Model-Based Exploratory Testing: A Controlled Experiment

Christopher J Schaefer *
* IBM Rochester
MN, USA
cjschaef@us.ibm.com

Hyunsook Do †
† North Dakota State University
Computer Science
hyunsook.do@ndsu.edu

***Abstract*—Exploratory testing provides an intuitive way for testing a software product that testers can apply, but the advantages of exploratory testing are generally outweighed by its disadvantages, mainly the time and resources necessary to perform it manually. To address this problem, in this research, we propose Model-Based Exploratory Testing (MBET), an approach that incorporates the advantages of exploratory testing and Model-Based Testing (MBT) that automates the testing processes. To support the MBET approach, we implemented an automated testing tool, the Crushinator. To assess our approach, we conducted an experiment using an educational game application with multiple versions and we collected the number and type of defects detected with the MBET and MBT approaches. Our results showed that, overall, MBET detected more defects than MBT. The results also showed that MBET detected certain defect types better than MBT while MBT detected other types better than MBET.**

***Index Terms*—exploratory testing, model-based testing, model-based exploratory testing, event-driven applications**

I. Introduction

Exploratory testing is a continuous process of learning, test design, and test execution for 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, software companies often overlook exploratory testing as a useful testing method because it tends to be more demanding of resource use. Further, exploratory testing is believed to have limited test coverage compared to other testing methods.

Automating exploratory testing can address this problem because automated tests can reduce the amount of time testers spend creating and executing test cases, which saves time and money [4], [5], [6], [7]. There are many approaches to automate software testing, such as data-driven testing, keyword-driven testing, and model-based testing [8], [9]. Of these automation approaches, model-based testing has been used widely because it requires less effort for implementation and maintenance.

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 standard modeling tools, such as the Unified Modeling Language (UML), models can be created from the system requirements and an abstract knowledge of how the SUT is supposed to function [14], [15], [16]. Such models can normally be created by the design team prior to the SUT development. Test cases can then be derived by traversing the model according to the coverage criteria. Further, by using the model to generate test cases, a better coverage of the SUT can be achieved [17].

Therefore, in this research, we propose a new testing approach, Model-Based Exploratory Testing (MBET), that supports automated exploratory testing. By integrating the two approaches, exploratory testing and MBT testing, we can address two problems. First, automation can save time and resources, which have been a major obstacle for the industry to adopt exploratory testing for its testing process. Second, using MBT alone can produce infeasible test cases due to its static nature. Exploratory testing, however, generates test cases dynamically, so the infeasibility problem will be properly handled. To support our approach, we implemented a MBET framework, the Crushinator. The Crushinator is an automated software testing tool that provides a framework for MBT and exploratory testing. The Crushinator was designed to test event-driven applications, more specifically educational game servers. The Crushinator automates the process of generating and executing test cases against the SUT.

To evaluate the MBET approach, we performed an experiment using an educational game developed by WoWiWe Instruction Co. [18]. This evaluation was done by comparing the number and type of defects found utilizing test cases generated by using a MBET process and those found by using a basic MBT process. The results from this experiment showed that, overall, MBET detected more defects than MBT. MBET detected certain defect types better than MBT while MBT detected other types better than MBET. The results indicated that the defects detected by these two testing methods are complementary and can generate a test suite that provides the advantages of both exploratory testing and automated testing methods.

The remaining portion of this paper is structured as follows. Section II presents background information about exploratory testing and model-based Testing as well as their related work. Section III presents the MBET process and detailed descriptions about the Crushinator. Sections IV and V present our experiment, including design, threats to validity, results, and analysis. Section VI discusses our results and finally, Section VII presents conclusions and discusses possible future work.

II. Background and Related Work

The process of exploratory testing involves a cycle of test design and execution where testers are constantly learning and adapting their design and execution from the collected results. In fact, testers practice this exploratory testing process without even knowing it [1], [2], [3]. They execute test cases, report bugs, and then re-execute test cases to verify that bugs have been fixed. Exploratory testing provides an intuitive way of testing a software product that testers can easily apply, but it has not been recognized by software companies as one of standard testing approaches because it is considered to be more time- and resource-demanding than a typical, scripted testing approach being used in industry [1], [2], [3].

The extensive time and resource allocation can be problematic during the testing phase because, often, the product release is delayed, and in that case, software companies cut back on testing activities in order to ensure a timely release of their product. Thus, many organizations do not tend to assign the appropriate amount of time and resources for testing. Moreover, exploratory testing tends to produce less testing documentation prior to test execution. Such documentation is generally used by 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. Because 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 can address some of these problems, but at the same time, it is a controversial topic because most research and discussion agree that effective exploratory testing cannot be fully automated [1], [3]. While completely automating the exploratory testing process would be difficult, some researchers and practitioners [1] suggested that, even if we could partially automate exploratory testing by using the existing automation approaches or methods (e.g., recording test execution steps and replaying them for regression testing), exploratory testing could reap the benefits of automated software testing. For example, Zylberman and Shenar [1] discuss several ways to automate exploratory testing, such as Passive Exploratory Automated Testing, where testers perform exploratory testing as their actions are recorded using a third party tool, or Active Exploratory Automated Testing, where keyword-driven testing is used to automate test execution for exploratory testing.

