AN AUTOMATED TESTING FRAMEWORK FOR THE VIRTUAL CELL GAME

A Paper
Submitted to the Graduate Faculty
of the
North Dakota State University
of Agriculture and Applied Science

By

Santosh Raj Dandey

In Partial Fulfillment
for the Degree of
MASTER OF SCIENCE

Major Department:
Computer Science

August 2013

Fargo, North Dakota

North Dakota State University
Graduate School

Title

An Automated Testing Tool For The Virtual Cell Game

By

Santosh Raj Dandey

The Supervisory Committee certifies that this ***disquisition*** complies with North Dakota State University's regulations and meets the accepted standards for the degree of

MASTER OF SCIENCE

SUPERVISORY COMMITTEE:

Dr. Brian Slator
Chair

Dr. Simon Ludwig

Dr. Limin Zhang

Approved:

09/18/2013
Date

Dr. Kenneth Magel
Department Chair

ABSTRACT

The NDSU Virtual Cell game is a role-based, multi-user computer game developed to aid in learning cellular concepts in biology. It is developed using client/server architecture with Lambda-Moo as a server and web-based Java swing applet as a client.

The Lambda-Moo (Legacy) Virtual Cell has several limitations, including development environment, performance, and maintenance. Therefore, a new virtual-cell system was developed using the Java-Moo framework to overcome the limitations of the legacy systems.

The focus of this paper is to propose, design, and implement two strategies to perform functional conformance testing between the legacy and new Virtual Cell gaming system. The Automated Testing Strategy performs simulation-based testing by automating the game play using XML test cases as an input. The Record and Replay Test strategy captures the test cases of the Virtual Cell (Lambda-Moo) in an XML document and executes them in the newer version (Java Moo).

ACKNOWLEDGEMENTS

I would like to acknowledge the help of many people who made this paper possible. I would like to thank my adviser, Dr. Brian Slator, for his continuous support, help, and direction. My sincere thanks to Dr. Simon Ludwig and Dr. Limin Zhang for serving on the committee. I would like to thank my sister Supriya Ambati and brother-in-law Venkat Ambati, my parents and my wife Mirunalini who encouraged me to complete my paper.

TABLE OF CONTENTS

ABSTRACT iii

ACKNOWLEDGEMENTS iv

LIST OF TABLES viii

LIST OF FIGURES ix

1. VIRTUAL CELL INTRODUCTION 1

  1.1. Background 1

    1.1.1. Virtual Cell (VCell) 1

    1.1.2. Problems with the Lambda-Moo VCell System (Legacy VCell) 3

    1.1.3. New Virtual Cell System (Java-Moo Version) 4

  1.2. Problem Definition 4

  1.3. Objective and Technical Approach 6

  1.4. Structure of the Paper 6

2. LITERATURE REVIEW 8

  2.1. A Systematic Capture and Replay Strategy for Testing Complex GUI Based Java Applications 8

  2.2. Our Approach 8

  2.3. Introduction to the Method 9

  2.4. Related Work 9

  2.6. Case Study Outlined 11

  2.7. An XML-Based Approach to Automated Software Testing 12

3. FUNCTIONAL SPECIFICATION FOR VIRTUAL CELL 15

  3.1. Introduction 15

  3.2. VCell Functional Categorization 16

    3.2.1. Utility Functionality 16

3.2.2. Common Functionality 16

3.2.3. Experiment Functionality 18

3.3. Deriving Functional Trees for VCell Functions 19

3.4. Derive Decision Trees from Functional Trees 22

4. VIRTUAL CELL AUTOMATED TESTING FRAMEWORK 24

4.1. Introduction 24

4.2. Our Approach for Designing an Automated Testing Framework 24

4.3. Derive XML Input Data for Each Goal 25

4.4. Derive the Test-Case Template for Each Goal 28

4.5. Writing the JFCUNIT Test Program for Each Goal 31

4.5.1. JFCUNIT Introduction 31

4.5.2. Advantages of JFCUNIT Testing 32

4.5.3. Limitations of JFCUNIT Testing 33

4.5.4. Design of JFCUNIT Test Programs for VCell game 33

4.5.5. Interface for the JFCUNIT Program 35

4.5.6. JFCUNIT Program for FindOrganelle 37

5. RECORD-AND-REPLAY TECHNIQUE FOR THE VCELL GAME 45

5.1. Overview 45

5.2. JFCUNIT Record-Replay API 47

5.2.1. Recording the XML Test Case 47

5.2.2. Replay Recorded XML Test Case 48

5.3. Virtual Cell JFCUNIT Customization 49

5.3.1. Limitations of the JFCUNIT 49

5.3.2. JFCUNIT Tag Handlers 49

5.3.3. Custom Tag Handlers for the VCell 52

5.3.4. Code Customization to Add Custom Events to the XML Recorder 59

6. TEST RESULTS 63

6.1. Test Results 63

6.1.1. Interpretation of Test Results 64

7. CONCLUSION AND FUTURE WORK 66

7.1. Conclusion 66

7.2. Future Work 69

REFERENCES 70

LIST OF TABLES

Table Page

1. JFCUNIT Test Programs for VCell 35

2. JFCUNIT Supported Tag Handlers 51

3. JFCUNIT Custom Tag Handlers for VCell 53

4. Summary of the Capture and Replay Results 64

5. Summary of the Total Number of Tested Functions 65

LIST OF FIGURES

Figure Page

1. Virtual Cell Game Showing the Laboratory Scene 2

2. Conformance Testing Functional Flow Diagram for Record-Replay of VCell................ 7

3. NAVCOMM Button Options for the VCell Experiment Goal Find a Damaged Cell..... 17

4. NAVCOMM Button Options for the VCell Experiment Goal ETC Level 0 17

5. Photosynthesis Level 1 Experiment.. 18

6. Functional Tree Showing the Experiment Goals for Each Module in VCell Game........ 20

7. Information Menu Help Topics for Organelle Identification Module 21

8. Function tree for VCell Experiment Goal Find Defective Item 23

9. Automated Testing Framework Design 25

10. Sample Design of XML Input Data 26

11. XML Data for the Goal Identify Organelle in the VCell Game 27

12. Test Case Template for Organelle Identification Experiment Goal 29

13. Test Cases for Identify Organelle Component 31

14. Sequence Diagram for Find Organelle Goal JFCUNIT Program.................................. 37

15. VCell Applet Showing Goal Information Dialog 50

16. Photosynthesis Module Level 1 Unidentified Components 59

1. VIRTUAL CELL INTRODUCTION

1.1. Background

1.1.1. Virtual Cell (VCell)

VCell is an online-based gaming system that was developed by NDSU World Wide Web Instructional Committee (WWWIC) members. Virtual Cell a goal-based and rule-based system with a 3D environment that allows students to learn about the structure and function of a cell. The Virtual Reality Modeling Language (VRML) based laboratory is the entry point for the Virtual Cell. The Virtual Cell has sub-cellular components (nucleus, endoplasmic reticulum, Golgi apparatus, mitochondria, chloroplast, and vacuoles) that are rendered as 3D objects using VRML.

The game was designed to motivate the learn-by-doing concept, and the learner received a specific assignment from the lab assistant. Initially, the learner was given small assignments to learn the basic concepts. As the students progress, they come back to the laboratory and get new assignments on different cellular components, conducting assays and experiments to learn more about these cellular components. The game has three main modules: Organelle Identification, Electron Transport Chain, and Photosynthesis. The player starts the game by exploring the Organelle Identification Module first and then does the Electron Transport Chain or Photosynthesis that he chooses. Students can get help solving assays or experiments by navigating through the Information Menu. The Information Menu provides the module-related help topics; as the player switches modules, the Information Menu options change.

Figure 1 shows the VCell application client's Laboratory screen. The lab guy is the virtual instructor who guides and assigns new goals to the players. The pop-up window's Goal Information shows the player's current experimental tasks.

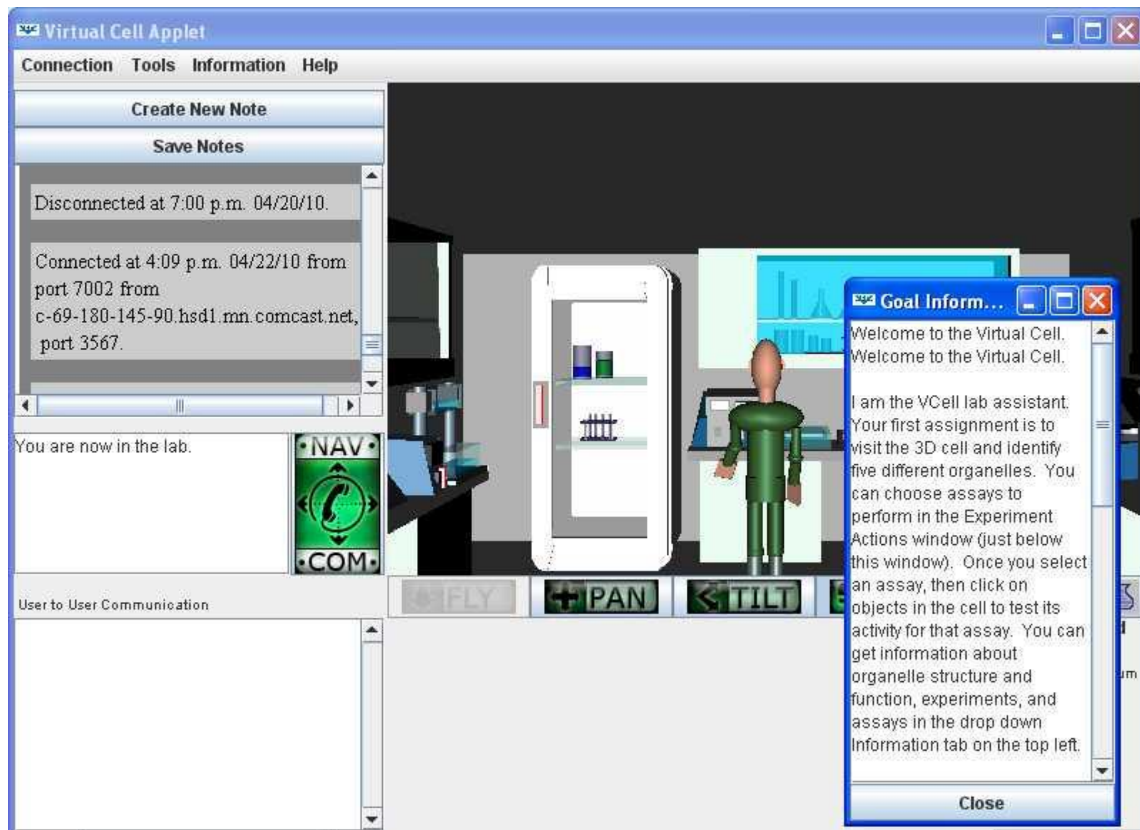

Figure 1. Virtual Cell Game Showing the Laboratory Scene

The early version of Virtual Cell (the LambdaMoo) was developed based on client-server architecture. A Lambda Moo multi-user environment server acts as a central point for the environment and performs process requests for any simulation associated with the environment. In addition, it uses an HTTP server to load graphics and model information to the Java-Swing Client and VRML2.0 to display the environments' 3D content.

Students use a standard WWW browser to launch the applet. The applet provides a connection to an object-oriented, multi-user domain where cellular processes are simulated

and multi-user viewpoints are synchronized. Virtual Cell uses an incremental scene-loading algorithm to simulate the cellular process based on node-stub and VRML scene framework.

1.1.2. Problems with the Lambda-Moo VCell System (Legacy VCell)

The Legacy Virtual Cell system was developed using Lambda-Moo; a network-based, dynamic, real-time simulation environment developed using an Object-Oriented database. It has several drawbacks and limitations in terms of performance, extensibility, and maintenance.

Unlike a modern gaming environment, such as Smalltalk and Java, Lambda-Moo is a byte-code interpreter without any support for just-in-time compilation. Because the Lambda-Moo development community is very small in size, the program did not see any major and new features added after its initial release in 1998. It's no more competitive to the other popular interpretive languages.

Lambda-Moo lacks support for code-version control and change conflict. It does not provide a way to maintain code in version control on a central repository and couldn't support version control's common functionalities like check-in, check-out, and commit from a central repository. The only way to make code changes is by using cumbersome command-line editor tools to make updates.

Lambda-MOO is not a developer friendly environment. Unlike the modern languages, it lacks support for rich graphical user interface (GUI) editors and debugging tools for easy programming. In addition, it does not provide any mechanism to protect code and data from server crashes and network outages.

1.1.3. New Virtual Cell System (Java-Moo Version)

Due to the limitations of the Lambda-Moo environment, the WWWIC community at North Dakota State University came up with new ideas to restructure the Virtual Cell game. Committee members decided to implement advanced technologies that measurably improve software speed, efficiency, and the failure rate.

The new Virtual Cell system was re-engineered to use the J2EE's client-server architecture by entirely re-writing the legacy Lambda-Moo server code in Java language. The new system uses open-source Apache Tomcat as a web server, and at the client side, it uses the same legacy java swing client used in legacy Lambda-Moo Virtual Cell. The client uses Remote Method Invocation (RMI) methodology for client-server communication through Java Moo objects which are stores in Derby database.

The JavaMoo server is a robust environment with many outgoing features, such as platform independence, portability, multi-threading, and security. Also, it has open-source GUI tools, such as eclipse, that provide the latest features and an easy-coding environment for the programmer.

Although there are considerable changes when implementing the new virtual-cell system, its working functionalities and the user interface remain same. The system's performance, reliability, and ease-of-maintenance should be significantly improved.

1.2. Problem Definition

The aim of converting LambaMOO to JavaMOO architecture is to VCell beyond the laboratory and into the realm of serious games and system for real-world deployment. To make these improvements, the existing system has to be moved on to more advance

learning technologies. This means moving beyond LambdaMOO implementation and onto next generation of learning system.

The goal is to measurably improve software speed, efficiency, and failure rate. As per considering this migration, the research and development for this migration projects was organized around the following five milestones [1] and metrics that we need to reach in completely converting the architecture to the Java-Moo architecture.

1. Convert LambdaMOO object memory to a JavaMOO relational database. Metric: 100% conversion of objects necessary for the VCell game
2. Convert VCell modules from LambdaMOO to JavaMOO. Metric: improve the mean time to failure by 50%.
3. Implement client-side simulations in Java, for each of the levels in the Electron Transport Chain and Photosynthesis modules.
4. Functional conformance between Lambda-Moo and Java-MOO and it should have 90% more conformance.
5. Conduct a system-performance experiment on the legacy LambdaMOO and the new JavaMOO and compare the run-time performance results of the two systems in terms of response times, connections times, and failure rates under a range of conditions (e.g. fast vs. client machines; fast vs. slow internet connections)..

Our goal for this paper is to implement a software testing methodology to reach milestones 4 and 5. Performing functional conformance testing between these two architectures is a challenge. The VCell game has a complex GUI structure. The user can interact with the game by utilizing a wide range of GUI interactions that range from a

simple GUI-clicking event to a complex, goal-driven sequence of interactions that involves events generated from Java GUI components and VRML window components.

1.3. Objective and Technical Approach

In this paper, we propose a methodology to test functional conformance between LambdaMOO and JavaMOO versions and to perform experiments on both of these architectures. We accomplish this goal by introducing two testing methodologies with Record-Replay technique and building an Automated Testing framework for VCell.

With Record-Replay testing technique the idea behind it is to play the VCell game in Legacy LambdaMOO system with recording mode enabled, Capture all the GUI and VRML-3D window component events and save them on a machine readable format and replay the saved events by applying these events sequentially to the new JavaMOO system. Figure.2 shows how conformance testing is done between the legacy and new systems.

