A 30 Minute Project Makeover Using Continuous Integration

JL Gray
Gordon M^cGregor
Verilab, Inc.
Austin, Texas, USA

***Abstract*—You've just spent a week working on a complex testbench change. You've regressed your changes and are ready to check them in. First, though, you pull in updates from other users and rerun regressions. Now you find that the testbench no longer compiles, or perhaps fails to run a basic test. You're late delivering your code and your manager is breathing down your neck. But it's not your fault! A Continuous Integration (CI) server can go a long way to preventing these situations. This paper describes the features and setup of one CI server and how you can apply it to your design projects, with minimal effort. We consider both the technical and managerial challenges of using continuous integration.**

Keywords-component; Software testing; Computer simulation; Logic design;

I. INTRODUCTION

You've just spent a week working on a complex testbench change. You've regressed your changes and are ready to check them in. However, you notice that there have been additional check-ins since you last updated your work area. As a thoughtful engineer you merge these changes and attempt to run a regression one more time to make sure everything still works. But you find that the testbench no longer compiles, or perhaps fails to run a basic test. You are certain that your changes are unrelated to the failure. To prove it you now have to trace back though multiple check-ins in a clean work area to root-cause the issue. Now you're late delivering your code and your manager is breathing down your neck. But it's not your fault!

If you're like most engineers, the above scenario has happened to you. And you have almost certainly been the cause of the problem for other engineers on your project. If we told you that 30 minutes of work could bring you a huge step closer to solving this issue, would you be interested?

In this paper, we will describe the problems that most teams face in managing a quality code base and a solution that we implemented on a recent project to resolve the issue. The solution is setting up a Continuous Integration (CI) server to monitor and test the quality of checked-in code. We picked the Jenkins Continuous Integration server. Jenkins is a freely available, open-source server that can execute regressions whenever check-ins have been made to a revision control system. Installing and managing the server requires little to no interaction with the IT department, and the initial setup can, in most cases, be performed in about 30 minutes. Jenkins works with most common version control systems and can integrate with your existing regression scripts. The basic requirements for implementing such a system are covered and the typical steps required to configure Jenkins will be explained.

After the initial setup, a period of tuning the CI server is often required. Tuning parameters such as frequency of polling, number of parallel jobs run by the CI server, and LSF or Sun Grid Engine slots allocated to the server at different times of the day will be discussed. There are also various modifications that can be made to existing regression scripts to enhance the integration with the Jenkins reporting methods, and these are also considered in this paper.

Finally, and perhaps most importantly, there are often management and engineering objections that arise when the topic of implementing continuous integration techniques comes up. For example, engineers (especially designers) often resist the idea that they need to make atomic check-ins that actually work.[1] Often engineers will discover an issue before they make a check-in, but assume the problem was caused by someone else and is not something they need to be concerned about. Managers who are unfamiliar with the concepts of CI may be hesitant to change a team's working model. They will also raise concerns about the number of engineers, licenses, and compute nodes required to adequately support the CI system. Each of these objections, and more, are covered along with tips for addressing them.

After reading this paper, an engineer will be able to quickly set up a CI server for use on his or her own project and address engineering and management issues that will almost certainly arise. The end result, based on our personal project experience, is a robust system of checks and balances enabling speedy progression toward tape-out of a quality code base.

II. THE NEED FOR CONTINUOUS INTEGRATION

Early in his career, one of the authors was in charge of the weekly integration of changes made by various design and verification commits. Engineers would verify their changes

[1] Changes that require updates from multiple users before they will work could be done on a branch, and then merged atomically to MAIN. Or just let the rest of the team know the build will be broken temporarily while all commits occur.

relative to the most recent, known good tag. Then at the end of the week the integrator would check out the latest source from CVS and attempt to run a regression. Chaos almost always ensued, and the process frequently took 2-3 hours - and occasionally a day or more. And this was for a team of only 15 people!

A couple of issues frequently occurred during the integration:

- The merged build did not compile.
- One or more regression tests failed.

These problems were often caused by:

- Changes from one engineer conflicting with changes from another engineer.
- Lack of validation that the initial check-in had actually passed all relevant regressions.
- Missing files.
- Environmental differences between engineers' shells.