Model-Based Testing (MBT) is one way to achieve such automation. MBT uses the behavior models of the SUT to guide the testing process [8], [9], [10], [19]. The models can be used to generate test cases or to validate the results of test cases [12], [14]. Because MBT offers many benefits, such as early exposure of defects in specification and design, and automated test generation, it has been extensively explored by many researchers. In particular, MBT has gained more popularity due to the widespread use of UML that has become the de-facto standard modeling language and the need to test the more complex systems being developed today [13]. Neto et al. [11] provide a comprehensive overview of model-based testing approaches, and many commercial and academic tools exist to support MBT approaches [20], [21].

III. Model-Based Exploratory Testing Approach

To support a Model-Based Exploratory Testing (MBET) approach, we built an automated testing tool, the Crushinator. This section provides an overview of the Crushinator and describes the MBET process supported by the Crushinator.

A. The Crushinator

The Crushinator was designed as a layered architectural system, as shown in Figure 1. This design allows a layer to be replaced without affecting the system when it is modified or upgraded. In the figure, the layers inside the dotted box are dependent upon the system under test (SUT), and the "Test Engine" and "Simulated Client" components are implemented for use directly on the SUT.

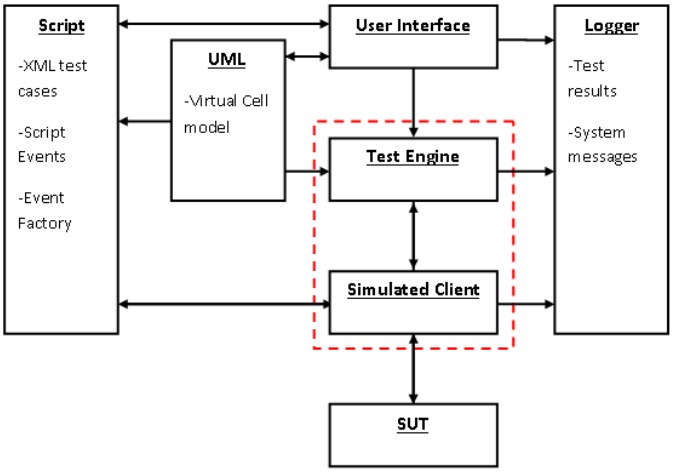

Fig. 1. Crushinator architectural diagram

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

As we mentioned earlier, our approach supports two testing approaches, Model-based testing and exploratory testing. The Crushinator implements these two approaches using classes in the "UML," "Test Engine," and "Simulated Client" components. The "UML" component is used to represent a state machine model at run-time. This model is based on the system specifications provided for the SUT. This information is then used to generate MBT test cases by finding paths through the state machine model depending on the type of coverage

selected (e.g., transition coverage). Test cases can then be executed by selecting them using the "User Interface" component and are then passed to the "Simulated Client" component to be executed. The model represented at run-time by the "UML" component is also used to perform exploratory testing. The "Simulated Client" component monitors its current state in the state machine model and then requests the available outgoing transitions from its current state, attempting to traverse one of these transitions. The "Simulated Client" continues to perform exploratory testing until it reaches a termination point. It then notifies the "Test Engine" component which can save the test case executed by the "Simulated Client" component.

Along with these layers, there are three assisting components used for handling a majority of the work associated with MBT and serializing test information. The "UML" component is responsible for generating a representation of a state machine model from an XML Metadata Interchange (XMI) [22] file (such as a model of the Virtual Cell server). The "UML" component is used by the "User Interface" component to allow a tester to select models to generate MBT test cases. The "UML" component is also used by the "Test Engine" component at run-time for the MBET test case generation.

The "Script" component 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 event type, along with its parameters, from an XML test case file format through the "Test Engine" to the game-dependent "Simulated Client" component. There, the information is handed to the "Event Factory" object in the "Script" component that turns the information into a SUT-dependent event to send to the SUT. The "Logger" is used throughout the layered components to serialize test results, configurations, warnings, etc. that occur while testing.

The Crushinator communicates directly with a game server, using a specified connection type (e.g., a HTTP socket connection), to send game-based events to the server, and to retrieve responses from that server, if necessary. The responses can then be used to collect results, update values, or parse data to send back to the server. The Crushinator project allows multiple connections types, such as Java RMI and socket connections, to be used inside the framework.

B. 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 application that can be represented using an UML state machine. The Crushinator testing tool framework described in Section III-A is an example of applying MBET to test multiple applications.

The following steps are taken to apply the MBET process:

1) Create a state machine model that represents 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 any partial initial test case if it exists.
5) Execute the triggers or events for an available outgoing transition from the current SUT state.
6) Check the run-time dependent transition guards to verify traversal of a transition and update the current state of the SUT if satisfied.
7) Calculate the available outgoing transitions from the current state.
8) Continue automated steps 5-7 until a pre-determined termination point, selected by the tester, is reached (e.g., completion of the game, time limit, or execution of a loop of transition(s)).
9) Output the series of transitions, along with their triggers and guard values, into a test case that can be used for future testing.

