A Guide to Using Continuous Integration Within the Verification Environment

Jason Sprott, Verilab Ltd., UK
André Winkelmann, Verilab Ltd., UK

Gordon McGregor, Nitero Inc., USA

March 5th 2014

***Abstract*—In 2012, we introduced the EDA world to the power of Continuous Integration (CI) with "A 30 Minute Project Makeover Using Continuous Integration"[1]. Now, two years later, there is still some confusion between the roles of Continuous Integration and EDA regression tools, when it comes to verification management and test execution. Also, some of the challenges faced when integrating CI with EDA tools, touched on in the previous paper, need to be expanded upon using concrete examples. While the technology for CI has been available for years, the methodology and mechanics of how best to use these tools in an EDA environment is not always obvious. In this paper we will clarify the roles of CI and regression management, and provide a practical guide on how users can quickly deploy CI within their verification environment**

***Keywords*—Software testing; Computer simulation; Digital Simulation; Logic design**

I. Introduction

A CI server can be installed and managed with little to no interaction with the IT department. The initial setup can often be done in as little as 30 minutes. In our examples we use Jenkins, which is a freely available open source tool, used on many large-scale software projects. Jenkins works with the most commonly used version control systems and can easily be integrated with existing regression scripts. The original paper [1] describes how to install and configure a basic set up, so this will not be covered again. Instead we will focus on the following:

- How to control simulation and regression tools from Jenkins
- How to get pass/fail results from simulations into Jenkins
- How to make simulation test results and reports available from Jenkins

A quick reminder of where Jenkins fits into a typical EDA development environment, using source control management (SCM), e.g. Subversion, Git, or ClearCase, is shown in Figure 1. The value of CI is built upon the idea of releasing and integrating code early and often. The main development branch is kept healthy, by Jenkins monitoring changes that developers check in, automatically running tests, and keeping the team continually informed of the current state. This paper focuses on the integration between Jenkins and the EDA environment, required in steps 3, 4 and (to some extent) 5, in the diagram.

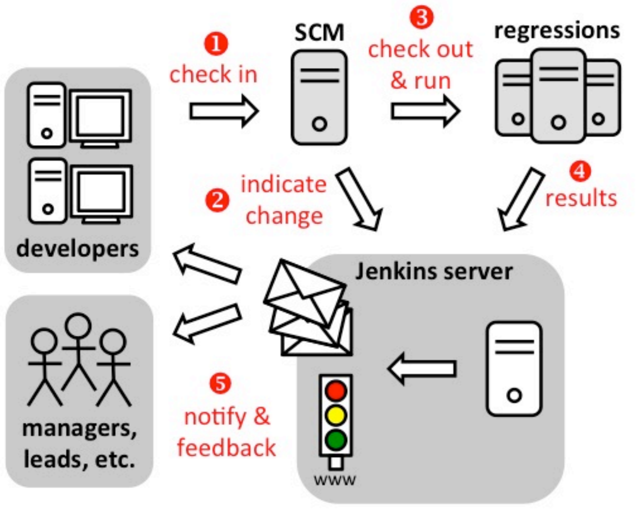

Figure 1: Jenkins Environment Overview

A typical so-called Metric Driven Verification (MDV) environment provides a platform for launching, controlling and triaging many simulations. The task of analyzing the results from such an environment can be daunting enough from a verification (finding bugs in the design) point of view, but as with any software project, there is an additional challenge of keeping the build healthy. Today's EDA tools have built-in features designed to optimize effectiveness on the verification side of things, and are typically tightly integrated with verification language features, methodologies used, and other complementary tools in the chain. The EDA tools aim to make sense of, and present the results from, the many metrics collected during simulation. This is crucial for the detailed debug and analysis required for verification closure.

However, we also need to ensure issues with the checked-in code are detected quickly, effectively and preferably automatically. This is typically done by running some of the same tests used for verification, in the same MDV environment, but with a more coarse-grained view of status. This is where CI tools such as Jenkins come into play. In the following sections we will look at where a line can be drawn between the features best left to the EDA tools and the additional benefits that Jenkins provides.

