Agile regression system testing

Agilt systemregressionstest

ANDREAS NORDVALL

KTH Technology
and Health

Master of Science Thesis
Stockholm, Sweden 2012

Detta examensarbete har utförts i samarbete med
Ericsson
Handledare på Ericsson: Thomas Kjellberg

Agile regression system testing

Agilt systemregressionstest

Andreas Nordvall

Examensarbete inom
Datateknik
Grundnivå, 15 hp
Handledare på KTH: Micael Lundvall
Examinator: Thomas Lindh
Skolan för teknik och hälsa
TRITA-STH 2012:72

Kungliga Tekniska Högskolan
Skolan för teknik och hälsa
136 40 Handen, Sweden
http://www.kth.se/sth

Abstract

This report describes the work on automating the testing of nodes at CCS (Common Control System) in Ericsson. The goal was to every three hours configure nodes with the latest build and run the tests. This process is to be fully automatic without user input. The existing configuration tool CICC (Core Integration node Control Center) is to be used for configuration. Before work started fault reports were analyzed and creating a usecase for testing restarts should reduce some faults.

The first step was to make the configuration tool CICC automated. To schedule the testing the continuous integration tool Jenkins was used. But Jenkins can't by itself run CICC nor interpret the result. Therefore a wrapper layer was implemented. When the wrapper is finished it stores the results of the configuration run in a XML (eXtensible Markup Language) file, which Jenkins reads. Results can then be seen in Jenkins through web interface. If there were any failures during configuration or testing the failed step will have an error message.

The project shows that automation is possible. Automating the testing reduce the time for correcting errors because they are more likely to be found early in the process. Before implementing this project in production some improvements should be made. The most significant improvement is making the configuration and testing of each node parallel with each other, in order to make the time limit for configuration and testing less of an issue.

Sammanfattning

Denna rapport beskriver arbetet med att automatisera testningen av noder hos CCS på Ericsson. Målet var att var tredje timma konfigurera noderna med binärfiler kompilerade från den senaste källkoden och sedan testa dem. Detta ska ske helt automatisk utan att användarens hjälp och konfigurationen ska använda det befintliga konfigurations verktyget CICC. Innan arbetet påbörjades skulle felrapporter analyseras för att se om det fanns något att tjäna på automaseringen.

Uppgiften löstes genom att först titta på felrapporterna och konstatera att det fanns rum för förbättringar, främst gällande omstarter. Efter det automatiserades CICC som tidigare körts via en GUI. För att schemalägga konfiguration och testning användes testverktyget Jenkins. Jenkins använder sig av ett s.k. wrapperskript som kör CICC och testfallen. Wrapperskriptet sköter även felhanteringen och skriver sedan resultatet av körningen till en XML fil som läses av Jenkins.

Resultaten av testen går sedan att se i Jenkins via ett webinterface. Där går det att se resultatet av wrapperskript körningen och testerna, om det blev några fel finns det felmeddelanden med anledningen till felet. Misslyckade tester visas också.

Projektet visar att med automatisk testning som sker oftare kan fler fel hittas tidigare och därför åtgärdas snabbare. Innan arbetet används skarpt bör förbättringar ske som tillexempel att köra konfiguration och testning av olika noder parallellt med varandra i wrapperskriptet, för att klara tidsbegränsningen när det är flera noder.

Contents

1. Introduction

1.1. Background

PDU (Product Development Unit) Platforms at Ericsson is switching to a more agile way of working. Because of this PDU Platforms will go from Daily Build, where code is built and regression tested once per day, to Continuous Integration, in which code is built and tested once every three hours.

As a part of this change CCS, part of PDU Platforms, wants to run low level test of their code at the same time as the high level tests to reduce the FST (Fault Slip Through) and improve the turn-around time for fixing the faults.

1.2. Goal and requirements

The goal of this project is to create an automated testing process for CCS and analyze if there is anything to gain from it. The requirements from Ericsson were:

*Analyze bug reports and find at least one area where improvements could be made. Reference to bug reports should exist for the chosen area.

*Specification of at least 10 new test cases in the area identified during the analyze phase. Existing framework should be used if possible.

