that far is difficult on current devices. Minor fluctuations in rendering and composition time, and switching from overlays to GLES composition, can cause us to miss a swap deadline and repeat the previous frame.[1]

SurfaceFlinger's buffer handling demonstrates the fence-based buffer management mentioned earlier. If we're animating at full speed, we need to have an acquired buffer for the display ("front") and an acquired buffer for the next flip ("back"). If we're showing the buffer on an overlay, the contents are being accessed directly by the display and must not be touched. But if you look at an active layer's BufferQueue state in the dumpsys SurfaceFlinger output, you'll see one acquired buffer, one queued buffer, and one free buffer. That's because, when SurfaceFlinger acquires the new "back" buffer, it releases the current "front" buffer to the queue. The "front" buffer is still in use by the display, so anything that dequeues it must wait for the fence to signal before drawing on it. So long as everybody follows the fencing rules, all of the queue-management IPC requests can happen in parallel with the display.[1]

2.5.3 Virtual Displays

SurfaceFlinger supports a "primary" display, i.e. what's built into your phone or tablet, and an "external" display, such as a television connected through HDMI. It also supports a number of "virtual" displays, which make composited output available within the system. Virtual displays can be used to record the screen or send it over a network.[1]

Virtual displays may share the same set of layers as the main display (the "layer stack") or have its own set. There is no VSYNC for a virtual display, so the VSYNC for the primary display is used to trigger composition for all displays.[1]

In the past, virtual displays were always composited with GLES. The Hardware Composer managed composition for only the primary display. In Android 4.4, the Hardware Composer gained the ability to participate in virtual display composition.[1]

The frames generated for a virtual display are written to a BufferQueue.[1]

Chapter 3

Automation in Display Validation

3.1 Introduction

The Android Test Automation Framework is designed to reduce the complexity of developing the actual test cases when the product team moves from one chipset to another there by providing a more generic re-usable software component that will not be modified very often.

This will enable more focus on automation of the test cases leaving very little number of test cases to be run manually. This Automation Framework basically aims at executing test cases from Linux Layer by developing C/C++ applications and/or scripts.

3.2 Design

The Android Test Automation Framework is designed in such a way that, test cases can be ported from one platform to another with minimal effort.

The below diagram depicts various components of the display automation framework and their interaction.

