Improving Regression Testing through Modified Ant Colony Algorithm on a Dependency Injected Test Pattern

Dr.K.Vivekanandan
Reader, Department of School of Management,
Bharathiar University,
Coimbatore - 641 046, Tamil Nadu, India

G.Keerthi Lakshmi
Research Scholar, Department of Computer Science,
Bharathiar University,
Coimbatore - 641 046, Tamil Nadu, India

ABSTRACT
Performing regression testing on a pre production environment is often viewed by software practitioners as a daunting task since often the test execution shall by-pass the stipulated downtime or the test coverage would be non linear.

Choosing the exact test cases to match this type of complexity not only needs prior knowledge of the system, but also a right use of calculations to set the goals right.

On systems that are just entering the production environment after getting promoted from the staging phase, trade-offs are often needed to between time and the test coverage to ensure the maximum test cases are covered within the stipulated time.

There arises a need to refine the test cases to accommodate the maximum test coverage it makes within the stipulated period of time since at most of the times, the most important test cases are often not deemed to qualify under the sanity test suite and any bugs that creped in them would go undetected until it is found out by the actual user at firsthand. Hence

An attempt has been made in the paper to layout a testing framework to address the process of improving the regression suite by adopting a modified version of the Ant Colony Algorithm over and thus dynamically injecting dependency over the best route encompassed by the ant colony.

Keywords
Regression testing, Software Testing, Ant colony, Dependency Injection

1. INTRODUCTION
Testing out the software delivery artifacts which is automated in the latest cutting edge technologies,, say the hourly agile builds in a real-time system and taking out the best build that is deemed to get qualified as a stable build is a tedious and time bound task, where the heuristics are to be applied at tandem, which constitutes the most effective test cases that could pave way to catch the high priority bugs**.**

On the other hand, dealing with high density of data transmissions that span least in megabytes is nowadays increasing due to the advent of more people connected on the Go. With the advent of smart phones that produce real time statistics like maps and weather that involves transmitting to the near broadcasting point to avoid delay in response, the applications that work beneath these phones are to be intelligent and cloud enabled that provide the services involving distributed spatial data. Testing these systems without dealing effective way of dealing with the high volumes of data is inevitable.

Each data transmission task consumes a finite bandwidth of network resources and time, and empowering such a request for testing would normally lead in failure of accomplishment of test objectives, as they are time and resource bound.

Most end point services use the load balancer (Fig 1) which decides on the appropriate cloud point (worker) to be called on these situations. The database nodes are connected in real-time with a snapshot replication for maintaining data integrity.

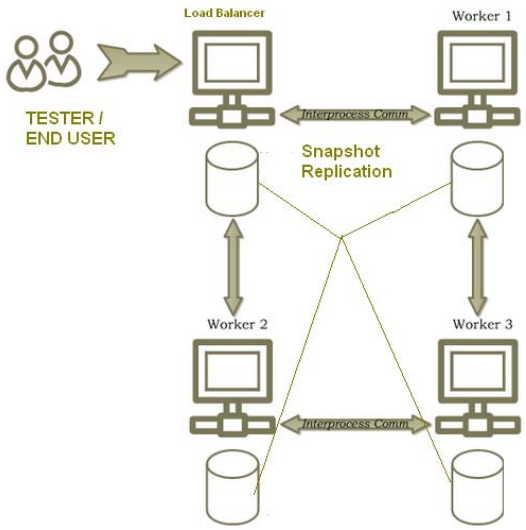

Fig 1 A Typical Cloud Hosting Architectural Environment

On Most cases, the service end point listens on the advertised point through the publicized protocol, which usually vary based on the domain cross functional requirements. There is also a possibility of multiple bindings running at a specified service end point to cater to various type of clients.

The end point request is finally tunneled to a single stream of processing thread despite presence of multiple bindings where the service is responded with the appropriate high density bandwidth data information.

1.1 Choosing the right testing strategy

Testing an application that is seeded at multiple locations like cloud service that carries high density of information across involves testing the individual end points by contacting the load balancers through seeding the cloud at every point of service end point and then awaiting for the response and validating the actual results against the expected results.

if an endpoint fails out the test strategy can be changed by referring to the next available service end point without sacrificing the intermediate test results arrived so far at. This being the advantage of cloud environment over the conventional distributed environment is that during testing a business case.