As it turns out, the longer the interval between integrations, the larger the chance of failure for reasons such as those described above. So the more frequent the integration, the better the odds of catching problems at the source, before they cascade into failures that can only be debugged by sorting through the results of many multiple check-ins. If you take this idea to its logical conclusion, it makes the most sense to continually perform integration on any changes, as soon as they happen. A change is made, the code is checked in, and tests are run. If those tests fail, then the person who submitted the failure knows that they've introduced a problem into the project database. Emails are sent, alarms are triggered and everyone is made aware of the issue, as close to when it happens as possible. The two main advantages are then that the issue can be addressed quickly, and also anyone who subsequently checks out the code is aware that they have a potentially broken version of the codebase. This removes the ambiguous situation of trying to establish if changes you made had broken the code, or if the problem already existed.

A. Basic Requirements

To make this work, there are a couple of fundamental mindset shifts that may be required. First, the idea that the HEAD[2] of the build should be a working, viable copy, all of the time, needs to be accepted. This is not the only version control methodology, but it is a required part of making the shift to using continuous integration. The justification is that the benefits of a known, working HEAD outweigh the typical local convenience of checking in broken code. With a known, working HEAD in the design database, an engineer can be certain that when they check out the latest version of the source code that the design compiles and passes all of the tests in the current Jenkins test suite. That base level of confidence enables them to start working with more confidence than if the source code was in an indeterminate state. Even if the HEAD is temporarily broken and someone happens to check out the database, they have a way to establish the health of the version they accessed via the Jenkins server. Certainly it is initially easier for each person to check-in code that may not play nicely with other check-ins, but the overall cost to the project is much higher. So, for CI to work, HEAD has to be a working build.

Second, it must become a primary goal to always keep the HEAD healthy. It should be a priority to fix a broken test that is in the HEAD of the build, as soon as the issue arises. A CI server will allow you to highlight the broken state of the HEAD, but there needs to be a cultural shift to keeping that healthy, in a timely fashion, as the number one goal. It is okay to break the HEAD, but it is not okay for it to stay broken for long. Fostering the attitude of maintaining the health of the build is something that different teams address in different ways and will be discussed more in a later section.

Note that we only need to keep the main development branch (or branches) clean for CI to work. Individuals can (and should) create personal branches where they can check-in unfinished or otherwise broken code. It is only when this code is merged back into the mainline that it must compile and run successfully. In general, use of CI strategies implies a "release early, release often" philosophy for check-ins. Check-ins should occur regularly, and private branches should quickly be merged back into the mainline of the source code repository. According to Martin Fowler, one of the main proponents of CI, developers should commit to the mainline every day.[1]

B. Selecting a CI Server

One of the biggest hurdles to adopting continuous integration can be simply selecting an appropriate CI server. There are many options available. Some freely available examples are:

TABLE I. FREELY AVAILABLE CI SERVERS[2]

Name	Link
Cruise Control	http://cruisecontrol.sourceforge.net
Jenkins	http://jenkins-ci.org
Hudson	http://hudson-ci.org
Integrity	http://integrityapp.com

There are also commercial CI servers:

TABLE II. COMMERCIAL CI SERVERS

Name	Link
Go	http://www.thoughtworks-studios.com/go-agile-release-management
Bamboo	http://www.atlassian.com/software/bamboo/overview

Jenkins and Hudson are essentially the same - with Jenkins arising out of a dispute that occurred after Oracle bought Sun. We started with Hudson but switched to Jenkins after the main developer of the tool, Kohsuke Kawaguchi, along with the

[2] The HEAD of a regression is the most up-to-date, last checked-in, version of every file

majority of developers, created the Jenkins fork and refocused their efforts there.[3]

Why Jenkins? We had previous experience attempting to get Cruise Control working with a block level verification environment and found it difficult to extend beyond its Java and Ant-centric origins. Hudson/Jenkins was suggested as a more flexible alternative. We found it to be simpler to set up than Cruise Control as well as more adaptable to EDA tool management.

III. Making The Case

Hopelessness... despair... frustration... all this after pulling yet another update from our revision control system and realizing that the latest check-ins were broken again. This was happening regularly – several times per week. The time being wasted by the members of our distributed team was delaying completion of key tasks. In the face of this, a decision was made to attempt to get some sort of CI server up and running. If it could be done quickly and without involvement from IT or other engineers, we could at least have our own private way of determining when something had gone wrong. As luck would have it, Jenkins had some key features that allowed our skunkworks project to take shape.

