

# Modeling Software Testing Processes

James S. Collofello  
Zhen Yang, John D. Tvedt  
Derek Merrill, Ioana Rus

Computer Science and Engineering Department  
Arizona State University  
Tempe, Arizona 85287  
(602) 965-3190, collofello@asu.edu

## Abstract

The production of a high quality software product requires application of both defect prevention and defect detection techniques. A common defect detection strategy is to subject the product to several phases of testing such as unit, integration, and system. These testing phases consume significant project resources and cycle time. As software companies continue to search for ways for reducing cycle time and development costs while increasing quality, software testing processes emerge as a prime target for investigation. This paper proposes the utilization of system dynamics models for better understanding testing processes. Motivation for modeling testing processes is presented along with an executable model of the unit test phase. Some sample model runs are described to illustrate the usefulness of the model.

## Motivation

Software testing activities consume a significant amount of project resources and development cycle time. Typical testing costs vary based on the criticality of the project from 40% to 85% of the development life cycle [4]. In light of these expenditures, any efforts to reduce development cycle time or cost must include the testing process. Our research is addressing this concern through the development of system dynamics models of testing processes. These models enable the testing processes to be better understood by clearly identifying testing activities, their dependencies and the feedback between various testing tasks. Through the addition of metric data, the process models can be executed to reveal areas for increasing the cost effectiveness of the testing activities. The models also provide an experimental vehicle for performing various "what if" type scenarios and assessing their impact on testing cost effectiveness. For example, a system dynamics model of the testing process can reveal the differences in cycle time of projects with minimal unit testing, unit testing which detects 70% of the defects and unit testing which detects 100% of the defects.

## Overview of System Dynamics Modeling

System dynamics modeling was developed in the late 1950's at M.I.T. It has recently been used to model "high-level" process improvements corresponding to SEI levels of maturity [5, 3]. System dynamics models differ from traditional cost estimation models, such as COCOMO [2], in that they are not based upon statistical correlations, but rather cause-effect relationships that are observable in a real system [1]. An example of a cause-effect relationship would be a project behind schedule (cause) leading to hiring more people (effect). These cause-effect relationships constantly interact while the model is being executed, thus the dynamic interactions of the system are being modeled, hence its name. A system dynamics model can contain relationships between people, product, and process in a software development organization. The most powerful feature of system dynamics modeling is realized when multiple cause-effect relationships are connected forming a circular relationship, known as a feedback loop. The concept of a feedback loop reveals that any actor in a system will eventually be affected by its own action. A simple example of the ideas of cause-effect relationships and feedback loops affecting people, product and process can be illustrated by the following scenario:

Consider the situation in which developers, perceiving their product is behind schedule (cause), modify their development process by performing less thorough unit testing (effect/cause), leading to a product of lower quality (effect/cause), but giving the temporary perception that the product is back on schedule (effect/close feedback loop).

The developers perceived the product to be behind schedule, took action, and finally perceived the product to be back on schedule, as a result of their actions. Secondary effects due to the developers' actions, however, such as the lower quality of the product, will also eventually impact the perception the developers have as to being on schedule and will necessitate

further actions being performed. These cause-effect relationships are explicitly modeled using system dynamics techniques.

Because system dynamics models incorporate the ways in which people, product, and process react to various situations, the models must be tuned to the environment that they are modeling. The above scenario of developers reacting to a product that is behind schedule would not be handled the same way in all organizations. In fact, different organizations will have different levels of productivity due to the experience level of the people working for them and the difficulty of the product being developed. Therefore, it is unrealistic for one model to accurately reflect all software development organizations, or even all projects within a single development organization.

Once a system dynamics model has been created and tailored to the specific development environment, it can be used to find ways to better manage the process to eliminate bottlenecks and reduce cycle time. Currently, the state-of-the-practice of software development is immature. Even immature software development environments, however, can benefit from this technique. The development of the model forces organizations to define their process and aids in identifying metrics to be collected. Furthermore, the metrics and the models that use them do not have to be exact in order to be useful in decision-making [6]. The model and its use result in a better understanding of the cause-effect relationships that underlie the development of software. The power of modeling software development using system dynamics techniques is its ability to take into account a number of factors that affect cycle time to determine the global impact of their interactions, which would be quite difficult to ascertain without a simulation. The model may be converted to a management flight simulator to allow decision-makers the ability to perform experiments in their simulated development environment.

