Codmon-VM: A multi-platform modular test environment.

Berend van Veenendaal

May 4, 2014

Abstract

In times when software projects become more and more complex, the testing of this software becomes more and more important. One of the problems with software testing is, that it is difficult to test software in different environments. Codmon-VM provides different virtual test environments on which projects can test their software. Codmon-VM is modular, which means that it is easy to add new test cases to the framework as well as software components that must be tested. It also provides mechanisms to add new pluggable utility modules to the Codmon-VM environment. The same Codmon-VM version runs on different platforms, so it is also multi-platform.

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 [15]. Typically software is only tested on a single platform, for instance only on a Linux or only on 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 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*[9] for Java-unit testing and *NUnit*[10] for C#-unit testing. There are also different environments like Hudson[4][20], Jenkins[6] 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. Both Hudson and Jenkins have the possibility to execute shell-scripts or batch scripts. So, if a user wants to start a test or program he must know in advance on which platform this script has to run. Only then he can decide if he needs a shell script or a batch script. So, although this is a powerful feature 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 com-

bination 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-VM 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. In this case he should download the Codmon-VM Ubuntu VM and install this in VM-ware or Virtual box. 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 the motivation for its development. It also identifies its shortcomings. 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. Next 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[16]. Originally Codmon was built for testing and performance monitoring Ibis projects[5] [21][13][22][18] on the DAS-2[3] computer. Codmon was able to perform both functional and performance tests.[1] If for some reason a particular test fails, Codmon raises an alarm and reports the failures. Codmon does so by sending an email directly to the programmer who made the last changes in the software that was tested. Codmon also reports in the same way in case the measurements change significantly. Next to sending an email in case of failing tests, Codmon also presents the results on a 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 two parts: the part that controls Codmon and the Codmon core program.

2.1.1 Controlling Codmon

We first describe the part that controls Codmon. With controlling we mean which test will run and in which order they will run. There are two different kinds of tests, functional and non-functional tests. The source code of the projects that will be tested is stored in a *CVS* repository. Both the functional and non-functional tests are described in so called *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 functional tests. This includes the compilation of the software, which is a special kind of functional test. These tests can either fail or succeed. Second there is the *graph* part. This part is used for the performance tests. Each test of such a test set is described by a *sensor-element*. 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 sensor.

