ELSEVIER

The Journal of Systems and Software 46 (1999) 173–182

The Journal of Systems and Software

Software process simulation for reliability management

Ioana Rus [a,1], James Collofello [a,2], Peter Lakey [b,*]

[a] Computer Science and Engineering Department, Arizona State University, AZ, USA
[b] Boeing Aircraft and Missiles, Mailcode S0343550, St. Louis, MO, 63166-0516, USA

Received 10 November 1998; accepted 11 November 1998

Abstract

This paper describes the use of a process simulator to support software project planning and management. The modeling approach here focuses on software reliability, but is just as applicable to other software quality factors, as well as to cost and schedule factors. The process simulator was developed as a part of a decision support system for assisting project managers in planning or tailoring the software development process, in a quality driven manner. The original simulator was developed using the *system dynamics* approach. As the model evolved by applying it to a real software development project, a need arose to incorporate the concepts of *discrete event modeling*. The system dynamics model and discrete event models each have unique characteristics that make them more applicable in specific situations. The continuous model can be used for project planning and for predicting the effect of management and reliability engineering decisions. It can also be used as a training tool for project managers. The discrete event implementation is more detailed and therefore more applicable to project tracking and control. In this paper the structure of the system dynamics model is presented. The use of the discrete event model to construct a software reliability prediction model for an army project, the Crusader, is described in detail.

1. Overview

The concept of process modeling and simulation was first applied to the software development process by Abdel-Hamid (Abdel-Hamid et al., 1991). Others (Madachy, 1996; Tvedt, 1996) have produced models as well, but there is little evidence that they have been successfully applied to real software development projects. The models described in this paper bring two new contributions to the software process modeling and simulation work. First, the modeling approach emphasizes software reliability, as opposed to general characteristics of the software development process. Second, the discrete event implementation of the model is being applied to a specific software project where useful results are expected.

2. Introduction

As the role of software is expanding rapidly in many aspects of modern life, quality and customer satisfaction become the main goal for software developers and an important marketing consideration for organizations. However, quality by itself is not a strategy that will ensure a competitive advantage. There are other project drivers like budget and delivery time that must be considered in relation to quality. Achieving the optimal balance among these three factors is a real challenge for any project manager (Boehm, 1996). The approach presented in this paper is intended to help managers and software process engineers to achieve this balance.

Software reliability engineering consists of a general set of engineering practices applied to the following tasks: defining the reliability objective of a software system, supporting the development of a system that achieves this objective and assessing the reliability of the product through testing and analysis. A detailed list of reliability engineering activities and methods, together with their purpose and description can be found in (Lakey and Neufelder, 1996). A software reliability program consists of a subset of the above practices and is defined for each project by combining different reliability achievement and assessment activities and methods, according to the project's characteristics. A number of methods are available to facilitate software reliability planning and control.

Use of cost estimation models and their corresponding software tools, like COCOMO (Boehm, 1984),

* Corresponding author. E-mail: peter.b.lakey@boeing.com
[1] E-mail: ioana.rus@asu.edu
[2] Email: collofello@asu.edu

0164-1212/99/$ – see front matter
PII: S0164-1212(99)00010-2

SLIM (Putnam, 1992) and Checkpoint (Jones, 1986), to estimate the impact of a practice on time and cost has the following disadvantages: because model tuning is needed not all relevant historical data might be available; cost drivers are at too coarse a level of granularity to reflect the specific technique whose impact needs to be evaluated. A major drawback of the models is their use of data from projects other than those to which they are being applied.

An alternative method to support strategy selection is process modeling and simulation, which involves analyzing the organizational software development process, creating a model of the process, and executing the model and simulating the real process. Although modeling and simulation has some limitations as well (the model accuracy depends on the quality of the model and of the calibration), there are many strengths of modeling that would advocate its use instead of other approaches mentioned. Modeling captures expert knowledge about the process of a specific organization; it does not need a real system to experiment with, so it does not affect the execution of real process; finally, dynamic modeling increases the understanding of a real process.

3. Software process modeling and simulation

Developing a model of the software development process involves the identification of entities, factors, variables, interactions and operations that are present in that process and are relevant to the intended use of the model. The *entities* include mainly people (developers, managers, customers, etc.), but can also include facilities, computer equipment, software tools, documentation and work instructions. *Factors* relevant to a general software development process include the application domain, the size of the project, the expected schedule and delivery date, the hardware platform, and other considerations. *Variables* include the number of software engineers, the skill levels of those individuals, the level of process maturity of an organization, the level of communication overhead, etc.

The modeling complexity increases when identifying *operations* and *interactions*. Operations are the tasks that need to be performed in the software process to transform user needs and requirements into executable software code. The typical operations include requirements analysis, architecture development, detailed design, implementation (code and unit test), integration, and system test. Each of these takes some input (entity) and generates some transformed output (another entity). For example, the design operation uses software requirements as the input and transforms these into a design of the software system.

