Coverage Driven Verification (CDV) for Embedded Software

The Use of CDV and Virtualization to Verify Linux Device Drivers

Jason Andrews
jason@cadence.com

Embedded Systems Conference
Boston, MA
October 29, 2008
## Contents

Contents .............................................................................................................................................. 2  
Introduction ........................................................................................................................................ 1  
  Why Linux Device Drivers? .............................................................................................................. 1  
Coverage Driven Verification ............................................................................................................. 2  
Anatomy of a Device Driver .............................................................................................................. 3  
  The usb_storage Device Driver .................................................................................................... 3  
Previous Work ................................................................................................................................... 4  
Virtualization .................................................................................................................................... 5  
CDV for usb_storage .......................................................................................................................... 6  
  Verification Flow .............................................................................................................................. 7  
  Environment Features ..................................................................................................................... 9  
  Investigating Historical Bugs ......................................................................................................... 10  
Experimental Results ...................................................................................................................... 10  
  Limitations .................................................................................................................................... 10  
Appendix .......................................................................................................................................... 11  
  System Definitions ......................................................................................................................... 11  
  Software Setup ............................................................................................................................... 12  
  Host Machine ................................................................................................................................. 12  
  Target Machine running under QEMU ........................................................................................... 12  
  Hardware ...................................................................................................................................... 12  
REFERENCES ...................................................................................................................................... 13
Introduction

Controllability, observability, and repeatability have long been desirable characteristics of any hardware verification environment. This paper discusses how these basic principles used in hardware verification, as well as more advanced techniques for automation, throughput, and scalability, can be applied to software verification. The research uses a Linux Device Driver from the USB subsystem running a Linux Kernel in a Virtual Machine to show how device driver verification can be done using Coverage Driven Verification (CDV). Historical bugs from the USB storage device driver are described and the complete flow from plan to closure including the verification environment for the driver is presented. Limitations of the constructed environment are also described along with a plan for future work to address them.

There is a long history in hardware verification related to a set of execution engines for descriptions of hardware designs including Verilog, VHDL, and SystemC. These engines take the form of logic simulation, simulation acceleration, and emulation. In addition to execution of hardware design descriptions, numerous verification capabilities are added to enable debugging, assertion based verification, constrained random verification, functional and code coverage, formal verification, reusable verification components, and more. Planning and management is then added on top of the verification capabilities to oversee the entire process from the initial plan to end of the project.

Embedded software is an area that can benefit from the same technology and methodology as hardware verification. In the past, many hardware verification users were primarily companies selling chips. Today, these same companies cannot compete in the market by providing a data sheet and device samples, but instead must develop chips, create evaluation platforms by putting chips on boards, and then provide all of the software to demonstrate a working system to prospective customers. This has lead to increased investment in software by chip design companies. Over the last few years, Coverage Driven Verification has been working to fill this verification gap by enabling users to better utilize software as part of the hardware verification process in the familiar verification environments of Verilog, VHDL, and SystemC. In the future CDV will continue to focus on the chip customers who are developing software using many execution engines for simulation and emulation.

There are also new opportunities for the application of Coverage Driven Verification technology in embedded software. This paper discusses research related to CDV for Linux Device Driver verification.

Why Linux Device Drivers?

There are many types of software that can benefit from improved verification such as GUI testing and library/API verification. For the initial research we decided to examine Linux Device Drivers. Historical data indicates a large majority of software problems come from bugs in device drivers [1]. Drivers are notoriously difficult to test due to concurrency issues. Race conditions are
also common. Combine this with difficulty in creating hardware error conditions and driver verification becomes an area that can benefit greatly from improved verification techniques.

At the same time, Linux is becoming more popular for embedded products. It is now the most popular embedded operating system with about 15% market share [2]. Another benefit of Linux is the driver source code is readily available along with the history of all the software modifications. We also know by observation that the Linux kernel is written and maintained by a relatively small number of experts who are less likely to introduce bugs into the code. Drivers on the other hand are not written by kernel experts but come from a diverse group of engineers and are more likely to contain bugs.