The paper will show two examples of a Jenkins integration. Two versions of a commonly used EDA tool (Cadence vManager), are used to illustrate how a significant tool API change affects the Jenkins integration code. One version of vManager makes use of the e-language API, while the other (vManager C/S) uses a new TCL-based API.

For a quick start, all necessary scripts and code examples have been put in a publicly available repository [7]. Instructions on how to set up the environment and use the code are provided in a README file. The solutions presented are based on knowledge gained from several years of CI use on live verification projects. The concepts and much of the code provided are generic and applicable to any MDV tool/environment.

II. Comparing CI and Regression Tools

The installation of a CI tool may be easy, but it's often surprising to learn that not everyone understands the difference between what a tool such as Jenkins does for the project, as compared to a typical EDA tool, handling verification planning, management and regression control. In fact both tools automate the submission of simulations and provide analysis of results, so you can see how there might be some initial confusion. Sometimes there's enough confusion to cause CI to be dismissed out of hand, as not being capable of adding enough value to a project. From a project management point of view, taking lessons from the many complex software design projects using CI, this is almost certainly a big mistake.

Cadence vManager, is a modern verification planning and management tool, focusing on MDV. The tool offers a lot of built-in automation capability, and sophisticated features for results management and analysis. This is useful in order to illustrate the split of responsibilities, between the CI and MDV environments.

The MDV environment provides the user with the ability to manage simulations (sequential and in parallel), as part of a regression, and the capability to efficiently analyze the complex stream of results. That's no small feat, as a typical verification environment these days must filter, analyze and correlate a huge amount of output, e.g. log file errors/warnings, assertions, code coverage, functional coverage, plan coverage. The goals of a MDV environment are to simplify the overall debugging effort by using sophisticated triage features, and manage the overall complexity of the data presented, in order to achieve verification closure.

Jenkins' job on the other hand is to automatically monitor the on-going state of checked-in code, and make the results of this easily available to all interested parties. This may be interesting in a wider context than just a single project. For example, an IP library might be shared across multiple projects, so a change that breaks something can have significant implications. Today's projects are full of such dependencies.

There is an overlap with the MDV environment as, in order to do this, simulations (unit tests) have to be run by Jenkins and the results analyzed. These unit tests are normally run in the usual way, using the MDV environment, but the CI and MDV tools are working on slightly different problems. The thing that a CI tool cares most about is the PASS/FAIL status of these unit tests. The tests can be a carefully selected subset of an overall regression; however, it's very important that the tests are able to pick up issues associated with the functionality of the checked-in code. In other words the tests say something about the state of the code that has just been added, or modified. This is a significant distinction between MDV and CI goals. It's possible to select tests that are meaningful in a verification context, i.e. provide functional coverage of the design under test, but are not sensitive enough to pick up errors, due to changes in the source code that triggered the test.

The type of triage done at the CI level is aimed at increasing project efficiency overall, by identifying problems *introduced* as early as possible. This means that failures (build or test), are tracked and presented at regular intervals, and fixed quickly. Although it's possible to pass all sorts of information from the MDV environment to the CI tool (e.g. verification plan or functional coverage figures), the most basic and most useful requirements are the per-test PASS/FAIL results, and links to detailed results (if available) in the MDV environment.

The quickest way to integrate the CI and MDV environments is to allow them to play to their own strengths. Out of the box, Jenkins tightly integrates with SCM tools, tracks per test PASS/FAIL trends, provides an easy mechanism to publish results to web pages and notify interested parties by email.

Out of the box, vManager provides regression management and features to analyze results; but most importantly for CI integration, the following two features are available:

- Customizable log file filtering (for PASS/FAIL status)
- A mechanism to link to the build and test results

Not all tools provide this filtering out of the box, but ultimately the environment somehow has to implement a common way to do it.. Leaving this filtering to the MDV, as opposed to providing a separate mechanism for the CI tool, is very important. It not only allows the MDV environment to do the heavy lifting on the log file pass/fail parsing, it also provides a consistency between CI and MDV environments, and keeps the amount of data passed to Jenkins to a minimum. Detailed analysis of the results is best handled in the MDV, but links to the results can easily be provided on the Jenkins dashboard.