A. An Experiment

First, Jenkins was written in Java, and so only required that a Java virtual machine be installed on the host server. We were able to use the version of Java installed on the server, without any modification. (Java version 1.6)

Second, while it can be used within a more robust web server (e.g., Apache), Jenkins ships with an integrated web server. Critically, we were able to get Jenkins up and running in less than an hour and executing on one of the most problematic block level testbenches - all without the involvement of management, IT, or other engineers on the project. All that was left to do was to sit back and wait for the build to fail.

Each time the build failed, we were able to see this directly on the CI server webpage (basic email support was not initially available from the server we used to run the CI server). Then, instead of spending time debugging why the latest code did not work, we could just inform the responsible engineer and ask them to resubmit a working version of their changes or any files they had forgotten to commit (that was a frequent cause of initial failures, new files that were not added to the repository when newly dependent files were submitted). And while we were at it, we could provide a link to the CI webpage demonstrating the failure based on their check-in. After about a week of this it became clear to additional team members and management (who had, by now, seen results from the CI server page) that the CI server was a beneficial to the team. Effectively, we were able to significantly reduce debug time and catch bad check-ins before they started to deeply impact everyone's development efforts.

B. Additional Targets, Additional Teams

Initially we selected a few critical block level testbenches that we had an active self-interest in maintaining under the watchful eye of the CI server. As our confidence in CI grew, we began to add targets for other block level environments, and even the full chip environment. A key discovery was that some blocks did not have self-checking testbenches. While CI is still beneficial as it can catch compile-time or blatant run-time errors (such as missing files or a tool seg fault), it is much more useful if a self-checking regression is available. Even the addition of a simple register test to a block-level testbench provided significant protection against a non-functional design.[3] This then motivated the team to add at least some basic level of self-checking to as many module-level benches as possible.

Once the server had been running for a few weeks, we discovered that Jenkins could read regression results if they were in the JUnit XML format. We soon added the ability to generate such information from our run scripts, allowing Jenkins to start reporting not just a binary pass/fail, but the percentage of tests from our small check-in regressions that passed, the cause of failures, and historical data going back across all previous check-ins.

IV. Setting Up A CI Server

Ease of use is critical to the success of CI. Engineers should be able to quickly add new regression targets and see results. Failures should be widely distributed so that everyone is up-to-date on the status of each relevant target. In this section we discuss the components required for a successful CI rollout - the server, testbenches and scripts, and feedback devices.

A. The Server

Basic setup and installation of Jenkins is startlingly simple. You need a recent Java Runtime Environment. You download one file. You run a single command.

```
java -jar jenkins.war
```

That is all that is required to get the web interface up and running. You then point a web browser at `http://<server>:8080` and finish the rest of the configuration in the browser. Jenkins can be used with most standard servlet engines that support Servlet 2.4/JSP 2.0, such as Glassfish v2, Tomcat 5. However, none of those are required for a small project team. We were able to support a team of 10 engineers running 38 build targets, using just the built-in Jenkins web server. We did further integration, with a startup script that rotates logfiles and a cron job to make sure the system is alive and healthy, but those additions are not necessary when getting started. Figure 1 below shows an example startup script for the Jenkins server that also manages rotation of the log files each time it is started. Figure 2 lists a typical configuration for the logrotate tool.

[3] How to architect your design blocks to be easily testable at the block level is beyond the scope of this paper.

```
#!/bin/bash
# Start Jenkins and rotate logs.
export JENKINS_BASE_DIR=/proj/work/jenkins-ci
export JENKINS_WAR=$JENKINS_BASE_DIR/jenkins.war
export JENKINS_LOG=$JENKINS_BASE_DIR/jenkins.log
export \
  JENKINS_HOME=$JENKINS_BASE_DIR/jenkins-work

# Set up JAVA path if needed...
export JAVA=java

# Rotate logs, then start Jenkins
/usr/sbin/logrotate –s
$JENKINS_BASE_DIR/logrotate/status
$JENKINS_BASE_DIR/logrotate/logrotate.conf

nohup nice $JAVA -jar $JENKINS_WAR >&
$JENKINS_LOG &
```

Figure 1. Jenkins Startup Script

```
# logrotate parameters. Taken from:
# www.techrepublic.com/article/manage-linux-log-
files-with-logrotate/1052474#

compress
notifempty
/proj/work/jenkins-ci/logrotate/jenkins.log {
    weekly
}
```

Figure 2. Logrotate Configuration (*logrotate.conf*)

Once the server is running, the next task is to create the first build target and hook into your revision control system. In our case, we were using Perforce, a commercial Version Control System, but Jenkins supports many version control systems through plug-ins. These plug-ins are installed and configured through the Jenkins GUI.