[1] At this moment both the DAS-2 and Codmon are not in use anymore.

```
1  <sensor id="update_ok"
2     name="CVS update"
3     cmd="perl CODMON_HOME/codmon/wrappers/time_wrapper.
          plCODMON_HOME sh codmon/local/cvsup.sh"
4     scope="ibis"
5     scheduled="false"
6     enabled="true"
7     graph="true"
8     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. Table 1 describes the attributes of a sensor[16].

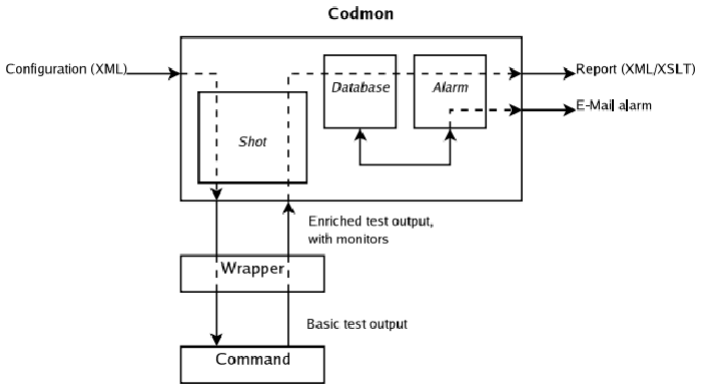
id	Unique ID of the sensor.
name	Name of the sensor.
cmd	Command to be ran, eventually inside the wrapper.
scope	The repository folder which should be analyzed in case of an alarm.
scheduled	Whether this sensor is scheduled downward. If true jobs will be launched in parallel, letting the system care about the real scheduling.
enabled	Whether this sensor is enabled. If true the sensor will be run.
graph	Whether this sensor is to be graphed. If true, a time graph will be generated.
fatal	Whether failing in this step is fatal. If true, no further tests should be executed.

Table 1: Table 1: Codmon sensor

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 in 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.

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. The time_wrapper measures the duration of a test. This can be very useful to see if changes to the program that is tested have influenced the duration of a test. This duration can give an indication if the performance of the software has changed. Figure 1 shows a schematic picture of the Codmon structure [16]. More technical details about the Codmon implementation can be found in [16].

Figure 1: *Codmon*



2.1.2 The Codmon core program

When a user wants to test one of the applications mentioned above, the only thing he must do is to make sure that the Codmon framework is installed on the Das-2. He needs to know nothing of the Codmon program, except for how it should be started. The *core-part* of Codmon is responsible for a few things. First it creates a *shot* of the actual state of the programs that are tested. It does this by *periodically* executing a set of *sensors* and collecting the output of these executions. In case of a performance test, the sensor file will make the distinction between functional test and a performance test, the test information is stored in a RRD database[8]. From this data, averages are calculated and eventually the graphs are generated[16]. The results of a performance test will be compared with the results of previous tests and if the performance result of a test changes significantly, an alarm is raised. When an alarm is raised, an email is sent to the last contributor to the tested code. The same happens when one

or more functional tests fail.

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 does not fit the bill. In this section we'll discuss these reasons in detail.

2.2.1 Multi-platform

First of all, Codmon is not multi-platform. To be multi-platform, without applying every change multiple times, Codmon itself must be written in a platform-independent language. In Codmon, at least three different languages are used. The Codmon program itself is written in Java, which is indeed platform independent[14], 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 Unix shell-script usually won't work in 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 in a Windows environment. Next to this there are also several separate shell-scripts, for example for CVS-checkouts and the startup of the Codmon framework. Taking this into account, we can easily see that the Codmon framework is far from platform independent.

2.2.2 Modularity

The second reason why Codmon doesn't fit the bill, is the modularity in Codmon. Due to the chaos of different scripts and languages it is difficult for programmers to add new modules or tests to the Codmon framework. Adding new tests and modules should be straightforward.

2.2.3 User-friendliness

The third problem is the user-friendliness of Codmon. Next to modularity, which also contributes to the user-friendliness of the test environment, Codmon requires quite a bit of configuration before it can be used. In the remainder of this thesis we'll explain in detail how we have solved these and other issues in the Codmon-VM environment.

Another issue concerning user-friendliness is the structure of the wrappers. As discussed 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[1] for instance.

A third issue regarding user-friendliness is the limited support of version control systems. Codmon only supports *CVS*[2]. From a user-friendliness perspective it would be much better if Codmon would support multiple version control systems, for instance SVN[19] or Git[17]. It also would be nice if other version control systems could be easily added to the test environment.

3 The road to Codmon-VM

As we described in section 1.3 our goal is to develop a test environment that must at least satisfy the following requirements:

- Codmon-VM should be *multiple-platform*, which means that the same source, without making any modifications, can compile and run on multiple platforms.
- Codmon-VM should be *modular*, which means that it must be possible for users to easily add new components to the environment. The same should apply to developers who maintain the environment.
- Codmon-VM should be *user-friendly*, which means that we don't want to bother the users of the Codmon-VM environment with its internal mechanisms of it. Also, 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.

3.1 General decisions

In the previous sections we have stated that Codmon has some good aspects as well as some aspects that aren't that good. For our new test environment we decided to reuse the good parts of Codmon and design new components where necessary to fulfill the requirements described above. Since the core program of Codmon is written in Java and Java is multi-platform we decided to reuse most of this code and to modify it where needed. In section 4 we show where, how and why we adapted the Codmon core program.

3.2 Multi-platform

Because the Codmon core program is written in Java already we've decided to write the all wrappers in Java as well. Using as few languages as possible will help in maintaining the environment. We also saw that in the original Codmon framework the core of a sensor consists of two parts: a wrapper and a shell-script command. In most cases this shell script command will start some program. This program should also be able to run on multiple platforms, so writing these programs in Java as well is an obvious choice. So now we have a Java wrapper and a Java program. We thus need a multi-platform construction that, at least, is able to start a program. We have chosen to use *Ant* for this. The main reason

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 an example of the new sensor structure.

