

Using Dynamic Models for the Evaluation of Integration and System Testing

João W. Cangussu
Department of Computer Science
University of Texas at Dallas
cangussu@utdallas.edu

Richard M. Karcich
Pillar Data System
Longmont, CO 80503
rkarcich@pillardata.com

Abstract

Side effects of one phase of the software development process are known to affect the subsequent phases. In this paper we analyze such side effects with respect to two consecutive phases: integration testing and system testing. The analysis is conducted based on a discrete event simulation model and focus on effort and effectiveness of integration testing and their effect on system testing.

1 Introduction

Testing can be used to assess how good software is, or to find faults and thus improve the software. This paper focuses on this second use of testing, and on a specific meaning of “improvement”, i.e., “making the software more reliable” while making better use of the available resources. There are many testing methods, and strong opinions on their relative merits, but empirical quantification of these opinions is difficult and hard to generalize. Rather, dynamic models are able to provide valuable insight about what we should expect from practical applications of testing, and what we should measure to guide the choice of V&V techniques that can be applied. One problem is to choose between testing methods in terms of the project risk that each implies, i.e., the risk of delivering a product of sub-standard reliability. The second problem is how best to combine different methods in the V&V of a product. There is a dilemma between applying diverse methods to take advantage of their different strengths, and looking instead for one best method and concentrating all resources on applying that method alone. The models explain how to resolve this dilemma: when is it that diversity pays off even if it means using methods that, on their own, are inferior, and which measures are needed to support such a decision.

One of the major difficulties stifling the productivity of software testing process is attributable to the process of software evolution. Software systems can evolve very rapidly during their development. Thus, the object of the test pro-

cess is liable to change very rapidly during the software testing process. No software test process can begin to be adequate unless the infrastructure is present to insure that the tests being executed today, in fact, reflect the status of the system as it is right now. The source code base may change substantially in a very short period. As it does, the operational specifications and functional specifications must also change to maintain complete specification traceability. A dynamic model can again be used to analyze the best alternatives in a constantly changing environment. In this paper we focus on the analysis of integration testing and system testing sharing the same debugging process. A discrete event simulation model is created for the phases and the results are analyzed.

The remainder of this paper is organized as follows. Section 2 presents a brief description of the testing process used in this study. A discrete event simulation model for a system testing/debugging process is presented in Section 3 while the results of the simulation runs are described in Section 4. Section 5 presents the concluding remarks.

2 Testing Process

The simulation model used here was created for the testing process of a specific company. Therefore a brief description of the overall testing process and severity classification used at the company is provided next. The details of integration, system testing, and debugging are left out as they follow the explanation of the simulation model in Section 3.

The company applies several testing techniques to the problem of verifying/validating its products. These include, but are not limited to, the classic models of unit and integration testing, as well as other more specialized approaches. The company utilizes a variety of techniques to test their products. To some extent, the first four of these can be arranged on a continuum of progressive complexity from low level unit testing to high level full integration testing. The last three are specialized for the particular product, and fit into the middle to high end of the continuum. Progress-

sive complexity is a testing philosophy emphasizing testing a product as early in the development process as possible and in the simplest controlled environment in which the elements of the product are functional. The objective is to facilitate the problem discovery and diagnostic process by discovering problems in an environment with the fewest number of unknowns.

Integration testing is accomplished by testing/debugging merged components and then promoting the successfully-tested merged-components to a full-build. The new internal build is shipped to system testing where regression testing is applied. A severity classification is applied both for integration and system testing. It has been show that the number of defects with high severity impacts the completion time of the system testing phase. The same result can be extrapolated to integration testing. The classification of defects with respect to severity is accomplished here by the use of five classes. Severity 1 is the most severe resulting in a temporary interruption of the testing process while Severity 5 represents the least severe defects.