The most important, and perhaps most difficult, modeling task is identifying and correctly representing the interactions among the factors and variables in the software process that are relevant to the modeling goal. For instance, how does the number of developers affect communication overhead and overall productivity? How does a defect prevention technique affect the number of defects injected in software documents or code? How does the effort allocated to regression testing affect failure intensity during system testing and the number of defects remaining at delivery time. These interactions need to be properly modeled in order to closely represent reality.

While the model is useful for representing the structure of the process, execution of the model is very helpful in understanding the behavior of the process. It gives management a tool that has been lacking for a long time. The simulation capability allows a manager to make a decision and have a high degree of confidence in what the results of that decision will be. Without the simulation the decision is purely speculation. The complexity of a decision is too overwhelming to be understandable without the dynamic execution of the software model.

4. Comparison to other approaches

Simulator presented here can be used for tracking the quality and reliability of the software throughout the development process. Reliability prediction and estimation models such as reliability growth models and prediction models have a similar goal. The difference is that the reliability growth models are analytical, address only the system testing phase, and each has unrealistic assumptions that restrict their applicability. The modeling and simulation approach addresses all the development phases and is tailored to the real process that is modeled. Reliability prediction models, as that developed for Rome Laboratory (SAIC, 1987) are static.

It is postulated here that running simulations to determine reasonable predicted defect levels at delivery in consideration of cost, schedule and staffing is a better way of predicting than using a static model. This approach allows the metrics tracked during the project to be used to make adjustments to the model to reflect what is really happening. That is, process behavior is explained by the model. With a static model, there is no way to know why deviations from predictions exist. The Rome Laboratory model is a regression model. All of these regression models inherently infer a cause–effect relationship between the independent variables and reliability, when in fact there is no proven causal relationship; rather the entire process is the cause of software defects and failures.

Tausworthe and Lyu (1996) developed a simulator of the software reliability process, but it is at a much higher

level of process abstraction and captures a very reduced number of process parameters.

Other system dynamics simulators of the software development process have been developed, but their purpose and scope is different. Most of the previous models, such as Abdel-Hamids model (Abdel-Hamid et al., 1991), focus on the management aspect of software development as opposed to the technical aspect; they also do not include the requirements analysis phase. Madachy's model (Madachy, 1996) was developed to analyze the effect of inspections, and Tvedt developed a model of the incremental life cycle (Tvedt, 1996) for analyzing cycle time reduction. The model presented in the next section has been developed to capture the technical issues of software quality and reliability and to assess the effect of various reliability practices.

5. System dynamics process model description

The following model was developed to support project planning for the purpose of evaluating software reliability engineering strategies. It is a generic model intended for use in high level decision making. The model represents the initial work performed by the authors in software process modeling and simulation and is part of a decision support system for software reliability engineering strategy selection and assessment (Rus, 1998).

The main components of the model correspond to the production phases of the software development process (requirements analysis, design and coding) and the system testing phase. A brief description of a production phase is presented here. More model details, as well as a description of the *SystemTesting* block, together with more simulation results and model use examples can be found in (Rus, 1998). The model was implemented in Extend V4.0, a simulation software package from Imagine That (Extend, 1995), by using hierarchical blocks. Each *ProductionPhase* block has a *Development* block and a *Management* block, corresponding to technical and managerial activities, respectively. The *Development* block models items production, as well as quality assurance activities. Items can be requirements documents, design documents, and source code. The *Development* block has two sub-blocks: *Items* and *Defects* that are shown in Fig. 1. Sub-blocks of *Items* and *Defects* are expanded in Fig. 2, at the lowest Extend implementation level.

The *Items* sub-block models items production and verification and validation (V&V) activities. Items from the previous phase enter the current production phase (*ItemsIn*) and items corresponding to the current phase are produced (for instance if the current phase is design, requirements documents are the input and design documents are generated). The production rate depends on many factors, such as the productivity of the personnel (*ProdProductIn*), and schedule pressure (*SchPresPrIn*). There are other factors affecting this rate that are not shown in the figure: process factors like methods, techniques, tool support, and metrics