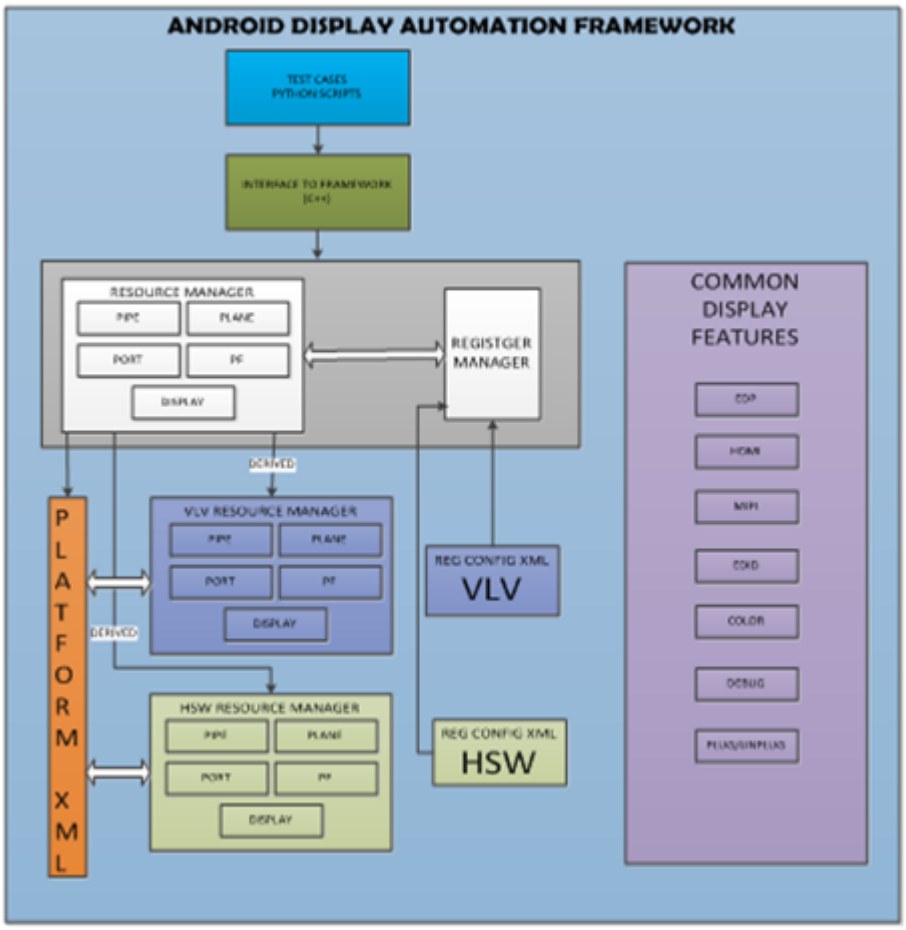


Figure 3.1: Android Display Automation Framework

3.3 Configuration Files

The design involves creating a generic framework that will be used across different platforms (display chipset) with each platform has its configuration put down in XML based configuration file(s).

There are three XML configuration files:

- Register map configuration file: It contains the display registers and register configuration information for a particular platform.
- Platform configuration file: It contains the configuration of display components in each platform.
- Debug Register/API configuration file: It contains the register details and debug information of the register whose state is to be verified.

3.3.1 Register Map Configuration File

File Name : <platform>-regconfig.xml

- The configuration specific to platform will be mentioned by the <config> </config> tags with each configuration have its name, type and value mentioned.
- The display registers, will be mentioned by <Register> </Register> tags with each of its factor mentioned by <factor> </factor> tag.
- The offset of the resister will be mentioned by offset tag in <Register> </Register>
- The bitmask and expected value are mention with in the <factor> </factor> tag.

3.3.2 Platform Configuration File

File Name: <platform>.xml

- This xml file will contain the configuration information for a particular platform.
- This file will contain information on the type of displays supported.
- Configuration information of how the display components like panelfitters, planes, pipes, ports are configured.
- The platform can be identified by the <platform name> tag.
- The configuration of different display components are mentioned by their respective tags.

3.3.3 Debug Configuration File

File Name : <platform>-testregisters.xml

- This is basically used to check the state of the registers i.e. for debugging the register values.
- This xml file contains register information for different specific to an API.

- Depending upon the API names the register to be verified and their expected values are mentioned.

The structure and layout of these XML configuration files are mentioned in their respective XML schema definition file (.xsd).

XSD/e Parser tool for C/C++ will be used to parse the contents of the XML configuration file (by using their .xsd file as input) and keep them in C++ objects to be accessible to the framework.

The Framework will be designed to contain feature classes for each module and components. e.g. eDP, HDMI, Color, Pipe, Plane, Port etc.

The Interface classes will be created that will be exposed to the test automation scripts (python scripts). These interfaces will act as an abstraction between the framework feature classes and the python scripts.

When a particular test case executed, corresponding libraries are loaded on to the system based upon the platform.

3.4 Resource Manager

The resource manager is heart of the display automation framework. The resource manager binds all the display components together for a particular platform.

The resource manager base class is implemented as an abstract factory class with virtual interfaces to access objects of components in display family.

The platform specific resource manager class is derived from this base resource manager class. The base class provides a method, which at runtime will identify the platform and create a factory instance of resource manager specific to that platform.

The main functionality of resource manager is to parse the platform.xml to retrieve configuration information specific to the platform and populate the objects of different display components. This is done by a method parseXml() defined in the resource manager class.

Another important functionality is to bind together the display components into a path object. Each path object contains the complete display components that are tied together to a particular display type. This functionality is implemented in a method called bindObjects().

Various methods are implemented in Resourse manager for initialization, parsing xml file, binding objects, get active plane/pipe/port/display, get display EDID, etc.

3.5 Register Manager

Register manager deals with the registermap xml configuration file specific to a platform. The register manager is implemented as a singleton class and provides interfaces to parse the registermap file and manipulate display registers. Register manager use methods defined in intel gpu tools in order to read or write register contents and compare with the expected value provided in the xml file.

Various methods are implemented in Register Manager for parsing xml file, gettnig configurations, getting register list, masking particular bits in a register, reading a register value, verifying register value, writing a register value, etc.

