Multi-Environment Software Testing on the Grid

Alexandre Duarte, Gustavo Wagner, Francisco Brasileiro, Walfredo Cirne
Departamento de Sistemas e Computação
Universidade Federal de Campina Grande
Campina Grande, Brazil
+55 83 3310 1365
{alex, gustavo, fubica, walfredo}@dsc.ufcg.edu.br

ABSTRACT

We propose a solution to improve the confidence on the correctness of applications designed to be executed in heterogeneous environments, like a grid. Our solution is motivated by the observation that the traditional ways to qualify test processes are based on code coverage metrics. We believe that this approach is not adequate when dealing with applications that can (and do) fail when interacting with heterogeneous execution environments. Besides code coverage, tests must also cover possible environments. As a solution we propose the utilization of InGriD to describe and deploy test environments and GridUnit to coordinate and monitor the execution of test sets. By combining these two solutions we provide a cost effective way to introduce environmental coverage to our test suites, which is complementary and orthogonal to traditional code coverage metrics. As a case study, we have shown how our solution could be applied to help testing a grid application called MyPhotoGrid, which uses the grid to parallelize the generation of large photograph albums.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging – *Testing tools*, *Distributed debugging*.

General Terms

Verification, Performance, Reliability

Keywords

Distributed Testing, Unit Testing, Computational Grid, JUnit

ISSTA'06, July 17–20, 2006, Portland, Maine, USA.

1. INTRODUCTION

Software testing is a fundamental part of software development. Examples of disasters caused by poorly or untested software are widely available in the literature [1][15]. Therefore, there is an increasing demand for better support in the process of testing software.

The quality of a testing process is traditionally measured in terms of code coverage, i.e., the extent to which a set of test cases covers (or exercises) a program. Several widely accepted test coverage metrics have been used in the last few years, most of which are due to the Miller's seminal work [7].

Although software components may be exhaustively tested in the *development environment*, thus scoring well on the traditional test coverage metrics, we believe that the possibility of running tests in a variety of environments can improve the confidence on the correctness of the system under test. This is because the production environment can be very different from the development environment, being a possible cause of failure. In fact, due to the complexity of systems, nowadays each computing environment is unique.

This seems to be especially important when testing applications aimed to run on very heterogeneous environments like computational grids [10][12][20]. Two surveys we have conducted with computational grid users, although receiving a small number of responses, presented good anecdotal evidence for this observation. A questionnaire containing 5 questions, including "What are the most frequent kinds of faults you face when using a grid?", was made available on the Web and advertised in several grid discussion lists. Answers were received via the Web form as well as by e-mail. We have conducted two advertisement campaigns. The first one was made during April 2003 and resulted in 22 responses [18]. The second one was conducted during April 2005 and resulted in 13 responses [4]. From the data collected we can state that the situation regarding the type of faults that are more frequent in 2005 remains almost the same as in 2003. The main kinds of faults are related to the environment configuration. In 2003, a little more than 75% of the responses pointed this out, while in 2005 this was the main complaint of a

little more than 60% of the respondents. These results corroborate with [1], showing that configuration mistakes in software installation are the major reasons for computer system errors.

These are not good news since grid applications are supposed to be correctly executed on a highly heterogeneous and dynamic environment, encompassing several different hardware/software configurations, including different operating systems, and administrated by different support teams, each one having its own administrative policies. Based on these observations we believe that a solution to improve the confidence on the correctness of applications designed to run on heterogeneous environments is to test them on a representative set of the different production environments. With this practice we expect to achieve an environmental coverage, which is orthogonal and complementary to the traditional code coverage metrics.

Creating a set of dedicated test environments looks like an interesting approach to take. It allows the developers to completely specify and deploy the selected test environments but may result in high costs (time and money related). If we need to maintain the same test execution time testing the application on *n* different environments we will need to use at least *n* machines. If we do not want to use any additional machine to run the tests we will increase the execution time by a factor of at least *n* plus the time needed to create each test environment.

This approach has another drawback when applied to test grid applications since it does not allow us to test the application in face of the different use and administrative policies present on the grid, which can be, by itself, the cause of several problems.

So, how can one test an application on several different environments without paying the high costs to create the test environments and yet improve one's confidence that the application will be correctly executed even in the presence of hostile administrative policies? We believe that the answer to this question is to test the applications on a grid.

