Integrating Process Simulation and Reliability Models

Ioana Rus
Fraunhofer Center for Experimental Software Engineering

James Collofello
Arizona State University

The problem addressed here is predicting how the reliability of a software product, as well as project cost and schedule, will be influenced by adopting different defect prevention and detection activities. The solution we propose using is a software process simulator for estimating the impact of different software reliability engineering practices. We have created a prototype simulation model of the dynamics of defect evolution (introduction, detection, and removal) and of the process factors that influence it throughout the entire development lifecycle. Our simulation model relates defects to failure occurrences by integrating existing reliability estimation and prediction models in the system-testing component of our system dynamics simulator. This article describes our model and presents a hypothetical example illustrating how an organization could use our simulation tool to make decisions regarding reliability strategies. The model is available free of charge to readers willing to work with us on our ongoing research on the impacts of process choices on reliability, cost, and schedule.

A reliability strategy is a set of software engineering practices defined for each project by combining different reliability achievement and assessment activities and methods, according to the software reliability goal and project's characteristics. In [1] is a description of a decision-support system for reliability strategy selection based on a set of product, project, and resources decision factors.

There are two main approaches to achieving high software reliability:
1. Avoiding defects in the final product.
2. Using fault tolerance methods.

Fault avoidance can be achieved by using fault prevention and fault detection and correction methods. Fault tolerance allows the system to continue to operate in the presence of latent faults, enabling the whole system to function as required.

Once the reliability strategy is selected, it must be assessed in terms of the projected product reliability, budget, and schedule constraints. One approach for strategy evaluation is process modeling and simulation, which involves analyzing the software development process, creating a model of the process, and then executing the model and simulating the real process. The simulation model presented here is meant to be a tool that supports reliability prediction as well as cost and schedule estimation. It helps the user to forecast the impact of different reliability practices not only on software reliability (effectiveness), but also on cost and development time (efficiency).

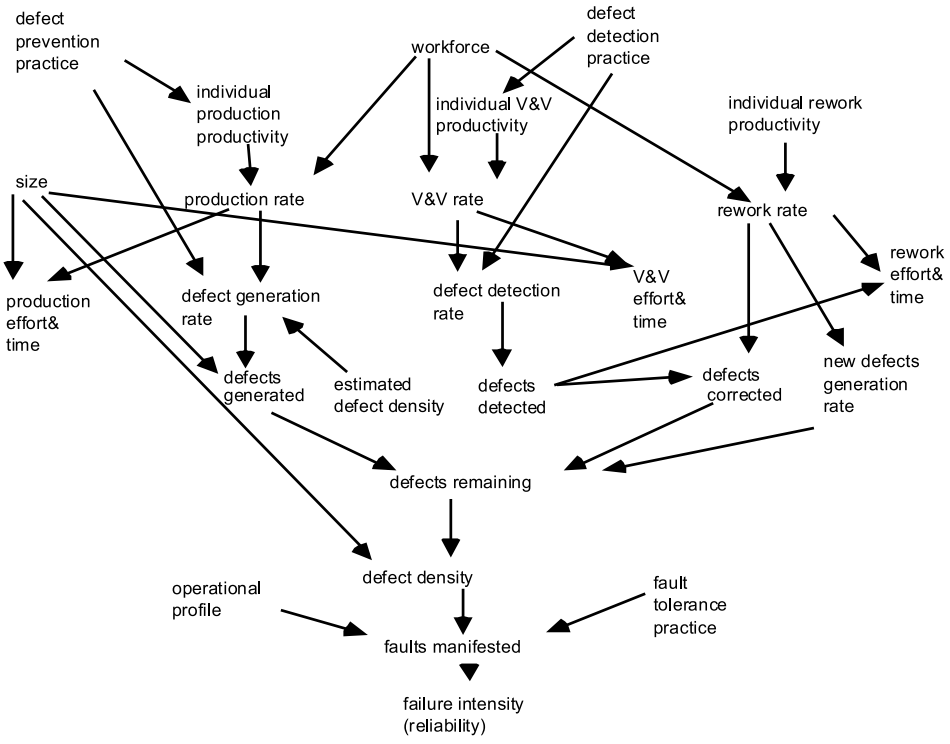
Reliability Practices

A model is an abstraction of a real object or system. Modeling a system means capturing and abstracting the system's components, relationships, and behavior, according to the modeling objective. The goal of developing this model was to capture the impact on quality, cost, and development time and effort of different software reliability practices during the development process of a software product. Currently our prototype model addresses new product development from inception to delivery and does not address maintenance.