The Automated Testing methodology does the tests by simulating game play. Because VCell is a goal-oriented system, a test program is returned for every goal. Each program reads the test cases defined in an XML document and simulates the game play for the goal.

1.4. Structure of the Paper

The paper is organized as follows. The first chapter gives the introduction, definition of the problem, and research objective. The second chapter explains the literature overview. The third chapter discusses about functional specification. The fourth and fifth chapter discusses about design and implementation of the Automated Testing and Record-Replay of test cases technique. The sixth and seventh chapter illustrates discusses the Test Results, Conclusion and Future work.

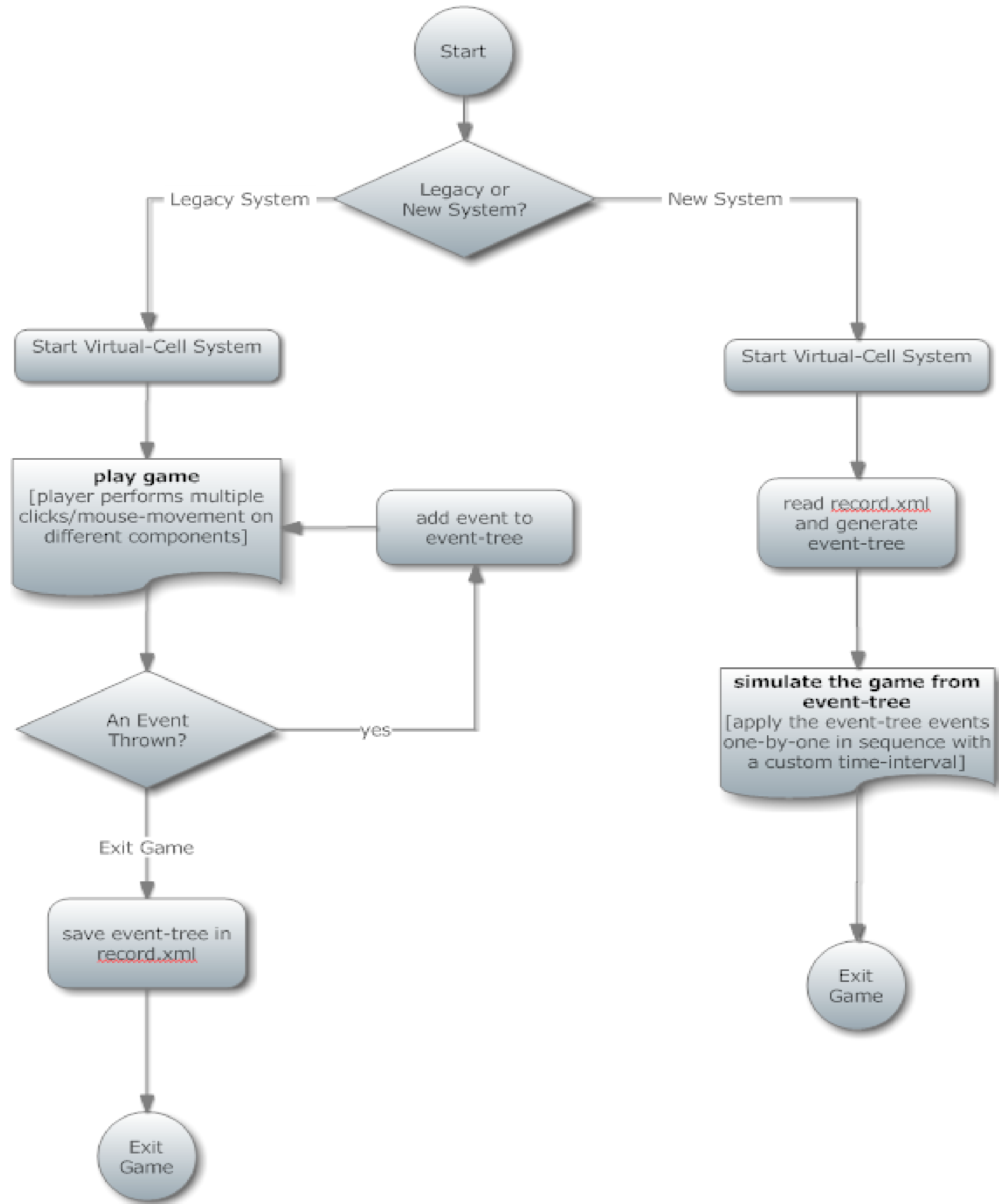

Figure 2. Conformance Testing Functional Flow Diagram for Record-Replay of VCell

2. LITERATURE REVIEW

2.1. A Systematic Capture and Replay Strategy for Testing Complex GUI Based Java Applications

There are several sources consulted to investigate this. The first piece of literature to review is a pdf file by the name, "A Systematic Capture and Replay Strategy for Testing Complex GUI based Java applications" by Omar El Ariss, Dianxiang Xu, Santosh Dandey, Brad Vender, Phil McClean, and Brian Slator. The authors are teaching staff at the North Dakota State University (Department of Computer Science), the National Center for Protection of the Financial Infrastructure and Department of Plant Science. The paper says that a capture and replay testing methodology are useful in catering for the different testing purposes such as regression testing, GUI convergence and functional testing.

2.2. Our Approach

The capture and replay methodology, or strategy, causes a dramatic improvement for replay tools and standard capture, depending on numerous aspects. The author do this by combining two main methods: utilizing the various automated test oracles and bridging a model-based testing approach by means of a capture-and-relay approach. From the literature, one discovers that tests from this model are essential in "exercising" the system in order to ensure correct behavior in terms of function. The key words "decision trees," "capture and replay," "GUI-based testing," "function trees," and "model-based testing" help one understand the fundamental topics under discussion. In order to institute a successful testing strategy that can target certain applications, it is necessary that the people involved in the process implement different oracles. These oracles include the system model's behavior based on the functional specifications or the use of trusted existing

systems to ensure that the tests derived from the model in order to exercise the system create a situation of correct functional behavior and address goal-oriented interactions. This case study applies a test strategy on a multi-user application with role-based characteristics.

2.3. Introduction to the Method

According to Paiva and Tillman [2], the introduction goes into detail about the various aspects of the graphical user interface (GUI) that includes how the users interact and manipulate the various components of the GUI and introduces the paper's main thesis. As a result of increased GUI use, its structure's applicability and complexity consequently improve. From this part, one discovers that, while the length of the event sequence goes up, the growth in the number of sequences the GUI carries out is significant. The introduction also reinforces the fact that, in order to ensure GUI's correct functions, the researcher has to use a testing strategy that employs a capture-and-replay testing approach. This part of the paper gives a preview of one test on the Virtual Cell. This testing strategy refers to the use of varied oracles for each phase, therefore leading to an improvement in the test-case evaluation. The introduction states that multiple oracles are a requirement because of the nature of the test cases which are from the decision and function trees. The introduction explains one reason why people have to test the various dynamics involved as well as the results or benefits that they expect.

2.4. Related Work

In this section, one finds that there are two methodologies for creating System under Test (SUT) test cases. These methodologies entail the specification-based approach as well as the capture-and-relay approach. The section goes on to postulate that capture-

and-replay tools form the main means to test an application's GUI and gives examples of these tools, such as CAPBAK, Rational Robot, and HP Quick Test Professional software. It also outlines the way the specification-based approach works to derive the system behavior by constructing a model for the GUI and then using the model to obtain test cases. The Finite State Machine (FSM) and Hierarchical Finite State Machines (HFSM), according to the literature, generate test cases and reduce the states in a FSM, respectively. The authors Paiva et al. and Memon [2] backed this argument. The literature says that the authors used planning techniques before embarking on graph modeling for the derivation of test cases. It also says that the authors represented the GUI as a remarkable graph known as an Event Flow Graph (EFG) before upgrading the work into Event Interaction Graphs (EIG). The use of an integration tree was vital in representing the interaction of components in the GUI according to their hierarchy (Memon and Pollack [3]).

Additionally, in third part of the same paper, there is an introductory statement that explains the issues, goals, and complexities of the testing strategy. The paper says that, in cases such as basic capture-and-replay tools, one captures events at certain component levels (e.g., the buttons and the mouse) due to the low level of abstraction. The section, however, says that goal-oriented event sequences are not fundamental to the two approaches in use. Thus, not only will the consideration of goals increase the number of GUI permutations a large margin tests, but it will also lead to complications regarding the GUI model's structure. Without a doubt, this test strategy will result in an infeasible model-based approach. The literature is clear that, in virtual environments, the user's view of the application changes depending on the number of actual users who log into the application. As a result, for both the specification and capture-and-replay based approaches, the

automated test replay must avoid reliance on the selected components and makes a good testing strategy for the limitations of both the capture-and-replay and specification-based approaches, saying that none of them supports persistent states.

Therefore, the testing strategy that the authors employ must assume the availability of a trusted system in order to derive an advantage from automating the test cases. It may also opt for modification in order to achieve accurate results.

2.6. Case Study Outlined

This section sheds light on the execution of the VCell, its modules, the relationship to derivation of the function tree, and how the various trees function. Two schematics detail the functions of the Find Defective Component (FDC) goal via the function and decision tree, and the pictures are remarkably easy to understand because of the flow diagram's nature. A case in point is the function tree for the FDC goal which clearly outlines the way the FDC component influences the navigation button or the perform assay. From the schematic, one can clearly see the link and influence exerted by the perform assay on the DNA synthesis, glycosyl transferase, phospholipid biosynthesis, and succinate dehydrogenase. This, of course, is in relation to a given biological process. The same thing applies in relation to the decision tree for the FDC goal. In this decision tree, the reader can see and learn how to handle a correct or incorrect experiment using a Boolean equation. For example, in the case of an incorrect experiment, the result will be a number of variables under "Transport me to Cell A." These variables are "go to cell," "perform assay," or "I want to file my report." In short, the paper explains the goals of the "trees" as identifying mismatches, checking for successful goal completion, and detecting

the functional error [1]. This section gives a summary of the capture-and-replay results based on modules/goals, interface mismatches, errors, and functional errors.

2.7. An XML-Based Approach to Automated Software Testing

The second piece of literature under review is a PDF document from AGM SIGSOFT with the title "An XML-Based Approach to Automated Software Testing'' [6]. The paper consists of an introduction, two headings, a conclusion, acknowledgments, and references. In the introduction, the paper brings out the main idea, or thesis, that automates the functional testing of application software along with its challenges. However, certain tests, such as those in a driver tool, simulate the responses of a human being and check the application software output while giving advantages. The paper introduces a certain software called XML (Extensible Markup Language) which has applications in the testing process. According to the literature, XML undergoes easy translation into both a readable statement and driver-tool test script. It also undergoes automated tests to check the output from the application software. This is because one can check an XML document using resources such as Xeena (a validating editor).

In the next section, titled "Wired for Learning," of the same cited paper ("An XML-Based Approach to Automated Software Testing" [6]) the literature [6] spells out the benefits of a driver tool that is useful in automated testing based on XML. This driver testing tool is extremely beneficial because it shows how the tool allows the creation of home pages. It does this by teaching easy dissemination of information; enhancing communication among parents, teachers, and students; and enabling them to plan for lessons and activities linked to national standards.

In the same section the authors also gives the technical definition of the application, explains its capabilities, and also introduces another tool called SET developed internally by IBM. This tool is useful in monitoring the operations of an independent workstation running the "Wired for Learning Application" [8]. The section also explains how the XML scripts undergo modification prior to incorporating them in the "Wired for Learning Application." This modification took place by putting them into an HTML configuration and then converting them to run tests on selected program scripts. The third section of this article deals with an example of an XML script, complete with the basic elements: the "epilog" element, verifying elements, and Document Type Declaration (DTD).

The authors of this paper described extremely informative information in stating how the various elements contribute in ensuring a successful testing process. If one takes the verifying element, for example, according to the literature, this element provides customization to the SET driver tool. It does this customization in order to permit verification of a certain area of the display. The content of this element has an output which reproduces to both the log file and the SET console. In the Prolog element, the literature reveals that this element and its set of sub elements are responsible for defining the test parameters.

The sig file attribute provides the name of the signature-holding files for each display buffer segment in order to carry out verification. Each element undergoes examination in this literature along with its necessary parameters, role, and the way the XML program manipulates it. In one part of this paper, HTML samples showing the various processes are available. The authors include verify, checking the beginning point; comment, moving with five tabs and then pressing Enter, checking that the resulting screen

is correct. The literature concludes by outlining how one can use XML to describe a functional test with regard to a certain application. As stated earlier, there are references and acknowledgements. The references are in the form of web pages from three different websites along with the necessary links. After the references, there is a full XML script which reveals the coding for the basic elements, such as <verify num + “ 2” >check resulting screen is correct</verify>.

For the test results, conclusion, and future work, the literature used “Conformance Testing for Re-Engineering the VCELL Game” [1] and “A Systematic Capture and Replay Strategy for Testing Complex GUI Based Java Applications” [8]

3. FUNCTIONAL SPECIFICATION FOR VIRTUAL CELL

3.1. Introduction

A functional specification document, in general, describes the product capabilities, appearance, and its interactions with the users. As per the testing perspective, the functional specification document allows a tester to understand the available functionality in the application and write test cases that covers all their functional areas.

The main consideration for this paper is to verify the functional behavior and the graphical user interface (GUI) structure of the new VCell system to conform to the legacy VCell. This functional conformance is a challenge because the client has a complicated GUI structure and event interactions, and also, the VCell application is left with just the source code that did not have any functional and technical documentation. Although the new and legacy VCell client has the same GUI structure, its internal workings are different due to changes in the server's architecture. For example, LambdaMOO and JavaMOO provide different mechanisms for client-server interactions. The JavaMOO client uses Remote Method Invocation (RMI) to request a proxy to an object that exists on the server object as if that object were local to the client by calling methods in the object, whereas LambdaMOO uses a text-based communication.

The Automated Testing framework is structured to execute test cases for both the legacy and new systems. First, the test suite is derived from our trusted system, legacy VCell. The same test suite is applied to the new system to assure its functional conformance with the legacy VCell.

3.2. VCell Functional Categorization

The System under test (SUT) for VCell is derived by categorizing the VCell functionalities. Functional Categorization helps in accentuating and defining the coverage criteria for the testing process. By categorizing the functionalities, we define whether we need to apply the Automated-Testing or Record-Replay strategy to achieve the testing goals. VCell functionalities are categorized as utilities, varying, and experiment functionality.

3.2.1. Utility Functionality

The utility functions are independent with respect to the goals and modules. This function does not have any dependency and can be tested with any user accounts that have varying current goals and modules; it is available for the users at all times and its behavior remains constant for all the user goals and modules in the VCell. The Create and Save Notes, User-User Communication, Utility Tools, and Help-Menu button functionalities come under this category. These functionalities can be tested using the Record-Replay strategy.

3.2.2. Common Functionality

The common functions are available for all users regardless of the goal and the module the user is playing and its behavior change as the user explores different goals and modules. All these functionalities have dependency towards the module and goal names in VCell. VRML window/3D components, Navigational Communication (NAVCOM) button, Tool-Panel, and Information-Menu are a few functions that come under this category. For example, the Navigation Communication (NAVCOMMN) button options are different for each goal. The NAVCOMMN options for the goal Find Damaged Cell in

Organelle Identification module (Figure 3) are different from the ETC Level 0 goal in the Electron Transport Chain Module (Figure 4). Each experiment utilizes these components to complete a goal.

Testing the functionalities under this category is done by using either of the two strategies: Record-Replay or Automated Testing. Automated testing provides flexibility to check these components by executing multiple test cases at the same time.