As shown in our previous work, the very high levels of parallelism provided by grids can speed up the execution of tests, increasing productivity and making the testing process of large software a less expensive endeavor [5]. We built a tool named GridUnit that is able to distribute and coordinate the execution of JUnit [8] test cases on a grid without any source code modification. Experiments conducted with this solution have showed a speed-up of almost 70 times in a grid with a hundred machines, reducing the duration of the test phase of a synthetic application from 24 hours to less than 30 minutes [5].

Now, we aim to improve our solution allowing it to fully explore the other key characteristic of grids, its huge heterogeneity, helping to improve the confidence on the correctness of the system under test. To achieve this objective we need to augment GridUnit with two new capabilities: a mechanism to specify and deploy, when necessary, the test environments and a way to define how such environments should be used to test a given application.

Our environment specification mechanism will allow us to overcome an important limitation of grids as a test platform: current grids provide only implicitly heterogeneous environments; there is no effective way to explicitly specify which environments should be used to test software.

Implicit heterogeneity can provide some benefits to software testing since it allows finding configurations that break the system under test. However, without knowing which environments are being used to test the software, one cannot correctly reproduce the test execution to figure out if a previously detected fault was effectively removed and if no faults were inserted since the last test session.

Providing a mechanism to explicitly specify desired grid test environments solves this problem only partially. The grid dynamicity provides a potentially huge number of different environments, there is no way to assure that all relevant environments to test a given application will be available on the grid during the test phase. Therefore, the solution to this important problem involves a mechanism to explicitly specify test environments and to deploy such environments if no suitable grid nodes were available at the test phase.

Note that environment refers to everything an application assumes about the resource where it is running, such as operating system, hardware platform, data, applications and libraries. So, a user may want to merge explicit and implicit heterogeneity by partially specifying the environment.

In this paper we discuss how we augmented GridUnit to fully explore the intrinsic characteristics of grids (parallelism and heterogeneity) to efficiently improve our confidence on the correctness of applications aimed to run on highly heterogeneous environments like grids.

The remaining of the paper is organized as follows. In Section 2 we present Incremental Grid Deployer (InGriD), our solution to specify and deploy execution environments on the grid. Next, in Section 3, we describe GridUnit, our distributed testing solution, and explain how we augmented it to use the InGriD's execution environments to test software. Following, in Section 4, we discuss some experiments we conducted with our solution, showing how InGriD and GridUnit can be used to effectively test grid applications on a production grid. Section 5 presents the related work and Section 6 concludes the paper summarizing our discussion and sketching future work proposals.

2. SPECIFYING AND DEPLOYING TEST ENVIRONMENTS ON THE GRID

The main goal of InGriD is to provide an easy way to deploy environments on a grid. For this purpose, there are three main tasks: i) description of the environments in terms of applications, configurations and data; ii) deploying the environments in a set of grid resources and, iii) accessing the applications and files deployed. To achieve these goals, we have developed a system using SmartFrog to make easier the deployment of environments to grid users [16]. SmartFrog is a framework formed by a language and a runtime environment developed by HP Labs Bristol. It was developed taking into account that configuration mistakes in software installation are the major reasons for computer system errors [1]. Thus, it was designed to automatically manage and deploy components across multiple machines. It consists of a language and a runtime environment that supports application deployment. Although SmartFrog is a great tool for system administrators, it is not integrated with any grid environment.

InGriD aims to extend SmartFrog in order to deal better with grid applications and to provide some predefined components that otherwise would require some additional efforts from a SmartFrog user.

One of the characteristics of grid applications is that they need to react to environment changes. So, we have developed components to download the applications and data incrementally, installing each part of the application in different moments.

2.1 InGrid Architecture

Figure 1 shows the overall InGriD architecture and its relations with Environment Descriptions. These descriptions include all the applications and files that the users need to run their jobs in the grid. From a tester's point of view, the description contains all applications and data required by the tests.

The first module is the *Installer*, which contains the rules used to install applications and download data for the specific environment. The InGriD user defines the original location of applications and data and their location on the grid machines. Note that the environment location in different machines may be different, since InGriD exports a global name for each application and data that can be used in the job description. This is possible due to the *Accesser* module discussed later.

The second module is the *Configurator*, which is responsible for configuring all the parameters that the applications of an environment need. For example, a user may specify to alter an application port number from 4998 (the default) to 6000 in a configuration file. In this case, the user does not need to open and alter the file manually. Instead, the InGriD performs this task through the deployment description. This reduces the complexity of dealing with many configuration files in different contexts.