The influence of defect prevention, detection, and fault tolerance practices on product and process variables captured in our model are shown in the influence diagram in Figure 1. Defect prevention practices contribute to the reduction of defects injected in the product but also might reduce the development productivity. Therefore, applying these practices could take more time and effort in production, but the time and effort spent later for detecting and reworking defects is reduced. Defect detection practices increase the number of defects detected and do not let them propagate to subsequent phases, where they are more expensive to remove.

Reducing the number of remaining defects in the final product results in increased reliability manifested in decreased failure intensity. Fault tolerance practices mask some of the remaining defects and do not allow them to manifest when the software is executed, so the number of failures is reduced. Testing with operational profiles will uncover the faults that are more likely to be encountered in

Figure 1. *Influence Diagrams*



the operational phase. More details about modeled variables relationships, inputs, and outputs can be found in [2].

Our model was designed to be modular; it consists of components corresponding to the requirements, design, coding, and system testing phases. Each component addresses both managerial and development aspects of each phase. Managerial aspects include effort and staff allocation, activity duration, cost, activity progress, and productivity for production, quality, and rework activities. Development aspects include modeling work product production as well as defect flows.

Figure 2 illustrates our modeling of the system-testing phase. The activities are test case execution, fault identification, rework, and regression testing. Test cases are executed in order to uncover remaining defects given testing equipment constraints (e.g., CPU execution time). When a failure is encountered, the corresponding faults and defects are identified (fault identification) and then corrected (rework). It is assumed that regression testing is performed periodically to detect possible new defects introduced (bad fixes) that are also subsequently corrected. Defects that do not manifest by causing failures and therefore are not detected during testing will remain in the delivered product, affecting the field value of software reliability.

System Dynamics Modeling Simulator

Our model was implemented using the system dynamics modeling (SDM) approach. System dynamics is defined as "the application of feedback control systems principles, and techniques to modeling, analyzing, and understanding the dynamic behavior of complex systems [3]."

The premise of SDM is that the dynamic behavior of a system is a consequence of its structure. SDM models the behavior of a system based on *cause-effect* or *influence* relationships between entities observable in a real system. These relationships constantly manifest while the model is being executed; thus the dynamics of the system are being modeled. One of the most powerful features of SDM is realized when multiple influence relationships are connected forming a circular relationship known as a feedback loop—a concept that reveals that any actor in a system will eventually be affected by its own action. The tool support existing for this modeling approach allows the computer model to be executed, thus simulating a real project [4].

Construction of the simulation model involved representing each of the software development and testing process activities using continuous modeling in simulation environment, along with process metrics for each activity corresponding to productivity and quality. For example, the test cases execution activity from Figure 2 would be represented with metrics corresponding to execution time and effectiveness in terms of identifying failures.

To examine the effect of defect avoidance practices on reliability, we integrated SDM with existing reliability prediction and growth models. To model the evolution of the failure rate and the number of remaining faults during system testing, we used an equation from an existing Poisson-type exponential-class model [5].

FailureRate = InitialFailureRate x
(1- Failures / TotalFailures)

In the context of system dynamics simulation, which is continuous simulation (as opposed to discrete event simulation), *rate* means the change in value for a variable (number of failures in this case), from time t to time t+1. The interval [t, t+1] is the simulation time interval. In a continuous simulation, time increases with constant increments. InitialFailureRate is the failure rate at the beginning of system testing. Failures is the number of failures encountered from the start of system testing to time t. TotalFailures is the total number of failures expected.