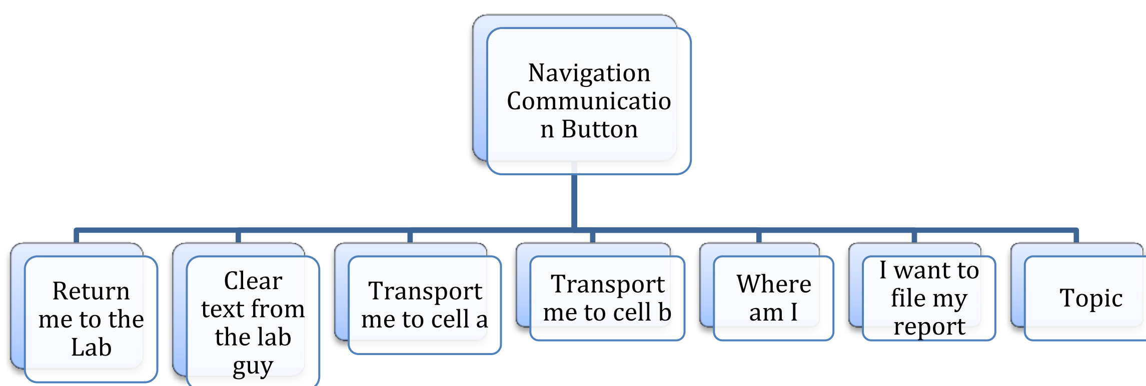

Figure 3. NAVCOMM Button Options for the VCell Experiment Goal Find a Damaged Cell

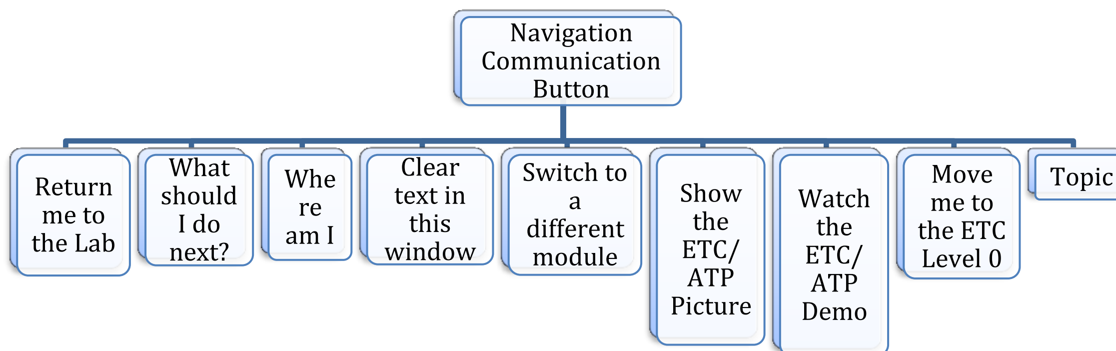

Figure 4. NAVCOMM Button Options for the VCell Experiment Goal ETC Level 0

3.2.3. Experiment Functionality

VCell is a goal-oriented game. Each goal and sub-goal in a module contains an experiment that the user does to complete a goal. The game logic gives importance to the order of events. The same events provide different results when executed in a different order. For example, in the Photosynthesis Level 1 experiment (Figure 5), clicking the Stop button after clicking the Start button has a different result compared to clicking the Start button after clicking the Stop button.

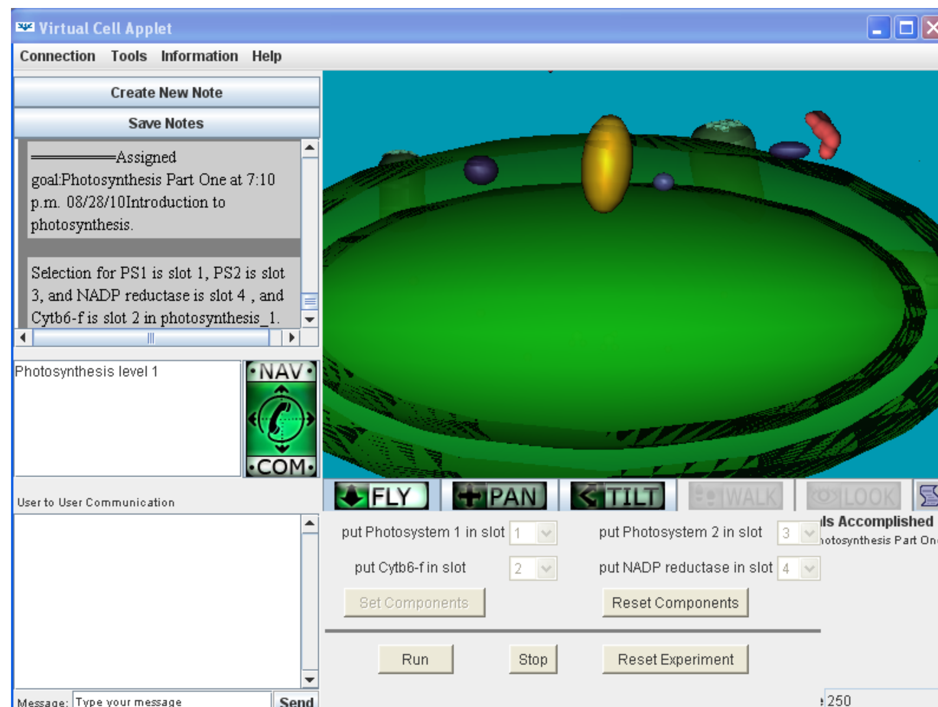

Figure 5. Photosynthesis Level 1 Experiment

To perform an experiment, the user utilizes varying functionality components and performs the experiment on the VRML components. For example, in the Photosynthesis Level 1 goal (Figure 5), the user does experiments on the VRML components, Photosystem1, Photosystem2, Cytb6-f, and NADP reductase, by placing the different

components on the slots using the tool-panel buttons (i.e., Set Components, Reset Components, Run, Stop, and Reset Experiment buttons).

3.3. Deriving Functional Trees for VCell Functions

Because there is no written documentation describing the VCell available functionalities and components, it was necessary for us to create functional diagrams to learn each goal's functionality and the components that are involved. To derive functional trees, we first played the game and wrote the user stories document capturing the components and functionalities involved for a goal, and the game-play logic to successfully complete the experiment goal. Using these captured user stories, functional models are built. VCell has a total of 22 goals. Figure 6 shows all the available goals available for the three modules Organelle Identification, Electron Transport Chain and Photosynthesis modules in VCell. For each goal, we constructed functional models. Through the functional trees, all the goal functionalities are depicted. The derived functional tree portrays the GUI interface components and all the events/actions that the user can apply for each goal. With this functional tree for each goal, we also constructed the user transfers from goal to goal. Figure 7 describes the functional tree and its NAVCOMM button options for the experiment Find Defective Item in Organelle Identification module.

This functional tree lists the functionalities and the components that are involved when playing this goal. The goal involves users interacting with the NAVCOMM button; Tool-Panel's components; and clicking VRML 3D components, such as Mitochondria, Endoplasmic Reticulum, Golgi apparatus, and Nucleus. In addition to it, the interactions also involves common operations such as clicking on Menu Items (in particular, Information menu option to read help tips about the cell's working) and VRML-3D

window navigation buttons, such as clicking FLY, PAN, TILT, WALK and LOOK button (as shown in Figure 5).

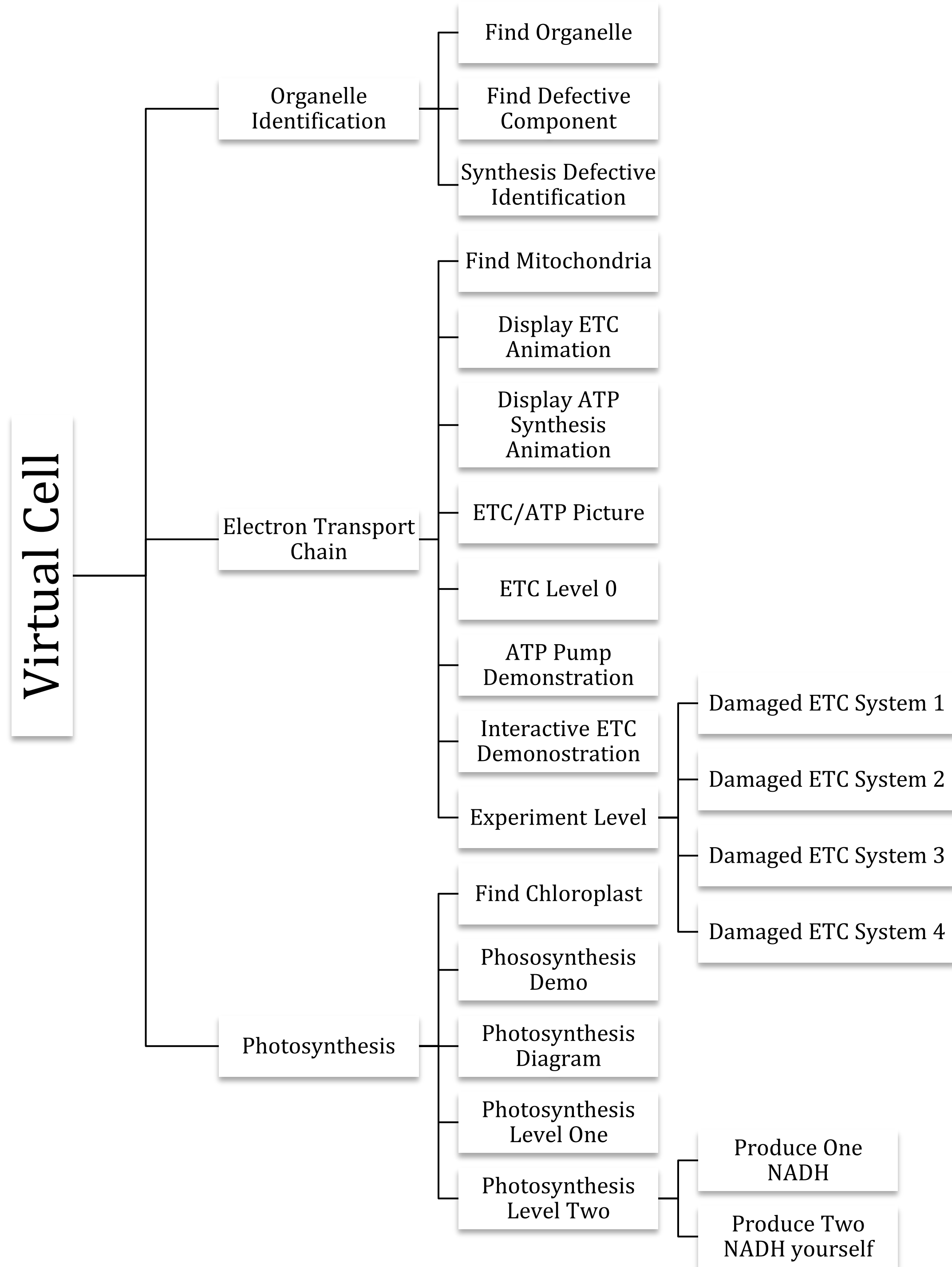

Figure 6. Functional Tree Showing the Experiment Goals for Each Module in VCell Game

Also, functional trees are constructed for Information menu help topics. These help topics are reference materials that the user can access throughout the module by utilizing the Information menu option. Each module has its own set of help topics. For example, Figure 7 describes the Information-Menu help topics that are available for organelle identification module. Similarly, Information menu help topics functional trees are created for each module.

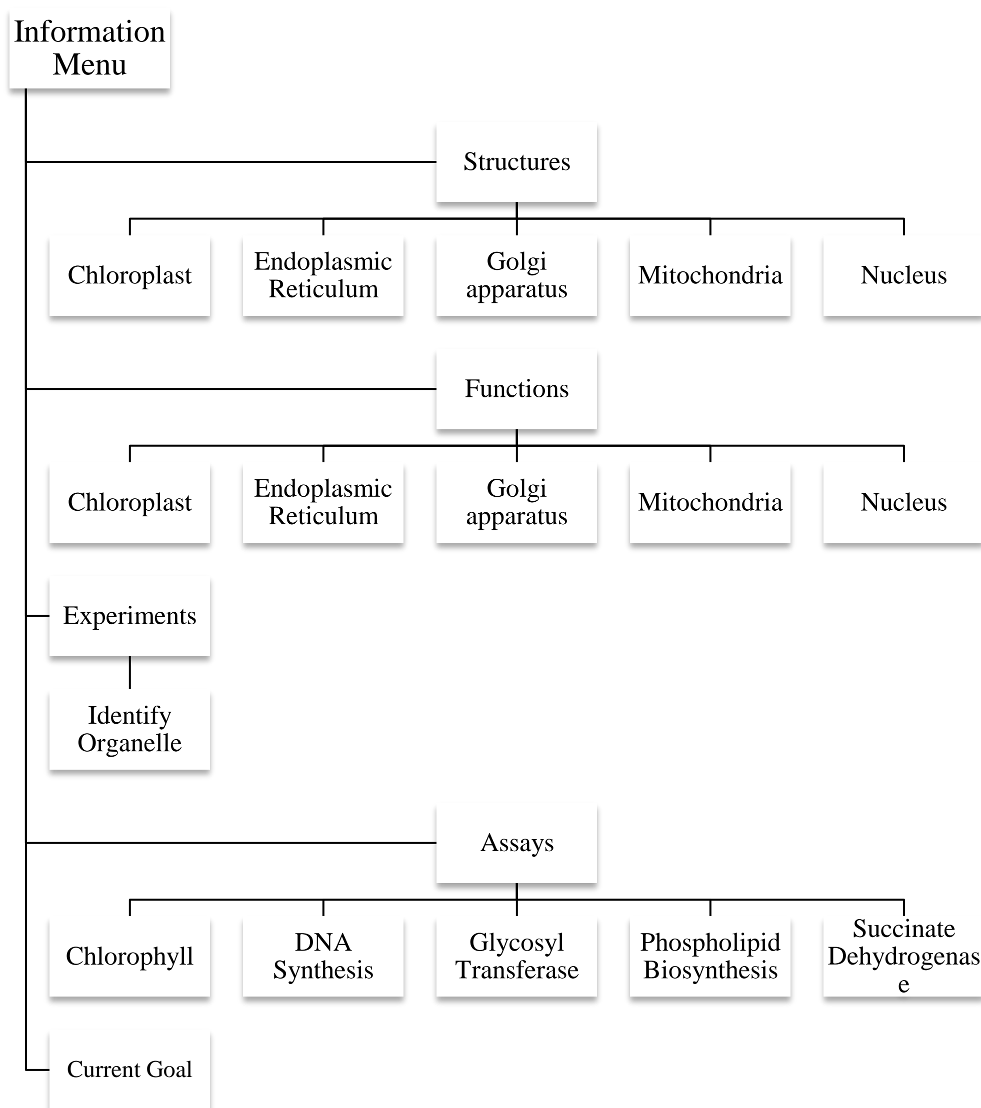

Figure 7. Information Menu Help Topics for Organelle Identification Module

Additionally, we constructed functional trees for the VCell operations that are common to all the three modules, and NAVCOMM button option for all the goals (as the button options changes for each goal).

3.4. Derive Decision Trees from Functional Trees

The functional tree to find defective component goal (Figure 8) provides a brief overview of the VCell interface components and the events that are involved with each goal. Tests should be directly derived from these function trees to ensure that all the Software under Test (SUT) interface components have been covered. The function tree model doesn't provide the event sequences that need to be performed for a successful or unsuccessful experiment. Similarly we derived functional trees for each goal in all VCell modules.