```
1 <sensor id="update_ok"
2    name="CVS update"
3    cmd="java <wrapper> <relative (to the Ant.class) path to build
         file > <ant target>"
4    scope="ibis"
5    scheduled="false"
6    graph="true"
7    fatal="true" />
```

Listing 2: *Sensor new structure*

In section 4 we'll show how this all is implemented in the Codmon-VM environment. The key idea is that all software that is to be tested is started by invoking Ant.

3.3 Modularity

Another problem we discussed in section 2.1 is the modularity of the environment. The way Codmon was designed it could only handle CVS-checkouts and updates. The Codmon-VM environment must be able to deal with different version control systems. To achieve this we decided to design separate *modules* outside of the Codmon-VM core program to take care of the checkouts and updates of the software that the users want to test. Each of these modules implement the *versionControl* interface for a different version-control system. Adding support for a new version-control system implies that the codmon developers (or the users of Codmon-VM) have to write a new module that implements this interface for this new version-control system. This defines mechanisms for fetching and updating the software. Next to this it also provides some basic mechanism to obtain the history log of the software. Listing 3 shows the versionControl interface. The user only has to indicate which version control system his software is using. We discuss the implementation in more depth in section 4.

```
1  /**
2  @author bvl300
3  Basic interface for version control modules
4  */
5  public interface VersionControl{
6
7          /**
8          *Fetches a repository
9          **/
10         public void update() throws VersionControlException;
11
12         /**
13         *If there is no log file it creates the log file.
14         *If there is a log file it updates the existing log file
15         **/
16         public void updateLog() throws VersionControlException;
17
18
19         /**
20
21         *Returns the revision nr of the last commit.
22         *Because this doesn't make sense for all version control
23         *system it may throw a MethodNotSupportedException.
24
25         **/
26         public long getRev() throws MethodNotSupportedException;
27 }
```

Listing 3: *VersionControl interface*

3.4 User-friendliness

When a user wants to test his software on multiple platforms he does not 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 the user with a set of virtual machines in which Codmon-VM is already installed en pre-configured. When a user wants to start using the Codmon-VM environment he only has to do the following steps.

- Download one or more of the Codmon-VM VM's.
- Install them into VM-ware or Virtual box.
- Add the details of the software he wants to test to the initialization file.

Now 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.

Another issue concerning user-friendliness is the one of adding new tests to the sensor file. As discussed in section 3.2 adding new tests also straightforward.

All a Codmon-VM user has to do is add a new test to the Sensor file, and create an Ant target for this test.

4 The implementation of Codmon-VM

In this section we'll describe how we've implemented the decisions made in section 3 in the Codmon-VM project. We'll start with the parts that were necessary to achieve that Codmon-VM is multi-platform followed by the parts concerning the modularity of Codmon-VM. We end this section with the parts that provide user-friendliness. When someone wants to use the Codmon-VM test environment there are two options to get access to this environment. We'll come to these options later in this section. For now we assume the user has access to a working Codmon-VM environment.

4.1 Multi-platform

In the previous sections we mentioned that the tangle of shell-scripts and PERL-code ensures that Codmon isn't multi-platform. To ensure that Codmon-VM is multi-platform we chose to implement the Codmon-VM environment completely in Java. First we describe how we implemented the initiation of a test run. Then we describe how both the tests and the software that is tested are connected together.

4.1.1 Initialization

In a single-platform environment it is very easy to create a shell-script (or a batch-file in case of Windows) to start a program. In Java, this is a little more complex.

In section 4.2 we describe that everything outside the Codmon-VM core program is a separated module, including the start-up module. The start-up module is called with one parameter namely: the name of the sensor file that describes the tests that will be executed in this run. The start-up module is responsible for several tasks. First it initializes the Codmon-VM environment. If it is the first time Codmon-VM is used, four result folders are created: Three history folders and a "current result" folder. Otherwise the results of the previous test-runs are copied to history folders in the current result folder, called *dday*, has space for the new test results [2].

When all the result files of the previous tests are copied to the history folders the Codmon-VM core program can be started. To increase the modularity of Codmon-VM this is done dynamically, so it is possible to start different programs with the same StartUp module. Because this is a completely new Java program we needed a way to pass the sensor file parameter to the core program. Remember at compile it is unknown which program has to be initiated. So to achieve this we have to make use of *dynamic class loading*. Dynamic class loading means that which (Java) class is called is determined at runtime. Listings 8 and 9 in Appendix A show how this is implemented in the Codmon-VM environment.

[2] For historic reasons the result folders are called dday,dday1,dday2 etc.

4.1.2 Ant-connector

A second issue we had to deal with were the sensors in the sensor files. In section 3.2 we have already seen the new sensor structure. A wrapper is a separate process which has to be started by the Codmon-VM core program. When the Codmon-VM core program evaluates a sensor it first extracts the wrapper from the sensor *cmd*-attribute. Then the Ant-target is extracted from the *cmd*-attribute. If we use the sensor of listing 4 the wrapper will be the *TimeWrapper*, while the Ant-target will be *"run"*.