3.6 Feature Classes

The feature classes have all the necessary methods and only required functionality will be exported out using the interface APIs. The features that can be tested from this Android framework are listed below:

- EDP contains all functionalities specific to eDP display.
- MIPI contains all functionalities specific to MIPI display.
- HDMI contains all functionalities specific to HDMI display.
- EDID contains all methods specific to EDID class.
- Pipe contains all the functionalities related to PIPE.
- Port contains all the functionalities related to PORT.
- Plane contains all the functionalities related to PLANE.
- PanelFitter contains all functionalities related to Panel Fitter.
- Color contains all Color related features.

3.6.1 eDP

The EDP class has features related to EDP display. It is derived from the base class display.

The eDP class supports the following methods:

- PSR active,
- Link Training,
- Get attached Pipe,
- Get attached Port,
- Get current Resolution,
- Hardware Rotation, etc.

3.6.2 HDMI

The HDMI class has features related to HDMI display. It is derived from the base class display.

The HDMI class supports the following methods:

- hotplug detect,
- hotplug,
- hotunplug,
- getsupported modes,
- setmode,
- Get attached Pipe,
- Get attached Port,
- Get current Resolution,
- Hardware Rotation, etc.

3.6.3 MIPI

The MIPI class has features related to MIPI display. It is derived from the base class display.

The MIPI class supports the following methods:

- DRRS active,
- MIPI Sequence,
- MIPI Timings,
- MIPI Coinfigs,
- Get attached Pipe,
- Get attached Port,
- Get current Resolution,
- Hardware Rotation, etc.

3.6.4 EDID

The EDID class has implementation of features specific to parsing and retrieving the EDID data.

The EDID class supports the following methods:

- Edid Parser,
- verify CEABlock,
- Get EDID baseblock,
- Get Native Resolution,
- Get supported modes, etc.

3.6.5 Pipe

The Pipe class has support related only to display pipe. The Pipe class supports following methods.

- is Pipe Enabled,
- get Pipe Resolution,
- get Pipe ID,
- get Maximum planes,
- get pipe string,
- set Pipe ID,
- set Pipe Registers, etc.

3.6.6 Port

The Port class has support related only to display port. The Port class supports following methods.

- is Port Enabled,
- get attached pipes,
- get display type,
- get Port ID,
- get Maximum planes,
- get port string,
- get Display ID,
- set Port Registers, etc.

3.6.7 Plane

The Plane class has support related only to display planes. The Plane class supports following methods.

- is Plane Enabled,
- get attached pipe,
- get plane ID,
- get plane sting,
- set plane ID,
- set Port Registers, etc.

3.7 Interface Framework APIs

The interface have the APIs for the framework which will be exposed to the outside world. i.e. applications (C/C++/python scripts) to validate the android display subsystem. These interfaces will act as an abstraction between the framework feature classes and the user application using them.

The APIs are created to provide an interface between the external program and the display framework. These APIs are implemented as C functions so that it can be equally accessible from a scripting language (like python) and programming language (like C). These APIs are available to external program as a shared library (.so).

3.8 Python for Automation

Manual testing takes a lot of man power and time as compared to automated testing. Manual testing is also prone to reduced accuracy and precision. With frequent code changes across several components of driver code, lot of bugs get introduced due to inter dependencies of different components. This adds more delay to the expensive turnaround time of a bug fix.

A scripting language like Python supports scripts written for a special runtime environment that can interpret and automate the execution of tasks which could alternatively be executed one-by-one by a human operator. Python is a flexible, clear and precise scripting language.