**Coverage Driven Verification**

Coverage Driven Verification takes it roots from hardware verification and has been covered extensively in references such as [3]. As verification complexity increased it became more difficult to think about how to create appropriate stimulus to apply to a design. Of course, random stimulus can be used but the random stimulus may or may not be meaningful to the design.

The concept of Coverage Driven Verification (CDV) was developed to first define a coverage model, a set of things that needs to be observed to demonstrate verification completeness. Once the coverage model is in place, constrained random stimulus can be applied to the design to fill the coverage model. By means of parallel execution with different random seeds the majority of the coverage model is completed. Over time more execution results in less new coverage being obtained. At this point the engineer examines the remaining uncovered items and adds additional constraints to guide the tests to hit the coverage holes. At the very end, the final coverage items may require a directed test to hit the desired coverage. CDV saves the few directed tests until last rather than try to think of a directed test for every scenario and write a large number of directed tests. With CDV the majority of the stimulus is generated automatically and only the small part of coverage that is hard to hit requires manual intervention. The CDV process is shown by the diagram shown in Figure 1.

![Figure 1. Coverage Driven Verification](image-url)
This process has several major advantages:

1. The exponential explosion in the number of tests is dealt with by a scalable automated process allowing the increased quantity of tests to be addressed with additional simulation resources rather than with human resources.

2. Verification becomes much more measurable and predictable with clear metrics defining verification progress and completeness. Managing verification projects by understanding the current position and adjusting resources accordingly to achieve targets, becomes more controllable.

3. In addition to hitting planned and expected corner case scenarios, many unplanned ones will be hit also, this has the advantage that many unexpected bugs will be exposed, significantly improving design quality.

**Anatomy of a Device Driver**

A device driver is a piece of software that controls hardware in the context of an operating system. Drivers provide the interface to make hardware useful to application software running on a computer. Device drivers have three interfaces they must contend with:

1. The **hardware**: programming the hardware and reacting to its behavior.

2. The **operating system**: drivers must be compiled and run according to the rules of the operating system and use the interfaces provided by the operating system to perform their function.

3. The **applications**: the driver must provide a useful interface for applications to utilize the hardware.

The way drivers are written and how they interact with the operating system is unique for each operating system. Device drivers also operate as part of the operating system, this means they have special privileges and are essentially part of the operating system itself. This means it is crucial to have high quality drivers since poor drivers reflect poorly on the operating system as users cannot perceive the difference between a malfunctioning driver and a malfunctioning operating system.

**The usb_storage Device Driver**

This research focuses on one specific device driver in the Linux operating system related to USB storage devices. The driver is called usb_storage and is used to control memory sticks, hard drives, and CD-ROM drives that are connected to USB. This driver was chosen for a variety of reasons:

- Memory sticks are commonly available
- USB is well understood (at least from the user view)
Everybody has seen their PC hang after repeated insertion and removal of USB devices.

Memory sticks can easily be demonstrated using a laptop.

The `usb_storage` driver can be found in most Linux distributions that have the kernel source tree installed at:

```
/usr/src/<kernel version>/drivers/usb/storage
```

When a memory stick is inserted into a Linux system the `usb_storage` driver is automatically loaded by the operating system. The command

```
% lsmod
```

displays all of the drivers that are currently dynamically loaded. Drivers can be manually loaded and unloaded using commands like `insmod`, `modprobe`, and `rmmod`.

### Previous Work

Device drivers are normally tested by running them on the real hardware and observing the results. Some techniques exist to help with driver debugging but there are none that help much with dynamic verification. Most drivers are tested using some kind of system “stress test” that attempts to strain the system and the drivers to make sure they are stable under heavy load conditions. Some techniques that help with debugging are listed in [4]. A short summary is provided here.

One way to help debug drivers is to enable additional debug information to be printed by the kernel. There are kernel configuration options to enable this additional debugging in hopes that it may help track down a problem or understand behavior.

Of course, there is always the use of print statements, in the case of the Linux kernel this means `printk()` statements. Similarly, `ioctl()` interfaces can be developed and used to query the driver status or communicate with it from application space for the purposes of testing.