The test environment need to take the alternate course of action upon detecting the tip of failure and need not wait for the complete failure to happen before declaring the test case as failure. The test environment must also be continuously learning from its mistakes and able to apply the learning towards the reduction of the testing time. Also, in real world situations the test service cannot take on the complete production bandwidth for testing, and also testing the production environment needs to be quick and resource conscious.

2. CHOOSING THE BEST COVERAGE PATH

All the test cases regression testing in the pre-production environment boils down to a NP-Hard problem which involves finding the least-cost cyclic route through all nodes of a weighted graph.

The first step of the problem is to apply ACO across computing the best order of traversal from the starting test case so that the maximum coverage is obtained.

In this case, the test cases are not executed but are just estimated for the best possible travel paths.

Dependency injection is defined as a design pattern evident in object-oriented computer programming whose primary purpose is to improve testability of the target system and thus simplify the deployment of components in large software systems.

Allowing the option to choose among multiple implementations of a provided interface at runtime, or via configuration files, which would be picked up at runtime is the primary purpose of the dependency injection pattern. The pattern is mostly useful in providing the fake or mock test implementations of complex components when adopting the process of testings.

Unit testing of components in large software systems is difficult, because components under test often require the presence of a substantial amount of infrastructure and set up in order to operate at all. Dependency injection simplifies the process of bringing up a working instance of an isolated component for testing. Because components declare their dependencies, a test can automatically bring up only those dependent components required to perform testing.

More importantly, injectors can be configured to swap in simplified "mock" implementations of dependent components when testing -- the idea being that the component under test can be tested in isolation as long as the substituted dependent components implement the contract of the dependent interface sufficiently to perform the unit test in question.

Using dependency injection, the components that provide access to the online service and back-end databases could be replaced altogether with a test implementation of the dependency interface contracts that provide just enough behavior to perform tests on the component under test.

An attempt is made in this paper to apply the concept of Partial Dependency Injection over the testing of the distributed cloud environment involving the spatial data.

A simple cloud based Ecommerce solution that sells out digitized media recording on the fly to the customers is considered for testing. The application is resident over various cloud endpoints as against a server farm technology the most of the ecommerce web solutions offer currently.

The results are compared with the testing of cloud by using the load balancer method that is in operation currently.

The results clearly indicate the dependency injection having a clear edge over the conventional testing method, with the advantages of mocking out an endpoint operation with the existing results of the service provided the service is stagnant in maneuverability.

3. THE EXISTING SYSTEM

The current system is an ecommerce portal which sells all sorts of digitized portfolio – like providing real time map service, streaming out a live –in concert over the web upon fulfillment of payment to selected users, selling other ecommerce products like normal websites offer including books, CD's etc..,

We studied the existing system by laying out the complete architecture of the system – The current system is cloud enabled by providing various service endpoints at geo-locations. All requests to the service reach the load balancer, which actually decides the service point which is capable of servicing the load based on the location of the request. The subsequent requests are directly carried out directly by the participating endpoint.

The endpoint intimates the load balancer about success or failure of the operation on the following occasions:

a. When the process is successfully completed, releasing the DMA token it shared with the client, granted by the load balancer.
b. When the process could not be completed, encountering a catastrophic failure. The load balancer senses the failure and takes the alternate course of action by determining the next service end point and contacting it from the last point of failure. In this occasion, the client is totally unaware of this switch made by the load balancer and that is the advantage of hosting over a cloud.
c. Over periodic intervals, the end point intimates the load balancer about the progressing point as a heart beat signal.

We simulated the behavior of the existing system after concluding the architecture and documented the various artifices that are needed to test the system. We choose the streaming model to virtualizes, as it was unusually different from the normal ecommerce business, which would pay way for effective test case generation and corner case identifications.

The Ecommerce cloud had 6 sources of generation across the geographic locations, one hosted in North America, one in Brazil, next 2 in France and Ireland. In APAC, it was hosted in India and japan, featuring a complete geo-clustered cloud. Each of these sources was supplied with fail-over support, 24/7 uninterrupted access and checking of vital information over the dashboard which is accessible in various channels like over web, smart phone light weight application and over the access of remote desktop.

The load balancer was registered with the DNS (Domain name server) over an IP address which would be known to the callers

3.1 The Testing Approach

Upon careful analysis of the system and analyzing the entire serviceable endpoint and the environment, the testing objectives were not to be myriad that the normal web testing methods which could have been implemented using a load tester software or the selenium.