```
<sensor id="checkout_ok" name="TestApps: checkout projects"
  cmd="java CODMON_HOME/codmon/wrappers/classes/TimeWrapper ../../
       local/checkoutApplications ant run"
  scope="checkOut"
  scheduled="false"
  graph="true"
fatal="true"/>
```

Listing 4: *sensor example*

When the Codmon-VM core program has extracted these values it starts a *Wrapper* process (See appendix C). Listing 5 shows how this is done. The Ant target is passed as a parameter to this Wrapper process. This Wrapper, which is a test, starts he Ant-connector, which is a small Java Class (see Appendix C) which evaluates and executes the Ant-target. Such an Ant-target can be a "simple" *build* target or in this case it is the *run* target. This is shown in listing 6. This "run"-target executes the software that is tested by, in this case, the TimeWrapper.

```
final Process pr = new ProcessBuilder(argList)
                   .directory(new File(dir))
                   .start();
```

Listing 5: *Start a new process: The different elements of the CMD attribute of the sensor described in listing 4 are separately stored in the* argList *argument. The dir argument indicates the location from where the process should start*

```
        <!--Run the Checkout program-->
        <target name="run">
                <java  fork="true" classname="Checkout" classpath="
                    ${env.CLASSPATH}" output="out.txt">
                        <classpath>
                                <path location="${jar.dir}/
                                    CheckoutApplications.jar"/>
                        </classpath>
                        <arg value="${arg0}" />
                        <arg value="${arg1}" />
                </java>
        </target>
```

Listing 6: *The run-target*

By implementing it this way it doesn't matter which (Java-)software[3] the wrapper wants to test, it will always be done in the same way as described above.

4.2 Modularity

In section 4.1 we described the improvements we made to achieve that Codmon-VM is multi-platform. In this section we'll describe how easy it is to add new modules to the Codmon-VM environment. To do this we use a module called *CheckoutApplications*.

4.2.1 CheckoutApplications module

Before Codmon-VM is able to test any software, this software must be available for the Codmon-VM environment. To be available this software must have been extracted from the repository where it is stored. The CheckoutApplications-module is responsible for this job. The CheckoutApplications itself consists of a main checkout module and different sub-modules which support a specific version control system. Currently, the CheckoutApplications module supports two different version control systems, namely Subversion (*SVN*) and Git.

The main checkout module is responsible for couple of tasks. First it reads the initialization file, which contains information about the test projects. We'll come back to the initialization file later. This information contains, among others, the kind of version control system that is used for this software. With this information both the update and updateLog methods for this software can be invoked (See Appendix D). For implementing these methods for SVN repositories we've used the *SVNKit* Api[12]. For the Git repositories we've used the *JGit* Api[7].

For a user, now it is easy to add a sub-module for a new kind of version control system. The only thing a developer has to do is implement the update and updateLog methods. And add the calls to this methods to the *checkoutProject* method which you could see in Appendix D. The *update* method is responsible to check if the latest version of the software is locally available for testing. If this is not the case it does a checkout or update on the software's repository. The *updateLog* method is responsible to maintain the log file of the software.

4.2.2 Tests

Adding new tests is relatively straightforward. As long as the test is written in Java a user can just add a reference to this test to the CMD-attribute of a sensor. This works the same way for, both functional and non-functional tests.

[3]Theoretically it is also possible to run non-Java applications, but they are out of scope for the Codmon-VM project. See also section 6.

4.3 User-friendliness

Finally it's time to discuss the user-friendliness of Codmon-VM. As we said before, we don't want to bother the Codmon-VM users with its internal mechanisms. Neither do we want to bother them with its time consuming configuration. This section describes why Codmon-VM is a very user friendly environment.

4.3.1 The initialization file

Before the Codmon-VM environment is able to run one or more tests on a software, it has to know where it can find this software and extract it from its repository. This information and other information has to be added to the *initialization file*, which is an XML file. Listing 7 shows the structure of the init file. The explanation of its elements is shown in table 2.