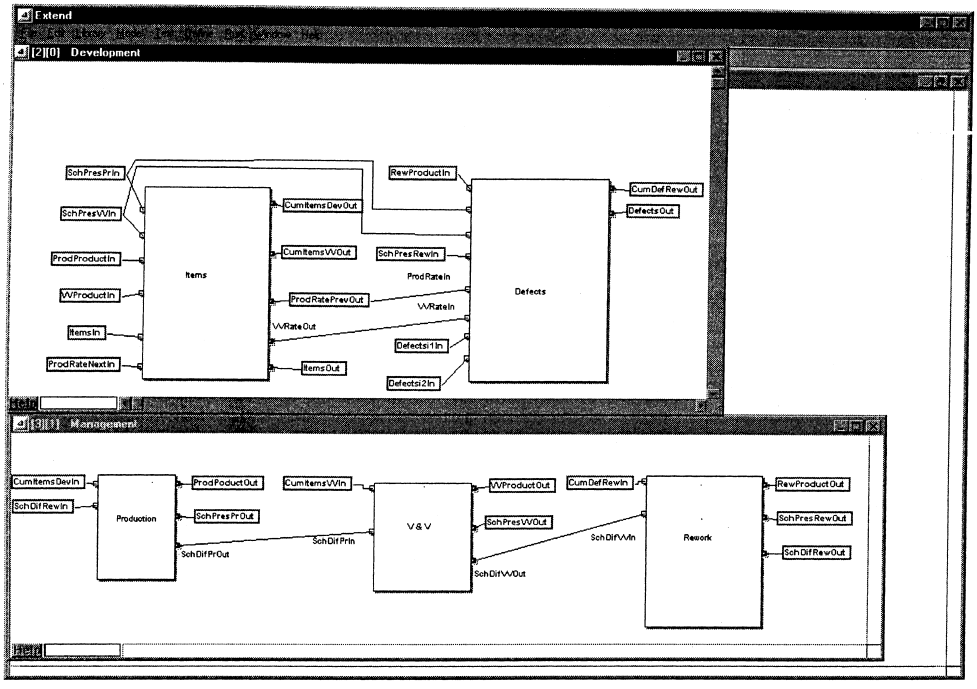

Fig. 1. *Development* and *Management* blocks of a *ProductionPhase*.

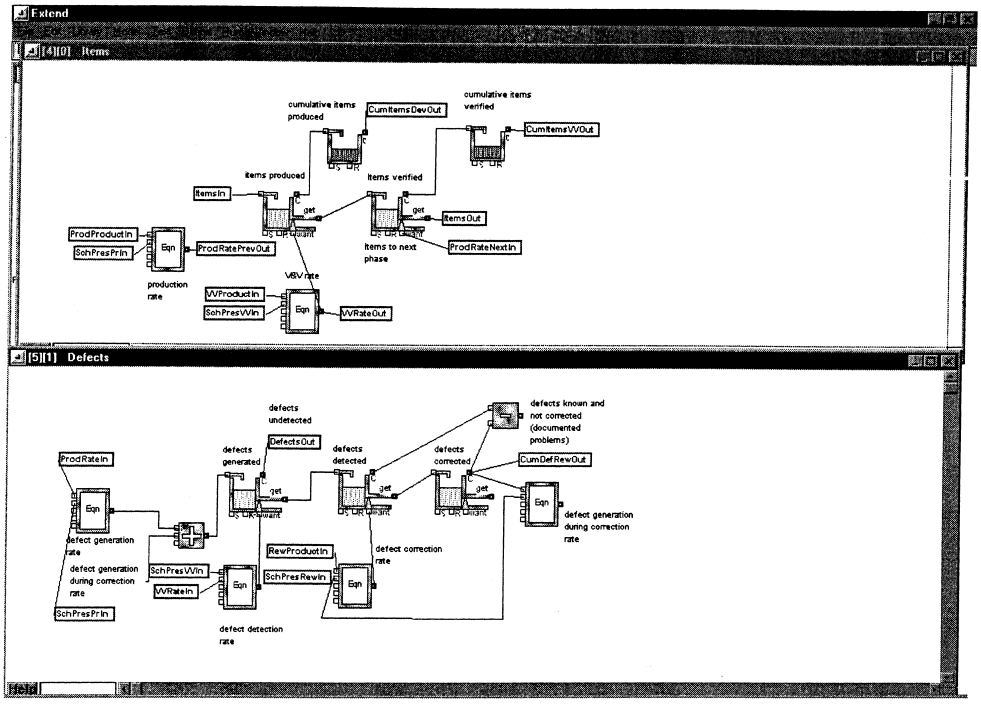

Fig. 2. *Items* and *Defects* blocks.

collection effort; and product factors like complexity, criticality and fault tolerance. The items produced are then verified (by reviews, walkthroughs, and/or inspections) at the *V&VRate*. This rate depends on factors like personnel productivity (*VVProductIn*) and schedule pressure (*SchPresVVIn*). After verification and validation, items are passed to the next phase (*ItemsOut*). For each item that is generated, there are also defects injected.

Defect generation, detection and correction is captured by the *Defects* sub-block. Defects are generated at a rate depending on the production rate, schedule pressure and other factors such as process maturity and development methodology. *Defectsi1In* and *Defectsi2In* are defects propagated from previous phases. The defect detection rate depends on the verification and validation rate (*VVRateIn*), efficiency of V&V activities, schedule pressure, defect density, and other factors. Detected defects will be corrected, and possibly new defects will be generated. Undetected defects propagate to subsequent phases (*DefectsOut*).

The *Management* block (Fig. 1) models the human resources, planning and control aspects of the process. For each of the production, verification and validation (V&V), and rework activities there is a corresponding sub-block in the *Management* block. Each of these three sub-blocks models the effort consumed (man-days) in the tasks of producing, verifying and reworking items. These modules also capture personnel experience increasing with time (learning) and schedule pressure resulting from the difference between the estimated schedule and the actual one. Actual productivity determines the rate at which items are produced.