The use of a source level debugger like `gdb` has been a controversial issue in the Linux kernel community. Linus Torvalds has strongly been opposed to the inclusion of a `gdb` interface in the mainline kernel. Recent efforts to get this included seem to be making some progress so there is a chance that using `gdb` on the kernel will become more common [5]. Today, it can be done using a kernel patch, but it is not a commonly used technique for debugging.

Code coverage using `gcov` is also possible using a special kernel patch. This shows engineers which lines of code were actually executed, which functions were called, and which branches were taken in the driver. Although execution does not mean the code is working it does provide some useful feedback on what happened in a driver during execution.

Although all of these tools and techniques are useful for testing drivers, there are no tools for verification planning, stimulus generation, functional coverage collection, and automated and
coordinated stimulus for both the driver and the hardware it is controlling. The theory behind this research is that the same evolution that occurred in hardware verification will occur in software verification. So far it seems this evolution has been lagging in some cases because software is “soft”, meaning it can easily be changed and does not have the same cost of failure compared to building a semiconductor device. In other cases, where cost of failure is critical (such as medical or aviation applications), the problem has been addressed using large numbers of engineers to inspect and test the code so extensively that no problems can escape notice. The hypothesis is that both of these “solutions” will become more and more difficult in the future.

Virtualization

Although it has been around for many years, it appears virtualization technology is now starting to achieve success in many areas of computing. The performance advances in microprocessors, and computers in general, are enabling multiple operating systems to run on a single machine. While the primary use of virtualization has been in IT for server consolidation to save money by requiring less physical hardware, there are also applications in embedded software and software testing in general. For example, when testing software installation programs for Microsoft Windows, it is very useful to have a virtual machine running Windows with support for saving and restoring snapshots. This means the tester can start with a fresh installation of Windows in a virtual machine, and then try the installation program for the software being tested. After the trial the virtual machine can easily be restored to the pre-installation state and then it can be tried again. This process can iterate until the installation program is deemed completed and on every attempt the virtual machine looks the same as it was on the first attempt.

Virtualization techniques date back to the 1960’s, but until recently have been challenging on the commonly available PC hardware because of the sluggish performance of running two or more operating systems on one machine. Most virtualization software operates by performing some form of just-in-time binary instruction translation to transform the target machine instructions to x86 instructions. This means the performance can be very close to the actual speed of the host machine. This provides the performance required for most software engineers who commonly complain about the performance of most EDA execution engines.

Virtualization also provides an added benefit in that all of the hardware of the virtual machine is not the real hardware, but a model of hardware. In many cases, virtual machines will bridge a model of hardware in the virtual machine to the actual hardware in the host machine. For example, if Linux is run in a virtual machine and the user inserts a memory stick into the physical USB connector, the virtual machine is capable of making use of the memory stick in the usual way. Having models which bridge to the real hardware provides the benefit of utilizing actual hardware and at the same time provides the ability to insert special behavior in the models that bridge to the actual hardware. For example, many device drivers are difficult to verify because it is nearly impossible to cause all of the error conditions that may come from the hardware when using real hardware. This leads to parts of the driver that cannot be tested. Virtualization is an ideal way to insert interesting conditions such as errors in the hardware model. In the current
investigation of the usb_storage driver we used the virtual machine to randomly initiate a USB disconnect of the memory stick to verify that the driver will continue to work correctly in case where the USB memory stick is disconnected.

The virtualization software used in this project is QEMU [6]. It was chosen because it is an open source virtual machine with excellent Linux support. Since the hardware models are open source we could also easily insert error conditions and understand the behavior of the model. Unlike most virtualization software, QEMU supports non-x86 architectures. It can run ARM, MIPS, and PowerPC operating systems which make it suitable for most embedded software projects.

Another side benefit of the virtual machines is a connection for debugging. Virtual machines can attach debuggers without special support from the operating system (beyond the usual requirement to compile the software –g). Both QEMU and VMware (starting with Workstations Version 6) support this type of remote debugging where gdb is run on the host machine and debugs kernel code running in the target. Since the virtual machine can be easily stopped it is much better than trying to stop an operating system running on a physical machine. Virtual machines can also provide capabilities like record and playback to make every problem easy to replicate for debugging.