```
1  <projects>
2          <project>
3                  <name>projectname</name>
4                  <location>http://www.sampleurl.com</location>
5                  <versionControl>
6                          <type>svn</type>
7                          <command>checkout</command>
8                  </versionControl>
9                  <run>true</run>
10                 <user>username</user>
11                 <pwd>pwd</pwd>
12         </project>
13 </projects>
```

Listing 7: *Initialization file*

The only tasks that are left for a user when he wants to test a new software component is adding a new project to the initialization file, create a new sensor file for it and add a run-target to the software's build file (See Appendix E.). When a user wants to use or create his own tests he must create its own wrapper(s) for this.

There is only one issue left that we have to mention here. The initialization file contains two fields, namely *user* and *pwd*. For this thesis it is OK to do it this way, but in real life this would be a security risk.

4.3.2 Virtual environments

We have said already a few times that we don't want to bother the users of the Codmon-VM environment with its configuration. We came up with the idea that it would be nice and extremely useful if a user just use a preconfigured Codmon-VM environment. To achieve this the Codmon-VM project provides

projects	List of all the projects that are involved in a test series.
project	Contains the information of a specific project.
name	The name of the project.
location	The URL of the location where the project can be found.
versionControl	Element that contains specific information about the version-control system
type	The type of version control system that is used for the project. e.g. *SVN* or *Git*
run	**Boolean** that indicates if a project should be tested.
user	username to login into the version-control system
pwd	password to login into the version-control system

Table 2: Elements of the initialization file. In Section 5 and section 6 we'll come back to the security issues regarding the user name and password

a set of virtual machines on which Codmon-VM is already installed and preconfigured. The only thing a user must do is install VM-ware (or Virtual box), download one of the provided Codmon-VM images and install them in VM-ware.

The developers of the Codmon-VM can easily provide new images, by installing a *"clean"* image of the target platform, Windows or one of the different Linux distributions for instance, and install and preconfigure Codmon-VM in this image. When this is done successfully, VM-ware provides options to create a new distributable image of the preconfigured environment.

5 Discussion and Conclusion

In section 1.3 we proposed the following question: *"Is it possible to design a multi-platform, modular test environment"*? In addition, we've studied if it would be possible to design such a 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 this test environment. In section 3 and 4 we showed that, at first sight, the answer to these questions can be answered with yes. Before making that our final conclusion let's first take a deeper look into what we have a achieved and discuss the pros and cons of Codmon-VM.

5.1 Multi-platform

Let's first take a look at the multi-platform requirement. A software program or environment is considered to be multi-platform *when it is compatible with or involving more than one type of computer or operating system*[11]. This will say that Codmon-VM must be able to run on multiple platforms and that its runtime behavior is the same on these platforms. To achieve this we built the whole Codmon-VM environment, in contrast to the Codmon framework, in the Java programming language. To connect the different Java components we made use of Ant. By doing this, at least we are sure Codmon-VM runs on multiple platforms[14].

That Codmon-VM runs on multiple platforms doesn't mean that the runtime behavior is also the same on these platforms. To see that the runtime behavior is the same on these platforms we created different images of Codmon-VM, both Windows and Linux. Running the same tests in these images showed us that the runtime behavior is also the same on these platforms. So we can conclude that Codmon-VMis a *multi-platform* environment 5.1.

5.2 Modularity

The second requirement we have to discuss is the modularity of the Codmon-VM environment. This is a little less straightforward than the Multi-platform requirement. The main question we have to answer is: *when do we consider the environment to be modular?*. Is it modular if a user of the environment can add new tests in a straightforward manner? Is it modular when it is easy to add new, or extend, utility modules like the checkout Applications module? Or is Codmon-VM environment only modular if the core program itself is modular as well? To answer this question we have to consider what the main purpose of the Codmon-VM environment is. When we summarize all the requirements we can see that the environment should be especially easy to use and maintain. This brings us nothing further. What do we mean by maintenance? Are we

talking about the Codmon-VM core program or about adding (and removing) functionality and tests?
When we only take the modularity regarding to usage and maintenance of the environment into account and not that of the Codmon-VM core program, Codmon-VM is also *modular*. When we take the Codmon-VM into account it's only partially modular.