According to the data collected, the majority of the defects fall into severity classes 1, 2, and 3 with a very small number associated with classes 4 and 5. A similar behavior has also been reported by Ostrand and Weyuker [5, 6]. The collected data also shows that defects classified as severity 1 occurs only once per time unit. This behavior is expected since severity 1 defects causes a temporary shut down of the testing process. If debugging cannot be done immediately or if it takes long to fix the problem, this will prevent resumption of the testing process and delay finding more defects, including additional severity 1 defects. Severity 2 defects present a higher frequency mainly due to its larger number and the fact that they do not cause any interruption on the process.

3 Simulation Model

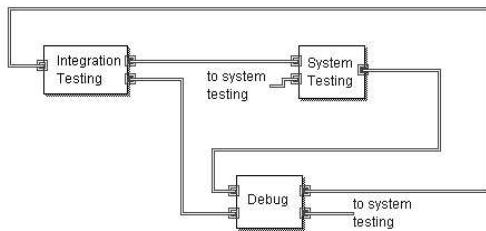


Figure 1. Top level model for integration and system testing and the debugging process.

A discrete event simulation [1] model, implemented using the Extend 6 tool [4], for a integration testing, system testing, and debugging process is presented in Figure 1.

As can be seen, both integration and system testing share the same debugging process. The probabilistic distributions used in the model are based on the behavior of actual testing processes. However, specific details are left out due to proprietary reasons.

The model for integration testing is presented in Figure 2. New test cases are generated according to a normal distribution on the “Generate Test Cases” block on the top of the figure. A group of test cases is held on the “Holding” block simulating the arrival of a new component to be integrated to the system. After that, the test cases are released and the integration testing process starts for that component. The new test cases are combined with test cases selected from regression testing and the verification of failed test cases originated from the debugging model. It is assumed here that new test cases have a lower priority when compared to test cases from regression and verification. All test cases are stored in a non-preemptive priority queue. A pool of testers is available at the “Testers” block. As soon as a test case is available, one tester is allocated to execute and verify the test case. The time associated with this task is determined at “Testing” block according to a normal distribution specified in the “Testing Time” block. It is assumed here that, in general, the severity of defects has no impact on the time required to execute and evaluate the test case. After completing the test case, the tester goes back to the pool and can start working on another test case. In the model, it is assumed a 30% chance that a test will discover a defect. Depending on the testing process organization, this percentage tends to decrease as the process proceeds presenting an exponential decay. Though, we have assumed a fixed failure rate, the same decay is observed as the number of test cases flowing through this block presents an exponential decay [2, 3]. Out of the 70% of successful test cases, 20% are selected for regression test and are sent back to the testing queue. The other test cases are split into two groups. The first group represents test cases that will have no effect on system testing and the second the ones that will have an effect. A decision block with a 50% chance of going to either one of the groups is presented in Figure 2. This percentage represents the effectiveness of the integration testing with respect to side effects on system testing and can be changed to simulate distinct scenarios as described in Section 4. The test cases affecting system testing are sent to the next phase through the connector “Con1Out” seen in Figure 2. The severity of the failed test cases are defined at the “Set Severity” block according to the percentage specified in the two lower input parameters. If a severity 1 defect is determined, the pool of testers is shut down, temporarily stopping testing. The severity is then used to prioritize the debugging process, severity 1 defects are debugged first. To simplify the model, only three severity classes are used here. The impact of this simplification is minimal since the number of

