MODEL-BASED EXPLORATORY TESTING: A CONTROLLED EXPERIMENT

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

By

Christopher Joseph Schaefer

In Partial Fulfillment
for the Degree of
MASTER OF SCIENCE

Major Program:
Computer Science

March 2013

Fargo, North Dakota

North Dakota State University
Graduate School

Title

Model-based Exploratory Testing: A Controlled Experiment

By

Christopher Joseph Schaefer

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. Hyunsook Do
Chair

Dr. Samee Khan

Dr. Kenneth Magel

Approved:

March 11, 2013	Dr. Brian Slator
Date	Department Chair

ABSTRACT

Testing methods of software systems have been receiving more and more focus in recent years. Exploratory Testing (ET) is one such testing method. The advantages of ET are generally outweighed by its disadvantages, mainly the time necessary to perform it. Multiple test automation methods and frameworks offer time saving benefits, in generally what is already a limited amount of time in the development process. In this study we propose Model-based Exploratory Testing (MBET), an approach that incorporates the advantages of ET and the time saving automation of Model-based Testing (MBT). To assess our approach, we conducted a controlled experiment focusing on the number, severity, and type of defects that were detected by MBT and MBET. Our results show that the defects detected by these two testing methods are complementary, and can generate a test suite that provides advantages of both ET and automated testing methods.

ACKNOWLEDGEMENTS

I would like to thank my advisory committee members and professors at North Dakota State University (NDSU), Dr. Hyunsook Do, Dr. Samee Khan, and Dr. Kenneth Magel for their valuable support and guidance. Dr. Do, as my advisory committee chair, helped me out when I found myself in a potential dead end. She made time in providing technical expertise and guidance when the sky was falling around me. I would not have been able to complete this project without her support.

I would also like to thank the Computer Science staff at NDSU for helping me complete all the paperwork, a second time in some cases, when things began to get complicated. I would like to thank my family and friends for their never ending support throughout my entire educational career.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF APPENDIX TABLES

LIST OF APPENDIX FIGURES

CHAPTER 1. INDRODUCTION

Exploratory testing a continuous process of learning, test design, and test execution against a software system. It is becoming a more desirable testing method as more and more projects begin using agile development processes [1,2,3]. However, exploratory testing is often overlooked as a useful testing method by many project managers since it tends to be more demanding of resource use. Further, exploratory testing is believed to have limited test coverage compared with other methods of testing. Automating exploratory testing can address this problem because automated testing reduces the amount of time spent creating and executing test cases by testers, which saves time and money [4,5,6,7]. Automation can also help improve the coverage of testing with certain automation frameworks.

There are many automation approaches to automate software testing, such as Data-driven testing, Keyword-driven testing, and Model-based testing [8,9]. Model-based Testing (MBT) provides a framework for automating test generation and verification based on a behavioral model of the system under test (SUT) [10,11,12,13]. Using defined standards provided by the Unified Modeling Language (UML), models can be generated from system requirements and an abstract knowledge of how the SUT is supposed function [14,15,16]. Such models can normally be created by the design team prior to the development of the SUT. Test cases can then be derived from paths that traverse this model. Also by using this model to generate test cases, a more complete coverage of the SUT can be achieved [17].

Numerous models can be used for MBT, such as finite state machines, statecharts, and UML [10]. Using a standardized language like UML, research has shown that models that are more defined representations of the SUT can be generated, resulting in better test coverage and test cases [11,17]. UML state machines and sequence diagrams are behavioral models of software systems [15,18]. UML state machines represent software systems by the states the system can exist in and the transitions between these states. However, guarded state machine transitions do pose problems with model based test cases. The transitions from one state to another state (or itself) can contain a guard, which is used to determine whether the transition has been traversed successfully and that the system is now in the target state [15]. Since a guarded transition is dependent on run-time values, MBT can generate infeasible test cases. Such test cases could be useful in some cases, however, if such guards could be factored into test case generation more comprehensive testing could be achieved.

In this research, we propose Model-based Exploratory Testing (MBET) to address the automation and infeasible test issues that have been stated. The implementation of MBT to automate portions of exploratory testing could prove extremely useful in saving time, money, labor, and other resources during the testing process. Automating the generation of test cases at run-time using a state machine model, results in a process that can be described as the automation of exploratory testing. The resulting test cases would be more realistic representations of an actual user interacting with the system at run-time. These cases could be compared with those generated from standard MBT methods, which as stated previously could be rather limited or infeasible representations of interactions with the system under test. It would be reasonable to suggest that test cases generated by a MBET process would be more significant in terms of representing realistic use of the SUT than those generated from MBT methods. This project is not recommending that the entire process of exploratory testing can be automated, but if a majority of the process can be, then exploratory testing methods might become more desirable for testing software systems.

To evaluate MBET, a controlled experiment was conducted on a software system that was currently being developed. This evaluation was done by comparing the number and type of defects found using test cases generated by using a MBET process and those found using a basic MBT process. The results from this experiment show that overall MBET detected more defects than MBT. MBET detected certain types of defects better than MBT, while MBT detected other types better than MBET.

The remaining portion of this thesis is structured as follows. Section 2 reviews existing research on automated testing, model-based testing, and exploratory testing. Section 3 presents the MBET process and the tool used to apply it. Section 4 provides an outline for the study including how the experiment was prepared and executed along with threats to validity and the collected data and results of the experiment. Section 5 discusses the results from Section 4. Finally, in Section 6 the conclusions and outline of possible future work are presented.

CHAPTER 2. BACKGROUND & RELATED WORK

This section of the thesis will supply background information about automated software testing, model-based software testing, exploratory testing and other related work.

2.1. Automated Software Testing

Automated software testing has become extremely useful for software testing. Automated software testing methods can help reduce the time, labor, and cost of software testing [4,5,6,7]. Commonly, testers are responsible for creating these test cases manually, which is time consuming and labor demanding. By acquiring or developing a testing tool that can automate the generation process, time spent generating these test cases can be vastly reduced. Also, the execution of the test cases can be expensive or extremely difficult when handled manually by a tester. By automating the execution process, test cases can be executed in large numbers and without tireless effort by a tester.

Automated software testing has allowed testers to increase their productivity and improve the systems they test. Software testing and quality assurance generally account for a majority of a project’s total costs [19]. This along with the all too common shorter schedules for testing during software development often lends itself to automating software testing. However, sometimes due to costs of purchasing an automated testing tool or developing one, automated software testing can fall short of its expectations [4]. Generally, with structured planning and analysis such issues can be avoided. Automated software testing has become a main staple of software testing processes in almost all projects today [8]. There are many different approaches to automating software testing, each providing their own advantages to the testing process.

Data-driven testing relies on data driven scripts and is usually the quickest and easiest to implement [8,9]. However, it generally leads to long-term failure since the staff required to maintain the tool and test cases becomes difficult. Keyword-driven testing offers the ability to insulate the tests from automation tool shortcomings and modifications to the system under test (SUT) [8]. Such testing allows simple keywords to be used as input for test cases, allowing them to be easily written and understood. Keyword-driven testing is extremely difficult to implement and can cost an enormous amount of time and effort to do so.

2.2. Model-based Software Testing

Model-based testing (MBT) focuses on using models of the SUT to guide the testing process [8,9,10,20]. MBT has gained more popularity due to standards specified by UML and the need to test the more complex systems being developed today [13]. These models can take many forms, but they represent the behavior of a software system [10]. These models can be used to generate test cases, or validate the results of a test case [12,14]. These models offer a major advantage of displaying a simple representation of the system that can be easier to understand than lengthy documentation. MBT can help address problems with expenses, lack of methodology and discipline [21].

To develop these models, a tester needs a high level of understanding of the system and its environment [10]. Once they have acquired this knowledge, they will then need to decide on the type of model that will best represent the system and build the model [14]. However, time can be saved if such a model is created by the design team prior to development of the SUT. Then, using the model, test cases can be generated, executed, and the results collected using automation methods [12,20]. The test results can be evaluated by comparing what the model specifies and what the software actually does. Evaluating the results of MBT can rely on verifying whether the proper state has been reached for example.

MBT offers an excellent guide for software testing. Many different languages can be used for MBT depending on the type of system that is being tested [10]. However, UML models are becoming more and more popular as the preferred language. For this project, UML finite state machines are selected as the type of model to represent the SUT [15,18].

UML has standardized multiple types of models to represent software systems over the years. One of these is the finite state machine model, or simply as we refer to it state machine [15,18,19]. A state machine model is a representation of the behavior of a system. Each state in the model represents a specific state the system can exist in at a time. Transitions in the model represent the movement of the system from one state to another state provided some action. Generally, these transitions include some type of trigger(s) and guard, and sometimes they include a behavioral expression.

The trigger(s) usually specify some type of event(s) that induces the transition from the state to another state (or back to itself) [15,19]. These events fire for the transition and if a guard exists, the guard is checked for completion of the transition. This guard is generally written in terms of a triggering event's parameters. A behavioral expression can also be included, which briefly describes what happens when a transition is traversed.

Also, included in a state machine is an initial and final state. The initial state is the entrance into the state machine, while the final state is the completion of the state machine. Each state may contain other states and transitions, and is called a composite state, with the internal states called substates. A simple example of this case is when a composite state can be broken down even further into multiple substates which are all directly related to the parent state.

2.3. Exploratory Testing

Most testers have performed exploratory testing without even knowing it [1,2,3]. Exploratory testing can be summarized to a simple sentence provided by James Bach [3],

"Exploratory testing is simultaneous learning, test design and test execution."

The process of exploratory testing involves a cycle of test design and execution in which a tester is constantly learning and adapting their design and execution from the collected results. Therefore, many testers that execute a test case, report a bug, and then re-execute the test case and variations of that test case to verify the bug has been fixed are performing a form of exploratory testing.

However, many organizations and companies do not look favorably on exploratory testing due to some of its drawbacks. For example, it is considered to be more time and resource demanding than scripted testing [1,2,3]. This extension of time and resource allocation can be problematic in projects since many organizations do not tend to assign the appropriate amount of time and resources for test execution in the first place. Moreover, exploratory testing tends to produce less test documentation prior to test execution. Such documentation is generally used by many organizations for visibility and transparency before test execution [1,2]. Documentation and test case generation can also be used as metrics to monitor the performance and progress of testing teams. Since many organizations tend to prefer these metrics, the lack of initial documentation and test cases from using exploratory testing tends to make it undesirable.

Automating exploratory testing is a controversial topic, because most research and discussion agree that effective exploratory testing cannot be fully automated [1,3]. However, some do believe exploratory testing can benefit from being automated, at least partially [1]. By using a structured approach or method, exploratory testing could reap the benefits of automated software testing. Such success could improve the view of exploratory testing in the software development process and testing community.

CHAPTER 3. MODEL-BASED EXPLORATORY TESTING APPROACH

To support a Model-based Exploratory Testing (MBET) approach, we build an automated testing tool, the Crushinator. This section provides an overview of the project's previous history and how the framework was implemented for MBET. This section also provides a simple example to explain this process in greater detail. An explanation of the transformation of UML for MBET is also included in this section.

3.1. The Crushinator

The original Crushinator (Crushinator version 1, Cv1) [22], was designed to be a testing tool for multiple event-driven client-server games. It was intended to be a multi-threaded application that could run automated tests on such games. However, Cv1 did not end up meeting the specified requirements.

The new Crushinator project (Crushinator version 2, Cv2), is a major refactoring of the original Crushinator. The Cv2 provides a game-independent, model-based, automated testing tool framework for any event-driven client-server game. The Cv2 was designed to implement multi-threading, automated software testing, MBT, load testing, and MBET.

The first portion of the project was to refactor and implement a model-based game-independent testing tool framework. Once this was achieved, the Cv2 framework was then implemented to include a simplified load testing graphical user interface (GUI) and MBET GUI along with the supporting backend framework. This framework was then used to develop a testing tool for the Virtual Cell game under development by the WoWiWe Instruction Co [23]. The testing of the Virtual Cell game is used to generate the data and results for this thesis.

The Cv2 was designed as a layer architectural system, as seen in Figure 3.1. This allows a layer to be replaced at anytime and not affect the system as a whole. The modules inside the hash boarder are dependent upon the system under test (SUT), and the "Test Engine" and "Simulated Client" modules are implemented to be used directly on the SUT.