To recap:

1. Download the jenkins.war file from www.jenkins-ci.org and optionally follow the installation instructions[4].
2. Create a "free-style software project"[4] for your environment. Ensure integration with your revision control system.
3. Ensure that your simulation run script appropriately returns a non-zero value if it fails, as opposed to just printing a message saying a failure occurred.

B. *Testbench and Scripts*

Installing the server is only the first step. The next thing to do is to start adding targets for units in your design. We recommend picking a small module first and build from there. The next consideration is what subset of tests for that module are appropriate for a useful sanity check. In our experience, the tests or regression need to complete in a short period of time to be most effective. An often-heard rule of thumb is that check-in tests should take approximately 10 minutes to complete. We found that most useful targets ran to completion in less than an hour, with the majority of those being under 10 minutes. Longer tests may provide higher levels of protection, but one of the main advantages of CI is in providing timely feedback to avoid propagation of the bad check-in.

In addition, you can still use Jenkins to run longer tests and nightly or weekly regressions. These are controlled by providing a more complex build trigger or customizing the polling schedule, in addition to the version control sensitive check.

C. *Reporting Test Results*

There are a variety of ways a simulation can complete and indicate success or failure. Jenkins uses the exit status of the final shell to indicate the success/ failure of the overall test. Typically, a return value of 0 (EXIT_SUCCESS in C) is considered a pass and any non-zero value, usually 1 (EXIT_FAILURE in C) indicates an error condition.

For simulations in particular, some care has to be taken when evaluating these exit status codes. A simulator can run correctly, with no errors, in the simulator application and exit with a successful status. However, it could be that it successfully simulated a design which had a failing test. The simulation executed as expected without problems, and correctly simulated the failure. The exit status in this case would indicate to Jenkins that no problem occurred, when really the simulation that we are interested in failed. Some failure cases, such as pointer errors, out of memory errors or other fatal errors, assertions or exit calls from C DPI routines may cause a simulator failure exit condition to occur. Build errors and similar compilation fails will indicate a failure status without modification, but verification environment and test failures can often indicate the failure by a text message to a log file, but still allow for a passing exit status to be returned.

Our solution to this problem is to post-process the simulation logfile and search for known error strings. The overall script that manages the test execution runs the simulation, and then runs a post-processing script on the output of the simulation. It then passes the exit status of the post-processing script out to its own calling environment, which is Jenkins. This somewhat convoluted passing of error statuses is required to catch all potential failures. The post-processing script checks for strings such as OVM_ERROR, but also allows for exclusions, as for example, the string 'OVM_ERROR' appears correctly at the end of most OVM tests, where it may show that there were no OVM_ERROR's seen. The post-processing script has to be intelligent enough to overlook the cases where the words it is scanning for are used for documentation purposes. As a further example., it is often the case that errors might be injected into a test and you may report 'error injected' into the log. The post-processing script must not flag an error due to these sorts of informational messages. We dealt with this by providing regular expression masks for phrases that can be ignored for certain test logs. Another common problem is when a simulation completes but does no useful work (e.g., it stops at time 0), or some other unknown and untrapped error occurs. We avoid this situation by requiring certain strings to be seen in the log file (e.g., a string such as 'SUCCESSFUL END OF TEST REACHED').

[4] More formal and robust installation instructions can be found at https://wiki.jenkins-ci.org/display/JENKINS/Installing+Jenkins

If and only if this string occurs in the test log, is the test considered to be a success.

We further enhanced the reporting of results in Jenkins by using the xUnit plugin, written by Gregory Boissinot[5]. This reads results in the popular java JUnit format and can report details about all of the tests in a regression, rather than just an overall pass/fail from a script as discussed in the previous section.

We implemented this enhancement by extending our post-processing script to generate an appropriately formatted XML file. In our particular case, we used a Perl post-processing script and the XML::Writer library from CPAN to generate correctly structured XML. Similar libraries are available for most scripting tools, or you can just generate the structured XML that is required using general text processing. We would recommend making use of an XML library, to ensure correct formatting and to avoid wasting time debugging XML syntax and structural documentation issues.