The third module is the *Updater*, responsible for the incremental deployment rules of InGriD. The user defines a frequency or a specific time to update each application and data. After the specified time/times, InGriD removes the old version of the application(s) and installs the new one. For data, InGriD may also just download the new one without removing the existing data, since some system should need the old one.

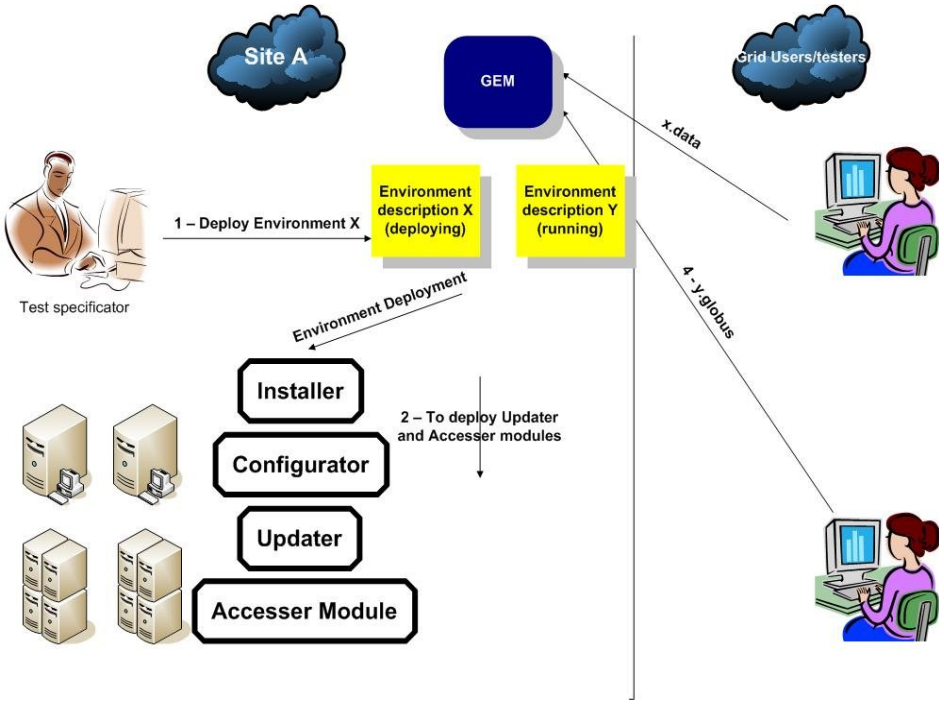

Figure 1: InGriD High Level Architecture

The fourth module is the *Accesser*, responsible for storing mappings from global to local names (the location of the environments in each grid machine). A global name is a name that gives access to applications and data and that is valid in all machines that compose the grid. For example, an environment called Globus 4.0.1, the user exports the Globus job service (MMJFS) location as "jobservice" and the grid data as "data". A tester may use "Globus.jobservice" to refer to the Globus location and "Globus.data" to refer to the specific data in a job description. In this example, "Globus.jobservice" and "Globus.data" are called global names, and point to the local names in each grid machine. This mapping is very

important to grid users, since they usually do not know the specific location of applications and data at each grid resource.

The fifth and last module is the Grid Environment Manager (GEM), which is responsible for all the environments deployed in a site. It contains information such as environment name and all machines that host it. It is useful to schedulers because they may use only machines that match the requirements of a particular job. Testers can use it as a matcher for their test environments.

2.2 Environment Specification

In order to describe an environment, the user needs to configure the components of the InGriD architecture. For this purpose, InGriD provides a set of predefined components that the user extends with the specific characteristics of the required environment. Figure 2 shows an example of an environment specification. In this example the user describes the location of the GEM, responsible for storing information about all the deployed environments. After that, the user defines all the components of the InGriD architecture. Note that in this example we use the word **extends** for each component. This means that, for example, *installer* component is an extension of *GlobusInstaller*. This is possible because the SmartFrog language is object oriented. Each defined component such as *GlobusInstaller* and *GlobusConfigurator* extends predefined components developed in InGriD.

```
environment extends EnvironmentTemplate {
    name "globusEnvironment";

    GEMLocation ATTRIB GlobusConfigurator:GEMLocation;

    environmentComponents extends LAZY Sequence {
        installer extends GlobusInstaller;
        updater extends GlobusUpdater;
        acesser extends GlobusAccesser;
        configurator extends GlobusConfigurator;
    }
}
sfConfig extends environment;
```

Figure 2: An example of environment description.