Figs. 3 and 4 present examples of model execution (process simulation) outputs. Fig. 3(a) shows the variation of the number of items produced and verified during a production phase. The number of defects injected, detected, and corrected throughout the phase can be seen in Fig. 3(b). Fig. 4(a) shows the evolution of the number of coding faults identified and corrected during system testing, and the coding faults remaining in the product. Fig. 4(b) presents an example of sensitivity analysis: the variation of failures encountered during system testing, with the number of design faults at the beginning of system testing. These values and shapes of these graphs will be unique to a specific company that calibrates the model with metrics and inputs specific to that company.

Software reliability engineering practices are modeled by considering their influences on different factors that impact defects. For instance, defect prevention techniques (e.g. formal methods) will decrease the value of defect generation rate. Defect detection techniques will increase the number of defects detected and reduce the number of defects propagated to the next phase. Fault tolerance methods (e.g. N-version programming) reduce the impact that remaining defects have on the reliability of the operational system.

There is, of course, some additional up-front effort and cost associated with these practices, but the overall development time and effort may be reduced, by long-term reduction in rework time. By reducing the number

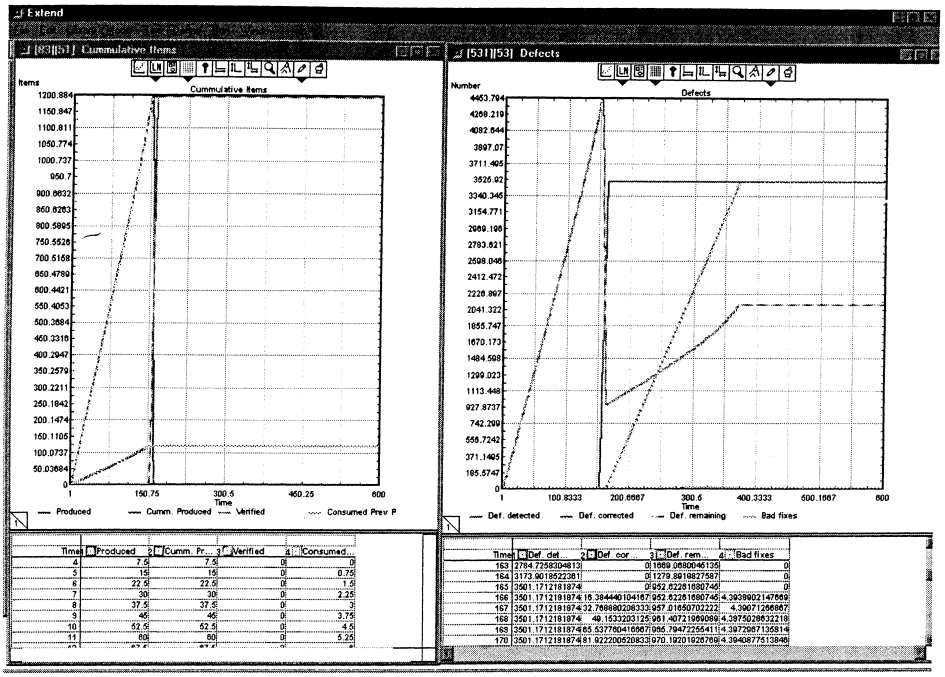

Fig. 3. Variation of the number of items (a) and defects (b) through a production phase.

of remaining defects, and/or masking them, the reliability of the product will increase. Simulation is used to analyze the effect of these practices in terms of failures, cost, and staffing, allowing trade-offs among reliability strategies.

6. Discussion on system dynamics model

The model described above was executed with assumed parameter values for each of the factors identified. By changing these factors individually the