## Modeling the Unit Test Phase

Our initial research efforts in the testing area focused on modeling the unit test phase. We chose the unit test phase both because it is the most well-understood of the testing phases as well as one of the most controversial. The controversy in the unit test phase revolves around the amount of unit testing that is performed. Although rigorous unit testing is recommended by many development standards, individual projects have been completed with various levels of unit testing dependent upon the other quality assurance tasks performed and the difficulty of creating a unit test environment. To investigate the impact of these various degrees of unit testing on software

development cycle time, we developed a model of the unit test phase. This model assumes that the unit test phase begins after clean compilation and completes when the unit test criteria have been met and all defects have been fixed. It is important to note that we view the unit test phase as including both defect detection and repair. Repair consists of reworking the code to remove the detected errors and retesting the code to verify the errors were removed. In order to model the impact of various unit test strategies, we also include a defect leakage cost in our model which addresses the cost of repairing defects missed by the unit test phase.

The basic inputs to our model are described below:

1. test size: the size of the unit test activity measured in lines of code to test
2. test thoroughness: the thoroughness of the testing activity defined as the percentage of defects detected by the testing
3. quality of code: defined as the number of defects per KLOC which are detectable by the unit testing
4. daily mp: the number of developers available for performing unit testing activities
5. rework prod: the number of errors fixed per developer-day
6. cost to fix later: the number of developer-days needed in a later test phase to fix an error missed by unit testing

The model outputs consist of:

1. total time for unit test: defined as the total number of days needed to complete the unit test phase
2. total cost for unit test: defined as the total number of developer-days needed to complete the unit test phase
3. leakage penalty: defined as the number of developer-days needed to repair the defects not detected during unit testing

A simplified view of our system dynamics model is presented in Figure 1 utilizing *ithink* notation [7]. The model illustrates code errors being detected based on an error detection rate which is dependent upon the testing rate, the quality of the code and the thoroughness of the testing. The thoroughness of the testing in turn affects the time needed to perform the

testing. The model also illustrates the rate that detected errors are fixed which is dependent upon the percentage of developer time available for defect repairs, the number of available developers and the

rework productivity. Defect leakage is also modeled along with the increased cost of repairing in later phases defects which were not detected by unit testing.