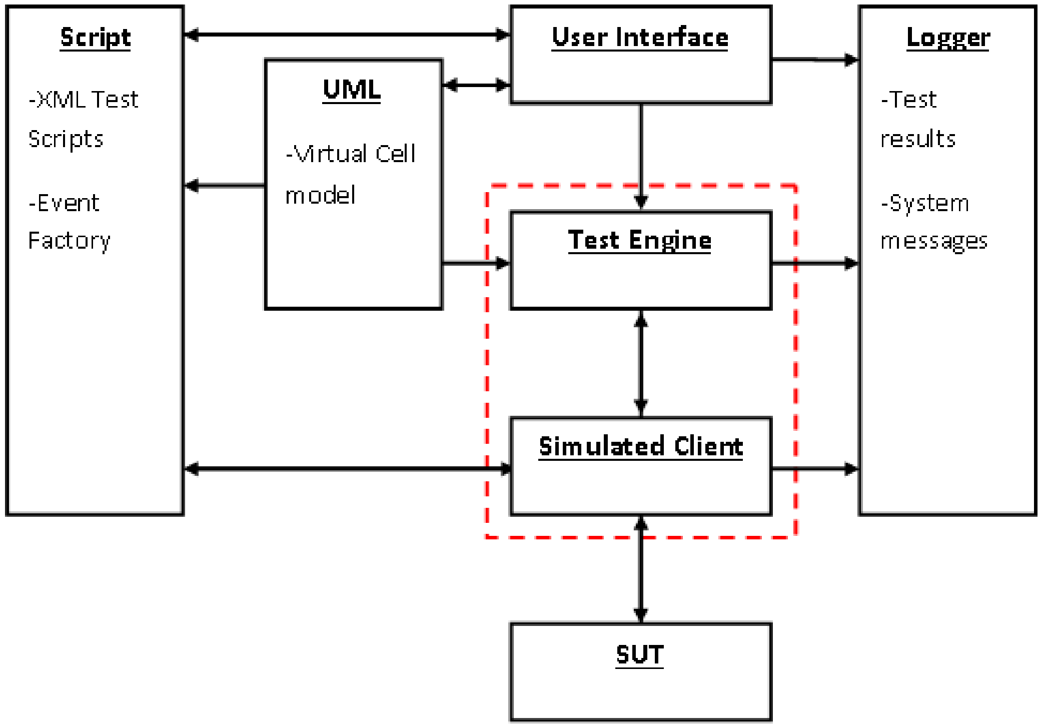


Figure 3.1. Crushinator version 2 architectural diagram.

The "User Interface" module manages the interactions of a tester and passes those requests down to the "Test Engine" module. The "Test Engine" module is responsible for managing and executing tests against the SUT. The "Test Engine" module handles the creation of clients and passing of test cases from the "User Interface" to the "Simulated Client" module that interacts directly with the SUT. The "Simulated Client" module is primarily responsible for interacting with the SUT. This module takes the test case from the “User Interface” module and sends the instantiated events to the SUT.

Along with these layered modules are three assisting modules used for handling a majority of the work associated with MBT and serializing test information. The “UML” module is responsible for generating a representation of a state machine model from an XML Metadata Interchange (XMI) [24] file (such as a model of the Virtual Cell server). It is used by the “User Interface” module to allow a tester to select models to generate MBT test cases. It is also used by the “Test Engine” module at run-time for MBET test case generation.

The “Script” module is used to represent events, known as “Script Events,” for the SUT in an abstract format to allow the “User Interface” to remain abstracted from the SUT. These “Script Event” objects are used to pass the type of event along with its parameters from an XML test case file format through the “Test Engine” to the game dependent “Simulated Client” module. There the information is handed off to the “Event Factory” object in the “Script” module that turns the information into a SUT dependent event to send to the SUT. The “Logger” module is used throughout the layered modules to serialize test results, configurations, warnings, etc. that occur during testing.

The Cv2 communicates directly with a game server, using a specified connection type, to send game based events to the server and retrieve the responses from that server if necessary. The responses can then be used to collect results, update values, or parsed for data to send back to the server. The Cv2 project allows for multiple connections types, such as RMI and socket connections for example, to be used inside the framework.

The Cv2 also generates and utilizes XML files as test cases that contain game events and their parameters, which can be sent to a game server. These XML files can be generated from multiple sources, such as manually selecting game events and saving the test case using the Cv2 GUI, generating test cases from model based state machines, using an implemented game client to generate test cases from actual game play, or from using MBET.

3.2. MBET Process

We now describe the MBET process as it is applied to the SUT in this study. MBET can be applied to any event-driven system that can be represented using a UML state machine. The Crushinator testing tool framework described in Section 3.1 is an example of applying MBET to test multiple systems.

The following steps are then taken to apply the MBET process:

1) Create a state machine model that is a representation of the SUT.
2) Export the state machine model into XMI which can then be used by the Crushinator
3) Use the Crushinator to connect to the SUT
4) Execute an initial test script (optional)
5) Execute the trigger(s) or event(s) for an available outgoing transition from the current state of the SUT
6) Check the run-time dependent transition guards to verify traversal of a transition
7) Calculate available outgoing transitions from the current state
8) Continue the automated steps 5-7 until a pre-determined termination point, selected by a tester, is reached (e.g. completion of the game, time limit, execute a loop of transition(s))
9) Output the series of transitions, along with their trigger(s) and guard values into a test case that can be used for future testing

The first step in the process of MBET is to create a state machine model that represents the SUT. An example model shown in Figure 3.2 is a small state machine model that is a subset of the state machine of the Virtual Cell game server used in the experiment described in the study section of this paper (Section 4). State machines models of the Virtual Cell server can be found in their entirety in Appendix A. This subset model has been slightly modified to include an initial and final state to conform to a standard state machine model.

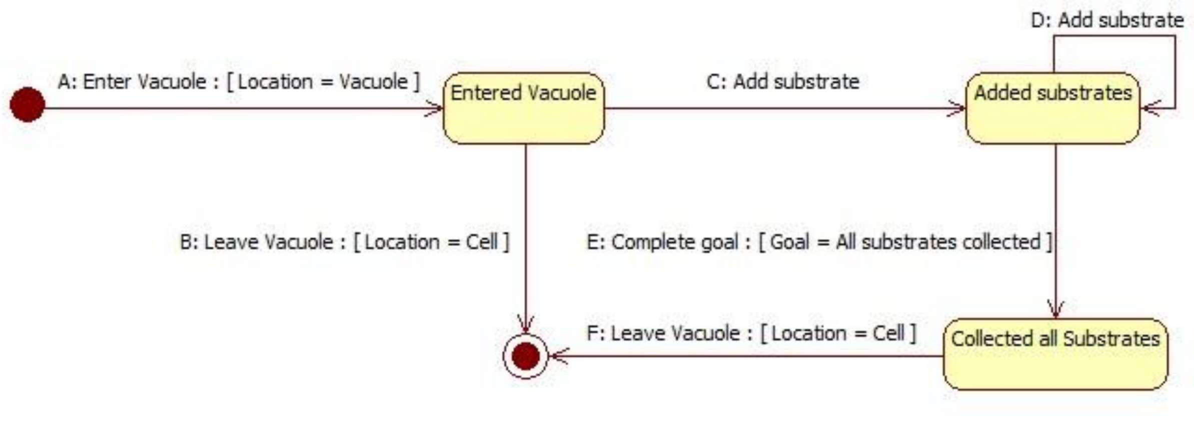

A: Enter Vacuole : [Location = Vacuole]
Entered Vacuole
C: Add substrate
Added substrates
D: Add substrate
B: Leave Vacuole : [Location = Cell]
E: Complete goal : [Goal = All substrates collected]
F: Leave Vacuole : [Location = Cell]
Collected all Substrates

Figure 3.2. Virtual Cell state machine subset.

The transitions of the model contain information necessary to represent the SUT. In this example the transition label “A: Enter Vacuole : [Location = Vacuole]” is a simplified version of that found in the models used in this study, in Appendix A, supplied for explanation of this example. The transition name “A,” used to uniquely identify each transition, is supplied along with the transition’s trigger “Enter Vacuole”, while the guard for this transition “Location = Vacuole” is structured to represent that change of a player’s location that must occur for the transition to be successfully traversed (the player’s location must be inside the vacuole).

To simplify the models used in this study, data is abstracted from the model and placed into configuration files, which is explained in Section 3.4. Abstracting this data, such as transition trigger(s) and guards, allows the model to retain its readability while fully representing the SUT. Also, when changes to the SUT are made during an agile development process, simple changes to triggers and guards can be made to the configuration file rather than the need to modify and re-serialize the model to XMI.

Once a model of the SUT has been generated, the model then needs to be serialized into XMI, a format of XML used to represent models. In this work, the UML modeling tool StarUML [25] is used to generate the state machine model, which can then be serialized to XMI. Many different modeling tools were examined; however StarUML was selected since it was open source software and had the feature to export models to XMI.

The Crushinator is there after used to connect and interact with the SUT. The tool should not only be able to communicate with the SUT, but should also be able to instantiate objects for communication reflectively if needed. The Crushinator automates this process. This framework provided the basic features for connecting to the SUT, instantiating objects reflectively, and managing the XMI files generated by StarUML.

Once connected to the SUT, using the Crushinator, the series of events specified in the initial test case (if one is supplied) are applied to the SUT. Once this is complete, available outgoing transitions are calculated from the current state of the SUT specified by the state machine model. From those transitions, one transition is selected and the trigger(s) for that transition are applied to the SUT. Selection of a transition from the available transitions can be done in multiple ways. For instance, assigning the transitions with a heuristic value can make one transition more desirable for selection over other transitions. The Crushinator sorts the transitions by these values, with transitions with smaller values being more desirable (e.g. a transition with a weight of zero is selected over a transition with a weight of five). This process is explained later in Section 3.4.3. Using a random selection from the available transitions can also be applied to this process.

After the trigger(s) of the selected transition have been applied to the SUT the most important part of MBET is performed; checking the run-time dependent guard of the transition. While MBT follows the steps explained above, this critical step is what differentiates MBET from standard MBT methods. The ability to check whether a transition's guard has been successfully fulfilled during test execution allows the current test to verify the traversal of a transition and movement of the SUT from one state to another. Since these tests rely on run-time dependent values to verify the current state of the SUT, they are more interactive than standard MBT methods.

MBT generates test cases prior to test execution. Therefore it can generate possible infeasible test cases. For instance, if a guarded transition that depends on run-time dependent values exists in the state machine model, MBT will not be able to determine whether the guard is fulfilled and therefore whether that transition was successfully traversed. These infeasible test cases can be used to verify that the model behavior conforms to the SUT. But if we do not consider these guards, we might not detect defects that reside in the SUT.

Using the guard to determine whether the current state of the SUT has changed, the available outgoing transitions are then again calculated for the new current state. Steps 5 through 7 are then repeated until a test termination point is reached. If however, the transition guard prevents the transition from being successfully traversed, the remaining transitions for the current state are used for the transition selection step. This means that all outgoing transitions for a specific state are attempted until a transition is successfully traversed. The following pseudo code explains the process in more detail.

```
availableTransitions = currentState.getOutgoingTransitions();
sortedTransitions = sortTransitionsByWeight(availableTransitions); //descending order
foreach (Transition t in sortedTransitions)
    applyTransitionTriggers(t.getTriggers());
    if (checkTransitionGuard(t.getGuard()))
        currentState = t.getTargetState();
        testPath.add(t);
        return;
    else
        continue;
```

If all of the outgoing transitions have been tried for the current state and none are successfully traversed, the test can then terminate or the current state can then be recalculated using the state machine model. The current state is recalculated by selecting the initial state from the model, and then selecting the states targeted by transitions whose goal-based guards, guards based on game goals and tasks for a player, have successfully passed. Using the selected states, the transition guards are again checked for those dependent on a location-based guard, a guard based on the current game room a player is located in, to select the state the successfully guarded transition targets. These guard types have been specified for the testing of the Virtual Cell game server directly and are part of the implementation added for testing Virtual Cell.

When a termination point has been reached, the test terminates and the path traversed through the state machine model is output to an XML file. This path contains all the data from the triggers of the transitions that have been attempted during the test. This file can then be used to re-execute the test using the Crushinator which generated it.

3.3. MBET Example

To explain the process of MBET in greater detail, this section will provide a simple example. This example will use the state machine subset from the actual Virtual Cell game supplied above in Figure 3.2. The Crushinator testing tool is used to execute the MBET for a player against the Virtual Cell game server.

The Crushinator is used to connect to the Virtual Cell game server, and a player is setup to be at the initial state of the state machine subset. Once a player has been successfully connected to the game server and logged in, and then the player is at the initial state of the state machine, the Crushinator will select a transition from the available outgoing transitions. In this example the only available transition is “A: Enter Vacuole”, which means the player then sends the trigger(s) (event(s) from this point on) for that transition. After the event(s) have been sent for the transition, the guard for the transition is then checked, in this case “Location = Vacuole”. If the guard fails for the transition, it is possible that the test is over since no other transitions are available to try (dependent on the test termination settings). This test may mean that a defect exists with the SUT since the player cannot traverse any available transitions.

If the guard does pass successfully, that means the player is now in the transition’s target state, “Entered Vacuole”. With the player in a new state, a transition is then selected from the available outgoing transitions of the new state (transitions “B: Leave Vacuole” and “C: Add substrate”). Again, the event(s) for the transition are sent to the server and if applicable, the guard for the transition is checked. If the “C: Add substrate” transition is attempted, the state should now be the “Added substrate” state since there is no guard for the transition. If the “B: Leave Vacuole” transition is attempted and the guard succeeds, the test is terminated since the final state has been reached. Otherwise, if the guard failed, the “C: Add substrate” transition can then be attempted.