III. The Solution

A. Overview

As previously mentioned, the solution presented in this paper is not specific to vManager. The concepts themselves apply generally. The implementation has been split into two distinct parts:

- Generic: code and techniques independent of the MDV environment. A standard Comma Separated Values (CSV) data interchange format and standard python libraries are used, to provide a reliable and easy to understand integration to Jenkins. One MDV tool could be swapped out for another tool without the need to modify the generic implementation.

- Specific: code and techniques tailored to the specific EDA tool being used (vManager and vManager C/S). The issues dealt with apply to most tools, but implementation is dealt with in a tool specific way.

Any solution will have a tool specific component. The requirements for this will depend on the capabilities of the tools used in the MDV environment. For example, not all tools have the ability to handle exiting at the right time (with respect to job control), with meaningful exit codes, or can export results as HTML and CSV. Their APIs may be in different languages and have different limitations.

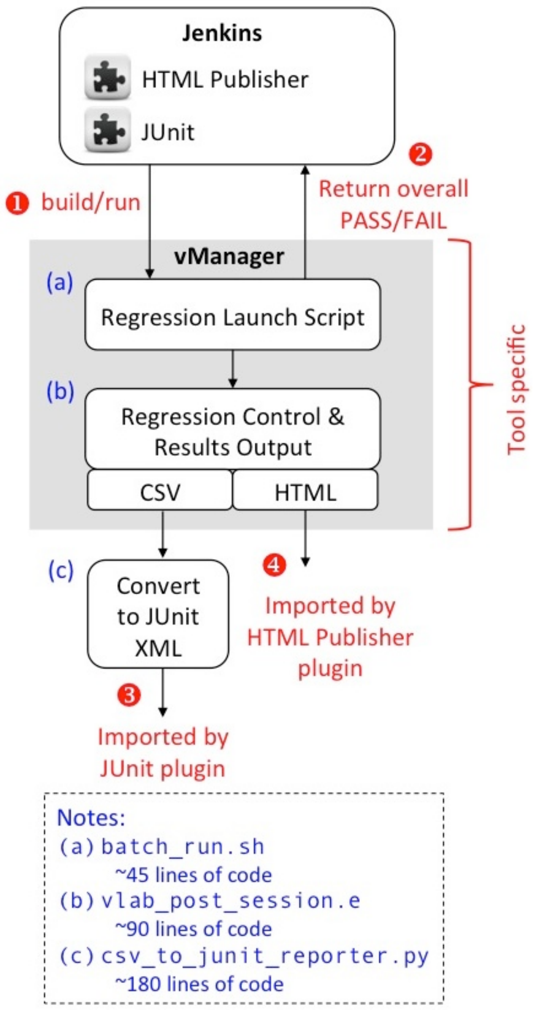

Figure 2: Overview of Jenkins/vManager Integration

Figure 2 provides an overview of the vManager solution. The tool-specific part of the solution is encapsulated in the grey vManager box. This is described in more detail in Section B, but it's useful to note the following:

- `batch_run.sh`: is the build/run regression launch script called by Jenkins, which starts the regression. This script runs all the steps necessary to create the input for Jenkins and generate the *overall* PASS/FAIL status of the run (return code from the script itself).

- `vm_launch.pl`: is the vManager launch script, called by `batch_run.sh`, which starts the vManager regression with the appropriate arguments. This is a script shipped by the vendor.

- `vlab_post_session.e`: is a simple script developed by the authors, to wait on the end of all simulations in the vManager regression run, and generate a CSV file with the aggregated per-test information the JUnit plugin needs.

- The fields required in the CSV file are defined by JUnit's reporting capabilities. This is a generic requirement.

- `csv_to_junit_reporter.py`: is a Python script developed by the authors, which builds the JUnit XML used by the Jenkins JUnit plugin, taking the vManager generated CSV file as input.

- vManager generated HTML reports are published as links in the Jenkins dashboard, using the Jenkins HTML Publisher plugin.

The generic part of the solution (including `csv_to_junit_reporter.py`), can be used as-is without modification. The details are covered in section C, but only to explain the construction and requirements considered.

B. Specific - vManager Integration

This section aims to describe the tool specific implementation, required to integrate Cadence vManager with Jenkins. The first section describes how to control a vManager session in a Jenkins-friendly way. We will also look in more detail at how to determine overall, and individual, test pass/fail statuses. Finally we show how to embed HTML reports from the vManager environment into the Jenkins dashboard.