3. DISTRIBUTED TESTING WITH GRIDUNIT

GridUnit [5] is a grid-based testing execution solution able to distribute the execution of JUnit [8] test suites in a grid with minimum user intervention. GridUnit is an open-source project, licensed under the GNU LGPL license terms, and can be freely downloaded from http://gridunit.sourceforge.net. GridUnit was developed on top of the OurGrid solution [20], although it can be easily adapted to use other grid flavors, such as Globus [19], for example.

OurGrid is an open, free-to-join, cooperative grid in which users donate their idle computational resources in exchange for accessing other users' idle resources when needed. It uses the Network of Favors [14], a peer-to-peer technology that makes it in each user's best interest to collaborate with the system by donating their idle resources. OurGrid is in production since December 2004 and now encompasses around 300 machines in 20 sites distributed over Brazil and Europe. A fresh snapshot of the running system can be seen at http://status.ourgrid.org.

3.1 GridUnit Architecture

GridUnit can be seen as an intermediary agent between the user/developer, who wants to run a JUnit test suite, and the computational resources needed to run the tests, in this case, the OurGrid. This brokering process involves four main tasks: i) creation of a job description from the JUnit TestSuite; ii) scheduling of the job's tasks for execution on the grid; iii) monitoring of the execution; and iv) presenting the results to the user.

Figure 3 summarizes the GridUnit high level architecture showing how it is constructed on top of OurGrid and how it uses JUnit components.

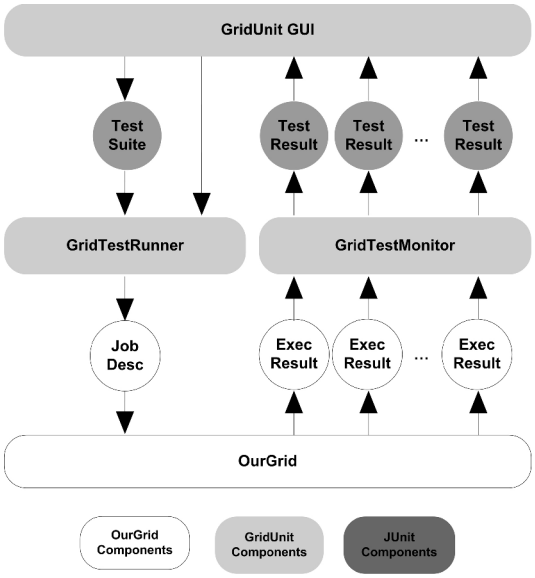

Figure 3: GridUnit High Level Architecture

The GridTestRunner is responsible for the creation of the job description from the TestSuite and for the scheduling of the job for execution on OurGrid. A JUnit TestSuite specifies a set of independent TestCases that should be executed in no pre-determined order. An OurGrid job description specifies a parallel application composed of independent tasks in which each task may be executed in any order and at any time. We refer to this kind of parallel application as Bag-of-Task. Due to the similarity of concepts, (a JUnit TestSuite is already a Bag-of-Tasks or a Bag-of-Tests) the conversion from a JUnit TestSuite to an OurGrid job description is straightforward. We simply create a job describing the TestSuite and a task for each TestCase in the TestSuite. After the conversion, the GridTestRunner schedules the execution

of the job on the grid using the WQR [6] scheduling heuristic provided by OurGrid.

Another component of GridUnit, the GridTestMonitor, is responsible for monitoring the execution of the tests on OurGrid and for converting the data resulted from the execution into a JUnit TestResult. This is not a trivial task due to the nature of the grid. There is no predefined order in the execution of the tests, and therefore, any test can end at anytime. Moreover, the new set of failures that can occur due to grid faults further complicates this task. So, the GridTestMonitor must be able to distinguish between test failures and grid failures in order to provide reliable results to the user. This is achieved by considering as test failures only the exceptions raised by the JUnit assertion mechanism. All the remaining exceptions are considered as unexpected errors and indicate that the test could be executed. GridUnit provides a detailed report about these errors so the user can identify it they are due to some defect in the system under test or are due to a problem in the grid middleware.

The last component of GridUnit is its graphical user interface, shown in Figure 4. This interface presents to the user all information about the execution of tests as if they were being executed in the local machine. In fact, to the GUI, it is the same for remote and local execution.

Figure 4 shows an example of the execution of a test suite on OurGrid. In this example, the test suite is composed of 288 test cases. The status bar, in the lower corner of the window, shows that at that moment 75% (or 217) of the 288 test cases were executed and seven of them have failed due to unsatisfied assertions. The progress bar is red because there are test cases that failed due to these unsatisfied assertions. It would be gray if some test case failed due to unanticipated errors (e.g. exceptions not raised by the JUnit assertion mechanism) and green if no test case failed at all.