5.3 User-friendliness

Where in section 5.1 the question was straightforward and in section 5.2 the discussion was only about two different viewpoints, the subject of user-friendliness is even more complex. Where one person might say it is user-friendly when the environment is quick and easy to use, another person might say that an environment is only user-friendly when is has a nice and fancy user interface.

Let's first look at how easy the Codmon-VM environment is in its usage. The user can easily download a Codmon-VM environment image and install this in VM-ware or Virtual box. When this is done, Codmon-VM is immediately ready for usage. In section 4.2 we also showed how easy it is to add new tests and (utility) modules like the checkoutApplications module. So when we only take this into account we can conclude that the Codmon-VM environment is user friendly.

Now let's take a look at the second issue. Earlier we've described mechanisms for both adding new software that should be tested, as well ass adding tests to a sensor file. The only difficulty some users could have with it is that they directly have to edit XML files. Doing this, using a nice and fancy user interface could improve the user-friendliness of Codmon-VM. Also, adding both to be tested software and new sensors via a graphical user interface will reduce the risk of errors in these files. Such a user interface could also be used to display the test results. Doing it this way, Codmon-VM would appear as one coherent environment to its users.

So looking at the *user-friendliness* of Codmon-VM it depends of the person how he or she thinks about user-friendliness. Though one thing we can say about Codmon-VM the possibility of preconfigured virtual machines and the way of adding new to be tested software, sensors and utility modules definitely offers some user-friendliness!

6 Future work and Recommendations

Finally we describe what future work and research, regarding to Codmon-VM, could be done to improve it. When we take a look at what we've described in sections 5.2 and **??** we see that the Codmon-VM core program is still one big monolithic program. To improve the maintainability of Codmon-VM it would be useful to see if and how we can transform this monolithic program into a program with a more modular design.

In sections 5.3 and **??** we described different views on user friendliness. In our opinion the biggest improvement could be gained with a nice and fancy user interface. It would really worth while to see if it is possible to combine Codmon-VM with a tool like Jenkins [6]. Jenkins could probably provide a user interface to at least start tests that use different sensor files. It probably also could generate nice and more fancy representations of the test results! It also could provide us with one and the same user interface that could control the whole Codmon-VM environment.

Another interesting question to investigate is: is it possible to test non-Java software with Codmon-VM? It is possible for an Ant target to run non-Java software. But a more interesting question is how can we assure that the software is compiled for the right platform?

The last issue that requires attention is the security regarding the version control modules. As we described in 4.3.1 the username and password are written in plaint text. To prevent security issues some work needs to be done on this.

References

[1] Apache ant. http://ant.apache.org/.

[2] Cvs. http://www.nongnu.org/cvs/.

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

[4] Hudson documentation. http://www.hudson-ci.org/docs/.

[5] The ibis project. http://www.cs.vu.nl/ibis/.

[6] Jenkins documentation. http://jenkins-ci.org/.

[7] Jgit. http://download.eclipse.org/jgit/docs/latest/apidocs/.

[8] Jrobin 1.4.0. http://www.opennms.org/wiki/Jrobin.

[9] Junit. http://junit.org/.

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

[11] Oxford dictionaries. http://www.oxforddictionaries.com/definition/english/multiplatform.

[12] Svnkit. http://www.svnkit.com/.

[13] Markus Bornemann, Rob V. van Nieuwpoort, and Thilo Kielmann. MPJ/Ibis: a flexible and efficient message passing platform for Java. In *Proceedings of 12th European PVM/MPI Users' Group Meeting*, pages 217–224, Sorrento, Italy, September 2005.

[14] James Gosling and Henry McGilton. The Java language environment: Contents. May 1996. Section 4.2.

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

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

[17] Jon Loeliger and Matthew McCullough. *Version control with Git; 2nd ed.* O'Reilly, Sebastopol, CA, 2012. On the cover: powerful tools and techniques for collaborative software development.

[18] Jason Maassen, Thilo Kielmann, and Henri E. Bal. GMI: Flexible and efficient group method invocation for parallel programming. In *LCR '02: Sixth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, Washington DC, USA, March 2002.

[19] Michael Pilato. *Version Control With Subversion.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2004.

[20] Winston Prakash. Introducing hudson. `http://hudson-ci.org/docs/HudsonIntro.pdf`.

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

[22] Rob V. van Nieuwpoort, Jason Maassen, Gosia Wrzesińska, 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.

A Dynamic class loading