To account for the influence of specific test process factors on failures' dynamics, we changed the above formula to:

FailureRate = InitialFailureRate x
(1- Failures / TotalFailures) x
Fct(TestEffect) x Fct(NrTestCasesEx)

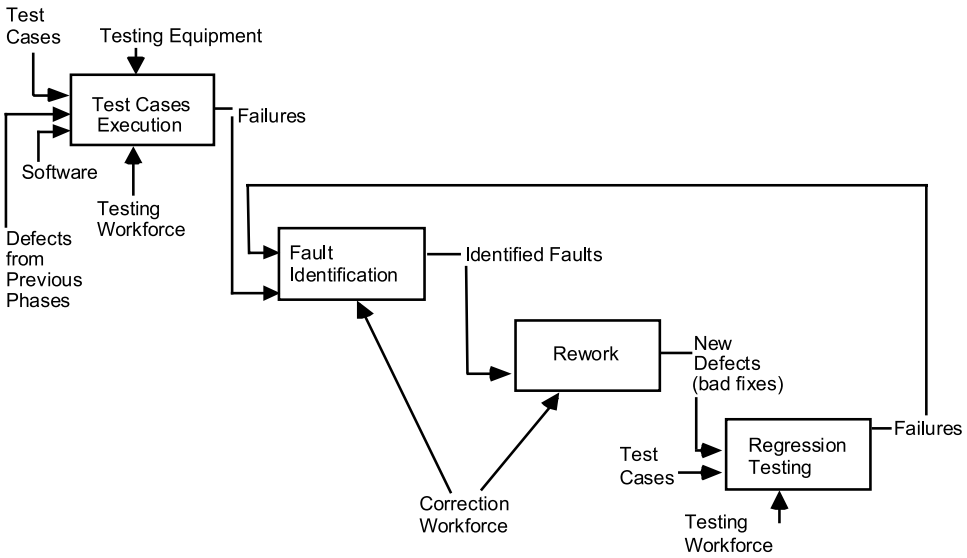
where Fct(TestEffect) is a function of the effectiveness (capability to uncover defects) of the test cases, and Fct(NrTestCasesEx) is a function of the number of test cases already executed. For our model, we assumed that failure intensity decreases with the increase of the number of test cases executed (in time, fewer and fewer failures are detected, and the failure rate decreases until its variation becomes extremely slow). The model's users should define these two functions according to historical data collected for their own projects. This is a part of the model calibration for a specific application, company, or type of projects.

For the InitialFailureRate model, users can input the value, if they know it, or an existing prediction model can be used. We used the Rome Laboratory model [6] that predicts the InitialFailureRate as a function of the remaining defect density (defects per size units, e.g., LOC) at the beginning of system testing.

InitialFailureRate = C x
RemainingDefectsDensity

Here C is a constant called transformation ratio, that depends on the application type, and was empirically determined as presented in Table 1.

Figure 2. *Activity based description of the system testing phase*



Application Type	Transformation Ratio
Airborne	6.2
Strategic	1.2
Tactical	13.8
Process control	3.8
Production Center	23
Developmental	Not available
Average	10.6

Table 1. *Fault density to failure rate transformation[6]*

Simulator users can replace the formula and table above with any other model that best describes their project.

Example: Simulator Use

By using the modeling environment [4], we implemented a computer process model that can be executed, simulating the behavior of the real process. To use the simulator, an organization must first customize the model (ensuring that the appropriate processes are represented) and populate it with their own metrics. Our current model follows a generic waterfall life cycle and is populated with default metrics based on industry averages. Once a model has been customized to an organization, specific project data can be entered such as project size, work force, target delivery date, etc.

To exemplify using the simulator, this section presents the execution of what-if scenarios for predicting how varying defect detection and correction activities could impact upon defects dynamics, reliability, effort, and schedule. We present the results of simulating two hypothetical projects that will be called Case1 and Case2 created by using industry benchmark data from [7]. The common characteristics of both projects are shown in Table 2 and serve as inputs to the simulation run.