Virtualization is positioned to become the logic simulator for software engineers and may change the way software is developed in much the same way the logic simulator changed the way hardware was developed. The benefits of virtualization fit perfectly with those of coverage driven verification. The demonstration of CDV techniques connected to a virtual machine is a key contribution of the project.

**CDV for usb_storage**

This section presents the flow for verification of the usb-storage Linux driver. It describes both the flow as a user would see it if used in a real project and gives the details of what was done in the context of research for the usb_storage driver. Verification starts with the establishment of a verification plan and ends with a set of regressions which measure verification against the plan.

The usb_storage driver environment is somewhat unique in that the verification tools are run on the host machine and the driver being verified is running in a virtual machine. Typical verification tool usage does not involve separate machines, but a network socket connection [7] makes this possible with only minor changes. The multiple machine host/target scenario is typical for device driver development since the engineer writing the driver normally compiles the code on the host machine and then moves it to a target machine for testing. This eliminates any danger of a malfunctioning driver causing data loss due to a crash or other undesired behavior.
Verification Flow

Verification Planning

Verification planning involves determining all of the items that must be exercised and measured as part of the verification process. Planning is done by capturing important items to be verified in the form an English document that be fed forward and read by verification tools. The verification plan defines the verification problem, when the plan achieves full coverage then verification is complete.

For the usb_storage driver there are items relating to the API used to call the driver from application space, items in the driver itself such as interesting values for a semaphore, and items relating to hardware behavior, such as a USB disconnect. There are also additional items such as an error code returned by an operating system function. Some examples of elements in the verification plan are:

- Open a file in all possible modes: read-only, write-only, write-read
- Open multiple files at the same time
- Open a file when the memory stick is disconnected
- Receive all possible error codes when trying to open a file
- Write data to the memory stick using data transfer sizes in many different ranges
- Disconnect the memory stick during read and write operations
- Define coverage values for the device driver semaphore to understand how often the driver was waiting for another thread to proceed

The concepts of verification planning for software are nearly identical to those in hardware verification except the items to be measured are not bus transactions or signals, but variable values and events, function arguments, and function return values.

Environment Creation

Verification environment creation involves creating an e verification environment with simple ports for variables and method ports for C functions to be called as well as automatic sequences and coverage. For more information about the e verification language refer to reference [8]. The flow for a device driver must be somewhat altered since the driver cannot be called directly. For this investigation, a C application was created that will be run in user space to call the device driver.

Additional modifications to the environment were done to add support for multiple programs running in parallel, to create the connection to the QEMU virtual machine to control the USB disconnect, and the fact that the host and target machines are different.
Sequence and Test Creation

Once the verification environment is in place additional sequences can be added to the sequence library and tests created on top of those. For the usb_storage driver a library of about 50 sequences was created and approximately 20 tests were developed. To stress the usb_storage driver multiple applications were run in parallel to get concurrent operations in the driver, with as many as 16 applications run in parallel all calling the driver to access the memory stick.

Some of the tests used the virtual machine to trigger a USB disconnect while the applications were accessing the memory stick.

The sequences mainly control the C test program and tell the virtual machine USB interface to disconnect at specified or random times. Additionally, an extra driver called usb_monitor was added to the kernel to watch the interesting usb_storage data structures for monitoring and coverage purposes. Sequences can also access this driver using an ioctl() interface.

Execution and Coverage Analysis

Executing tests is done in the same way it is done for hardware verification. The verification environment is loaded as well as the test file. Tests can be run with the default seed or with a random seed. The coverage GUI displays the achieved coverage for a particular test.

Execution of all of the target machine activities was completely automated by the verification environment. This means all of the test setup and copying of any results back to the host is done automatically. Any machine with a memory stick inserted can be used for running the tests. This enables a methodology where it is possible to run on many machines in parallel. Since the memory stick is still a physical resource that cannot be shared by multiple virtual machines, it’s not possible to run many copies of the target machine on the same host, but if the physical memory stick was not used, but instead replaced by a model it would be possible to run many virtual machines on the same physical machine.