No matter which regression tool is used, there are some common requirements in order to ensure the Jenkins integration solution is robust and provides consistent results. The following should be considered:

- If the regression environment has built-in capability to determine the pass/fail status of a test, it should be used. This avoids inconsistent log file parsing results, between built-in and any Jenkins rules.

- Pass/fail results in Jenkins must absolutely match pass/fail results of the regression. The first point above hopefully ensures this. It is a crucial requirement, as issues here can destroy trust in the results presented by Jenkins.

- If the initial build/compile fails, the entire regression in Jenkins has to be marked as failed. Nothing can be guaranteed if this happens. As well as running the risk of picking up bogus code/executables, building cleanly is essential good housekeeping.

- If (for whatever reason), the generated CSV file, or the JUnit XML is missing, the regression has to be marked as failed.

Where possible, it's a good idea to avoid repeating functionality, or work done, by the regression tool itself. In vManager it is possible to utilize the user "batch API" to analyze individual runs of a regression and output the results in CSV format. This can then be processed by `csv_to_junit_reporter.py`. Since the input to this generic part of the solution handles a standard (CSV) format, it's perfectly acceptable to have the regression tool generate this for us.

The details of the vManager specific implementation can be broken down as follows:

1) Launching the vManager regression

Jenkins will consider a non-zero return on any of the commands executed in the launch script as an overall failure, so it's important to program with that in mind. Each command in the toplevel script has been designed to return zero on success, and non-zero when it fails. This ensures that if any step in the flow fails, Jenkins will recognize it.

To give some context to the following discussion, a simplified view of the steps in the `batch_run.sh` script called by Jenkins is shown in the following listing:

```
# launch vmanager in batch mode
vm_launch.pl ...
# Check if overall regression passed
grep PASS regression_result.txt
# generate junit XML from CSV output
python csv_to_junit_reporter.py <CSV> junit.xml
#create links to HTML reports for Jenkins
...
ln -s <PATH-TO-REGRESSION-RESULTS> <LOCAL-PATH>
```

Two things are important in order to run a command as part of a Jenkins build:

- The command needs to run solely in batch mode
- The command should not return before all its threads (i.e. running simulations and any processing), have completed.

The authors decided to use the vendor supplied `vm_launch.pl` utility, which simplifies starting a vManager batch session, and satisfies the two aforementioned points. Figure 3 details options used for the `vm_launch.pl` script. The same options can also be used on vManager directly.

vm_launch.pl / vManager Options
`-vsif <path/to/my.vsif>`
`-batch`
`-command "load vlab_post_session.e; sys.vlab_analyse_session()"`
`-output_mode log_only`

Figure 3: vm_launch and vManager options

The `batch` and `log_only` options, make the session described in the vManager session input control file `(vsif)`, start automatically in batch mode with no windows. To satisfy the requirement of only returning once all simulations have completed, the `vlab_post_session.e` script contains the following call:

```
vm_util.trace_all_sessions_to_finish();
```

Once this call has returned, the results of the session are analyzed.

2) e-language user API

vManager has an e-language user "batch API", which provides the functionality needed to output regression results in a format we can post-process easily. In the new version of vManager C/S, the e-language API has been replaced by a TCL API (see section D). The same CSV file can be generated through a TCL function (via `csv_export` available from version 13.20 onwards). The following listing shows the e-language API in use:

```
vsof: vm_vsof =  vm_manager.get_all_sessions()[0];
vsof_result: bool =
  vsof.get_failures().is_empty();
runs: list of vm_run = vsof.get_runs();
attr: string = runs[0].get_attribute_value(
  vm_manager.get_attribute_by_name("test_name"));
run_result: bool = runs[0].has_passed();
```

The `vm_manager` object is the entry point into the batch API, providing access to all loaded, or completed sessions, via the `get_all_sessions()` method. This returns a list of sessions, represented by `vm_vsof` objects. It is possible to load multiple sessions into vManager. In order to access the tests executed in a session, a list of `vm_run` objects can be obtained using the `get_runs()` method of `vm_vsof`. More information can be found in the documentation [8].