A MBT method would not check these guards during execution and instead would simply send the events for a path through the model. So even though a MBT path would traverse the transitions “A: Enter Vacuole,” “B: Leave Vacuole” without checking guard conditions, the transition guard for “B: Leave Vacuole” could prevent the actual state of a player from reaching the final state, an infeasible path. When we run this test generated from MBT, the test wouldn’t exercise the system correctly. MBET would attempt transition “B: Leave Vacuole,” check the guard for the transition and then adjust the player’s actual current state if it changed. This allows a player’s current state to be updated at run-time, verifying that a player has actually changed states when a transition is traversed.

This process of selecting and attempting transitions and verifying the guards of the attempted transition, if applicable, continues until a termination point is reached. In this case, it can be when the final state of the state machine is reached, or if no transitions can be successfully traversed out of the current state. A key point that should be made is the possibility of traversing loops in the state machine. For instance, the “D: Add substrate” transition could generate an infinite traversal loop and the need for a termination point such as a time limit, loop iteration count, or others will be necessary. The Crushinator can then serialize the series of events that were sent to the server, allowing the test to be re-executed at any time.

3.4. Transforming UML for MBET

To simplify the UML state machines models into a format that fully represents the SUT while remaining human readable, we transformed the UML model. This kept the model as simple as possible while retaining a complete representation of the SUT. The Object Constraint Language (OCL) [26] was designed to allow UML to represent complex systems, but OCL does not address our issue of simplifying the state machine models and still retain a full representation of the SUT. Our implementation of UML for MBET consists of three components. The first one is the use of state machine keywords to help keep the state machine model as simple as possible by abstracting certain data and information from the triggers and guards of the model's transitions. The second one is removing the lengthy values and data contained in the triggers and guards of the model and replacing them with shorter unique labels that can be used to extract the data from an external configuration file. The last one is for the weighting of transitions from the state machine model. The following subsections provide the details of these three components.

3.4.1. State Machine Keywords

Generating a state machine model of the SUT posed some serious issues. Due to the openness of the SUT design, the structure of the state machine model became too complex. The ability of the SUT to change from one state to another state through numerous similar transitions rendered the model nearly ineffective as a representation. Since these transitions contained the same series of trigger(s) with slightly different parameters and the same guard, these transitions were reduced to a single transition. However, in order to symbolize the different parameter values for the transition's trigger(s), a system of keywords was introduced to retain the full representation of the SUT.

These keywords, referenced to as State Machine Keywords (SMK) for the remainder of this thesis, allow multiple transitions that are exactly the same besides a single trigger parameter to be simplified. By replacing this parameter with a SMK, the model can be reduced dramatically. For instance, instead of a model requiring 6 separate transitions that are almost exactly the same besides a single trigger parameter, the use of SMK replaces all of these transitions with a single transition. These SMK's are then stored in a single configuration file, which allows the testing tool to select one of these values at random to use as the trigger's parameter. This functionality retains the fully representation of the SUT while keeping the state machine model as simple as possible. This configuration file also allows the tester to modify these SMK values in between tests, giving the tester greater influence in MBET as well.

3.4.2 Transition Triggers & Guards

As the state machine model of the SUT was created, it became evident that placing all the information for the transition triggers and guards into the diagram would not be beneficial. The idea of simplifying the representation of a system by using a diagram is nullified when the diagram becomes congested with data and text. To prevent this issue, this information was removed from the diagram. Rather than placing the complex and lengthy trigger data and guards directly into the state machine diagram, short unique labels were inserted. These labels allowed the diagram to remain simple, but also contain references to all the necessary information. These labels can then be used to retrieve the proper data from the matching configuration file.

Besides keeping the diagrams as simple as possible, the configuration files serve another purpose. Whenever a change was made to the triggers or guards of the SUT, instead of having to modify the entire model, just the configuration files would need changes. The process of using the GUI supplied by StarUML to modify a transition's triggers was lengthy and cumbersome, while editing a simple XML configuration file was drastically easier.

3.4.3. Transition Weights

When a state has multiple outgoing transitions, one needs to be selected using some type of criteria. In this case, a weighting system, based on values stored in a configuration file, was applied to provide options to select one transition over the other available outgoing transitions. This weighting system allows a tester to influence the selection of one transition out of the available transitions during MBET. The tester is able to modify the weight of the transitions in between tests to adjust how the transitions are selected. These weights can be adjusted when a transition has been successfully traversed by the testing tool at run-time as well. This would allow the test to select a transition based on its weight the first time, then when that same state is reached a second time, select a different transition since the first transition's weight was adjusted after it was traversed. These weighted values can then be serialized back to the configuration file, allowing the tester a quick reference to the transitions that were traversed the most during the test. Using this serialized data, a tester can adjust these weights for the next test iteration, learning from the previous test execution to influence the next series of tests.

CHAPTER 4. EMPIRICAL STUDY

To investigate the effectiveness of the MBET approach we described in Section 3, we conducted a controlled experiment considering the following research question:

RQ: "Can a MBET approach improve the effectiveness of testing compared to MBT in terms of the number of defects that are detected?"

In addition to this research question, we further examine whether the two testing techniques detect different types of defects.

The following subsections describe the objects of analysis, independent variables, dependent variables and measures, experiment design, threats to validity, and data analysis. We discuss further implications of the data and results in Section 5.

4.1. Objects of Analysis

In this experiment, we used the Virtual Cell server as our object of analysis. The Virtual Cell game is an event-driven client-server educational game developed by WoWiWe Instruction Co. Two versions of the Virtual Cell game exist, an old version (version 1) [27] and a new version (version 2) [28]. The old version of Virtual Cell was developed in the late 1990's and early 2000's and was written in Java. It provided a three dimensional environment for students to explore and learn about cellular biology. It was made up of three separate modules that contained a combined 306 classes, consisting of 35,435 lines of code.

A new version is currently under development to update and add new features to the game. The new server is being developed in Java, while the new client is being developed in C#. The new server is made up of four separate modules that contain a total of 516 classes, consisting of 72,142 lines of code. The development team is made up of 11 members in 4 different teams: server, client, graphics, and testing. The server development team consists of 3 developers, the client team is made up of 4 developers, the graphics team consists of 2 artists, and the testing team consists of 2 testers. Although the Crushinator version 2 (Cv2) was instrumented for both versions, the main focus is on the new version currently being developed to aid in testing the project. The primary focus on Virtual Cell for this experiment is testing on the game server. In this experiment, we use the new version of the Virtual Cell game server.

The Virtual Cell game is a graphics heavy 3D virtual environment to teach users about cellular biology. Virtual Cell currently contains three separate modules, or levels, that can be accessed independently. These modules teach users the fundamentals of certain aspects of cellular biology. They include: an Organelle Identification (ID) module for teaching the functions and processes of specific organelles inside of a cell, an Electron Transport Chain (ETC) module for teaching the process of the ETC, and a Photosynthesis module for teaching the process of photosynthesis. These three modules can be modeled and tested independently, allowing separate test iterations to be executed on each.

Since the Virtual Cell application is currently under development, a specific revision of the system had to be selected to execute our experiment on. This meant that a revision of the system was selected where a majority of the project had been configured, but a large amount of defects had not then been addressed. Moreover, because the development of a framework and API alongside the game was also being pursued, the proper revision selection of those portions had to coincide with the Virtual Cell revision. The framework and Application Programming Interface (API) were provided to build a framework for future event-driven client-server game development and to help simplify the project by allowing access to the available game events to the entire project team. The selected revision of each of these separate modules used in our experiment can be found in Appendix B.

4.2. Variables & Measures

Independent Variable – For the purpose of investigating our research question, we control a single independent variable: the method of software testing applied to a system under test (SUT). We will use two different methods of software testing for the experiment: Model-based Testing (MBT; this serves as experimental control) and Model-based Exploratory Testing (MBET). These two software testing methods will be applied to the exact same version of the SUT, containing the same functional defects and bugs.

Dependent Variable & Measures – The dependent variable for this experiment is the number of defects that are detected by the software testing method. The number of defects found by each testing method can be further broken down by the severity of the defect, the type of defects (explained further in Section 4.5), and in which modules the defects occurred.

4.3. Experiment Setup & Procedures

To investigate our research question, we need a system that contains a variety of defects and that can be modeled using a finite state machine. The Virtual Cell application facilitates these needs, in particular, the application contains real defects. Because the Virtual Cell server is an event-driven application, its behavior was easily modeled using a finite state machine. We used UML as a modeling language.

Prior to executing any tests, a few steps had to be taken to prepare for the experiment. First, the selected version of the SUT was installed and started on a Linux based system which provided access to the server from any machine with an internet connection to connect the game server. During the installation of the SUT, a configuration file for the server had to be modified to allow connections to stay open up to 15 minutes without any traffic. This was done to control the issues that can be caused by the limited network speed and connections and to control any problems a system might have managing a large number of threads for a test. The focus of this experiment is to find defects, not to worry about response latency, although that data is available for every test as well. The system specifications of the machine used to run the SUT can be found in Appendix B.

A few guidelines were developed to help keep all test iterations as similar as possible.

1) All test iterations were executed against the same exact revision of the Virtual Cell server.
2) For each test iteration against the Virtual Cell server, the server's IP address and port was entered into the Crushinator's setting controls to allow connection to the server.
3) Whenever a test case is executed a new player(s) is created on the server using a randomized name. This name is up to 30 characters long, and is made up of upper and lower case letters, numbers, and special characters. This prevented 2 players from having the same name during a test, which would cause unpredicted issues.

4) The SUT was re-installed after each test execution to prevent issues from the previous execution from influencing the results of the next execution.
5) Log files were kept separate for each test execution as well to prevent any confusion between tests. These files were created by the Crushinator and the Virtual Cell server, which were then stored and backed up.
6) A test iteration was executed for each of the three modules separately.

The following additional guidelines apply only to MBET test iterations:

7) One hundred players were used for all iterations of MBET, which means 100 separate test cases are generated for each iteration.
8) The duration of each iteration of MBET was set at 5 to 60 minutes (times for each test can be found in Appendix B), after which the test terminated.
9) The transition weights are initially assigned to zero by a tester, and are adjusted at run-time for each successful transition traversal.

With the SUT up and running, the Crushinator testing tool was used to generate test cases for MBT prior to execution. The Crushinator was executed on two different systems due to time availability of the SUT for our experiment (the specifications of both systems can be found in Appendix B). The Crushinator was used to generate the XML test cases for the three different models. Since the SUT can be broken down into three specific modules, or levels, of game play, three different state machine models were generated, one for each module with its own set of configuration files (these files included state machine transition triggers and guards and SMK's).

To reduce the complexity of a single state machine that would represent the game server in its entirety, which would produce a very large and extremely complex state machine, we performed our experiment by testing each of the three modules (levels) separately from the others, The testing of these three modules was executed separately since each module was designed to be independent of the other, allowing them to be accessible in any order. More specifically, they do not require a player to play them sequentially, which allows testing of each module to remain independent from the others. Using these three separate models and configuration files, the Crushinator was used to generate a test suite of XML test cases for each module, shown in Table 4.1. Each XML test case represented a path through the state machine model. It is important to note that since a player can quit the game at any time that player can reach a final state from any other state in the model. This means that a large number of XML test cases are generated for each module.

Module Name	ID	ETC	Photosynthesis
Number of test cases	267	201	451

Table 4.1. MBT generated test cases per module.

Now using the test suite of generated test cases for each module, the Crushinator's command line interface (CLI) was utilized to perform a series of tests on the SUT. The Crushinator's CLI allows a tester to define an entire directory of test cases to execute all at once, allowing the tester to execute multiple test cases using any number of players they wish. Since each player's state on the server is independent of each other, these test cases can be executed in parallel. These tests were repeated a number of times, with slight adjustments being made to configuration files between iterations. If adjustments were made to the triggers and/or guard configuration files, then it became necessary to regenerate the test suite as well. A full list of applied adjustments between test iterations can be seen in Appendix B. For instance, since the server's ability to handle a certain event was not functioning properly, to ignore that defect and search for others, the event had to be removed from certain transitions. Once all the transitions that contained this trigger were modified, any test cases that traversed one of those transitions became outdated. So, the test cases then needed to be regenerated for that module, which could then be executed during the next series of MBT.

After any configurations were made and the test suite was regenerated, if needed, the next series of tests was executed on the re-installed server. After each test was completed, all the log files from both the Crushinator and the Virtual Cell server were collected. The Crushinator files were then parsed by a custom parsing program developed to uniquely format such files and all of the collected data was then stored and backed up for the test iteration.

To execute MBET, no test cases had to be generated prior to execution since they are generated at run-time (MBET allows a tester to generate test cases purely from the MBET process, or supply a test script as the starting point for a MBET test). Each series of MBET was executed for each module of the Virtual Cell server following the steps mentioned in Section 3.2. A state machine model was selected, one model for each game module, and loaded into the Crushinator. For each test, the 100 players were logged into the server and each player began their own MBET process. Once the time limit was reached, the test automatically terminated for all players that were still performing MBET and all the log files were then collected along with the MBET generated test cases. The Crushinator's log files were parsed by the custom tool and all collected data was stored and backed up for the test iteration.

