CONTINUOUS AUTOMATED TESTING OF SDR SOFTWARE

Jeremy Nimmer, Brian Fallik, Nick Martin, and John Chapin
Vanu, Inc.   Cambridge, MA, USA   `info@vanu.com`

ABSTRACT

Software testing is vital for the success of any SDR engineering organization, given the complexity and reliability requirements of radio communications. At Vanu, Inc. all waveforms under development are tested 24 hours a day by an automated system. The test system has been running continuously since 2002. In that time it has evolved in sophistication and become an integral part of the company's software engineering methodology.

As soon as an engineer checks a new software version into the code repository, the automated test system checks it out, compiles versions of all derived and inter-operating software, and commences testing. Test results are reported continuously to all interested members of the engineering team via web-based reports and an online chat room.

Since Vanu waveforms are entirely implemented on general purpose processors using standard operating systems, the tests can run on standard servers without loss of fidelity. A radio channel simulator enables end-to-end testing of communication among multiple servers connected by Ethernet. Since our radio heads exchange digital samples with the baseband processing server via Ethernet, inserting the simulator rather than actual radio hardware is fully transparent to the software under test.

This paper describes the architecture of the test system and the design of its major components. Synergies with aspects of the Vanu, Inc. SDR design approach are highlighted.

1. INTRODUCTION

Testing is vital for the success of any SDR engineering organization. However, testing SDRs only in their full and final form—hardware and software combined—is phenomenally expensive. Not only must testers have access to enough radio hardware to complete the test suite in a reasonable amount of time, but the ability to test at all is gated on the availability of the hardware. If the hardware platform is new or under revision, software testing may be impossible for months while the hardware platform is completed and verified.

This paper suggests solutions to the challenges of software test of SDRs by explaining the test practices in use at Vanu, Inc. Vanu uses software radio technology to build cellular basestations, radio access networks, and related products. The company earned the first FCC certification ever granted under the SDR rules, and has deployed radio access networks for multiple customers in North America.

At Vanu, all waveforms under development are continuously tested 24 hours a day by an automated system. The test system runs on off-the-shelf x86 servers, and requires no radio hardware. Expanding the system's capacity is as simple as purchasing a new server or repurposing an unused workstation. The test system is detailed in Section 3.

Testing SDR software without using the underlying radio hardware is made possible in part by a Network Channel Simulator (NCS). The NCS supports the digital baseband interface used by our radios and simulates varied RF channel models as directed by the specific test case. The NCS enables full-system software tests that match real-world conditions in as much detail as the channel model provides. The NCS is detailed in Section 4.

The automated test system, channel simulator, and other techniques have been effective in both removing defects before software is fielded and accelerating software development. Section 5 notes synergies that make our approach particularly effective, and provides suggestions for other development organizations working on similar problems.

2. VALUE OF TEST AUTOMATION

Improving test capabilities provides significant benefits for SDR product development, for a number of reasons.

- Testing represents a significant fraction of the engineering investment by SDR developers.
- The quality and thoroughness of testing directly impacts end user experience with the product.
- Regulatory agencies such as the US Federal Communications Commission are concerned about interference due to software failures, so the quality of testing can directly impact whether or not a device is certified for sale.
- New device types such as cognitive radios are being introduced that require a high level of software assurance for their complex spectrum access subsystems.

Within this general area, test automation is a particularly valuable way to improve SDR engineering.

- Automation reduces staffing requirements.
- Fully automating a company's software tests supports efficient use of expensive resources such as test stations.
- Discovering errors earlier in the development cycle makes them cheaper to fix. Without full automation, it may be weeks or months between the creation of a software module and the first time it is run through the full test suite. A continuous automated testing system like the one we describe reduces this to a few hours, providing significant engineering efficiency gains.

Data that supports these observations is rarely published in the open literature on SDR. One valuable source of information is a presentation given in April 2006 by Joe Miller of General Dynamics C4 Systems [1]. He surveyed seven years of experience by GDC4S developing the Digital Modular Radio (DMR) Maritime, a 4-channel SDR device with more than 10 waveforms and 5 crypto algorithms.