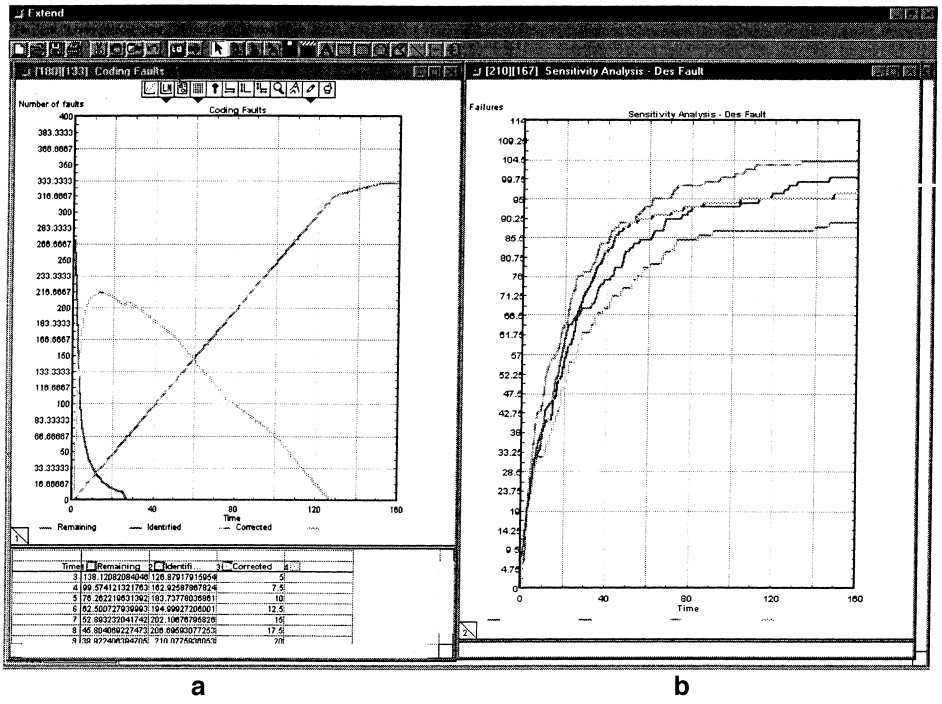

Fig. 4. (a) Variation of the number of coding faults during system testing. (b) Variation of failures encountered during system testing, with the number of design faults at the beginning of system testing.

corresponding impact on the outputs of the model (cost, schedule and quality) can be evaluated. The model is implemented with building blocks derived from the generic *ProductionPhase* block by adding details specific to each development phase while maintaining the common features and architecture. As a result, the model is modular, extensible, adaptable and flexible. The example that we used in (Rus, 1998) corresponds to a waterfall development life cycle model, but the model can be adapted to represent for example an incremental process.

The system dynamics approach, which is a continuos modeling paradigm, considers that items are identical and treats them uniformly. This is an assumption that works at a higher level of modeling. If the use and the goal of the model require more detail, then differences between entities and entities' attributes must be considered. For example, design and code items differ by structural properties such as size, complexity, and modularity; usage, and effort allocated for development. Similarly, defects can have different types, consequences, generation and detection phase, etc. In Extend, when using discrete event modeling, items can have attributes attached, with random values (within a specified range), and with a specified distribution. Activity duration and item generation and processing times can vary with other model parameters (variables), according to a specified equation, and can have a probabilistic distribution around the computed value. Subprocesses such as detailed design and system testing (reliability growth testing) appear to be more like event-driven processes rather than continuous.

Therefore, because discrete event modeling allows a more detailed modeling capability which was required by the application described in the next section, the system dynamics model was transformed into an alternative discrete event model for a production phase, as illustrated by the preliminary design phase example.

7. Discrete event model application

The discrete event model implementation of the software process was developed to support a specific project, Crusader. The Crusader is a large army development project. There are two vehicles, called Segments, in each Crusader System. The system consists of the Self-propelled Howitzer (SPH) and the Re-supply Vehicle (RSV). The software is being developed using object-oriented analysis and design (OOA/D) and Rational Rose CASE tool. An incremental software life cycle approach is being used, with specific functionality in five planned major builds.

A top-level architecture has been constructed for the Crusader software system. It consists of 40 Computer Software Configuration Items (CSCIs). These could also be called subsystems. Each CSCI is developed and managed independently of the other CSCIs. The approach taken is to model one of these CSCIs, using it as a prototype for learning the relevant dynamics associated with defects, reliability and other process parameters.

Many of the important metrics associated with reliability that are captured in the discrete event model are shown in Fig. 5. Some of the most important metrics

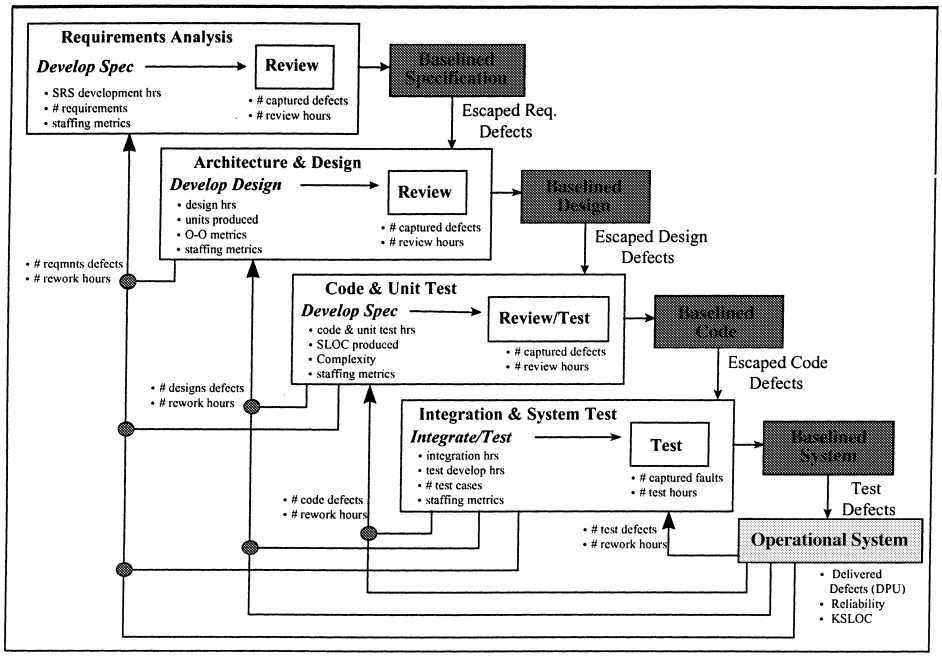

Fig. 5. Crusader software reliability management metrics.

needed to support software reliability management are related to *rework*. As it pertains to reliability, rework is important because it helps determine the amount of reliability improvement that can be expected for a given schedule and staffing level. Software development activities may be completed on time, but if no time is built in for rework, then the defects that are found during testing cannot be fixed and the reliability cannot improve significantly.

