Crushinator: A game-independent testing tool framework

Christopher Schaefer *
* North Dakota State University
Computer Science
chris.j.schaefer@ndsu.edu

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

Brian M. Slator ‡
‡ North Dakota State University
Computer Science
brian.slator@ndsu.edu

***Abstract*—Testing game applications relies heavily on beta testing methods. The effectiveness of beta testing depends on how well beta testers represent the common game-application users and if users are willing to participate in the beta test. An automated testing tool framework could reduce the dependence upon beta testing by most companies to analyze their game applications. This paper presents the Crushinator as one such framework. This framework provides a game-independent testing tool that implements multiple testing methods that can assist and possibly replace the use of beta testing.**

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

I. Introduction

The testing process for game applications usually involves large amounts of beta testing. Beta testing consists of releasing a version of the application with limited functionality, a beta version, to a group of human users [1]. These users then proceed to play the game's beta version. The data are then collected from these users actions either automatically or directly from the user. Beta testing is helpful in showing how actual users interact with the application, and can be used to determine load capacities and minimal performance specifications.

Due to its benefits, beta testing has been widely used and has become a primary testing approach for checking game applications. Beta testing, however, can be inconsistent when using small groups of beta testers because those individuals do not necessarily represent the target users and because it is hard to replicate a tester's actions [2]. Also, depending on the popularity of the game application, it may be difficult to find users who are interested in testing the application's beta version. Typically, tools that support beta testing are game application-dependent, so it is hard to use the tool that has been built for a specific game application when we test different applications [3]. Therefore, developing a game-independent testing tool framework that addresses the inconsistencies of beta testing would provide great benefits for the game-application testing area [4].

In addition to beta testing tools that mainly focus on examining the functionalities of the game applications, there are tools that provide load or performance testing for them[5], [6], [7]. They simulate multiple virtual clients to determine bottlenecks in a server system. These simulated clients must, however, represent a real client as closely as possible for effective testing [8]. These tools have shown how effective automated tools can be in place of beta testing. However, merely utilizing load or performance testing techniques can limit the effectiveness of such tools because the methods do not take into account the compliance of the application to requirements not related to system performance or bottlenecks.

These issues are addressed by a few tools such as a requirements-based testing tool framework [9] and a System Under Test (SUT)-independent framework for testing GUI applications [4]. These tools show the effectiveness of frameworks for testing methods besides load and performance testing. Tools which implement multiple testing techniques have been shown to have more successful results [10]. If a testing tool framework which incorporates multiple testing methods can be developed for game applications, the dependence upon beta testing could be reduced. Further, of the tools we researched, almost all were game-dependent and could not be easily implemented for other game applications. Other tools automated portions of the testing process but not the entire process (e.g., test case generation, execution, and evaluation).

To address these limitations, we implemented a game-independent tool, the Crushinator [11]. The Crushinator provides a framework that tests event-driven, client-server based game applications and automates processes by incorporating multiple testing methods such as load and performance testing, Model-based Testing (MBT), and exploratory testing. MBT automates the testing process by using behavior models for the system being tested [12]. These models are designed to represent the behavior of the SUT, and paths through the model can then be used to generate test cases. Further, by adapting an exploratory testing method to the Crushinator, we we expand the Crushinator's functionality as a testing tool.

We applied this framework to test the Virtual Cell [13] game server, developed by WoWiWe Instruction Co. [14] to see whether the framework could be implemented easily and be an effective testing tool. This proof of concept shows that a game-independent framework that successfully incorporates MBT, exploratory testing, and load testing methods can be developed. This framework can be used in conjunction with beta testing to achieve a more complete coverage compared with only utilizing beta testing.

The remainder of this paper is structured as follows: In

Section II, we discuss the Crushinator framework, an automated test tool that implements multiple testing methods. In Section III, we describe the implementation of the Crushinator framework, including how the Crushinator tests the Virtual Cell game server. In Section IV, we present our conclusions, limitations, and possible future work for the Crushinator framework.

II. Crushinator Framework

The Crushinator is unique in its design because it incorporates Model-based Testing and exploratory testing. While, typically, most in-house game-application testing tools are dependent on the applications under test, the Crushinator provides a game-application independent testing framework. Also, most game-application testing is done as beta testing by releasing a beta version of the game for users to play and to inadvertently find defects within the system. The use of MBT and exploratory testing for game applications could possibly replace the need for beta testing or at least reduce the dependency on it. Although beta testing is a commonly used method, its general inconsistencies (e.g., participating users, replicable actions, etc.) may not be as helpful as scripted testing which can provide automated test case generation, execution, evaluation, and test-case replication.