*Automatic configuration of the system with the help of existing tools. The configuration should run before every test iteration. The configuration should not take more than 30 minutes. The configuration shall not use a GUI. The tool should configure as much of the system as it can.

*Modify the chosen test framework so it can be used with systems in smaller labs.

The project could also achieve the following:

*Create additional test for the identified area but other areas as well.

*Improve the tools to make it possible to configure larger systems.

1.3. Scope and limitations

Existing tools will be used as much as possible. Improvements of the tests will be in areas that is easy to understand and don't require too much system knowledge. The configuration tool will support the CBM1, CMB3 and EPB1 cards.

This report will focus on the node configuration before testing. The testing process will be covered in *Agile regression system testing*, 2011, by Anwar Aodah and Bora Öcüt [5]. The main goals for this project are:

*Analyze bug reports and find at least one area where improvements could be made. Reference to bug reports should exist for the chosen area.

*Automatic configuration of the system with the help of existing tools that runs before

every test. The configuration should not take more than 30 minutes. The configuration shall not use a GUI (Graphical User Interface). The tool should configure as much of the system as it can.

The project could also achieve the following:

*Improve the tools to make it possible to configure larger systems.

1.4. What is Agile?

Agile software development is based on iterative and incremental development. The term was introduced in the Agile Manifesto [10] in 2001. One of the principles in the manifesto is *"Our highest priority is to satisfy the customer through early and continuous delivery of valuable softwar*e. ". Continuous delivery requires continuous testing to make sure the code is working correct. For more information on agile see the Agile Alliance webpage [11]

2. Current situation

PDU platforms at Ericsson build and test all of its code once per day. During this test all code is built and then tested for faults. There will be a change to agile work methods at Ericsson. Therefore there will not be daily tests but continuous testing every three hours instead. [6]

Since the test will be running more often the CCS department at Ericsson want the run their own tests parallel to the complete build. Testing CCS separate without building everything else on top of it should reduce the FST and reduce turnover when finding faults.

The entire test process should be automated. But before running tests the node must be configured. The tool to configure used to configure the nodes is called CICC. It's a script written in Expectk which provide the user with a GUI to help with the configuration. For this project CICC must be able to run without user input. CICC have some support to do this but the code is outdated and don't support all of the cards so it must be updated to support them.

The nodes are mainly at two locations, the big lab and minilab. When working towards the big lab it's possible to telnet to nodes directly. To connect to the nodes in the minilab require a SSH connection to the gateway machine. Then from the gateway machine it's possible to telnet the nodes.

3. Preliminary study

The FST (Fault Slip Through) was analyzed to see how much was to gain on improving the testing process. There was also the question about how to implement a solution and what environment to use.

3.1 Analysis of Fault Slip Through

Fault Slip Through are faults which are detected later in the process than they ought to be [3]. At every stage from design to finished product there are tests supposed to find errors, bugs and faults. Detecting errors early has several benefits such as it's easier to find and fix. A defect detected post-release that should been found when working on the architecture will be 25-100 times more expensive to fix [8]. Therefore when errors are detected later then they should have been they are classified as a FST error. This way there is statistics available to improve the testing process so fewer faults are missed the next time.

During the last three months there was a total of 40 FST originating in subsystems BABS and ICS, meaning they should have been found during PDU Platforms testing. These where then separated into different groups depending on their nature.

- Eight software related faults, nodes that behaved incorrect because of procedures not working correctly or returning the correct values.
- Four hardware related faults. Nodes that behaved incorrect because of faults in the hardware.
- Four errors in crash handling and recovery. These errors were related to log files and crash dumps not being made correctly.
- Seven faults related to restarting the node. Most of it was the node taking too long to start and sometimes starting incorrectly.
- Eight formal issues. These weren't faults really most of them was code reviews and a FST was made to make sure the review was done.
- Five for adding new/missing functionality. These where for functionally that was forgotten or needed to be added.
- Two FST regarding documentation. The manual was wrong and had to be corrected.
- Two syntax errors. Unbalanced code there brackets was missing.