Rework is also important from the standpoint of process improvement. Reducing rework increases the total amount of quality software that can be developed per time unit. The model helps pinpoint areas where improvements should be made to reduce rework. To substantially reduce rework a significant investment needs to be made in process improvement during implementation of the software development schedule. This will increase costs in the short term, and may lengthen the schedule. These dynamics are captured in the model.

8. Prototype discrete event model description

The initial model being developed captures the process for *Preliminary Design*. The structure of the model, developed using Extend V4.0, is described below. The prototype discrete event model represents the Crusader Software Development Process, and it is being piloted on a single CSCI, which will be called *Module X* here. This discrete event model is actually a hybrid model that also incorporates system dynamics concepts, using feedback loops. The main purpose of the model is to predict, track, and control software defects and failures over the entire Crusader development, through the year 2005. It also facilitates tracking and control of software project costs and schedules. The model currently addresses only the *Preliminary Design* phase. Later on this year it will be expanded to include *Detailed Design*, *Code and Unit Test*, *Subsystem Test*, *Element Test* and *Segment Test*. There will be two modules. The *Defect Module* will predict defects, effort, and rework for each phase through *Code and Unit Test*. The *Failure Module* will estimate reliability and test effort during testing.

There are three types of inputs to *Preliminary Design*: *Segment Use Cases/Scenarios, Software Requirements Specifications*, and the architecture in the form of *Rose Models*. The Crusader project has defined 10 Use Cases, 40 CSCIs and approximately 40 Rose Subsystems to correspond to the 40 CSCIs. Module X is responsible for approx. 50 Segment Scenarios, has 200 CSCI Scenarios of its own, and will use 200 classes in its design. In a pure discrete event model the activity of *Refining the Scenarios* would be represented by a single *Operation* block, with a single set of outputs, including schedule and quality. The problem with that approach (only one output data point) is that no dynamics can be incorporated into the activity during the process. In order to incorporate feedback the *Refine Scenarios Activity* has been divided into five discrete steps, allowing the outputs of one step to be fed into the next step. The inputs each are divided up into five sets to be processed during each iteration. In other words, 10 Segment Scenarios, 40 CSCI Scenarios and 40 Rose Classes are processed for each step. This is different from actual process implementation, but the approximation should be acceptable for modeling.

Random data is generated for each of the input products to the *Preliminary Design* activity. This data is stored in a spreadsheet file. All of this stored data is accessed throughout the simulation. The values for *Segment Scenario* size are normally distributed around the average value of 10. The values for *Scenario Quality* refers to the number of defects associated with a particular scenario prior to it being used to define CSCI scenarios. Class size can be characterized through a number of different object-oriented design metrics, such as number of instance methods in a class, number of instance variables in a class or number of class methods.

The screen capture in Fig. 6 shows the three types of input products, *Segment Scenarios*, *CSCI Scenarios* and *Classes*, being read into an *Operation* block where that data is processed and resources are expended to perform the *Refine CSCI Scenarios* task. The data from the spreadsheet is used to calculate some project parameters during the simulation. Fig. 6 represents three main steps for the activity of *Refining CSCI Scenarios*. First, the scenarios are developed using the *Segment Scenarios* allocated to MCS. Then, a *Walkthrough* is performed to evaluate the completeness and correctness of the *CSCI Scenarios* and their mapping to *Segment Scenarios*. Finally, any defects found are documented and those are corrected through the *Rework Activity*.

During each of these activities data is generated for *Effort* – in terms of labor hours expended, and *Quality* – in terms of numbers of defects. All of this calculated data is passed on to a final block in the model, which plots this data graphically over the course of a run. The list of outputs from the model includes development effort, review effort, rework effort, defects generated, defects found and escaped defects. These parameters are each dependent on a number of factors that interact with each other over the course of a software development project. The basis of the model is the set of factors for each parameter and the equations used to calculate the factor values during execution of the model. These factors include both process and product factors. Some of the process factors included in the model are the *Manpower Factor*, *Schedule Factor*, *Communication Overhead Factor*, *Work Rate Factor*, *Process Maturity Factor* and *Tool Support Factor*. The product factors include *Size*, *Quality*, and *Complexity*.