Software Code Coverage

Code coverage has always been one of the commonly used metrics for measuring software execution. CDV focuses on functional coverage metrics which are valuable in meeting verification goals, but just as in hardware design and verification code coverage is also a useful metric to examine which code was executed. A coverage adapter was used to import code coverage data into the functional coverage model. This enables the gcov data collected on the usb_storage driver to be measured across multiple verification runs.

Since the device driver is running in the target system and the test on the host, the coverage results must be copied from the target back to the host machine at the end of test execution to be added to the coverage results. This was done using networking between the host and target via scp. Code coverage results are then displayed in the coverage GUI with metrics for statement, call, and branch coverage. The metrics are available in the coverage GUI. The gcov annotated source file is also available from the GUI.
Regression with Results Compared to a Plan

Although there are many ways to write scripts and deploy regressions tests on a farm of machines, it is important to match the results against a plan.

The usb_storage driver regression is fully automatic. To automate this requires that there is no user interaction necessary on the target system. The tasks of mounting the memory stick, loading and unloading the usb_storage driver and usb_monitor driver, collecting gcov results, capturing messages that occur in the system logfile are completely automatic. Making everything controllable from the host machine enables a complete regression environment. The results are easily mapped to the verification plan by loading the XML description of the verification plan and seeing the coverage results. For the usb_storage driver a complete verification plan was not created, but high coverage was achieved for the parts of the plan that were created.

Environment Features

The usb_storage driver environment has a number of useful features. First, a C application on the target system to exercise the usb_storage driver and connect to the test environment using a networking socket. Support for many copies of the test application, all connected to a single instance of the test were also created to further stress the driver. The test application has basic mount, open, close, read, and write functionality for both the file system interface to the memory stick as well as the block interface. Support for gcov software code coverage on the usb_storage driver to provide statement, branch, and call coverage metrics that are integrated into the coverage model. An additional driver to monitor the usb_storage driver data structure was created to collect coverage on the usb_storage driver without requiring any changes to the original source code. A mechanism to monitor system log files on the target system and pass them back to the host so they are automatically included in the rest of the verification environment output stream as they occur was created. The ability to coordinate USB disconnects from within verification sequences by connecting the test environment to the QEMU virtual machine enabled automated verification USB. The final feature to note is that all operations can be controlled and automated from the host machine. There is no need to login or manually retrieve any information from the target system. The constructed environment is shown in Figure 2.

Figure 2. usb_storage Verification Environment.
Investigating Historical Bugs

As a way to confirm the usefulness of the constructed environment the history of the usb_storage driver was examined to see some of the typical bugs were that were reported and fixed over time. As suspected, many were related to race conditions, data structure locking, and USB disconnects. To see how effective the constructed environment was at finding bugs we decided to run with a version of the driver that was not the latest and greatest, but was known to have a bug in it that we should be able to identify when the test suite was run.

The specific bug that was investigated is detailed in [9]. It has to do with the USB disconnect and how the usb_storage driver interacts with the SCSI subsystem. It turns out that USB storage devices are treated as SCSI drives so there is interaction between usb_storage and the SCSI subsystem.

We were able to hit the target bug using the constrained random environment combined with disconnects of the USB memory stick from the virtual machine, but for various reasons explained in the section on limitations, it was not as quick or consistent as we had hoped.

Experimental Results

The application of CDV to a USB device driver has demonstrated good potential to find historical bugs in the driver and bring a new planning based coverage driven verification methodology to software that is outside of the traditional hardware verification usage. Further application to additional drivers and historical bugs will provide clear evidence of success.

At the same time a number of issues were discovered that need to be addressed to make this methodology usable for a general audience. The majority of the issues are small and many are in the process of being addressed by normal development or will be addressed in the future.

Additionally, there is also the open question about how to train engineers to use the proposed flow. Clearly, the flow requires a verification background in planning, environment design, constrained random generation, etc. In the early days of hardware verification it became important for chip design teams to separate into design and verification so the verification skill sets could be acquired. The same shift does not yet exist in software verification. Today’s structure is usually designer (or developer) and tester (or quality assurance, QA). Although there is often a gap in pay and respect between verification engineer and hardware designer, the gap between developer and QA in software is probably even larger (and not always justified).