Improving testing of restarts first should be easier and it should reduce the FST. Writing a usecase to test the restart of the node you restart them several times, both cold and warm restarts, and make sure they are back up and responding within twenty seconds to two minutes depending on what restart was made. If they aren't there is something wrong with the restart that needs to be investigated and dealt with. Software related errors are harder to find compared to the restarts. The software related issues are very different from each other and it isn't possible to just test them by doing a restart. It needs more thoroughly testing. Therefore was it decided to start with doing new test cases for restarts.

3.2 Proposed solutions

CICC was to be used as the tool for configuration but the configuration and testing should be continuous. Systems already being used at Ericsson were looked at in first hand. In the end there were two options that seemed workable. The current testing tool Daily Test or the continuous integration tool Jenkins, which was used by another department at Ericsson.

3.2.1 Daily Test

Daily Test is a process made by Ericsson and is used to test code every day. During the night Daily Test builds the code and the next day it's possible to see the result of the build. Daily test was deemed too difficult to use because making the configuration tool CICC to work with Daily Test would require an almost full rewrite of CICC. In Daily Test CICC wouldn't be able to access binaries and other files the way it's done now and CICC have a lot of files to access.

3.2.2 Jenkins

Jenkins is an open source application used for automatic testing. Working with Jenkins would not require a complete rewrite of CICC. Jenkins could be scheduled to call a wrapper script, no work was done with Jenkins because it's used in production, this wrapper script will then do the configuration and run the tests. When the wrapper script was finished it would have to write a XML file that Jenkins would read to know the result of the testing. This way CICC would work as it is because it would be accessed through a normal shell by the wrapper script. This seemed like the better solution and was the one chosen.

3.2.3 Wrapper for Jenkins

The wrapper would do three things. First configure the node with CICC, after configuration the wrapper will run the tests and finally report the result to Jenkins. Expect was the language chosen for the wrapper because then everything could be tested manually and later when automating the task it is done exactly the same way. CICC had to be automated so it could run from single command from the shell without user input. The wrapper could call CICC and then expect a start and end message. Any other messages will mean that CICC had an error and couldn't complete the configuration.

4. Technical background

Here are the script language and tools used described briefly.

4.1 Expect

Expect is a script language based on Tcl. Don Libes writes in his book Exploring Expect, 1995:

> *"Expect is a program to control interactive applications. These applications interactively prompt and expect a user to enter keystrokes in response. By using Expect, you can write simple scripts to automate these interactions. And using automated interactive programs, you will be able to solve problems that you never would have even considered before." [1]*

What makes Expect work is the expect command. With the expect command it's possible to match the response from the shell with regular expressions and proceed with the script depending on the answer. It makes it easy to use because it's done like it would have been done manually. But instead of having a user reading the output and reacting based on it Expect do it for the user.

4.2 Jenkins

Jenkins is the platform that will be used for the continuous integration testing. No work was done in Jenkins because it is used in Ericsson live environment. Jenkins was set up to call a script which we could modify and then read the results from a file.

> *"Jenkins is an award-winning application that monitors executions of repeated jobs, such as building a software project or jobs run by cron. Among those things, current Jenkins focuses on the following two jobs:*
>
> 1. ***Building/testing software projects continuously**, just like Cruise-Control or DamageControl. In a nutshell, Jenkins provides an easy-to-use so-called continuous integration system, making it easier for developers to integrate changes to the project, and making it easier for users to obtain a fresh build. The automated, continuous build increases the productivity.*
> 2. ***Monitoring executions of externally-run jobs**, such as cron jobs and procmail jobs, even those that are run on a remote machine. For example, with cron, all you receive is regular e-mails that capture the output, and it is up to you to look at them diligently and notice when it broke. Jenkins keeps those outputs and makes it easy for you to notice when something is wrong. " [2]*

4.3 CICC

Currently the nodes are configured with a tool called CICC. CICC uses a graphical interface based on the script language Expectk. The user sets the configuration with the graphical user interface, theses configurations can be saved to a file and later loaded.

After the configuration is set the user configure the node by going through several steps. During this process the user is prompted for answers on a number of questions such as the location of files and questions if the user wants to delete old files and restart the node.

Figure 4.1 The CICC user interface

Here follow a brief explanation of all the steps, buttons to click, to configure the node.