Miller reported the following level of effort per software release of the DMR, which was normally performed once or twice a year.

- 6,720 radio-channel hours of testing (~3.4 man-years)
- Four weeks of 24x7 testing using 150 radio channels
- Greater than 40 test stations

During seven years of development, GDC4S invested over 100 person-years of effort on testing, which corresponds to about 15% of their software and support staff effort.

- Test procedure generation > 71,000 hours
- Incremental smoke test > 70,000 hours
- System test > 50,000 hours

The reported numbers may actually be conservative as a fraction of total effort. Miller did not describe what positions were included in the support staff category. Removing personnel such as management and accounting, if included in that category, would increase the effort fraction on testing to well above 15% of software engineering.

In our experience, both in projects internal to Vanu and in working with other SDR development organizations, this level of effort on testing is within the typical range. The high level of effort involved means that test automation techniques—which reduce staff effort, improve usage of test equipment, and speed the tests themselve—are of high value to SDR engineering organizations.

3. TINDERBOX TEST SYSTEM

3.1 Initial History

Since the company's inception in 1998, Vanu, Inc. has followed a software process where some form of automated self-test, such as unit tests or system tests, is always developed in parallel with waveform software. By running the self-test before committing changes to the source code repository, software engineers can confirm that their changes do not break the software.

With growth in the organization, the honor system for executing these tests became unworkable—engineers were not running every self-test before every commit. Furthermore, the software was also now being ported to multiple processors, such as x86, ARM, and PowerPC. Even conscientious developers with x86 workstations who ran the tests regularly could not completely guard against introducing errors on other platforms.

As a result, the company introduced a nightly build system during the summer of 2001. Every night, a batch job would wake up, check out the source code from the repository, compile it, and run the self-tests. Upon completion, it emailed test results to a distribution list of interested engineers and managers.

This reduced the extent of the problem, since an error would be detected by the next morning at the latest. However, erroneous software committed early the in the day could still interfere with other work during that day. The engineers needed even more immediate feedback.

Not surprisingly, Vanu engineers were not the first to encounter this problem, and the solution—continuous testing—was not new. In the fall of 2002, Vanu began to develop and use a continuous automated testing system.

3.2 Tinderbox Architecture

The continuous automated testing system in use at Vanu, called Tinderbox, is built on the tinderbox system originally developed by the Mozilla open-source project [2]. This section explains the system architecture of the Vanu Tinderbox, as shown in Figure 1.

In an environment of networked hosts, independent build servers check out a project's source code from the central repository, compile, measure, and test it, then email the results to a central database server and web display.

The web display offers a number of summary and detail views of the current test results. A summary is available as a browser sidebar that shows one line for each software configuration. This lets developers monitor an overview of all tests. The primary detail display is a temporal "waterfall" for each software module or project, as shown in Figure 2. The page shows when each configuration was built and tested, with a color code indicating success or failure. By clicking on a specific configuration, a developer can view the full compilation and test log for that build. This assists in investigating failures.