The Crushinator was designed as a layered architecture to limit the dependencies between modules, allowing a majority of the Crushinator to be as game-independent as possible. As shown in Figure 1, this layered architecture isolates the "User Interface," "UML," and "Script" packages from the software system that is being tested (SUT). This isolation provides the game independence of the Crushinator, allowing the generic framework to be implemented rather quickly and painlessly for testing different game applications. In the figure, the layers inside the hash border are dependent upon the system under test (SUT), and the "Test Engine" and "Simulated Client" components are implemented for use directly on the SUT.

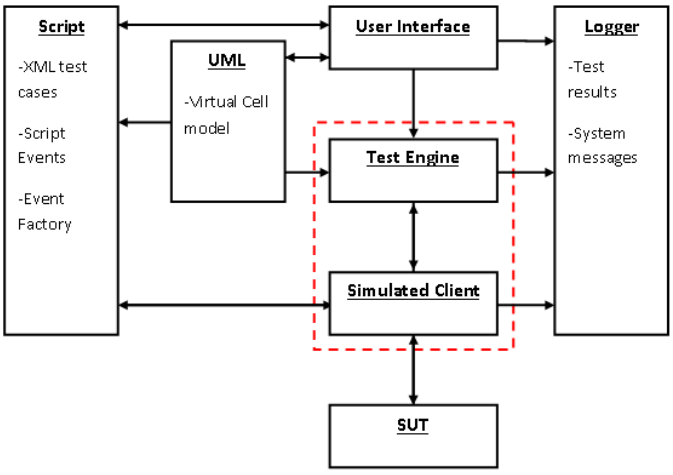

Fig. 1. Crushinator architectural diagram

The "User Interface" package handles user inputs and provides testing options for the user. The "UML" package is responsible for extracting a Unified Modeling Language (UML) state machine model from an XML Metadata Interchange (XMI) file and for saving those data in the memory for MBT use. A simplified set of controls allows a user to generate a test suite from a state machine model as shown in Figure 2. (This figure is a screen shot of the Crushinator that tests the Virtual Cell game server.) These controls allow a user to select a series of calculated paths through the model from which to generate a suite of test cases. Using the controls provided in the left panels, a user can select a state machine model from an XMI file to generate an MBT test suite. Then, selecting the path-generation criterion, a suite of paths through the model is displayed below the path criterion, allowing a user to select which paths are utilized to generate test cases. The right panel shows the contents of the XMI file or the series of paths that have currently been selected to create test cases.

The "Script" package is responsible for importing and exporting test cases from and to XML files. These "Script Event" objects can be passed from the "User Interface" down to the game-dependent packages. The "Logger" package is used by multiple layers to output changes to the Crushinator's settings, test case execution configuration, and test case results.

The "Engine," "Simulated Client," and the "SUT" are all game-dependent layers. The "SUT" layer represents the software system that is being tested. Above it, the "Simulated Client" package is responsible for connecting to the SUT, sending game events, and retrieving game responses from the SUT. The "Engine" package is responsible for managing these players as separate threads and for passing test cases down from the "User Interface" package to the "Simulated Client" package.

III. Implementation of Application-dependent Components

As explained in the previous section, to apply the Crushinator to the game applications, we needed to implement two components that are dependent on the application under test: "Test Engine" and "Simulated Client." Thus, in this section, we chose the Virtual Cell (VC) game server as an SUT and described how we implemented the two components. Table I shows the number of classes and the number of code lines for the Crushinator framework and the VC-dependent portions of the Crushinator. Both the Crushinator and VC game server were implemented in Java.

TABLE I
Software Metrics for Crushinator Application

	Lines of Code	No. of Classes
Crushinator	8065	38
VC-dependent	2637	11

For the VC game server, we implemented four separate test engines in the "Test Engine" package for performing different testing methods and for two types of connections used to