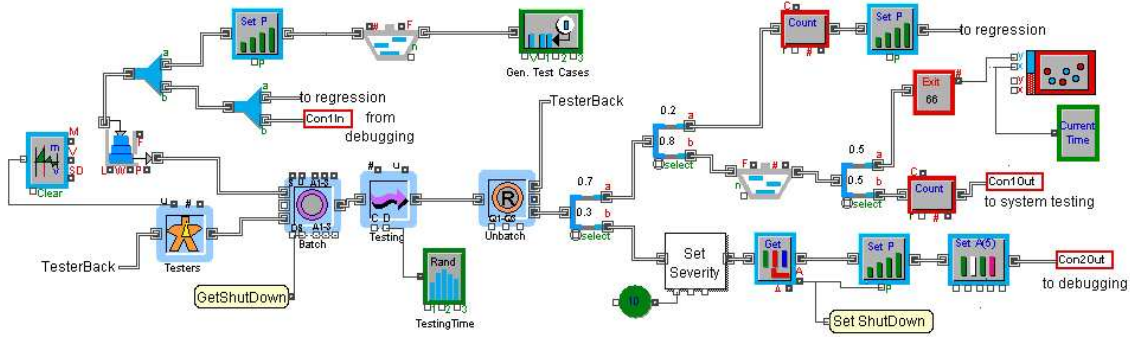


Figure 2. Discrete event simulation model, implemented in Extend v6, for the integration testing phase of a software development process.

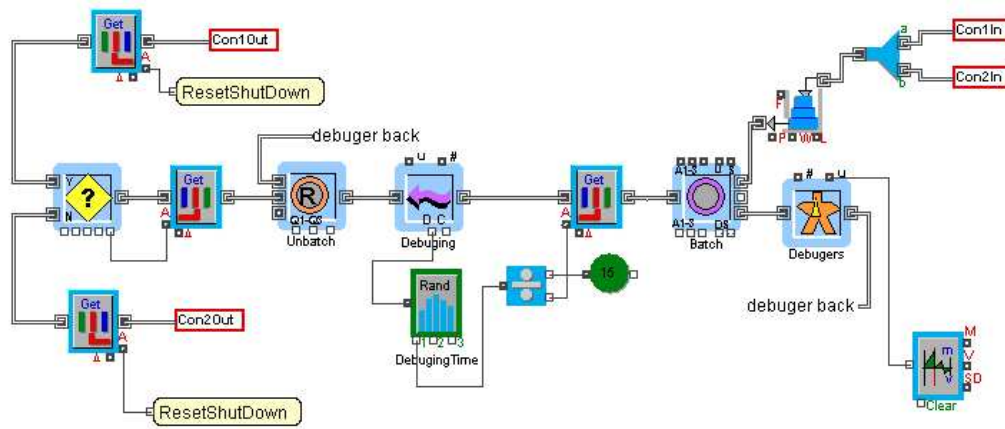


Figure 3. Discrete event simulation model, implemented in Extend v6, for the debugging process.

severity 4 and 5 defects found on the testing process under consideration is almost insignificant when compared to the first three severity classes.

The model of the debugging process is shown in Figure 3. As can be seen, failed test cases from integration and system testing are combined into a single priority queue. The priority is based on the severity of the defects and not on its source. That is, defects with same severity are served in a FIFO (first come first served) strategy independent if they are originated from integration or from system testing. Such a scenario is analyzed in Section 4. The structure for debuggers is the same as for testers. However, the debugging time now is dependent on the severity of the defect. The debugging time is inversely proportional to the severity class. That is, severity 1 defects present a higher debugging time than severity 2 that in turn consumes more time in debugging than severity three defects. More severe defects generally necessitate involvement of more people, spending more time to resolve the defect. Once a severity 1 defect has been debugged, the testers pool is made available again and

the testing process can resume. The fixed defects are split according to their origin, integration or system testing, and then fed back to the respective testing queue.

A model for the system testing, presented in Figure 4, has also been developed. The model is similar to the one in Figure 2 as it also presents the pool of testers and failed test cases are sent to the debugging process. There are two major differences between the models in Figures 2 and 4. The first is that successful test cases not selected for regression just exit the system. The second is the existence of two regression paths, one for regression of the system testing itself and another when a new build originated from the integration of a new component has arrived. The simulation stops when all the test cases exit the system, i.e., all have been successful.