We can find out if there are any failures related to the session runner itself using the `get_failures()` method, e.g. a vManager "dispatching failure". This does *not* include failures of individual tests.

The `get_attribute_value()` method lets you access any attribute you can find in a `vsof` file. In order to retrieve the same pass/fail status as shown in vManager itself, the `has_passed()` method of a `vm_run` object can be used.

3) Determine overall regression pass fail result

The authors decided to output a single file (`regression_result.txt`), with just the word “PASS” or “FAIL” in it, to signify the overall regression result to Jenkins. This allows a simple grep to return a non-zero exit code from the build/run script executed by Jenkins, if the simulation has failed. If this happens, it will appear as a failed regression run in Jenkins. This technique also provides a safety net in case any of the scripting fails, e.g. if the `regression_result.txt` is not generated, the grep will fail.

To find overall status, the `vlab_post_session.e` script uses the `get_failures()` method of the `vm_vsof` object, first to determine if the overall regression runner had any issues. Secondly, if this method returns an empty list, the script continues to go through each `vm_run` and evaluate the `has_passed()` method. If all runs pass and the session does not include any failures, we write a “PASS” to `regression_result.txt`.Otherwise we write “FAIL”, denoting the regression has failed for some reason.

4) Generating CSV output for per test results

The `vlab_post_session.e` script generates a CSV file which provides individual test results. This file is the direct input to `csv_to_junit_reporter.py,` which translates the CSV into JUnit XML report. The e-language script iterates over the list of `vm_run` objects and extracts the data to be written to the CSV file, using the `get_attribute_value()` method. The following data (by CSV column names) is extracted. The CSV column names are the same as the CSV file outputted by vManager C/S:

- `Test Name`: The name can be directly extracted from the `vsof` attribute "test_name".
- `Test Group`: The JUnit testsuite name is extracted from `vsof` attribute "test_group" e.g. "my_session/my_group”.
- `Seed`: This column uses the `vsof` attribute "seed" which is used by the Cadence Specman tool.
- `SV Seed`: This column uses the `vsof` attribute "sv_seed" which is used by SystemVerilog testbenches.
- `Status`: This column uses the `vsof` attribute "status" and represents the pass/fail status of the test.
- `CPU Time (ms.)`: Recorded CPU time to run the test. The `vsof` attribute "cpu_time" is used for that.
- `Log File`: This is the full path to the logfile of the test. The `vsof` attribute log_file contains a list of log files. For this reason the script picks one log file. It either chooses the first "irun.log" it can find, or if this does not exist, picks the first alternative in the list.
- `First Failure Description`: The `vsof` includes an attribute called "first_failure" which includes the text of the first error message leading to the failure of the test. If no failure is present, this column will stay empty. The formatting of this column is crucial, since the "first_failure" attribute could include new lines, or commas, which could break the CSV format for the next processing step. To handle this, the string is simply double quoted in this column. The python CSV parser works properly afterwards even for multiline strings.

5) Linking to HTML reports in Jenkins

The Jenkins HTML Publisher plugin is used to handle publishing HTML reports, produced by the tools, on the Jenkins dashboard pages. The plugin can take any HTML page, or directory of files, and link that into the Jenkins project page. There is an option in Jenkins to actually store these pages on disk for historical reference if desired.

In vManager we can generate a regression report showing overall session statistics, including detailed information about any collected coverage, and correlation of this data to a verification plan. This level of information is very useful to have access to, but does not need to be displayed natively on the Jenkins dashboard. In our experience this is an area the MDV tools, specifically in this case vManager, handle very well themselves. Creating the links to these reports is low effort, and is the best way to make the information available from Jenkins.

In order to publish the vManager reports in Jenkins, two things have to be set up. Firstly, in the vManager `vsif` file, the following attributes must be setup:

```
post_session_script  : "vm_analyze.pl";
auto_regr_report     : ON;
auto_report_vplan    : "path/to/my_module.vplan";
```

Secondly, as part of the Jenkins project configuration, add a post build action "Publish HTML reports" specifying the directory which includes all the HTML files needed for the report. This directory must include an index page, which is also specified in the setup.

vManager happens to generate reports in a directory, with an index page, a subfolder with the HTML files we’re interested in, plus some other stuff we aren’t. Some of these files can be quite large, so the authors decided to restructure things a bit, and only link to the HTML reports we’re actually interested in from Jenkins. This was done by copying the index page to a new directory and creating symbolic links.