The central web server also maintains a database of test results over time. Vanu, Inc. has developed an instant messenger *bot* that monitors the build status database and posts to an internal engineering chat room when the status changes. An example is shown in Figure 3.
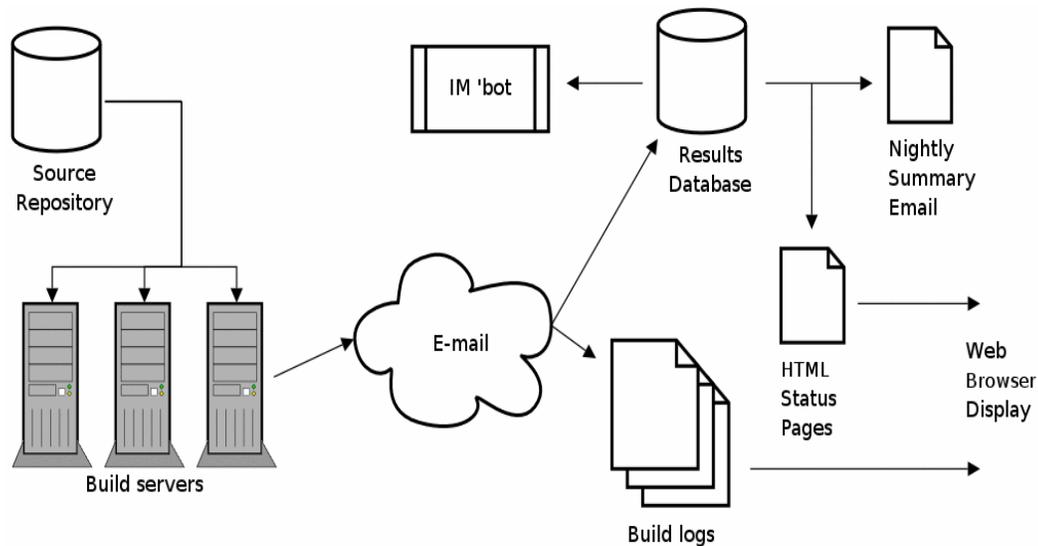
Figure 1: Tinderbox architecture in use at Vanu, Inc. Continuously-running build servers check out project source code and configuration settings from a source repository, build and test the projects' software, and email results to an automated collection system. The collection system saves the results to disk, updates the projects' status in a results database, and generates HTML status pages. Engineers' web browsers automatically reload the status pages for updates. An instant messenger *bot* also monitors the results and posts real-time notifications to a chat room.

3.3 Features of Automated Tests

By automating the build, test, and result collection process, Vanu, Inc. has been able to amplify the benefits of the unit tests that we develop along with every software module.

Our tinderbox tests all software using the Valgrind [3] memory checker (similar to Purify or Insure++). Valgrind helps identify memory leaks and uninitialized memory usage in C and C++ code, but imposes as much as a 10x performance penalty at runtime. Valgrind runs by default as part each developer's own testing prior to committing changes, but developers are free to selectively skip it if time is short, knowing that the tinderbox will run the full test suite soon enough.

We also build and test our code under a variety of different platforms and configurations. It would be challenging and expensive to provide every developer with 10 different platforms on which they are expected to test; in contrast, it is straightforward to configure 10 different platforms for use as the tinderbox build servers, which all projects can share. Using the tinderbox, we are able to test on a mix of Linux distributions, kernel versions, compiler versions, compiler and linker settings, and build system tools (make, autoconf, etc.).

We automatically test against a range of compilers and compiler diagnostic settings. Our tinderbox currently provides builds for the GNU compiler (GCC) versions 2.95 through 4.1, and the Intel compiler (ICC) from 6.0 through 9.1. We also vary the compiler diagnostic settings (such as turning on additional warnings), optimization levels, and CPU-specific compilation support (such as tuning for PentiumPro vs. Pentium 4).

Testing such a variety of compilers helps us ensure portability. Traditional DSP software engineering organizations select a single compiler tool chain for their work. However, this approach usually results in very high costs when porting to a new platform. By making multiple tool chain compatibility part of everyday software engineering, we avoid building incompatibilities deeply into the source base.

Similarly, enabling extra warnings helps us remove questionable or non-portable constructs from our code. The varied optimization levels help us avoid compilation-specific bugs that may not be revealed if only full debugging and full optimization levels were tested.

To supplement the compiler diagnostics, we are also able to automatically and continuously run a commercial C/C++ static checker (Flexelint [4]) over our entire source code base, reporting results through the same web page and announcement system as a traditional compile-and-test result. The static checker identifies over 800 kinds of programming mistakes, most of which are not detected by traditional complier diagnostics.

We also configure builds to compile and test each project's code using the gcov test coverage tool [5]. As a result the tinderbox automatically creates up-to-date test coverage reports for all code in the repository. This is a convenient way for developers to identify missing tests.