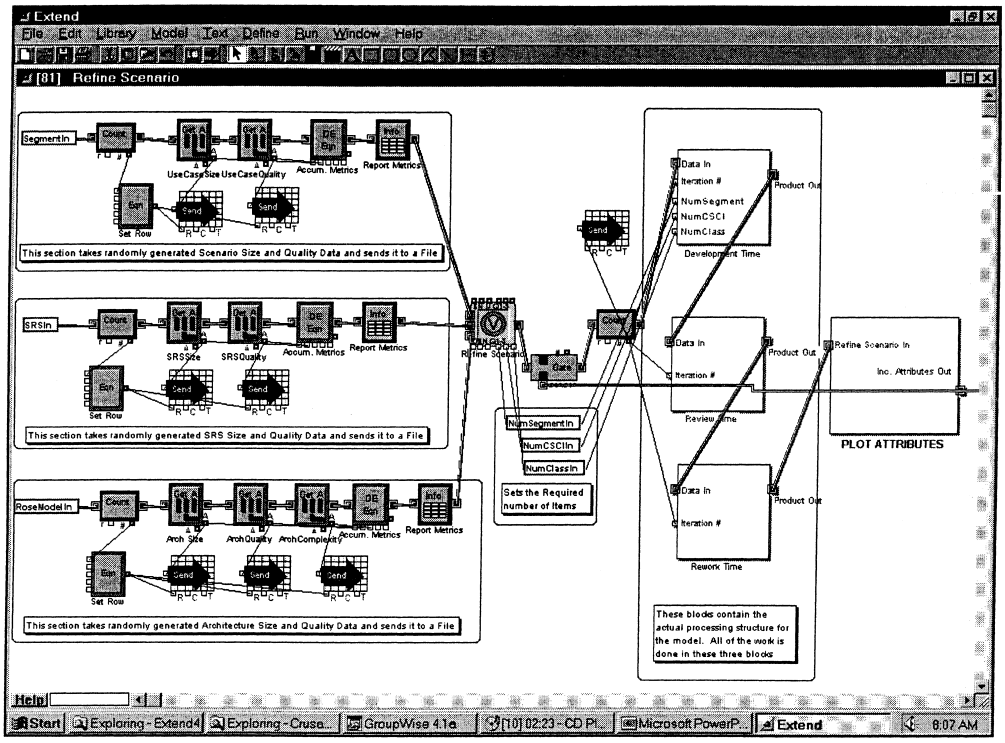

Fig. 6. Refine scenarios process for Modules X.

The model factors are dynamic. That is, they change after each event (iteration) occurs in a simulation. Many of the factors affect other factors, which in turn are affected by the same factors. This provides feedback loops. The interaction between factors in the model is intended to represent actual Crusader software development project dynamics. As work is scheduled and accomplished, the project team will often find that the effort required for a task is larger than originally anticipated. This occurs in the middle of a development activity. When the project gets behind schedule decisions are made that change the expected state. Work is sped up or postponed. Certain activities may be cancelled, such as periodic walkthroughs. These decisions impact overall effort and the number of defects injected and found. Quality can suffer if the schedule is emphasized.

The values used for model factors currently are assumptions – no historical project data is available from Crusader to represent the relationships among the factors. These assumptions will be replaced with factors calculated from data collected on the Crusader project as development progresses and the relationships are more clearly understood. The intent is to have a working model by the end of 2000 useful for management decisions.

The screen capture in Fig. 7 shows the output for a single simulation run. The *Actual* (simulated) values are different from the *Expected* values for each set data being plotted. There are always be differences as the model is currently set up because the *Size* and *Quality* data is randomized for each iteration, making it different from the expected size and quality data. This causes the *Size* and *Quality Factors* to be greater than or less than 1. Depending on amplitude and direction of the differences as compared to expected, the actual values would be above or below the expected values for each of the plots above.

This model uses a Monte Carlo simulation, which means there are random draws for each execution, making the results different. With all inputs being the same, in a single execution of the Module X Preliminary Design process, the outcome could be as shown in Fig. 7 or some other outcome.

9. Conclusions and future work

A dynamic model of the software development process allows answer questions such as: "How much time and effort will it take to complete the Preliminary Design phase for Module X?", "How many defects are going to be generated during that phase? How will that affect product reliability?". "How much will the effort increase if defect prevention practices are used? How will that shorten system testing time?". These questions are very difficult to answer using methods currently available in industry. There is a great deal of variability in the possible outcomes using static models. The modeling