3.8.1 Python wrapper

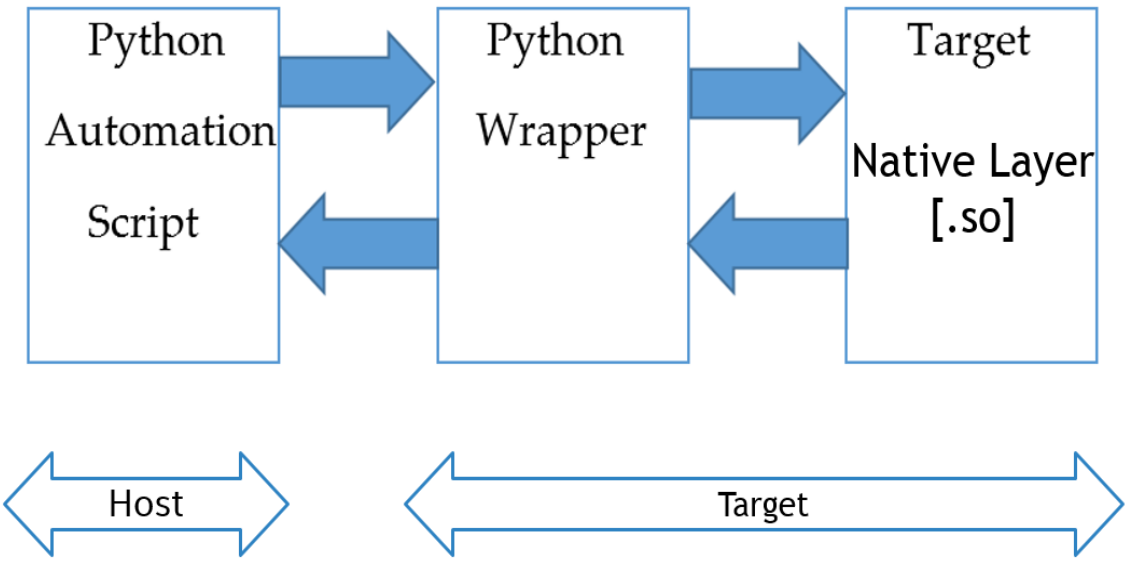


Figure 3.2: Python Wrapper

The automation framework written is framework is compiled with android source code to generate shared object files that can run on device with Android OS. But Python script running on host cannot directly interact with .so files in target though it can interact with python interpreter on device. So we need a python wrapper function to interact with .so files. A wrapper function is a subroutine in a computer program whose main purpose is to make a system call with little or no additional computation.

3.8.2 Automation of Automated Tests

AAT is a pool of Target Machines at different locations. AAT provides the requirements and system is found out of its own with given OS and drivers installed. AAT can run periodically as well as event based depending on users choice. Results analysis of all the tests running in AAT is generated automatically and sent to concerned stack holders.

AAT is an automated validation environment which takes input as:

- **Test Binaries:** It consists of things like test scripts, shared object (so) files, golden images, edids, etc. which are required to run tests in ATT. It can be reused for different requests of execution.
- **Test Name:** It is a command along with command line argument for each test.
- **Collection of Test Lists:** This is a combination of different test so that we can make a test suite to run all the tests in suite together.
- **Request for Execution:** This is the main component of AAT. It consists information on which test binaries to be used and which test suite is to be executed.
- **Message to Start Execution:** It is just the trigger message to start the tests execution. It consists of information on if the trigger is time based or even based.

Provided this input, test scripts will start running once a target is available and required images are flashed. If AAT has more than one target available for execution, it will divide the given tests into 2 sets, each for running in available target machine. This makes execution in half the time.

Test results can be one of the three: Passed, Failed or Erred. Failed and Erred tests are run for the second time to make sure that those are the genuine failures and not due to test environment.

It also provides logs for all the tests that are ran on ATT in order to debug failures and errors.

Thus, ATT is in real sense automation in Validation which can run whole set of test cases with a single start message and provide all the results to respective stack holders automatically.

Chapter 4

Tools for Automated Display Kernel Validation

4.1 PAGE Tool

4.1.1 Problem Statement

Finding and fixing issues in display driver at early stages of product lifecycle helps to reduce the overall effort required to deliver quality driver. Considering this, Early Test plays crucial role in validation lifecycle.

While a driver developer develops bug fix, called as patch, in the driver, it needs to be validated before it is merged. We already have test suite developed for all the features of the driver. The most significant test cases are identified and used to verify the quality of the patch. This is known as Early Test (ET). In existing scenario VCO (Validation Component Owner) plans the test manually for each ET request from development team and allow the developer to check-in the patch based on ET results. VCO might plan minimal grid or extensive grid based on his/her understanding on code changes happened and also about the Test suite. Issue with first case is high risk whereas with the second case is resource bandwidth is consumed more and might include some redundant test cases. So there is a need for tool which improves the efficiency of ET grid planning by focusing time and resources on the most critical parts of the code which is modified by developers and also check for regression which means bug fix for a feature should not break other feature.

4.1.2 Introduction to PAGE Tool

Patch based Automated Guided Execution (PAGE) Tool aims at automating the test grid planning effectively. Automatically analyze code changes for an ET and then generate a test suite that achieves high coverage. This tool can prove to be useful for various software validation teams that use a version control system like GIT. Back tracing from coverage elements to the corresponding test cases enables users to analyze grid and extend the test cases for better coverage, with higher efficiency.