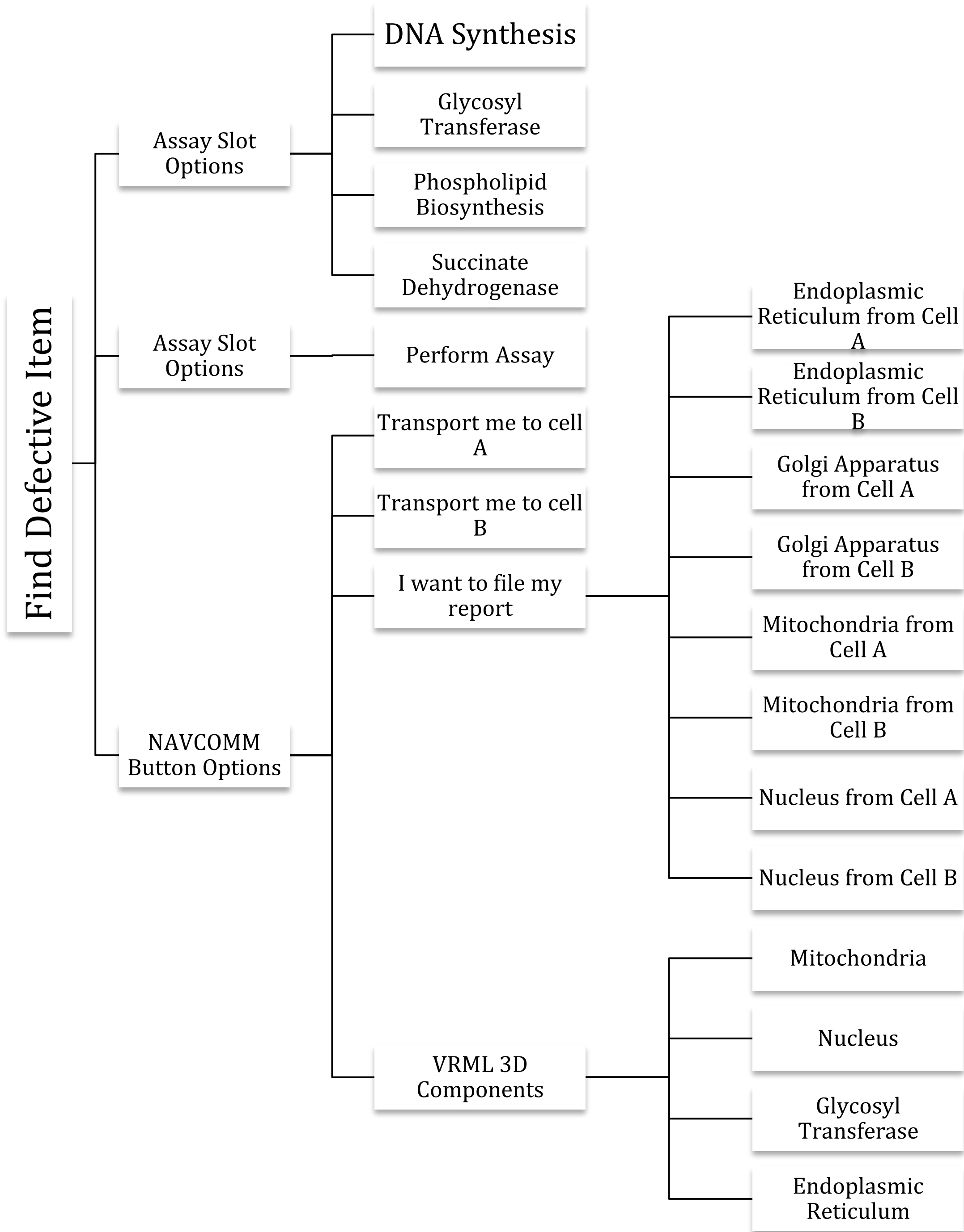

Figure 8. Function tree for VCell Experiment Goal Find Defective Item

4. VIRTUAL CELL AUTOMATED TESTING FRAMEWORK

4.1. Introduction

The VCell client is event-driven, and its test automation is complicated compared to data-driven applications. GUI based testing needs the combination of event and data stimuli. Events are triggered when a user interacts with the GUI interface. Typical GUI interactions in VCell include clicking on GUI components (i.e., buttons and VRML-3D window components), selecting options, performing experiments, etc.

The Virtual Cell SUT components are limited to experiment functionality. A separate test program was coded for each goal, simulating the game play by executing the sequence of events described in the XML test case. (We will discuss how to include a test case in the XML document in a separate section.)

4.2. Our Approach for Designing an Automated Testing Framework

As VCell didn't have functional and technical documentation, we have to reverse engineer the existing application to extract the functionality by building functional trees for each goal in VCell. We followed the similar approach described in Figure 9 to develop the Automated Testing Framework for the VCell game using JFCUNIT. We divide the process of writing JFCUNIT test programs for each goal into four phases. The goal for Phase 1 is to play the VCell game and prepare goal-based user stories for each experiment. Phase 2 goal is to derive functional and decision trees from the user stories (we covered the Phase 1 and Phase 2 topics in Chapter 3 on how we derived functional and decision trees for each experiment goal in VCell). The rest of this chapter discusses the remaining modules (Phase 3 and Phase 4). Section 4.2.1 explains how we derived functional trees for each goal. Section 4.2.2 is about designing an XML test-case template for each goal.

Section 4.2.3 illustrates code design and implementation of the JFCUNIT test program for each goal to simulate the experiment by using XML data as the input and the XML test-case template as the test suite. Finally, Section 4.2.4 provides Java code that simulates the find mitochondria goal.

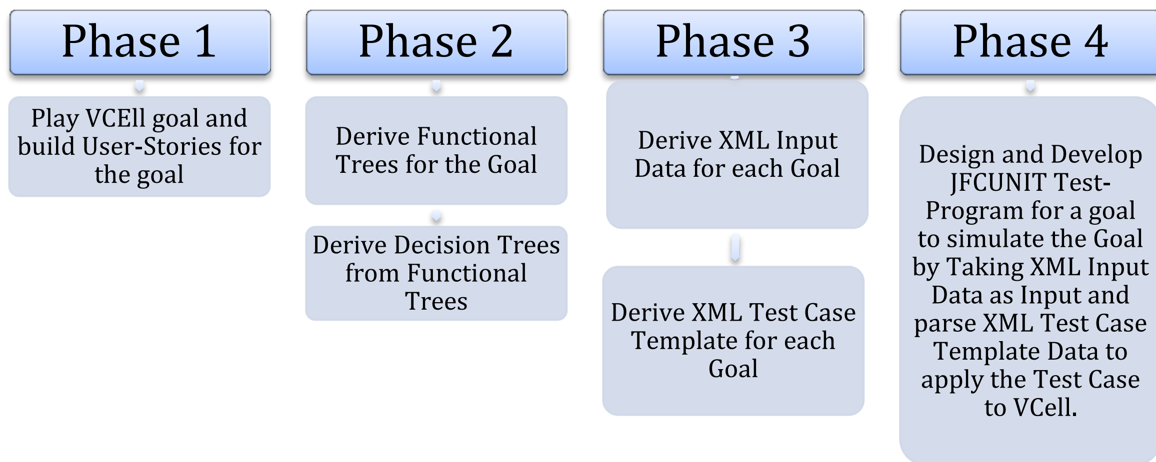

Figure 9. Automated Testing Framework Design

4.3. Derive XML Input Data for Each Goal

The data portion of a test case includes simulated input data, live input data, and predicted output data. Simulated input data are prepared beforehand to exercise the system during a given test. These data can be prepared either by an automated test-data generation tool or manually. For the VCell, the input data are derived manually using the functional trees. These input data are categorized by VCell's ModuleName and sub-categorized by GoalName. The XML input data for each goal define all the components with which the player interacts to complete the goal. The JFCUNIT test program developed for each VCell goal reads the component definitions for a goal and applies this definition to run the test cases. Figure 10 shows the high-level design of the VCell's XML input data. Each

component/sub-component has property attributes defined (i.e., Prop1, Prop2, etc.). The property attributes differ for each component/sub-component. For example Figure 11 shows the sub-components of object selection and organelle components have different attributes.

```
<VirtualCell>
    <Module ModuleName="??????">
        ....
        <Goal GoalName="???????">
            ...
        <ComponentName1 Prop1=Value1 Prop2=Value2 Prop3=Value3....>
                <SubComponenet1 Prop1=Vale1 Prop2=Value2 Prop3=Value3...>
                <...>
        </ComponentName2>
        ...

        </Goal>
        ...
        ...
    <?Module>
    ...
    ...
<VirtualCell
```

Figure 10. Sample Design of XML Input Data

Figure 11 represents the XML input data to perform the Identify Organelle experiment in the Organelle Identification Module. The Identify Organelle goal has four main components with which the user interacts (Communication Button options, Organelle Action drop-down box, Object Selection drop-down box, and VRML Components). The JFCUNIT test program designed for each goal runs the test cases by applying the below input data as component definitions and executing them. Similar to the Identify Organelle goal, the XML input data have to be created for all goals in the VCell game.

```xml
<CommunicationButton ModuleName="Organelle Identification" Goal="Identify Organelle">
  <Options Name="Return me to the Lab" Index="0" />
  <Options Name="Where am I" Index="1" />
  <Options Name="Switch to Different Module" Index="2">
    <SubOptions Name="Organelle Identification" Index="0" />
    <SubOptions Name="Electron Transport" Index="1" />
    <SubOptions Name="Photosynthesis" Index="2" />
    <SubOptions Name="Introduction Module" Index="3" />
  </Options>
  <Options Name="Transport me to the cell" Index="3" />
</CommunicationButton>


<OrganelleAction ModuleName="Organelle Identification" Goal="Identify Organelle">
  <Action Name="Perform Assay" Index="0" >
    <Object Name="Chlorophyll" Index="0" CompName="ChloroBackRight"/>
    <Object Name="DNA Synthesis" Index="1" CompName="Nucleus"/>
    <Object Name="Glycosyl Transferase" Index="2" CompName="GolgiFrontCenter"/>
    <Object Name="Phospholipid Biosynthesis" Index="3" CompName="ER"/>
    <Object Name="Succinate Dehydrogenase" Index="4" CompName="MitoBackRight" />
  </Action>
  <Action Name="Identify" Index="1">
    <Object Name="Identify Chloroplast" Index="0" CompName="ChloroBackRight"/>
    <Object Name="Identify Endoplasmic Reticulum" Index="1" CompName="ER"/>
    <Object Name="Identify Golgi Apparatus" Index="2" CompName="GolgiFrontCenter"/>
    <Object Name="Identify Mitochondria" Index="3" CompName="MitoBackRight"/>
    <Object Name="Identify Nucleus" Index="4" CompName="Nucleus"/>
  </Action>
</OrganelleAction>

<ObjectSelection ModuleName="Organelle Identification" Goal="Identify Organelle">
  <Object Name="Identify Chloroplast" Index="0" CompName="ChloroBackRight"
          Assays="Chlorophyll" AssayIndex="0"/>
  <Object Name="Identify Endoplasmic Reticulum" Index="1" CompName="ER"
          Assays="Phospholipid Biosynthesis" AssayIndex="3"/>
  <Object Name="Identify Golgi Apparatus" Index="2" CompName="GolgiFrontCenter"
          Assays="Glycosyl Transferase" AssayIndex="2"/>
  <Object Name="Identify Mitochondria" Index="3" CompName="MitoBackRight"
          Assays="Succinate Dehydrogenase" AssayIndex="4"/>
  <Object Name="Identify Nucleus" Index="4" CompName="Nucleus"
          Assays="DNA Synthesis" AssayIndex="1"/>
</ObjectSelection>

<OrganelleComponents ModuleName="Organelle Identification" Goal="Identify Organelle">
  <Components CompName="MitoBackRight" CompID="686" Structures="Mitochondria"
              Function="Mitochondria" Assays="Succinate Dehydrogenase"/>
  <Components CompName="GolgiFrontCenter" CompID="737" Structures="Golgi Apparatus"
              Function="Golgi Apparatus" Assays="Glycosyl Transferase"/>
  <Components CompName="ER" CompID="286" Structures="Endoplasmic Reticulum"
              Function="Endoplasmic Reticulum" Assays="Phospholipid Biosynthesis"/>
  <Components CompName="ChloroBackRight" CompID="697" Structures="Chloroplast"
              Function="Chloroplast" Assays="Chlorophyll"/>
  <Components CompName="Nucleus" CompID="711" Structures="Nucleus"
              Function="Nucleus" Assays="DNA Synthesis"/>
</OrganelleComponents>
```

Figure 11. XML Data for the Goal Identify Organelle in the VCell Game

4.4. Derive the Test-Case Template for Each Goal

The VCell is an event-driven game with common user interactions that include moving or clicking the mouse, selecting a graphic object, typing into a text field, or closing a window. A simple test-case template is designed for each VCell experiment so that testers can include the test cases for the experiments. This template is built to adhere to the principles of being easy to write, easy to understand, and easy to automate. Although certain restrictions in the naming conventions and the creation order of GUI components are imposed by the developer, the test-case writer should be aware of these restrictions. In case if the testers failed to practice the restrictions will result into an unspecified behavior that cannot be triggered by a test execution. In such a situation, the application may enter an unspecified state where the automated test contains no user action that will cause a transition. Say for e.g., using the XML input data of Find Defective Item goal in Organelle Identification will result in an unspecified behavior and eventually the test-case will fail. This human error is a common problem in the GUI testing environment. One main restriction applied to the test-case template (VCell Automated Testing Framework) design is to separate the test cases in a goal-by-goal fashion. Therefore, test cases are separated by module category and by sub-category goals. Also, the Automated Test Program that runs the test cases is designed to have separate programs for each goal to run the test cases. To avoid this unspecified behavior, the tester needs a continuous collaboration with a developer when writing test cases using the test-case templates.

```
<!--
    TEST CASE TEMPLATE DESIGN FOR IDENTIFY ORGANELLE GOAL
-->
<OrganelleCombs Goal="Identify Organelle">

<!-- Allowed Components and its Abbrevation

     Organelle Action Selection Drop Down List Component: OA,
     Organelle Selection Drop Down List Component : OS,
     VRML Object Component: COMP

     Allowed Components Sequence
     OA -> OS -> COMP
     OS -> OA -> COMP

     Bad Components Sequence
     COMP -> OS -> OA
     COMP -> OA -> OS
     OS -> COMP -> OA
     OA -> COMP -> OS

    Note: The order of Object Selection changes depending on action.
    say for an e.g. The index of DNA Synthesis is 1 for perform Assay, and
    for Nucleus it is 4 for Identify Organelle. so, the indexes are different
    for an assay and identify to identify the exact component

     OS:0 -> Select Identify Chloroplast or Chlorophyll from the drop-down list
     OS:1 -> Select Identify Endoplasmic Reticulum or Phospholipid Biosnthesis from the drop-down list
     OS:2 -> Select Identify Golgi Apparatus or Glycosyl Transferase from the drop-down list
     OS:3 -> Select Identify Mitochondria or Succinate Dehydrogenase from the drop-down list
     OS:4 -> Select Identify Nucleus or DNA Sunthesis from the drop-down list
    -->

    <Combs Order = "{OA:[Identify/PerformAssay], OS:[0|1|2|3|4], COMP:[ChloroBackRight|Nucleus|
    GolgiFrontCenter|ER|MitoBackRight]}" Type="Valid/Invalid"/>

</OrganelleCombs>
```

Figure 12. Test Case Template for Organelle Identification Experiment Goal

The human interaction logic to simulate the mouse/click operation on components is handled by the JFCUNIT test program. The test program runs the VCell game and applies the test cases in a sequential manner. Figure 12 shows the test-cases for organelle identification goal in VCell game using the XML test-case template. The test-case template provides a way for testers to write test-cases on the XML document. Each test case in the XML document is represented with a <Combs> XML tag. Each test-case has Order and Type attributes. Order defines comma-delimited game actions that can be applicable to the goal (e.g., **COMP:ChloroBackRight** in the Combs Order means to click the VRML component that has the name ChloroBackRight). The Type attribute defines the valid/invalid test-case scenarios that can be applied to the VCell game to complete the goal.