The tree at the left corner represents the test case hierarchy and shows the status of the execution of each test case using tiny colored icons. It provides an overview of the execution process. The GridUnit GUI provides the same amount of detail about the execution of a test case as the traditional JUnit test runners along with some additional information about the environment where the tests were executed.

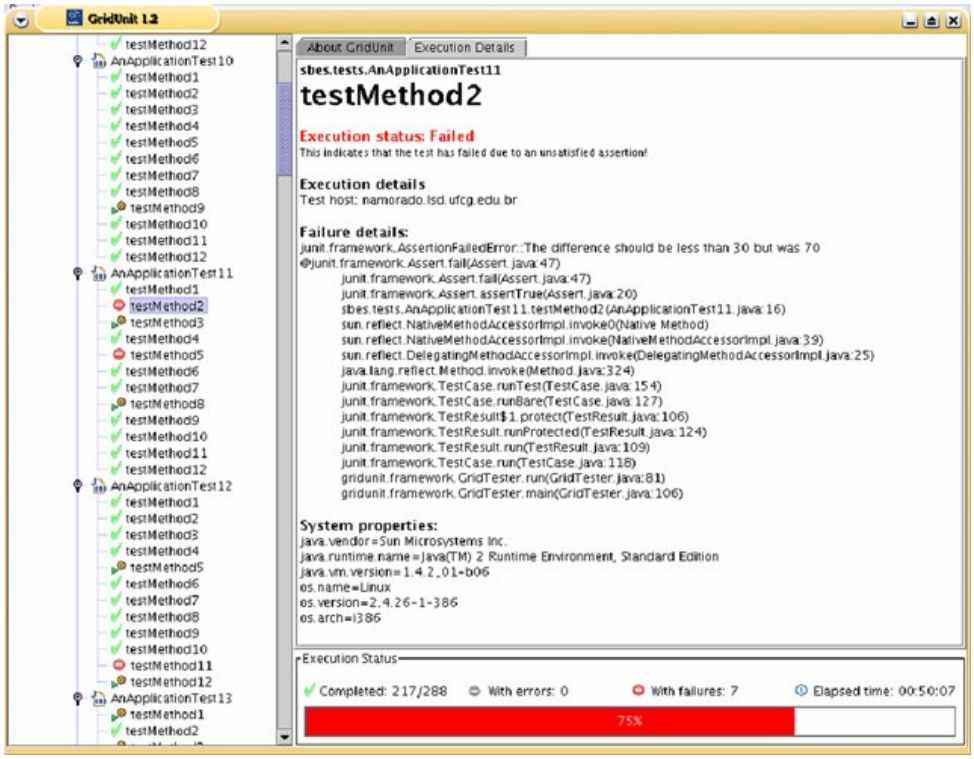

Figure 4: GridUnit Graphical User Interface

3.2 Main Features

The main features of GridUnit are those pointed by Kapfhammer [11] as important aspects that a test distribution tool must consider in order to improve the cost effectiveness of the testing process:

Transparent and Automatic Distribution: GridUnit considers each JUnit test as an independent task, scheduling its execution on the grid without any user intervention. Moreover, GridUnit does not require any modification in the application source code.

Test Case Contamination Avoidance: Each test is executed using the resource virtualization provided by OurGrid, preventing that the execution of a test alters the normal outcome of other tests.

Test Load Distribution: The job scheduler provided by OurGrid achieves load distribution by allocating each test for execution in the first available grid machine. So, each grid machine receives a slice of the work proportional to its computational power. To diminish the chance that a slow machine slows down the overall computation the scheduler replicates the work on a small number of machines.

Test Suite Integrity: The default JUnit test runner runs each unit test as an independent task. For each test, it creates an instance of the TestCase class, calls the setUp() method, calls the testMethod(), calls the tearDown() method and then destroys the instance. GridUnit reproduces the

same behavior with the difference that each test is potentially executed in a different machine on the grid.

Test Execution Control: GridUnit graphical user interface provides controls to start and stop the execution of the tests of a given test suite. It also monitors the execution of the tests and presents the result of the execution of each test as soon as it is available.

3.3 Test Session Specification

After describing and deploying test environments using InGriD, the tester must decide which of the available environments will be used to test an application. To accommodate this requirement we have augmented GridUnit to provide support to use a Test Session Specification Language (TSSL) based on XML. A TestSession is composed of a list of TestEnvironments, followed by a list of TestSuites, followed by a list of TestCases.