We illustrate how MBET steps work by using an example. The example model shown in Figure 2 is a small state machine model that is a subset of the state machine of the Virtual Cell game server used for the experiment described in Section IV. This subset model has been slightly modified to include an initial and final state to conform to a standard UML state machine model.

The Virtual Cell game is an event-driven, client-server educational game. Figure 2 models the partial behavior of this game application. The model represents a player who is logged into the game playing the Electron Transport Chain (ETC) game module. While inside the Cell room, the player can enter the Vacuole room. From here, the player can either exit the Vacuole room and return to the Cell room, or collect items, known as substrates, in the Vacuole room. The player continues to collect substrates until all necessary substrates have been collected. The player then returns to the Cell room.

Once we have finished building the state machine model (Figure 2), we use StarUML [23] to export the model into an XMI file (Step 2). We then connect to the SUT using the implemented Crushinator (Step 3) and execute an initial test case, if one is supplied (Step 4), to set up the implemented "Simulated Client" for the test. The "Simulated Client" component then executes the triggers for one of the outgoing transitions (Transition A in Figure 2) from its current state, in this example the initial state in Figure 2 (Step 5). The "Simulated Client" then checks the transition guard, if one exists (Location = Vacuole), to determine whether the transition was traversed, updating the current state if the guard is satisfied (Step 6). The set of outgoing transitions is then calculated from the current state (Step 7), and the selection and execution process (Steps 5-7) is repeated until a termination point is reach (Step 8). The test case executed by the "Simulated Client" is then saved for re-execution at a later time (Step 9).

The model transitions contain information necessary to represent the SUT. In this example, the transition label "A: Enter Vacuole : [Location = Vacuole]" is a simplified version of the one found in the models used in this study, 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

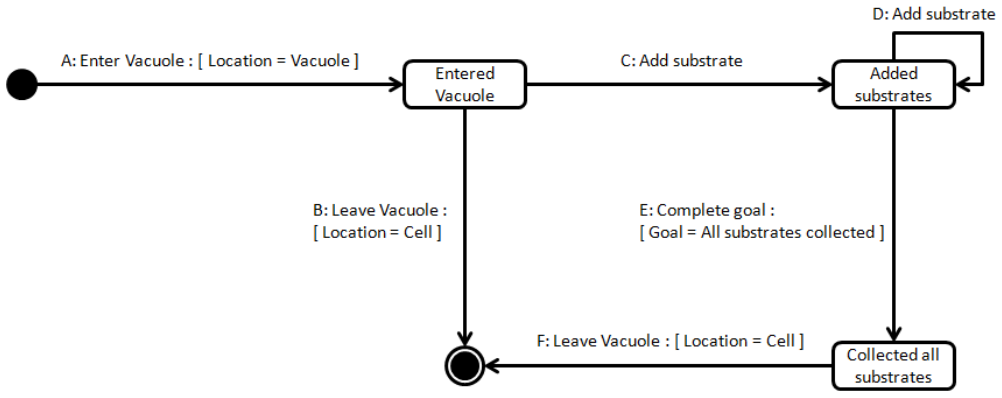

Fig. 2. Virtual Cell state machine subset

transition, "Location = Vacuole," is structured to represent the 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 are abstracted from the model and placed into configuration files, which are explained in Section III-C. Abstracting these data, such as transition triggers and guards, allows the model to retain its readability while fully representing the SUT. Also, when SUT changes 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 needs to be serialized into XMI, a format of XML used to represent models. In this work, the UML modeling tool StarUML 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 because it was open-source and had a feature to export models to XMI.

The Crushinator is thereafter 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 provides 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 process is complete, available outgoing transitions are calculated from the current state of the SUT specified by the state machine model. From those transitions, one is selected, and the triggers for that transition are applied to the SUT. Selecting 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. The Crushinator sorts the transitions by these values, with transitions having 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 in more detail in Schaefer's MS thesis [24]. A random selection from the available transitions can also be applied to this process.

After the triggers 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. Because 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. 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 again calculated for the new current state. Steps 5 through 7 are 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, meaning 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.

```
1. transitions = currentState.getOutgoing();
2. sorted = sort(transitions); //descend. order
3. foreach (Transition t in sorted)
4.     applyTriggers(t.getTriggers());
5.     if (checkGuard(t.getGuard()))
6.         currentState = t.getTargetState();
7.         testPath.add(t);
8.         return;
9.      else
10.        continue;
```

In line 1, we collect a set of outgoing transitions from the current state. In line 2, we sort the transitions by heuristic value in descending order. We then iterate through each sorted transition in line 3 and, in line 4, apply the triggers for that transition. In line 5, we check the transition's guard and, if the guard is satisfied, update the current state in line 6. We then add the traversed transition to the test case path in line 7 and return to the caller in line 8. If the guard is not satisfied in line 5, we continue, in line 10, to iterate the sorted transitions in line 3.