If any changes had to be made to the configuration files (as mentioned in the MBT explanation), they were made and the next set of tests was ready to execute. Since the test cases are generated at run-time, simply updating any configuration files or even the state machine models does not require the need to make any changes to the previously generated test cases manually, like other forms of automated testing.

Once testing was completed for each module using both testing methods, the data was thereafter imported into the spreadsheets. This data is then reviewed to determine what whether the defects were detected, if any. Any found defects are recorded and any necessary changes to the configuration files are calculated. Entry into the spreadsheets allowed for filtering of the log data for certain information making the review of the data quick and easy. The data collected from these tests is presented in Section 4.5 in further detail.

4.4. Threats to Validity

We will now briefly describe the threats to validity to this study. We discuss some of the threats to the construct, internal, and external validity of our study.

4.4.1. Construct Validity

Two threats to construct validity that may arise in our experiment are the severity level assigned to each defect and the type assigned to each defect. When a defect was detected during the experiment, a severity level was assigned to that defect (low, medium, high). We use these severity levels as a measurement of how effective MBET is compared to MBT. To make sure these levels were assigned appropriately, the server development team was consulted on defects that were not easily determined.

The type of defect was also used to measure how effective MBET is compared to MBT. The assignment of a type to each defect was studied carefully, and assigned to how the defect affected or related to the SUT. The four defect types were created only after the experiment was completed and all defects had be detected and described in detail.

4.4.2. Internal Validity

An important portion of MBET that affected our results is the termination point of the MBET process. A player reaching the final state of a state machine model is a great termination point for a test since there are no transitions to other states for a final state. Other methods were implemented to terminate a MBET test in case a final state could not be reached. A simple time limit for the MBET test was implemented for our experiment, but others could also be applied. Transition traversal failures (when a transition cannot be traversed successfully) and performing loops through a state machine model (when the same transition or series of transitions are constantly being traversed) were also implemented for the termination of MBET.

We attempted to simplify the state machine model by using SMK (as mentioned in Section 3.4.1) to replace multiple similar transitions. At run-time these SMK values were replaced at random from values supplied in a configuration file. These values can be controlled by the tester executing MBET in between tests, but the random selection of these values can affect the test results. Dependence on a random number generator to select a value for a SMK could be replaced by a more desirable method, like using a heuristic value to select the SMK replacement.

Transitions were weighted using a simple heuristic value to persuade MBET to select one transition over other transitions. These values can be adjusted in between tests by a tester to affect how MBET selects transitions from one test to the next. These values can also be adjusted at run-time, dependent on whether the transition is traversed successfully. For our experiment, all of the transition weights were given an initial value of zero for all test iterations, transitions with lower value weights were selected over others, and the values were incremented by one for each successful traversal. If multiple test iterations were executed and the results used to adjust the initial weights, MBET could then be influenced to achieve complete transition or state coverage of the state machine model and thereby the SUT.

4.4.3. External Validity

The results from this experiment will not apply to all other systems and testing situations. The Virtual Cell server is a specific type of system, which best fits the process of MBET. The event driven server allows for a behavior model, such as a finite state machine, to easily represent the system. Not every system will be well represented with a behavior model. Such systems would benefit from testing methods other than MBT and MBET.

Our experiment only included a single system that was being tested. Including multiple systems could help us generalize this experiment to other applications, and is planned for future work. Also, the results generated from the type of system we tested, event-driven, is difficult to generalize to other types of applications.

4.5. Data & Analysis

This section contains an overview of the data that was collected from the experiment. This data is represented in Tables 4.2 through 4.7 and in Figures 4.1 through 4.5. We present the data as it applies to our research question, followed by the data that applies to our further interest in detecting different defect types.

Once all the test iterations were completed and the data compiled, the number of defects detected by each testing method was calculated, shown in Figure 4.2. The two methods were able to detect some similar defects, but each also found some unique defects that were not detected by the other method.

	MBT	MBET
Number detected	12	21

Table 4.2. Defects detected by testing method.

These defects can be broken down by which game modules they were detected in, with each module in a different development status. For instance, the Organelle Identification (ID) module was feature complete, and had been strenuously tested during it development. The Electron Transport Chain (ETC) module was nearing development completion, but still required some features to finish development. The Photosynthesis module had just recently started development, with the structure of several features just finished, or currently in process. The three separate modules allow us to examine how effect each testing method is at different stages in the development process. The breakdown of the detected defects by module can be seen in Table 4.3.

	ID		ETC		Photosynthesis	
Method	MBT	MBET	MBT	MBET	MBT	MBET
Number detected	3	8	8	6	1	7

Table 4.3. Defects detected by method for each module.

To show our results visually, we present them in a bar graph, as shown in Figure 4.1. Overall MBET was more effective in detecting defects than MBT (21 vs. 12). MBT was slightly more effective in detecting defects in the ETC module (8 vs. 6), but significantly less effective in detecting defects in the other two modules (3 vs. 8 in ID, and 1 vs. 7 in Photosynthesis).

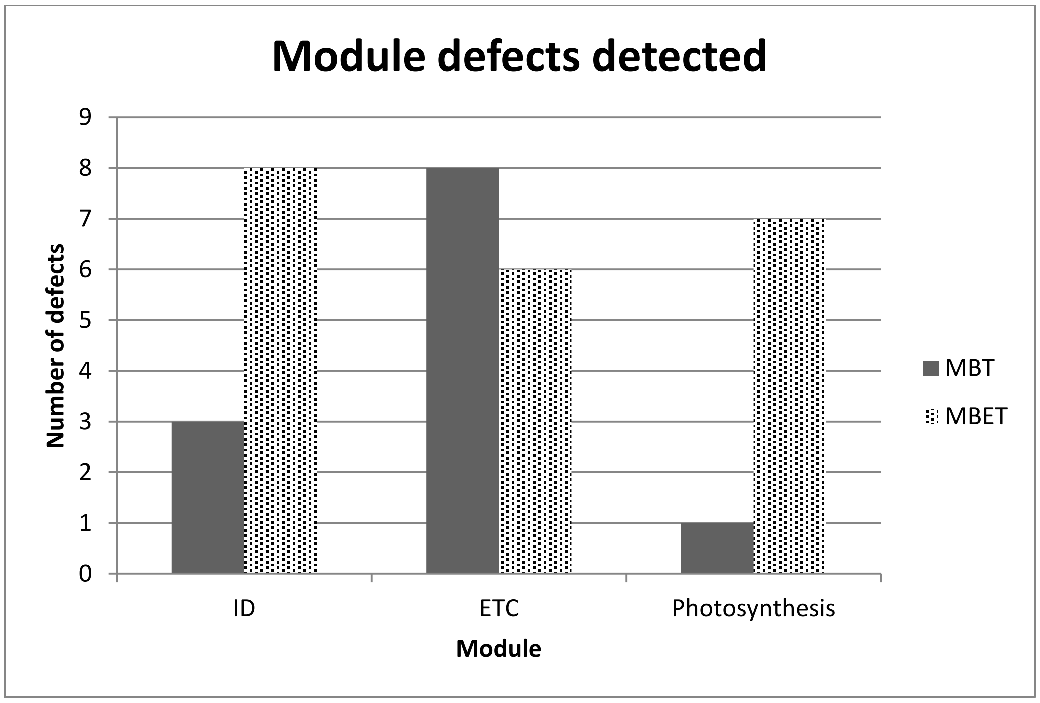


Figure 4.1. Detected defects for each testing method by module.

The defects were discussed with the development team and were then assigned a severity level. Each detected defect was assigned a severity level dependent on how it affected the function of the system, shown in Table 4.4.

Severity	Low	Medium	High
MBT	2	5	5
MBET	4	6	11

Table 4.4. Defect severity.

For instance, defects that caused the connection to the server to be unexpectedly interrupted or closed were designated as a “high” severity because failure of creating and maintaining a connection to the SUT prevented further use or testing of the SUT. Defects that related to missing game functionality or response message success were designated as a "medium" severity since they affected the functionality of the SUT, but did not prevent further use or testing of the SUT. Those that related to a missing instance of multiple objects in a game room were designated as a "low" severity because they did not affect the functionality of the SUT, but still caused unexpected results. Most of the defects fit into one of these levels, but those that did not were assigned a severity level after discussion of their importance with the development team. The severity of these defects detected by each testing method can be seen in Figure 4.2.

As shown in Figure 4.2, both of the methods detected about the same number of medium severity defects (MBT-5, MBET-6), and nearly the same number of low severity defects (MBT-2, MBET-4). However, the number of high severity defects detected was considerably different between the two methods, with MBET detecting two times as many (11 vs. 5).

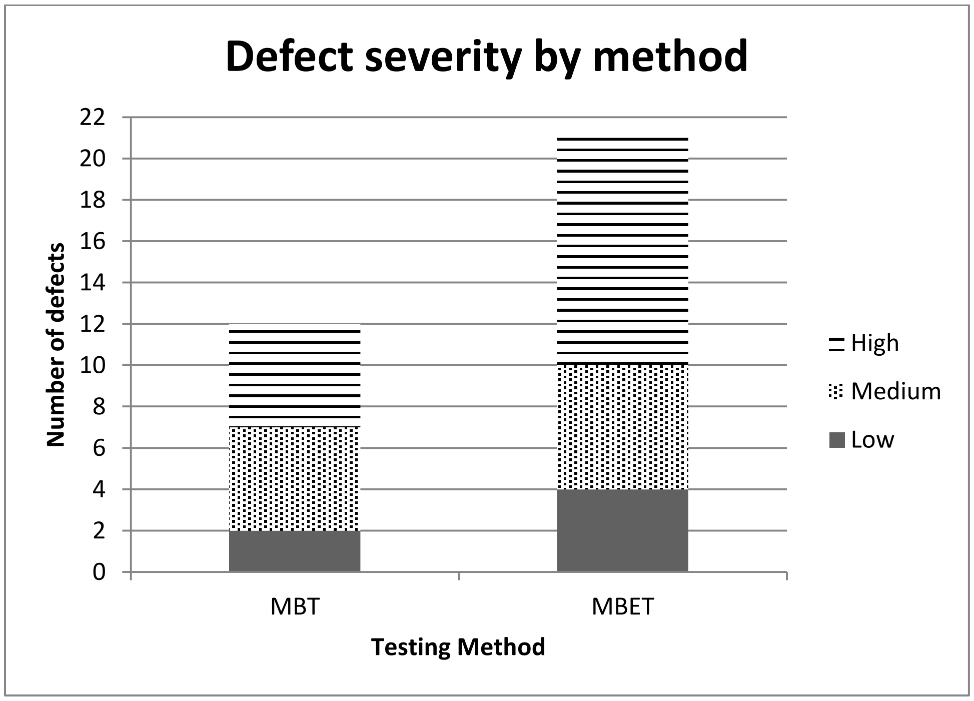


Figure 4.2. Detected defect severity for each testing method.

Using these severity levels, we can then breakdown the defect severity levels detected by the testing methods for each module, shown in Table 4.5.

Module	Testing Method	Severity Level		
		Low	Medium	High
ID	MBT	1	0	2
	MBET	2	2	4
ETC	MBT	1	4	3
	MBET	2	1	3
Photosynthesis	MBT	0	1	0
	MBET	0	3	4

Table 4.5. Defect severity level detected for each module.

The number of low level defects detected in the Photosynthesis module is zero because the module is missing a large portion of functionality. Although the ID module was completed and underwent testing during its development phase, a large number of high level defects were still detected. The ETC module contained the largest number of defects, with a similar number of medium and high level defects being detected. The defects can then be categorized by which testing method detected them in each module, as shown in Figure 4.3.

As seen in Figure 4.3, MBET detected the same number or more high level defects in all three modules (4 vs. 2 in ID, 3vs. 3 in ETC, and 4 vs. 0 in Photosynthesis). It also detected the same number or more low level defects in all three modules (2 vs. 1 in ID, 2 vs. 1 in ETC, and 0 vs. 0 in Photosynthesis). MBT was quite more effective in detecting medium level defects in the ETC module (4 vs. 1), but MBET was slightly more effective in the other two modules (2 vs. 0 in ID and 3 vs. 1 in Photosynthesis).