PAGE Tool proves to be of crucial importance because the changed ETs may not have well rounded verification when the tests are planned manually. PAGE tool is robust. This tool considers what functions were changed and which test cases cover those functions. This approach makes it easier to cover all corner cases. By executing the test suite that PAGE tool suggests, we can have quality assurance metric which determines how thoroughly the test suite exercises a given program code change.

4.1.3 Tool Design

The objective of the PAGE Tool is to automate the task of planning the ETs based on submitted patches. This tool applies the patch and lists out the most significant tests which need to be executed, to verify the patch. Overall tool process is divided into four tasks:

- Kernel Instrumentation
- Data Gathering
- Patch Analyzer
- Test case Identifier

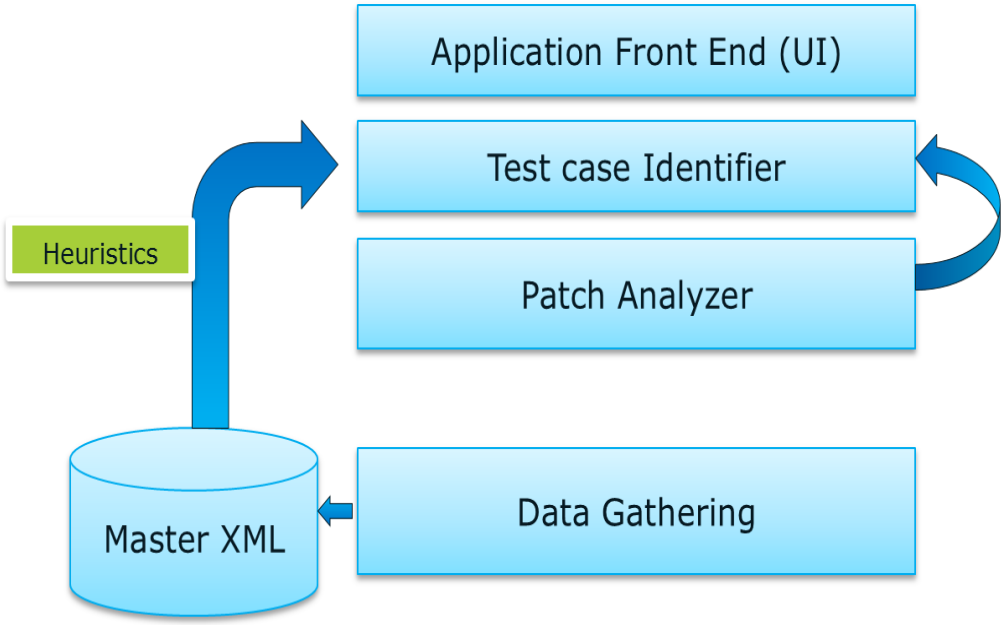


Figure 4.1: PAGE Tool Block Diagram

- **Kernel Instrumentation**:
  Kernel Instrumentation consists of inserting additional configuration in kernel config files. As Linux kernel inbuilt supports Gcov, it just needs to be enabled in Makefile for getting coverage details.

  Building the kernel with gcov configuration will generate .gcno files which will have all files structure of all the files included in kernel. From the Kernel is booted up and gcov is enabled, every kernel activity is recoded by gcov and that data is stored in .gcda files.

- **Data Gathering**:
  Data Gathering consists of storing coverage data collected during test runtime. Coverage data will be collected for all the tests available. This process produces gcda files corresponding to each test case. These gcda file along with gcno files are then processed by Lcov generating html files for each test.

  PAGE then parses all of these html files into one xml file for each test case, which are then merged into a single Master Xml file. The Master Xml is a function to test case mapping which gives the crucial information of what test cases hit which function in driver.

