Codmon: A multi-platform modular test environment.

Berend van Veenendaal

April 3, 2014

TODO:Abstract

Preface

TODO: Preface,acknowledgements

Contents

1 Introduction

This chapter introduces the Codmon-VM project by giving a brief description of background of my research and of the previous version of the Codmon project. It also describes the structure of this thesis.

1.1 Background

In times when software projects become more and more complex, testing of this software becomes more and more important. Many software related problems are caused by lack of testing of the software [14]. Most of the times software testers only test software on one platform. For instance only on a Linux or a Windows platform. Setting-up and configuring again and again all the different test environments on different platforms simply costs too much time. So one of the challenges of software testing is to make sure that the software behaves in the same way on different platforms, without spending too much time on the installation and configuration of the test environment on all these platforms. Even when the test environment is written in such a way that it is able to run on different platforms, there are still issues that must be dealt with, before one is able to run and test the software. So in an ideal world we can test the software without being worried about setting up the test environment.

1.2 Problem indication

Nowdays there are numerous test frameworks and test environments available. For example there is *Junit*[7] for Java-unit testing and *NUnit*[10] for C#-unit testing. There are also different environments like Hudson[3][11], Jenkins[4] which can build a project and run a series of (unit) tests against this project. These frameworks and environments have both their advantages and disadvantages. One of the advantages of unit testing is that a software developer can easily add new *functional* unit tests. One of the disadvantages is that standard unit testing ignores non-functional tests like performance testing and the deployment of the software. Jenkins and Hudson,like Unit tests, also have their disadvantages. For instance, although they both run on several platforms, in their usage they are not really platform independent. For example, if you want to make a connection from Hudson or Jenkins to a remote machine you do this by executing a shell script, so a user must know in advance on which platform this script has to run. So although it is possible to connect to different machines it is not a 100% platform independent environment.

1.3 Problem statement

The test frameworks and test environments mentioned in section 1.2 can be criticized on one or more aspects. What we are looking for is in fact, a combination of the positive aspects of the described frameworks and environments, without the undesirable aspects. So the central question is, is it possible to design a multi-platform, modular test environment? In addition, we study if it is possible to design the test environment

in a *user friendly* way, meaning that it must be possible to easily add both new test cases and software without knowing anything about the internal mechanisms of the test environment.

This thesis describes a multi-platform, user friendly modular test environment called Codmon-VM. The Codmon-VM project provides users with a set of virtual machines, in which Codmon is already installed and preconfigured. The purpose of the virtual machines is to make it easy for users to test software in several environments. When a user wants to test his software on Windows 7, he should download the Codmon-VM windows 7 VM and install it in VM-ware or Virtual box. The same applies for testing his software in an Ubuntu environment. By doing it this way the only tasks a user of Codmon-VM has to do are 1) add their project to an initialization file and 2) add the tests to a so called wrapper file. This will be discussed in more detail in section 4.

1.4 Thesis outline

Section 2 first describes the original Codmon framework and why it was built. It also identifies the problems it has. In section 3, *The road to Codmon-VM*, we explain how we got to the final Codmon-VM design. Section 4 describes the implementation of the Codmon-VM project. It starts with a general description of the project followed by a detailed explanation of the different modules of Codmon-VM. After this we will evaluate the choices and their consequences in section 5. In Section 6 we discuss the results based on section 5. We end this section with a brief discussion of related work.

2 Codmon

In this section we describe the original Codmon framework and its shortcomings. The original Codmon framework was built in 2005 by François Lesuer[8]. Originally Codmon was built for testing and performance monitoring Ibis projects[12][13][9][15][6] on the DAS-2[2] computer. Codmon was able to perform both functional and performance tests. [1] If for some reason a particular test failed, Codmon raised an alarm and reported the failures. Codmon does this by sending an email directly to the programmer who made the last changes in the software that was tested. Codmon also reported in the same way in case the performance drops below a certain threshold. Next to sending an email in case of failing tests, Codmon also marks them on a results web page. It was also the intention that Codmon would be extensible. In this the Codmon programmers succeeded only partially. We will discuss this more in depth in section 2.2.

2.1 The working of Codmon

In this section we will describe the design of the Codmon framework. The Codmon framework is more or less modular and consists of a two parts. The first part is the actual Codmon program. This part is responsible for a few things, which we will explain in a few minutes. When a user wants to test one of the applications mentioned above, the only thing he or she must do is to make sure that the Codmon framework is installed on the Das-2. He or she needs to know nothing of the Codmon program, except for how it should be started. This *core-part* of Codmon is responsible for a few things. first it creates a *shot* of the actual state of the of the programs that are tested. It does this by executing a set of *sensors* and collecting the output of these executions. later in this section we'll explain what sensors are. In case of a performance tests, the sensor file will make the distinction between functional test and performance tests, the test information is stored in a RRD database[1]. From this data averages are calculated and eventually the graphs are generated[8]. The results of a performance test will be compared with the results of previous tests and if the performance result of a test drops below a certain threshold, an alarm is raised. When an alarm is raised, an email is sent to the last contributor of the code of which a test fails. The same happens when one or more functional tests are failing.