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

C. Transforming UML for MBET

To simplify the UML state machines model into a format that fully represents the SUT while remaining human readable, we transformed the UML model. This transformation kept the model as simple as possible while retaining a complete representation of the SUT. The Object Constraint Language (OCL) [25] was designed to allow UML to represent complex systems, but OCL does not address our issue of simplifying the state machine models and still retaining a full representation of the SUT. Our implementation of UML for MBET consists of three components. The first one is using 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 model's triggers and guards and replacing them with shorter, unique labels that can be used to extract the data from an external configuration file. The last one is weighting transitions from the state machine model. More detailed descriptions of these three components are available in [24].

IV. Empirical Study

To investigate the effectiveness of the MBET approach we described in Section III, we conducted an 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 and severities of defects.

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

A. 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. [18]. The Virtual Cell game is a graphics-heavy, 3D, virtual environment to teach users about cellular biology. Virtual Cell contains three separate modules 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 a cell, an Electron Transport Chain (ETC) module for teaching the process of the ETC, and a Photosynthesis (Photo) 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 one. A high-level state diagram shows the structure of the Virtual Cell game in Figure 3. We built a state machine model based on the system specifications for each of the three game modules.

TABLE I
Software Metrics for Virtual Cell Application

Version	Lines of code	No. of Classes
VC2.1	58865	425
VC2.2	66343	474
VC2.3	69307	499

The Virtual Cell goal structure allows the players to move from one module to another once they complete the tutorial, as can be seen in Figure 3. This structure allows the players to complete the game modules in any order. The players can also exit the game at any time. Once they exit, they can restart the game from any available module, depending upon their completion of the tutorial.

Two versions of the Virtual Cell game exist, an old version (version 1) [26] and a new version (version 2) [27]. The old version of Virtual Cell was developed in the late 1990s and early 2000s, and was written in Java. The 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 development team has 11 members on 4 different teams: server, client, graphics, and testing. The server-development team consists of 3 developers; the client team has 4 developers; the graphics team consists of 2 artists; and the testing team has 2 testers.

The primary focus on Virtual Cell for this experiment is testing the game server for the new version (version 2), and we chose three revisions that have modules at different development statuses. For instance, with the first server revision (VC2.1), the ID module was feature complete and had been strenuously tested during development. The ETC module was nearing development completion but still required some features to finish development. The Photo module had just recently been started, with the structure of several features just finished or currently in process. For the next revision (VC2.2), the ID module remained nearly the same, with a few of the defects having been addressed. The ETC module

was temporarily feature complete. (Further development had not been planned at that time.) The Photo module was still in the middle of development. The final revision (VC2.3) featured an ID module with further implementation to address detected defects, an ETC module with new features, and a Photo module that was nearing developmental completion.

The three separate revisions allow us to examine how effective each testing method is at different stages in the development process. Table I shows the number of classes and the number of code lines for each revision.

B. Variables and Measures

1) Independent Variable: To investigate our research question, we controlled a single independent variable: software testing method. We used two different testing methods: Model-Based Testing (MBT) and Model-Based Exploratory Testing (MBET). MBET is the heuristic method that we explained in Section III, and MBT is the experimental control method. To apply MBT, we generated test cases based on the complete transitional coverage criterion. To do so, starting from the initial state of the state machine model, test paths were generated so that all transitions in the model were covered while avoiding infinite loops that exist in the model. These paths were then used to generate test cases by combining the triggers for all the transitions in the path.

2) Dependent Variable and Measures: The dependent variable for this experiment was the number of defects that are detected by the software testing method. We also measured the severity of the defects, the type of defects (explained further in Section V), and in which modules the defects occurred to gain more insight about the two methods.

We assigned the severity for each defect by consulting with the development team. Each defect was assigned a severity level considering how seriously it can damage the system. For instance, defects that caused the server connection to be unexpectedly interrupted or closed were designated as a "high" severity because a failure to create and maintain 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 because they affected the functionality of the SUT but did not prevent further SUT use or testing. Defects that related to a missing instance of multiple objects in a game room were designated as "low" severity because they did not affect the functionality of the SUT but still caused unexpected results. Most defects fit into one of these levels, but those that did not were assigned a severity level after a discussion about their importance with the development team.

The defects were also categorized by their type after consulting with the development team. All defects were placed into four groups: authentication/connection-based defects (A/C), API-based defects (API), database-based defects (DB), and missing or malformed functionality defects (FM). Each type is completely distinguishable from the others, preventing any defect-type overlap. A/C-type defects generally result from problems creating or maintaining a connection with the server or authenticating that connection. API-type defects are those that are inherent in game events used to communicate and are provided by the server development team to the rest of the project team. These defects generally cause problems with the instantiation or the data contained in the game events. DB-type defects cause problems related to the information of game objects that is stored in the database and managed by the server. FM-type defects result from the functionality that is missing, incomplete, or malformed with the game.

C. Experiment Setup and Procedures

To investigate our research question, we needed a system that contains a variety of defects and that could be modeled using a finite state machine. The Virtual Cell application facilitated these needs; in particular, the application contained 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 the modeling language.