Figure 3 demonstrates the key parts of the JUnit format. The XML header indicates the document type, and then the various test suites are included within an enclosing `testsuites` tag that provides the overall name for the regression (*results)* as an attribute. In our results we include the build and test outcomes as separate test suites, so it is easy to recognize if a build failed or that individual tests failed, while the overall build was healthy. The first build `testcase` contains the results of the build step. The second `testsuite`, with attribute `test_regression`, documents the result of each individual test. Each test case can have an optional `<system-out>` element, which contains the log messages from the test execution. These are then viewable within Jenkins, allowing users to explore test results within the GUI. For tests that fail, a sub-element, `<failure>`, indicates the problem and can provide further information about the type of failure. For passing tests, nothing is required to indicate correct completion other than the lack of a `<failure>` element. As you can see, the XML is fairly straightforward and easy to generate from a script that traverses your test results within the regression run directories.

This additional level of integration with Jenkins is very powerful, providing a history over time of particular test cases and the pass rates for a given set of tests. These are viewable in the GUI and can be graphed and interrogated to understand the long-term health of the design. One caveat to this we found is that particularly long log files can cause problems for the xUnit parser. We worked around that issue by truncating the log files to just the head and tail sections. These sections provided information required to re-execute the simulation (command line, switches, randomization seed, etc.) and associated error and failure messages. The truncated log entry reduced the overhead of providing all of the log messages in Jenkins, particularly if a test was especially verbose. We also implemented means to suppress log messages for correctly passing tests, which again reduced the size of the reporting files.

```
<?xml version="1.0" encoding="UTF-8"?>

<testsuites name="results">
  <testsuite name="build" time="376"
tests="2" failures="1" passes="1">
    <testcase time="" name="sim build">
       <system-out>Log message
here</system-out>
    </testcase>
  </testsuite>
  <testsuite name=”test_regression"
time="376" tests="2" failures="1"
passes="1">
    <testcase time="166" name="test_0">
    <system-out>OVM Log message</system-out>
      <failure type="Fail"
message=""></failure>
    </testcase>
    <testcase time="210" name="test_1">
      <system-out>OVM Log message for test
one.
Log message elements can be the entire log
on multiple lines
      </system-out>
    </testcase>
  </testsuite>
</testsuites>
```

Figure 3. JUnit XML report format

D. *Tuning Parameters*

Build targets in Jenkins have a few options that can be used to tune their behavior and performance. One of the most important options is to determine how and when new build are triggered. The two most likely choices are:

- Poll SCM
- Build periodically

Polling periods can be set in a fashion similar to a cron job. To poll the repository every minute for changes, use:

```
* * * * *
```

To poll at 5 minutes past the hour, you use:

```
5 * * * *
```

When a polling window is reached, if one or more checkins are detected Jenkins will initiate a new build of the target in question. To prevent backups in Jenkins given limited compute resources, we usually set the polling period relative to the length of time it took to complete the regression. So targets that took 10 minutes or less to complete might be polled every minute. Targets that took an hour might only get polled once per hour or two. And special, long-running targets such as synthesis runs might only be exercised once or twice per week.

To further reduce the load on licenses and compute resources, Jenkins can implement a quiet period after it has triggered, so that check-ins that occur close to each other within a time window will not trigger multiple regressions, but will just run the final checkin. This does reduce somewhat the visibility into which specific checkin caused an error, but assuming the system is tuned to a small window of a few minutes, can avoid triggering failures due to common mistakes

such as forgetting to add a new file to a commit then realizing very quickly afterwards and checking it in a few minutes later.

You can also configure Jenkins to build targets *regardless* of whether or not there have been changes detected in the build. Though it is best practice to keep track of all items required to build your design and testbench code, the reality is that some things such as EDA tools and sometimes scripts are not versioned in the same way as your design and verification source. Thus Jenkins would not be able to tell, for example, that the installed version of the simulator had changed such that your code would no longer compile. Building periodically allows for a safety check to ensure your environment has not changed in a way that would cause an otherwise stable code base to start failing. This periodic execution, without source code changes, can also be used to run nightly and weekly random regressions that vary the seed and explore more parts of a design.

One other important parameter is the number of outstanding parallel jobs Jenkins can execute at a time. Based on the number of servers and licenses available to us, we did not allow Jenkins to run more than four targets at a time. Note that each target may still have used multiple compute nodes. So our Oracle GridEngine batch processing software also came into play when determining how our compute resources were allocated.