If the Jenkins plugin is configured to store HTML reports for historical reference, available disk space should be considered. Some of files can be very large.

C. Generic– Jenkins JUnit Integration

The `csv_to_junit_reporter.py` script covered here can be used without modification. The details are only covered to allow a better understanding of the implementation.

Jenkins is useful out of the box, using only the overall PASS/FAIL status of the whole regression. However, functionality can be extended, using support for JUnit reports, to provide history for *individual tests* within a regression. This includes the following:

- PASS/FAIL status over time
- Runtime of a test (in addition to the whole regression)
- Specific error messages
- Log files from each simulation run

The JUnit reporting can be especially useful for quickly triaging regression failures, watching the behavior of tests over time, and assessing the cause of a test failures, before detailed debugging starts. This data is input to Jenkins from a JUnit format XML report, generated after the regression is complete, by running the `csv_to_junit_reporter.py` script.

While working on projects, the authors noticed on that the features available to tie the results of simulations back into Jenkins, are not very well understood. There is built-in support for handling JUnit reports, but the XML schema itself, and in particular how it is then used within Jenkins, isn't well documented. There have also been many fragile, and often half-hearted, attempts at writing scripts that partially wrap up results into something close to the correct format. These tend to fall over quite often, due to hitting XML corner cases, or input formatting issues.

In this paper we try to address reliability issues, by using a clean encapsulation of a JUnit XML generator, in `csv_to_junit_reporter.py`, to produce test results for Jenkins consumption. It's built on top of the Python standard library XML implementations. The output is well formatted, and potentially illegal characters are managed without additional user code. XML elements and attributes are properly included and well formed. This encapsulation simplifies the creation of results that meet the JUnit schema.

However, Jenkins only makes use of a small part of the full JUnit specification. As a result, it becomes important to understand which particular elements and attributes are used in Jenkins and how they will be presented on the Jenkins dashboard. This knowledge can be used to structure the JUnit results to include useful data that can help track regression performance and test history.

The following is a list of CSV column names and the corresponding specific structures in JUnit that are useful from a Jenkins perspective:

- `Test Name`: `<testcase> name` attribute. This is the name of an individual test, which can be within a group hierarchy defined by the `classname` attribute.
- `Test Group`: `<testcase> classname` attribute. This is the name of the hierarchical group that contains one or more testcases. The `classname` describes the hierarchy using dots to separate each layer, e.g. "smoke_tests.dma_controller". Since the CSV column `Test Group` contains slashes as separators, these are replaced with dots by the Python script.
- `Seed`: Seed value for the testcase. In case of a failure, it is printed as part of the `First Failure Description`.
- `SV Seed`: SystemVerilog seed value for the testcase. In case of a failure, it is printed as part of the `First Failure Description`. Only valid when SystemVerilog is used.
- `Status`: This represents the pass/fail status of the test.
- `CPU Time (ms.)`: `<testcase> time` attribute. Time taken for the particular testcase to run.
- `Log File`: Full path to the logfile of the test. In case of a failure it is printed as part of the `First Failure Description`.
- `First Failure Description`: `<failure>` element text. The first error message which lead to the failure of the testcase.

Using the script is straightforward. An instance of the report is created as follows:

```
report = JunitReport()
```

A JUnit report for Jenkins has one toplevel `<testsuites>` element, which is called 'results' in this implementation. This element then contains one or more `<testsuite>` elements, which are containers for a variety of `<testcase>` result elements.

```
<testsuites>
     <testsuite>
             <testcase>
             <testcase>
             ...
             <testcase>
     </testsuite>
     <testsuite>
     ...
     </testsuite>
</testsuuites>
```

A `<testsuite>` element can have several attributes:

- `name`: The name of the testsuite
- `id:` An index number for the testsuite
- `time`: CPU time (seconds) taken to execute the testsuite
- `failures`: The number of failing tests within the testsuite
- `passes`: The number of passing tests within the testsuite