Limitations

Numerous other minor details were discovered related to collecting code coverage, measuring functional coverage on the driver itself, logging driver messages to the verification environment, and constrained random timing of hardware stimulus (such as USB disconnects) that will help guide methodology development to better perform software verification. Issues were also
uncovered related to the functionality of the network socket and its ability to handle a large number of connections and still provide the required stress on the driver. Another issue related to the socket is how to detect a driver or system hang or correctly timeout the tests when a failure occurs. While none of the issues are major, all must be improved to have an environment that useful for general purpose driver verification.

The other issue, which is not as easy to solve, relates to the repeatability of the environment. As the environment is constructed today, the tests are not repeatable, even with the same random seed. We found that we can induce the historical bug sometimes, but not always because of the timing in the system. A virtual machine is better than a physical machine in terms of controllability, but remember that it connects to a network, has a keyboard and mouse, and is an operating system running about 100 processes at any given time. It is impossible to send some stimulus via network connections and expect the machine to process and execute the information with the same timing in such an environment. In order to be truly repeatable, it seems we would need to close off any external communication between the outside world and the virtual machine. We would then need to somehow connect the clock (or similar timing control) of the virtual machine to that of the verification environment to keep these in sync. Although I mentioned the virtual machine is quite possibly the equivalent of the logic simulator for software engineers, it doesn’t behave as nicely as a logic simulator because it has uncontrolled input from the outside world.

One proposal to get deterministic results is to connect to the virtual machine itself instead of each individual user space application. Virtual machines like QEMU have the ability to access memory directly (as was previously mentioned with the ability to use gdb on the operating system). If the communication with the target system memory was via this backdoor in the virtual machine and the test was synchronized to the virtual machine clock they could operate in lock step synchronization instead of the free-running synchronization that is present in the initial usb_storage environment. Record and playback techniques available in some virtual machines can also be investigated as a solution.

Appendix

Some additional details for interested readers are provided here about the constructed usb_storage environment.

System Definitions

Target – In the target system of development and debugging, the developer does all work on his application or driver from within the targeted operating system. The tools used are limited to the target machine. Problems in the target outside of the new development, such as other driver bugs or system instability, may influence or cause problems to appear that are not directly linked to the new application or driver.
Host and Target – In the host and target system of development and debugging, the driver or application is developed on the host system. Once the code is ready, it is moved to the target system for debugging and testing. The target system is accessed via a COM, USB, or Ethernet port to directly connect to a remote debugger running on the host system. Development in this manner reduces the influence of target OS problems and is using a quality assurance “black box” methodology. This is the historical method of development.

Host and emulated Target – In this development method, the host is used to emulate the target operating system. The development is then contained within a single machine. The driver or application is developed on the host system and moved to the target system for debugging and testing. The environment contains the advantages of the Host and Target development environment along with other benefits. Improvements to the target operating system simply require a new emulation image to be distributed. No scheduling of limited target hardware resources is required in order to complete work. Target system crashes and hangs requiring reboots of the target only need to kill the emulator and restart with a debugger connected. This is the preferred method of development and easiest to use for demonstration. The emulated target also enables additional capability such as injecting hardware errors to observe software responses.

Software Setup

The setup being used for this plan is the Host and emulated Target. Software for the configuration is as follows:

Host Machine

Specman 6.2 including Incisive Software Extensions (ISX)
Enterprise Manager 6.2
QEMU 0.9.0 – Open source Linux OS emulator

Any version of Linux supported by Specman (may be run under VMware). Most work was done with Suse 10.2 and Ubuntu 7.04
The host machine must have the usb_storage kernel module disabled so that the memory stick can be used in the target system.

Target Machine running under QEMU

Uses the 2.6.13 kernel running under Arch Linux (www.archlinux.org) to minimize the size of the file system needed.

USB core support enabled

Hardware

Any USB flash drive
REFERENCES