The values for the other simulator inputs, such as effort and resource allocation, productivity, verification and validation (V&V), and correction rates, and efficiency of V&V activities, are found in [2].

V&V and correction activities are performed in requirements, design (reviews or inspections), and coding (reviews or inspections, and unit testing) for both cases. The difference between Case2 and Case1 is that the effort initially allocated in the design phase for V&V activities and rework is reduced to 20 percent in Case2 from that used in the Case1. The remaining 80 percent of the design V&V and rework effort is instead allocated to the system testing phase for Case2. This simulation is performed to investigate whether allocating more effort in defect detection and correction earlier in the life cycle improves the reliability of the final product and ultimately saves effort and time.

Size of final delivered software	130 KLOC
Programming language	C
Requirements size	1,000 function points
Estimated completion time	3 years
Work force size	5-18 people over phases
Required defect density in final product	at most 1.3 defects / KLOC

Table 2. *Simulation inputs – Common characteristics of Case1 and Case2*

The outputs of the simulation are presented in Table 3. There is a dynamic evolution of defects and failures for Case1 and Case2. Defects generated in different phases (requirement defects, design defects, and coding defects) evolve in time throughout the development life cycle; their number increases or decreases as they are introduced, detected, corrected, and reintroduced by bad fixes. Due to the fact that V&V activities and rework are performed in all phases, a significant number of defects are discovered and corrected before system testing begins. The number of defects detected for Case2 in the design phase is smaller than the same number for Case1 (as expected). Hence, the number of coding defects for Case2 is also higher than for Case1. That is because design defects, if not detected in design, will generate coding defects. Therefore, the total number of defects in the product at the beginning of system testing is around 330 for Case1 and 1,100 for Case2.

These cases present the variation of failures encountered and estimated to remain in the product during the system-testing phase. Although more effort was allocated to system testing for Case2 and the number of failures encountered by the end of system testing (project completion) is higher for Case2 (841) than for Case1 (270), the number of estimated remaining failures is 598 for Case2, much higher than for Case1 (197).

These results show that for the project modeled here, allocating more effort to early defect detection activities will improve reliability. The results of simulating both Case1 and Case2 are summarized in Table 3.

Table 3. *Simulation Outputs: Effort, Time, and Quality Data*

Project Parameter	Case1	Case2
Effort (staff hours)		
Coding	2,670	2,470
V&V in Design	265	175
Coding	830	870
System testing	21,710	23,416
Total V&V effort per project	*22,880*	*24,536*
Rework in Design	2,300	1,000
Coding	4,905	4,962
System testing	3,820	8,495
Total rework effort per project	*11,370*	*14,802*
Total effort per project	**41,035**	**45,923**
Time to complete (days)		
Design	69	57
Coding	113	109
System testing	378	451
Total project duration	***593***	***650***
Remaining defects in the product		
Design defects	22	92
Coding defects	151	430
Total defects delivered	173	522
Defect density in delivered product	1.3	4
Reliability		
Failures encountered	270	841
Estimated failures remaining	197	598

Since the requirements phase is identical for the two projects, only the data that are different for the two cases, corresponding to the design, coding, and testing phases are presented.

From the simulation output data in Table 3, the following observations can be made:

- Although the overall effort initially estimated and allocated is the same for the two projects, due to a different effort distribution to defect detection and correction activities over phases, the overall consumed effort is about 4,900 staff hours greater for Case2 than for Case1.
- Case2 is completed 57 days later than Case1.

These results confirm, for the simulated projects, the hypothesis that allocating more effort earlier in the development will eventually save effort and time.

An unexpected result of the simulation, observable in Table 3, is that for Case2 the actual effort to produce code is smaller than for Case1, although the same value would be expected. This is an example of what is called model's surprise behavior. Revealing these results and getting people to think about the causes of this behavior and to understand the complex relationships of a development project is one of the benefits of a system dynamics simulator.