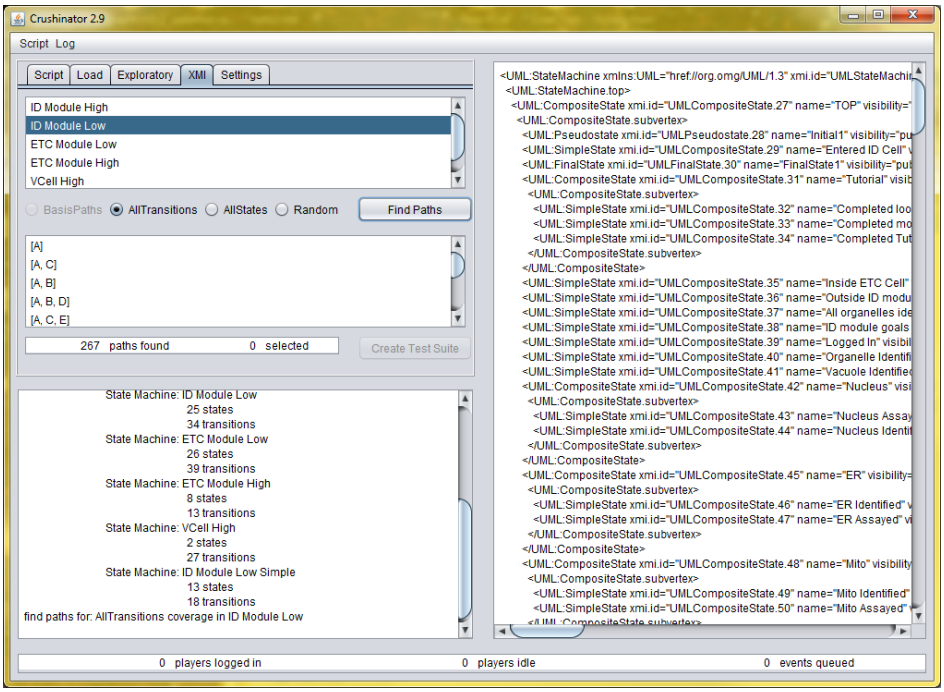

Fig. 2. Crushinator MBT controls

connect to the VC game server. They were a concrete load test engine, exploratory test engine, an HTTP test engine, and a standard socket test engine. Figure 3 shows a screen shot of the Crushinator with the left panel providing controls for the load and performance testing, Model-based Testing, and exploratory testing methods. These controls can be utilized to perform the testing methods on the SUT. Below this panel is a log output panel that displays log messages generated by the Crushinator. The right panel displays test case file contents, XMI file contents, or other important content.

We also implemented seven classes in the "Simulated Client" package that are used by the different test engines. They consisted of three concrete "Connection" classes utilized to communicate with the game server, two concrete "Player" classes utilized to mimic real players, and two additional classes to assist with handling responses from the server. In total, we needed to implement 11 classes for the "Test Engine" and "Simulated Client" packages to test the VC game server.

After implementing all those classes, we modeled the VC using UML state machine models. We then tested the VC game server using the Crushinator. During our testing, we used multiple test methods that are available for the Crushinator: load and performance testing, Model-based Testing (MBT), and exploratory testing. The Crushinator was able to simulate multiple virtual clients that were connecting and communicating with the VC game server. This simulation allowed the development team to determine minimum system requirements to run the game server along with the maximum number of players that can be connected to a single game server.

Our testing results showed that the Crushinator was also able to detect numerous defects in the VC game server by using both MBT and exploratory testing methods. More specifically, the Crushinator was able to detect 33 defects in an early development stage of the VC game server. The ability of the Crushinator to determine system requirements and bottlenecks, along with detecting defects in the VC, shows the effectiveness of using an automated tool in place of beta testing.

IV. Conclusions and Future Work

We developed the Crushinator to provide a game-independent testing tool framework to automate the process of game-application testing. The Crushinator allows a tester to automate large numbers of virtual clients utilizing different types of test methods. This automation saves time and money by reducing the dependency upon beta testing for game applications. By applying the Crushinator to the Virtual Cell game server, we found that the Crushinator was able to detect numerous defects in the game server.

Our testing experiences with the Crushinator suggested several avenues for future work. First, the Crushinator framework allows a tester to use multiple test methods (MBT, load/performance,and exploratory testing) during game application testing, but the architecture that allows such functionality is complex. Future improvement could reduce this complexity, helping simplify implementation of the framework.

Second, during implementation of the Crushinator framework, it became necessary to include the SUT-dependent events at compile-time. Although the Crushinator instantiates these events at run-time, by including these events at compile-time, the Crushinator's executable file can become much too large. Further work could determine a way to abstract these