A TestEnvironment is the name of an InGriD environment where each of the TestSuites and TestCases described in the TestSession must be executed. A TestSuite is a reference to a JUnit test suite, where the id parameter represents the fully qualified test suite class name. It is followed by an optional list of additional test environments where all tests of the TestSuite must be executed. A TestCase is described in a similar way, but it refers to a single JUnit test case.

Figure 5 shows an example of a TSSL file describing a simple TestSession. In this example, we illustrate the cumulative nature of our description. We have defined a test session named SimpleTestSession, a test suite named SimpleTestSuite and two test cases named TC1 and TC2. We have also referenced four execution environments previously defined using InGriD (ENV_A, ENV_B, ENV_C and ENV_D).

```
<TestSession id="SimpleTestSession">
    <TestEnvironment>Env_A</TestEnvironment>
    <TestSuite id="SimpleTestSuite">
        <TestEnvironment>Env_B</TestEnvironment>
        <TestCase id="TC1">
            <TestEnvironment>Env_C</TestEnvironment>
        </TestCase>
    </TestSuite>
    <TestCase id="TC2">
        <TestEnvironment>Env_D</TestEnvironment>
    </TestCase>
</TestSession>
```

Figure 5: Simple TSSL file

With this simple description all tests located in SimpleTestSession will be executed on the environment ENV_A, all tests located inside SimpleTestSuite will be executed on the environments ENV_A and ENV_B, the test case TC1 will be executed on the environments ENV_A, ENV_B and ENV_C and the test case TC2 will be executed on the environments ENV_A and ENV_D.

We believe that this description is very simple and powerful enough to allow one to specify environment dependencies for any test case. In the next section we will put all these things together and present a real case study.

4. TESTING MYPHOTOGRID

The introduction of high resolution digital cameras placed regular users as the producers of enormous amounts of data. For example, one of the authors of this paper produced 35 GB of digital photos in the last 6 months. An interesting alternative would be to use the grid to create digital albums from these photos.

MyPhotoGrid creates a web album from a large set of digital images. A possible kind of digital albums is composed of three images (a thumbnail, a regular size image and a big one) generated from the original picture. The album index contains a list of thumbnails that are hyperlinks to the regular image, which links to the big image.

The process of album creation is time consuming because processing thousands of images can take a long time. Photo editing is independent from each other, which makes it an embarrassingly parallel application or, simply, a Bag-of-Tasks application. The idea is to use a grid to make photo editing parallel. When all photos processing results are available, the index will be built locally.

MyPhotoGrid has two main modules, one that runs in each remote grid node and is responsible for resizing and sharpening the images and create appropriate links for them and another one that runs in the local machine and is responsible for combining the results and producing the full HTML album.

It is interesting to test the remote part of MyPhotoGrid using different hardware/software configurations in order to improve the confidence in its correctness. So, it is a good target for our grid testing approach.

Analyzing the current OurGrid deployment, we found that there are two different Java virtual machine implementations installed in the grid nodes: Sun JVM 5.0 and JRockit 5.0. Also, our previous analysis has shown that there are two main hardware architectures available: Intel Pentium 4 and Intel Itanium 2, and we want to test the software using these two hardware platforms. Based on these observations we have four testing scenarios: Sun JVM + Pentium 4, Sun JVM + Itanium 2, JRockit + Pentium 4 and JRockit + Itanium 2.

Our solution requires three steps in order to achieve this objective:

1. First, we need to write the appropriate InGriD environment specifications, as shown in Figure 6. There, we can see two component descriptions. The first one,

environment, describes the components that InGriD needs to deploy an application. *Installer*, *Configurator* and *Accesser* are extension of template components. So, the InGriD user needs just to describe specific configuration, as we can see on the SunJVMInstaller component. The tester describes where is the tarball of the JVM, where InGriD will install the JVM and the steps to install the JVM (installScript in the figure). The tester needs also to describe the *configurator* and *accesser* modules. Although it is possible for the InGriD user to define the *Updater* component, we had not defined it here since testers do not need to update the JVM versions. With this description, the tester sends it to SmartFrog in all test's machines and the environment will be deployed.

2. Then, we need to instruct GridUnit to run the MyPhotoGrid test cases using the four test scenarios, as shown in Figure 7. As we can see, this description is fairly simple. We create a test session containing one single test suite (myphotogrid.tests.AllTests), and then we specify that our test suite should be executed using four different test environments (Sun_P4, Sun_Itanium, JRockit_P4, JRockit_Itanium).