Configure – GUI for the configuration. It contains connection information to the node, node name and other info. It's possible to save and load the configuration file.

Get HW Info – Get hardware information gets the node type and slot information. The information will be saved to a file.

Create db_lazy & ARMAMENT – Creates the db_lazy and ARMAMENT file based on the hardware information file. The files decide how the node should be configured. The user can chose different configuration in the GUI.

Send files to target – Transfer files to the node using ftp and then reloads the node.

Modify hardware config file – Not used during a standard configuration.

Execute db_lazy script on target – Runs the db_lazy script on the node to configure it. After this step the node is configured and in basic.

Update db_lazy & ARMAMENT – Updates the db_lazy and ARMAMENT file.

U-BOOT test – Not used for configuration. Used to test the U-BOOT

Quit – Quits the script.

5. Implementation

5.1 Automating a default configuration with CICC

The first step was to document the CICC API (Application Programming Interface) on an internal Ericsson wiki page. If the script is going to be able to run without user input it must have all the information available when starting. By documenting the API it's easy to see what CICC expects and what it returns on the different procedures. Trying to do this without documenting it first would take a lot of time because CICC consists of over 10000 lines of code.

CICC should be able to run from a function call the name of the configuration file that has connection info to the node. After that all files will be saved in a default path where CICC can find them. Any prompts will be answered with the default answer. If CICC can't complete the configuration for any reason the user will be informed of the failure with a message for it and then CICC will exit.

There has been some work on automating CICC already. Unfortunately it doesn't support the CBM1, CMB3 and EPB1 cards. It must be modified to support them. The error handling in CICC is very good and in most cases correct and easy to locate. Since CICC are divided in pretty distinct steps the method to automate it was to run CICC automated and see where it failed. Then identify where, and why it failed, and then fix the error so it worked as intended. Then when the entire step was complete it was tested several times until there were no issues with the code and then move on to the next step.

It was done this way because the code is extensive but it is divided into procedures and errors were usually procedures returning the wrong values because they used outdated commands and methods. When the procedure returns a correct value again the code work some procedures had to be rewritten because they were outdated. When the script worked additional testing was made and more errors that only arose in less common situation was found and fixed.

5.1.1 Errors in CICC and solutions

Here follows all errors found and fixed while running a default configuration.

Error in startup script: invalid command name "create_sel_list"
The script tried to call a procedure that no longer used since it isn't needed anymore. The function call was removed

Error in startup script: couldn't compile regular expression pattern: parentheses () not balanced
Just as the error says the expression wasn't balanced. A parenthesis was removed.

*** ERROR: Mount drive "//c2"
The wrong command was used during reload, because of that the node started in an incorrect state. The command was changed to the right one.

Now setting IP address on target...*** ERROR: Attempt to execute
CICC determined the node state wrong and thought it was in one state when in fact it was in another. New code was written to determine the node state.

*** ERROR: Failed to reformat the /d partition on following slot(s): 1. Aborting...
The format command removed all files but one. The file left is used for system recovery but the script expected nothing to be left. Added an exception for the file the formatting is considered a success if only that file is left.

***ERROR: No SMN number found in te log. Check your hardware and try again.
The te log got flooded so searching it took longer fifteen second which led to a timeout. Now it's cleared before every run of the script.

*** ERROR: Unexpected response from 'board_status': mirror: command not found
The command shouldn't exist but it used to say unknown command instead of not found. Added not found as a response to expect.

TARGET PATH=/c/java/install
The CICC determined the current state wrong. New code was written to determine the node state.

Error creating db_lazy and ARMAMENT file.
***ERROR: Cannot find any Core configuration for ROJ208394/1 board at 0,7. Aborting...
The EPB1 card is new and there was no code on how to configure them. Default configuration of EPB1 cards was added, EPB1 + RPU.

When running a default configuration if the password for connecting to the gateway wasn't in the configuration file CICC prompted the user for it. Code was added to prevent prompts during a default configuration.

5.2 Wrapper for Jenkins

Jenkins isn't capable of running CICC and know if it was successful on its own. Therefore a wrapper had to be made. Jenkins call this wrapper through the source script and when it's finished Jenkins get the result from an xml file made by the wrapper. The wrapper will consist of procedures for configuration of the node, testing of the node and generating an xml file.