We set up our Virtual Cell server on a dedicated machine similar to one that will host the game server once it is available to the public in order to simulate real system conditions. The server PC was running Ubuntu 8.04 with 12G RAM and an Intel Xeon E5530, with 8 cores at 2.4 GHz. The server was tested using the Crushinator running on one or two separate machines, depending on the availability of each machine. These machines consisted of a desktop, a laptop, and a virtual

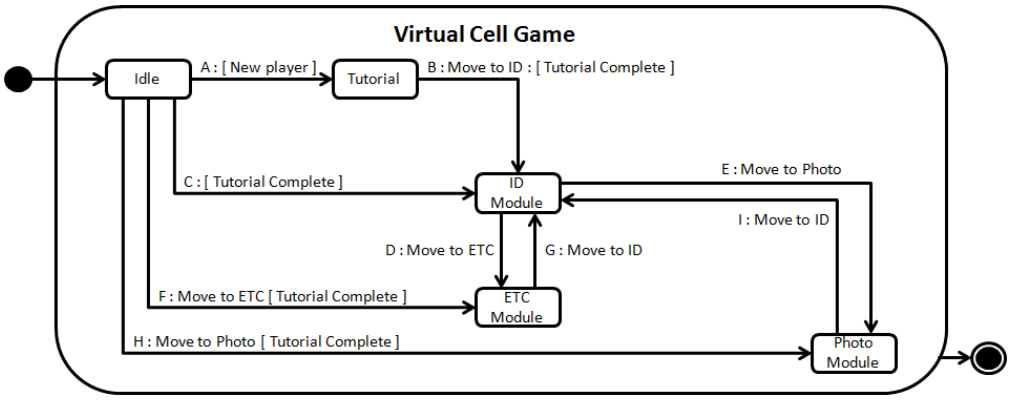

Fig. 3. Virtual Cell high-level state diagram

machine to reduce the overall duration of the experiment.

While our main focus was to collect the number of defects, we also measured the average amount of time each testing method took to execute tests on the Virtual Cell server. We set the maximum test-execution time at 30 minutes for MBET. This limitation prevented the test cases from executing a loop of transitions which would result in infinite test execution. This infinite execution is due to loops in the state machine models for all game modules. MBT test cases were not affected by these loops. The algorithm utilized to find paths in a state machine model used linearly independent paths, preventing the infinite execution of loops. Due to these finite paths, MBT test cases did not need a maximum execution time limit and generally finished within 15 minutes. Both MBT and MBET test cases sometimes terminated prematurely when a serious defect was found and prevented the test case from completing. Infeasible test cases generated by MBT also ended prematurely. As such, the test times were inconsistent and were not included in our study.

Our testing involved three separate revisions of the Virtual Cell server. To test the three versions, we followed the same guidelines utilized for previous work [24]: For MBT, a separate player was used to execute test cases in parallel. This setup simulates a situation that multiple users access the system at the same time. For MBET, three separate test iterations were executed for each module, involving 100 virtual players per iteration. Between iterations we addressed any issues with MBET configuration files to reduce any influence a malformed file may have on our experiment. The Virtual Cell can be used by multiple players independently, and for MBET tests, we selected 100 players because the server could manage 100 players without losing significant connection quality. An iteration consisted of starting 100 players at the initial state and executing the MBET test for a maximum test duration of 30 minutes. Once testing was completed for each module using both testing methods, the data were thereafter imported into the spreadsheets. These data were then reviewed to determine what defects were detected. The found defects were recorded and any necessary changes to the configuration files were calculated. Spreadsheet entry allowed filtering of the log data for certain information, making the data review quick and easy. The data collected from these tests are presented in Section V.

D. Threats to Validity

In this section, we discuss the construct, internal, and external threats to the validity of our study, as well as the approaches we used to limit these threats.

1) Construct Validity: Three issues involve threats to construct validity: the severity level, the defect type assigned to each defect, and the coverage criterion. When a defect was detected during the experiment, the severity level (low, medium, or high) and the defect type were assigned to each defect. The assignment can be subjective, thus our results can be biased by it. However, we tried to reduce this threat by consulting development team members when we assigned severity level and defect type. Further, when we generated test cases, we used one single coverage criterion, but different coverage criteria could affect our results.

2) Internal Validity: Four issues involve threats to internal validity. (1) The termination point of the MBET process could have affected our results. A player reaching the final state of a state machine model is a typical termination point for a test case. However, to prevent infinite test execution, other methods were implemented to terminate a MBET test if 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) could also be implemented for terminating MBET tests.

(2) The potential faults in the state model we used could have affected our results. To control this threat, we carefully followed the system specification when we built the models, and we validated the models using many scenarios that conformed to the specification.

(3) We attempted to simplify the state machine model by using State Machine Keywords (SMK) [24] to replace multiple similar transitions. The simplified model could have affected our results, but we carefully used SMK only when the state machine model became too complex to generate MBT and MBET test cases.

(4) Transitions were weighted using a simple heuristic value to allow MBET to select one transition over other transitions. All 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. Our weight-value selection method could have affected the results, and this threat can be addressed through additional studies with different weight-value section mechanisms.