<Combs> represents comma delimited actions that needs to be sequentially applied to the VCell client to complete the experiment goal and the XML type attribute says whether it's a valid or invalid test-case, valid test-case represents the goal has been successfully completed and validates that the screen pop-up has appeared after finished applying the events in the XML order attribute. The invalid test-case represents the experiment is not successful finished and the experiment got aborted before completing the goal. The JFCUNIT test program provides flexibility for testers to write multiple test-cases using the <Combs> XML attribute. For example, Figure 13 shows 3 test-cases written for Identify Organelle Component. Also, the JFCUNIT test program provides options to run all test-cases at same time or pick a random test-case and run by setting the test-case configuration options.

The JFCUNIT Test program is designed based on the XML test-case template. The program reads the <Combs Order> comma-delimited game actions and applies the mouse-click or component-click events one-by-one in a sequential manner to the VCell application. Figure 13 shows the test-cases that were written using the test-case template shown in Figure 12.

```
<OrganelleCombs Goal="Identify Organelle">

    <Combs Order="{OA:Identify,
    OS:0,COMP:ChloroBackRight,
    OS:1,COMP:Nucleus,
    OS:2,COMP:GolgiFrontCenter,
    OS:3,COMP:ER,
    OS:4,COMP:MitoBackRight,
    OA:PerformAssay,
    COMP:ChloroBackRight,
    OS:1,COMP:Nucleus,
    OS:2,COMP:GolgiFrontCenter,
    OS:3,COMP:ER,
    OS:4,COMP:MitoBackRight}"
    Type="Invalid" />

    <Combs Order="{OA:PerformAssay,
    OS:0,COMP:ChloroBackRight,
    OS:1,COMP:Nucleus,
    OS:2,COMP:GolgiFrontCenter,
    OS:3,COMP:ER,
    OS:4,COMP:MitoBackRight}"
    Type="Invalid" />

    <Combs Order="{OA:PerformAssay,
    OS:0,COMP:ChloroBackRight,
    OS:1,COMP:Nucleus,
    OS:2,COMP:GolgiFrontCenter,
    OS:3,COMP:ER,
    OS:4,COMP:MitoBackRight,
    OA:Identify,
    OS:0,COMP:ChloroBackRight,
    OS:1,COMP:ER,
    OS:2,COMP:GolgiFrontCenter,
    OS:3,COMP:MitoBackRight,
    OS:4,COMP:Nucleus}"
    Type="Valid" />

</OrganelleCombs>
```

Figure 13. Test Cases for Identify Organelle Component

4.5. Writing the JFCUNIT Test Program for Each Goal

4.5.1. JFCUNIT Introduction

JFCUNIT is an extension of the JUNIT testing framework that provides a reach Application Program Interface (API) to write test cases for Java swing-based applications at the user interface layer. JFCUNIT provides support to write GUI tests that mimic a user's interaction with the User Interface (UI). JFCUNIT provides the following functionalities:

1. Obtaining handles on the windows/dialogs opened by the Java code.

2. Locating components within a component hierarchy that occur within the containers found.
3. Raising events on the found components, e.g., clicking a button or typing text in a Text Component, or selecting a value in a combo-box.
4. Handling testing of components in a thread safe manner.

4.5.2. Advantages of JFCUNIT Testing

There are many advantages of JFCUNIT Testing, or Test Driven Development (TDD), namely early detection of problems, the facilitation of change, documentation, and the simplification of integration to name a few. When TDD is useful in the context of an automatic-testing framework, a number of things occur. The unit tests are created before the actual code is typed out, and when the tests are positive (showing no errors), the code is complete. Those same tests are then frequently run against the function as the large code foundation is developed and the code changes. If there is a failure of the unit test, the developer can conclude that a bug exists in the actual tests or the changed code. Through the unit tests, the fault location is traceable at a remarkably early stage. Testing frameworks and unit tests allow developers or programmers to refactor code at a later period and to employ regression testing to make sure the module works correctly. The basic procedure for testing the framework is to write test cases for all the necessary functions so that anything that causes a fault can be detected and rectified as soon as possible. In short, readily available tests and testing frameworks allow a programmer to see whether a piece of code is working correctly and provide sustained maintenance for the executable code in addition to foster code accuracy.

4.5.3. Limitations of JFCUNIT Testing

The very first limitation we experienced with JFCUNIT Testing while using it for VCell game is its inability or partial support for identifying Java's Abstract Windows Toolkit's (AWT) components. The built-in JFC Unit's API *Finder* and *NamedComponentFinder* class methods could not find the AWT component while searching for it using the name. We discovered this issue only while doing code development. This limitation is not mentioned explicitly in the API documentation or on the site. The VCell game developers have extensively used both Java Swing and AWT components for game development.

Another known limitation of the JFCUNIT is its incompatibility with VRML component detection. The JFCUUNIT does not have an API to support the detection of VRML components. Due to these limitations, we could not use the JFCUNIT's built-in Component Finder functionality to perform select or click operations on Java AWT components and in VRML windows. To overcome this limitation, we wrote a custom tag handler for each AWT component for every experiment window. We discussed more about custom tag handlers in detail in Chapter 5 about the record-replay technique.

4.5.4. Design of JFCUNIT Test Programs for VCell game

Because VCell is a goal-oriented game, the player starts with a simple goal and unlocks the next level after finishing a goal. Each game level has different goals, different experimental components (VRML, Swing, and AWT components), and different interaction sequences between the components to complete a goal. The design is to write separate JUNIT testing programs for each goal, automating the user play of each goal.

The Java program is designed using the XML input data and XML test case that we derived from Functional and Decision Trees (as described in Figure 9 from chapter 4). The program extracts the sequences of events in the order defined using the tag <Combs order="{}"/> and applies it to the game (for example, the Test Program reads the Test Cases defined in XML file as described in Figure 13). The Order has the sequence of abbreviated events. The program reads the meaning for each of the XML test-case abbreviated component events, reads the configuration for the detected component from the XML data (as defined in Figure 12), and performs action on that component.

Consider the XML test case <Combs Order= *"{OA:Identify, OS:0, COMP:ChloroBackRight}"/>*. When the JFCUNIT program reads this test-case order, it interprets the order to perform the following actions in sequence.

1. OA:Identify – interprets to select the value Action on the drop-down combo-box list item.
2. OS:0 – interprets to select button to "Identify" and the Object Selection drop-down button to "Identify Chloroplast," and then the program performs a click operation on the "ChloroBackRight" VRML component. The Java program reading this order sequence gets the configurations of each component from the <OrganelleAction>, <ObjectSelection>, and <OrganelleComponents> XML data and applies the configurations appropriately.

Table 1 shows test programs we developed for each goal in VCell game. Each program provides options either to run all test cases defined in the XML document or to run a randomly selected test case. The following sections shows the skeleton for a typical

JFCUNIT test program; all the JUNIT programs described in Table 1 follow the same interface shown in the next section.

Table 1. JFCUNIT Test Programs for VCell

Module Name	***Goal Name***	***Java Class***
Organelle Identification	*Identify Organelle*	**TestModule1_FindOrganelle.java**
	Find Defective *Item*	**TestModule1_FDI.java**
	Synthesis Defective *Identification*	**TestModule1_SDI.java**
Electron Transport Chain	Find Mitochondria	TestModule2_FindMitochondria.java
	Move to ETC Level 0	TestModule2_ETC0.java
	Move to the ATP Pump Demo	TestModule2_ATPPump.java
	Interactive ETC Demo	TestModule2_InteractiveETC.java
	Identifying a Defective ETC System	TestModule2_ETCDamagedCell.java
Photosynthesis	Find Chloroplast	TestModule3_FindChloroplast.java
	First Photosynthesis Level	TestModule3_PSLevel1.java
	Second Photosynthesis Level [Task1 and Task2]	TestModule3_PSLev2Task1.java TestModule3_PSLev2Task2.java

4.5.5. Interface for the JFCUNIT Program

The JFCUNIT programs (described in the Figure14) follows a similar pattern when writing the JFCUNIT test programs for VCell. The below java interface shows java methods signature that each program needs to implement in order to develop the actual implementation of the JFCUNIT program.

```
public interface <ModuleName>_<GoalName> extends JFCTestCase implements
VCELLUIConstants {
        public <ModuleName>_<GoalName>(String name) {}
        protected void setUp() throws Exception {}
        protected void tearDown() throws Exception {}
```

```
	public void testGoalName() {}

	public void parse<GoalName>(String xmlURL, String miniMeExp,	String goalNameCombsOrder) {}

	public void <goalName>TestCases(org.jdom.Element nodeOrganelleCombs, org.jdom.Element nodeMiniMe) {}

	public void randomCombs(org.jdom.Element nodeOrganelleCombs, org.jdom.Element nodeMiniMe) {}

	public void iterativeCombs(org.jdom.Element nodeOrganelleCombs, org.jdom.Element nodeMiniMe) {}

	private void clickMiniMeOptionByName(org.jdom.Element nodeMiniMe,String option) {
	}

	private void findFrame(String title) {}

	private void bypassInformationDialog(int waitTime) {}

	private void clickJButtonComponent(String buttonText) {}

	private void clickCommandDialogActionButton(boolean ActionButton) {}

	private void clickNamedComponentFinder(String text, Container cont,  String errorMessage, int index) {}

	private static void waitfor(long duration) {}
}
```