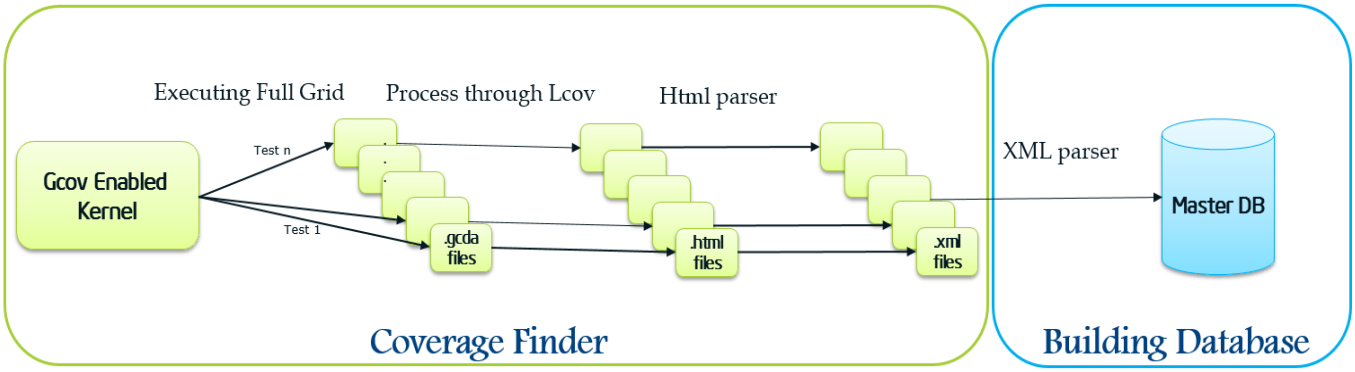


Figure 4.2: Creating Master Database

This is one time task needs to be done if any new function or file is added to the driver source code.

- **Patch Analyzer:**
  Once the patch is committed to Git, we get what files and respective content of a file changed. From this data PAGE tool parses the names of the functions that are changed and stores this patch information in an xml file.

- **Test case Identifier:**
  Output of Patch Analyzer and Mater database will allow inferring what test cases should be run for the patch. Identifying the most significant test cases is also part of test case identifier, which helps VCO to pick up the more efficient tests considering the resource bandwidth availability. If more than one function is modified for an ET, PAGE Tool gives the list of tests in an order where tests which hit all functions are shown up first.

4.1.4 Using PAGE Tool

Using PAGE is extremely simple. For any ET request, simply execute PAGE tool with patch id and platform as inputs. PAGE Tool with output list of recommended test cases.

Figure 4.3: Easy of using PAGE Tool

4.1.5 Results

PAGE Tool proposed introduced here can automate the manual work of planning test cases, leaving no scope of error and performing the job of verifying the patch in totality. Pilot execution on 3 ETs gave the expected results and clearly shows the need for tool.

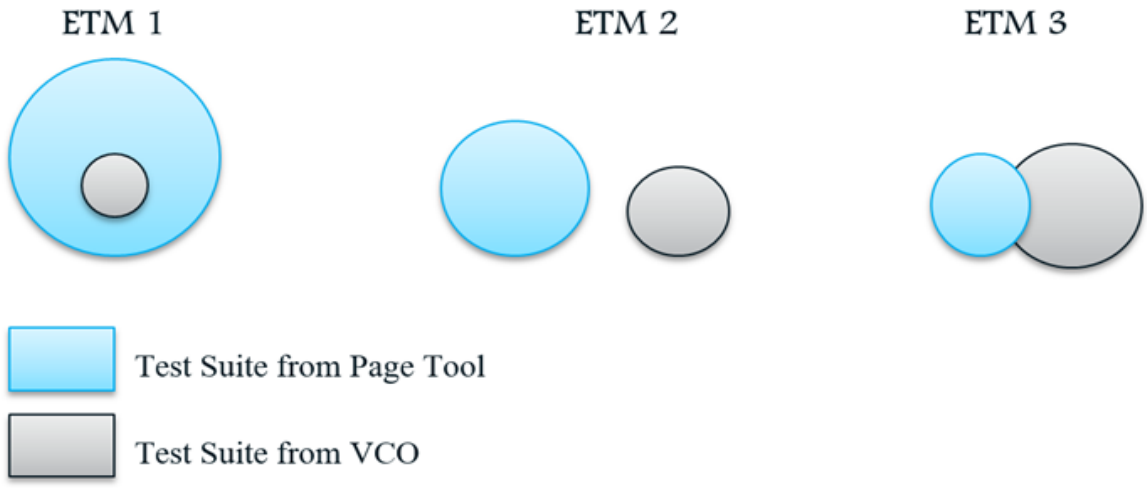


Figure 4.4: PAGE Tool Results

ET1: Test suite from tool finds areas in driver that is not covered by manually planned test suite, enabling us to plan more tests that cover otherwise untested parts of driver. Manually planned grid focus is mainly on functionality under test, PAGE Tool principle of verification is that all the tests that hit the function will have an impact if that function is modified, thus the results verified the change in totality.