3) External Validity: The application we used for our experiment, the Virtual Cell, is a specific type of system which best fits the MBET process. The event-driven server allows for a behavior model, such as a finite state machine, to easily represent the system, but not every system will be well represented with a behavior model. Thus, our results cannot be generalized to other application types. Further, we only used a single system in this experiment. Expanding our study with multiple systems could help us generalize this experiment to other event-driven applications.

V. Data and Analysis

In this section, we present the results of our study. We summarize the data in Tables II to V and Figure 4. We present the data as they apply to our research question, followed by the data that apply to our further interest in detecting different defect types and severities. We discuss further implications of the data and results in Section VI.

TABLE II
TEST CASES GENERATED BY MBT AND MBET

	ID	ETC	Photo	Total
MBT	267	201	451	919
MBET	300	300	300	900

TABLE III
DEFECTS DETECTED BY MBT AND MBET

	VC2.1		VC2.2		VC2.3	
	MBT	MBET	MBT	MBET	MBT	MBET
ID	3	8	4	4	3	5
ETC	8	6	4	7	6	8
Photo	1	7	6	6	5	2
Total	**12**	**21**	**14**	**17**	**14**	**15**

Table II shows the number of test cases generated by MBT and MBET. As mentioned earlier, three modules, Organelle Identification (ID), Electron Transport Chain (ETC), and Photosynthesis (Photo), were tested separately, so the table shows the number of test cases for each module. MBET produced 300 test cases per module by executing 3 test iterations of 100 players. The number of test cases produced by MBT for each module was dependent upon the paths found that cover the state machine model that represents the module. Although we tested three different revisions of the Virtual Cell server, the model for each game module did not change. This process resulted in the same number of test cases produced for all three revisions using MBT.

Table III shows the number of defects detected by MBT and MBET. To show our results visually, we also present them in boxplots as shown in Figure 4. Examining the results, we see that overall, MBET was more effective in detecting defects than MBT across all versions. In particular, for VC2.1, the difference between MBET and MBT was more outstanding; MBET detected 21 defects, and MBT detected 12 in total. When we examined the results for each module, MBET outperformed MBT for 5 out of 9 cases; MBET and MBT produced the same results for 2 cases; and MBT outperformed MBET for 2 cases.

Table IV shows the severity of defects detected by MBT and MBET. The results show that, overall, MBET was more effective for detecting high-severity defects than MBT. MBET detected 26 high-severity defects, and MBT detected 17. For medium-severity defects, MBET detected 21, and MBT detected 18; and for low-severity defects, MBET detected 6, and MBT detected 5.

When we examined the types of defects detected by MBET and MBT, shown in Table V, we observed the following trend. The A/C-type defects were the majority of those detected, in particular, by MBET. The number of API-type defects was also large compared to two other types (DB and FM). We also observed that MBET detected a noticeably large number of A/C- and FM-type defects compared to MBT. MBET detected 24 and MBT detected 14 for A/C; MBET detected 12 and MBT detected 6 for FM. MBT was slightly better in detecting the API- and DB-type defects than MBET.

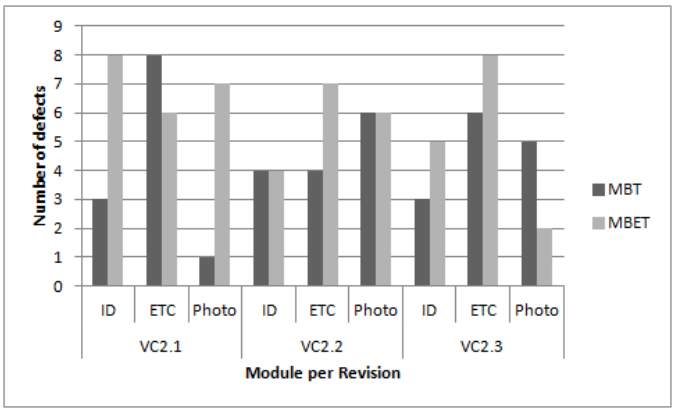

Fig. 4. Defects detected by MBT and MBET

TABLE IV
DEFECT SEVERITY LEVEL DETECTED FOR EACH MODULE

Mod.	Version	Severity Level					
		Low		Medium		High	
		MBT	MBET	MBT	MBET	MBT	MBET
ID	VC2.1	1	2	0	2	2	4
	VC2.2	0	0	2	1	2	3
	VC2.3	0	0	1	3	2	2
ETC	VC2.1	1	2	4	1	3	3
	VC2.2	0	2	3	2	1	3
	VC2.3	2	0	3	5	1	3
Photo	VC2.1	0	0	1	3	0	4
	VC2.2	1	0	2	3	3	3
	VC2.3	0	0	2	1	3	1
Total		**5**	**6**	**18**	**21**	**17**	**26**

VI. DISCUSSION

We now discuss the the results of our analysis and practical implications of these results. The results from our experiment indicate that, overall, the MBET method found more defects than the MBT method.

Figure 5 presents lineplots for the total number of defects detected by MBET and MBT for all three versions. For VC2.1, the number of defects detected by MBET is very high compared to MBT. However, we can see that, as development progresses, the number of defects found by each method tends to converge. This trend would lead us to believe that MBET can be more effective detecting defects earlier in the development process, helping reduce costs.