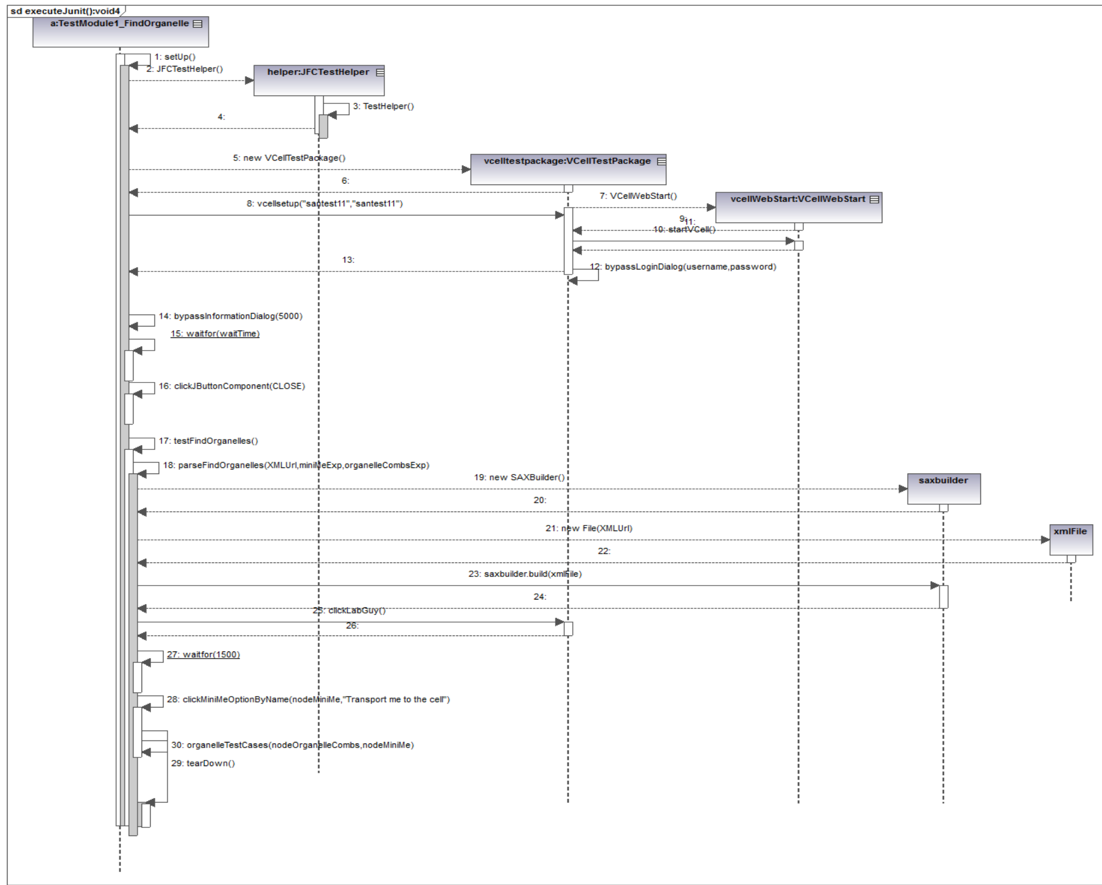

Figure 14. Sequence Diagram for Find Organelle Goal JFCUNIT Program

4.5.6. JFCUNIT Program for FindOrganelle

The below java program shows the complete implementation of JFCUNIT program for VCell goal Find Organelle in module Organelle Identification. Executing this java program will perform experiment based on the test case defined in the XML document. Figure 14 sequence diagram shows the sequence of operations or methods that gets executed and, the interactions in between different java components of VCell. (This sequence diagram is generated using the tool Altova UModel tool with the maximum invocation depth value 3).

```
package edu.nodak.ndsu.games.test;

import java.awt.Container;
```

```java
import java.io.File;
import java.io.IOException;
import java.util.List;
import javax.swing.JButton;
import javax.swing.JComponent;
import org.jdom.Element;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;
import edu.nodak.ndsu.games.util.CommunicatorBar;
import edu.nodak.ndsu.games.vcell.CommandDialog;
import edu.nodak.ndsu.games.vcell.NodeStub;
import junit.extensions.jfcunit.JFCTestCase;
import junit.extensions.jfcunit.JFCTestHelper;
import junit.extensions.jfcunit.TestHelper;
import junit.extensions.jfcunit.eventdata.MouseEventData;
import junit.extensions.jfcunit.finder.AbstractButtonFinder;
import junit.extensions.jfcunit.finder.FrameFinder;
import junit.extensions.jfcunit.finder.NamedComponentFinder;


import junit.framework.AssertionFailedError;
import junit.framework.TestCase;

public class TestModule1_FindOrganelle extends JFCTestCase implements
VCELLUIConstants {

private VCellTestPackage vcelltestpackage = null;
private TestHelper helper = null;

public TestModule1_FindOrganelle(String name) {
		super(name);  }

protected void setUp() throws Exception {
	super.setUp();
	helper = new JFCTestHelper();
	vcelltestpackage = new VCellTestPackage();
	vcelltestpackage.vcellsetup("santest11", "santest11");
	bypassInformationDialog(5000);
}

protected void tearDown() throws Exception {
	super.tearDown();
}

public void testFindOrganelles() {
	String XMLUrl = UISPECURL;
```

```
        String miniMeExp = EXP_MINIME_MOD1_IO;
        String organelleCombsExp = EXP_COMBS_ORGANELLE_IO;
        parseFindOrganelles(XMLUrl, miniMeExp, organelleCombsExp);
}

private void parseFindOrganelles(String XMLUrl, String miniMeExp,     String
organelleCombsExp) {
        org.jdom.Document XMLDocument;
        SAXBuilder saxbuilder = new SAXBuilder();

try {
        File xmlFile = new File(XMLUrl);
        XMLDocument = saxbuilder.build(xmlFile);
        org.jdom.Element nodeMiniMe = (org.jdom.Element)
        org.jdom.xpath.XPath.selectSingleNode(XMLDocument, miniMeExp);
        assertNotNull("Error in reading MiniMe Options from XML Document",
                nodeMiniMe);
        org.jdom.Element nodeOrganelleCombs = (org.jdom.Element)
        org.jdom.xpath.XPath.selectSingleNode(XMLDocument, organelleCombsExp);
        assertNotNull("Error in reading Organelle Combination elements from XML
        Document",                       nodeOrganelleCombs);

        vcelltestpackage.clickLabGuy();
        waitfor(1500);
        clickMiniMeOptionByName(nodeMiniMe, "Transport me to the cell");
        organelleTestCases(nodeOrganelleCombs, nodeMiniMe);
} catch (JDOMException e) {
        e.printStackTrace();
} catch (IOException e) {
        e.printStackTrace();
        }
}

/** this method performs three types of test, Randomized Valid Test Case, Randomized
Invalid Test Case and Iterative test-case.
@param nodeOrganelleCombs Organelle Combinations node from XML Specification
Document
@param nodeMiniMe miniMe node from XML Specification Document */

private void organelleTestCases(org.jdom.Element nodeOrganelleCombs,
org.jdom.Element nodeMiniMe) {

// get a random combination that has the type Invalid and perform assays and identification
randomCombs(nodeOrganelleCombs, false);

// get a random combinaton that has the type valid and perform assays and identification
```

```
randomCombs(nodeOrganelleCombs, true);

//iterative testcases
iterativeCombs(nodeOrganelleCombs, nodeMiniMe);
		}

/** This method iteratively performs the click operation by selecting each combination
present in OrganelleCombs node
* @param nodeOrganelleCombs
* @param nodeMiniMe
*/
private void iterativeCombs(org.jdom.Element nodeOrganelleCombs,
org.jdom.Element nodeMiniMe) {

List OrganelleCombsList = nodeOrganelleCombs.getChildren();
org.jdom.Element elmtCombs;
String goalName;
for (int combsIndex = 0; combsIndex < OrganelleCombsList.size(); combsIndex++) {

elmtCombs = (org.jdom.Element) OrganelleCombsList.get(combsIndex);

goalName = elmtCombs.getParentElement().getAttributeValue(GOAL);

assertNotNull("Error in Reading Goal Information", goalName);

System.out.println("Testing.....Goal Name we retrieved:" + goalName);

vcelltestpackage.clickOrganelleComp(elmtCombs, goalName);

if (elmtCombs.getAttributeValue(TYPE).equals(VALID)) {
		System.out.println(" Testing....Successfully Completed the Find Organelle Goal and
		exiting....");
		return;
		}
  }
}


/** This Method Randomly Selects a Combination from OrganelleCombs Node and
iteratively performs click operations on the components. each combination will have an
attribute type VALID/INVALID. VALID represents, the combination can successfully
completes the goal and vice-versa is the INVALID
		* @param nodeOrganelleCombs
		* @param ISVALID
		*		true->VALID, false->INVALID
		*/
@SuppressWarnings( { "static-access", "static-access" })
```

```
private void randomCombs(org.jdom.Element nodeOrganelleCombs,
        boolean ISVALID) {
org.jdom.Element elmtRandomCombs = vcelltestpackage.getRandomElement(
nodeOrganelleCombs, ISVALID);
assertNotNull("Random Element not found", elmtRandomCombs);
String goalName = elmtRandomCombs.getParentElement(). getAttributeValue(GOAL);
System.out.println("Testing.....Goal Name we retrieved:" + goalName);
System.out.println("Testing....element in randomCombs has the following order:"
                                                        +
elmtRandomCombs.getAttributeValue(ORDER));
vcelltestpackage.clickOrganelleComp(elmtRandomCombs, goalName);
        }

/**
* Performs a click operation on the DoIt Button of Communication Button by using the
given OptionName

* @param nodeMiniMe
* Element Node for the MiniMe
* @param option
* Option Name to Click
 */
private void clickMiniMeOptionByName(Element nodeMiniMe, String option) {

int index = vcelltestpackage.findMiniMeOptions(nodeMiniMe, option);

if (!(index == -1)) {
        clickMiniMeButton(index);
        System.out.println("Testing...Clicked Mini Me Option:" + option);
        }
else
        throw new AssertionFailedError("Error in reading Mini-me Option for goal " +
        nodeMiniMe.getAttributeValue(GOAL));
}


/** This method will perform a click operation on MiniMe button, select the given index
value and performs a click operation on do button
* @param index  index of the miniMeOption to be selected
*/
private void clickMiniMeButton(int index) {
        String errmsg = "Communication Button Not found";
        clickNamedComponentFinder(CommunicatorBar.COMMBUTTON,
        vcelltestpackage.getVCellApplet().getcontrolsPanel().getcommButton(), errmsg, 0);
        findFrame("Tell Lab Guy");
        vcelltestpackage.clickCommandDialogButton(index);
```

```
        waitfor(1000);
        clickCommandDialogActionButton(true);
}

/**
* this method will look for the frame with the given title
* @param title
* title of the frame
*/
private void findFrame(String title) {
        FrameFinder finder = new FrameFinder(title, true);
        finder.find();
        assertNotNull(title + " frame not found", finder);
        System.out.println("Testing.....found frame:"
                + finder.getTitle().toString());
}

private void bypassInformationDialog(int waitTime) {
        waitfor(waitTime);
        clickJButtonComponent(CLOSE);
}

/** finds and click the Button Component that has the label name buttonText
 @param buttonText
* label name for the button
*/
private void clickJButtonComponent(String buttonText) {

AbstractButtonFinder finder = new AbstractButtonFinder(buttonText, true);
JButton compButton = (JButton) finder.find();
assertNotNull("Couldn't find the " + buttonText, compButton);

try {
        helper.enterClickAndLeave(new MouseEventData(this, compButton));
        System.out.println("Testing....Clicked ||" + buttonText
                                        + "|| Button");
        } catch (Exception ex) {
                ex.printStackTrace();
        }
                waitfor(1000);
}

/**
 * @param ActionButton
 * true/false ==> click on DOButton/Click on CANCELButton
 */
```

```
private void clickCommandDialogActionButton(boolean ActionButton) {
waitfor(500);
String errmsg = "Command Dialog's DO Button Not FOund";
if (ActionButton) {
	clickNamedComponentFinder(CommandDialog.CMDDIALOG_DOBUTTON,
	vcelltestpackage.getVCellApplet().getcommandDialog(),
	errmsg, 0);
	System.out.println("Testing....Clicked Command Dialog Do
	Button");
	}
else {
	clickNamedComponentFinder(CommandDialog.CMDDIALOG_CANCELBUTTON
	,vcelltestpackage.getVCellApplet().getcommandDialog(),
	errmsg, 0);
	System.out.println("Testing....Clicked Command Dialog Cancel Button");

	}
	waitfor(2000);
}

/**
* perform a click operation on the given component
* @param text component name
* @param cont container to search
* @param errormessage assert error message to display
* @param index index of the component to be clicked if multiple components present on
the same name
*/
private void clickNamedComponentFinder(String text, Container cont,	String
errormessage, int index) {

NamedComponentFinder finder = new NamedComponentFinder(
	JComponent.class, text);

JButton commButton = (JButton) finder.find(cont, index);

assertNotNull(errormessage, commButton);
try {
	helper.enterClickAndLeave(new MouseEventData(this, commButton));
} catch (Exception ex) {
	ex.printStackTrace();
}
	waitfor(1000);
}

private static void waitfor(long duration) {
```

```
try {
        Thread.sleep(duration);
        } catch (InterruptedException e) {
                e.printStackTrace();
        }
}


}
```

5. RECORD-AND-REPLAY TECHNIQUE FOR THE VCELL GAME

5.1. Overview

The record-and-replay program replicates test cases for the user playing the VCell game. This technique allows the user to play the game, and the program automatically generates a test case that is an exact match for the actions taken by the user on the VCell GUI. This test case can then be executed to repeat the user's game playing. This testing technique reduces the difficulties in testing the VCell GUI by significantly reducing the process of writing test cases. One benefit of using this technique is that the method allows testers to play the game simultaneously on different VCell goals and capture all the events from the game's start until completing a goal and store them in a flat file, and simulate the game play when needed using the test cases stored in a flat file.

The record-and-replay technique is implemented using Java, the JFCUNIT, and the JFCEventManager API to help record and replay the GUI events. The record-and-replay program is composed of two parts:

1. Record part: This part is the automated test-generation process. It is responsible for monitoring the user's actions on the GUI and recording all the actions, such as name of the clicked component, mouse click and position change events, and the proprietary VRML window's object-click events. These sequences of captured actions are then transformed to an XML file with each action stored as an XML tag with the JFCUNIT's event name as an XML tag name. The set of events in the stored XML file form a test case.

2. Replay part: This part is the automated test-executing process. It is responsible for executing the recorded.xml test cases by reading the JFCUNIT events sequentially from the XML file and applying them with a configurable, delayed time.

An advantage of the existing capture-and-replay tools is that the capture part records GUI events without having to deal with low-level GUI interactions such as mouse positions and button-click events. Rather than capturing the position of the clicked component, we capture the component name. Then, the replay program only needs the component name to activate the event. In addition, clicked mouse positions are still recorded and used to simulate the mouse movement while the script is being replayed.

Another advantage of capture-and-replay tools is that the record part of the program can be run on two different systems, the legacy and the new VCell. Recording test cases from the legacy system is very useful for the functional conformance between the legacy VCell and the SUT.

Also, recording test cases from the SUT side simplifies the process of creating automated test cases for the new system. The test cases are intended to derive tests from functional trees by exercising all the GUI components. For each goal, the same test cases are recorded on both the legacy and the new client side; then, the cases are compared to find any functional or GUI mismatches. Examples of mismatches that we find in these processes are missing or additional buttons; a different goal order; inconsistency between the information dialogs being displayed, such as different instructions and messages; and a different task order.

5.2. JFCUNIT Record-Replay API

The JFCUNIT API provides built-in support for recording and replaying Java Swing applets. A separate program is written to record and replay XML test cases in the VCell applet. The following two sections describe how we accomplished the two events.

5.2.1. Recording the XML Test Case

A separate Java program, with recording enabled, is written to run the VCell applet. The following code snippet shows how to start the VCell applet with recording enabled and to store the recorded events in a saved.xml file.

```
public TestXMLRecording() {
    super("xmlrecordingtemplate.xml",
        XMLUtil.readFileFromClassContext(
          TestXMLRecording.class,
          "xmlrecordingtemplate.xml"));
      vcellWebStart = new VCellWebStart();
            vcellapplet= vcellWebStart.getVCellApplet();
            vcellWebStart.startVCell();
            vcellapplet.getSocketReaderThread().ISRECORDING = true;
            vcellapplet.waitForStartup();
    //VCellWebStart.main(new String[] {});
    try {
     Thread.currentThread().sleep(3000);
    } catch (InterruptedException ex) {
    } }
```

The above program captures all the low-level component events such as mouse-clicks, button-click events, and drop-down events without any customization to the VCell applet. The *vcellapplet*.getSocketReaderThread().*ISRECORDING* = **true** assignment operation in the above code allows the recording of custom events that we developed for the VCell game due to the limitations of the JFCUNIT (as discussed in section 4.5.3) for not supporting AWT and VRML window components . We discuss this code customization in detail on a separate section in Section 5.3 (Virtual Cell JFCUNIT Customization).

5.2.2. Replay Recorded XML Test Case

The TestXMLReplay.java program is written to replay the saved XML test-case file to the VCell applet. The following code snippet takes care of replaying the XML test case to the VCell applet. The program initially reads the events in the saved.xml, and the JFCUNIT library applies the events one-by-one after starting the VCell applet.

```
public TestXMLReplay() {
    super("saved.xml",  openFile("saved.xml"));
    XMLRecorder.setReplay(true);
            repvcellWebStart = new VCellWebStart();
            repvcellapplet= repvcellWebStart.getVCellApplet();
            repvcellWebStart.startVCell();
            repvcellapplet.waitForStartup();

    try {
      Thread.currentThread().sleep(3000);
```

```
    } catch (Exception e) {
      // Ignore
    }
  }
```

5.3. Virtual Cell JFCUNIT Customization

5.3.1. Limitations of the JFCUNIT

Capturing all the events in the VCell user-interface interactions create some challenges. As mentioned in the JFCUNIT API Limitations section of Chapter 4, the JFCEventManager does not handle the interactive 3D components of the VRML window and the GUI components that are built using the Java AWT toolkit.

5.3.2. JFCUNIT Tag Handlers

To overcome the limitations, we have extended the JFCUNIT's tag handler functionality to support VRML window components and Java AWT components. The JFCUNIT recording template program in the background uses tag handlers to record and store the events in an XML file. For example, clicking the close button (as shown in Figure 15) in the Goal Information dialog creates two events, <find> and <click>, in an XML file (as shown below) while running the JFCUNIT recording template program.

```
<find class="javax.swing.JButton" container="JFrame4"
finder="NamedComponentFinder" id="Component5" index="0"
name="TEST_INFORMATIONDIALOG_CLOSEBUTTON"
operation="equals"/>
```

<click position="custom" reference="144,8" refid="Component5"
type="MouseEventData"/>

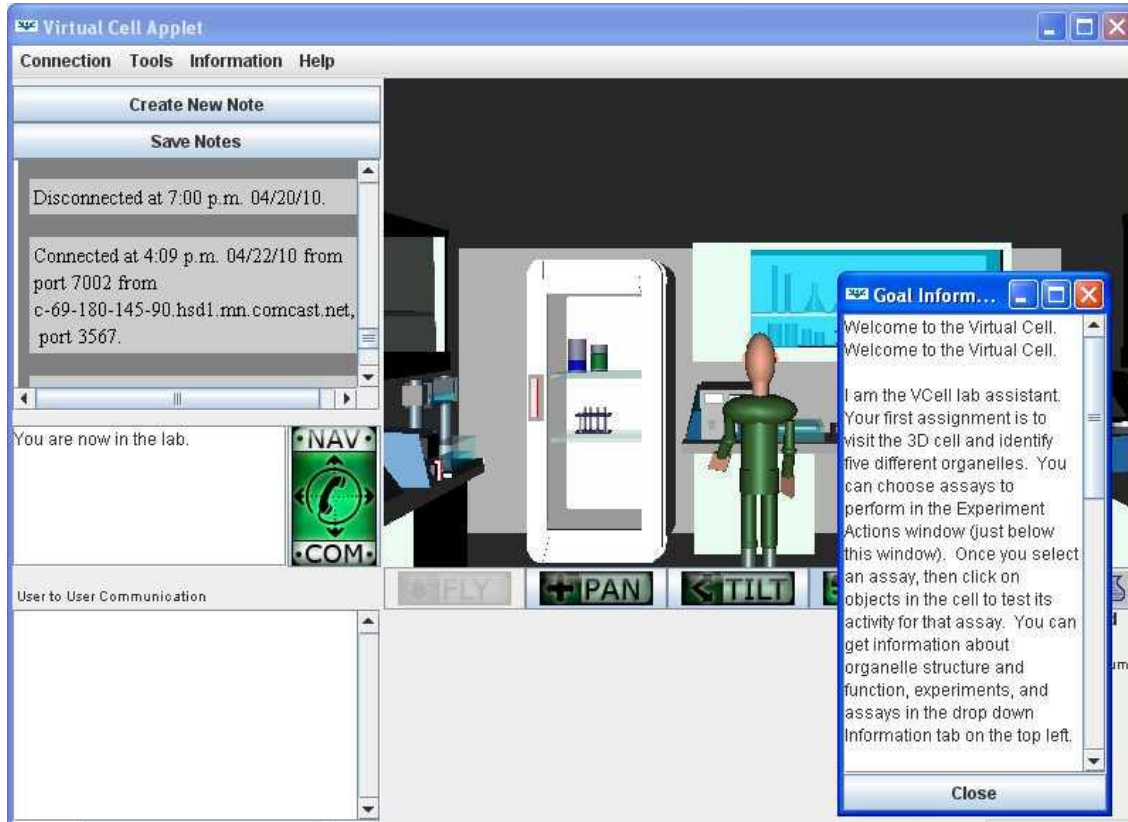

Figure 15. VCell Applet Showing Goal Information Dialog

The replay JFCUNIT program loads the XML test case and reads the events one-by-one, and the program calls the corresponding tag handlers and executes the operation. For example, the replay program calls the FindTagHandler.java class for the <find> tag and the ClickTagHandler.java program for the <click> tag. Table 2 shows a list of built-in event and tag handlers support for JFCUNIT. The JFCUNIT record program keeps listening to the Java applet's component events and activates the appropriate tag handler in the run-time to record that event and save the event in an XML file. The replay program

then reads the events in the XML file and executes the associated tag handlers for the event in a sequential manner to simulate the recorded play.