4 Analysis of the Simulation Model

The availability of a dynamic model representing the behavior of integration and system testing as well as the de-

Effectiveness of Integration Testing	Sensitivity Value
	Completion Time
70%	-0.50
60%	-0.42
50%	-0.59
40%	-0.47

Table 1. Sensitivity results for completion time when the effectiveness of integration testing ranges from 80% to 40%

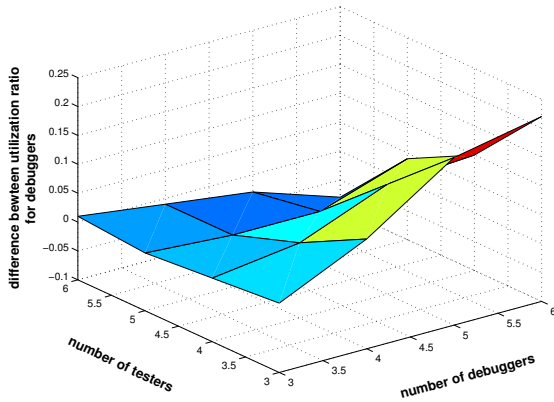


Figure 6. Difference in the debuggers' utilization ratio for changes in the number of integration testers and system testers as the number of debuggers change.

changing the number of integration testers, system testers are kept constant at 5. The same is true when the changes are made with respect to system testers. Assume matrix $DU_{4 \times 4}^{int}$ represents the utilization ratio for debuggers ranging from 3 to 6 while the number of integration testers also ranges from 3 to 6. $DU_{4 \times 4}^{sys}$ presents the same values associated with system testing. Computing the difference $DU_{4 \times 4}^{int} - DU_{4 \times 4}^{sys}$ produces the results in Figure 6 where it can be seen that the difference in utilization ratio increases as the number of testers and debuggers increase. That is, increasing the number of integration testers has a larger effect on the utilization ratio of debuggers than increases in the number of system testers.

When the difference above is computed for the utilization ratio of system testers an almost constant value is observed as the number of debuggers increase. However, when the number of system testers increase, their utilization ratio decreases and the difference when compare to increases in the number of integration testers also increases. This behavior can be observed in Figure 7.

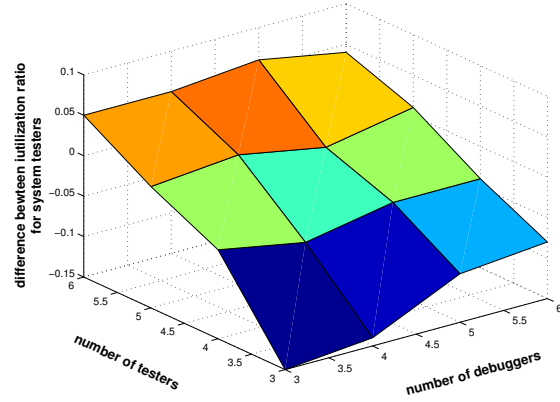


Figure 7. Difference in the system testers' utilization ratio for changes in the number of integration testers and system testers as the number of debuggers change.

Figure 8 and 9 shows the completion time associated with changes in the number of integration and system testers, respectively. Increases in the number of integration testers and debuggers shows a decrease in completion time. However, increases in the number of system testers increases the completion time. The frequency of severity 1 defects from system testing increases with more testers doing system testing. Since these defects have a higher priority than severity 2 and 3, they will delay the termination of integration testing and consequently delay the completion time for the entire process.

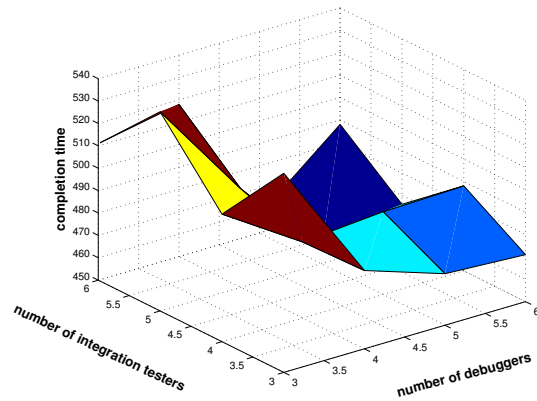


Figure 8. Completion time for changes in the number of debuggers and integration testers.