When we examine the results for each game module from Table III, MBET detected an equal or greater number of defects for all cases but two (VC2.1 ETC and VC2.3 Photo). Upon further investigation, we found that, for VC2.1, ETC contained more API-type defects, and MBT was able to detect half the API-type defects for that version. The API-type defects found in this experiment could have been detected from simple unit testing of the game‘s event classes. These defects are related to instantiation of the game events and accessing the data for an 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 lack of testing in mind, these defects could have been addressed before system testing and would not have been detected during our experiment.

TABLE V
DEFECTS DETECTED FOR EACH MODULE BY TYPE

Mod.	Version	Defect Type							
		A/C		API		DB		FM	
		MBT	MBET	MBT	MBET	MBT	MBET	MBT	MBET
ID	VC2.1	0	6	2	0	1	2	0	0
	VC2.2	2	1	2	2	0	0	0	1
	VC2.3	2	3	1	2	0	0	0	0
ETC	VC2.1	1	3	4	0	1	2	2	1
	VC2.2	1	1	1	2	2	1	0	3
	VC2.3	3	4	0	1	1	0	2	3
Photo	VC2.1	1	3	0	2	0	0	0	2
	VC2.2	2	3	2	2	1	0	1	1
	VC2.3	2	0	1	1	1	0	1	1
Total		**14**	**24**	**13**	**12**	**7**	**5**	**6**	**12**

Figure 6 presents lineplots for the total number of high-severity defects detected by MBET and MBT for all three versions. Similar to the results for the total number of defects, the number of high-severity defects detected by MBET was much higher than that detected by MBT for VC2.1. As development progresses, however, the gap between two methods became smaller. Again, these data led us to believe that MBET is more effective for detecting high-severity defects earlier in the development process.

As we reported in Section V, MBET was more effective for detecting A/C-type defects than MBT. (MBET detected 24 and MBT detected 14.) A/C defects that deal with player authentication and the connection to the game server contain the largest number of high-severity defects compared to the other defect types. This trend is mainly because the defects associated with connecting to the server and authenticating a player‘s login and ability to send messages are the most important part of the application. This type of defect can affect the ability of a client to connect and communicate with the server properly, thus without detecting and fixing defects, they will likely cause catastrophic system failure.

From our results, we also observed that different module development stages affected the effectiveness of the two methods. MBET was as or more effective at detecting defects in all revisions of the ID module, which was developmentally complete for all revisions. In the case of ETC, which was nearing development completion, for VC2.1, MBT was more effective than MBET, but when the ETC module had additional development for new features (VC2.2 and VC2.3), we noticed that MBET was more effective than MBT. In the case of Photo, which was just being started, overall, MBT was more effective than MBET.

It is also important to note that the two methods produced and executed test cases in different ways. MBT, for instance, executes test cases that start from the initial state and end at the final state of the model, and the expected results are then used to determine whether defects have been detected. MBET uses dynamic information about the system while the test case is being generated and executed, and it detects defects when the system fails to comply with the model. MBT test cases may contain infeasible paths, because they do not incorporate guards that might need dynamic information and may detect defects in such a path. MBET prevents infeasible paths from being created and may not detect the defects that MBT would. However, if the system conforms to the model, an infeasible path should not be possible in the SUT, thus the defects detected by infeasible paths from MBT would be false defects that do not require the tester‘s attention. For instance, until a player successfully completes a pre-requisite game task, they

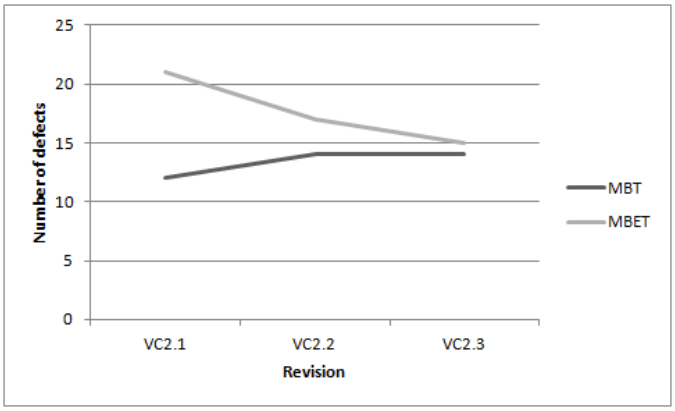

Fig. 5. Total defects found for each testing method

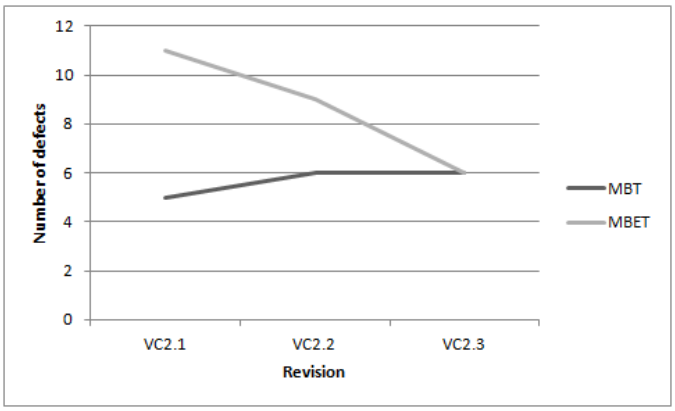

Fig. 6. Total high-severity defects found with MBET and MBT

cannot complete the task's subsequent tasks, but MBT tests could detect this type of defect, which is considered a false defect.