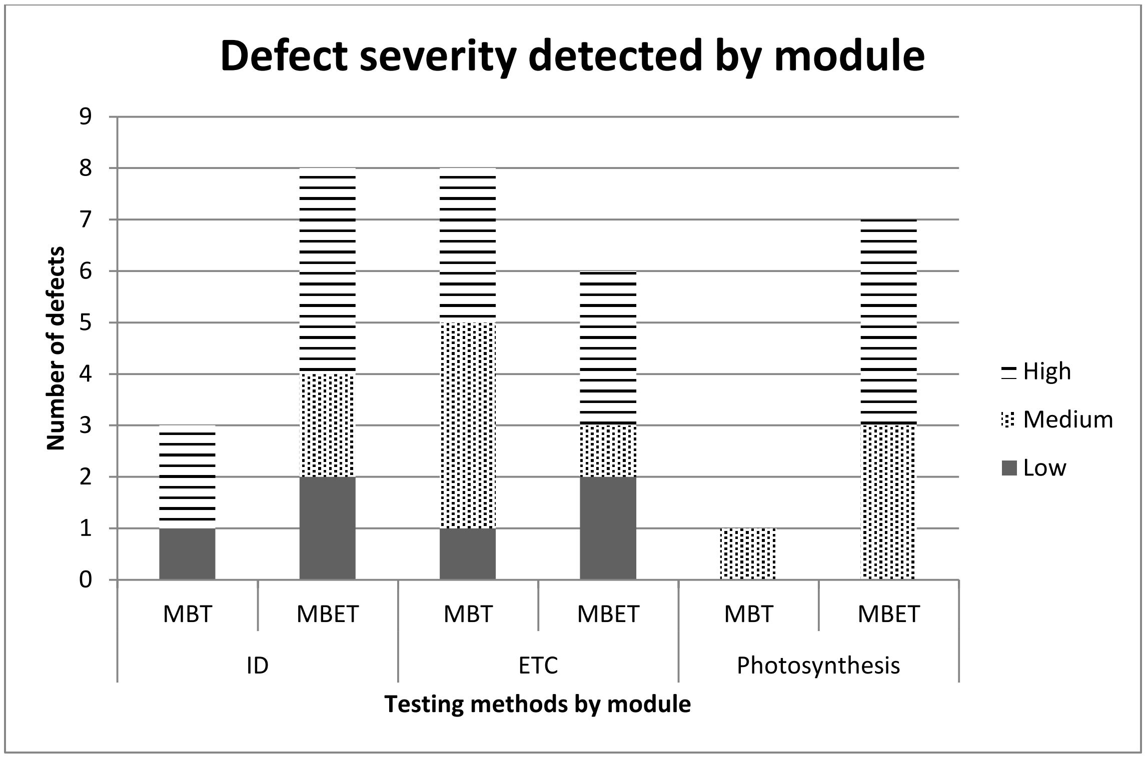


Figure 4.3. Defect severity levels detected for each testing method by module.

Besides the severity level, the defects were categorized by their type. We examined all of the defects that were detected and they all fall into four separate categories. These type categories were created from investigating the causes and effects of the defects detected in the experiment. They are: Authentication/Connection based defects (A/C), API based defects (API), database based defects (DB), and missing or malformed functionality defects (FM). Each of these types is completely distinguishable from the others, preventing any defect type overlap.

A/C type defects generally result from issues creating or maintaining a connection with the server or authenticating that connection. API type defects are those that are inherent in game events used by the server that are provided by the server development team to the rest of the project team. These generally are issues with the instantiation or data contained in the game events themselves. DB type defects are issues related to information of game objects that is stored in the database and managed by the server. FM type defects result from functionality that is missing, incomplete, or malformed with the game, such as missing goals or tasks.

The number of each defect detected dependent on its type is shown in Table 4.6. The number of A/C type defects is the majority of those detected, with a large number of API type defects also being detected. The DB and FM type defects only make up about a third of all the defects detected.

Type	A/C	API	DB	FM
MBT	2	6	2	2
MBET	12	2	4	3

Table 4.6. Defects detected by type.

The testing method used can then be applied to the type of defects found. As shown in Figure 4.4, each method found unique defects dependent on their type.

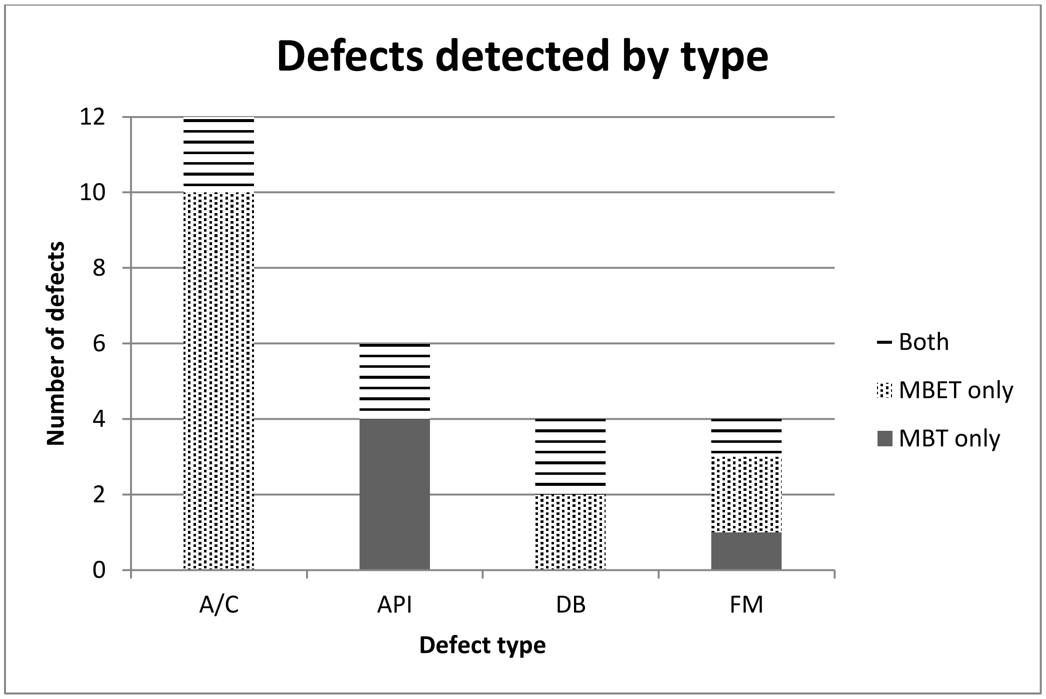


Figure 4.4. Defects detected for each type by testing methods.

In all of the cases, both testing methods detected at least one defect that was the same (designated as the “Both” category in Figure 4.4). It is important to note that defects detected by both methods are included in each method’s total, so they are counted twice in overall totals. MBET was significantly more effective in detecting A/C type defects than MBT (12 vs. 2), while MBT was significantly more effective in detecting API type defects than MBET (6 vs. 2). Both methods were similar, or not noticeably more effective than the other, in effectiveness in detecting DB and FM type defects.

The type of defects that were detected can then be broken down by the module in which they were detected in, shown in Table 4.7.

Module	Testing Method	Defect Type			
		A/C	API	DB	FM
ID	MBT	0	2	1	0
	MBET	6	0	2	0
ETC	MBT	1	4	1	2
	MBET	3	0	2	1
Photosynthesis	MBT	1	0	0	0
	MBET	3	2	0	2

Table 4.7. Defects detected for each module by type.

The ETC module contains the most of each type of defect, except for A/C type defects. The ID module and Photosynthesis modules did not detect defects in all defect type categories; no FM type defects were detected in the ID module and no DB type defects were detected in the Photosynthesis module.

The number of defects detected for each module by type can then be separated by the testing method that detected these defects, shown in Figure 4.5. MBET was more effective in detecting A/C type defects for all three modules (6 vs. 0 in ID, 3 vs. 1 in ETC, and 3 vs. 1 in

Photosynthesis). MBT was more effective in detecting API type defects in two of the three modules (2 vs. 0 in ID and 4 vs. 0 in ETC), while MBET was more effective in the third module (2 vs. 0 in Photosynthesis). MBET was slightly more effective in detecting DB type defects in two of the three modules (2 vs. 1 in ID and 2 vs. 1 in ETC), while neither detected any in the Photosynthesis module. The effectiveness of both testing methods in detecting FM type defects is similar, or not significant.

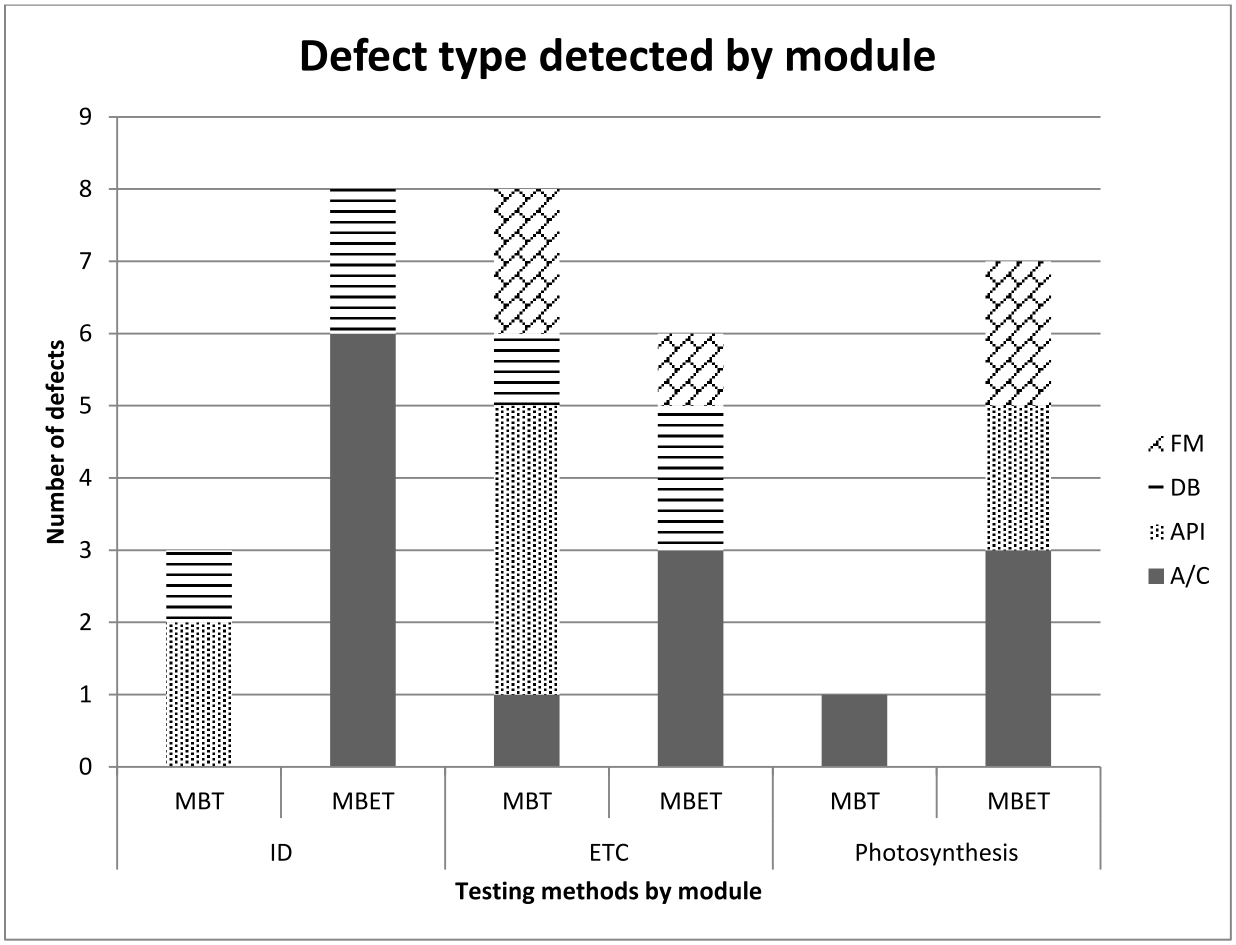


Figure 4.5. Type of defects detected by testing method for each module.

We also briefly investigated the generation of infeasible test cases by MBT. In our case, MBT generated a test suite of test cases that covered all transitions in a state machine model. For the ID module, a suite of 267 unique test cases was created. Of these test cases, 120 of them are infeasible test cases. Due to the complexity of the ID module, the state machine model contains multiple loops that must be completed, in any order, before the path to the final state is accessible. A simple guard prevents the accessibility of this path until all the aforementioned loops have been traversed. Because no test cases include the traversal of all of the loops, none should proceed past this guard, although some of the test cases contain such a path. These test cases incorporate an infeasible path that can only be prevented by verifying a run-time dependent guard.

More information and figures can be found in Appendix C, which go into greater detail of the data collected from the experiment. The data found in this section will be discussed in the next section, Section 5.

CHAPTER 5. DISCUSSION

We now discuss the results and practical implications of our results from the previous section, Section 4.5. The results from our experiment show that overall the MBET method found more defects than the MBT method. When broken down by game module, two of the three modules follow the same trend. In the third module, ETC, MBT found more defects than MBET, but upon further investigation the cause of this is from MBT detecting API type defects that MBET did not. The API type defects detected in this experiment all could have been detected from simple unit testing on the game’s event classes. These defects all related to instantiation of the game events and accessing the data of that event. It is important to point out that no unit testing was done on the server or API classes prior to our experiment. With this in mind, these defects could have been addressed before system testing and would not have been detected during our experiment. It is also important to point out that a majority of defects that were detected overall in our experiment by MBT were API type defects.

The severity of the defects that were detected by MBET was generally higher than those detected by MBT. In each module, MBET found more or the same amount of “high” level severity defects than MBT. Besides the ETC module, the same can be said about the “medium” level severity defects. These “medium” level severity defects can be traced back to the same API type defects mentioned above, which if addressed prior to our experiment would greatly influence the results of our experiment. When determining the severity of the defects, the server development team was consulted to assign levels to defects that were initially unknown.