Table 2. JFCUNIT Supported Tag Handlers

Event Name	***Tag Handler Class***
Topwatch	StopWatchTagHandler
assertequals	AssertEqualsTagHandler
assertnotequals	AssertNotEqualsTagHandler
assertnotsame	AssertNotSameTagHandler
assertsame	AssertSameTagHandler
assertnull	AssertNullTagHandler
assertnotnull	AssertNotNullTagHandler
fail	FailTagHandler
assertenabled	AssertEnabledTagHandler
asserthasfocus	AssertHasFocusTagHandler
asserttextfieldcontains	AssertTextFieldContainsTagHandler
asserttablecontains	AssertTableContainsTagHandler
noop	NoOpTagHandler
evaluate	EvaluateTagHandler
echo	EchoTagHandler
suite	SuiteTagHandler
test	TestTagHandler
test	TestTagHandler
property	PropertyTagHandler
taghandlers	TagHandlersTagHandler
awteventqueue	AWTEventQueueTagHandler
find	FindTagHandler
key	KeyTagHandler
manager	JFCEventManagerTagHandler
wheel	MouseWheelEventDataTagHandler
click	ClickTagHandler
drag	DragTagHandler
sleep	SleepTagHandler
record	XMLRecorder
save	SaveTagHandler
file	FileTagHandler
dump	DumpTagHandler
for-each	ForeachTagHandler
choose	ChooseTagHandler
indexof	IndexOfTagHandler
procedure	ProcedureTagHandler
while	WhileTagHandler
getparent	ParentInstanceTagHandler

Table 2. JFCUNIT Supported Tag Handlers (continued)

Event Name	***Tag Handler Class***
pathdata	PathTagHandler
tokenize	TokenizeTagHandler
JComboBoxMouseEventData	JComboBoxMouseEventDataTagHandler
JListMouseEventData	JListMouseEventDataTagHandler
JSpinnerMouseEventData	JSpinnerMouseEventDataTagHandler
JTabbedPaneMouseEventData	JTabbedPaneMouseEventDataTagHandler
JTableHeaderMouseEventData	JTableHeaderMouseEventDataTagHandler
JTableMouseEventData	JTableMouseEventDataTagHandler
JTextComponentMouseEventData	JTextComponentMouseEventDataTagHandler
JTreeMouseEventData	JTreeMouseEventDataTagHandler
JMenuMouseEventData	JMenuMouseEventDataTagHandler
MouseEventData	MouseEventDataTagHandler
AbstractButtonFinder	AbstractButtonFinderTagHandler
ComponentFinder	ComponentFinder
JLabelFinder	JLabelFinderTagHandler
JMenuItemFinder	JMenuItemFinderTagHandler
NamedComponentFinder	NamedComponentFinderTagHandler
DialogFinder	DialogFinderTagHandler
FrameFinder	FrameFinderTagHandler
JWindowFinder	JWindowFinderTagHandler
LabeledComponentFinder	LabeledComponentFinderTagHandler
JInternalFrameFinder	JInternalFrameFinderTagHandler
JPopupMenuFinder	JPopupMenuFinderTagHandler

5.3.3. Custom Tag Handlers for the VCell

The JFCUNIT provides easy extensibility for developers to capture and replay a user's custom events. We utilized this feature for VCell to recognize unidentified events by writing tag handlers for each custom event. First, we played the games with the recording mode enabled to identify events that were unrecognized by JFCUNIT. After reviewing the recorded XML test cases, we found that the following components were not recognized

1. AWT buttons and drop-down (combo-box) components on tool panel window in the Electron Transport Chain module's did not recognize for button click and

drop-down change events for the experiment goals ATP Pump Demonstration, Move to ETC Level 0, and Identify ETC Damaged Cell.

2. AWT buttons and drop-down components didn't recognize on tool panel window of photosynthesis level 1 and photosynthesis level 2 experiments from Photosynthesis module.
3. VRML click, rotation, and orientation events inside the VRML game window were not recognized.

For these un-recognized components, we made a solution to create custom JFCUNIT events for each of the experiment's unidentified components. A separate event is created for button clicks and for select items in the combo box of each experiment in module2 and module3. Table 3 shows the list of events we created for the experiments along with the associated tag handlers we constructed to handle the event.

Table 3. JFCUNIT Custom Tag Handlers for VCell

Event Name	***Java Class***
ATPPUMP	ATPPumpButtonTagHandler.java
ETCBUTTONCLICK	ETCButtonTagHandler.java
ETCDMGDCLICK	ETCDamagedCellButtonTagHandler.java
ETCDMGDSUBSTRATES	ETCDamagedCellSelectTagHandler.java
PSLEV1CLICK	PSLev1ButtonTagHandler.java
PSLEV1	PSLev1SelectTagHandler.java
PSLEV2CLICK	PSLev2ButtonTagHandler.java
SIBSTRATEHANDLER	SetSubstrateTagHandler.java
ETCSUBSTRATES	SubstrateButtonTagHandler.java
VRMLCLICK	VRMLButtonTagHandler.java
VRMLORIENTATION	VRMLOrientationTagHandler.java
VRMLPOSITION	VRMLPositionTagHandler.java

Section 5.3.4 explains the code customization for the VCell game modules to support JFCUNIT XML recording and how we added additional code to the VCell Java project to support JFCUNIT recording for unidentified events in each experiment while running the TestXMLRecording.java program. Section 5.3.3.1 shows a sample, recorded XML test case that was generated while running the TestXMLRecording.java program on an experiment in the Electron Transport Chain (ETC) module with the following custom events: <VRML POSITION>, <PUMPHANDLER>, and <ETCBUTTONCLICK>. The TestXMLReplay.java program takes the Section 5.3.3.1 content as input, reads the events in sequence, and calls subsequent tag handlers for each event by looking into the TagMapping.properties file of the JFCUNIT which maintains the list of Java tag-handler classes associated with each event.

(Note: All the custom JFCUNIT tag handlers for the VCell game that are defined in Table 3 are defined in the TagMapping.properties file). For example, the <find> event will execute FindTagHandler.java class to process the <find> element event, and the <VRMLPOSITION> event will execute the Java class VRMLPositionTagHandler.java.

Each custom tag handler processes the event by reading the XML event value and executes the code inside the processElement() method of the tag-handler Java class to apply the event in run time on the VCell applet. The following java program shows the code that is executed for the <ETCBUTTONCLICK> event (as shown in Section 5.3.3.1).

ETCButtonTagHandler.java

```
package edu.nodak.ndsu.games.taghandlers;

import junit.extensions.xml.IXMLTestCase;
import junit.extensions.xml.XMLException;
import junit.extensions.xml.elements.AbstractTagHandler;
import org.w3c.dom.Element;
```

```java
import edu.nodak.ndsu.games.test.TestXMLReplay;
import edu.nodak.ndsu.games.vcell.ETC_0;

public class ETCButtonTagHandler extends AbstractTagHandler {

	public ETCButtonTagHandler(Element element, IXMLTestCase testCase) {
		super(element, testCase);
	}

	public String getButtonText(){
		return getString(BUTTONTEXT);
	}
	public void processElement() throws XMLException {
		validateElement();
		ETC_0 etc =
		TestXMLReplay.repvcellapplet.getExperimentWindow().getETC_0();
		etc.setButtonListeners();
		String buttonText = getButtonText();

		if (buttonText.equalsIgnoreCase(etc.ETC0_SETCOMPS))
			etc.setlistener.setComponentAction();

		else if (buttonText.equalsIgnoreCase(etc.ETC0_RUN))
			etc.runexperimentlistener.setRunAction();

		else if (buttonText.equalsIgnoreCase(etc.ETC0_RESETEXPERIMENT))
			etc.resetexperimentlistener.setResetExperimentAction();

		else if (buttonText.equalsIgnoreCase(etc.ETC0_STOP))
			etc.stopexperimentlistener.setStopAction();

		getXMLTestCase().addProperty(buttonText, BUTTONTEXT);
	}

	public void validateElement(){
		checkElementTagName(ETCBUTTONCLICK);
		checkRequiredAttribute(BUTTONTEXT);
	}
}
```

5.3.3.1. Sample, Recorded XML Test Case

The below XML document shows the auto-generated XML file (saved.xml) while playing the VCell game with the recording mode enabled. These auto-generated XML file

can be executed to simulate the game play to mimic the interactions of the player again on legacy or new VCell system by running the TestXMLReplay.java program with input as the saved.xml file.


```xml
<?xml version="1.0" encoding="UTF-8"?>
<suite name="Recording test suite">
<test debug="true" name="Recording test" robot="true">
<find finder="FrameFinder" id="JFrame1" index="0" operation="equals" title="Connect to Virtual Cell"/>
<find class="javax.swing.JTextField" container="JFrame1" finder="ComponentFinder" id="Component2" index="0" operation="equals"/>
<click index="0" refid="Component2" type="JTextComponentMouseEventData"/>
<sleep duration="300"/> <key refid="Component2" string="santest"/>
<sleep duration="300"/> <key refid="Component2" string="1"/>
<sleep duration="300"/> <key refid="Component2" string="0"/>
<sleep duration="300"/> <key code="9" refid="Component2"/>
<sleep duration="300"/>
<find class="javax.swing.JPasswordField" container="JFrame1" finder="ComponentFinder" id="Component3" index="0" operation="equals"/>
<key refid="Component3" string="santest11"/>
<sleep duration="300"/> <key code="8" refid="Component3"/>
<sleep duration="300"/> <key refid="Component3" string="0"/>
<sleep duration="300"/> <key code="10" refid="Component3"/>
<sleep duration="300"/> <VRMLPOSITION value="0.0,1.81,3.515255"/>
```

```
<sleep duration="300"/> <VRMLPOSITION value="0.0,1.81,3.515255"/>
<sleep duration="300"/> <find finder="FrameFinder" id="JFrame4" index="0"
operation="equals" title="Goal Information"/>
<find class="javax.swing.JButton" container="JFrame4"
finder="NamedComponentFinder" id="Component5" index="0"
name="TEST_INFORMATIONDIALOG_CLOSEBUTTON" operation="equals"/>
<click position="custom" reference="144,8" refid="Component5"
type="MouseEventData"/> <sleep duration="300"/>
<find finder="FrameFinder" id="JFrame6" index="0" operation="equals" title="Virtual
Cell Applet"/>
<find class="javax.swing.JButton" container="JFrame6"
finder="NamedComponentFinder" id="Component7" index="0"
name="TEST_COMMBUTTON" operation="equals"/>
<click position="custom" reference="30,51" refid="Component7"
type="MouseEventData"/> <sleep duration="300"/>
<find finder="FrameFinder" id="JFrame8" index="0" operation="equals" title="Tell Lab
Guy ..."/>
<find class="javax.swing.JList" container="JFrame8" finder="ComponentFinder"
id="Component9" index="0" operation="equals"/>
<click index="8" refid="Component9" type="JListMouseEventData"/>
<sleep duration="300"/>
<find class="javax.swing.JButton" container="JFrame8"
finder="NamedComponentFinder" id="Component10" index="0"
```

```
name="TEST_CMDDLG_DOBUTTON" operation="equals"/>
<click position="custom" reference="19,16" refid="Component10"
type="MouseEventData"/>
&lt;sleep duration="300"/&gt;
&lt;VRMLPOSITION value="4.2536144,-7.814967,11.999736"/&gt;
&lt;sleep duration="300"/&gt;
&lt;PUMPHANDLER PUMPHANDLERTYPE="NADHPumpHandler" index="1"/&gt;
&lt;PUMPHANDLER PUMPHANDLERTYPE="WaterPumpHandler" index="1"/&gt;
&lt;sleep duration="1500"/&gt;
&lt;PUMPHANDLER PUMPHANDLERTYPE="WaterPumpHandler" index="2"/&gt;
&lt;sleep duration="1500"/&gt;
&lt;ETCBUTTONCLICK BUTTONTEXT="Run"/&gt;
&lt;sleep duration="60000"/&gt;
&lt;ETCBUTTONCLICK BUTTONTEXT="Stop"/&gt;
&lt;sleep duration="2500"/&gt;
&lt;ETCBUTTONCLICK BUTTONTEXT="Reset Experiment"/&gt;
&lt;sleep duration="2500"/&gt;
&lt;record encoding="UTF-8" file="saved.xml"/&gt;
&lt;/test&gt;
&lt;/suite&gt;
```

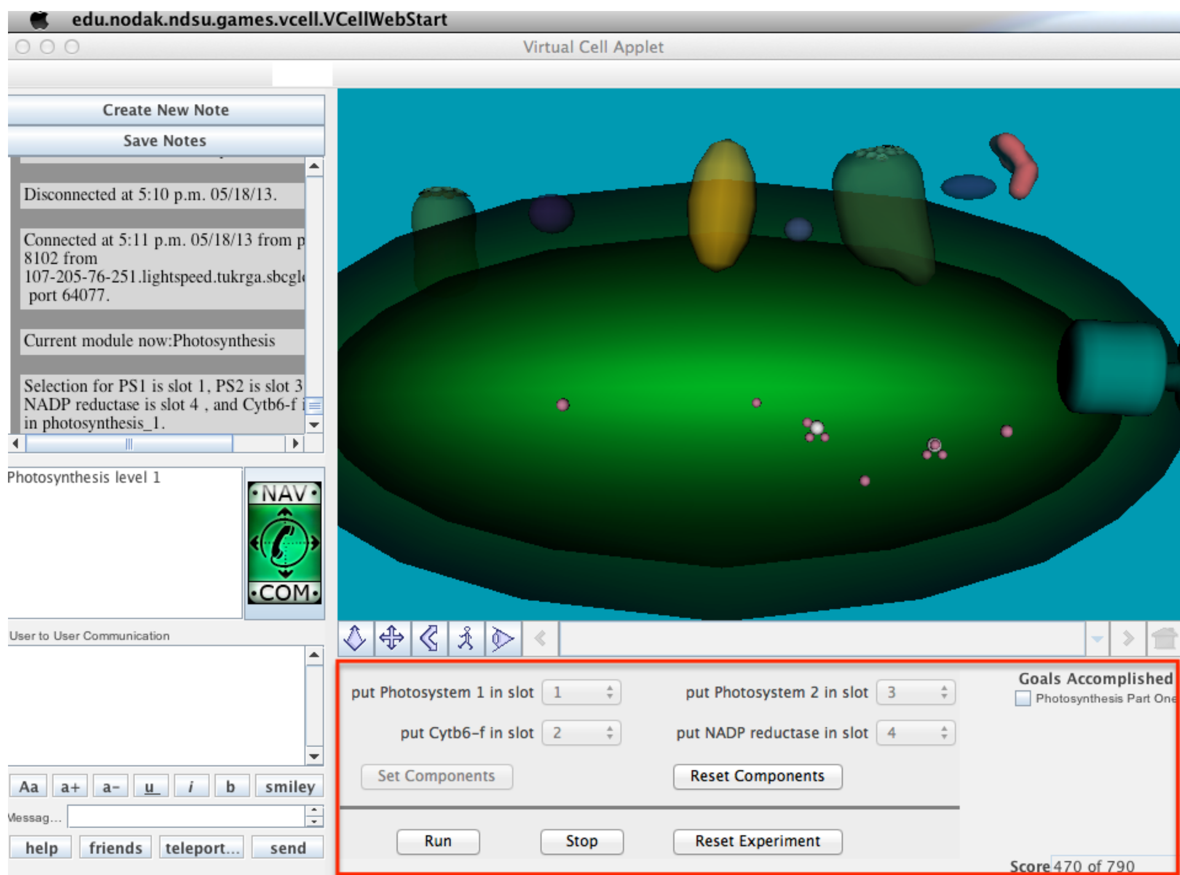

Figure 16. Photosynthesis Module Level 1 Unidentified Components

5.3.4. Code Customization to Add Custom Events to the XML Recorder

For all the unrecognized events in the VCell applet (as shown in Table 3), a process has to be followed to add each event to the XML recording template and to write a tag handler for the replay program to emulate this event. Below we explain the steps for how we processed the button-click event and the combo-box select event for the Photosynthesis Level 1 experiment in the Photosynthesis module, i.e., the PSLEV1CLICK (for the button) and PSLEV1 (for the combo-box) events as shown in the tool-panel components of Figure 16. We followed similar steps to develop a custom JFCUNIT recording customization for each unrecognized component event of every experiment in Electron Transport Chain and Photosynthesis module.