The automated build-and-test system also makes it easy to mix different versions of our source code modules. We regularly build each waveform against both a stable, verified suite of dependent libraries, as well as the newest

Tinderbox Status Page tree: Halifax

Back to Tinderbox home | Halifax project wiki | Add to Mozilla Sidebar

Test coverage summary (directory)
Built documentation: halifax.ps, supplement.ps, doxygen (doc directory).

Build Time	sarge26 debug in-place depend	sarge26 debug separate clobber gcov	sarge26 no debug in-place depend	x sarge gcc34 debug in-place depend	x sarge26 icc81 in-place depend	x sarge26 icc91 in-place depend noval
09/23 17:45			l L			
17:35				l L		
17:25						
17:15						
17:05		l L			l L	
09/23 16:55						
16:45	l L					
16:35						
16:25				l L		
16:15						
16:05			l L			
09/23 15:55						
15:45					l L	
15:35						
15:25						
15:15						l L
15:05				l L		
09/23 14:55						
14:35			l L			

Figure 2: Web status "waterfall" display of current builds. Each build configuration is its own column. Shaded cells indicate that a build was active at the given time; the cells are colored to indicate success, test failure, or build failure. Links in each shaded region lead to the corresponding build logs. Links at the top of the page provide the latest test coverage reports and generated project documentation.

developmental versions of those libraries. This continuous integration testing helps find integration problems earlier, thus reducing their cost to fix.

Finally, we are able to continuously integrate with new platforms and configurations. We maintain a separate pseudo-project where our current software is built on new or newly-integrated platforms (for example, release candidates put out for testing by Linux distributions). As developers have a free moment or slack time, they can take a break and experiment with the new platform, incrementally fixing any porting issues or test failures reported by the tinderbox.

4. NETWORK CHANNEL SIMULATOR

The Network Channel Simulator (NCS) application facilitates end-to-end tests of Vanu waveform applications. The NCS is a simulator written in C++ that leverages existing and mature software technologies to emulate communication channels between multiple radios. The NCS supports pluggable channel models for testing a range of over-the-air environments, including noise, fading, interference, distance variations, and other effects.

A key enabler for the NCS is the use of RF Over Ethernet (RFOE) [6] in Vanu systems. RFOE uses Ethernet to exchange streams of RF samples between the baseband signal processing unit and an RF Front End (FE) that contains all the analog components of the radio. The "Network" in NCS refers to the role of Ethernet in the simulator architecture.

Vanu waveform implementations send and receive ethernet packets containing RF samples. The waveforms normally connect directly to a packet socket bound to a physical ethernet port. To insert the NCS, we instead bind to virtual ethernet devices provided by Linux's `tap` facility, so the waveform software is unchanged. (Most OSs support a similar mechanism.)

Typically, the NCS is used as part of end-to-end communications tests exercising all waveform software modules and layers. These system tests complement the unit tests built for each module. We begin system test as early in the development cycle as possible, often before coding on most software modules or features has even been started. Early and continuous end-to-end system test:
- exposes integration complexities early;

```
[16:30:15] <tinderbox> === Status BUILD_FAILED from Halifax x-sarge24-gcc34-dbg-inp-dep build ===
[16:37:47] <percent> halifax is me
[17:01:15] <tinderbox> === Status TEST_FAILED from Halifax sarge26-ndbg-inp-dep build ===
[17:18:15] <tinderbox> === Status TEST_FAILED from Halifax x-sarge24-icc81-inp-dep build ===
[17:44:15] <tinderbox> === Status TEST_FAILED from Halifax sarge26-dbg-sep-clbr-gcov build ===
[18:12:15] <tinderbox> === Status TEST_FAILED from Base sarge26-dbg-inp-dep build ===
[18:15:15] <tinderbox> === Status TEST_FAILED from Halifax sarge26-dbg-inp-dep build ===
[18:26:15] <tinderbox> === Status TEST_FAILED from Base sarge24-dbg-inp-dep build ===
[18:28:51] <jnimmer> tinderbox: Base break is vichronio TimeConverter
[18:30:05] <rico> that'll be me...
[18:31:03] <rico> ah, I've run afoul of sigc 2
[18:35:54] <jnimmer> rico: you want '*this', not 'this'
[18:37:15] <tinderbox> === Status SUCCESS from Halifax x-sarge24-gcc34-dbg-inp-dep build ===
[18:38:00] <percent> yay!
[18:51:15] <tinderbox> === Status TEST_FAILED from Base sarge24-ndbg-inp-dep build ===
[18:51:15] <tinderbox> === Status SUCCESS from Halifax sarge26-ndbg-inp-dep build ===
[18:51:47] <rico> tinderbox: Base fix landed
[19:04:15] <tinderbox> === Status SUCCESS from Base sarge26-dbg-inp-dep build ===
[19:04:44] <rico> tinderbox: whee.
```