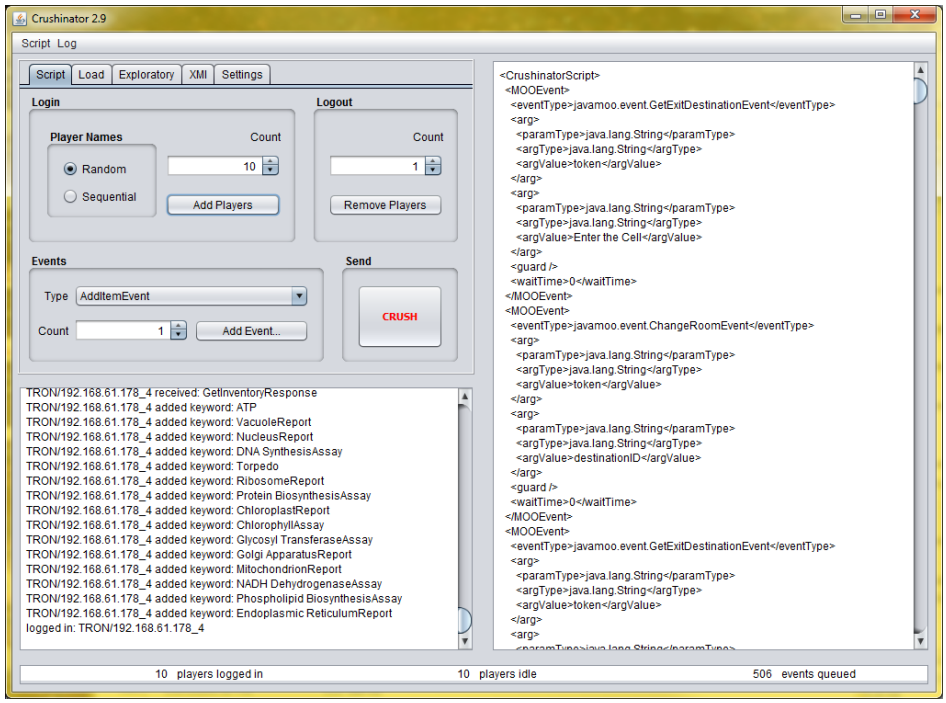

Fig. 3. Crushinator screen shot

events at compile-time, reducing the size of the distributable file.

Third, the Crushinator allows a user to generate MBT test cases from a state machine model given a certain type of coverage criteria. Currently, these coverage criteria only specify all transitions, all states, or random coverage. However, different types of coverage criteria could be incorporated into the Crushinator framework for more coverage options.

Finally, the Crushinator was implemented to test a particular application type, event-driven game servers, so other system types may not benefit from testing with the Crushinator framework. More research about testing new and different systems with the Crushinator is required.

Acknowledgments

This work was supported, in part, by NSF CAREER Award CCF-1149389 to North Dakota State University. The implementation of 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. Thanks go to Adam Jacobs for his assistance in developing the Crushinator, to Bob Cosmano for the original legacy implementation and to the staff of the WoWiWe Instruction Co. for providing a game application to test.

REFERENCES

[1] Z. Zemin, "Study on beta testing of web application," in *The 2nd International Conference on Computer and Automation Engineering (ICCAE)*, vol. 1, 2010, pp. 423–426.

[2] R. Krull, "Is more beta better? [beta testing]," in *Professional Communication Conference, Proceedings of 2000 Joint IEEE International and 18th Annual Conference on Computer Documentation (IPCC/SIGDOC 2000)*, 2000, pp. 301–308.

[3] K. M. Mustafa, R. E. Al-Qutaish, and M. I. Muhairat, "Classification of software testing tools based on the software testing methods," in *Second International Conference on Computer and Electrical Engineering*, vol. 1, 2009, pp. 229–233.

[4] T. Pajunen, T. Takala, and M. Katara, "Model-based testing with a general purpose keyword-driven test automation framework," in *EEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2011, pp. 242–251.

[5] C.-S. Cho, D.-C. Lee, K.-M. Sohn, C.-J. Park, and J.-H. Kang, "Scenario-based approach for blackbox load testing of online game servers," in *International Conference on Cyber-Enabled Distributed Computing and Knowledge*, 2010, pp. 259–265.

[6] J. Y. Kim, J. R. Kim, and C.-J. Park, "Methodology for verifying the load limit point and bottle-neck of a game server using the large scale virtual clients," in *10th International Conference on Advanced Communication Technology*, vol. 1, 2008, pp. 382–386.

[7] P. Quax, P. Monsieurs, T. Jehaes, W. Lamotte, and L. U. Centrum, "Using autonomous avatars to simulate a large-scale multi-user networked virtual environment," in *In Proceedings of the International Conference on Virtual-Reality Continuum and its Applications in Industry (VRCAI2004*, 2004, pp. 88–94.

[8] A. Denault and J. Kienzle, "The perils of using simulations to evaluate massively multiplayer online game performance," in *3rd International Conference on Simulation Tools and Techniques*, 2010.

[9] S. Mirarab, A. Ganjali, L. Tahvildari, S. Li, W. Liu, and M. Morrissey, "A requirement-based software testing framework: An industrial practice," in *IEEE International Conference on Software Maintenance*, 2008, pp. 452–455.

[10] E. H. Kim, J. C. Na, and S. M. Ryoo, "Test automation framework for implementing continuous integration," in *Sixth International Conference on Information Technology: New Generations*, 2009, pp. 784–789.

[11] A. Jacobs and C. J. Schaefer, "New crushinator (version 2.9)," http://vcell.wowiwe.net/crushinator, 2012.

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

[13] "Virtual cell," http://wowiwe.net/virtual_cell.php, 2012.

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