It can also be stated from the results that MBET was more effective at detecting A/C type defects than MBT. These defects that deal with player authentication and the connection to the game server contain the most “high” level severity defects than any of the other types of defects. This is mainly because the defects associated with the connection to the server and authenticating a player’s login and ability to send messages are the most important. These defects affect the ability of a client to connect and communicate with the server properly, which can cause significant problems with the client-server functionality. Without detecting these defects, they will cause unexpected results that may cause catastrophic failure of the system.

Another point to keep in mind is the effectiveness of each testing method on the three modules in different stages of development. While MBET was significantly more effective in detecting defects in the ID module, which had finished development, and the Photosynthesis module, which was just starting development, MBT was slightly more effective in detecting defects in the ETC module, which was nearing developmental completion. MBET was able to detect defects in multiple stages in the developmental lifecycle of a game module for the Virtual Cell project, while MBT was only able to detect a significant number of defects in the ETC module (near completion) and a few in the ID module (completed).

It is also important to note that test cases produced by each method rely on different types of accessible data. MBT, for instance, executes a test case in its entirety, and the end result of the test case (external data) is then used to determine whether a defect has been detected. MBET using dynamic information about the system while the test case is being generated and performed (internal data), and would detect a defect when the system fails to comply with the model. MBT test cases may contain infeasible paths through the state machine model, since they do not incorporate guards and may detect defects in such a path. MBET would prevent an infeasible path from being executed, and may not detect the defects that MBT would. However, if the model conforms to the system that is being tested, an infeasible path should not be possible in the SUT, and would possibly generate false defects.

Although defects found with test cases from a MBET process may not always cause problems important to system integrity or a catastrophic failure when compared to test cases produced from MBT methods, the MBET test cases can be more useful. The set of MBET test cases could be designated as test cases that are more likely to find the defects that a user would discover while using the system. Moreover, when defects are found using any type of test case (manually generated, recorded from an instrumented client, MBT generated, or MBET generated), that test case can be used as an initial input for a new MBET test which generates a new test case and helps expand the test as a tester looks for related defects. The new MBET test case can also help verify that the initial defect has actually been fixed as well.

To summarize, MBET was more effective in detecting some types of defects than MBT. MBT detected more defects in one of the three game modules, ETC, while MBET detected more defects in two of the three game modules. By generating a test suite using both methods to complement each other, different types of defects can be detected, resulting in more complete test coverage of the SUT.

CHAPTER 6. CONCLUSION

In this thesis we have proposed a software testing method, MBET, and conducted an empirical study to investigate whether it is an effective method for detecting defects in a software system. The results of our experiment showed that MBET was indeed more effective overall than MBT in detecting defects. However, upon further investigation into the types of defects that were detected by each testing method, we found that each method has its own strengths. We concluded that if a test suite could be generated by using both MBT and MBET, more complete test coverage, compared to using a single testing method, can then be achieved for the SUT.

The combination of using scripted testing methods, such as MBT, and exploratory testing methods, like our proposed MBET, complement each other and detect defects that the other testing method might miss. James Bach sums up this idea in a single statement [3],

"*I find that most situations benefit from a mix of scripted and exploratory approaches.*"

Our experiment was executed against a single revision of the Virtual Cell server. We plan on expanding our study to include multiple revisions of the Virtual Cell server and collect more data for analysis. Including more revisions of the Virtual Cell server could improve our understanding of the effectiveness of MBET compared to MBT.

After all the data was collected from our experiment, it still had to be evaluated manually by a tester to determine what was found from the experiment. This process of evaluating the data can be difficult and expensive in terms of time and labor. If it is possible, then automating the evaluation of such results could help improve the MBET process even further by reducing the need for time and labor resources.

To address the external threats to validity in our study, we plan to expand the experiment to include other applications and other types of applications besides event-driven systems. By expanding our study to include these applications, we can address the threat to validity of generalizing our study's results to other systems.

Two further projects that can be investigated and applied to our next experiment are the selection of termination points for MBET and adjustments to the transition weighting system. Using the results from our experiment, other possible termination points can be selected for the next experiment, which will affect that experiment's results. Our results can also be used to possibly calculate a better weighting system or heuristic function for the selection of transitions by MBET.

REFERENCES

[1] N. Shenar, and A. Zylberman. “Automated Exploratory Testing.” Internet: http://www.testingexcellence.com/automated-exploratory-testing-2/, Feb. 2010.

[2] J. A. Whittaker. *Exploratory Software Testing: Tips, Tricks, Tours and Techniques to Guide Test Design*. Indianapolis: Addison-Wesley, 2010.

[3] James Bach. “Exploratory Testing Explained.” Internet: http://www.satisfice.com/articles/et-article.pdf, 2003.

[4] K. Li, and M. Wu. *Effective Software Test Automation: Developing an Automated Software Testing Tool*. San Fransico: Sybex, 2004.

[5] E. Dustin. *Effective Software Testing: 50 specific ways to improve your testing*. New York: Addison-Wesley, 2003.

[6] E. Dustin, T. Garrett, and B. Gauf. *Implementing Automated Software Testing: How to Save Time and Lower Costs while raising quality, 1st Edition*. Indianapolis: Addison-Wesley, 2009.

[7] A. Bacioccola, M. Catelani, L. Ciani, and V. L. Scarano. “Software automated testing: A solution to maximize the test plan coverage and to increase software reliability and quality in use.” *Computer Standards & Interfaces*. vol. 33 issue 2, Feb. 2011, pp. 152-158.

[8] M. Kelly. “Choosing a test automation framework.” Internet: http://www.ibm.com/developerworks/rational/library/591.html, Nov 2003.

[9] F. Bouquet, C. Grandpierre, B. Legeard, F. Peureux, N. Vacelet, and M. Utting. "A subset of precise UML for Model-based Testing." in *Proceedings of the 3rd international workshop on Advances in model-based testing*. Jul. 2007, pp. 95-104.

[10] I. K. El-Far and J. A. Whittaker. "Model-based Software Testing." *Encyclopedia on Software Engineering*. 2001.

[11] A.C. Dias Neto, R. Subramanyan, G. H. Travassos, and M. Vieira. "A Survey on Model-based Testing Approaches: A Systematic Review." *International Conference on Automated Software Engineering 2007*. 2007, pp. 31-36.

[12] A. Pretschner. "Model-based Testing." in *Proceedings of the 27th International Conference on Software Engineering*. May 2005, pp. 722-723.

[13] A. C. Dias Neto and G. H. Travassos. "Supporting the Selection of Model-based Testing Approaches for Software Projects." in *Proceedings of the 3rd International workshop on Automation of software test*. May 2008, pp. 21-24.

[14] PC Anitha, M Mahesh, PVR Murthy, and R Subramanyan. "Test Ready UML Statechart Models." in *Proceedings of the 2006 International workshop on Scenarios and state machines: models, algorithms, and tools*. May 2006, pp. 75-82.

[15] "State Machine Diagrams." Internet: http://www.uml-diagrams.org/state-machine-diagrams.html, 2010.

[16] P. Frohlich and J. Link. "Automated Test Case Generation from Dynamic Models." *European Conference on Object-Oriented Programming*. 2000, pp.472-491.

[17] C. J. Nagle. "Test Automation Frameworks." Internet: http://safsdev.sourceforge.net/FRAMESDataDrivenTestAutomationFrameworks.htm , 2000.

[18] "Introduction to OMG's Unified Modeling Language." Internet: http://www.omg.org/gettingstarted/what_is_uml.htm, Jul. 2005.

[19] A. Abdurazik and J. Offutt. "Generating Test Cases from UML Specifications." in *Proceedings of the 2nd International Conference on the Unified Modeling Language: beyond the standard*. 1999, pp. 416-429.

[20] V. Braberman, W. Grieskamp, N. Kicillof, and N. Tillmann. "Achieving Both Model and Code Coverage with Automated Grey-Box Testing." in *Proceedings of the 3rd International workshop on Advances in model-based testing*. Jul. 2007, pp. 1-11.

[21] R. Ramler and K. Wolfmaier. "Economic Perspectives in Test Automation: Balancing Automated and Manual Testing with Opportunity Cost." in *Proceedings of the 2006 International workshop on Automation of software test*. May 2006, pp. 85-91.

[22] Crushinator (Cv1). Internet: http://wwwic.ndsu.edu/websvn/listing.php?repname=crushinator, 2002.

[23] WoWiWe Instruction Co. Internet: http://www.wowiwe.net, 2012.

[24] S. A. Brodsky, G. C. Doney, and T. J. Grose. *Mastering XMI Java Programming with XMI, XML, and UML*. New York: John Wiley & Sons, 2002.

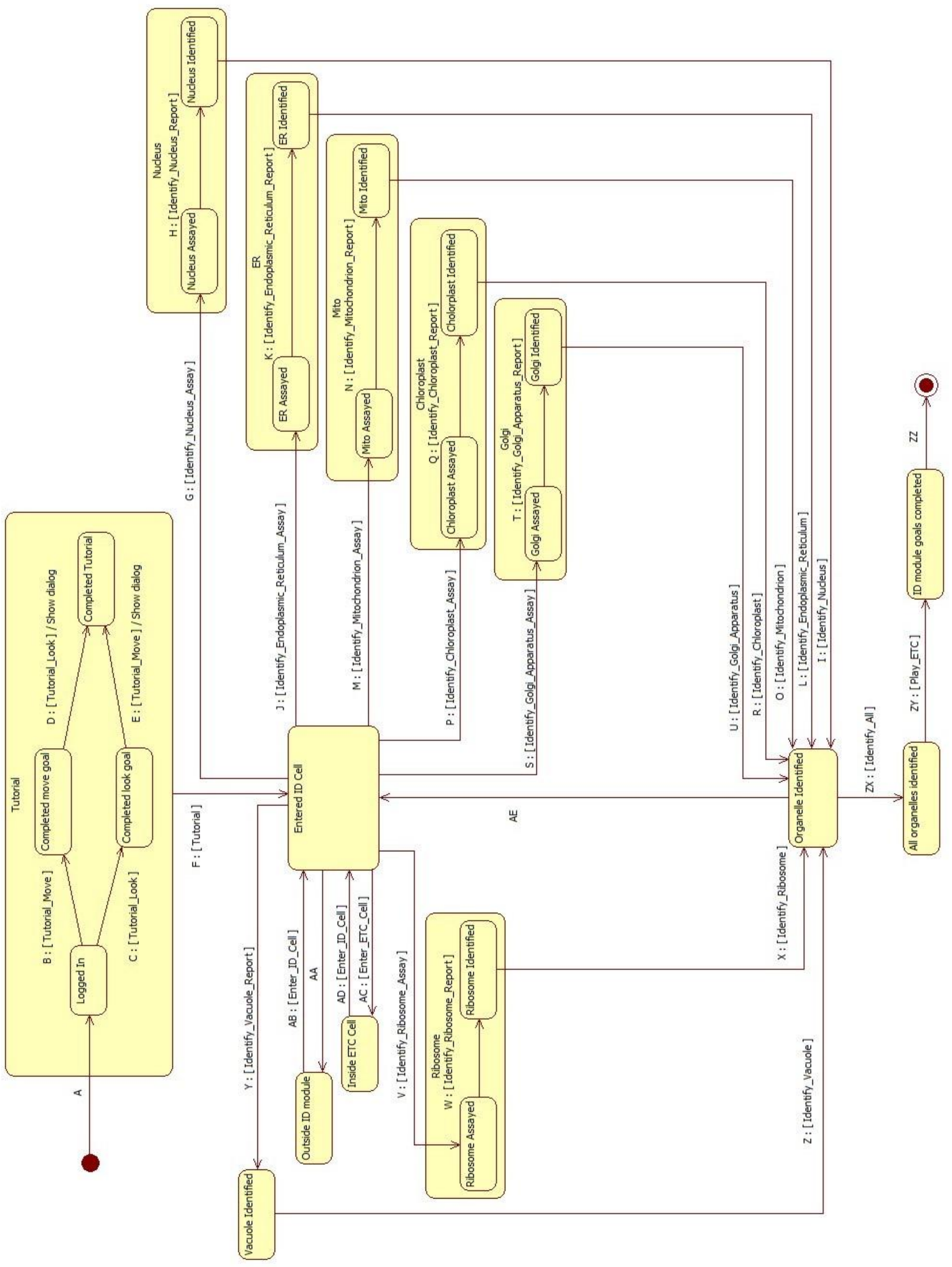
[25] StarUML 5.0. Internet: http://staruml.sourceforge.net/en/, 2005.

[26] "OMG Object Constraint Language." Internet: http://www.omg.org/spec/OCL/2.3.1/PDF/, January 2012.

[27] Virtual Cell (version 1). Internet: http://vcell.wowiwe.net/, 2006.

[28] Virtual Cell (version 2). Internet: http://wowiwe.net/virtual_cell.php, 2012.