The intention of the testing is below:

a. To cover the maximum coverage criteria
b. To attain (a) within the best available time
c. To use the limited set of resources as the testing directly affects the enduser bandwidth since we are testing on the production environment.

4. EXISTING APPROACH

The story board technique was used in arriving at the test cases and a complete suite contained of the following test case:

a. A customer logs in to the system
b. Customer checks the various live-in concerts/ live feed copyrighted movies available for the day
c. Customer chooses a concert and a sample of the clipping is streamed to the customer from the site for a stipulated duration.
d. Customer decides to go for the concert and completes the payment processing formalities.
e. Customer is given a direct link to view the concert / copyrighted movie for a particular duration for the generated IP address.

The normal testing approach is not to test the QoS parameters, but to effectively test this live scenario in the production environment without stealing the devoted bandwidth allocated to the real customers.

A total of 6 test cases was framed covering the above mentioned scenario and was executed over 4 users who were from India. The offset of the other users wasn't taken into account as we were measuring the test execution times for Indian users only.

The user 2 was introduced to the system 10 seconds after the other users have logged in to deal with dead-lock starvation handling and the results to execute the test cases are tabulated below:

Table: 1 Output Metrics of the Current System

(In Seconds)

	Tc1	Tc2	Tc3	Tc4	Tc 5	Tc6	Total (mm:ss)
keerthi	11	34	14	443	3	8	08:55
Vivek*	12	21	23	1027**	3	9	20:04
User3	17	22	8	385	2	9	07:38
User 4	14	28	44**	457	3	9	09:25
Total time:							20:04

* Started at 00:10

** Retip of the service end point to the LB

Table 2: Code/ Statement Coverage

	Tc1	Tc2	Tc3	Tc4	Tc5	Tc6
keerthi	100	90	-	100	-	100
Vivek	100	75	-	100	-	100
User3	100	90	-	100	-	100
User 4	100	75	-	100	-	100

Tc3 and Tc5 were not put in for code coverage as they had streaming and a direct link respectively, where there was no code to cover.

While we ran the entire scenario 3 times, the results above are for the iteration 2. The system retipped to the load balancer twice – first time at 00:52, when all the four users were active at executing Tc3 that the APAC cloud gave up when the user4 accessed the system, causing the delayed execution of Tc3 for user 4, which is greater than average of 23+14+8 /3 = 15. The retipping happened at 10^{th} second after User4 had pipelined to the machine at test case Tc3, causing another 30 seconds to fetch from the next APAC server at Japan.

The second retipping happened at around 01:54 when the Tc4 was executing, which was to provide a sample clip of the streaming, which collapsed the entire end point from recovery and brought in the fail-over protection node end point to action. Meanwhile , the load balancer detected the heart beat was missing from the end point and it routed the request to the next serviceable endpoint. When the service end point was restored back by the failover node cluster, the heart beat had resumed and the load balancer again started streaming from the same end point which caused the failure. The end user experience was the streaming was continuous with good QoS, but with 3 times resuming the same clips as a result of which the total execution time creped up above 20 minutes, featuring the segment highest.

5. PROBLEMS IN THE EXISTING APPROACH

Though the current approach is versatile in terms of completeness of the system, it would be skeptical to deal with the current approach in terms of execution times when it comes to deal with scalability. A few of the drawbacks are as listed below:

A. The test case execution engine always relies on load balancer to post the request and get the response. Those this may sound fairly simple, on occasions when the retipping happens, the execution engine gets no clue and had to wait on the load balancer to complete the response.

B. On the systems where the test execution is limited to time, it is not possible to tune the system to attribute to the complete coverage. For Example, the tests cannot be flexed if a definite time interval is given for execution.

C. There is no option to bye-pass the scenario and continue the testing. For e..g.: if implementation for Tc4 is kaput, the load balancer has no alternative course of action than to return a failure to the caller, thus not proceeding with the testing. Tc4 is a sample renderer and most users may opt to bye-pass it and continue their purchase for the live-in concert.