3. Now, after the two configurations steps above, we use GridUnit to coordinate and monitor the execution of the tests. It is as simple as using the regular JUnit test runners.

```
environment  extends EnvironmentTemplate {
    name "Sun_P4";

    environmentsLocation ATTRIB JVMConfigurator:environmentsLocation;

    environmentComponents extends Sequence {
        installer extends SunJVMInstaller;
        configurator extends JVMConfigurator;
        accesser extends JVMAccesser;
    }
}

sfConfig extends environment;
SunJVMInstaller extends GenericInstaller {
    name "sunJVMIntaller";

    protocol "http://";
    webServerHost "lambari.lsd.ufcg.edu.br";
    webServerPort "8080";
    tarLocation "/webdav/";

    url (protocol ++ webServerHost ++ ":" ++ webServerPort ++ tarLocation);
    installLocation ATTRIB JVMConfigurator:basedir;

    file "j2sdk1.5.tar.gz";

    home (installLocation ++ "j2sdk1.5");

    installScript [ (cd ++ home), ("mkdir -p " ++ JVMConfiguration:basedir) ];
}
```

Figure 6 - MyPhotoGrid Environment Description

5. RELATED WORK

Parallel testing has been used for a while to test hardware systems but, to the best of our knowledge, Starkloff [9] was the first to advocate the application of parallel and distributed technologies to testing of computer software systems. Starkloff summarizes some general advantages of this approach and briefly reports on a multithreaded tester that can be used to run several independent test sequences in parallel. Kapfhammer [11] describes the conceptual foundation, design, and implementation of Joshua, a tool that distributes the execution of regression test suites for Java-based software systems. Lastovetsky [2] relates the experience of the use of parallel computing technologies to accelerate the testing of a complex distributed programming system. The design and implementation of the parallel testing system are described and some experimental results show that the system is efficient, speeding up the test execution

```
<TestSession  id="MyPhotoGridTestSession">
  <TestSuite id="myphotogrid.tests.AllTests">
    <TestEnvironment>Sun_P4</TestEnvironment>
    <TestEnvironment>Sun_Itanium</TestEnvironment>
    <TestEnvironment>JRockit_P4</TestEnvironment>
    <TestEnvironment>JRockit_Itanium</TestEnvironment>
  </TestSuite>
</TestSession>
```

Figure 7: MyPhotoGrid Test Session Specification

All these works are focused only on speeding up the execution of the tests. They are of no help when one needs to force the execution of tests using different hardware/software configurations in order to improve the confidence on the correctness of the system under test.

The Skoll project [3] proposes a quality assurance process that aims to use distributed computational resource around the world to test software in a variety of configuration scenarios. It provides a mechanism to specify test scenarios (configuration model) but it lacks the support to deploy unavailable scenarios. Additionally, Skoll users need to have direct access to all computational resources they may need to execute the tests since it does not relies on any resource sharing mechanism, like a grid.

There are a lot of works regarding system deployment. LCFG [15] is a deployment tool that is typically used to install Linux environments. It is used in environments that change constantly their configurations. LCFG is formed by a set of machine profiles, which are descriptions of how to configure and install applications. LCFG has a central server that manages all the profiles and, when profile description changes, all the machines that contain those profiles are updated. The problem of using LCFG in grids is its centralized design, which does not match with the decentralized characteristic of grids.

As far as we know, there is no previous work targeted to improve the software testability by exploring heterogeneous execution environments or to use the huge environment heterogeneity provide by grids to provide an environmental test coverage.

6. CONCLUSIONS AND FUTURE WORK PROPOSALS

We have presented a proposal of a solution to improve the confidence on the correctness of applications designed to be executed in heterogeneous environments.

Our solution is motivated by the observation that the traditional ways to qualify test processes are based on code coverage metrics. We believe that this approach is not enough when dealing with applications that can (and do) fail when interacting with heterogeneous environments.

We have good anecdotal evidence [18][4] corroborating the results presented in [1], that the major causes of failures of grid applications are due to environment configuration errors. These kinds of errors are difficult to test and discover during development time. Thus, even if the testing processes score well on the traditional test coverage metrics, it does not necessarily mean that the software is well tested since configuration related errors only show when the system interacts with the production environment.

We propose the utilization of InGriD to describe and deploy test environments and GridUnit to coordinate and monitor the execution of test sets. By combining these two solutions we provide a way to introduce an environmental coverage metric to our test sets, which is complementary and orthogonal to traditional test coverage metrics.