Figure 3: Excerpt from a Vanu, Inc. internal chat room log. An automated system monitors build status and posts announcements of significant events under the username tinderbox; the other postings are by engineers. In this case, the build failure of the "Halifax" project was caused when an engineer accidentally omitted one file from a commit. His personal build worked, but the tinderbox caught the discrepancy in a fresh checkout of the source code. In the "Base" project, the tinderbox uncovered a portability problem between versions 1.0 and 2.0 of the **sigc++** API. Version 1.0 (as used by the engineer's own testing) provided a pair of overloaded functions, but version 2.0 removed one. The tinderbox compiled against all versions in turn, and the error was uncovered

- tests waveform load, startup, and shutdown functions;
- develops a suite of tests that can exercise the actual SDR hardware as soon as it is available; and
- often discovers unexpected and untested code paths.

An obvious limitation of all-software system tests is the quality and maturity of the NCS itself. NCS faults can be hard to isolate, since it is frequently unclear whether a particular behavior (e.g., excessive bit errors) comes from an NCS issue or from immature waveform software. Furthermore, accurately emulating FE behavior and timing makes the NCS complex. An SDR development organization that starts using NCS-style tests should expect to make an ongoing investment in iterative improvements.

4.1 Timing-independent testing

The Vanu RFOE protocol includes a timestamp in each packet of RF samples. Waveform applications are insensitive to their wallclock execution rate; they pay attention only to the RFOE timestamps. As a result, simulations are able to run faster or slower than real time.

To demonstrate this, we performed an experiment in which the same system test was repeated on multiple platforms with different execution rates. One of the platforms was a workstation with a single Intel Celeron CPU at 2.6 GHz. The other was a server with two Dual-Core Intel Xeon 5100 Series CPUs at 2.66 GHz, for a total of four processing cores.

The system test consisted of two instances of a waveform application, configured as the two ends of a point-to-point radio link simulated by the NCS. All processes including the NCS ran on a single machine.

The waveform was designed for a real-time sample rate of 400 kilosamples per second. The actual rate of execution varied depending on the test platform, as shown in the table.

Hardware	Optimizations	KSamples/sec
workstation	−O0	72.5
workstation	−O2+sse	151.6
server	−O0	650.1
server	−O2+sse	2,584.6

The waveform test results were identical in all runs. This illustrates that the wallclock execution speed can vary significantly without affecting the behavior of the software.

Timing-independent testing has been highly valuable for our SDR development. It reduces test duration for tests that can run faster than real time, enables use of sophisticated channel models that run slower than real time, and supports functional testing of early software versions that have not yet been optimized to run in real time.

Of course, errors due to timing races in the SDR software may be masked by executing it at a non-real-time speed. However, execution at a range of speeds is often an effective way to expose timing errors that may occur only rarely when operating at the target speed.