```
1  private ClassLoader getClassLoader(String[] jars) throws
       MalformedURLException, SecurityException{
2    ArrayList<URL> paths = new ArrayList<URL>();
3
4    for (String externalJar : jars) {
5      paths.add(new File(externalJar).toURI().toURL());
6    }
7
8    URL[] urls = paths.toArray(new URL[paths.size()]);
9    return new URLClassLoader(urls);
10 }
11
12 /**
13  *@author bvl300
14  *Loads codmon.jar so I can Use it here
15  **/
16 private Method getStartMethod(String[] argv){
17   Method m = null;
18   Class<?> cl = null;
19   String[] jars = getJars();
20   try{
21     ClassLoader loader = getClassLoader(jars);
22     cl = loader.loadClass(argv[0]);
23     m = cl.getMethod("main", new Class[] { argv.getClass() });
24   }catch(Exception e){
25     System.out.println(e.getMessage());
26     System.exit(1);
27   }
28
29   if(!m.isAccessible()){
30     final Method temporary_method = m;
31     AccessController.doPrivileged(new PrivilegedAction<Object>() {
32       public Object run() {
33         temporary_method.setAccessible(true);
34         return null;
35       }
36     });
37   }
38   return m;
39 }
```

Listing 8: *dynamic class loading*

```
1         /**
2         *@author bvl300
3         *Invoke method m with the correct parameters
4         */
5         private void run(Method m,String[] argv){
6                 String sensor = argv[1];
7                 String[] statsArgs = new String[2];
8                 statsArgs[0] = "../sensors-"+sensor+".xml";
9                 statsArgs[1] = "../dday/shot-"+sensor+".xml";
10                try{
11                        m.invoke(null,new Object[]{statsArgs});
12                }catch(Exception e){
13                         System.out.println(e.getMessage());
14                }
15        }
```

Listing 9: *invocation of the method*

B The TimeWrapper

```
1  import java.text.DecimalFormat;
2  /**
3   *@author bvl300
4   *This wrapper measures the time of the module
5   *that is executed.
6   */
7  public class TimeWrapper{
8
9
10         public TimeWrapper(String argv[]){
11                 String dir = argv[0];
12                 String cmd = argv[1];
13                 String target;
14                 if(argv.length==3){
15                         target = argv[2];
16                 }else{
17                         target = "main";
18                 }
19                 long startTime;
20                 double duration;
21
22                 Ant ant = new Ant(dir,target);
23                 ant.init();
24
25                 startTime = System.nanoTime();
26                 try{
27                         ant.run();
28                 }catch(Exception e){
29                         System.out.println(e+"\n<br/>\n");
30                 }finally{
31                         duration = (double)((System.nanoTime()-
                               startTime)/1000000000.0);
32                         DecimalFormat df = new DecimalFormat("#.##"
                               );
33                         System.out.println("<test id=\"time\" name
                               =\"Time\" value=\""+df.format(duration)
                               +"\" unit=\"s\"/>\n");
34                 }
35         }
36
37
38         /**
39         *@author bvl300
40         * */
41         public static void main(String argv[]){
42                 new TimeWrapper(argv);
43         }
44
45 }
```

Listing 10: *The TimeWrapper*

C The Ant-connector