1. Identify the Java class that is invoked for the Photosynthesis Module level 1 experiment tool panel: We identified that the *Photosynthesis_1_1.java* program is invoked for both buttons and the combo-box Java AWT components (i.e., Set Components, Reset Components, Run, Stop, and Reset Experiment buttons in the panel and the four combo-boxes) to set the components for the experiment as shown in Figure 16.

2. Identify the events and tag handlers created for the experiment: This experiment has five buttons and four combo-boxes. We created two custom events, the "PSLEV1CLICK" and "PSLEV1" events, to handle button clicks and combo-box item-change events.

3. Add the identified events to the XMLRecorder.java (of the JFCUNIT API): the XMLRecorder.java provides code logic to add the event to the XML file while running the record program. In this case, we need to include methods to add the "PSLEV1CLICK" and "PSLEV1" events to the XML file. The following code includes a method to generate an XML tag and to insert it into an XML file for the "PSLEVEL1CLICK" and "PSLEV1" events.

```
public static void generatePSLevel1Select(String SlotIndex, String SlotType){
        Element e = m_doc.createElement(PSLEV1);
        e.setAttribute(SLOT, SlotType);
        e.setAttribute(INDEX, SlotIndex);
        insertNode(e);
  }

  public static void generatePSLevel1Click(String ButtonText){
        Element e = m_doc.createElement(PSLEV1CLICK);
        e.setAttribute(BUTTONTEXT, ButtonText);
        insertNode(e);
}
```

4. Identify the listener classes for each of the button and combo-box items in the experiment: We identified nine listener classes inside the Photosynthesis_1_1.java program: "PS1Handler," "PS2Handler," "NADPHandler," and "CytHandler" for the combo-boxes and "setListener," "resetListener," "runExperimentListener," "resetExperimentListener," and "stopExperimentListener" for the buttons.
5. Modify the identified listener classes' set methods to include logic to add the event in the XML file if recording is enabled: For example, the following code shows the modification we did for the setPS1Handler() method of the PS1Handler listener.

```
public void setPS2Handler(){
        int n = PS2.getSelectedIndex();
                if(SocketReaderThread.ISRECORDING){
                        XMLRecorder.generatePSLevel1Select(String.valueOf(n),
                PS1_PHOTOSYSTEM2);
                        XMLRecorder.generateSleep("1500");
                }
        nPS2 = n + 1;
        MyParent.writeLn(
        ";player.location:set(\"PS2Selection" + "\"," + nPS2 + ")\n");
}
```

6. Similarly, follow step 5 to include the generatePSLev1Select or generatePSLev1Button events to the set methods of the identified listener classes: "setNADPHandler()," "setPS1Handler()," "setCytHandler(),"

"setRunExperiment()," "setResetExperiment()," "setListener()," and "setStopExperiment()."

7. Run TestXMLRecording.java on the Photosynthesis Module Level 1 experiment program to test the functionality by checking the recorded XML file to see that all the buttons and combo-box items are added in the XML file: The following example shows the events that are added to the XML file when you click the Set Component button and select the SLOT photosystem1 with item "1."

```
<PSLEV1CLICK BUTTONTEXT="Set Components"/>
<PSLEV1SELECT slot="PHOTOSYSTEM1" index="1"/>
```

8. The TextXMLReplay.java program reads the above-generated XML test-case events and applies the events one-by-one to the VCell applet by executing the corresponding tag handlers: For example, the PSLEV1CLICK event will execute the PSLev1ButtonTagHandler.java class that has logic to call the set listener classes to perform a click or item-select operation based on the BUTTONTEXT for button click events and with SLOT and INDEX for combo-box item select events.

6. TEST RESULTS

6.1. Test Results

Table 4 shows the summary of the capture and replay results and Table 5 shows the total number of tested functions. The “operations common to all modules” had 10 interface mismatches, 1 functional error, and 11 errors. The Find Organelle had 9 interface mismatches, 0 functional errors, and 9 errors while the find defective component had 2 interface mismatches, 0 functional errors, and 3 errors. On the other hand, the Synthesis Defective Identification had 2 mismatches, 0 functional errors, and 3 errors. The ETC/ATP animation and ETC level 0 had 0, 1, and 1 interface mismatches, functional errors, and errors for the former 3, 1, and 4 interface mismatches, functional errors and 4 errors for the latter. The ATC pump demonstration and damaged ETC system had 4, 0, and 4 interface mismatches, functional errors, and errors in the former as well as 2 interface mismatches, 0 functional errors, and 2 errors in the latter. The total figure for the capture-and-replay results was 39 total interface mismatches, 7 functional errors, and 46 other errors. These results were for 3 modules and only included the goals that had detectable errors. From the test results, it is possible to form an opinion concerning the usefulness of automated testing. For example, the conformance of the SUT is measured by considering how to pass many functions without detecting errors in them.

From the literature, the conformance percentage is by an equation: the conformance percentage is the number of conformed functions divided by the total number of tested functions. This function refers to the total sum of all the interface components which undergo testing and the sum of all game logic. The number of conformed is the total number of the tested functions subtracted from the total number of errors that constitute the

total sum of errors found by the test programs, capture and replay. Again, Table 6.2 shows the total tested functions, tested game logic, and the number of tested interface functions obtained from the decision and function trees.

Table 4. Summary of the Capture and Replay Results

Module/Goal	*Interface mismatches*	*Functional errors*	*Errors*
Operations common to all modules	10	1	11
Find Organelle	9	0	9
Find Defected Component	2	0	2
Synthesis Defected Identification	2	0	2
ETC/ATP Animation	0	1	1
ETC Level 0	3	1	4
ATP Pump Demonstration	4	0	4
Damaged ETC system	2	0	2
Photosynthesis Demo	0	1	1
Photosynthesis Level One	1	0	1
Photosynthesis Level Two	6	3	9
Total	39	7	46

6.1.1. Interpretation of Test Results

The data shows (as shown in Table 5) that the total number of tested functions for the entire system stands at 327. On the other hand, the testing methodology's errors before corrective activity stand at 59. From these results, it is possible for one to calculate the initial conformance percentage as 81.9%. That is, (327-59) divided by 327. It is imperative to run the tests again after performing the right actions to prevent them from reducing. This is possible by calculating the functional conformance percentage as follows: (327-13) divided by 327, giving 96% [1].

Table 5 implies that there are advantages to systematic testing because the functional conformance of the SUT rose from 81.9% to 96%. Another thing that is worth noting is that modifying the interactive ETC demonstration goal resulted in the re-run ofthe test cases for the second module. Three errors in the Damaged ETC system, in which

Table 5. Summary of the Total Number of Tested Functions

Module/Goal	*# of interface functions*	*# of game logic*	*# of tested functions*
All Modules	19	24	43
Find Organelle	8	14	22
Find Defected Component	6	14	20
Synthesis Defect Identification	6	14	20
Find Mitochondria	3	6	9
ETC/ATP Animation	1	3	4
ETC Level 0	8	10	18
ATP Pump Demonstration	6	11	17
Interactive ETC Demonstration	6	40	46
Damaged ETC system	36	20	56
Find Chloroplast	3	7	10
Photosynthesis Demo	1	2	3
Photosynthesis Level One	9	10	19
Photosynthesis Level Two	22	18	40
Total	134	193	327

one of them was a game logic error, were detected by the recorded test cases. The results of the process highlighted the importance of being conversant with threats to validity. In this experimental process, these threats allowed to deal with the unexpected impact (such as biased results) of the independent variables on the dependent variables. The threat of biased results is the reason why the process used two techniques: capture and replay as well as the model-based technique. From the testing process, it is clear that the major threats to validity came from implementing the testing techniques and using Java programs. Thus, although the strategy aimed at dealing with complicated event sequences of the VCell, the modifications, or customizations, were independent of the system. The strategy also widened the adaptability of this approach to other varying complex systems.

7. CONCLUSION AND FUTURE WORK

7.1. Conclusion

Testing is an important process by which one can improve software quality by collecting information about the software's performance. Through the JFCUNIT's virtual testing frameworks, the software reliability can significantly improve without resorting to inflexible tools or drivers with high overheads. Scalability and flexibility are, thus, possible using JFCUNIT testing for the desired results. The specification language used in the testing process underscores a number of testing items, namely user-defined testing, structural testing, code testing, and data-flow testing. The specification language has both textual and visual forms to allow it carry out its duties. The graphical tool, for instance, can collect the test results and present them to the end user with a test analyzer, which highlights, or marks, the parts that need rectification. The basic components of a testing framework such as this one include attest analyzers, test planners, test virtual machines, and test specifies.

When testing software, a developer can apply a multitude of different tests to various code regions. The basic application specifies the tests to apply and the appropriate conditions. The test-specification language represents the desired test process and consists of a "Body" and "Definitions." Definitions, as the name suggests, describe the regions of code to undergo testing by declaring code regions for referencing in the Body section. The Definitions has the test specifications, such as the conditions to meet before the test goes through successfully. In the JFCUNIT method, the test planner is invoked or consulted every time the compiler loads a method in order to execute and employ it. The planner is

also responsible for retrieving, from a Java file, the source code to byte code line-number mapping.

Mapping is a strategy employed during the testing process that involves mapping the GUI component names in the database to the GUI component names in the SUT. This mapping of the domain names' logic structure to a database containing test data improves testability. It also demands, in order to facilitate testability, that the graphical user interface components have a given internal name for the JFCUNIT to find them easily. In conclusion, this unit framework provides the developer with a simple set of classes that are suitable for unit testing. The main class of the JUNIT is the virtual framework through which unit tests are run and the results stored. Through a test case, there is implementing for the defined interface, providing methodologies to set up the test conditions, running the desired tests, collecting the results, and then discarding the tests once they are complete. It is important to note that the JUNIT also provides elements such as the GUI class and the command-line class. An example of the command-line class is junit.textui.TestRunner. The actual tests in JUnit, by definition, are {testConcat( )}, {testSubString ( )}, and {testLength ( )}. The test verifies if the conditions of the software's assertion are true, and if so, the JUnit marks the test as positive, or "pass." The JFCUNIT is one that often causes problems for developers who are not keen, for example running the JUnit on the same thread as the application. Therefore, one should always run JUNIT on a separate thread to prevent the tests from running before the Java Swing application commences. If a developer started the Swing application and then started running the test methods, the result would be a complete failure. In addition, a developer cannot locate and manipulate Swing components from outside the desired application although Java automatically enters mouse

events and key strokes. The creation of an additional API, however, can enable locating the components required for the test. JFCUNIT provides an extension to the test case that makes the unit test work efficiently with the AWT thread of the Swing application. JFCUNIT also listens to events happening on the AWT thread while keeping track of all created components and handles any component through the finder methods. Finally, the JFCUNIT has two methods that it employs to enable the AWT process events: awtSleep and awtSleep long time. It is possible to conclude that the JFCUNIT is very dynamic as can be seen from the previous explanations. The testing techniques in use underwent modification and adaption to handle various issues that arose when testing the VCell. The strategy involved the reimplementation of an already functional multi-user computer game that tests the functional conformity of software re-engineering. Three different automated oracles and two different testing techniques were necessary for the process to be successful. It is clear that, as the model's behavior and GUI structures increase in complexity, there is no adverse effect on the decision tree. In fact, one could still use decision tree without additional and unmanageable complexities. This process led to the conclusion that any modifications geared towards the GUI structure or system behavior would not affect derived test cases, decision trees, and function trees.

Researchers [5] are currently looking into the issue of dependency on persistent states as in the case of replaying the test, capture, and relay techniques. One should set the goal and module manually to see to the replay stage success. Although the paper looked into manual setting and automated test programs, it did not cover the capture-and-replay technique due to the requirement to capture and trade information as it entailed modifying the customer and server codes.

7.2. Future Work

In the future, it will be necessary to employ the function and decision trees in the generation of automated tests. It is also possible that people will employ test cases for performance testing and need to run parallel. In the future, test overheads on benchmark parts of the virtualization will be extremely low while the framework, tools, and drivers will be extended to support other test types, such as def-use coverage and statement coverage. Optimized Java code and the latest in this code will also be tested in the future as technology knowhow increases [1]

REFERENCES

[1] El Ariss, Omar, Xu, Dianxiang, Dandey, Santosh, Vender, Bradley, Mcclean, Phillip & Slator, Brian: Conformance Testing for Reengineering the VCell Game. Submitted journal for review IEEE Transactions on Systems, Man and Cybernatics Part C:

[2] Memon, Anthony & Pollack, Mike. Using a Goal Driven Approach to Generate Test Cases for GUI's. Proceedings of the 21st International Conference on Software Engineering. New York: ACM Press, 2009. Print.

[3] Paiva, Antoine & Tillman, Nicholas. Towards the Integration of Visual and Formula Models for GUI Testing. Electrical Engineering Theory Notes

[4] Quan, Xie & Memon, Antoine. Using a Pilot Study to Derive a GUI Model for Automated Testing ACM Trans on Software Engineering and Methodology. New York: Phalanx Press, 2008. Print.

[5] Sara Sprenkle, Emily Gibson, Sreedevi Sampath, and Lori Pollock. Automated Replay and Failure Detection for Web Applications. Automated Software Engineering (ACE)'05 Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering

[6] Colin Bird, Andrew Sermon: An XML-based approach to automated software testing. ACM SIGSOFT Software Engineering Notes 26(2): 64-65 (2001)

[7] Xeena XML editor, http://www.alphaworks.ibm.cem/tech/xeena

[8] Reinventing Education, Wired for Learning, http://www.ibm.comlibmlibmg;lveslgrantleducationlprogramslreinventing/wfl.html

[9] Omar el Ariss, Dianxiang Xu, Santosh Dandey, Bradley Vender, Philip E. McClean, Brian M. Slator: A Systematic Capture and Replay Strategy for Testing Complex GUI Based Java Applications. ITNG 2010: 1038-1043