The second part of the Codmon framework are the sensor files. A sensor file describes a "test set" that can be executed by the Codmon program. Such a sensor file consists of two parts. First there is the *onoff* part, which is used for compilation parts and the functional tests. Second there is the *graph* part. This part is used for the performance tests. The core of each sensor element in a sensor file is the CMD attribute. This CMD attribute consists of two basic parts: a wrapper and a shell-script command. This shell-script command can be anything. For instance it can be a SVN command, or a ANT job or something else. Listing 1 gives an example of such a senor.

[1] At this moment both the das2 and Codmon aren't in use anymore.

```
<sensor id="update_ok"
    name="CVS update"
    cmd="perl CODMON_HOME/codmon/wrappers/time_wrapper.pl
    CODMON_HOME sh codmon/local/cvsup.sh"
    scope="ibis"
    scheduled="false"
    graph="true"
    fatal="true" />
```

Listing 1: *Sensor example*

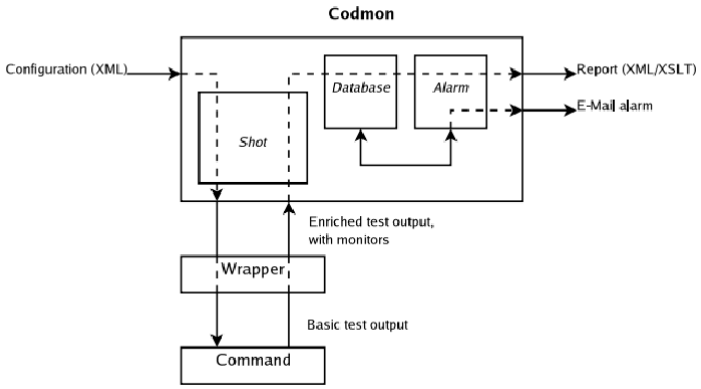
In this case the CMD attribute apparently consists of a wrapper called *time_wrapper.pl* and a shell-script command called *cvsup.sh*. The wrapper indicates the kind of test that is executed. The shell-script command indicates which program is tested. In case of Listing 1 the update time of the CVS repository is is measured.

Where a *sensor* describes the structure of a set of tests, a so called *wrapper* describes the actual test. Wrappers are small programs that are written int the PERL language. The return value of a wrapper indicates if a test was successful or not. In case of a failure an alarm is raised and both the return value and the actual error code will be mailed to the programmer who made the last contribution to the code. The results of the performance tests are plotted in a graph, which makes it easy for the developers to see the performance behavior. It is fundamental to understand that a *wrapper* is not part of the Codmon program itself. Leaving the wrappers outside of Codmon gives a user the possibility to add their own wrappers as well as using general wrappers without knowing the the details of the actual Codmon program. A general wrapper is a wrapper that can be used for each software component. The *time_wrapper* is a good example of a general wrapper. Figure 1 shows a schematic picture of the Codmon structure [8]. More technical details about the Codmon implementation can be found in [8].

2.2 Codmon problems

The goal of this research was to see if it's possible to design a multi-platform, user friendly modular test environment. There are multiple reasons why Codmon doesn't fit the bill. First, to be multi-platform, without applying every change multiple times, the platform itself must be written in a platform-independent language. Codmon uses at least three different languages used. The Codmon program itself is written in Java, which is indeed platform independent[5], so this is not the real problem. As we've explained in section 2.1 the core-part of the sensors is a combination of a *shell-script* command and a *wrapper*. Since a Linux shell-script usually won't work on a Windows environment this part is definitely not platform independent. The same can be said about the PERL language, this will without special effort, also not work on a Windows environment. Next to this there are also several separate scripts for example, for CVS-checkouts and the startup of the Codmon framework. Taking this into account, we

Figure 1: *figure 1: Codmon*



can easily see that the Codmon framework is far from platform independent. Due to the chaos of different scripts and languages it is also difficult for programmers to add new modules or tests to the Codmon framework. The second one is the structure of the wrappers. As you could see in section 2.1 the CMD attribute of a sensor consists of two basic parts: a wrapper and a shell-script command, of which the shell-script command could be anything. From a users perspective it would be much easier if there was only one type of command possible, an Ant-job for instance. A third reason is the portability of the test-framework. Codmon requires kind a bit of configuration,of which you don't want to bother anyone, before a user is able to use it. In the reminder of this thesis we'll explain how Codmon-VM solves these and other issues.

3 The Road to Codmon-VM

As you could read in section 1.3 we're looking for a multi-platform, user friendly modular test environment. This environment must at least satisfy the next few requirements. The most important requirement is that it runs on multiple platforms. A second requirement is that we don't want to bother the users of the test environment with the internal mechanisms of the environment. A third requirement is that is should be modular and pluggable. This means that it is possible for both users to easily add new components to the environment. The same should apply to developers who maintain the environment. A last requirement is that it, to be user friendly, the sensors as described in section 2.1 should all have the same structure so a user can add tests always in the same way.