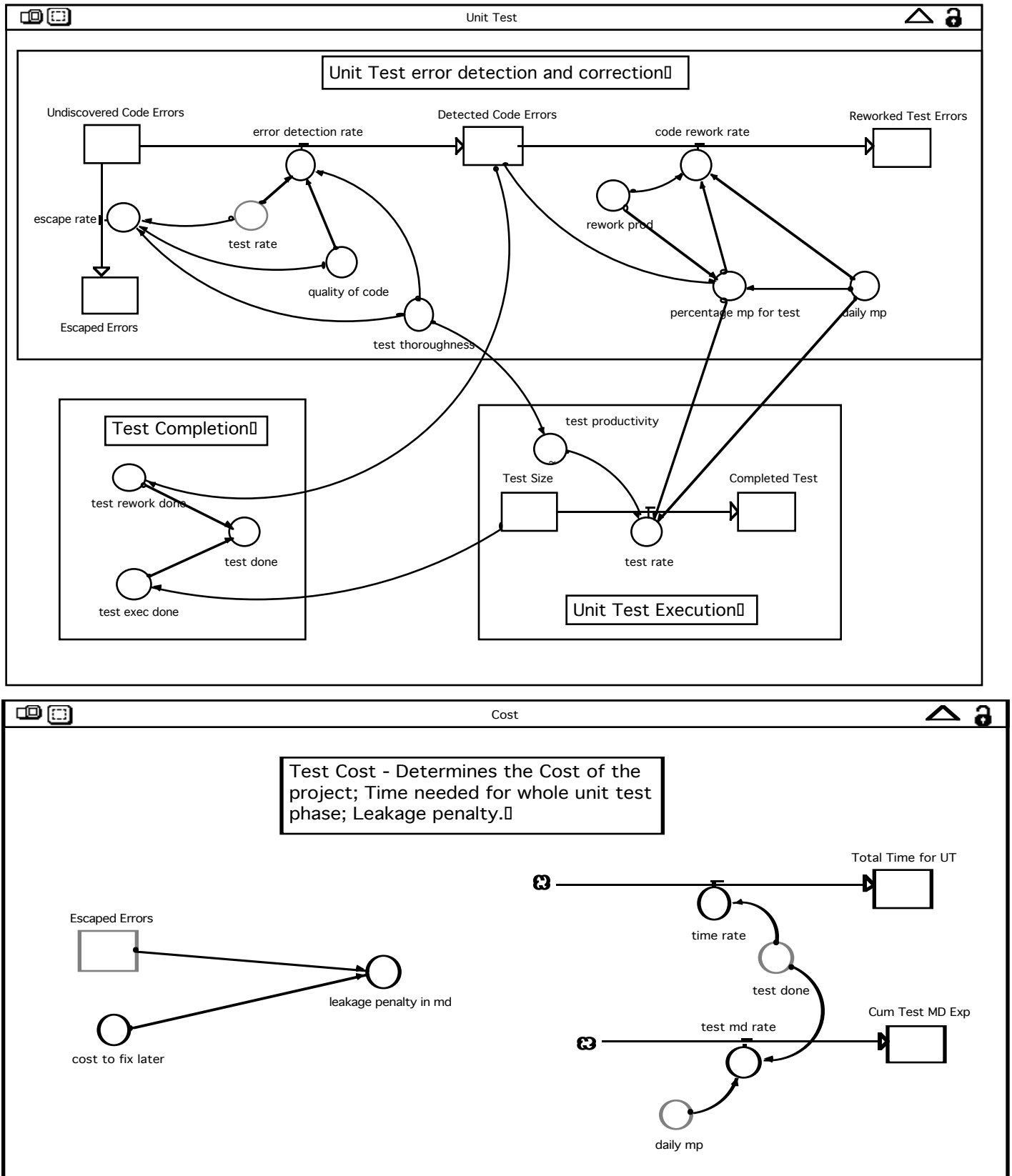


Figure 1 Unit Test Model.

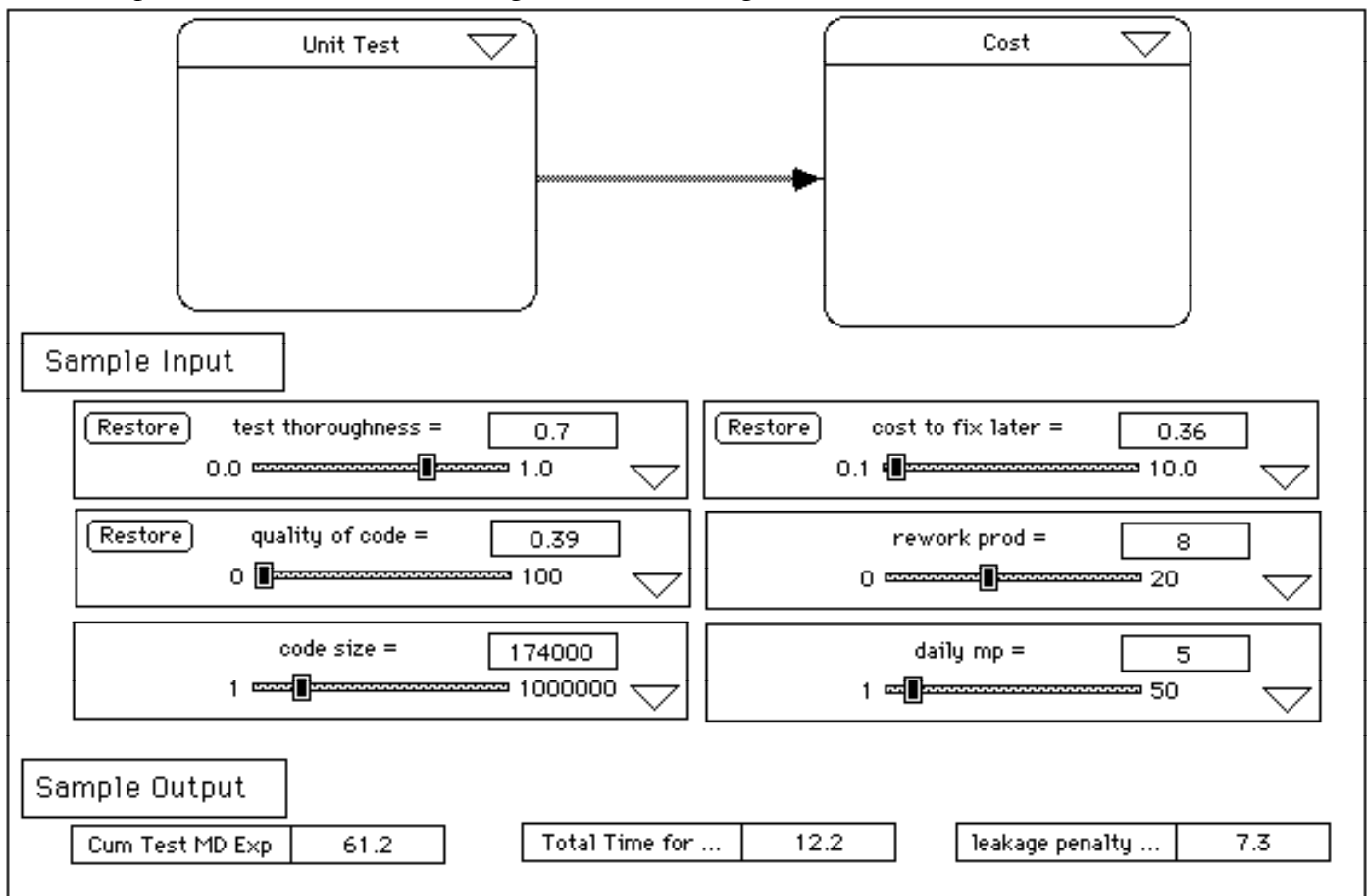
To illustrate the kind of information which can be produced by this model, we extracted unit test data from an engineering organization. Three scenarios were executed with various levels of unit test thoroughness. The levels of test thoroughness were:

- 0.1: corresponding to very minimal unit testing
- 0.7 corresponding to a level of test thoroughness in which 70% of detectable defects were detected
- 1.0 corresponding to an idealized level of thoroughness in which all defects were detected.

1. test size: 174,000 assembly equivalent lines of code
2. quality of code: 0.39 defects per KLOC
3. daily mp: 5 developers available for performing unit testing activities
4. rework prod: 8 errors fixed per developer-day
5. cost to fix later: 0.36 developer-days needed to fix an error missed by unit testing in a later test phase

The model user can input the values for a particular scenario using the interface provided by the model in Figure 2.

The other inputs were fixed at the following values:



**Figure 2** Model User Interface

The results for each of the scenarios are presented in Table 1.

To interpret the cost effectiveness of the unit test activity it is necessary to combine the columns for Total Cost for Unit Test and Leakage Penalty. For this particular organization's project scenario, the results indicate the benefit of reducing the unit testing effort.

This can be explained by the low cost to fix a defect not detected during unit testing as determined by the metrics input to the model. Obviously these results will not apply to all projects since variations of the input parameters will significantly alter the Total Cost for Unit Test and Leakage Penalty. For instance, when the cost to fix a defect not detected during unit testing is 1.0 errors per developer-day a test

thoroughness goal of 0.7 results in a lower overall cost.

Test Thoroughness	Total Time for Unit Test	Total Cost for Unit Test	Leakage Penalty
0.1	8.5	42.5	22.3
0.7	12.2	61.2	7.3
1.0	50.0	250.0	0.0

**Table 1** Results of Varying Test Thoroughness on Unit Test

### Conclusion and Future Research

The development of our unit test phase system dynamics model has increased our understanding of unit testing and defect repair activities and their relationships. The model provides a framework for interpreting testing metrics and analyzing areas for optimizing testing processes. We are currently in the process of calibrating our testing model with actual industry metrics in order to provide projects with guidance on selecting their testing strategy. Our future plans are to expand our modeling to include the integration and system test phases. Our testing models will then be integrated with our incremental software development system dynamics model in order to more accurately assess the impact of testing activities in an incremental development environment [8].

### References

- [1] Tarek Abdel-Hamid and Stuart E. Madnick, *Software Project Dynamics An Integrated Approach*, Prentice-Hall, Englewood Cliffs, New Jersey, 1991.
- [2] Barry W. Boehm, *Software Engineering Economics*, Prentice-Hall, Englewood Cliffs, New Jersey, 1981.
- [3] Mark C. Paulk, Bill Curtis, Mary Beth Chrissis and Charles V. Weber, "Capability Maturity Model, Version 1.1," *IEEE Software*, July 1993, pp. 18-27.
- [4] Roger S. Pressman, *Software Engineering A Practitioner's Approach*, McGraw-Hill, 1992.
- [5] Howard A. Rubin, Margaret Johnson and Ed Yourdon, "With the SEI as My Copilot Using Software Process 'Flight Simulation' to Predict the Impact of Improvements in Process Maturity," *American Programmer*, September 1994, pp. 50-57.

- [6] George Stark, Robert C. Durst and C. W. Vowell, "Using Metrics in Management Decision Making," *IEEE Computer*, September 1994, pp. 42-48.
- [7] *ithink Manual*, High Performance Systems, Inc., Hanover, NH.
- [8] John Tvedt and James Collofello, "Evaluating the Effectiveness of Process Improvements on Software Development Cycle Time via System Dynamics Modeling", *Proceedings of COMPSAC*, 1995, pp. 318-325.