Expect was used to make the wrapper because it has been used before and is well suited for the task. Automating it means making the script what is done manually. A configuration file was made for the wrapper. The configuration file contains the node name, the path to the test and the name of the test loadmodules that should be run. Picture 5.1 is an example of a configuration file for the wrapper

```
{
enolans_split
/d/loadmodules
xjobb.ppc
/vobs/cpp/src/babs_ss/control_test_dm/continous_ccs_test/bin
} {
qthokje_split
/d/loadmodules
xjobb.ppc
/vobs/cpp/src/babs_ss/control_test_dm/continous_ccs_test/bin
}
```

Figure 5.1 Contains of a configuration file for the wrapper.

In this case there are two nodes and they are running the same test. Above is how the configuration file is built. Each node has four lines of info and has brackets before and after.

- The first line contains the node name.
- Second line is in what directory on the node the test should be put in before being run.
- On the third line is the name of the test.
- The fourth line is a directory path to where the test file is located.

```
{
<node name>
<directory on the node to ftp test file to>
<name of the test file>
<directory test file is located in>
}
```

Figure 5.2 Adding additonel nodes to the configuration file.

The node name is used as an argument when calling CICC. With it CICC can locate the node configuration file for that node and configure it according to it. This means that node configuration files must be created manually with CICC for the nodes. But this is only done once and doesn't take too long. The wrapper script will also use the node configuration files to get the connection info to the node so it can connect and run the tests.

Configuration and testing of the nodes are run one after another. After the first node is configured and tested it will move on to the next one. After every procedure in the script the result of it is added to a list. Everything in the list will later be written to the xml file. If there is an error it will be written to the list and the script will start on the next node. After all nodes have been tested the script will generate the xml file and then exit.

5.2.1 Transfer of test files by the wrapper

After configuration and before running tests the test file must be transferred to the node. This is done by the wrapper. There is two different ways it is done. The node is either located in the lab or minilab this info is available in the configuration file for CICC,

which the wrapper can locate and read. If the node is in the regular lab FTP (File Transfer Protocol) is used once. The configuration file for the wrapper has the info on where to get the file is located and where it should go.

If the node is in the minilab it's more difficult. First SFTP (SSH File Transfer Protocol) is used to transfer the file to the gateway machine, the ip-address to the gateway machine is in the CICC configuration file, and put in a transfer directory. Then the script SSH to the gateway machine and from there FTP the file to the node.

After sending the files to the node the wrapper will connect to the node, if it's in the minilab it will connect through the gateway, and then run the tests in the file just transferred.

5.3 XML file for Jenkins

For Jenkins to know if the configuration and testing was successful or not it will check the xml file defined in Jenkins for the result. Jenkins has a plug-in installed that supports XML test reports that are made by ANT and compatible with JUnit [4] [7] to see how its built see figure 5.2.

```
<?xml version="1.0" encoding="UTF-8"?>
<testsuites name="CTs for XYZ" errors="12" failures="5" tests="24" host-
name="localhost" time="...">
    <testsuite errors="3" failures="0" tests="3" id="abcde_tests" name="CTs
for ABCDE" time="...">
        <testcase classname="abcde" name="abcde.ct1" time="..." />
        <testcase classname="abcde" name="abcde.ct2" time="..." />
        <testcase classname="abcde" name="abcde.ct3" time="...">
            <error type="Timeout issue">[log]</error>
        </testcase>
    </testsuite>
    <testsuite errors="0" failures="5" tests="8" id="fghjkl_tests" name="CTs
for FGHJKL" time="...">
        <testcase classname="fghjkl" name="fghjkl.ct1" time="..." />
        <testcase classname="fghjkl" name="fghjkl.ct2" time="..." />
        <testcase classname="fghjkl" name="fghjkl.ct3" time="...">
            <failure type="Timeout issue">[log]</failure>
        </testcase>
    </testsuite>
    ...
</testsuites>
```

Figure 5.3 The layout of the xml file.