APPENDIX A. VIRTUAL CELL STATE MACHINE MODELS



Tutorial
Logged In
B : [Tutorial_Move]
C : [Tutorial_Look]
Completed move goal
Completed look goal
D : [Tutorial_Look] / Show dialog
E : [Tutorial_Move] / Show dialog
Completed Tutorial
A
F : [Tutorial]
Entered ID Cell
G : [Identify_Nucleus_Assay]
Nucleus
H : [Identify_Nucleus_Report]
Nucleus Assayed
Nucleus Identified
J : [Identify_Endoplasmic_Reticulum_Assay]
ER
K : [Identify_Endoplasmic_Reticulum_Report]
ER Assayed
ER Identified
M : [Identify_Mitochondrion_Assay]
Mito
N : [Identify_Mitochondrion_Report]
Mito Assayed
Mito Identified
P : [Identify_Chloroplast_Assay]
Chloroplast
Q : [Identify_Chloroplast_Report]
Chloroplast Assayed
Cholorplast Identified
S : [Identify_Golgi_Apparatus_Assay]
Golgi
T : [Identify_Golgi_Apparatus_Report]
Golgi Assayed
Golgi Identified
U : [Identify_Golgi_Apparatus]
R : [Identify_Chloroplast]
O : [Identify_Mitochondrion]
L : [Identify_Endoplasmic_Reticulum]
I : [Identify_Nucleus]
Organelle Identified
AE
ZX : [Identify_All]
All organelles identified
ZY : [Play_ETC]
ID module goals completed
ZZ
Y : [Identify_Vacuole_Report]
Vacuole Identified
AB : [Enter_ID_Cell]
AA
Outside ID module
AD : [Enter_ID_Cell]
AC : [Enter_ETC_Cell]
Inside ETC Cell
V : [Identify_Ribosome_Assay]
Ribosome
W : [Identify_Ribosome_Report]
Ribosome Assayed
Ribosome Identified
X : [Identify_Ribosome]
Z : [Identify_Vacuole]

Figure A.1. ID module state machine.

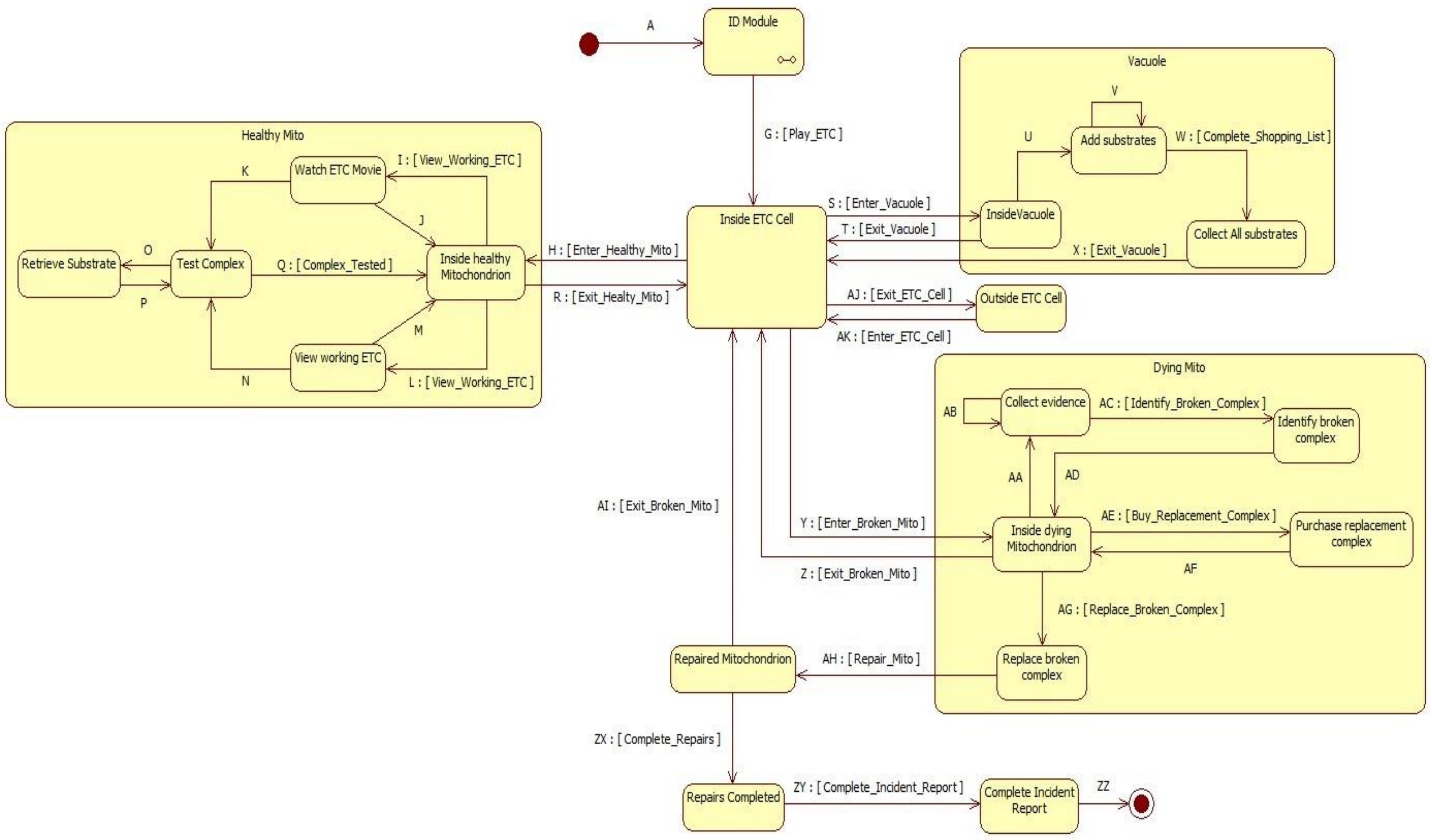


Figure A.2. ETC module state machine.

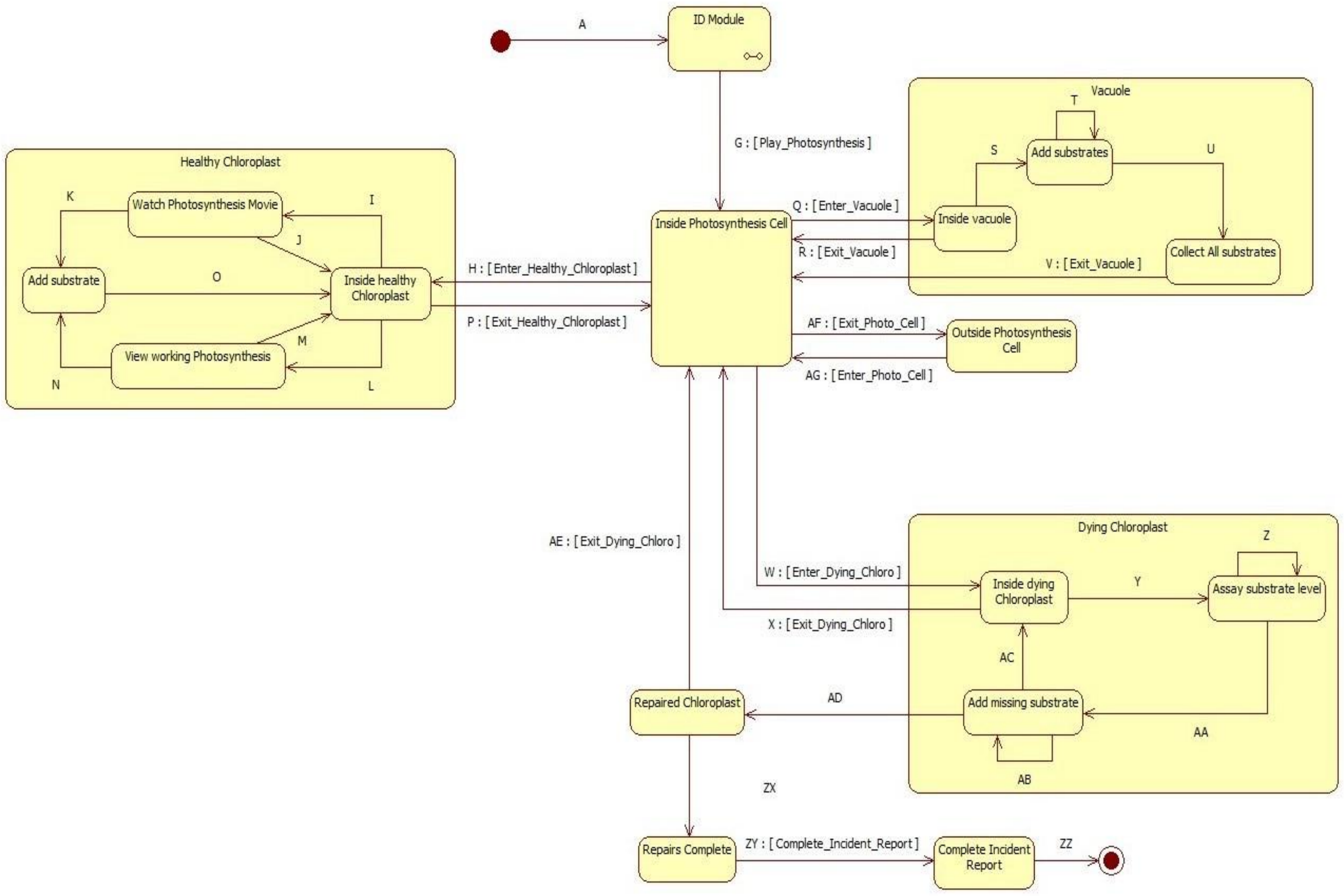

ID Module
A
G : [Play_Photosynthesis]
Inside Photosynthesis Cell
Healthy Chloroplast
Watch Photosynthesis Movie
Add substrate
Inside healthy Chloroplast
View working Photosynthesis
K
I
J
O
M
N
L
H : [Enter_Healthy_Chloroplast]
P : [Exit_Healthy_Chloroplast]
Vacuole
T
S
Add substrates
U
Inside vacuole
Collect All substrates
Q : [Enter_Vacuole]
R : [Exit_Vacuole]
V : [Exit_Vacuole]
AF : [Exit_Photo_Cell]
Outside Photosynthesis Cell
AG : [Enter_Photo_Cell]
AE : [Exit_Dying_Chloro]
Dying Chloroplast
Z
W : [Enter_Dying_Chloro]
Inside dying Chloroplast
Y
Assay substrate level
X : [Exit_Dying_Chloro]
AC
AD
Repaired Chloroplast
Add missing substrate
AA
AB
ZX
Repairs Complete
ZY : [Complete_Incident_Report]
Complete Incident Report
ZZ

Figure A.3. Photosynthesis module state machine.

APPENDIX B. TEST CONFIGURATION

B.1. Server Module Revisions

For this experiment we selected a specific revision of the Virtual Cell server. Since the server was made up of four separate architectural modules each with its own subversion repository, a specific revision of all modules had to be selected in order for them all to match up. The revision numbers used for the experiment is shown is Table B.1.

Architectural Module	Revision used
JavaMOO API	82
JavaMOO Server	6135
VCell API	24
VCell Server	680

Table B.1. Virtual Cell server module revisions.

B.2. System Specifications

A single system was used to setup and execute the Virtual Cell server that was tested in our experiment. The following are the system specifications:

- CPU: Intel Xeon CPU E5530 with 8 cores @ 2.4 GHz
- RAM: 12 GB
- Network: Intel Corp 8257EB Gigabit Network Connection (x2)

Two separate systems were used to execute the Crushinator that tested the Virtual Cell server in our experiment. The following are the system specifications of those two machines:

System 1

- CPU: Intel Core i7 CPU 920 with 8 cores @ 2.67 GHz
- RAM: 6 GB
- Network: Intel Corp Gigabit Network Connection (x2)

System 2

- CPU: Intel Core 2 Duo CPU T7300 with 2 cores @ 2 GHz
- RAM: 4 GB
- Network: Intel Wireless WiFi Link 4965AGN Card

B.3. MBET Termination Points

MBET was developed with a built in default termination point when it reached a final state in a state machine model. Since the state machine model can be extremely complex and possibly contain transition loops, another termination point was necessary. For our experiment we selected a maximum time limit (in minutes) for the test iteration to execute. This maximum duration value, once reached, would terminate any tests that were still executing. These values can be found in Table B.2.

Iteration 1	
Module test	Termination point (mins)
ID test A	5
ID test B	30
ETC test A	5
ETC test B	30
Photosynthesis test A	5
Photosynthesis test B	30
Iteration 2	
Module test	Termination point (mins)
ID	60
ETC test A	60
ETC test B	60
Iteration 3	
Module test	Termination point (mins)
ID	60
ETC	30
Photosynthesis	30

Table B.2. MBET maximum duration value per iteration.

B.4. Test Configuration Files

In between test executions of MBT and MBET, the configuration files used by the Crushinator were modified slightly. Generally this was done to address issues that applied to the version of the Virtual Cell server that was being tested. For instance, between certain versions of the Virtual Cell server, the names or spelling of certain objects was modified. The changes made in between test iterations can be found in Table B.3, describing the changes made to the specific configuration file for the test iteration. These configuration files are used by both MBT and MBET, and separate configuration files are required for each game module (for example the ID module has different configuration files than those used for the ETC module). The table below also corresponds to the separate files, and lists the configuration changes for each module separately.