As stated before the severity of defects determines their priority in the debugging queue. Severity 1 defects have the highest priority while severity 3 have the lowest. Defects of same severity are served in a first come first served (FIFO)

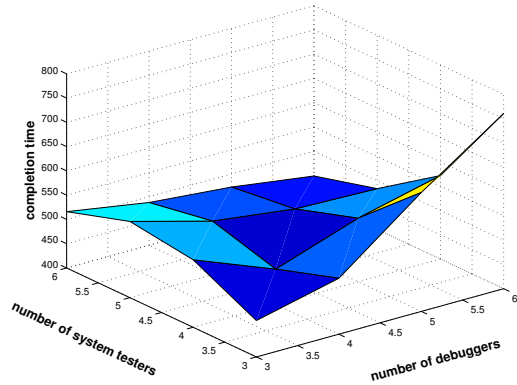


Figure 9. Completion time for changes in the number of debuggers and system testers.

basis independent of their origin. The results of simulations runs for this scenario are shown in the first line of Table 2. Now let us consider scenarios where priority is given to either defects from integration or system testing. That is, if priority is given to integration, defects of the same severity from integration are served first versus if they were originated from system testing. The results from Table 2 show a clear improvement in the process when priority is given to integration testing defects. Completion time decreases more than 30% when prioritizing integration testing while a increase of 13% is observed when given priority to system testing. System testing cannot be finished before integration testing is completed. Giving higher priority to system testing increases the delay in integration testing that consequently increases the overall completion time.

Priority	CT	ITUR	DUR	STUR
Severity Only	457	0.19	0.88	0.54
Integration Test	318	0.26	0.85	0.88
System Test	516	0.28	0.67	0.56

Table 2. Different priority strategies and the corresponding completion time (CT), and utilization ratios for integration testers (ITUR), debuggers (DUR), and system testers (STUR).

5 Concluding Remarks

The use of a dynamic model for integration and system testing and debugging has allowed for the analysis of distinct scenarios to justify possible changes in the process. The results have show that, under certain circumstances, increasing work force on system testing may increase the completion time. Only if integration testing has been al-

ready completed such changes will have a positive impact. Since system testing depends on integration testing, investments are more justifiable for the latter. Any improvement on integration testing will have a positive effect on system testing and consequently on the entire process. Also, when debuggers are the same for integration and system testing, prioritization of defects from integration shows a considerable improvement in the completion time of the process.

The results produced here are based on a specific process and therefore cannot be generalized, though some of them, such as the priority for integration testing, seems to be true for most of processes. The model needs to be adjusted according to the specifics of a process. This includes not only changes in the tasks simulated by the model but also on the distributions associated with them.

References

- [1] J. Banks, J. S. C. II, B. L. Nelson, and D. M. Nicol. *Discrete Event System Simulation*. Prentice Hall International series, Upper Saddle River, NJ, third edition, 2001.
- [2] J. W. Cangussu, R. A. DeCarlo, and A. P. Mathur. A formal model for the software test process. *IEEE Transactions on Software Engineering*, 28(8):782–796, August 2002.
- [3] J. W. Cangussu, R. A. DeCarlo, and A. P. Mathur. Monitoring the software test process using statistical process control: A logarithmic approach. In *Proceedings of joint 9th European Software Engineering Conference (ESEC) and the 11th SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, pages 158–167, Helsinki, Finland, September 1-5 2003. ACM SIGSOFT.
- [4] Imagine That, San Jose, CA. *Extend v6 User’s Guide*, 2002.
- [5] T. J. Ostrand and E. J. Weyuker. The distribution of faults in a large industrial software system. In *ISSTA ’02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 55–64. ACM Press, 2002.
- [6] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Using static analysis to determine where to focus dynamic testing effort. In *Second International Workshop on Dynamic Analysis, co-located with the 26th International Conference on Software Engineering (ICSE 2004)*, pages 1–8, Edinburgh, Scotland, May 2004.
- [7] A. Saltelli, K. Chan, and E. M. Scott, editors. *Sensitivity Analysis*. John Wiley & Sons, Chichester, New York, 2000.