JUnit intended as a testing framework for Java applications [9]. But it is possible to use the XML format to report the result from the wrapper. Every node will be a testsuite and then the testcase will be used to indicate the result from configuration and testing. Errors will be used when the wrapper failed because of wrong configuration info or other mishaps which prevented the tests from running. Failures will be used for tests on the node which didn't pass.

5.4 Seeing the results in Jenkins

After the wrapper have run Jenkins will read the xml and the result can be seen on with a browser.

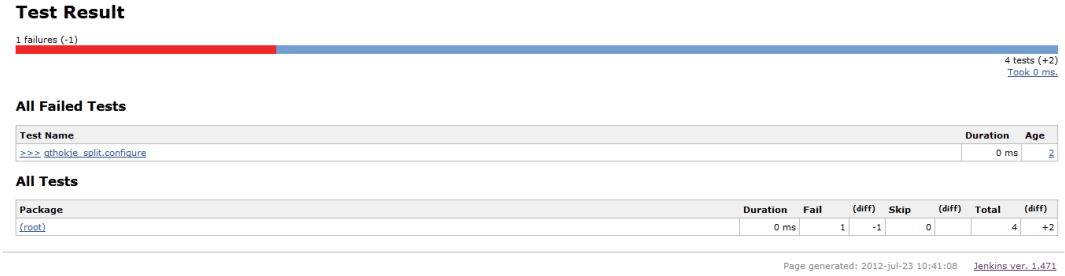


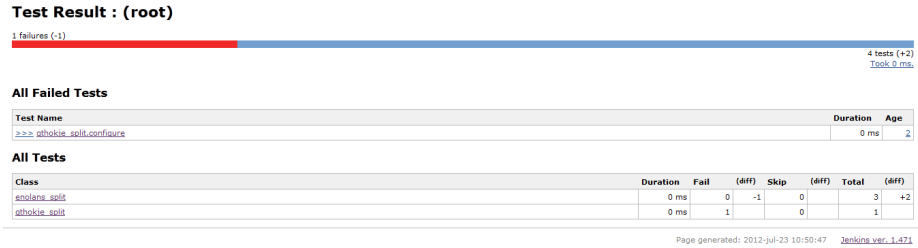
Figure 5.4 the qthokje_split node had an error during configuration!

This is how the result looks in Jenkins. The configuration figure 5.1 used was and the qthokje_split node failed during configuration and the enolans_split node passed the test. Pressing qthokje_split.configure will give us the reason for the error.

Figure 5.5 the error qthokje_split had. It couldn't read the configuration file.

This is the error message the wrapper got from CICC and it wasn't unexpected because the qthokje_split.config file is incorrect. This made CICC fail trying to read it. Pressing

root will show the full overview.



Test Result : (root)

1 failures (-1)

4 tests (+2)
Took 0 ms.

All Failed Tests

Test Name	Duration	Age
>>> qthokje_split.configure	0 ms	2

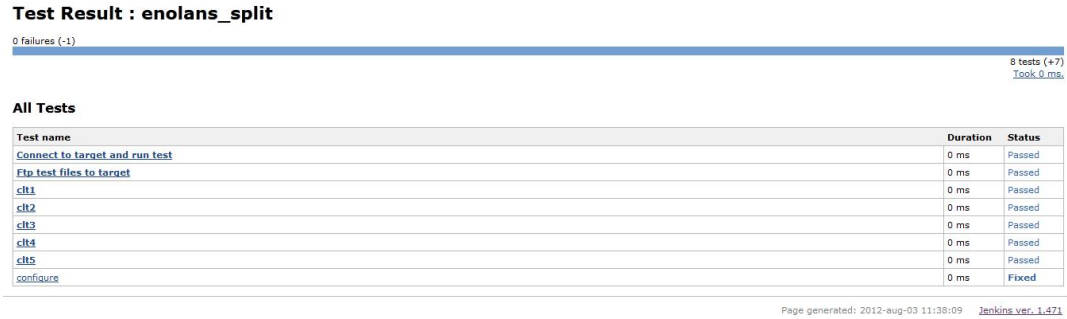
All Tests

Class	Duration	Fail	(diff)	Skip	(diff)	Total	(diff)
enolans_split	0 ms	0	-1	0		3	+2
qthokje_split	0 ms	1		0		1	

Page generated: 2012-jul-23 10:50:47 Jenkins ver. 1.471

Figure 5.6 no errors on enolans_split.