In this case, there may be several explanations for the reduced coding time. One possibility might be that since V&V and rework during design take longer than planned, in the coding phase people would work under pressure and complete the implementation faster. This might also explain why the number of code defects introduced is higher for Case2 than for Case1, since schedule pressure often could reduce coding quality.

Although we can obtain numerical values (as shown in Table 3) as the output of the simulation, system dynamics models are best if used not for point predictions, but for analyzing trends of behavior, as reflected in the graphs that may be seen on the Web version of this paper [available at www.stsc.hill.af.mil].

Conclusions

We presented a process model and simulator valuable for:

- Increasing *understanding* and *communication* about the software development process structure, relations, and behavior; the influence of the software reliability engineering practices on reliability and process parameters; and more insights into the trade-off between reliability on one side and cost and time on the other side.
- Providing an experimentation tool that helps in decision making to improve *planning* and *tracking* the development process.
- Enabling *behavior estimation* prediction, i.e., trends in the dynamic evolution of a set of project parameters such as defect and reliability prediction.

The model was developed for a project that follows a waterfall life cycle process (but can be adapted to a different life cycle) and is calibrated for a hypothetical project. To utilize the model in a real project, it has to be calibrated to that specific environment. Calibration has several levels, from modifying numerical values of some variables, to changing relationships and equations, or even the structure of the model. The more the model is used and tuned to an organization and application domain, using their own historical data, the more accurate and valuable the model and the simulation will be. The model is available to organizations at no charge. Make inquiries to Dr. Ioana Rus.◆

References

1. Rus, I. and Collofello J.S., A Decision Support System for Software Reliability Engineering Strategy Selection, *Proceedings of COMPSAC*, Scottsdale, Ariz., October 1999.
2. Rus, I., Modeling the Impact on Cost and Schedule of Software Quality Engineering Practices, Ph.D. Dissertation, Arizona State University, 1998.
3. Abdel-Hamid, T. and Madnick, S.E., *Software Project Dynamics An Integrated Approach*, Prentice-Hall, Englewood Cliffs, New Jersey, 1991.
4. *Extend – Performance modeling for decision support, User's Manual*, Imagine That Inc., 1995.
5. Musa, J.D., Iannino, A. and Okumoto, K. *Software Reliability Measurement, Prediction, Application*, McGraw-Hill, 1987.
6. McCall, J., Randall, W., Bowen, C., McKelvey, N. Senn, R., Morris, J., Hecht, H., Fenwick, S. Yates, P. Hecht, M. and Vienneau, R. *Methodology for Software Reliability Prediction*, Rome Air Development Center Technical Report, RADC-TR-87-171, 1987, Vols. 1 and 2.
7. Jones, Capers, *Applied Software Measurement: Assuring Productivity and Quality*, McGraw-Hill, 1997.

Note

1. An *error* that is a result of a human action (misconception) results in a *defect* (a product anomaly), that will lead to a *fault* (a manifestation of a software error) that, if encountered, might cause a *failure* (termination of the ability of a functional unit to perform as required).

About the Authors

Ioana Rus is a scientist for Fraunhofer Center for Experimental Software Engineering, University of Maryland. She holds a doctorate in computer science and engineering from Arizona State University. Her research interests include software process improvement, modeling and simulation, measurement and experimentation in software engineering, and artificial intelligence.

Fraunhofer Center for Experimental Software Engineering
4321 Hartwick Rd., Suite 5000
College Park, Md. 20742-3290
Voice: 301-403-8971
Fax: 301-403-8976
E-mail: irus@fc-md.umd.edu
Internet: fc-md.umd.edu

James S. Collofello is a professor in the Department of Computer Science and Engineering at Arizona State University. He received his doctorate in computer science from Northwestern University. His teaching and research interests are in software quality assurance, software reliability, safety, and maintainability, software testing, software error analysis, and software project management.

Arizona State University, Dept. of Computer Science & Engineering
Box 875406, Tempe, Ariz. 85287
Voice: 480-965-3733
Fax: 480-965-2751
E-mail: collofello@asu.edu
Internet: www.eas.asu.edu:80/~csedept/people/faculty/collo.html