At creation time of the testsuite, depending on how the regression results are being accessed, you may not know the values of some or all of these attributes. The `JunitReport()` object allows you to create a testsuite element in advance, which can be returned to later, to update any of the attributes as needed. This can be useful when iterating over a set of results and don't yet know how many passed or failed. It's possible to create the `<testsuite>` element, add all the associated `<testcase>` elements, tracking the pass and fail counts along the way, then update the `<testsuite>` element appropriately.

```
testsuite = report.add_testsuite(
   name="regression",
   id=1, time=200,
   failures=0, passes=0)
```

The object is stateful and remembers the currently active testsuite or testcase, defaulting to adding any new information to the previously created testsuite/testcase. If results span multiple testsuites, the testsuite reference returned from the `add_testsuite()` call can be used to selectively add results to different containers. Otherwise, the `add_testcase()` method will add the result to the last testsuite that was created, which is typical the required behavior.

Once the testsuite is created, it is then a matter of iterating over each testcase within that testsuite and adding the appropriate information. A `<testcase>` element can take several attributes, shown below:

- `name`: The name of the testcase
- `classname`: A grouping hierarchy for the test
- `time`: Time taken for the testcase to run (in seconds)

```
report.add_testcase(
   name="first_test",
   classname="sample.class",
   time=100)
```

It is maybe a surprise that there is no pass/fail indication in the attributes. Instead this information is captured by sub-elements of the `<testcase>.` A `<testcase>` actually supports several sub-elements that Jenkins can use. These are encapsulated in the `JunitReport` object, using the `add_failure()` and `add_log()` methods. The two sub-elements supported here are `<failure>` and `<system-out>`. The `<failure>` element has an associated type attribute (e.g. Fail) and will also take a message attribute (first_error), that Jenkins will display separately from any log added. This is useful as the first_error string is displayed at the top of the particular test results page, making for quick triage of failures, instead of reading an entire log file.

```
report.add_failure(
   type="Fail",
   message="UVM_ERROR: The data is invalid.")
```

For passing testcases, no additional information is added after the `add_testcase()` method has been executed. Typically only logs for failing testcases are added to save space. The `add_log()` method automatically truncates the log by capturing only the first and last 500 lines of the log, keeping just the top and tail of the logfile. This helps to avoid occasional problems where Jenkins and the Java VM run out of memory when processing very large logfiles. Every now and again huge log files are generated, e.g. when debug features are turned on with full verbosity. This is often by accident, since such large log files can cause all sorts of problems. By using the top and tail method, we avoid a small mistake causing the entire regression system to fail, which could ruin the results of many other runs. The size of the truncation can be varied by changing the `TRUNCATE_LINES` value in the script.

After adding all of the testcases associated with a particular testsuite, the number of passes and fails can be updated, by calling:

```
report.update_testsuite(
   failures=number_of_failures,
   passes=number_of_passes)
```

Again, the default is to update the last created testsuite. Alternatively a different testsuite handle can be passed to the `update_testsuite()` method.

D. Example 2: vManager C/S TCL API

vManager C/S is the successor of vManager, adding a whole new client/server architecture and new functionality. However, the most relevant difference between the versions for Jenkins integration, is the change to the tool's API. The e-language based API has been replaced with a TCL based API. This is the API that provides us with access to the information Jenkins needs, e.g. simulation results and HTML report generation. This example was provided to illustrate the effect a tool, or API, change can have on the implementation.

As well as the API implementation language, some of the API functionality has been changed. In the previous version, the authors needed to program their own CSV export in the `vlab_post_session.e` script. vManager C/S provides a TCL function `csv_export()` which does that. However, in order to produce the same CSV file, with the same columns, one needs to configure a so called "view". A view stores a set of column names which can be used in GUI reports, and are also used by the TCL `csv_export()` command. It is stored on the vManager C/S server, so that all users will have access to it. Unfortunately, at the time of writing, it was not possible to automatically create a view through the TCL API. That step has to be performed manually using the vManager C/S GUI. This only needs to be done once. For the example provided, the authors created a view called "vlabCSV". The scripts provided work out of the box using that view name, and will export the correct CSV format. The README file provides detailed instructions on how to create a view.