ID Module		
Iteration	Configuration file	Description
1	SMTransitions	Removed FireItemEvent's from transitions
2	SMGuards	Goal guard changed to fix spelling issue
3	SMKeywords	Changed all lowercase ribosome names to both uppercase and lowercase names
3	SMKeywords	Removed ribosome entrances (store entrances)
3	SMGuards	Changed name of goal to have uppercase value rather than lowercase
4	SMGuards	Changed goal guard text that had improper value
ETC Module		
Iteration	Configuration file	Description
2	SMTransitions	Removed FireItemEvent's from transitions
2	SMGuards	Changed multiple goal guards to match spelling/content
3	SMGuards	Changed location guard from "ETC_Vacuole" to just "Vacuole"
4	SMKeywords	Removed listing of 5 molecules of each substrate that would be in vacuole, since substrates don't respawn
Photosynthesis Module		
Iteration	Configuration file	Description
2	SMGuards	Removed goal gurads for portions of module that are not implemented
3	SMGuards	Changed location guard from "Photo_Vacuole" to just "Vacuole"
4	SMTransitions	Fixed spelling error with reference to SMK
4	SMKeywords	Removed listing of 5 molecules of each substrate that would be in vacuole, since substrates don't respawn

Table B.3. Configuration files changes.

APPENDIX C. EXTRA FIGURES

This appendix contains extra tables and figures representing the collected data from the experiment that were not shown or described in the results section (Section 4.5).

The following figure, Figure C.1, shows the percentage of defects that each testing method detected for each game module out of the total detected in that module. The total number of defects takes into account defects found by both testing methods, only counting such defects once in the total.

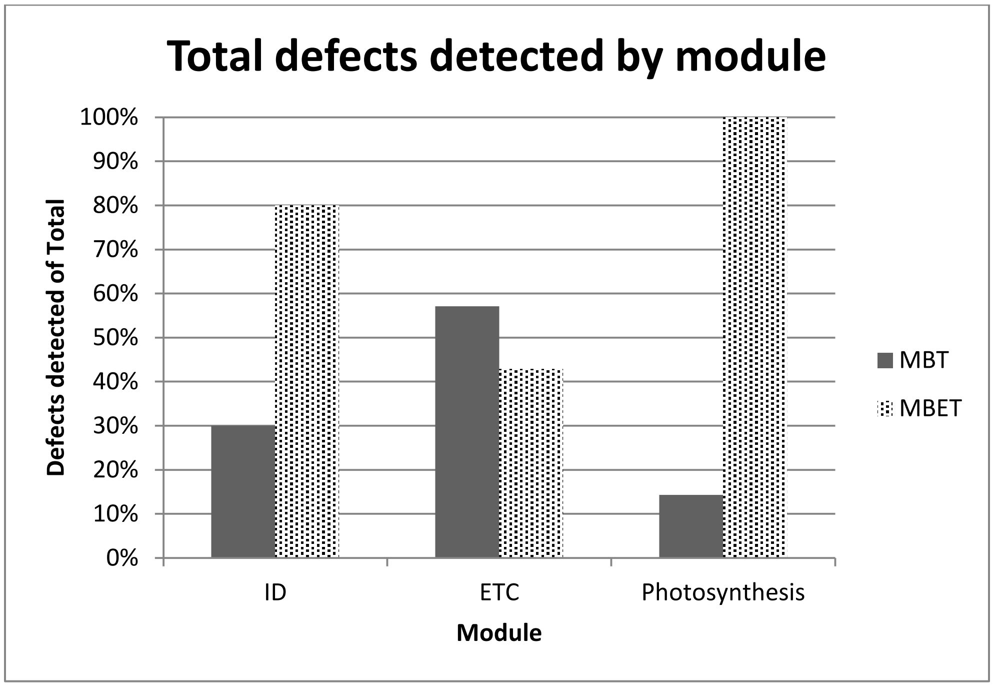


Figure C.1. Percentage of defects detected for a module by each testing method.

Figure C.1 shows that MBET detected a majority of all the defects detected in the ID module, and all of the defects that were detected in the Photosynthesis module. MBT detected just over half of the total detected in the ETC module, while MBET detected less than half of the total detected in the ETC module. It should be kept in mind that a majority of the defects detected in the ETC module by MBT were API type defects, which would generally be detected by simple unit testing of those classes.

Another interesting set of data to investigate is how effective each method was at detecting each type of defect in each game module. Figures C.2 through C.4 show this data for each module. Figure C.2 shows the percentage of each type of defect that was detected by each testing method in the ID module. For instance, MBET detected all of the A/C and DB type defects that were detected in the ID module. MBT detected all of the API type defects and half of the DB type defects that were detected in the ID module. No FM type defects were detected by either method.

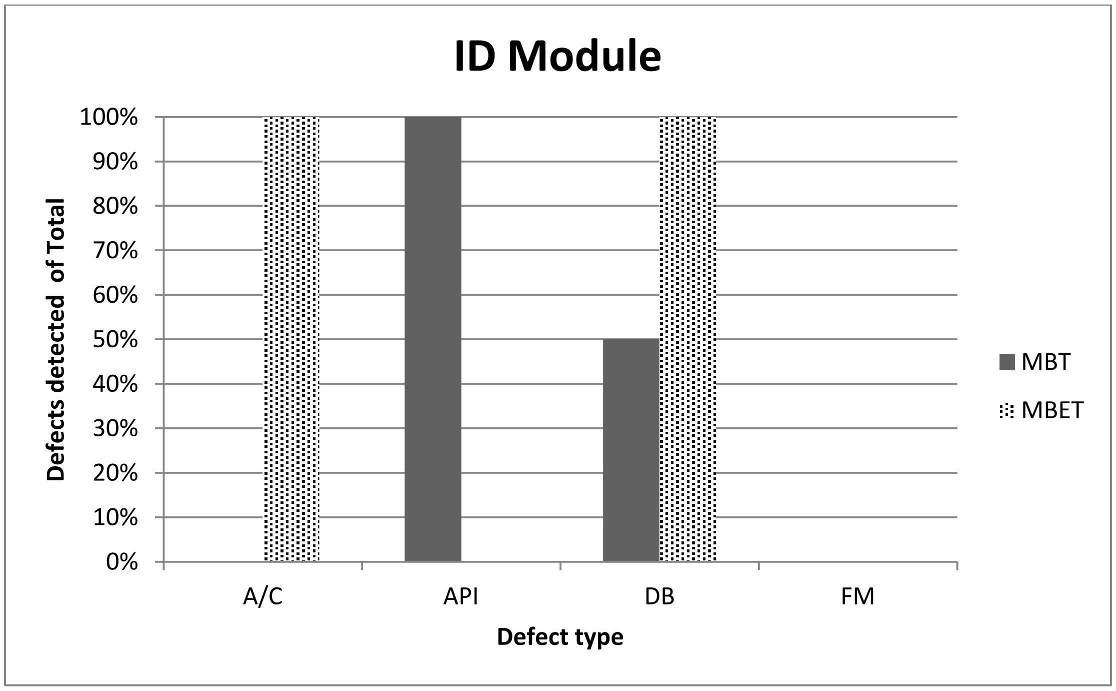


Figure C.2. Percent of defects detected in the ID module.

Figure C.3 shows the percentage of each type of defect that was detected by each testing method in the ETC module. In this module, MBT detected all of the API type defects, a majority of the FM type defects, and a small percentage of the A/C type defects. MBT did not detect any of the DB type defects in the ETC module. MBET detected all of the DB defects, a majority of the A/C type defects, and a small percentage of the FM type defects. MBET did not detect any of the API type defects in the ETC module. This graph shows a great example of how the two testing methods complement each other.

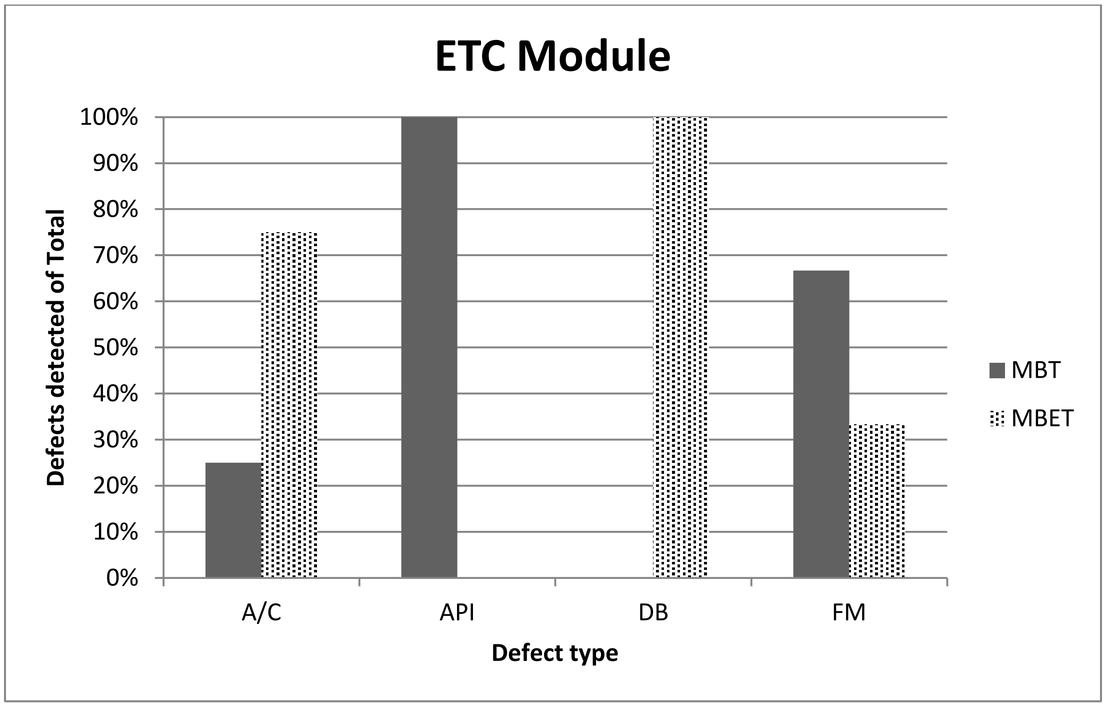


Figure C.3. Percent of defects detected in the ETC module.

Figure C.4 shows the percentage of each type of defect that was detected by each testing method in the Photosynthesis module. MBET was extremely effective in the Photosynthesis module. It detected all of the A/C, API, and FM type defects found in the Photosynthesis module. MBT was not very effective at all, finding only a small percentage of the A/C type defects. No DB type defects were detected by either method in the Photosynthesis module.

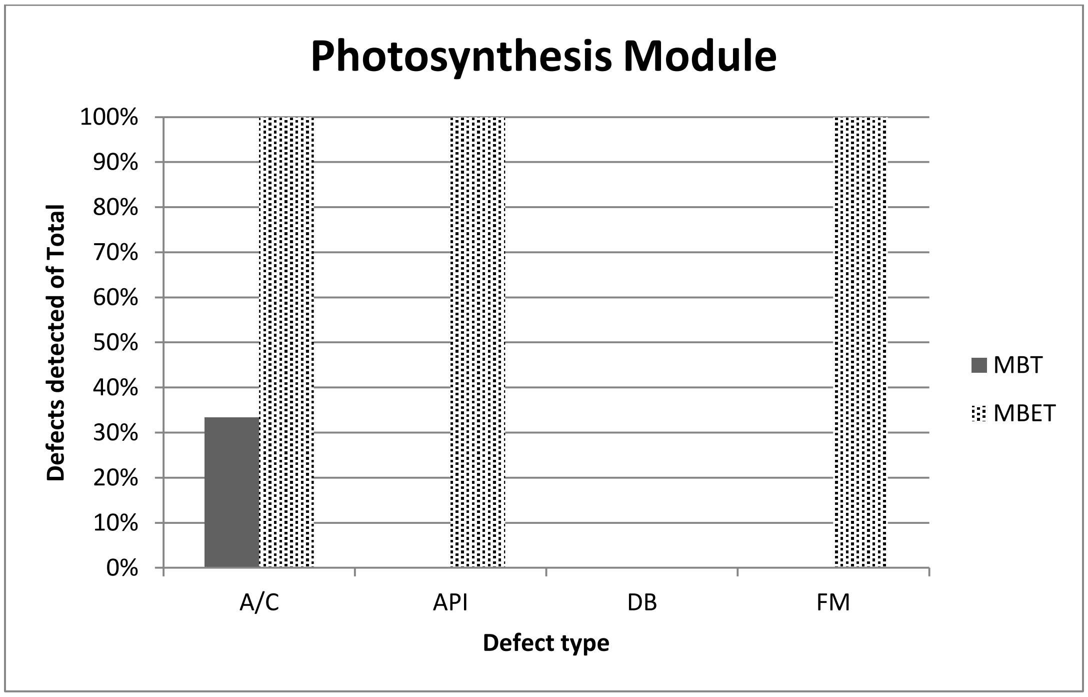


Figure C.4. Percent of defects detected in the Photosynthesis module.