In summary, MBET was more effective for detecting some defect types than MBT. MBET detected more defects, overall, in all three game modules compared with MBT. The trends show that, as development progresses, the number of defects detected by each method tends to converge. Further, a 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. We also learned that, by generating a test suite using both methods to complement each other, different defect types can be detected, resulting in a more complete test coverage of the SUT.

VII. Conclusions and Future Work

In this paper, we proposed a new software testing method, MBET, and conducted an empirical study to investigate whether it can be more effective for detecting defects compared to MBT. The results of our experiment showed that MBET was, indeed, more effective overall than MBT in detecting defects. However, upon further investigation about 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, a more complete test coverage, compared to using a single testing method, can be achieved for the SUT. The combination of automated testing (e.g., MBT) and exploratory testing methods complements each other and detects defects that one testing method might miss.

The experimental processes we used suggested several avenues for future work. First, after collecting all the data, we still had to evaluate the results manually by sifting through the Crushinator's log files to determine detected defects. This process of data evaluation can be difficult and expensive in terms of time and labor. Automating the evaluation of such results could help improve the MBET process by reducing the need for time and resources. For example, by extracting the collected data for each player and comparing those data with the test case executed by that player, we can determine whether the player completed the test case successfully.

Second, to address the external threats to validity that we discussed, we plan to perform additional experiments using multiple applications and different types of event-driven systems other than game applications.

Third, we plan to investigate our approach further by considering different termination methods for MBET and different transition weighting heuristics. We used simple test-termination methods in our experiment (e.g., time limit when loops exist), but other possible termination methods can be used as we discussed in the threats to validity section.

Acknowledgments

This work was supported, in part, by NSF CAREER Award CCF-1149389 to North Dakota State University. The implementation of the MBET testing framework, the Crushinator, was supported by Award Number R44RR024779 from the National Center for Research Resources. The content is solely the responsibility of the authors and does not necessarily represent the official views of the National Center for Research Resources or the National Institutes of Health.

References

[1] A. Zylberman and N. Shenar, "Automated exploratory testing," 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] J. Bach, "Exploratory testing explained," 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 Francisco: 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. Guaf, *Implementing Automated Software Testing: How to Save Time and Lower Costs while raising quality*, 1st ed. 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*, pp. 152–158, Feb. 2011.

[8] M. Kelly, "Choosing a test automation framework," 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 *Int'l. Workshop A-MOST*, Jul. 2007, pp. 95–104.

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

[11] A. C. D. Neto, R. Subramanyan, G. H. Travassos, and M. Vieria, "A survey on model-based testing approaches: A systematic review," in *Int'l. Conf. ASE*, 2007, pp. 31–36.

[12] A. Pretschner, "Model-based testing," in *Int'l. Conf. SE*, May 2005, pp. 722–723.

[13] A. C. D. Neto and G. H. Travassons, "Supporting the selection of model-based testing approaches for software projects," in *Int'l Workshop ASE*, May 2008, pp. 21–24.

[14] P. Anitha, M. Mahesh, P. Murthy, and R. Subramanyan, "Test ready uml statechart models," in *Int'l Workshop Scenarios and statemachines: models, algorithms, and tools*, May 2006, pp. 75–82.

[15] "State machine diagrams," 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*, pp. 472–491, 2000.

[17] C. J. Nagle, "Test automation frameworks," http://safsdev.sourceforge.net/FRAMESDataDrivenTestAutomationFrameworks.htm, 2000.

[18] "Wowiwe instruction co." http://www.wowiwe.net, 2012.

[19] V. Braberman, W. Grieskamp, N. Kicillof, and N. Tillmann, "Achieving both model and code coverage with automated grey-box testing," in *Int'l. Workshop Advances in model-based testing*, Jul. 2007, pp. 1–11.

[20] D. Cohen, S. Dalal, M. Freedman, and G. Patton, "The aetg system: An approach to testing based on combinatorial design," *IEEE TSE*, pp. 437–444, 1997.

[21] M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillman, and L. Nachmanson, *Model-based Testing of Object-Oriented Reactive Systems with Spec Explorer (Lecture Notes in Computer Science)*. Berlin: Springer, 2008.

[22] 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.

[23] "Staruml 5.0," http://staruml.sourceforge.net/en, 2005.

[24] C. J. Schaefer, "Model-based exploratory testing: A controlled experiment," MS thesis, North Dakota State University, Fargo, ND, Mar. 2013.

[25] "Omg object constraint language," http://www.omg.org/spec/OCL/2.3.1/PDF, Jan. 2012.

[26] "Virtual cell (version 1)," http://vcell.wowiwe.net, 2006.

[27] "Virtual cell (version 2)," http://wowiwe.net/virtual_cell.php, 2012.