The enolans_split node passed the test and qthokje_split failed during configuration. Figure 5.4 is from another test on the enolans_split node. Clt1-5 is the tests which all passed. Configure are for CICC, the reason it says fixed is because it failed the previously run. Connect to target and run tests only fail if the script for some reason was unable to run all of the tests. Ftp test files to target fails if the test files can't be sent to the node.



Test Result : enolans_split

0 failures (-1)

8 tests (+7)
Took 0 ms.

All Tests

Test name	Duration	Status
Connect to target and run test	0 ms	Passed
Ftp test files to target	0 ms	Passed
clt1	0 ms	Passed
clt2	0 ms	Passed
clt3	0 ms	Passed
clt4	0 ms	Passed
clt5	0 ms	Passed
configure	0 ms	Fixed

Page generated: 2012-aug-03 11:38:09 Jenkins ver. 1.471

Figure 5.7 a successful test on the enolans_split node.

6. Result

The project resulted in automatic testing and configuration of the CBM cards, with limited support for EPB1. CICC was modified to fully support the default configuration function, an EPB1 card must manually be set in the non-forced backup state to work, and then used by Jenkins through a wrapper to run every three hours. Testing is done on the node after configuration.

6.1 Cicc

The configuration tool CICC was improved to work without a GUI. Previous work on it had been done but it wasn't finished and some things were done in an incorrect way, correct for the old cards but wrong now. When all the errors for the CBM cards were done work started on supporting the EPB1 card. First a default configuration had to be added to the EPB1 card since it's new. The configuration is same as CBM cards but with an EPB1 instead.

CICC does not fully support the EPB1 card. If the card is manually set in non-forced backup CICC will work in most cases. If CICC is going to take the card automatic to non-forced backup it will not work. This is because the EPB1 card is unstable and a restart can take somewhere between one to six minutes. It also had a tendency to briefly go up in forced backup and after five minutes or so maybe switch to non-forced backup. Getting to non-forced backup also required the initial image to be cleared. After clearing and restarting the card it would start in U-boot and then a new image had to be TFTP (Trivial File Transfer Protocol). But doing this would in a lot of cases just outright break the card and require a hard restart to get it working again. Therefore it's only supported when it is already set in non-forced backup.

6.2 Automation of the configuration and testing

The testing will be scheduled to run in Jenkins every three hours. Jenkins calls the wrapper. The wrapper script then reads a configuration file which contains info on the node names and what test to run on them. Then each node is configured and tested one after another. If the script fail during the configuration or setting up the test it will abort testing and report an error later to Jenkins. If the testcase is failed a failure will be reported to Jenkins. After all nodes been tested the results will be written to the xml file then when the wrapper script exits Jenkins will read the result from the xml file. The result can then be seen through the Jenkins web interface.