4.2 Hardware-independent testing

An additional benefit of NCS testing is that the test system itself is hardware-independent. It can run on any of a range

of platforms. The obvious benefit is testing in advance of target SDR device availability. There are other advantages:

- Assigning testing tasks to machines is simplified.
- Older hardware can perform testing. We have found it cost-effective at the company level to reuse workstations retired from developer desktops as test servers.
- Developers can repeat any test using their desktop.

5. SOFTWARE RADIO SYNERGIES

The continuous automated test system has been particularly effective at Vanu because of the way we implement SDR [7]. Vanu Software Radio minimizes the use of embedded devices such as DSPs or FPGAs. We execute most or all of the high speed signal processing functions of the radio as portable application-level software running on a general purpose processor and standard OS.

Our focus on software portability enables rapidly exploiting Moore's Law improvements in hardware components. This is important because alternate engineering approaches, which employ less portable firmware technologies, result in hardware lock-in and hence rapid obsolescence. Our focus on general-purpose processors and standard operating systems enables use of off-the-shelf components. This allows customers to acquire exactly the hardware that matches their requirements without paying expensive NRE for hardware design. Depending on the market, Vanu, Inc. customers have selected low-cost high-volume rackmount servers, high-reliability NEBS compliant blades, flight-qualified avionics units, and small-size battery-powered devices.

The all-software approach makes continuous automated testing both easier and more effective.

- The test system can leverage off-the-shelf servers while still providing the exact operational environment that the radio software will experience in the field. For radio designs with significant firmware components, test systems must use either slow simulators or expensive hardware-in-the-loop setups.
- A wide range of sophisticated test, analysis and logging tools are available for these standard platforms. These tools were easily incorporated into the automated test system as described in Section 3.
- When automated testing detects a problem, any test can be repeated on an engineer's desktop workstation with full fidelity. This dramatically reduces the pressure on the test equipment. In engineering organizations where all tests must be run on a small number of dedicated systems, engineers normally must sign up for a time slot and then wait before being able to repeat a test. This ongoing contention reduces engineering productivity for critical analysis and debugging activities.

As a result of these effects, a tinderbox-style test system would provide somewhat lesser benefits to most other SDR developers than it does to Vanu, Inc. However, the benefits could still be significant, particularly if care is taken in a few areas when setting up the engineering environment.

- The waveform software should be stored in a source code repository, so the test system can regularly check out the latest versions transparently to the engineers.
- The tools used to build and deploy the waveform software must be scriptable. Integrated Development Environments (IDEs) sometimes require that GUIs be used for certain operations, which is highly undesirable in this setting.
- A hardware-in-the-loop test system, if used, must support power-cycling of the target hardware under software control.
- Any other hardware devices used, such as channel simulators or oscilloscopes, must support remote software control of their configuration and access to their status reports.

6. CONCLUSION

Continuous automated testing, if used effectively, reduces development cost and time while improving software quality. However, achieving these benefits requires sustained investment by an engineering organization. The tools must be developed and iteratively improved. At the same time, the engineering culture must evolve to make best use of the tools.

The test system described in this paper evolved over many years. Our tinderbox system, Network Channel Simulator, RF over Ethernet, and all-software waveform designs work together to support a high-efficiency engineering and test process. While each of the components are useful, it is the way they interact that truly accelerates software development. This testing approach has been a key contributor to effective SDR engineering at Vanu, Inc.

REFERENCES

[1] J. Miller. SDR Development Realities. In AIE Military Radios Conference, April 2006.
[2] Tinderbox 2.0. http://www.mozilla.org/projects/tinderbox/.
[3] Valgrind home. http://www.valgrind.org/.
[4] Flexelint for C/C++. http://www.gimpel.com/html/flex.htm.
[5] gcov. http://gcc.gnu.org/onlinedocs/gcc/Gcov.html.
[6] G. Britton, B. Kubert, and J. Chapin. RF over Ethernet for Wireless Infrastructure. In Software Defined Radio Technical Conference, Nov. 2005.
[7] J. Chapin and V. Bose. The Vanu Software Radio System. In Software Defined Radio Technical Conference, Nov. 2002.