6. THE PROPOSED SOLUTION

The following metrics are derived and are used for computations.

$$\text{Bug Contribution Ratio (BCR)} = \frac{\sum \text{Bugs detected on all the test-runs}}{\text{Number of Times the test case has run}}$$

$$\text{Average Contribution Ratio (ACR)} = \frac{\sum \text{Sum of total bugs revealed on all the test –runs of all test cases}}{\text{Number of times of full test suite execution}}$$

The idea is to analyze the records of the existing test runs and compute the decision of introducing a dependency injection container called “Mock”, if the scenario reaches any of the below conditions:

a. The Bug Contribution Ratio is less than the Average Contribution Ratio
b. The time for execution of the test-case is greater than the average time of execution.
c. The provided net test execution time is less than the sum of individual test case execution times.

7. STEP 1: DETERMINE THE OPTIMAL PATH BY MODIFIED ANT COLONY ALGORITHM

The observation of real ants in nature is the inspiration of this original algorithm. Each ant is considered blind, weak and

almost insignificant. Yet, as a team they express strength by cooperative living and survival and demonstrate complex behavior. Finding the shortest route to the food zone is regarded as the natural behavior of ants.

Ants lay down special chemicals called "pheromones." When ants start using a specific path frequently for travelling, the pheromone concentration on it increases, other ants tend to follow this path which has got highest concentration of the pheromone.

An ant is placed randomly in each service point of the test case and, during each iteration, chooses the next testcase to go to. Dorigo (11) describes the below formula which chooses the choices.Each ant located at place **i** traverses to a place **j** selected among the test-cases, where the probability of not visiting is inferred as:

$$p_k(i,j) = \begin{cases} \dfrac{\tau_{ij}^{\alpha} \cdot d_{ij}^{\beta}}{\sum_{u \in J_k(i)} \tau_{iu}^{\alpha} \cdot d_{iu}^{\beta}} \\ 0 \end{cases}$$

where:

- $p_k(i,j)$ is the probability that ant **k** in service point **i** will go to servicepoint **j**.
- $j \in J_k(i)$ is the set of service points that have not yet been visited by ant **k** currently in service point **i**.
- α is the relative importance of the pheromone trail.
- β is the relative importance of the distance between the testcases (often determined during test strategy formation).

The exact probability of a test case getting choosen matters how much pheromone already exists on that path. By changing the α and β parameters, we can find out which of these has larger weight. When a tour is finished completely, the next step is to calculate the pheromone evaporation of the edges. The following formula (11) is used to calculate the quantity of pheromone deposits of the ant on the complete traversal

$$\tau(i,j) = \rho \cdot \tau(i,j) + \sum_{k=1}^{m} \Delta\tau_k(i,j)$$

$$\Delta\tau_k = \begin{cases} \dfrac{1}{L_k} & (i,j) \in tour_k \\ 0 & \text{if otherwise} \end{cases}$$

, where:

- $\rho \cdot \tau(i,j)$ multiplies the pheromone concentration on the edge between cities **i** and **j** by **p(RHO)**. The value of this constant of evaporation lies between zero and one, when set low it rapidly evaporates and vice versa.
- $\sum_{k=1}^{m} \Delta\tau_k(i,j)$ is the amount of pheromone an ant **k** deposits on an edge, as defined by L_k which is the length of the sample space that originated from this ant. As a natural tendency it can be inferred that short distance of traversals will yield a graph which has highest concentrate of in pheromone on the edges.

8. STEP 2 INJECT DYNAMICALLY THE NEEDED DEPENDENCIES

The idea of Dependency Injection, originally coined by Martin Fowler in 2004, states that: “Do not instantiate the dependencies explicitly in your class. Instead, declaratively express dependencies in your class definition. Use the test engine to obtain valid instances of your object's dependencies and pass them to your object during the object's creation and/or initialization”.

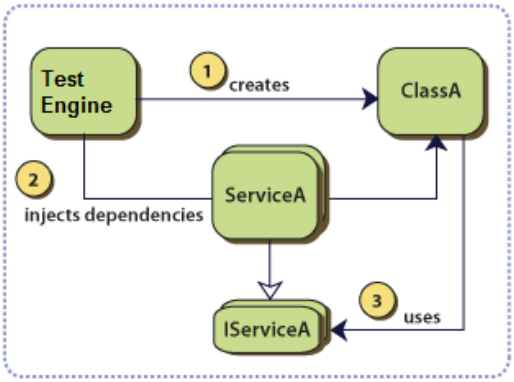

Fig 4 –Dependency Injection Architecture on Test Bed

In our case of distributed cloud, the test engine is the actual Builder which creates the Class-A, which involves a real object, i.e., testing the load balancer. The test engine then creates a service-A which has the similar properties of class-A but whose implementation is mocked up as it has injected the dependency into it upon reading the public interface of Class-A. The subsequent requests to the test engine are decided based on the available time whether to use the direct implementation or proceed with the mock implementation.