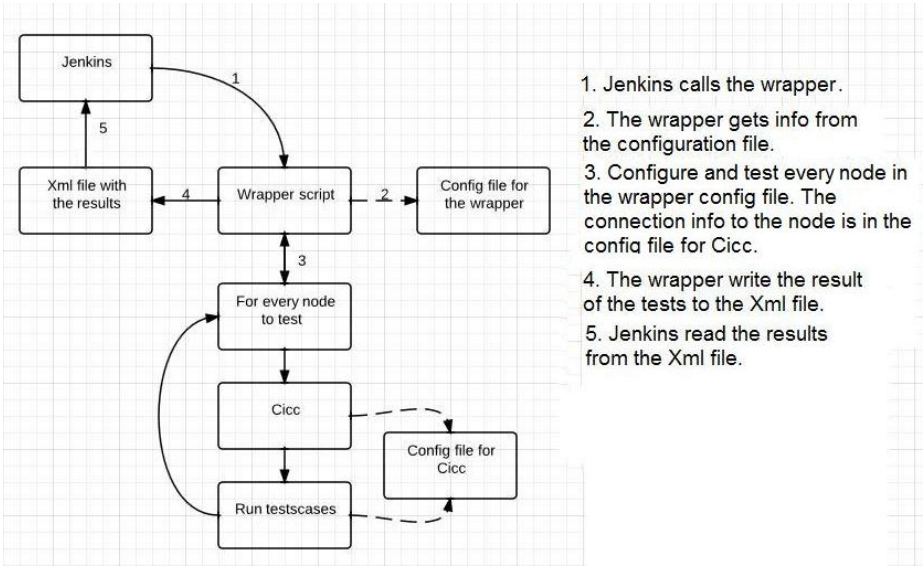


Figure 6.1 picture of how the testing works.

CICC is called by the wrapper with the node name as an argument. With the node name CICC can locate the configuration file located in the ./nodes directory. This configuration file has to be created manually in CICC before doing an automatic run, with the default name. The wrapper also accesses the same CICC configuration file to get the connection info to the node so it can connect to the node, ftp files and run the testcase. To change what tests to run and on which nodes changes are to be made in the configuration file for the wrapper.

7. Conclusion

The configuration and testing are automated and can run every three hours in Jenkins. CICC is used to configure the nodes and have been improved to support automatic configuration of the CBM1 and CBM3 cards. There is limited support for the EPB1 card but it's not fully supported because it is unpredictable and more work is needed to cover all ifs.

After configuration the node will be tested with the test cases defined in the configuration file. For more information about this see *Agile regression system testing*, 2011, by Anwar Aodah and Bora Öcüt [5]

This project has proved that automatic configuration is possible to implement at CCS. This should reduce the FST, if new tests are written to keep up with the development, are and mean less time spent on resolving errors. By building and testing the code every three hours instead of once per day a programmer can see what impact changes have earlier and fix them if something doesn't work as intended.

8. Future work

Several things should be improved on if this work is to be used in production.

The following improvements should be made to CICC:

- Add functionality to CICC so it can build the newest binaries before configuration. As it stands now CICC configure the nodes with the latest binaries but they are built once per day.
- Add full support for the EPB1 card by implementing procedures to deal with setting it in the correct state no matter the card behaves.

These improvements would be good to make in CICC:

- Make it so CICC tries an automatic configuration again if it fails the first time. Most errors during configuration happen once and on a second try there are no problems. This could be done in the wrapper but fixing the problem at the source would be preferable.

The following improvements should be made to the wrapper for Jenkins:

- All the testing and configuration should be done parallel with each other. Now the nodes are configured one after another in a loop. This takes too long when there are more nodes to configure.

References

[1] Libes, Don, Exploring Expect, A Tcl-based Toolkit for Automating Interactive Programs, O'Reilly & Associates, Inc. 1995

[2] Jenkins CI, *Meet Jenkins*, 2012-04-27, https://wiki.jenkins-ci.org/display/JENKINS/Meet+Jenkins (2012-07-06)

[3] Z. Antolic, *Fault slip through measurement process implementation in CPP software verification*, 2007-05-16 http://www.ericsson.com/hr/about/events/archieve/2007/mipro_2007/mipro_1187.pdf (2012-07-16)

[4] Jenkins CI, *JUnit Attachments Plugin - Jenkins - Jenkins Wiki*, 2012-07-13 https://wiki.jenkins-ci.org/display/JENKINS/JUnit+Attachments+Plugin

[5] Aodah, A. and Öcüt, B. Agile regression system testing, 2011,

[6] Thomas Kjellberg, Software Developer, Ericsson

[7] JUnit.org, *Resources for Test Driven Development*, 2008-06-08 http://www.junit.org/node/399 (2012-08-02)

[8] McConnell, Steve (2004). *Code Complete* (2nd ed.). Microsoft Press. p. 29. ISBN 0-7356-1967-0.

[9] Methods & Tools, Fall 2010, http://www.methodsandtools.com/mt/download.php?fall10 (2012-08-10)

[10] Beck, Kent (2001). *Manifesto for Agile Software Development*. Agile Alliance. http://agilemanifesto.org/ (2012-08-30)

[11] Agile Alliance http://www.agilealliance.org/ (2012-08-30)

TRITA-STH-2012:72

www.kth.se