We have shown how our solution could be applied to help test a grid application called MyPhotoGrid, which uses the grid to parallelize the generation of large photograph albums.

The present work is aimed mainly to deal with unit tests which can execute in a single machine. Our current work aims at building a complete framework to allow the distributed execution of system tests, involving mechanisms to describe distributed tests, interact with distributed components and detect distributed termination, which is important to determine when the system reach a state were assertions can be made over the final results of the test.

ACKNOWLEDGMENTS

Authors would also like to thank the financial support from CNPq/Brazil and CAPES/Brazil, as well as the help provided by the SmartFrog team. Thanks also to Katia Saikoski for her suggestions and comments. Thanks to Loreno de Oliveira for developing MyPhotoGrid.

REFERENCE

[1] A. Brown, D. A. Patterson, To Err is Human, First Workshop on Evaluating and Architecting System Dependability (EASY '01)

[2] A. Lastovetsky, Parallel Testing of Distributed Software, Information and Software Technology 47(10), pp.657-662, Elsevier, 2005

[3] A. Memon, A. Porter, C. Yilmaz, A. Nagarajan, D. C. Schmidt, and B. Natarajan, Skoll: Distributed Continuous Quality Assurance, in Proceedings of the 26th IEEE/ACM International Conference on Software Engineering (ICSE), Edinburgh, Scotland, May 2004

[4] A. N. Duarte, F. V. Brasileiro, W. Cirne, J. S. Alencar Filho, Collaborative Fault Diagnosis in Grids through Automated Tests, in: the Proceedings of the 20th IEEE International Conference on Advanced Information Networking and Applications Vienna, Austria, April 2006.

[5] A. N. Duarte, W. Cirne, F. V. Brasileiro and P. D. L. Machado, GridUnit: Software Testing on the Grid, in: Proceeding of the 28th ACM/IEEE International Conference on Software Engineering, Shanghai, China, May 2006.

[6] D. Paranhos, W. Cirne and F. Brasileiro, Trading Information for Cycles: Using Replication to Schedule Bag of Tasks Applications on Computational Grids, in: Proceedings of the Euro-Par 2003: International Conference on Parallel and Distributed Computing, 2003.

[7] E. F. Miller, Program Testing Techniques, COMPSAC'77 IEEE Computer Society, 1977.

[8] E. Gamma, K. Beck. JUnit: A cook's tour. Java Report, 4(5):27-38, May 1999

[9] E. Starkloff, Designing a Parallel, Distributed Test System, Proceedings of the IEEE AUTOTESTCON, 2000.

[10] F. Berman, G. Fox, A. J. G. Hey, Grid Computing: Making the Global Infrastructure a Reality, John Wiley & Sons Inc., 2003.

[11] G. M. Kapfhammer, Automatically and Transparently Distributing the Execution of Regression Test Suites, in: Proceedings of the 18th International Conference on Testing Computer Software, 2001.

[12] I. Foster and C. Kesselman. The Grid: Blueprint for a New Computing Infrastructure, 2nd Ed, Morgan Kaufmann, 2004.

[13] M. Ben-Ari. The bug that destroyed a rocket. Journal of Computer Science Education, 13(2):15–16, 1999.

[14] N. Andrade, F. Brasileiro, W. Cirne, M. Mowbray, Discouraging Free-riding in a Peer-to-Peer CPU-

Sharing Grid. Proceedings of the 13th High Performance Distributed Computing Symposium, June 2004.

[15] P. Anderson, A. Scobie, LCFG: The Next Generation. UKUUG Winter Conference, 2002

[16] P. Anderson, P. Goldsack and J. Paterson, SmartFrog meets LCFG: Autonomous Reconfiguration with Central Policy Control, in: Proceedings of the 2002 Large Installations Systems Administration, 2002.

[17] P. Mellor, CAD: Computer-Aided Disaster, High. Integr. Syst., 1(2):101-156, 1994.

[18] R. Medeiros, W. Cirne, F. Brasileiro and J. Sauvé, Faults in grids: why are they so bad and what can be done about it?, in: Proceedings of the Fourth International Workshop on Grid Computing, 2003, 18-24.

[19] The Globus Alliance, Globus. http://www.globus.org, 2005.

[20] W. Cirne, F. Brasileiro, N. Andrade, R. Santos, A. Andrade, R. Novaes and M. Mowbray, Labs of the World, Unite!!!, Accepted for publication by JoGC. http://www.ourgrid.org/