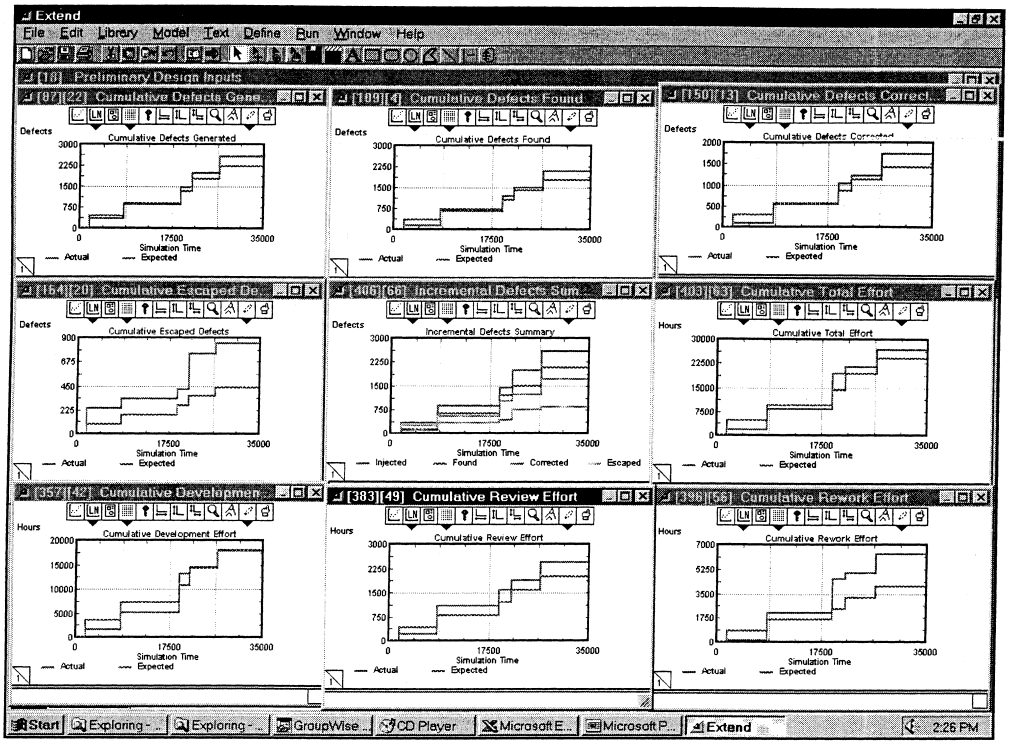

Fig. 7. Output screen for Module X Discrete Event Model.

methods described in this paper provide insight into why and how things occur.

Crusader management believes there is a great potential benefit for using the discrete event modeling approach not only for reliability, but for many other aspects of project management, including cost and schedule.

With good data this model can be continuously updated and improved throughout the current program phase, which concludes at the end of 2000. Prior to the next phase, Engineering and Manufacturing Development (EMD), the model should be validated to the point where it supports effective management decisions. Outputs predicted by simulation will be compared with actual data collected from the project. Discrepancies will be identified and used to improve the validity and accuracy of the model. With a real-world model, management can make good decisions throughout the EMD phase of the program to meet software reliability requirements.

References

Abdel-Hamid, T., Madnick, S.E., 1991. Software Project Dynamics An Integrated Approach. Prentice-Hall, Englewood Cliffs, NJ.

Boehm, B.W., 1984. Software engineering economics. IEEE Transactions on Software Engineering 10 (1), 5–21.

Boehm, B.W., Hoh, 1996. Identifying quality-requirements conflicts. IEEE Software 25–35.

Jones, C., 1986. The SPR feature point method. Software Productivity Research.

Extend – Performance modeling for decision support, 1995. User's Manual, Imagine That.

Lakey, P., Neufelder, A.M., 1996. System and Software Reliability Assurance Notebook, Produced for Rome Laboratory by Soft-Rel. Contact peter.b.lakey@boeing.com for a copy of the Notebook.

Madachy, R., 1996. System dynamics modeling of an inspection-based process. Proceedings of the Eighteenth International Conference on Software Engineering, Berlin, Germany.

Putnam, L.H., 1992. Measures for Excellence – Reliable Software on Time, Within Budget, Yourdon Press Computing Series.

Rus, I., 1998. Modeling the impact on project cost and schedule of software reliability engineering strategies. Ph.D. Dissertation, Arizona State University, Tempe, Arizona.

SAIC, 1987. Methodology for software reliability prediction. Rome Laboratory RADC-TR-87-171.

Tausworthe, R., Lyu, M., 1996. Software reliability simulation. In: Lyu, M. (Eds.), Handbook of Software Reliability Engineering. IEEE Computer Society Press, McGraw-Hill, New York.

Tvedt, J.D., 1996. An extensible model for evaluating the impact of process improvements on software development cycle time. Ph.D. Dissertation, Arizona State University, Tempe, Arizona.

Ioana Rus is a Ph.D. candidate in the Department of Computer Science and Engineering at Arizona State University. She received her Master in Computer Science degree from Arizona State University and Bachelors of Science from Polytechnical Institute Cluj, Romania. Her research interests include software process improvement, process modeling, software quality and reliability, artificial intelligence, and neural networks. She is a member of the IEEE Computer Society and ACM.

James S. Collofello is a professor in the Department of Computer Science and Engineering at Arizona State University. He received his Doctor of Philosophy degree in Computer Science from Northwestern University, Master of Science in Mathematics and Computer Science from Northern Illinois University, and Bachelors of Science in

Mathematics and Computer Science from Northern Illinois 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. He is a member of the IEEE Computer Society and ACM.

Peter B. Lakey received his MS in Engineering Management from University of Missouri-Rolla, MBA from Washington University, and his BSEE from Northwestern University. He joined McDonnell Douglas (now Boeing) in 1990 as a Reliability Engineer. As an engineer in the Software Engineering Process Group (SEPG), Mr. Lakey has been responsible for developing tools and processes that can be used by internal customers to improve software product quality. His main responsibility now is for managing a Software Reliability Support subcontract with United Defense in Minneapolis on a large Army program called the Crusader.