E. Feedback

One of the fundamental ideas of CI is that if the build is broken, everyone should know and do something about it. The default Jenkins server will send emails to a notification list, or the users that broke the build. However, something more visible can prove to be very effective in keeping your build healthy.

Jenkins has several customizable views that can indicate project health for a kiosk style display. On our project, we set up a very simple web server that displayed the build health in a public area in our office. It was easy to see from the display if any particular subsets of the design were in a broken state or had failing tests. This feedback display happened to be in the area where we had a brief daily status meeting so that any issues could be discussed and then resolved quickly after the meeting. Making sure that the build status was front and center in everyone's mind helped ensure that failures didn't stick around for long. Placing the build status next to the coffee machine can be a good place too! Jenkins can be configured to display the username of the engineer who broke the build, so that it is clear who caused the problem. There is also a 'game' built into Jenkins that keeps a running score of who has caused or fixed the most problems. Having that scoreboard visible can also be a useful mechanism to promote build health.

An eXtreme Feedback Device (XFD) can be used to distill all the available information from a Jenkins CI server down to one single bit of healthy/non-healthy status and then provide a quick, noticeable way for everyone on the project to know that the build isn't healthy. It can become quite competitive to not be the person to break the build. This single bit status can then be displayed in a variety of different ways. Some XFD suggestions include: klaxons, flashing sirens, real, full size traffic light displays. Lights and sounds in general are good options.

Jenkins provides a flexible query API for programmatic control of the CI server. There are JSON, XML and Python interfaces to the server[6]. One of the authors was able to quickly write a simple Python script to interrogate the server to ascertain the overall status of all builds and all tests. This information was then communicated over a serial link to an Arduino development system[7]. From there it is trivial with a few components to control LEDs or relays to switch larger, more visible feedback sources.

We found that keeping the feedback visible and fostering a competitive attitude to not breaking the build helped us to introduce CI to the project team. At first we could jokingly chide people who had a broken test or check-in. After a short period of time people started seeing the value of keeping the overall projects healthy and would tend to fix things before having to be asked. The publicly visible nature of the feedback helped to reinforce this behavior change across the team.

V. CONCLUSION

The main contribution of running a Jenkins server is that it provides accountability for breaking the codebase. Engineers are held responsible for wasting the time of other engineers on the project, when they check in broken code. You are left with a high degree of visibility into the health of the design as it is being developed. If your regression fails, you can more easily establish if your changes are causing problems, or refer back to the CI server to check if the code was already in a non-functional state.

Introducing the Jenkins CI server on our verification effort saved significant time that had previously been spent each week debugging why the mainline of our source code repository was non-functional. It also improved code quality by encouraging engineers to create self-checking testbenches. In the end, there were over 38 testbench targets configured for a team of 10-15 engineers. This enabled a fine-grained coverage of the status of each portion of the design and testbench. Some portions of synthesis results were also checked, and some random regressions were also controlled via Jenkins.

VI. REFERENCES

[1] M. Fowler, "Continuous Integration," *martinfowler.com*, 01-May-2006. [Online]. Available: http://martinfowler.com/articles/continuousIntegration.html. [Accessed: 30-Jan.-2012].

[2] "Comparison of Continuous Integration Software - Wikipedia, the free encyclopedia," *en.wikipedia.org*. [Online]. Available: http://en.wikipedia.org/wiki/Comparison_of_continuous_integration_software. [Accessed: 31-Jan.-2012].

[3] J. F. Smart, *Jenkins: The Definitive Guide*. O'Reilly Media, Inc., 2011, p. 406.

[4] "Building a software project - Jenkins - Jenkins Wiki," *wiki.jenkins-ci.org*. [Online]. Available: https://wiki.jenkins-ci.org/display/JENKINS/Building+a+software+project. [Accessed: 31-Jan.-2012].

[5] "xUnit Plugin - Jenkins - Jenkins Wiki," *wiki.jenkins-ci.org*. [Online]. Available: https://wiki.jenkins-ci.org/display/JENKINS/xUnit+Plugin. [Accessed: 31-Jan.-2012].

[6] "Remote access API - Jenkins - Jenkins Wiki," *wiki.jenkins-ci.org*.

[Online]. Available: https://wiki.jenkins-ci.org/display/JENKINS/Remote+access+API. [Accessed: 31-Jan.-2012].

[7] "Arduino - HomePage," *arduino.cc*. [Online]. Available: http://www.arduino.cc/. [Accessed: 03-Feb.-2012].