The test engine also partially switches from real object to mock object if it realizes the current run time of a particular test case has gone above the average run time and its BCR is less than ACR.

Table:3 Proposed Algorithm Psedo-Code

```
DynamicMock dataAccess = new
DynamicMock(typeof(ICloudDataAccess));
Ecom mockecom = new Ecom
((ICloudDataAccess)dataAccess.MockInstance);
Ecom realecom= new Ecom ();
While(test suite complete or  execution time elapsed==
false)
{ if(bug found==true) { report and return to next case; }
If(BCR < ACR && test is insignificant)
{
Realecom=Mockecom.createInstance();
Exit();
}
realEcom.doOperations();
```

The proposed implementation is made on the simulated results of the existing system and the actual results are tabulated as below:

Table: 3 Output Metrics of the Current System

(In Seconds)

	Tc1	Tc2	Tc3	Tc4	Tc 5	Tc6	Total (mm:ss)
keerthi	11	34	14	443	3	8	08:55
Vivek	12	21	23	45	3	9	01:53
User3	17	22	8	45	2	9	01:41
User 4	14	28	2*	45	3	9	01:41
Total time:							10:48

The second user is not similarly started 00:10 seconds as in the existing system testing, but instead is started at 09:05 minutes as the first user has finished testing, which constitutes the “Dry-Run” phase of the system. When Tc4 is evaluated, it constitutes above the average time and hence the mock object is evaluated instead of the real object thereby concurring only 45 seconds for the direct playback time to elapse, improving the system.

For the second and subsequent iterations, all the other users programmatically start at the same time. When the TC4 is evaluated, due to the load, the server had a retip causing the heart beat signal to fail, the test engine senses this immediately and provides a mock response, which yields http status code 200, and response same as the previous iteration testing, which informs the client application to proceed further with the tests.

If the test engine is configured for 100% coverage, then it behaves like the existing system and no mock is switched over in case of tipping.

9. CONCLUSION

The concept of Dynamic injection is so powerful that when combined with Artificial intelligence techniques for computing testing strategies with an option to include Decision Sub system, it performs excellently to reduce the test execution time and not compromising on test run quality.

The implementation is also subjective to catch the bugs till the point mock instance takes over so that early bugs in the system can be easily detected in the production environment without consuming enough of bandwidth.

10. REFERENCES

[1] Armando Roggio Ecommerce Know-How: Cloud Computing in the Ecommerce Forecast in Practical-Ecommerce, April 2009
[2] Atul Jain Impact of Cloud Service Models on eCommerce, HCL Blogs, 2010
[3]D.Chays,Y.Deng,P.Frankl,S.Dan,F.Vokolos,and E.Weyuker. An agenda for testing relational database applications.SoftwareTesting,Verification and Reliability,1 4:17-44, Mar 2004.
[4]Y.Deng,P.Frankl,andJ.Wang. Testing of web database applications.In Workshop on Testing, Analysis and Verification of WebServices, July2004.
[5]S.Elbaum, G.Rothermel, S.Karre, and M.Fisher.Leveraging user session data to support web-application testing. IEEE Transactions on Software Engineering, May 2005.
[6]J.A.JonesandM.J.Harrold. Test suite reduction and prioritization for modified condition/decision coverage. IEEE TransactionsonSoftwareEngineering,29(3),March2003.
[7]E.Kirda,M.Jazayeri,C.Kerer,and M.Schranz. Experiences in engineering flexible webservice. IEEE MultiMedia, 8(1):58–65,2001
[8]http://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle
[9] Edwards, W. and Barron, F.H. "SMARTS and SMARTER: Improved Simple Methods for Multiattribute Utility Measurement", Organizational Behavior and Human Decision Processes, 60, (1994), pp. 306–25.
[10] http://www.martinfowler.com/articles/injection.html
[11] Buckland M, Artificial Intelligence Techniques for Game Developers, Premier Press, Jan 2002.