In the previous sections we have stated that Codmon has some good aspects as well as some aspects that aren't that good. To design a test environment which satisfies the above requirements we decided to reuse the good parts of Codmon and design new components where necessary. Since the core program of Codmon is written in Java, we decided to reuse most of this code and only adapt it if there is no other option. In the section 4 we'll show where,how and why we've adapted the Codmon core program. Because the Codmon core program is written in Java already we've decided to write the all wrappers in Java as well. Using as little languages as possible will help the people who have to maintain the environment.

We also saw that the core of sensor consists out of two parts a wrapper and a shell-script command. In most cases this shell script command will be to start some program. This program should also be platform independent. As we discussed above in Codmon-VM all the wrappers will be written in Java. Hereby writing these programs in Java as well will be an obvious choice. So now we have a Java wrapper and a Java program. We thus need a platform independent construction that, at least, is able to start a program. We have chosen to use *ANT* for this. One of the reasons to choose Ant is that it provides us with a lot of flexibility. Lets take a look at the following example. A user who wants to test a piece of Java software. probably already has a build file for building the software. So there is nothing new here. The only thing the user has to add is a new ant-target, which starts the program. Listing 2 shows the new sensor structure:

```
<sensor id="update_ok"
    name="CVS update"
    cmd="java <wrapper> <path to build file > <ant> <target>"
    scope="ibis"
    scheduled="false"
    graph="true"
    fatal="true" />
```

Listing 2: *Sensor new structure*

The <target> is an optional value. When the target value is left out Codmon will

run the default ant target. In section 4 we'll show how this all is implemented in the Codmon-VM environment.

An other problem we discussed in section 2.1 is the *CVS-Checkout* mechanism. The way Codmon was designed it could only handle CVS-checkouts and updates. The Codmon-VM environment must be able to deal with different versioning systems. To achieve this we decided to design a separate *module* outside of Codmon-VM that takes care of the checkouts and updates of the software that the users want to test. If this modules all implement the same interface, it is easy for (extern-) developers to add new version control systems. This interface should at least define mechanisms for fetching and updating the software. Next to this is must also provide some basic mechanism which provide the users with the history log of the software. The only thing a user has to do, is indicate which version control system his software is using. We discuss the implementation in more depth in section 4.

When you want to test your software on multiple platforms you don't want to download, install and configure the test environment over and over. The Codmon-VM project comes up with a nice and powerful solution for this problem. It provides a set of virtual machines in which Codmon-VM is already installed en pre-configured. The only things a user has to do before he can use the by Codmon-VM supplied tests is to download one or more of these virtual machines, load them into VM-ware and add the details of his software to an init file. By doing is this way the nasty configuration details are hidden for the users of the Codmon-VM environment. Also when a a new version of an operating system arrives, the Codmon-VM developers only have to configure one new image, which is directly available for all the users. How this works exactly we'll see in section 4.

4 The implementation of Codmon-VM

//TODO: Section describes the implementation of the Codmon-VM project.

4.1 The init.XML file

4.2 Version control

4.3 wrappers

5 evaluation

TODO: This section evaluates the choices that are made in the previous sections'

6 Conclusion and related work

TODO: give answers to the questions from section Problem statement TODO: Discus related and future work

References

[1] "JRobin 1.4.0.". http://www.opennms.org/wiki/jrobin.

[2] "Das-2". http://www.cs.vu.nl/das2/.

[3] "Hudson Documentation". http://www.hudson-ci.org/docs/.

[4] "Jenkins Documentation". http://jenkins-ci.org/.

[5] "Henry McGilton" "James Gosling". The java language environment: Contents. May 1996. Section 4.2.

[6] Henri E. Bal Jason Maassen, Thilo Kielmann. GMI: Flexible and efficient group method invocation for parallel programming. March 2002.

[7] "Junit". http://junit.org/.

[8] "François Lesuer". Codmon: a source code monitoring tool. August 2005.

[9] Thilo Kielmann Markus Bornemann, Rob V. van Nieuwpoort. MPJ/Ibis: a flexible and efficient message passing platform for Java. pages 217–224, September 2005.

[10] "Nunit". http://nunit.org/.

[11] "Winston Prakash". Introducing hudson.

[12] "The Ibis Project". http://www.cs.vu.nl/ibis/.

[13] Thilo Kielmann. Henri E. Bal Rob V. van Nieuwpoort, Jason Maassen. Satin: Simple and efficient Java-based grid programming. *Scalable Computing: Practice and Experience*, 6(3):19–32, September 2005.

[14] "Masato Shinagawa" "Toshiaki Kurkowa". Technical trends and challenges of software testing. *Science and technology trends*, 29, 2008.

[15] Rob V. van Nieuwpoort, Jason Maassen, Gosia Wrzesinska, Rutger Hofman, Ceriel Jacobs, Thilo Kielmann, and Henri E. Bal. Ibis: a flexible and efficient Java based grid programming environment. *Concurrency and Computation: Practice and Experience*, 17(7-8):1079–1107, June 2005.