There is one difference worth noting with regard to the CSV output. The `Test Group` column originally stored a value that contained the session name, e.g. "my_session/my_group/inner_group". vManager C/S does not store the session name, e.g. "/my_group/inner_group" in this column. The authors don't believe this impacts the usability of the presented solution, based on the expected use-model with vManager C/S.

Once the CSV file has been created the remaining steps are exactly the same as before, using the `csv_to_junit_reporter.py` script unmodified.

vManager C/S handles the generation of HTML reports slightly differently, with more options and reports to select from, but the VSIF attributes to generate HTML reports have been removed. The tool now provides a set of TCL functions for HTML generation, e.g. the TCL function "report_vplan" creates an HTML report containing detailed coverage information annotated to the vPlan. It's possible to generate several different HTML reports (adding to `vmanagercs/scripts/run.tcl`), and publish each of them on the Jenkins dashboard.

The HTML reports generated by vManager C/S are Jenkins-friendly, in the sense that all files including the index page are stored in one directory. That way the Jenkins HTML publisher plugin can be configured to use these reports, without the need for any linking or copying into a different directory structure, as was required with the previous version.

IV. Continuous Integration Methodology

Even with a perfectly working Jenkins integration, the authors still find one of the biggest challenges to making CI work for the project, comes down to people. A sometimes contentious, but key point, in CI is the idea of a working main branch that everyone commits to. In other words, HEAD is live and is kept working. Releases aren't hidden away and integrated once a week, once a month, or once in a blue moon. Everyone works on the latest version of everything, as much as is possible.

Changes as a result are smaller. Branches can exist and are used for shared development, but should exist for as short a time as possible. Changes are propagated to the rest of the project as early as possible. This means the integration effort is low and divergence is minimized. Architectural disagreements are brought to light quickly. Cultural team behavior has to develop to ensure people care about the health of the build and maintain it as a top priority.

Also, often verification and RTL changes need to be combined to allow a working check-in. Such changes are typically done by more than one person. As a result it is useful for the verification engineer and designer to be able to share code and ensure it builds together, first, before checking in to the repository. While this can be done with branches in any useful version control tool, it may be interesting to mention that this requirement for shared data between users is ideally suited to the use of distributed version control tools, such as Git and Mercurial, which make versioned exchange of files for this purpose much easier.

V. Conclusions

In this paper we showed that Jenkins plays a complementary role with MDV tools, such as vManager. Jenkins is an industrial strength Open Source tool, used to automate the continuous build of software (or in this case chip design) projects, monitor the execution of externally run jobs, and integrates seamlessly with commonly used SCM solutions. This is a useful layer on top of what we commonly consider the MDV environment, which provides specialized, sophisticated regression and analysis capabilities.

Jenkins does not replace MDV tools, but instead provides further useful automation, and a way to easily publish a developer-view of regression results and code status. We can leverage features and plugins in Jenkins to make reports from the MDV regressions available to developers, where detailed analysis is required.

Using vManager and vManager C/S as examples, we show that it's possible to integrate Jenkins with the MDV environment quickly and neatly. We provide implementation code for the generic part of this solution, written in Python, using standard libraries. We also provide example code for the two specific examples. The code we've provided should make it possible to get a basic CI server installed and running with vManager in just a few hours.

Acknowledgements

The authors would like to thank everyone in the Cadence team for helping with vManager, especially the e-language and new TCL APIs for vManager and vManager C/S respectively. In particular a big thanks to John Brennan, John MacBeth, John Nehls and Dan Leibovich.

References

[1] JL Gray, Gordon McGregor, "A 30 Minute Project Makeover Using Continuous Integration," DVCON 2012

[2] http://nelsonwells.net/2012/09/how-jenkins-ci-parses-and-displays-junit-output/

[3] http://windyroad.com.au/dl/Open%20Source/JUnit.xsd

[4] http://stackoverflow.com/questions/136168/get-last-n-lines-of-a-file-with-python-similar-to-tail

[5] http://pymotw.com/2/csv/

[6] http://martinfowler.com/articles/continuousIntegration.html

[7] https://bitbucket.org/verilab/jenkinsintegration

[8] Incisive® Enterprise Manager Managing Regressions Appendix D Enterprise Manager Batch API