ET2: This ET is different scenario in which hotplug source was modified to fix local display sleep issue. Manually VCO planned sleep related stress test on local display to verify the patch. PAGE tool lists both the sleep related and hotplug related functional tests, as the scope of PAGE tool is functional testing. Observation from this is, display configuration that has to plan for ET also depend up on code change. Issue might be related to one display type but it might impact other display types as well.

ET3: Test suite from PAGE tool is being overlapped with manually planned test suite and there are mutual exclusive tests in both suites. Manually planned suite has tests which dont have any impact on code change. There may be many tests available which covers same scenario, PAGE tool identifies and optimizes the test suite based on test priority which was defined using heuristics module.

4.2 GRAD Tool

4.2.1 Problem Statement

Display driver code for Intel platforms running on Android/Linux OS is available in Linux Kernel. The driver code is too large as it supports latest platforms as well as legacy platforms. It is important to design a validation test suite which validates driver code as a whole. It is so possible that some driver codes are not verified with any test cases, we which case we might upstream a code which is not validated and it might also go in product to OEMs.

4.2.2 GRAD Tool Introduction

GRAD (Gap analysis and Risk Assessment for Display driver) Tool give info on validation coverage over the driver code. It aims to make validation more promising and proactive to any kind of gaps or escapes. It also helps test owner about how good their test is in order to validate a feature.

4.2.3 Tool Design

Figure 4.5: GRAD Tool Block Diagram

GRAD is a three stage tool:

- Collect Test coverage data
- Collect Driver data
- Compare the above two data

GRAD Tool work at function level which means it outputs information on driver code and test suite coverage in terms of functions.

Gcov tool is used in obtaining code coverage data. Gcov must be enabled in kernel before the kernel is built. Using gcov we get list of functions of driver code that is hit during a test and store that information in an xml file in hierarchy form as folder/file/function. As seen in PAGE Tool Coverage data will be collected for all the tests and parsed into a single Master Xml file. The Master Xml is a function to test case mapping which gives the crucial information of what test cases hit which function in driver.

Using the same procedure, list of all the functions present in the driver code can be obtained using gcov tool. This information is stored in another Xml file in hierarchy form as folder/file/function.

GRAD Tool compares the two xml files, one with driver data and another with test suite coverage data, and outputs all the functions that are not called.

4.2.4 Future Scope

For each feature, list of function that must be called can be obtained from test owners. We already have the coverage data for the tests related to that feature. Both these data can be compared and difference can tell us what driver function needs to be covered under validation which otherwise would have gone without validation.

Chapter 5

Conclusion and Future Scope

5.1 Conclusion

The Automation framework proved to be very useful in Display Kernel Validation. It makes Validation generic which means only configuration files are specific to a platform/chipset. Rest all the test cases are common irrespective to platform of display type.

Python is used to create various use case scenarios where we need to validate a display feature in different situations. We have well built python library which help test run irrespective to it past states as far as it is turned on.

Automation of Automated Tests is a helpful environment to run test automatically as and wen needed. This is used in testing daily integration of new code or bug fixes in driver. It also helps keep an eye on effects of modifying code for one feature which should not break other functionality.

PAGE Tool can be used extensively in identifying the most significant test cases to verify a patch. The analysis is based on the code coverage data that we obtain using GCOV Tool. It helps not only in identifying the corner cases but also checks for regression.

GRAD Tool grades the display validation test suite. It tell how good is the validation test suit to validate the whole display driver source code.

All these framework and tools have made validation task much easier and more accurate than before.

5.2 Future Scope

GRAD Tool can be improver more to sort out which function is part of which feature. This will make test owners job easy to identify what driver function they must be validating, which presently are not validated.

Bibliography

[1] Android graphics official documentation: source.android.com

[2] Linux Device Driver: https://lwn.net/Kernel/LDD3/

[3] Gcov: https://gcc.gnu.org/onlinedocs/gcc/Gcov.html

[4] Lcov: ltp.sourceforge.net/coverage/lcov.php