```
1  import java.io.File;
2  import org.apache.tools.ant.ProjectHelper;
3  import org.apache.tools.ant.Project;
4  import org.apache.tools.ant.ProjectHelper;
5
6  public class Ant{
7          File buildFile;
8          Project project;
9          ProjectHelper projectHelper;
10         String dir;
11         String target;
12
13         public Ant(String dir,String target){
14                 this.dir= dir;
15                 this.target = target;
16                 buildFile = new File(dir+"/build.xml");
17                 project = new Project();
18                 projectHelper = ProjectHelper.getProjectHelper();
19         }
20
21         public void init(){;
22                 project.setUserProperty("antFile",buildFile.
                       getAbsolutePath());
23                 project.init();
24                 project.addReference("ant.projectHelper",
                       projectHelper);
25                 projectHelper.parse(project,buildFile);
26         }
27
28         public void run(){
29                 this.project.executeTarget(target);
30         }
31
32 }
```

Listing 11: *The Ant Class*

D Checkout Applications

```
1  private void checkoutProject(Node project) throws
       VersionControlException{
2          String url, type, projectName,user,pwd,command;
3          if (project.getNodeType() == Node.ELEMENT_NODE) {
4                  Element eElement = (Element) project;
5
6                  url = eElement.getElementsByTagName("location").
                       item(0).getTextContent();
7                  type = eElement.getElementsByTagName("type").item
                       (0).getTextContent();
8                  command = eElement.getElementsByTagName("command").
                       item(0).getTextContent();
9                  projectName = eElement.getElementsByTagName("name")
                       .item(0).getTextContent();
10                 user = eElement.getElementsByTagName("user").item
                       (0).getTextContent();
11                 pwd = eElement.getElementsByTagName("pwd").item(0).
                       getTextContent();
12                 if(type.equals("svn")){
13                         VersionControl svnRep = new SVN(basePath,
                               projectName, user,pwd,url,command);
14                         fetch(svnRep,projectName);
15                 }else if(type.equals("git")){
16                         VersionControl gitRep = new GitObject(
                               basePath,projectName,url,user,pwd);
17                         fetch(gitRep,projectName);
18                 }else{
19                         throw new VersionControlException("Version
                               control system not found");
20                 }
21         }
22 }
23
24
25 private void fetch(VersionControl vc,String projectName) throws
       VersionControlException{
26         long rev =-1;
27         vc.update();
28         try{
29                 rev = vc.getRev();
30         }catch(MethodNotSupportedException e){
31                 System.out.println(e.getMessage());
32         }
33         if(checkOldLog(projectName,rev)){
34                 vc.updateLog();
35         }
36 }
```

Listing 12: *Invoking the right update method*

E Manual for testing new software

When a user wants to test his software in the Codmon-VM environment he must execute the following steps.

First the projects to be tested should be added to the initialization file. This file is stored at "codmon/local/checkoutApplications/"

```
<projects>
        <project>
                <name>projectname</name>
                <location>http://www.sampleurl.com</location>
                <versionControl>
                        <type>svn</type>
                        <command>checkout</command>
                </versionControl>
                <run>true</run>
                <user>username</user>
                <pwd>pwd</pwd>
        </project>
</projects>
```

Listing 13: *Initialization file*

Second a sensor file, which describes the the test set, should be created. In this sensor file each sensor describes a test. The sensor file should be stored at "codmon/". In listing 14 The Time wrapper test is used to test the building time of the checkoutApplications module.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="libs/style.xml"?>

<sensors>
        <onoff>
                <!-- build checkout module-->
                <sensor id="build_ok" name="TestApps: Build
                    Checkout" cmd="java CODMON_HOME/codmon/wrappers
                    /classes/TimeWrapper ../../local/
                    checkoutApplications ant" scope="checkOut"
                    scheduled="false" graph="true" fatal="true" />
        </onoff>

        <graph>
        </graph>
</sensors>
```

Listing 14: *Sensor file*

The last thing the user has to do is adding a run-target to the build.xml of his software, which can be invoked by the Ant-connector.

```
1  <!--Run the program-->
2          <target name="run">
3                  <java  fork="true" classname="Checkout" classpath="
                       ${env.CLASSPATH}" output="out.txt">
4                      <classpath>
5                              <path location="${jar.dir}/software
                                   .jar"/>
6                      </classpath>
7                      <arg value="${arg0}" />
8                      <arg value="${arg1}" />
9                  </java>
10         </target>
```

Listing 15: *Ant run-target*